Metamath

A Computer Language for Mathematical Proofs

Norman Megill

with extensive revisions by
David A. Wheeler

2019-06-02

~ Public Domain ~

This book (including its later revisions) has been released into the Public Domain by Norman Megill per the Creative Commons CC0 1.0 Universal (CC0 1.0) Public Domain Dedication (`https://creativecommons.org/publicdomain/zero/1.0/`). David A. Wheeler has done the same. The public domain release applies worldwide. In case this is not legally possible, the right is granted to use the work for any purpose, without any conditions, unless such conditions are required by law.

Several short, attributed quotations from copyrighted works appear in this book under the "fair use" provision of Section 107 of the United States Copyright Act (Title 17 of the *United States Code*). The public-domain status of this book is not applicable to those quotations.

Any trademarks used in this book are the property of their owners.

ISBN: 978-0-359-70223-7

Lulu Press
Morrisville, North Carolina
USA

Norman Megill
93 Bridge St., Lexington, MA 02421
E-mail address: `nm@alum.mit.edu`

David A. Wheeler
E-mail address: `dwheeler@dwheeler.com`

`http://metamath.org`

Contents

Preface ix

1 Introduction **1**
1.1 Mathematics as a Computer Language 4
1.1.1 Is Mathematics "User-Friendly"? 4
1.1.2 Mathematics and the Non-Specialist 12
1.1.3 An Impossible Dream? 14
1.1.4 Beauty 15
1.1.5 Simplicity 16
1.1.6 Rigor 18
1.2 Computers and Mathematicians 20
1.2.1 Trusting the Computer 21
1.2.2 Trusting the Mathematician 22
1.3 The Use of Computers in Mathematics 24
1.3.1 Computer Algebra Systems 24
1.3.2 Automated Theorem Provers 25
1.3.3 Interactive Theorem Provers 27
1.3.4 Proof Verifiers 28
1.3.5 Creating a Database of Formalized Mathematics . . . 29
1.3.6 In Summary 31
1.4 Mathematics and Metamath 31
1.4.1 Standard Mathematics 31
1.4.2 Other Formal Systems 32
1.4.3 Metamath and Its Philosophy 33
1.4.4 A History of the Approach Behind Metamath 33
1.4.5 Metamath and First-Order Logic 34

2 Using the Metamath Program **37**
2.1 Installation 37
2.2 Your First Formal System 38
2.2.1 From Nothing to Zero 38
2.2.2 Converting It to Metamath 40
2.3 A Trial Run 44
2.3.1 Some Hints for Using the Command Line Interface . . 49

2.4 Your First Proof 50
2.5 A Note About Editing a Database File 57

3 Abstract Mathematics Revealed 59
3.1 Logic and Set Theory 59
3.2 The Axioms for All of Mathematics 62
3.2.1 Propositional Calculus 62
3.2.2 Predicate Calculus 64
3.2.3 Set Theory 68
3.2.4 Other Axioms 70
3.3 The Axioms in the Metamath Language 70
3.3.1 Propositional Calculus 70
3.3.2 Axioms of Predicate Calculus with Equality—Tarski's S2 71
3.3.3 Axioms of Predicate Calculus with Equality—Auxiliary 71
3.3.4 Set Theory 71
3.3.5 That's It 72
3.4 A Hierarchy of Definitions 72
3.4.1 Definitions for Propositional Calculus 74
3.4.2 Definitions for Predicate Calculus 76
3.4.3 Definitions for Set Theory 77
3.5 Tricks of the Trade 85
3.6 A Theorem Sampler 86
3.7 Axioms for Real and Complex Numbers 89
3.7.1 The Axioms for Real and Complex Numbers Themselves 90
3.7.2 Complex Number Axioms in Analysis Texts 91
3.7.3 Eliminating Unnecessary Complex Number Axioms . . 92
3.8 Two Plus Two Equals Four 92
3.9 Deduction 94
3.9.1 The Standard Deduction Theorem 94
3.9.2 Weak Deduction Theorem 95
3.9.3 Deduction Style 97
3.9.4 Natural Deduction 98
3.9.5 Strengths of Our Approach 101
3.10 Exploring the Set Theory Database 102
3.10.1 A Note on the "Compact" Proof Format 109

4 The Metamath Language 111
4.1 Specification of the Metamath Language 112
4.1.1 Preliminaries 112
4.1.2 Preprocessing 113
4.1.3 Basic Syntax 113
4.1.4 Proof Verification 115
4.2 The Basic Keywords 116
4.2.1 User-Defined Tokens 117
4.2.2 Constants and Variables 119

4.2.3 The $c and $v Declaration Statements . . . 119
4.2.4 The $d Statement . . . 120
4.2.5 The $f and $e Statements . . . 125
4.2.6 Assertions ($a and $p Statements) . . . 127
4.2.7 Frames . . . 129
4.2.8 Scoping Statements ($\{ and $\}) . . . 132
4.3 The Anatomy of a Proof . . . 135
4.3.1 The Concept of Unification . . . 139
4.4 Extensions to the Metamath Language . . . 139
4.4.1 Comments in the Metamath Language . . . 139
4.4.2 The Typesetting Comment ($t) . . . 144
4.4.3 Additional Information Comment ($j) . . . 147
4.4.4 Including Other Files in a Metamath Source File . . . 148
4.4.5 Compressed Proof Format . . . 149
4.4.6 Specifying Unknown Proofs or Subproofs . . . 150
4.5 Axioms vs. Definitions . . . 150
4.5.1 What is a Definition? . . . 151
4.5.2 The Approach to Definitions in set.mm . . . 152
4.5.3 Adding Constraints on Definitions . . . 154
4.5.4 Summary of Approach to Definitions . . . 155

5 The Metamath Program 157
5.1 Invoking Metamath . . . 157
5.2 Controlling Metamath . . . 158
5.2.1 exit Command . . . 159
5.2.2 open log Command . . . 159
5.2.3 close log Command . . . 160
5.2.4 submit Command . . . 160
5.2.5 erase Command . . . 160
5.2.6 set echo Command . . . 160
5.2.7 set scroll Command . . . 160
5.2.8 set width Command . . . 161
5.2.9 set height Command . . . 161
5.2.10 beep Command . . . 161
5.2.11 more Command . . . 161
5.2.12 Operating System Commands . . . 161
5.2.13 Size Limitations in Metamath . . . 162
5.3 Reading and Writing Files . . . 162
5.3.1 read Command . . . 162
5.3.2 write source Command . . . 163
5.4 Showing Status and Statements . . . 163
5.4.1 show settings Command . . . 163
5.4.2 show memory Command . . . 163
5.4.3 show labels Command . . . 163
5.4.4 show statement Command . . . 164

5.4.5 `search` Command 164
5.5 Displaying and Verifying Proofs 165
5.5.1 `show proof` Command 165
5.5.2 `show usage` Command 166
5.5.3 `show trace_back` Command 166
5.5.4 `verify proof` Command 166
5.5.5 `verify markup` Command 167
5.5.6 `save proof` Command 167
5.6 Creating Proofs 168
5.6.1 `prove` Command 170
5.6.2 `set unification_timeout` Command 170
5.6.3 `set empty_substitution` Command 171
5.6.4 `set search_limit` Command 171
5.6.5 `show new_proof` Command 171
5.6.6 `assign` Command 172
5.6.7 `match` Command 172
5.6.8 `let` Command 173
5.6.9 `unify` Command 173
5.6.10 `initialize` Command 174
5.6.11 `delete` Command 174
5.6.12 `improve` Command 175
5.6.13 `save new_proof` Command 175
5.7 Creating LaTeX Output 176
5.7.1 `open tex` Command 176
5.7.2 `close tex` Command 177
5.8 Creating HTML Output 177
5.8.1 `write theorem_list` Command 178
5.8.2 `write bibliography` Command 178
5.8.3 `write recent_additions` Command 178
5.9 Text File Utilities 179
5.9.1 `tools` Command 179
5.9.2 `help` Command (in `tools`) 179
5.9.3 Using `tools` to Build Metamath `submit` Scripts . . . 180
5.9.4 Example of a `tools` Session 180

A Sample Representations **183**

B Compressed Proofs **187**

C Metamath's Formal System **189**
C.1 Introduction 189
C.2 The Formal Description 190
C.2.1 Preliminaries 190
C.2.2 Constants, Variables, and Expressions 190
C.2.3 Substitution 191

C.2.4 Statements 191
C.2.5 Formal Systems 193
C.3 Examples of Formal Systems 194
C.3.1 Example 1—Propositional Calculus 194
C.3.2 Example 2—Predicate Calculus with Equality 196
C.3.3 Free Variables and Proper Substitution 198
C.3.4 Metalogical Completeness 199
C.3.5 Example 3—Metalogically Complete Predicate Calculus with Equality 199
C.3.6 Example 4—Adding Definitions 201
C.3.7 Example 5—ZFC Set Theory 202
C.3.8 Example 6—Class Notation in Set Theory 202
C.4 Metamath as a Formal System 204

D The MIU System **207**

E Metamath Language EBNF **211**

Bibliography **215**

Index **221**

Preface

Overview

Metamath is a computer language and an associated computer program for archiving, verifying, and studying mathematical proofs at a very detailed level. The Metamath language incorporates no mathematics per se but treats all mathematical statements as mere sequences of symbols. You provide Metamath with certain special sequences (axioms) that tell it what rules of inference are allowed. Metamath is not limited to any specific field of mathematics. The Metamath language is simple and robust, with an almost total absence of hard-wired syntax, and we[1] believe that it provides about the simplest possible framework that allows essentially all of mathematics to be expressed with absolute rigor.

Using the Metamath language, you can build formal or mathematical systems[2] that involve inferences from axioms. Although a database is provided that includes a recommended set of axioms for standard mathematics, if you wish you can supply your own symbols, syntax, axioms, rules, and definitions.

The name "Metamath" was chosen to suggest that the language provides a means for *describing* mathematics rather than *being* the mathematics itself. Actually in some sense any mathematical language is metamathematical. Symbols written on paper, or stored in a computer, are not mathematics itself but rather a way of expressing mathematics. For example "7" and "VII" are symbols for denoting the number seven in Arabic and Roman numerals; neither *is* the number seven.

If you are able to understand and write computer programs, you should be able to follow abstract mathematics with the aid of Metamath. Used in

[1] Unless otherwise noted, the words "I," "me," and "my" refer to Norman Megill, while "we," "us," and "our" refer to Norman Megill and David A. Wheeler.

[2] A formal or mathematical system consists of a collection of symbols (such as 2, 4, + and =), syntax rules that describe how symbols may be combined to form a legal expression (called a well-formed formula or *wff*, pronounced "whiff"), some starting wffs called axioms, and inference rules that describe how theorems may be derived (proved) from the axioms. A theorem is a mathematical fact such as $2 + 2 = 4$. Strictly speaking, even an obvious fact such as this must be proved from axioms to be formally acceptable to a mathematician.

conjunction with standard textbooks, Metamath can guide you step-by-step towards an understanding of abstract mathematics from a very rigorous viewpoint, even if you have no formal abstract mathematics background. By using a single, consistent notation to express proofs, once you grasp its basic concepts Metamath provides you with the ability to immediately follow and dissect proofs even in totally unfamiliar areas.

Of course, just being able follow a proof will not necessarily give you an intuitive familiarity with mathematics. Memorizing the rules of chess does not give you the ability to appreciate the game of a master, and knowing how the notes on a musical score map to piano keys does not give you the ability to hear in your head how it would sound. But each of these can be a first step.

Metamath allows you to explore proofs in the sense that you can see the theorem referenced at any step expanded in as much detail as you want, right down to the underlying axioms of logic and set theory (in the case of the set theory database provided). While Metamath will not replace the higher-level understanding that can only be acquired through exercises and hard work, being able to see how gaps in a proof are filled in can give you increased confidence that can speed up the learning process and save you time when you get stuck.

The Metamath language breaks down a mathematical proof into its tiniest possible parts. These can be pieced together, like interlocking pieces in a puzzle, only in a way that produces correct and absolutely rigorous mathematics.

The nature of Metamath enforces very precise mathematical thinking, similar to that involved in writing a computer program. A crucial difference, though, is that once a proof is verified (by the Metamath program) to be correct, it is definitely correct; it can never have a hidden "bug." After getting used to the kind of rigor and accuracy provided by Metamath, you might even be tempted to adopt the attitude that a proof should never be considered correct until it has been verified by a computer, just as you would not completely trust a manual calculation until you have verified it on a calculator.

My goal for Metamath was a system for describing and verifying mathematics that is completely universal yet conceptually as simple as possible. In approaching mathematics from an axiomatic, formal viewpoint, I wanted Metamath to be able to handle almost any mathematical system, not necessarily with ease, but at least in principle and hopefully in practice. I wanted it to verify proofs with absolute rigor, and for this reason Metamath is what might be thought of as a "compile-only" language rather than an algorithmic or Turing-machine language (Pascal, C, Prolog, Mathematica, etc.). In other words, a database written in the Metamath language doesn't "do" anything; it merely exhibits mathematical knowledge and permits this knowledge to be verified as being correct. A program in an algorithmic

language can potentially have hidden bugs as well as possibly being hard to understand. But each token in a Metamath database must be consistent with the database's earlier contents according to simple, fixed rules. If a database is verified to be correct,[3] then the mathematical content is correct if the verifier is correct and the axioms are correct. The verification program could be incorrect, but the verification algorithm is relatively simple (making it unlikely to be implemented incorrectly by the Metamath program), and there are over a dozen Metamath database verifiers written by different people in different programming languages (so these different verifiers can act as multiple reviewers of a database). The most-used Metamath database, the Metamath Proof Explorer (aka `set.mm`), is currently verified by four different Metamath verifiers written by four different people in four different languages, including the original Metamath program described in this book. The only "bugs" that can exist are in the statement of the axioms, for example if the axioms are inconsistent (a famous problem shown to be unsolvable by Gödel's incompleteness theorem). However, real mathematical systems have very few axioms, and these can be carefully studied. All of this provides extraordinarily high confidence that the verified database is in fact correct.

The Metamath program doesn't prove theorems automatically but is designed to verify proofs that you supply to it. The underlying Metamath language is completely general and has no built-in, preconceived notions about your formal system, its logic or its syntax. For constructing proofs, the Metamath program has a Proof Assistant which helps you fill in some of a proof step's details, shows you what choices you have at any step, and verifies the proof as you build it; but you are still expected to provide the proof.

There are many other programs that can process or generate information in the Metamath language, and more continue to be written. This is in part because the Metamath language itself is very simple and intentionally easy to automatically process. Some programs, such as `mmj2`, include a proof assistant that can automate some steps beyond what the Metamath program can do. Mario Carneiro has developed an algorithm for converting proofs from the OpenTheory interchange format, which can be translated to and from any of the HOL family of proof languages (HOL4, HOL Light, ProofPower, and Isabelle), into the Metamath language [10]. Daniel Whalen has developed Holophrasm, which can automatically prove many Metamath proofs using machine learning approaches (including multiple neural networks)[72]. However, a discussion of these other programs is beyond the scope of this book.

Like most computer languages, the Metamath language uses the standard (ASCII) characters on a computer keyboard, so it cannot directly represent many of the special symbols that mathematicians use. A useful feature

[3]This includes verification that a sequential list of proof steps results in the specified theorem.

of the Metamath program is its ability to convert its notation into the LaTeX typesetting language. This feature lets you convert the ASCII tokens you've defined into standard mathematical symbols, so you end up with symbols and formulas you are familiar with instead of somewhat cryptic ASCII representations of them. The Metamath program can also generate HTML, making it easy to view results on the web and to see related information by using hypertext links.

Metamath is probably conceptually different from anything you've seen before and some aspects may take some getting used to. This book will help you decide whether Metamath suits your specific needs.

Setting Your Expectations

It is important for you to understand what Metamath is and is not. As mentioned, the Metamath program is *not* an automated theorem prover but rather a proof verifier. Developing a database can be tedious, hard work, especially if you want to make the proofs as short as possible, but it becomes easier as you build up a collection of useful theorems. The purpose of Metamath is simply to document existing mathematics in an absolutely rigorous, computer-verifiable way, not to aid directly in the creation of new mathematics. It also is not a magic solution for learning abstract mathematics, although it may be helpful to be able to actually see the implied rigor behind what you are learning from textbooks, as well as providing hints to work out proofs that you are stumped on.

As of this writing, a sizable set theory database has been developed to provide a foundation for many fields of mathematics, but much more work would be required to develop useful databases for specific fields.

Metamath "knows no math;" it just provides a framework in which to express mathematics. Its language is very small. You can define two kinds of symbols, constants and variables. The only thing Metamath knows how to do is to substitute strings of symbols for the variables in an expression based on instructions you provide it in a proof, subject to certain constraints you specify for the variables. Even the decimal representation of a number is merely a string of certain constants (digits) which together, in a specific context, correspond to whatever mathematical object you choose to define for it; unlike other computer languages, there is no actual number stored inside the computer. In a proof, you in effect instruct Metamath what symbol substitutions to make in previous axioms or theorems and join a sequence of them together to result in the desired theorem. This kind of symbol manipulation captures the essence of mathematics at a preaxiomatic level.

Metamath and Mathematical Literature

In advanced mathematical literature, proofs are usually presented in the form of short outlines that often only an expert can follow. This is partly out

of a desire for brevity, but it would also be unwise (even if it were practical) to present proofs in complete formal detail, since the overall picture would be lost.

A solution I envision that would allow mathematics to remain acceptable to the expert, yet increase its accessibility to non-specialists, consists of a combination of the traditional short, informal proof in print accompanied by a complete formal proof stored in a computer database. In an analogy with a computer program, the informal proof is like a set of comments that describe the overall reasoning and content of the proof, whereas the computer database is like the actual program and provides a means for anyone, even a non-expert, to follow the proof in as much detail as desired, exploring it back through layers of theorems (like subroutines that call other subroutines) all the way back to the axioms of the theory. In addition, the computer database would have the advantage of providing absolute assurance that the proof is correct, since each step can be verified automatically.

There are several other approaches besides Metamath to a project such as this. Section 1.3.4 discusses some of these.

To us, a noble goal would be a database with hundreds of thousands of theorems and their computer-verifiable proofs, encompassing a significant fraction of known mathematics and available for instant access. These would be fully verified by multiple independently-implemented verifiers, to provide extremely high confidence that the proofs are completely correct. The database would allow people to investigate whatever details they were interested in, so that they could confirm whatever portions they wished. Whether or not Metamath is an appropriate choice remains to be seen, but in principle we believe it is sufficient.

Formalism

Over the past fifty years, a group of French mathematicians working collectively under the pseudonym of Bourbaki have co-authored a series of monographs that attempt to rigorously and consistently formalize large bodies of mathematics from foundations. On the one hand, certainly such an effort has its merits; on the other hand, the Bourbaki project has been criticized for its "scholasticism" and "hyperaxiomatics" that hide the intuitive steps that lead to the results [3, p. 191].

Metamath unabashedly carries this philosophy to its extreme and no doubt is subject to the same kind of criticism. Nonetheless I think that in conjunction with conventional approaches to mathematics Metamath can serve a useful purpose. The Bourbaki approach is essentially pedagogic, requiring the reader to become intimately familiar with each detail in a very large hierarchy before he or she can proceed to the next step. The difference with Metamath is that the "reader" (user) knows that all details are contained in its computer database, available as needed; it does not demand that the user know everything but conveniently makes available those portions that

are of interest. As the body of all mathematical knowledge grows larger and larger, no one individual can have a thorough grasp of its entirety. Metamath can finalize and put to rest any questions about the validity of any part of it and can make any part of it accessible, in principle, to a non-specialist.

A Personal Note

Why did I develop Metamath? I enjoy abstract mathematics, but I sometimes get lost in a barrage of definitions and start to lose confidence that my proofs are correct. Or I reach a point where I lose sight of how anything I'm doing relates to the axioms that a theory is based on and am sometimes suspicious that there may be some overlooked implicit axiom accidentally introduced along the way (as happened historically with Euclidean geometry, whose omission of Pasch's axiom went unnoticed for 2000 years [15, p. 160]!). I'm also somewhat lazy and wish to avoid the effort involved in re-verifying the gaps in informal proofs "left to the reader;" I prefer to figure them out just once and not have to go through the same frustration a year from now when I've forgotten what I did. Metamath provides better recovery of my efforts than scraps of paper that I can't decipher anymore. But mostly I find very appealing the idea of rigorously archiving mathematical knowledge in a computer database, providing precision, certainty, and elimination of human error.

Note on Bibliography and Index

The Bibliography usually includes the Library of Congress classification for a work to make it easier for you to find it in on a university library shelf. The Index has author references to pages where their works are cited, even though the authors' names may not appear on those pages.

Acknowledgments

Acknowledgments are first due to my wife, Deborah (who passed away on September 4, 1998), for critiquing the manuscript but most of all for her patience and support. I also wish to thank Joe Wright, Richard Becker, Clarke Evans, Buddha Buck, and Jeremy Henty for helpful comments. Any errors, omissions, and other shortcomings are of course my responsibility.

Note Added June 22, 2005

The original, unpublished version of this book was written in 1997 and distributed via the web. The present edition has been updated to reflect the current Metamath program and databases, as well as more current URLs for Internet sites. Thanks to Josh Purinton, One Hand Clapping, Mel L. O'Cat, and Roy F. Longton for pointing out typographical and other errors. I have also benefitted from numerous discussions with Raph Levien, who

has extended Metamath's philosophy of rigor to result in his *Ghilbert* proof language (`http://ghilbert.org`).

Robert (Bob) Solovay communicated a new result of A. R. D. Mathias on the system of Bourbaki, and the text has been updated accordingly (p. 15).

Bob also pointed out a clarification of the literature regarding category theory and inaccessible cardinals (p. 34), and a misleading statement was removed from the text. Specifically, contrary to a statement in previous editions, it is possible to express "There is a proper class of inaccessible cardinals" in the language of ZFC. This can be done as follows: "For every set x there is an inaccessible cardinal κ such that κ is not in x." Bob writes:[4]

> This axiom is how Grothendieck presents category theory. To each inaccessible cardinal κ one associates a Grothendieck universe $U(\kappa)$. $U(\kappa)$ consists of those sets which lie in a transitive set of cardinality less than κ. Instead of the "category of all groups," one works relative to a universe [considering the category of groups of cardinality less than κ]. Now the category whose objects are all categories "relative to the universe $U(\kappa)$" will be a category not relative to this universe but to the next universe.
>
> All of the things category theorists like to do can be done in this framework. The only controversial point is whether the Grothendieck axiom is too strong for the needs of category theorists. Mac Lane argues that "one universe is enough" and Feferman has argued that one can get by with ordinary ZFC. I don't find Feferman's arguments persuasive. Mac Lane may be right, but when I think about category theory I do it à la Grothendieck.
>
> By the way Mizar adds the axiom "there is a proper class of inaccessibles" precisely so as to do category theory.

The most current information on the Metamath program and databases can always be found at `http://metamath.org`.

Note Added June 24, 2006

The Metamath spec was restricted slightly to make parsers easier to write. See the footnote on p. 114.

Note Added March 10, 2007

I am grateful to Anthony Williams for writing the LaTeX package called `realref.sty` and contributing it to the public domain. This package allows the internal hyperlinks in a PDF file to anchor to specific page numbers instead of just section titles, making the navigation of the PDF file for this book much more pleasant and "logical."

[4] Private communication, Nov. 30, 2002.

A typographical error found by Martin Kiselkov was corrected. A confusing remark about unification was deleted per suggestion of Mel O'Cat.

Note Added May 27, 2009

Several typos found by Kim Sparre were corrected. A note was added that the Poincaré conjecture has been proved (p. 24).

Note Added Nov. 17, 2014

The statement of the Schröder–Bernstein theorem was corrected in Section 1.2.2. Thanks to Bob Solovay for pointing out the error.

Note Added May 25, 2016

Thanks to Jerry James for correcting 16 typos.

Note Added February 25, 2019

David A. Wheeler made a large number of improvements and updates, in coordination with Norman Megill. The predicate calculus axioms were renumbered, and the text makes it clear that they are based on Tarski's system S2; the one slight deviation in axiom ax-6 is explained and justified. The real and complex number axioms were modified to be consistent with `set.mm`. Long-awaited specification changes "1–8" were made, which clarified previously ambiguous points. Some errors in the text involving `$f` and `$d` statements were corrected (the spec was correct, but the in-book explanations unintentionally contradicted the spec). We now have a system for automatically generating narrow PDFs, so that those with smartphones can have easy access to the current version of this document. A new section on deduction was added; it discusses the standard deduction theorem, the weak deduction theorem, deduction style, and natural deduction. Many minor corrections (too numerous to list here) were also made.

Note Added March 7, 2019

This added a description of the Matamath language syntax in Extended Backus–Naur Form (EBNF) in Appendix E, added a brief explanation about typecodes, inserted more examples in the deduction section, and added a variety of smaller improvements.

Note Added April 7, 2019

This version clarified the proper substitution notation, improved the discussion on the weak deduction theorem and natural deduction, documented the `undo` command, updated the information on `write source`, changed

the typecode from `set` to `setvar` to be consistent with the current version of `set.mm`, added more documentation about comment markup (e.g., documented how to create headings), and clarified the differences between various assertion forms (in particular deduction form).

Note Added June 2, 2019

This version fixes a large number of small issues reported by Benoît Jubin, such as editorial issues and the need to document `verify markup` (thank you!). This version also includes specific examples of forms (deduction form, inference form, and closed form).

Chapter 1

Introduction

> I.M.: No, no. There's nothing subjective about it! Everybody knows what a proof is. Just read some books, take courses from a competent mathematician, and you'll catch on.
>
> *Student:* Are you sure?
>
> *I.M.:* Well—it is possible that you won't, if you don't have any aptitude for it. That can happen, too.
>
> *Student:* Then *you* decide what a proof is, and if I don't learn to decide in the same way, you decide I don't have any aptitude.
>
> *I.M.:* If not me, then who?
>
> "The Ideal Mathematician" [1]

Brilliant mathematicians have discovered almost unimaginably profound results that rank among the crowning intellectual achievements of mankind. However, there is a sense in which modern abstract mathematics is behind the times, stuck in an era before computers existed. While no one disputes the remarkable results that have been achieved, communicating these results in a precise way to the uninitiated is virtually impossible. To describe these results, a terse informal language is used which despite its elegance is very difficult to learn. This informal language is not imprecise, far from it, but rather it often has omitted detail and symbols with hidden context that are implicitly understood by an expert but few others. Extremely complex technical meanings are associated with innocent-sounding English words such as "compact" and "measurable" that barely hint at what is actually being said. Anyone who does not keep the precise technical meaning constantly in mind is bound to fail, and acquiring the ability to do this can be achieved only through much practice and hard work. Only the few who complete the painful learning experience can join the small in-group of pure mathematicians. The informal language effectively cuts off the true nature of their knowledge from most everyone else.

[1] [15], p. 40.

Metamath makes abstract mathematics more concrete. It allows a computer to keep track of the complexity associated with each word or symbol with absolute rigor. You can explore this complexity at your leisure, to whatever degree you desire. Whether or not you believe that concepts such as infinity actually "exist" outside of the mind, Metamath lets you get to the foundation for what's really being said.

Metamath also enables completely rigorous and thorough proof verification. Its language is simple enough so that you don't have to rely on the authority of experts but can verify the results yourself, step by step. If you want to attempt to derive your own results, Metamath will not let you make a mistake in reasoning. Even professional mathematicians make mistakes; Metamath makes it possible to thoroughly verify that proofs are correct.

Metamath is a computer language and an associated computer program for archiving, verifying, and studying mathematical proofs at a very detailed level. The Metamath language describes formal mathematical systems and expresses proofs of theorems in those systems. Such a language is called a metalanguage by mathematicians. The Metamath program is a computer program that verifies proofs expressed in the Metamath language. The Metamath program does not have the built-in ability to make logical inferences; it just makes a series of symbol substitutions according to instructions given to it in a proof and verifies that the result matches the expected theorem. It makes logical inferences based only on rules of logic that are contained in a set of axioms, or first principles, that you provide to it as the starting point for proofs.

The complete specification of the Metamath language is only four pages long (Section 4.1, p. 112). Its simplicity may at first make you wonder how it can do much of anything at all. But in fact the kinds of symbol manipulations it performs are the ones that are implicitly done in all mathematical systems at the lowest level. You can learn it relatively quickly and have complete confidence in any mathematical proof that it verifies. On the other hand, it is powerful and general enough so that virtually any mathematical theory, from the most basic to the deeply abstract, can be described with it.

Although in principle Metamath can be used with any kind of mathematics, it is best suited for abstract or "pure" mathematics that is mostly concerned with theorems and their proofs, as opposed to the kind of mathematics that deals with the practical manipulation of numbers. Examples of branches of pure mathematics are logic,[2] set theory,[3] number theory,[4]

[2]Logic is the study of statements that are universally true regardless of the objects being described by the statements. An example is the statement, "if P implies Q, then either P is false or Q is true."

[3]Set theory is the study of general-purpose mathematical objects called "sets," and from it essentially all of mathematics can be derived. For example, numbers can be defined as specific sets, and their properties can be explored using the tools of set theory.

[4]Number theory deals with the properties of positive and negative integers (whole numbers).

group theory,[5] abstract algebra,[6], analysis [7] and topology.[8] Even in physics, Metamath could be applied to certain branches that make use of abstract mathematics, such as quantum logic (used to study aspects of quantum mechanics).

On the other hand, Metamath is less suited to applications that deal primarily with intensive numeric computations. Metamath does not have any built-in representation of numbers; instead, a specific string of symbols (digits) must be syntactically constructed as part of any proof in which an ordinary number is used. For this reason, numbers in Metamath are best limited to specific constants that arise during the course of a theorem or its proof. Numbers are only a tiny part of the world of abstract mathematics. The exclusion of built-in numbers was a conscious decision to help achieve Metamath's simplicity, and there are other software tools if you have different mathematical needs. If you wish to quickly solve algebraic problems, the computer algebra programs MACSYMA, Mathematica, and Maple are specifically suited to handling numbers and algebra efficiently. If you wish to simply calculate numeric or matrix expressions easily, tools such as Octave may be a better choice.

After learning Metamath's basic statement types, any technically oriented person, mathematician or not, can immediately trace any theorem proved in the language as far back as he or she wants, all the way to the axioms on which the theorem is based. This ability suggests a non-traditional way of learning about pure mathematics. Used in conjunction with traditional methods, Metamath could make pure mathematics accessible to people who are not sufficiently skilled to figure out the implicit detail in ordinary textbook proofs. Once you learn the axioms of a theory, you can have complete confidence that everything you need to understand a proof you are studying is all there, at your beck and call, allowing you to focus in on any proof step you don't understand in as much depth as you need, without worrying about getting stuck on a step you can't figure out.[9]

[5]Group theory studies the properties of mathematical objects called groups that obey a simple set of axioms and have properties of symmetry that make them useful in many other fields.

[6]Abstract algebra includes group theory and also studies groups with additional properties that qualify them as "rings" and "fields." The set of real numbers is a familiar example of a field.

[7]Analysis is the study of real and complex numbers.

[8]One area studied by topology are properties that remain unchanged when geometrical objects undergo stretching deformations; for example a doughnut and a coffee cup each have one hole (the cup's hole is in its handle) and are thus considered topologically equivalent. In general, though, topology is the study of abstract mathematical objects that obey a certain (surprisingly simple) set of axioms. See, for example, Munkres [48].

[9]On the other hand, writing proofs in the Metamath language is challenging, requiring a degree of rigor far in excess of that normally taught to students. In a classroom setting, I doubt that writing Metamath proofs would ever replace traditional homework exercises involving informal proofs, because the time needed to work out the details would not allow a course to cover much material. For students who have trouble grasping the implied

Metamath is probably unlike anything you have encountered before. In this first chapter we will look at the philosophy and use of computers in mathematics in order to better understand the motivation behind Metamath. The material in this chapter is not required in order to use Metamath. You may skip it if you are impatient, but I hope you will find it educational and enjoyable. If you want to start experimenting with the Metamath program right away, proceed directly to Chapter 2 (p. 37). To learn the Metamath language, skim Chapter 2 then proceed to Chapter 4 (p. 111).

1.1 Mathematics as a Computer Language

> *The study of mathematics is apt to commence in disappointment. . . .*
> *We are told that by its aid the stars are weighted and the billions of molecules in a drop of water are counted. Yet, like the ghost of Hamlet's father, this great science eludes the efforts of our mental weapons to grasp it.*
>
> Alfred North Whitehead[10]

1.1.1 Is Mathematics "User-Friendly"?

Suppose you have no formal training in abstract mathematics. But popular books you've read offer tempting glimpses of this world filled with profound ideas that have stirred the human spirit. You are not satisfied with the informal, watered-down descriptions you've read but feel it is important to grasp the underlying mathematics itself to understand its true meaning. It's not practical to go back to school to learn it, though; you don't want to dedicate years of your life to it. There are many important things in life, and you have to set priorities for what's important to you. What would happen if you tried to pursue it on your own, in your spare time?

After all, you were able to learn a computer programming language such as Pascal on your own without too much difficulty, even though you had no formal training in computers. You don't claim to be an expert in software design, but you can write a passable program when necessary to suit your needs. Even more important, you know that you can look at anyone else's Pascal program, no matter how complex, and with enough patience figure out exactly how it works, even though you are not a specialist. Pascal allows you do anything that a computer can do, at least in principle. Thus you

rigor in traditional material, writing a few simple proofs in the Metamath language might help clarify fuzzy thought processes. Although somewhat difficult at first, it eventually becomes fun to do, like solving a puzzle, because of the instant feedback provided by the computer.

[10] [73], ch. 1.

know you have the ability, in principle, to follow anything that a computer program can do: you just have to break it down into small enough pieces.

Here's an imaginary scenario of what might happen if you naively adopted this same view of abstract mathematics and tried to pick it up on your own, in a period of time comparable to, say, learning a computer programming language.

A Non-Mathematician's Quest for Truth

> *. . . my daughters have been studying (chemistry) for several semesters, think they have learned differential and integral calculus in school, and yet even today don't know why $x \cdot y = y \cdot x$ is true.*
>
> EDMUND LANDAU[11]

> *Minus times minus is plus,*
> *The reason for this we need not discuss.*
>
> W. H. AUDEN[12]

We'll suppose you are a technically oriented professional, perhaps an engineer, a computer programmer, or a physicist, but probably not a mathematician. You consider yourself reasonably intelligent. You did well in school, learning a variety of methods and techniques in practical mathematics such as calculus and differential equations. But rarely did your courses get into anything resembling modern abstract mathematics, and proofs were something that appeared only occasionally in your textbooks, a kind of necessary evil that was supposed to convince you of a certain key result. Most of your homework consisted of exercises that gave you practice in the techniques, and you were hardly ever asked to come up with a proof of your own.

You find yourself curious about advanced, abstract mathematics. You are driven by an inner conviction that it is important to understand and appreciate some of the most profound knowledge discovered by mankind. But it seems very hard to learn, something that only certain gifted longhairs can access and understand. You are frustrated that it seems forever cut off from you.

Eventually your curiosity drives you to do something about it. You set for yourself a goal of "really" understanding mathematics: not just how to manipulate equations in algebra or calculus according to cookbook rules, but rather to gain a deep understanding of where those rules come from. In fact, you're not thinking about this kind of ordinary mathematics at all, but about a much more abstract, ethereal realm of pure mathematics, where famous

[11] [35], p. vi.
[12] As quoted in [20], p. 64.

results such as Gödel's incompleteness theorem and Cantor's different kinds of infinities reside.

You have probably read a number of popular books, with titles like *Infinity and the Mind* [58], on topics such as these. You found them inspiring but at the same time somewhat unsatisfactory. They gave you a general idea of what these results are about, but if someone asked you to prove them, you wouldn't have the faintest idea of where to begin. Sure, you could give the same overall outline that you learned from the popular books; and in a general sort of way, you do have an understanding. But deep down inside, you know that there is a rigor that is missing, that probably there are many subtle steps and pitfalls along the way, and ultimately it seems you have to place your trust in the experts in the field. You don't like this; you want to be able to verify these results for yourself.

So where do you go next? As a first step, you decide to look up some of the original papers on the theorems you are curious about, or better, obtain some standard textbooks in the field. You look up a theorem you want to understand. Sure enough, it's there, but it's expressed with strange terms and odd symbols that mean absolutely nothing to you. It might as well be written in a foreign language you've never seen before, whose symbols are totally alien. You look at the proof, and you haven't the foggiest notion what each step means, much less how one step follows from another. Well, obviously you have a lot to learn if you want to understand this stuff.

You feel that you could probably understand it by going back to college for another three to six years and getting a math degree. But that does not fit in with your career and the other things in your life and would serve no practical purpose. You decide to seek a quicker path. You figure you'll just trace your way back to the beginning, step by step, as you would do with a computer program, until you understand it. But you quickly find that this is not possible, since you can't even understand enough to know what you have to trace back to.

Maybe a different approach is in order—maybe you should start at the beginning and work your way up. First, you read the introduction to the book to find out what the prerequisites are. In a similar fashion, you trace your way back through two or three more books, finally arriving at one that seems to start at a beginning: it lists the axioms of arithmetic. "Aha!" you naively think, "This must be the starting point, the source of all mathematical knowledge." Or at least the starting point for mathematics dealing with numbers; you have to start somewhere and have no idea what the starting point for other mathematics would be. But the word "axioms" looks promising. So you eagerly read along and work through some elementary exercises at the beginning of the book. You feel vaguely bothered: these don't seem like axioms at all, at least not in the sense that you want to think of axioms. Axioms imply a starting point from which everything else can be built up, according to precise rules specified in the axiom system.

Even though you can understand the first few proofs in an informal way, and are able to do some of the exercises, it's hard to pin down precisely what the rules are. Sure, each step seems to follow logically from the others, but exactly what does that mean? Is the "logic" just a matter of common sense, something vague that we all understand but can never quite state precisely?

You've spent a number of years, off and on, programming computers, and you know that in the case of computer languages there is no question of what the rules are—they are precise and crystal clear. If you follow them, your program will work, and if you don't, it won't. No matter how complex a program, it can always be broken down into simpler and simpler pieces, until you can ultimately identify the bits that are moved around to perform a specific function. Some programs might require a lot of perseverance to accomplish this, but if you focus on a specific portion of it, you don't even necessarily have to know how the rest of it works. Shouldn't there be an analogy in mathematics?

You decide to apply the ultimate test: you ask yourself how a computer could verify or ensure that the steps in these proofs follow from one another. Certainly mathematics must be at least as precisely defined as a computer language, if not more so; after all, computer science itself is based on it. If you can get a computer to verify these proofs, then you should also be able, in principle, to understand them yourself in a very crystal clear, precise way.

You're in for a surprise: you can conceive of no way to convert the proofs, which are in English, to a form that the computer can understand. The proofs are filled with phrases such as "assume there exists a unique $x\ldots$" and "given any y, let z be the number such that…" This isn't the kind of logic you are used to in computer programming, where everything, even arithmetic, reduces to Boolean ones and zeroes if you care to break it down sufficiently. Even though you think you understand the proofs, there seems to be some kind of higher reasoning involved rather than precise rules that define how you manipulate the symbols in the axioms. Whatever it is, it just isn't obvious how you would express it to a computer, and the more you think about it, the more puzzled and confused you get, to the point where you even wonder whether *you* really understand it. There's a lot more to these axioms of arithmetic than meets the eye.

Nobody ever talked about this in school in your applied math and engineering courses. You just learned the rules they gave you, not quite understanding how or why they worked, sometimes vaguely suspicious or uncertain of them, and through homework problems and osmosis learned how to present solutions that satisfied the instructor and earned you an "A." Rarely did you actually "prove" anything in a rigorous way, and the math majors who did do stuff like that seemed to be in a different world.

Of course, there are computer algebra programs that can do mathematics, and rather impressively. They can instantly solve the integrals that you struggled with in freshman calculus, and do much, much more. But when

you look at these programs, what you see is a big collection of algorithms and techniques that evolved and were added to over time, along with some basic software that manipulates symbols. Each algorithm that is built in is the result of someone's theorem whose proof is omitted; you just have to trust the person who proved it and the person who programmed it in and hope there are no bugs. Somehow this doesn't seem to be the essence of mathematics. Although computer algebra systems can generate theorems with amazing speed, they can't actually prove a single one of them.

After some puzzlement, you revisit some popular books on what mathematics is all about. Somewhere you read that all of mathematics is actually derived from something called "set theory." This is a little confusing, because nowhere in the book that presented the axioms of arithmetic was there any mention of set theory, or if there was, it seemed to be just a tool that helps you describe things better—the set of even numbers, that sort of thing. If set theory is the basis for all mathematics, then why are additional axioms needed for arithmetic?

Something is wrong but you're not sure what. One of your friends is a pure mathematician. He knows he is unable to communicate to you what he does for a living and seems to have little interest in trying. You do know that for him, proofs are what mathematics is all about. You ask him what a proof is, and he essentially tells you that, while of course it's based on logic, really it's something you learn by doing it over and over until you pick it up. He refers you to a book, *How to Read and Do Proofs* [63]. Although this book helps you understand traditional informal proofs, there is still something missing you can't seem to pin down yet.

You ask your friend how you would go about having a computer verify a proof. At first he seems puzzled by the question; why would you want to do that? Then he says it's not something that would make any sense to do, but he's heard that you'd have to break the proof down into thousands or even millions of individual steps to do such a thing, because the reasoning involved is at such a high level of abstraction. He says that maybe it's something you could do up to a point, but the computer would be completely impractical once you get into any meaningful mathematics. There, the only way you can verify a proof is by hand, and you can only acquire the ability to do this by specializing in the field for a couple of years in grad school. Anyway, he thinks it all has to do with set theory, although he has never taken a formal course in set theory but just learned what he needed as he went along.

You are intrigued and amazed. Apparently a mathematician can grasp as a single concept something that would take a computer a thousand or a million steps to verify, and have complete confidence in it. Each one of these thousand or million steps must be absolutely correct, or else the whole proof is meaningless. If you added a million numbers by hand, would you trust the result? How do you really know that all these steps are correct, that there isn't some subtle pitfall in one of these million steps, like a bug in a

computer program? After all, you've read that famous mathematicians have occasionally made mistakes, and you certainly know you've made your share on your math homework problems in school.

You recall the analogy with a computer program. Sure, you can understand what a large computer program such as a word processor does, as a single high-level concept or a small set of such concepts, but your ability to understand it in no way ensures that the program is correct and doesn't have hidden bugs. Even if you wrote the program yourself you can't really know this; most large programs that you've written have had bugs that crop up at some later date, no matter how careful you tried to be while writing them.

OK, so now it seems the reason you can't figure out how to make a computer verify proofs is because each step really corresponds to a million small steps. Well, you say, a computer can do a million calculations in a second, so maybe it's still practical to do. Now the puzzle becomes how to figure out what the million steps are that each English-language step corresponds to. Your mathematician friend hasn't a clue, but suggests that maybe you would find the answer by studying set theory. Actually, your friend thinks you're a little off the wall for even wondering such a thing. For him, this is not what mathematics is all about.

The subject of set theory keeps popping up, so you decide it's time to look it up.

You decide to start off on a careful footing, so you start reading a couple of very elementary books on set theory. A lot of it seems pretty obvious, like intersections, subsets, and Venn diagrams. You thumb through one of the books; nowhere is anything about axioms mentioned. The other book relegates to an appendix a brief discussion that mentions a set of axioms called "Zermelo–Fraenkel set theory" and states them in English. You look at them and have no idea what they really mean or what you can do with them. The comments in this appendix say that the purpose of mentioning them is to expose you to the idea, but imply that they are not necessary for basic understanding and that they are really the subject matter of advanced treatments where fine points such as a certain paradox (Russell's paradox[13]) are resolved. Wait a minute—shouldn't the axioms be a starting point, not an ending point? If there are paradoxes that arise without the axioms, how do you know you won't stumble across one accidentally when using the informal approach?

And nowhere do these books describe how "all of mathematics can be derived from set theory" which by now you've heard a few times.

You find a more advanced book on set theory. This one actually lists the axioms of ZF set theory in plain English on page one. *Now* you think your quest has ended and you've finally found the source of all mathematical

[13]Russell's paradox assumes that there exists a set S that is a collection of all sets that don't contain themselves. Now, either S contains itself or it doesn't. If it contains itself, it contradicts its definition. But if it doesn't contain itself, it also contradicts its definition. Russell's paradox is resolved in ZF set theory by denying that such a set S exists.

knowledge; you just have to understand what it means. Here, in one place, is the basis for all of mathematics! You stare at the axioms in awe, puzzle over them, memorize them, hoping that if you just meditate on them long enough they will become clear. Of course, you haven't the slightest idea how the rest of mathematics is "derived" from them; in particular, if these are the axioms of mathematics, then why do arithmetic, group theory, and so on need their own axioms?

You start reading this advanced book carefully, pondering the meaning of every word, because by now you're really determined to get to the bottom of this. The first thing the book does is explain how the axioms came about, which was to resolve Russell's paradox. In fact that seems to be the main purpose of their existence; that they supposedly can be used to derive all of mathematics seems irrelevant and is not even mentioned. Well, you go on. You hope the book will explain to you clearly, step by step, how to derive things from the axioms. After all, this is the starting point of mathematics, like a book that explains the basics of a computer programming language. But something is missing. You find you can't even understand the first proof or do the first exercise. Symbols such as $\exists$ and $\forall$ permeate the page without any mention of where they came from or how to manipulate them; the author assumes you are totally familiar with them and doesn't even tell you what they mean. By now you know that $\exists$ means "there exists" and $\forall$ means "for all," but shouldn't the rules for manipulating these symbols be part of the axioms? You still have no idea how you could even describe the axioms to a computer.

Certainly there is something much different here from the technical literature you're used to reading. A computer language manual almost always explains very clearly what all the symbols mean, precisely what they do, and the rules used for combining them, and you work your way up from there.

After glancing at four or five other such books, you come to the realization that there is another whole field of study that you need just to get to the point at which you can understand the axioms of set theory. The field is called "logic." In fact, some of the books did recommend it as a prerequisite, but it just didn't sink in. You assumed logic was, well, just logic, something that a person with common sense intuitively understood. Why waste your time reading boring treatises on symbolic logic, the manipulation of 1's and 0's that computers do, when you already know that? But this is a different kind of logic, quite alien to you. The subject of NAND and NOR gates is not even touched upon or in any case has to do with only a very small part of this field.

So your quest continues. Skimming through the first couple of introductory books, you get a general idea of what logic is about and what quantifiers ("for all," "there exists") mean, but you find their examples somewhat trivial and mildly annoying ("all dogs are animals," "some animals are dogs," and

such). But all you want to know is what the rules are for manipulating the symbols so you can apply them to set theory. Some formulas describing the relationships among quantifiers ($\exists$ and $\forall$) are listed in tables, along with some verbal reasoning to justify them. Presumably, if you want to find out if a formula is correct, you go through this same kind of mental reasoning process, possibly using images of dogs and animals. Intuitively, the formulas seem to make sense. But when you ask yourself, "What are the rules I need to get a computer to figure out whether this formula is correct?", you still don't know. Certainly you don't ask the computer to imagine dogs and animals.

You look at some more advanced logic books. Many of them have an introductory chapter summarizing set theory, which turns out to be a prerequisite. You need logic to understand set theory, but it seems you also need set theory to understand logic! These books jump right into proving rather advanced theorems about logic, without offering the faintest clue about where the logic came from that allows them to prove these theorems.

Luckily, you come across an elementary book of logic that, halfway through, after the usual truth tables and metaphors, presents in a clear, precise way what you've been looking for all along: the axioms! They're divided into propositional calculus (also called sentential logic) and predicate calculus (also called first-order logic), with rules so simple and crystal clear that now you can finally program a computer to understand them. Indeed, they're no harder than learning how to play a game of chess. As far as what you seem to need is concerned, the whole book could have been written in five pages!

Now you think you've found the ultimate source of mathematical truth. So—the axioms of mathematics consist of these axioms of logic, together with the axioms of ZF set theory. (By now you've also been able to figure out how to translate the ZF axioms from English into the actual symbols of logic which you can now manipulate according to precise, easy-to-understand rules.)

Of course, you still don't understand how "all of mathematics can be derived from set theory," but maybe this will reveal itself in due course.

You eagerly set out to program the axioms and rules into a computer and start to look at the theorems you will have to prove as the logic is developed. All sorts of important theorems start popping up: the deduction theorem, the substitution theorem, the completeness theorem of propositional calculus, the completeness theorem of predicate calculus. Uh-oh, there seems to be trouble. They all get harder and harder, and not one of them can be derived with the axioms and rules of logic you've just been handed. Instead, they all require "metalogic" for their proofs, a kind of mixture of logic and set theory that allows you to prove things *about* the axioms and theorems of logic rather than *with* them.

You plow ahead anyway. A month later, you've spent much of your

free time getting the computer to verify proofs in propositional calculus. You've programmed in the axioms, but you've also had to program in the deduction theorem, the substitution theorem, and the completeness theorem of propositional calculus, which by now you've resigned yourself to treating as rather complex additional axioms, since they can't be proved from the axioms you were given. You can now get the computer to verify and even generate complete, rigorous, formal proofs. Never mind that they may have 100,000 steps—at least now you can have complete, absolute confidence in them. Unfortunately, the only theorems you have proved are pretty trivial and you can easily verify them in a few minutes with truth tables, if not by inspection.

It looks like your mathematician friend was right. Getting the computer to do serious mathematics with this kind of rigor seems almost hopeless. Even worse, it seems that the further along you get, the more "axioms" you have to add, as each new theorem seems to involve additional "metamathematical" reasoning that hasn't been formalized, and none of it can be derived from the axioms of logic. Not only do the proofs keep growing exponentially as you get further along, but the program to verify them keeps getting bigger and bigger as you program in more "metatheorems."[14] The bugs that have cropped up so far have already made you start to lose faith in the rigor you seem to have achieved, and you know it's just going to get worse as your program gets larger.

1.1.2 Mathematics and the Non-Specialist

> *A real proof is not checkable by a machine, or even by any mathematician not privy to the gestalt, the mode of thought of the particular field of mathematics in which the proof is located.*
>
> Davis and Hersh [15]

The bulk of abstract or theoretical mathematics is ordinarily outside the reach of anyone but a few specialists in each field who have completed the necessary difficult internship in order to enter its coterie. The typical intelligent layperson has no reasonable hope of understanding much of it, nor even the specialist mathematician of understanding other fields. It is like a foreign language that has no dictionary to look up the translation; the only way you can learn it is by living in the country for a few years. It is argued that the effort involved in learning a specialty is a necessary process

[14] A metatheorem is usually a statement that is too general to be directly provable in a theory. For example, "if n_1, n_2, and n_3 are integers, then $n_1 + n_2 + n_3$ is an integer" is a theorem of number theory. But "for any integer $k > 1$, if $n_1, \ldots, n_k$ are integers, then $n_1 + \ldots + n_k$ is an integer" is a metatheorem, in other words a family of theorems, one for every k. The reason it is not a theorem is that the general sum $n_1 + \ldots + n_k$ (as a function of k) is not an operation that can be defined directly in number theory.

[15] [15], p. 354.

for acquiring a deep understanding. Of course, this is almost certainly true if one is to make significant contributions to a field; in particular, "doing" proofs is probably the most important part of a mathematician's training. But is it also necessary to deny outsiders access to it? Is it necessary that abstract mathematics be so hard for a layperson to grasp?

A computer normally is of no help whatsoever. Most published proofs are actually just series of hints written in an informal style that requires considerable knowledge of the field to understand. These are the "real proofs" referred to by Davis and Hersh. There is an implicit understanding that, in principle, such a proof could be converted to a complete formal proof. However, it is said that no one would ever attempt such a conversion, even if they could, because that would presumably require millions of steps (Section 1.1.3). Unfortunately the informal style automatically excludes the understanding of the proof by anyone who hasn't gone through the necessary apprenticeship. The best that the intelligent layperson can do is to read popular books about deep and famous results; while this can be helpful, it can also be misleading, and the lack of detail usually leaves the reader with no ability whatsoever to explore any aspect of the field being described.

The statements of theorems often use sophisticated notation that makes them inaccessible to the non-specialist. For a non-specialist who wants to achieve a deeper understanding of a proof, the process of tracing definitions and lemmas back through their hierarchy quickly becomes confusing and discouraging. Textbooks are usually written to train mathematicians or to communicate to people who are already mathematicians, and large gaps in proofs are often left as exercises to the reader who is left at an impasse if he or she becomes stuck.

I believe that eventually computers will enable non-specialists and even intelligent laypersons to follow almost any mathematical proof in any field. Metamath is an attempt in that direction. If all of mathematics were as easily accessible as a computer programming language, I could envision computer programmers and hobbyists who otherwise lack mathematical sophistication exploring and being amazed by the world of theorems and proofs in obscure specialties, perhaps even coming up with results of their own. A tremendous advantage would be that anyone could experiment with conjectures in any field—the computer would offer instant feedback as to whether an inference step was correct.

Mathematicians sometimes have to put up with the annoyance of cranks who lack a fundamental understanding of mathematics but insist that their "proofs" of, say, Fermat's Last Theorem be taken seriously. I think part of the problem is that these people are misled by informal mathematical language, treating it as if they were reading ordinary expository English and failing to appreciate the implicit underlying rigor. Such cranks are rare in the field of computers, because computer languages are much more explicit, and ultimately the proof is in whether a computer program works or not. With

easily accessible computer-based abstract mathematics, a mathematician could say to a crank, "don't bother me until you've demonstrated your claim on the computer!"

1.1.3 An Impossible Dream?

> *Even quite basic theorems would demand almost unbelievably vast books to display their proofs.*
>
> ROBERT E. EDWARDS[16]

> *Oh, of course no one ever really* does *it. It would take forever! You just show that you could do it, that's sufficient.*
>
> "THE IDEAL MATHEMATICIAN" [17]

> *There is a theorem in the primitive notation of set theory that corresponds to the arithmetic theorem '*$1000 + 2000 = 3000$*'. The formula would be forbiddingly long... even if [one] knows the definitions and is asked to simplify the long formula according to them, chances are he will make errors and arrive at some incorrect result.*
>
> HAO WANG[18]

> *The* Principia Mathematica *was the crowning achievement of the formalists. It was also the deathblow of the formalist view.... [Russell] failed, in three enormous volumes, to get beyond the elementary facts of arithmetic. He showed what can be done in principle and what cannot be done in practice. If the mathematical process were really one of strict, logical progression, we would still be counting our fingers.... One theoretician estimates... that a demonstration of one of Ramanujan's conjectures assuming set theory and elementary analysis would take about two thousand pages; the length of a deduction from first principles is nearly inconceivable... The probabilists argue that... any very long proof can at best be viewed as only probably correct...*
>
> RICHARD DE MILLO ET. AL.[19]

[16] [17], p. 68.
[17] [15], p. 40.
[18] [71], p. 140.
[19] [16], pp. 269, 271.

A number of writers have conveyed the impression that the kind of absolute rigor provided by Metamath is an impossible dream, suggesting that a complete, formal verification of a typical theorem would take millions of steps in untold volumes of books. Even if it could be done, the thinking sometimes goes, all meaning would be lost in such a monstrous, tedious verification.

These writers assume, however, that in order to achieve the kind of complete formal verification they desire one must break down a proof into individual primitive steps that make direct reference to the axioms. This is not necessary. There is no reason not to make use of previously proved theorems rather than proving them over and over.

Just as important, definitions can be introduced along the way, allowing very complex formulas to be represented with few symbols. Not doing this can lead to absurdly long formulas. For example, the mere statement of Gödel's incompleteness theorem, which can be expressed with a small number of defined symbols, would require about 20,000 primitive symbols to express it.[20] An extreme example is Bourbaki's language for set theory, which requires 4,523,659,424,929 symbols plus 1,179,618,517,981 disambiguatory links (lines connecting symbol pairs, usually drawn below or above the formula) to express the number "one" [40].

A hierarchy of theorems and definitions permits an exponential growth in the formula sizes and primitive proof steps to be described with only a linear growth in the number of symbols used. Of course, this is how ordinary informal mathematics is normally done anyway, but with Metamath it can be done with absolute rigor and precision.

1.1.4 Beauty

> *No one shall be able to drive us from the paradise that Cantor has created for us.*
>
> David Hilbert[21]

> *Mathematics possesses not only truth, but some supreme beauty —a beauty cold and austere, like that of a sculpture.*
>
> Bertrand Russell[22]

> *Euclid alone has looked on Beauty bare.*
>
> Edna Millay[23]

[20] George S. Boolos, lecture at Massachusetts Institute of Technology, spring 1990.
[21] As quoted in [47], p. 131.
[22] [60].
[23] As quoted in [15], p. 150.

For most people, abstract mathematics is distant, strange, and incomprehensible. Many popular books have tried to convey some of the sense of beauty in famous theorems. But even an intelligent layperson is left with only a general idea of what a theorem is about and is hardly given the tools needed to make use of it. Traditionally, it is only after years of arduous study that one can grasp the concepts needed for deep understanding. Metamath allows you to approach the proof of the theorem from a quite different perspective, peeling apart the formulas and definitions layer by layer until an entirely different kind of understanding is achieved. Every step of the proof is there, pieced together with absolute precision and instantly available for inspection through a microscope with a magnification as powerful as you desire.

A proof in itself can be considered an object of beauty. Constructing an elegant proof is an art. Once a famous theorem has been proved, often considerable effort is made to find simpler and more easily understood proofs. Creating and communicating elegant proofs is a major concern of mathematicians. Metamath is one way of providing a common language for archiving and preserving this information.

The length of a proof can, to a certain extent, be considered an objective measure of its "beauty," since shorter proofs are usually considered more elegant. In the set theory database `set.mm` provided with Metamath, one goal was to make all proofs as short as possible.

1.1.5 Simplicity

> *God made man simple; man's complex problems are of his own devising.*
>
> ECCLES. 7:29[24]

> *God made integers, all else is the work of man.*
>
> LEOPOLD KRONECKER[25]

> *For what is clear and easily comprehended attracts; the complicated repels.*
>
> DAVID HILBERT[26]

The Metamath language is simple and Spartan. Metamath treats all mathematical expressions as simple sequences of symbols, devoid of meaning. The higher-level or "metamathematical" notions underlying Metamath are

[24] Jerusalem Bible.
[25] *Jahresbericht der Deutschen Mathematiker-Vereinigung* , vol. 2, p. 19.
[26] As quoted in [16], p. 273.

about as simple as they could possibly be. Each individual step in a proof involves a single basic concept, the substitution of an expression for a variable, so that in principle almost anyone, whether mathematician or not, can completely understand how it was arrived at.

In one of its most basic applications, Metamath can be used to develop the foundations of mathematics from the very beginning. This is done in the set theory database that is provided with the Metamath package and is the subject matter of Chapter 3. Any language (a metalanguage) used to describe mathematics (an object language) must have a mathematical content of its own, but it is desirable to keep this content down to a bare minimum, namely that needed to make use of the inference rules specified by the axioms. With any metalanguage there is a "chicken and egg" problem somewhat like circular reasoning: you must assume the validity of the mathematics of the metalanguage in order to prove the validity of the mathematics of the object language. The mathematical content of Metamath itself is quite limited. Like the rules of a game of chess, the essential concepts are simple enough so that virtually anyone should be able to understand them (although that in itself will not let you play like a master). The symbols that Metamath manipulates do not in themselves have any intrinsic meaning. Your interpretation of the axioms that you supply to Metamath is what gives them meaning. Metamath is an attempt to strip down mathematical thought to its bare essence and show you exactly how the symbols are manipulated.

Philosophers and logicians, with various motivations, have often thought it important to study "weak" fragments of logic [2] [42], other unconventional systems of logic (such as "modal" logic [8, ch. 27]), and quantum logic in physics [51]. Metamath provides a framework in which such systems can be expressed, with an absolute precision that makes all underlying metamathematical assumptions rigorous and crystal clear.

Some schools of philosophical thought, for example intuitionism and constructivism, demand that the notions underlying any mathematical system be as simple and concrete as possible. Metamath should meet the requirements of these philosophies. Metamath must be taught the symbols, axioms, and rules for a specific theory, from the skeptical (such as intuitionism[27]) to the bold (such as the axiom of choice in set theory[28]).

[27]Intuitionism does not accept the law of excluded middle ("either something is true or it is not true"). See [70, p. xi] for discussion and references on this topic. Consider the theorem, "There exist irrational numbers a and b such that a^b is rational." An intuitionist would reject the following proof: If $\sqrt{2}^{\sqrt{2}}$ is rational, we are done. Otherwise, let $a = \sqrt{2}^{\sqrt{2}}$ and $b = \sqrt{2}$. Then $a^b = 2$, which is rational.

[28]The axiom of choice asserts that given any collection of pairwise disjoint nonempty sets, there exists a set that has exactly one element in common with each set of the collection. It is used to prove many important theorems in standard mathematics. Some philosophers object to it because it asserts the existence of a set without specifying what the set contains [18, p. 154]. In one foundation for mathematics due to Quine, that has not been otherwise shown to be inconsistent, the axiom of choice turns out to be false [14, p. 23]. The `show trace_back` command of the Metamath program allows you to find out

The simplicity of the Metamath language lets the algorithm (computer program) that verifies the validity of a Metamath proof be straightforward and robust. You can have confidence that the theorems it verifies really can be derived from your axioms.

1.1.6 Rigor

> *Rigor became a goal with the Greeks... But the efforts to pursue rigor to the utmost have led to an impasse in which there is no longer any agreement on what it really means. Mathematics remains alive and vital, but only on a pragmatic basis.*
>
> Morris Kline[29]

Kline refers to a much deeper kind of rigor than that which we will discuss in this section. Gödel's incompleteness theorem showed that it is impossible to achieve absolute rigor in standard mathematics because we can never prove that mathematics is consistent (free from contradictions). If mathematics is consistent, we will never know it, but must rely on faith. If mathematics is inconsistent, the best we can hope for is that some clever future mathematician will discover the inconsistency. In this case, the axioms would probably be revised slightly to eliminate the inconsistency, as was done in the case of Russell's paradox, but the bulk of mathematics would probably not be affected by such a discovery. Russell's paradox, for example, did not affect most of the remarkable results achieved by 19th-century and earlier mathematicians. It mainly invalidated some of Gottlob Frege's work on the foundations of mathematics in the late 1800's; in fact Frege's work inspired Russell's discovery. Despite the paradox, Frege's work contains important concepts that have significantly influenced modern logic. Kline's *Mathematics, The Loss of Certainty* [31] has an interesting discussion of this topic.

What *can* be achieved with absolute certainty is the knowledge that if we assume the axioms are consistent and true, then the results derived from them are true. Part of the beauty of mathematics is that it is the one area of human endeavor where absolute certainty can be achieved in this sense. A mathematical truth will remain such for eternity. However, our actual knowledge of whether a particular statement is a mathematical truth is only as certain as the correctness of the proof that establishes it. If the proof of a statement is questionable or vague, we can't have absolute confidence in the truth that the statement claims.

Let us look at some traditional ways of expressing proofs.

Except in the field of formal logic, almost all traditional proofs in mathematics are really not proofs at all, but rather proof outlines or hints as to

whether the axiom of choice, or any other axiom, was assumed by a proof.

[29] [30], p. 1209.

how to go about constructing the proof. Many gaps are left for the reader to fill in. There are several reasons for this. First, it is usually assumed in mathematical literature that the person reading the proof is a mathematician familiar with the specialty being described, and that the missing steps are obvious to such a reader or at least that the reader is capable of filling them in. This attitude is fine for professional mathematicians in the specialty, but unfortunately it often has the drawback of cutting off the rest of the world, including mathematicians in other specialties, from understanding the proof. We discussed one possible resolution to this on p. xiii. Second, it is often assumed that a complete formal proof would require countless millions of symbols (Section 1.1.3). This might be true if the proof were to be expressed directly in terms of the axioms of logic and set theory, but it is usually not true if we allow ourselves a hierarchy of definitions and theorems to build upon, using a notation that allows us to introduce new symbols, definitions, and theorems in a precisely specified way.

Even in formal logic, formal proofs that are considered complete still contain hidden or implicit information. For example, a "proof" is usually defined as a sequence of wffs,[30] each of which is an axiom or follows from a rule applied to previous wffs in the sequence. The implicit part of the proof is the algorithm by which a sequence of symbols is verified to be a valid wff, given the definition of a wff. The algorithm in this case is rather simple, but for a computer to verify the proof, it must have the algorithm built into its verification program.[31] If one deals exclusively with axioms and elementary wffs, it is straightforward to implement such an algorithm. But as more and more definitions are added to the theory in order to make the expression of wffs more compact, the algorithm becomes more and more complicated. A computer program that implements the algorithm becomes larger and harder to understand as each definition is introduced, and thus more prone to bugs. The larger the program, the more suspicious the mathematician may be about the validity of its algorithms. This is especially true because computer programs are inherently hard to follow to begin with, and few people enjoy verifying them manually in detail.

Metamath takes a different approach. Metamath's "knowledge" is limited to the ability to substitute variables for expressions, subject to some

[30] A *wff* or well-formed formula is a mathematical expression (string of symbols) constructed according to some precise rules. A formal mathematical system contains (1) the rules for constructing syntactically correct wffs, (2) a list of starting wffs called axioms, and (3) one or more rules prescribing how to derive new wffs, called theorems, from the axioms or previously derived theorems. An example of such a system is contained in Metamath's set theory database, which defines a formal system from which all of standard mathematics can be derived. Section 2.2.1 steps you through a complete example of a formal system, and you may want to skim it now if you are unfamiliar with the concept.

[31] It is possible, of course, to specify wff construction syntax outside of the program itself with a suitable input language (the Metamath language being an example), but some proof-verification or theorem-proving programs lack the ability to extend wff syntax in such a fashion.

simple constraints. Once the basic algorithm of Metamath is assumed to be debugged, and perhaps independently confirmed, it can be trusted once and for all. The information that Metamath needs to "understand" mathematics is contained entirely in the body of knowledge presented to Metamath. Any errors in reasoning can only be errors in the axioms or definitions contained in this body of knowledge. As a "constructive" language Metamath has no conditional branches or loops like the ones that make computer programs hard to decipher; instead, the language can only build new sequences of symbols from earlier sequences of symbols.

The simplicity of the rules that underlie Metamath not only makes Metamath easy to learn but also gives Metamath a great deal of flexibility. For example, Metamath is not limited to describing standard first-order logic; higher-order logics and fragments of logic can be described just as easily. Metamath gives you the freedom to define whatever wff notation you prefer; it has no built-in conception of the syntax of a wff. With suitable axioms and definitions, Metamath can even describe and prove things about itself. (John Harrison discusses the "reflection" principle involved in self-descriptive systems in [22].)

The flexibility of Metamath requires that its proofs specify a lot of detail, much more than in an ordinary "formal" proof. For example, in an ordinary formal proof, a single step consists of displaying the wff that constitutes that step. In order for a computer program to verify that the step is acceptable, it first must verify that the symbol sequence being displayed is an acceptable wff. Most proof verifiers have at least basic wff syntax built into their programs. Metamath has no hard-wired knowledge of what constitutes a wff built into it; instead every wff must be explicitly constructed based on rules defining wffs that are present in a database. Thus a single step in an ordinary formal proof may be correspond to many steps in a Metamath proof. Despite the larger number of steps, though, this does not mean that a Metamath proof must be significantly larger than an ordinary formal proof. The reason is that since we have constructed the wff from scratch, we know what the wff is, so there is no reason to display it. We only need to refer to a sequence of statements that construct it. In a sense, the display of the wff in an ordinary formal proof is an implicit proof of its own validity as a wff; Metamath just makes the proof explicit. (Section 4.3 describes Metamath's proof notation.)

1.2 Computers and Mathematicians

The computer is important, but not to mathematics.

Paul Halmos[32]

[32]As quoted in [1], p. 121.

Pure mathematicians have traditionally been indifferent to computers, even to the point of disdain. Computer science itself is sometimes considered to fall in the mundane realm of "applied" mathematics, perhaps essential for the real world but intellectually unexciting to those who seek the deepest truths in mathematics. Perhaps a reason for this attitude towards computers is that there is little or no computer software that meets their needs, and there may be a general feeling that such software could not even exist. On the one hand, there are the practical computer algebra systems, which can perform amazing symbolic manipulations in algebra and calculus, yet can't prove the simplest existence theorem, if the idea of a proof is present at all. On the other hand, there are specialized automated theorem provers that technically speaking may generate correct proofs. But sometimes their specialized input notation may be cryptic and their output perceived to be long, inelegant, incomprehensible proofs. The output may be viewed with suspicion, since the program that generates it tends to be very large, and its size increases the potential for bugs. Such a proof may be considered trustworthy only if independently verified and "understood" by a human, but no one wants to waste their time on such a boring, unrewarding chore.

1.2.1 Trusting the Computer

> *. . . I continue to find the quasi-empirical interpretation of computer proofs to be the more plausible. . . . Since not everything that claims to be a computer proof can be accepted as valid, what are the mathematical criteria for acceptable computer proofs?*
>
> Thomas Tymoczko[33]

In some cases, computers have been essential tools for proving famous theorems. But if a proof is so long and obscure that it can be verified in a practical way only with a computer, it is vaguely felt to be suspicious. For example, proving the famous four-color theorem ("a map needs no more than four colors to prevent any two adjacent countries from having the same color") can presently only be done with the aid of a very complex computer program which originally required 1200 hours of computer time. There has been considerable debate about whether such a proof can be trusted and whether such a proof is "real" mathematics [65].

However, under normal circumstances even a skeptical mathematician would have a great deal of confidence in the result of multiplying two numbers on a pocket calculator, even though the precise details of what goes on are hidden from its user. Even the verification on a supercomputer that a huge number is prime is trusted, especially if there is independent verification; no one bothers to debate the philosophical significance of its "proof," even though the actual proof would be so large that it would be completely

[33] [70], p. 245.

impractical to ever write it down on paper. It seems that if the algorithm used by the computer is simple enough to be readily understood, then the computer can be trusted.

Metamath adopts this philosophy. The simplicity of its language makes it easy to learn, and because of its simplicity one can have essentially absolute confidence that a proof is correct. All axioms, rules, and definitions are available for inspection at any time because they are defined by the user; there are no hidden or built-in rules that may be prone to subtle bugs. The basic algorithm at the heart of Metamath is simple and fixed, and it can be assumed to be bug-free and robust with a degree of confidence approaching certainty. Independently written implementations of the Metamath verifier can reduce any residual doubt on the part of a skeptic even further; there are now over a dozen such implementations, written by many people.

1.2.2 Trusting the Mathematician

> *There is no Algebraist nor Mathematician so expert in his science, as to place entire confidence in any truth immediately upon his discovery of it, or regard it as any thing, but a mere probability. Every time he runs over his proofs, his confidence encreases; but still more by the approbation of his friends; and is rais'd to its utmost perfection by the universal assent and applauses of the learned world.*
>
> David Hume[34]

> *Stanislaw Ulam estimates that mathematicians publish 200,000 theorems every year. A number of these are subsequently contradicted or otherwise disallowed, others are thrown into doubt, and most are ignored.*
>
> Richard de Millo et. al.[35]

Whether or not the computer can be trusted, humans of course will occasionally err. Only the most memorable proofs get independently verified, and of these only a handful of truly great ones achieve the status of being "known" mathematical truths that are used without giving a second thought to their correctness.

There are many famous examples of incorrect theorems and proofs in mathematical literature.

- There have been thousands of purported proofs of Fermat's Last Theorem ("no integer solutions exist to $x^n + y^n = z^n$ for $n > 2$"), by

[34] *A Treatise of Human Nature*, as quoted in [16], p. 267.
[35] [16], p. 269.

amateurs, cranks, and well-regarded mathematicians [64, p. 5]. Fermat wrote a note in his copy of Bachet's *Diophantus* that he found "a truly marvelous proof of this theorem but this margin is too narrow to contain it" [33, p. 507]. A recent, much publicized proof by Yoichi Miyaoka was shown to be incorrect (*Science News*, April 9, 1988, p. 230). The theorem was finally proved by Andrew Wiles (*Science News*, July 3, 1993, p. 5), but it initially had some gaps and took over a year after its announcement to be checked thoroughly by experts. On Oct. 25, 1994, Wiles announced that the last gap found in his proof had been filled in.

- In 1882, M. Pasch discovered that an axiom was omitted from Euclid's formulation of geometry; without it, the proofs of important theorems of Euclid are not valid. Pasch's axiom states that a line that intersects one side of a triangle must also intersect another side, provided that it does not touch any of the triangle's vertices. The omission of Pasch's axiom went unnoticed for 2000 years [15, p. 160], in spite of (one presumes) the thousands of students, instructors, and mathematicians who studied Euclid.

- The first published proof of the famous Schröder–Bernstein theorem in set theory was incorrect [18, p. 148]. This theorem states that if there exists a 1-to-1 function[36] from set A into set B and vice-versa, then sets A and B have a 1-to-1 correspondence. Although it sounds simple and obvious, the standard proof is quite long and complex.

- In the early 1900's, Hilbert published a purported proof of the continuum hypothesis, which was eventually established as unprovable by Cohen in 1963 [18, p. 166]. The continuum hypothesis states that no infinity ("transfinite cardinal number") exists whose size (or "cardinality") is between the size of the set of integers and the size of the set of real numbers. This hypothesis originated with German mathematician Georg Cantor in the late 1800's, and his inability to prove it is said to have contributed to mental illness that afflicted him in his later years.

- An incorrect proof of the four-color theorem was published by Kempe in 1879 [13, p. 582]; it stood for 11 years before its flaw was discovered. This theorem states that any map can be colored using only four colors, so that no two adjacent countries have the same color. In 1976 the theorem was finally proved by the famous computer-assisted proof of Haken, Appel, and Koch [65]. Or at least it seems that way. Mathematician H. S. M. Coxeter has doubts [15, p. 58]: "I have a feeling that that is an untidy kind of use of the computers, and the more you correspond with Haken and Appel, the more shaky you seem to be."

[36] A *set* is any collection of objects. A *function* or *mapping* is a rule that assigns to each element of one set (called the function's *domain*) an element from another set.

- Many false "proofs" of the Poincaré conjecture have been proposed over the years. This conjecture states that any object that mathematically behaves like a three-dimensional sphere is a three-dimensional sphere topologically, regardless of how it is distorted. In March 1986, mathematicians Colin Rourke and Eduardo Rêgo caused a stir in the mathematical community by announcing that they had found a proof; in November of that year the proof was found to be false [53, p. 218]. It was finally proved in 2003 by Grigory Perelman [66].

Many counterexamples to "theorems" in recent mathematical literature related to Clifford algebras have been found by Pertti Lounesto (who passed away in 2002). See the web page `http://mathforum.org/library/view/4933.html`.

One of the purposes of Metamath is to allow proofs to be expressed with absolute precision. Developing a proof in the Metamath language can be challenging, because Metamath will not permit even the tiniest mistake. But once the proof is created, its correctness can be trusted immediately, without having to depend on the process of peer review for confirmation.

1.3 The Use of Computers in Mathematics

1.3.1 Computer Algebra Systems

For the most part, you will find that Metamath is not a practical tool for manipulating numbers. (Even proving that $2 + 2 = 4$, if you start with set theory, can be quite complex!) Several commercial mathematics packages are quite good at arithmetic, algebra, and calculus, and as practical tools they are invaluable. But they have no notion of proof, and cannot understand statements starting with "there exists such and such...".

Software packages such as Mathematica [76] do not concern themselves with proofs but instead work directly with known results. These packages primarily emphasize heuristic rules such as the substitution of equals for equals to achieve simpler expressions or expressions in a different form. Starting with a rich collection of built-in rules and algorithms, users can add to the collection by means of a powerful programming language. However, results such as, say, the existence of a certain abstract object without displaying the actual object cannot be expressed (directly) in their languages. The idea of a proof from a small set of axioms is absent. Instead this software simply assumes that each fact or rule you add to the built-in collection of algorithms is valid. One way to view the software is as a large collection of axioms from which the software, with certain goals, attempts to derive new theorems, for example equating a complex expression with a simpler equivalent. But the terms "theorem" and "proof," for example, are not even mentioned in the index of the user's manual for Mathematica. What is also unsatisfactory from a philosophical point of view is that there is no way to

ensure the validity of the results other than by trusting the writer of each application module or tediously checking each module by hand, similar to checking a computer program for bugs.[37] While of course extremely valuable in applied mathematics, computer algebra systems tend to be of little interest to the theoretical mathematician except as aids for exploring certain specific problems.

Because of possible bugs, trusting the output of a computer algebra system for use as theorems in a proof-verifier would defeat the latter's goal of rigor. On the other hand, a fact such that a certain relatively large number is prime, while easy for a computer algebra system to derive, might have a long, tedious proof that could overwhelm a proof-verifier. One approach for linking computer algebra systems to a proof-verifier while retaining the advantages of both is to add a hypothesis to each such theorem indicating its source. For example, a constant MAPLE could indicate the theorem came from the Maple package, and would mean "assuming Maple is consistent, then..." This and many other topics concerning the formalization of mathematics are discussed in John Harrison's very interesting PhD thesis [23].

1.3.2 Automated Theorem Provers

A mathematical theory is "decidable" if a mechanical method or algorithm exists that is guaranteed to determine whether or not a particular formula is a theorem. Among the few theories that are decidable is elementary geometry, as was shown by a classic result of logician Alfred Tarski in 1948 [68].[38] But most theories, including elementary arithmetic, are undecidable. This fact contributes to keeping mathematics alive and well, since many mathematicians believe that they will never be replaced by computers (if they believe Roger Penrose's argument that a computer can never replace the brain [52]). In fact, elementary geometry is often considered a "dead" field for the simple reason that it is decidable.

On the other hand, the undecidability of a theory does not mean that one cannot use a computer to search for proofs, providing one is willing to

[37] Two examples illustrate why the knowledge database of computer algebra systems should sometimes be regarded with a certain caution. If you ask Mathematica (version 3.0) to `Solve[x^n + y^n == z^n , n]` it will respond with {{`n->-2`}, {`n->-1`}, {`n->1`}, {`n->2`}}. In other words, Mathematica seems to "know" that Fermat's Last Theorem is true! (At the time this version of Mathematica was released this fact was unknown.) If you ask Maple to `solve(x^2 = 2^x)` then `simplify({"})`, it returns the solution set {2, 4}, apparently unaware that $-0.7666647\ldots$ is also a solution.

[38] Tarski's result actually applies to a subset of the geometry discussed in elementary textbooks. This subset includes most of what would be considered elementary geometry but it is not powerful enough to express, among other things, the notions of the circumference and area of a circle. Extending the theory in a way that includes notions such as these makes the theory undecidable, as was also shown by Tarski. Tarski's algorithm is far too inefficient to implement practically on a computer. A practical algorithm for proving a smaller subset of geometry theorems (those not involving concepts of "order" or "continuity") was discovered by Chinese mathematician Wu Wen-tsün in 1977 [12].

give up if a proof is not found after a reasonable amount of time. The field of automated theorem proving specializes in pursuing such computer searches. Among the more successful results to date are those based on an algorithm known as Robinson's resolution principle [56].

Automated theorem provers can be excellent tools for those willing to learn how to use them. But they are not widely used in mainstream pure mathematics, even though they could probably be useful in many areas. There are several reasons for this. Probably most important, the main goal in pure mathematics is to arrive at results that are considered to be deep or important; proving them is essential but secondary. Usually, an automated theorem prover cannot assist in this main goal, and by the time the main goal is achieved, the mathematician may have already figured out the proof as a by-product. There is also a notational problem. Mathematicians are used to using very compact syntax where one or two symbols (heavily dependent on context) can represent very complex concepts; this is part of the hierarchy they have built up to tackle difficult problems. A theorem prover on the other hand might require that a theorem be expressed in "first-order logic," which is the logic on which most of mathematics is ultimately based but which is not ordinarily used directly because expressions can become very long. Some automated theorem provers are experimental programs, limited in their use to very specialized areas, and the goal of many is simply research into the nature of automated theorem proving itself. Finally, much research remains to be done to enable them to prove very deep theorems. One significant result was a computer proof by Larry Wos and colleagues that every Robbins algebra is a Boolean algebra (*New York Times*, Dec. 10, 1996).[39]

How does Metamath relate to automated theorem provers? A theorem prover is primarily concerned with one theorem at a time (perhaps tapping into a small database of known theorems) whereas Metamath is more like a theorem archiving system, storing both the theorem and its proof in a database for access and verification. Metamath is one answer to "what do you do with the output of a theorem prover?" and could be viewed as the next step in the process. Automated theorem provers could be useful tools for

[39] In 1933, E. V. Huntington presented the following axiom system for Boolean algebra with a unary operation n and a binary operation $+$:

$$x + y = y + x$$
$$(x + y) + z = x + (y + z)$$
$$n(n(x) + y) + n(n(x) + n(y)) = x$$

Herbert Robbins, a student of Huntington, conjectured that the last equation can be replaced with a simpler one:

$$n(n(x + y) + n(x + n(y))) = x$$

Robbins and Huntington could not find a proof. The problem was later studied unsuccessfully by Tarski and his students, and it remained an unsolved problem until a computer found the proof in 1996. For more information on the Robbins algebra problem see [77].

helping develop its database. Note that very long, automatically generated proofs can make your database fat and ugly and cause Metamath's proof verification to take a long time to run. Unless you have a particularly good program that generates very concise proofs, it might be best to consider the use of automatically generated proofs as a quick-and-dirty approach, to be manually rewritten at some later date.

The program OTTER[40], later succeeded by prover9[41], have been historically influential. The E prover[42] is a maintained automated theorem prover for full first-order logic with equality. There are many other automated theorem provers as well.

If you want to combine automated theorem provers with Metamath consider investigating the book *Automated Reasoning: Introduction and Applications* [77]. This book discusses how to use OTTER in a way that can not only able to generate relatively efficient proofs, it can even be instructed to search for shorter proofs. The effective use of OTTER (and similar tools) does require a certain amount of experience, skill, and patience. The axiom system used in the `set.mm` set theory database can be expressed to OTTER using a method described in [41].[43] When successful, this method tends to generate short and clever proofs, but my experiments with it indicate that the method will find proofs within a reasonable time only for relatively easy theorems. It is still fun to experiment with.

Reference [7] surveys a number of approaches people have explored in the field of automated theorem proving.

1.3.3 Interactive Theorem Provers

Finding proofs completely automatically is difficult, so there are some interactive theorem provers that allow a human to guide the computer to find a proof. Examples include HOL Light[44], Isabelle[45], HOL[46], and Coq[47].

A major difference between most of these tools and Metamath is that the "proofs" are actually programs that guide the program to find a proof, and not the proof itself. For example, an Isabelle/HOL proof might apply a step `apply (blast dest: rearrange reduction)`. The `blast` instruction applies an automatic tableux prover and returns if it found a sequence of proof steps that work... but the sequence is not considered part of the proof.

A good overview of higher-level proof verification languages (such as LCF and HOL) is given in [22]. All of these languages are fundamentally

[40] `http://www.cs.unm.edu/~mccune/otter/`.

[41] `https://www.cs.unm.edu/~mccune/mace4/`.

[42] `https://github.com/eprover/eprover`.

[43] To use those axioms with OTTER, they must be restated in a way that eliminates the need for "dummy variables." See the Comment on p. 125.

[44] `https://www.cl.cam.ac.uk/~jrh13/hol-light/`.

[45] `http://www.cl.cam.ac.uk/Research/HVG/Isabelle`.

[46] `https://hol-theorem-prover.org/`.

[47] `https://coq.inria.fr/`.

different from Metamath in that much of the mathematical foundational knowledge is embedded in the underlying proof-verification program, rather than placed directly in the database that is being verified. These can have a steep learning curve for those without a mathematical background. For example, one usually must have a fair understanding of mathematical logic in order to follow their proofs.

1.3.4 Proof Verifiers

A proof verifier is a program that doesn't generate proofs but instead verifies proofs that you give it. Many proof verifiers have limited built-in automated proof capabilities, such as figuring out simple logical inferences (while still being guided by a person who provides the overall proof). Metamath has no built-in automated proof capability other than the limited capability of its Proof Assistant.

Proof-verification languages are not used as frequently as they might be. Pure mathematicians are more concerned with producing new results, and such detail and rigor would interfere with that goal. The use of computers in pure mathematics is primarily focused on automated theorem provers (not verifiers), again with the ultimate goal of aiding the creation of new mathematics. Automated theorem provers are usually concerned with attacking one theorem at time rather than making a large, organized database easily available to the user. Metamath is one way to help close this gap.

By itself Metamath is a mostly a proof verifier. This does not mean that other approaches can't be used; the difference is that in Metamath, the results of various provers must be recorded step-by-step so that they can be verified.

Another proof-verification language is Mizar, which can display its proofs in the informal language that mathematicians are accustomed to. Information on the Mizar language is available at `http://mizar.org`.

For the working mathematician, Mizar is an excellent tool for rigorously documenting proofs. Mizar typesets its proofs in the informal English used by mathematicians (and, while fine for them, are just as inscrutable by laypersons!). A price paid for Mizar is a relatively steep learning curve of a couple of weeks. Several mathematicians are actively formalizing different areas of mathematics using Mizar and publishing the proofs in a dedicated journal. Unfortunately the task of formalizing mathematics is still looked down upon to a certain extent since it doesn't involve the creation of "new" mathematics.

The closest system to Metamath is the *Ghilbert* proof language (`http://ghilbert.org`) system developed by Raph Levien. Ghilbert is a formal proof checker heavily inspired by Metamath. Ghilbert statements are s-expressions (a la Lisp), which is easy for computers to parse but many people find them hard to read. There are a number of differences in their specific constructs, but there is at least one tool to translate some Metamath

materials into Ghilbert. As of 2019 the Ghilbert community is smaller and less active than the Metamath community. That said, the Metamath and Ghilbert communities overlap, and fruitful conversations between them have occurred many times over the years.

1.3.5 Creating a Database of Formalized Mathematics

Besides Metamath, there are several other ongoing projects with the goal of formalizing mathematics into computer-verifiable databases. Understanding some history will help.

The QED[48] project arose in 1993 and its goals were outlined in the QED manifesto. The QED manifesto was a proposal for a computer-based database of all mathematical knowledge, strictly formalized and with all proofs having been checked automatically. The project had a conference in 1994 and another in 1995; there was also a "twenty years of the QED manifesto" workshop in 2014. Its ideals are regularly reraised.

In a 2007 paper, Freek Wiedijk identified two reasons for the failure of the QED project as originally envisioned:[75]

- Very few people are working on formalization of mathematics. There is no compelling application for fully mechanized mathematics.
- Formalized mathematics does not yet resemble traditional mathematics. This is partly due to the complexity of mathematical notation, and partly to the limitations of existing theorem provers and proof assistants.

But this did not end the dream of formalizing mathematics into computer-verifiable databases. The problems that led to the QED manifesto are still with us, even though the challenges were harder than originally considered. What has happened instead is that various independent projects have worked towards formalizing mathematics into computer-verifiable databases, each simultaneously competing and cooperating with each other.

A concrete way to see this is Freek Wiedijk's "Formalizing 100 Theorems" list[49] which shows the progress different systems have made on a challenge list of 100 mathematical theorems.[50] The top systems as of February 2019 (in order of the number of challenges completed) are HOL Light, Isabelle, Metamath, Coq, and Mizar.

The Metamath 100[51] page (maintained by David A. Wheeler) shows the progress of Metamath (specifically its `set.mm` database) against this challenge list maintained by Freek Wiedijk. The Metamath `set.mm` database

[48] `http://www-unix.mcs.anl.gov/qed`.

[49] `http://www.cs.ru.nl/%7Efreek/100/`.

[50] This is not the only list of "interesting" theorems. Another interesting list was posted by Oliver Knill's list [32].

[51] `http://us.metamath.org/mm_100.html`

has made a lot of progress over the years, in part because working to prove those challenge theorems required defining various terms and proving their properties as a prerequisite. Here are just a few of the many statements that have been formally proven with Metamath:

- 1. The Irrationality of the Square Root of 2 (`sqr2irr`, by Norman Megill, 2001-08-20)
- 2. The Fundamental Theorem of Algebra (`fta`, by Mario Carneiro, 2014-09-15)
- 22. The Non-Denumerability of the Continuum (`ruc`, by Norman Megill, 2004-08-13)
- 54. The Konigsberg Bridge Problem (`konigsberg`, by Mario Carneiro, 2015-04-16)
- 83. The Friendship Theorem (`friendship`, by Alexander W. van der Vekens, 2018-10-09)

We thank all of those who have developed at least one of the Metamath 100 proofs, and we particularly thank Mario Carneiro who has contributed the most Metamath 100 proofs as of 2019. The Metamath 100 page shows the list of all people who have contributed a proof, and links to graphs and charts showing progress over time. We encourage others to work on proving theorems not yet proven in Metamath, since doing so improves the work as a whole.

Each of the math formalization systems (including Metamath) has different strengths and weaknesses, depending on what you value. Key aspects that differentiate Metamath from the other top systems are:

- Metamath is not tied to any particular set of axioms.
- Metamath can show every step of every proof, no exceptions. Most other provers only assert that a proof can be found, and do not show every step. This also makes verification fast, because the system does not need to rediscover proof details.
- The Metamath verifier has been re-implemented in many different programming languages, so verification can be done by multiple implementations. In particular, the `set.mm` database is verified by four different verifiers written in four different languages by four different authors. This greatly reduces the risk of accepting an invalid proof due to an error in the verifier.
- Proofs stay proven. In some systems, changes to the system's syntax or how a tactic works causes proofs to fail in later versions, causing older work to become essentially lost. Metamath's language is extremely

small and fixed, so once a proof is added to a database, the database can be rechecked with later versions of the Metamath program and with other verifiers of Metamath databases. If an axiom or key definition needs to be changed, it is easy to manipulate the database as a whole to handle the change without touching the underlying verifier. Since re-verification of an entire database takes seconds, there is never a reason to delay complete verification. This aspect is especially compelling if your goal is to have a long-term database of proofs.

- Licensing is generous. The main Metamath databases are released to the public domain, and the main Metamath program is open source software under a standard, widely-used license.

- Substitutions are easy to understand, even by those who are not professional mathematicians.

Of course, other systems may have advantages over Metamath that are more compelling, depending on what you value. In any case, we hope this helps you understand Metamath within a wider context.

1.3.6 In Summary

To summarize our discussions of computers and mathematics, computer algebra systems can be viewed as theorem generators focusing on a narrow realm of mathematics (numbers and their properties), automated theorem provers as proof generators for specific theorems in a much broader realm covered by a built-in formal system such as first-order logic, interactive theorem provers require human guidance, proof verifiers verify proofs but historically they have been restricted to first-order logic. Metamath, in contrast, is a proof verifier and documenter whose realm is essentially unlimited.

1.4 Mathematics and Metamath

1.4.1 Standard Mathematics

There are a number of ways that Metamath can be used with standard mathematics. The most satisfying way philosophically is to start at the very beginning, and develop the desired mathematics from the axioms of logic and set theory. This is the approach taken in the `set.mm` database (also known as the Metamath Proof Explorer). Among other things, this database builds up to the axioms of real and complex numbers (see Section 3.7), and a standard development of analysis, for example, could start at that point, using it as a basis. Besides this philosophical advantage, there are practical advantages to having all of the tools of set theory available in the supporting infrastructure.

On the other hand, you may wish to start with the standard axioms of a mathematical theory without going through the set theoretical proofs of those axioms. You will need mathematical logic to make inferences, but if you wish you can simply introduce theorems of logic as "axioms" wherever you need them, with the implicit assumption that in principle they can be proved, if they are obvious to you. If you choose this approach, you will probably want to review the notation used in `set.mm` so that your own notation will be consistent with it.

1.4.2 Other Formal Systems

Unlike some programs, Metamath is not limited to any specific area of mathematics, nor committed to any particular mathematical philosophy such as classical logic versus intuitionism, nor limited, say, to expressions in first-order logic. Although the database `set.mm` describes standard logic and set theory, Metamath is actually a general-purpose language for describing a wide variety of formal systems. Non-standard systems such as modal logic, intuitionist logic, higher-order logic, quantum logic, and category theory can all be described with the Metamath language. You define the symbols you prefer and tell Metamath the axioms and rules you want to start from, and Metamath will verify any inferences you make from those axioms and rules. A simple example of a non-standard formal system is Hofstadter's MIU system, whose Metamath description is presented in Appendix D.

This is not hypothetical. The largest Metamath database is `set.mm`), aka the Metamath Proof Explorer, which uses the most common axioms for mathematical foundations (specifically classical logic combined with Zermelo–Fraenkel set theory with the Axiom of Choice). But other Metamath databases are available:

- The database `iset.mm`, aka the Intuitionistic Logic Explorer, uses intuitionistic logic (a constructivist point of view) instead of classical logic.
- The database `nf.mm`, aka the New Foundations Explorer, constructs mathematics from scratch, starting from Quine's New Foundations (NF) set theory axioms.
- The database `hol.mm`, aka the Higher-Order Logic (HOL) Explorer, starts with HOL (also called simple type theory) and derives equivalents to ZFC axioms, connecting the two approaches.

Since the days of David Hilbert, mathematicians have been concerned with the fact that the metalanguage used to describe mathematics may be stronger than the mathematics being described. Metamath's underlying finitary, constructive nature provides a good philosophical basis for studying even the weakest logics.

The usual treatment of many non-standard formal systems uses model theory or proof theory to describe these systems; these theories, in turn, are based on standard set theory. In other words, a non-standard formal system is defined as a set with certain properties, and standard set theory is used to derive additional properties of this set. The standard set theory database provided with Metamath can be used for this purpose, and when used this way the development of a special axiom system for the non-standard formal system becomes unnecessary. The model- or proof-theoretic approach often allows you to prove much deeper results with less effort.

Metamath supports both approaches. You can define the non-standard formal system directly, or define the non-standard formal system as a set with certain properties, whichever you find most helpful.

1.4.3 Metamath and Its Philosophy

Closely related to Metamath is a philosophy or way of looking at mathematics. This philosophy is related to the formalist philosophy of Hilbert and his followers [30, pp. 1203–1208] [4, p. 6]. In this philosophy, mathematics is viewed as nothing more than a set of rules that manipulate symbols, together with the consequences of those rules. While the mathematics being described may be complex, the rules used to describe it (the "metamathematics") should be as simple as possible. In particular, proofs should be restricted to dealing with concrete objects (the symbols we write on paper rather than the abstract concepts they represent) in a constructive manner; these are called "finitary" proofs [62, pp. 2–3].

Whether or not you find Metamath interesting or useful will in part depend on the appeal you find in its philosophy, and this appeal will probably depend on your particular goals with respect to mathematics. For example, if you are a pure mathematician at the forefront of discovering new mathematical knowledge, you will probably find that the rigid formality of Metamath stifles your creativity. On the other hand, we would argue that once this knowledge is discovered, there are advantages to documenting it in a standard format that will make it accessible to others. Sixty years from now, your field may be dormant, and as Davis and Hersh put it, your "writings would become less translatable than those of the Maya" [15, p. 37].

1.4.4 A History of the Approach Behind Metamath

Probably the one work that has had the most motivating influence on Metamath is Whitehead and Russell's monumental *Principia Mathematica* [74], whose aim was to deduce all of mathematics from a small number of primitive ideas, in a very explicit way that in principle anyone could understand and follow. While this work was tremendously influential in its time, from a modern perspective it suffers from several drawbacks. Both its notation and its underlying axioms are now considered dated and are

no longer used. From our point of view, its development is not really as accessible as we would like to see; for practical reasons, proofs become more and more sketchy as its mathematics progresses, and working them out in fine detail requires a degree of mathematical skill and patience that many people don't have. There are numerous small errors, which is understandable given the tedious, technical nature of the proofs and the lack of a computer to verify the details. However, even today *Principia Mathematica* stands out as the work closest in spirit to Metamath. It remains a mind-boggling work, and one can't help but be amazed at seeing "$1 + 1 = 2$" finally appear on page 83 of Volume II (Theorem *110.643).

The origin of the proof notation used by Metamath dates back to the 1950's, when the logician C. A. Meredith expressed his proofs in a compact notation called "condensed detachment" [25] [29] [45] [54]. This notation allows proofs to be communicated unambiguously by merely referencing the axiom, rule, or theorem used at each step, without explicitly indicating the substitutions that have to be made to the variables in that axiom, rule, or theorem. Ordinarily, condensed detachment is more or less limited to propositional calculus. The concept has been extended to first-order logic in [41], making it is easy to write a small computer program to verify proofs of simple first-order logic theorems.

A key concept behind the notation of condensed detachment is called "unification," which is an algorithm for determining what substitutions to variables have to be made to make two expressions match each other. Unification was first precisely defined by the logician J. A. Robinson, who used it in the development of a powerful theorem-proving technique called the "resolution principle" [56]. Metamath does not make use of the resolution principle, which is intended for systems of first-order logic. Metamath's use is not restricted to first-order logic, and as we have mentioned it does not automatically discover proofs. However, unification is a key idea behind Metamath's proof notation, and Metamath makes use of a very simple version of it (Section 4.3.1).

1.4.5 Metamath and First-Order Logic

First-order logic is the supporting structure for standard mathematics. On top of it is set theory, which contains the axioms from which virtually all of mathematics can be derived—a remarkable fact.[52]

One of the things that makes Metamath more practical for first-order theories is a set of axioms for first-order logic designed specifically with Metamath's approach in mind. These are included in the database `set.mm`.

[52] An exception seems to be category theory. There are several schools of thought on whether category theory is derivable from set theory. At a minimum, it appears that an additional axiom is needed that asserts the existence of an "inaccessible cardinal" (a type of infinity so large that standard set theory can't prove or deny that it exists). For more information, see [24, pp. 328–331] and [6].

See Chapter 3 for a detailed description; the axioms are shown in Section 3.3. While logically equivalent to standard axiom systems, our axiom system breaks up the standard axioms into smaller pieces such that from them, you can directly derive what in other systems can only be derived as higher-level "metatheorems." In other words, it is more powerful than the standard axioms from a metalogical point of view. A rigorous justification for this system and its "metalogical completeness" is found in [41]. The system is closely related to a system developed by Monk and Tarski in 1965 [46].

For example, the formula $\exists x\, x = y$ (given y, there exists some x equal to it) is a theorem of logic,[53] whether or not x and y are distinct variables. In many systems of logic, we would have to prove two theorems to arrive at this result. First we would prove "$\exists x\, x = x$," then we would separately prove "$\exists x\, x = y$, where x and y are distinct variables." We would then combine these two special cases "outside of the system" (i.e. in our heads) to be able to claim, "$\exists x\, x = y$, regardless of whether x and y are distinct." In other words, the combination of the two special cases is a metatheorem. In the system of logic used in Metamath's set theory database, the axioms of logic are broken down into small pieces that allow them to be reassembled in such a way that theorems such as these can be proved directly.

Breaking down the axioms in this way makes them look peculiar and not very intuitive at first, but rest assured that they are correct and complete. Their correctness is ensured because they are theorem schemes of standard first-order logic (which you can easily verify if you are a logician). Their completeness follows from the fact that we explicitly derive the standard axioms of first-order logic as theorems. Deriving the standard axioms is somewhat tricky, but once we're there, we have at our disposal a system that is less awkward to work with in formal proofs. In technical terms that logicians understand, we eliminate the cumbersome concepts of "free variable," "bound variable," and "proper substitution" as primitive notions. These concepts are present in our system but are defined in terms of concepts expressed by the axioms and can be eliminated in principle. In standard systems, these concepts are really like additional, implicit axioms that are somewhat complex and cannot be eliminated.

The traditional approach to logic, wherein free variables and proper substitution is defined, is also possible to do directly in the Metamath language. However, the notation tends to become awkward, and there are disadvantages: for example, extending the definition of a wff with a definition is awkward, because the free variable and proper substitution concepts have

[53] Specifically, it is a theorem of those systems of logic that assume non-empty domains. It is not a theorem of more general systems that include the empty domain, in which nothing exists, period! Such systems are called "free logics." For a discussion of these systems, see [36]. Since our use for logic is as a basis for set theory, which has a non-empty domain, it is more convenient (and more traditional) to use a less general system. An interesting curiosity is that, using a free logic as a basis for Zermelo–Fraenkel set theory (with the redundant Axiom of the Null Set omitted), we cannot even prove the existence of a single set without assuming the axiom of infinity!

to have their definitions also extended. Our choice of axioms for `set.mm` is to a certain extent a matter of style, in an attempt to achieve overall simplicity, but you should also be aware that the traditional approach is possible as well if you should choose it.

Chapter 2

Using the Metamath Program

2.1 Installation

The way that you install Metamath on your computer system will vary for different computers. Current instructions are provided with the Metamath program download at `http://metamath.org`. In general, the installation is simple. There is one file containing the Metamath program itself. This file is usually called `metamath` or `metamath.`*xxx* where *xxx* is the convention (such as `exe`) for an executable program on your operating system. There are several additional files containing samples of the Metamath language, all ending with `.mm`. The file `set.mm` contains logic and set theory and can be used as a starting point for other areas of mathematics.

You will also need a text editor capable of editing plain ASCII[1] text in order to prepare your input files. Most computers have this capability built in. Note that plain text is not necessarily the default for some word processing programs, especially if they can handle different fonts; for example, with Microsoft Word, you must save the file in the format "Text Only With Line Breaks" to get a plain text file.[2]

On some computer systems, Metamath does not have the capability to print its output directly; instead, you send its output to a file (using the `open` commands described later). The way you print this output file depends on your computer. Some computers have a print command, whereas with others, you may have to read the file into an editor and print it from there.

[1] American Standard Code for Information Interchange.

[2] It is recommended that all lines in a Metamath source file be 79 characters or less in length for compatibility among different computer terminals. When creating a source file on an editor such as Word, select a monospaced font such as Courier or Monaco to make this easier to achieve. Better yet, just use a plain text editor such as Notepad.

If you want to print your Metamath source files with typeset formulas containing standard mathematical symbols, you will need the LaTeX typesetting program, which is widely and freely available for most operating systems. It runs natively on Unix and Linux, and can be installed on Windows as part of the free Cygwin package (`http://cygwin.com`).

You can also produce HTML[3] web pages. The `help html` command in the Metamath program will assist you with this feature.

2.2 Your First Formal System

2.2.1 From Nothing to Zero

To give you a feel for what the Metamath language looks like, we will take a look at a very simple example from formal number theory. This example is taken from Mendelson [43, p. 123].[4] We will look at a small subset of this theory, namely that part needed for the first number theory theorem proved in [43].

First we will look at a standard formal proof for the example we have picked, then we will look at the Metamath version. If you have never been exposed to formal proofs, the notation may seem to be such overkill to express such simple notions that you may wonder if you are missing something. You aren't. The concepts involved are in fact very simple, and a detailed breakdown in this fashion is necessary to express the proof in a way that can be verified mechanically. And as you will see, Metamath breaks the proof down into even finer pieces so that the mechanical verification process can be about as simple as possible.

Before we can introduce the axioms of the theory, we must define the syntax rules for forming legal expressions (combinations of symbols) with which those axioms can be used. The number 0 is a **term**; and if t and r are terms, so is $(t + r)$. Here, t and r are "metavariables" ranging over terms; they themselves do not appear as symbols in an actual term. Some examples of actual terms are $(0 + 0)$ and $((0 + 0) + 0)$. (Note that our theory describes only the number zero and sums of zeroes. Of course, not much can be done with such a trivial theory, but remember that we have picked a very small subset of complete number theory for our example. The important thing for you to focus on is our definitions that describe how symbols are combined to form valid expressions, and not on the content or meaning of those expressions.) If t and r are terms, an expression of the form $t = r$ is a **wff** (well-formed formula); and if P and Q are wffs, so is $(P \rightarrow Q)$ (which means "P implies Q" or "if P then Q"). Here P and Q are metavariables ranging over wffs. Examples of actual wffs are $0 = 0$,

[3] HyperText Markup Language.

[4] To keep the example simple, we have changed the formalism slightly, and what we call axioms are strictly speaking theorems in [43].

$(0 + 0) = 0$, $(0 = 0 \to (0 + 0) = 0)$, and $(0 = 0 \to (0 = 0 \to 0 = (0 + 0)))$. (Our notation makes use of more parentheses than are customary, but the elimination of ambiguity this way simplifies our example by avoiding the need to define operator precedence.)

The **axioms** of our theory are all wffs of the following form, where t, r, and s are any terms:

(A1) $$(t = r \to (t = s \to r = s))$$

(A2) $$(t + 0) = t$$

Note that there are an infinite number of axioms since there are an infinite number of possible terms. A1 and A2 are properly called "axiom schemes," but we will refer to them as "axioms" for brevity.

An axiom is a **theorem**; and if P and $(P \to Q)$ are theorems (where P and Q are wffs), then Q is also a theorem. The second part of this definition is called the modus ponens (MP) rule of inference. It allows us to obtain new theorems from old ones.

The **proof** of a theorem is a sequence of one or more theorems, each of which is either an axiom or the result of modus ponens applied to two previous theorems in the sequence, and the last of which is the theorem being proved.

The theorem we will prove for our example is very simple: $t = t$. The proof of our theorem follows. Study it carefully until you feel sure you understand it.

1.	$(t + 0) = t$	(by axiom A2)
2.	$(t + 0) = t$	(by axiom A2)
3.	$((t + 0) = t \to ((t + 0) = t \to t = t))$	(by axiom A1)
4.	$((t + 0) = t \to t = t)$	(by MP applied to steps 2 and 3)
5.	$t = t$	(by MP applied to steps 1 and 4)

(You may wonder why step 1 is repeated twice. This is not necessary in the formal language we have defined, but in Metamath's "reverse Polish notation" for proofs, a previous step can be referred to only once. The repetition of step 1 here will enable you to see more clearly the correspondence of this proof with the Metamath version on p. 48.)

Our theorem is more properly called a "theorem scheme," for it represents an infinite number of theorems, one for each possible term t. Two examples of actual theorems would be $0 = 0$ and $(0 + 0) = (0 + 0)$. Rarely do we prove actual theorems, since by proving schemes we can prove an infinite number of theorems in one fell swoop. Similarly, our proof should really be called a "proof scheme." To obtain an actual proof, pick an actual term to use in place of t, and substitute it for t throughout the proof.

Let's discuss what we have done here. The axioms of our theory, A1 and A2, are trivial and obvious. Everyone knows that adding zero to something doesn't change it, and also that if two things are equal to a third, then they are equal to each other. In fact, stating the trivial and obvious is a goal to strive for in any axiomatic system. From trivial and obvious truths that everyone agrees upon, we can prove results that are not so obvious yet have absolute faith in them. If we trust the axioms and the rules, we must, by definition, trust the consequences of those axioms and rules, if logic is to mean anything at all.

Our rule of inference, modus ponens, is also pretty obvious once you understand what it means. If we prove a fact P, and we also prove that P implies Q, then Q necessarily follows as a new fact. The rule provides us with a means for obtaining new facts (i.e. theorems) from old ones.

The theorem that we have proved, $t = t$, is so fundamental that you may wonder why it isn't one of the axioms. In some axiom systems of arithmetic, it *is* an axiom. The choice of axioms in a theory is to some extent arbitrary and even an art form, constrained only by the requirement that any two equivalent axiom systems be able to derive each other as theorems. We could imagine that the inventor of our axiom system originally included $t = t$ as an axiom, then discovered that it could be derived as a theorem from the other axioms. Because of this, it was not necessary to keep it as an axiom. By eliminating it, the final set of axioms became that much simpler.

Unless you have worked with formal proofs before, it probably wasn't apparent to you that $t = t$ could be derived from our two axioms until you saw the proof. While you certainly believe that $t = t$ is true, you might not be able to convince an imaginary skeptic who believes only in our two axioms until you produce the proof. Formal proofs such as this are hard to come up with when you first start working with them, but after you get used to them they can become interesting and fun. Once you understand the idea behind formal proofs you will have grasped the fundamental principle that underlies all of mathematics. As the mathematics becomes more sophisticated, its proofs become more challenging, but ultimately they all can be broken down into individual steps as simple as the ones in our proof above.

Mendelson's book, from which our example was taken, contains a number of detailed formal proofs such as these, and you may be interested in looking it up. The book is intended for mathematicians, however, and most of it is rather advanced. Popular literature describing formal proofs include [58, p. 296] and [26, pp. 204–230].

2.2.2 Converting It to Metamath

Formal proofs such as the one in our example break down logical reasoning into small, precise steps that leave little doubt that the results follow from the axioms. You might think that this would be the finest breakdown we can achieve in mathematics. However, there is more to the proof than meets the

eye. Although our axioms were rather simple, a lot of verbiage was needed before we could even state them: we needed to define "term," "wff," and so on. In addition, there are a number of implied rules that we haven't even mentioned. For example, how do we know that step 3 of our proof follows from axiom A1? There is some hidden reasoning involved in determining this. Axiom A1 has two occurrences of the letter t. One of the implied rules states that whatever we substitute for t must be a legal term.[5] The expression $t + 0$ is pretty obviously a legal term whenever t is, but suppose we wanted to substitute a huge term with thousands of symbols? Certainly a lot of work would be involved in determining that it really is a term, but in ordinary formal proofs all of this work would be considered a single "step."

To express our axiom system in the Metamath language, we must describe this auxiliary information in addition to the axioms themselves. Metamath does not know what a "term" or a "wff" is. In Metamath, the specification of the ways in which we can combine symbols to obtain terms and wffs are like little axioms in themselves. These auxiliary axioms are expressed in the same notation as the "real" axioms, and Metamath does not distinguish between the two. The distinction is made by you, i.e. by the way in which you interpret the notation you have chosen to express these two kinds of axioms.

The Metamath language breaks down mathematical proofs into tiny pieces, much more so than in ordinary formal proofs. If a single step involves the substitution of a complex term for one of its variables, Metamath must see this single step broken down into many small steps. This fine-grained breakdown is what gives Metamath generality and flexibility as it lets it not be limited to any particular mathematical notation.

Metamath's proof notation is not, in itself, intended to be read by humans but rather is in a compact format intended for a machine. The Metamath program will convert this notation to a form you can understand, using the `show proof` command. You can tell the program what level of detail of the proof you want to look at. You may want to look at just the logical inference steps that correspond to ordinary formal proof steps, or you may want to see the fine-grained steps that prove that an expression is a term.

Here, without further ado, is our example converted to the Metamath language:

```
$( Declare the constant symbols we will use $)
    $c 0 + = -> ( ) term wff |- $.
$( Declare the metavariables we will use $)
    $v t r s P Q $.
$( Specify properties of the metavariables $)
    tt $f term t $.
    tr $f term r $.
```

[5]Some authors make this implied rule explicit by stating, "only expressions of the above form are terms," after defining "term."

```
    ts $f term s $.
    wp $f wff P $.
    wq $f wff Q $.
$( Define "term" and "wff" $)
    tze $a term 0 $.
    tpl $a term ( t + r ) $.
    weq $a wff t = r $.
    wim $a wff ( P -> Q ) $.
$( State the axioms $)
    a1 $a |- ( t = r -> ( t = s -> r = s ) ) $.
    a2 $a |- ( t + 0 ) = t $.
$( Define the modus ponens inference rule $)
    ${
       min $e |- P $.
       maj $e |- ( P -> Q ) $.
       mp  $a |- Q $.
    $}
$( Prove a theorem $)
    th1 $p |- t = t $=
  $( Here is its proof: $)
       tt tze tpl tt weq tt tt weq tt a2 tt tze tpl
       tt weq tt tze tpl tt weq tt tt weq wim tt a2
       tt tze tpl tt tt a1 mp mp
     $.
```

A "database" is a set of one or more ASCII source files. Here's a brief description of this Metamath database (which consists of this single source file), so that you can understand in general terms what is going on. To understand the source file in detail, you should read Chapter 4.

The database is a sequence of "tokens," which are normally separated by spaces or line breaks. The only tokens that are built into the Metamath language are those beginning with `$`. These tokens are called "keywords." All other tokens are user-defined, and their names are arbitrary.

As you might have guessed, the Metamath token `$(` starts a comment and `$)` ends a comment.

The Metamath tokens `$c`, `$v`, `$e`, `$f`, `$a`, and `$p` specify "statements" that end with `$.`.

The Metamath tokens `$c` and `$v` each declare a list of user-defined tokens, called "math symbols," that the database will reference later on. All of the math symbols they define you have seen earlier except the turnstile symbol `|-` ($\vdash$), which is commonly used by logicians to mean "a proof exists for." For us the turnstile is just a convenient symbol that distinguishes expressions that are axioms or theorems from expressions that are terms or wffs.

The `$c` statement declares "constants" and the `$v` statement declares "variables" (or more precisely, metavariables). A variable may be substituted

with sequences of math symbols whereas a constant may not be substituted with anything.

It may seem redundant to require both `$c` and `$v` statements (since any math symbol not specified with a `$c` statement could be presumed to be a variable), but this provides for better error checking and also allows math symbols to be redeclared (Section 4.2.8).

The token `$f` specifies a statement called a "variable-type hypothesis" (also called a "floating hypothesis") and `$e` specifies a "logical hypothesis" (also called an "essential hypothesis"). The token `$a` specifies an "axiomatic assertion," and `$p` specifies a "provable assertion." To the left of each occurrence of these four tokens is a "label" that identifies the hypothesis or assertion for later reference. For example, the label of the first axiomatic assertion is `tze`. A `$f` statement must contain exactly two math symbols, a constant followed by a variable. The `$e`, `$a`, and `$p` statements each start with a constant followed by, in general, an arbitrary sequence of math symbols.

Associated with each assertion is a set of hypotheses that must be satisfied in order for the assertion to be used in a proof. These are called the "mandatory hypotheses" of the assertion. Among those hypotheses whose "scope" (described below) includes the assertion, `$e` hypotheses are always mandatory and `$f` hypotheses are mandatory when they share their variable with the assertion or its `$e` hypotheses. The exact rules for determining which hypotheses are mandatory are described in detail in Sections 4.2.7 and 4.2.8. For example, the mandatory hypotheses of assertion `tpl` are `tt` and `tr`, whereas assertion `tze` has no mandatory hypotheses because it contains no variables and has no `$e` hypothesis. Metamath's `show statement` command, described in the next section, will show you a statement's mandatory hypotheses.

Sometimes we need to make a hypothesis relevant to only certain assertions. The set of statements to which a hypothesis is relevant is called its "scope." The Metamath brackets, `${` and `$}`, define a "block" that delimits the scope of any hypothesis contained between them. The assertion `mp` has mandatory hypotheses `wp`, `wq`, `min`, and `maj`. The only mandatory hypothesis of `th1`, on the other hand, is `tt`, since `th1` occurs outside of the block containing `min` and `maj`.

Note that `${` and `$}` do not affect the scope of assertions (`$a` and `$p`). Assertions are always available to be referenced by any later proof in the source file.

Each provable assertion (`$p` statement) has two parts. The first part is the assertion itself, which is a sequence of math symbol tokens placed between the `$p` token and a `$=` token. The second part is a "proof," which is a list of label tokens placed between the `$=` token and the `$.` token that ends the statement.[6] The proof acts as a series of instructions to the Metamath

[6] If you've looked at the `set.mm` database, you may have noticed another notation used

program, telling it how to build up the sequence of math symbols contained in the assertion part of the `$p` statement, making use of the hypotheses of the `$p` statement and previous assertions. The construction takes place according to precise rules. If the list of labels in the proof causes these rules to be violated, or if the final sequence that results does not match the assertion, the Metamath program will notify you with an error message.

If you are familiar with reverse Polish notation (RPN), which is sometimes used on pocket calculators, here in a nutshell is how a proof works. Each hypothesis label in the proof is pushed onto the RPN stack as it is encountered. Each assertion label pops off the stack as many entries as the referenced assertion has mandatory hypotheses. Variable substitutions are computed which, when made to the referenced assertion's mandatory hypotheses, cause these hypotheses to match the stack entries. These same substitutions are then made to the variables in the referenced assertion itself, which is then pushed onto the stack. At the end of the proof, there should be one stack entry, namely the assertion being proved. This process is explained in detail in Section 4.3.

Metamath's proof notation is not very readable for humans, but it allows the proof to be stored compactly in a file. The Metamath program has proof display features that let you see what's going on in a more readable way, as you will see in the next section.

The rules used in verifying a proof are not based on any built-in syntax of the symbol sequence in an assertion nor on any built-in meanings attached to specific symbol names. They are based strictly on symbol matching: constants must match themselves, and variables may be replaced with anything that allows a match to occur. For example, instead of `term`, `0`, and `|-` we could have just as well used `yellow`, `zero`, and `provable`, as long as we did so consistently throughout the database. Also, we could have used `is provable` (two tokens) instead of `|-` (one token) throughout the database. In each of these cases, the proof would be exactly the same. The independence of proofs and notation means that you have a lot of flexibility to change the notation you use without having to change any proofs.

2.3 A Trial Run

Now you are ready to try out the Metamath program.

On all computer systems, Metamath has a standard "command line interface" (CLI) that allows you to interact with it. You supply commands to the CLI by typing them on the keyboard and pressing your keyboard's *return* key after each line you enter. The CLI is designed to be easy to use and has built-in help features.

---

for proofs. The other notation is called "compressed." It reduces the amount of space needed to store a proof in the database and is described in Appendix B. In the example above, we use "normal" notation.

The first thing you should do is to use a text editor to create a file called `demo0.mm` and type into it the Metamath source shown on p. 41. Actually, this file is included with your Metamath software package, so check that first. If you type it in, make sure that you save it in the form of "plain ASCII text with line breaks." Most word processors will have this feature.

Next you must run the Metamath program. Depending on your computer system and how Metamath is installed, this could range from clicking the mouse on the Metamath icon to typing `run metamath` to typing simply `metamath`. (Metamath's `help invoke` command describes alternate ways of invoking the Metamath program.)

When you first enter Metamath, it will be at the CLI, waiting for your input. You will see something like the following on your screen:

```
Metamath - Version 0.177 27-Apr-2019
Type HELP for help, EXIT to exit.
MM>
```

The `MM>` prompt means that Metamath is waiting for a command. Command keywords are not case sensitive; we will use lower-case commands in our examples. The version number and its release date will probably be different on your system from the one we show above.

The first thing that you need to do is to read in your database:[7]

```
MM> read demo0.mm
```

Remember to press the *return* key after entering this command. If you omit the file name, Metamath will prompt you for one. The syntax for specifying a Macintosh file name path is given in a footnote on p. 149.

If there are any syntax errors in the database, Metamath will let you know when it reads in the file. The one thing that Metamath does not check when reading in a database is that all proofs are correct, because this would slow it down too much. It is a good idea to periodically verify the proofs in a database you are making changes to. To do this, use the following command (and do it for your `demo0.mm` file now). Note that the `*` is a "wild card" meaning all proofs in the file.

```
MM> verify proof *
```

Metamath will report any proofs that are incorrect.

It is often useful to save the information that the Metamath program displays on the screen. You can save everything that happens on the screen by opening a log file. You may want to do this before you read in a database so that you can examine any errors later on. To open a log file, type

[7] If a directory path is needed on Unix, you should enclose the path/file name in quotes to prevent Metamath from thinking that the `/` in the path name is a command qualifier, e.g., `read "db/set.mm"`. Quotes are optional when there is no ambiguity.

```
MM> open log abc.log
```

This will open a file called `abc.log`, and everything that appears on the screen from this point on will be stored in this file. The name of the log file is arbitrary. To close the log file, type

```
MM> close log
```

Several commands let you examine what's inside your database. Section 3.10 has an overview of some useful ones. The `show labels` command lets you see what statement labels exist. A `*` matches any combination of characters, and `t*` refers to all labels starting with the letter `t`. The `/all` is a "command qualifier" that tells Metamath to include labels of hypotheses. (To see the syntax explained, type `help show labels`.) Type

```
MM> show labels t* /all
```

Metamath will respond with

```
The statement number, label, and type are shown.
3 tt $f       4 tr $f       5 ts $f       8 tze $a
9 tpl $a      19 th1 $p
```

You can use the `show statement` command to get information about a particular statement. For example, you can get information about the statement with label `mp` by typing

```
MM> show statement mp /full
```

Metamath will respond with

```
Statement 17 is located on line 43 of the file
"demo0.mm".
"Define the modus ponens inference rule"
17 mp $a |- Q $.
Its mandatory hypotheses in RPN order are:
  wp $f wff P $.
  wq $f wff Q $.
  min $e |- P $.
  maj $e |- ( P -> Q ) $.
The statement and its hypotheses require the
      variables:  Q P
The variables it contains are:  Q P
```

The mandatory hypotheses and their order are useful to know when you are trying to understand or debug a proof.

Now you are ready to look at what's really inside our proof. First, here is how to look at every step in the proof—not just the ones corresponding to an ordinary formal proof, but also the ones that build up the formulas that appear in each ordinary formal proof step.

```
MM> show proof th1 /lemmon /all
```

This will display the proof on the screen in the following format:

```
 1 tt            $f term t
 2 tze           $a term 0
 3 1,2 tpl       $a term ( t + 0 )
 4 tt            $f term t
 5 3,4 weq       $a wff ( t + 0 ) = t
 6 tt            $f term t
 7 tt            $f term t
 8 6,7 weq       $a wff t = t
 9 tt            $f term t
10 9 a2          $a |- ( t + 0 ) = t
11 tt            $f term t
12 tze           $a term 0
13 11,12 tpl     $a term ( t + 0 )
14 tt            $f term t
15 13,14 weq     $a wff ( t + 0 ) = t
16 tt            $f term t
17 tze           $a term 0
18 16,17 tpl     $a term ( t + 0 )
19 tt            $f term t
20 18,19 weq     $a wff ( t + 0 ) = t
21 tt            $f term t
22 tt            $f term t
23 21,22 weq     $a wff t = t
24 20,23 wim     $a wff ( ( t + 0 ) = t -> t = t )
25 tt            $f term t
26 25 a2         $a |- ( t + 0 ) = t
27 tt            $f term t
28 tze           $a term 0
29 27,28 tpl     $a term ( t + 0 )
30 tt            $f term t
31 tt            $f term t
32 29,30,31 a1   $a |- ( ( t + 0 ) = t -> ( ( t + 0 )
                                        = t -> t = t ) )
33 15,24,26,32 mp  $a |- ( ( t + 0 ) = t -> t = t )
34 5,8,10,33 mp  $a |- t = t
```

The `/lemmon` command qualifier specifies what is known as a Lemmon-style display. Omitting the `/lemmon` qualifier results in a tree-style proof (see p. 138 for an example) that is somewhat less explicit but easier to follow once you get used to it.

The first number on each line is the step number of the proof. Any numbers that follow are step numbers assigned to the hypotheses of the

statement referenced by that step. Next is the label of the statement referenced by the step. The statement type of the statement referenced comes next, followed by the math symbol string constructed by the proof up to that step.

The last step, 34, contains the statement that is being proved.

Looking at a small piece of the proof, notice that steps 3 and 4 have established that `( t + 0 )` and `t` are `term`s, and step 5 makes use of steps 3 and 4 to establish that `( t + 0 ) = t` is a `wff`. Let Metamath itself tell us in detail what is happening in step 5. Note that the "target hypothesis" refers to where step 5 is eventually used, i.e., in step 34.

```
MM> show proof th1 /detailed_step 5
Proof step 5:  wp=weq $a wff ( t + 0 ) = t
This step assigns source "weq" ($a) to target "wp"
($f).  The source assertion requires the hypotheses
"tt" ($f, step 3) and "tr" ($f, step 4).  The parent
assertion of the target hypothesis is "mp" ($a,
step 34).
The source assertion before substitution was:
    weq $a wff t = r
The following substitutions were made to the source
assertion:
    Variable  Substituted with
     t         ( t + 0 )
     r         t
The target hypothesis before substitution was:
    wp $f wff P
The following substitution was made to the target
hypothesis:
    Variable  Substituted with
     P         ( t + 0 ) = t
```

The full proof just shown is useful to understand what is going on in detail. However, most of the time you will just be interested in the "essential" or logical steps of a proof, i.e. those steps that correspond to an ordinary formal proof. If you type

```
MM> show proof th1 /lemmon /renumber
```

you will see

```
1 a2             $a |- ( t + 0 ) = t
2 a2             $a |- ( t + 0 ) = t
3 a1             $a |- ( ( t + 0 ) = t -> ( ( t + 0 )
                                 = t -> t = t ) )
4 2,3 mp         $a |- ( ( t + 0 ) = t -> t = t )
5 1,4 mp         $a |- t = t
```

Compare this to the formal proof on p. 39 and notice the resemblance. By default Metamath does not show `$f` hypotheses and everything branching off of them in the proof tree when the proof is displayed; this makes the proof look more like an ordinary mathematical proof, which does not normally incorporate the explicit construction of expressions. This is called the "essential" view (at one time you had to add the `/essential` qualifier in the `show proof` command to get this view, but this is now the default). You can could use the `/all` qualifier in the `show proof` command to also show the explicit construction of expressions. The `/renumber` qualifier means to renumber the steps to correspond only to what is displayed.

To exit Metamath, type

```
MM> exit
```

2.3.1 Some Hints for Using the Command Line Interface

We will conclude this quick introduction to Metamath with some helpful hints on how to navigate your way through the commands.

When you type commands into Metamath's CLI, you only have to type as many characters of a command keyword as are needed to make it unambiguous. If you type too few characters, Metamath will tell you what the choices are. In the case of the `read` command, only the `r` is needed to specify it unambiguously, so you could have typed

```
MM> r demo0.mm
```

instead of

```
MM> read demo0.mm
```

In our description, we always show the full command words. When using the Metamath CLI commands in a command file (to be read with the `submit` command), it is good practice to use the unabbreviated command to ensure your instructions will not become ambiguous if more commands are added to the Metamath program in the future.

The command keywords are not case sensitive; you may type either `read` or `ReAd`. File names may or may not be case sensitive, depending on your computer's operating system. Metamath label and math symbol tokens are case-sensitive.

The `help` command will provide you with a list of topics you can get help on. You can then type `help` *topic* to get help on that topic.

If you are uncertain of a command's spelling, just type as many characters as you remember of the command. If you have not typed enough characters to specify it unambiguously, Metamath will tell you what choices you have.

```
MM> show s
         ^
```

```
?Ambiguous keyword - please specify SETTINGS,
STATEMENT, or SOURCE.
```

If you don't know what argument to use as part of a command, type a `?` at the argument position. Metamath will tell you what it expected there.

```
MM> show ?
         ^
?Expected SETTINGS, LABELS, STATEMENT, SOURCE, PROOF,
MEMORY, TRACE_BACK, or USAGE.
```

Finally, you may type just the first word or words of a command followed by *return*. Metamath will prompt you for the remaining part of the command, showing you the choices at each step. For example, instead of typing `show statement th1 /full` you could interact in the following manner:

```
MM> show
SETTINGS, LABELS, STATEMENT, SOURCE, PROOF,
MEMORY, TRACE_BACK, or USAGE <SETTINGS>? st
What is the statement label <th1>?
/ or nothing <nothing>? /
TEX, COMMENT_ONLY, or FULL <TEX>? f
/ or nothing <nothing>?
19 th1 $p |- t = t $= ... $.
```

After each `?` in this mode, you must give Metamath the information it requests. Sometimes Metamath gives you a list of choices with the default choice indicated by brackets `< >` . Pressing *return* after the `?` will select the default choice. Answering anything else will override the default. Note that the `/` in command qualifiers is considered a separate token by the parser, and this is why it is asked for separately.

2.4 Your First Proof

Proofs are developed with the aid of the Proof Assistant. We will now show you how the proof of theorem `th1` was built. So that you can repeat these steps, we will first have the Proof Assistant erase the proof in Metamath's source buffer, then reconstruct it. (The source buffer is the place in memory where Metamath stores the information in the database when it is `read` in. New or modified proofs are kept in the source buffer until a `write source` command is issued.) In practice, you would place a `?` between `$=` and `$.` in the database to indicate to Metamath that the proof is unknown, and that would be your starting point. Whenever the `verify proof` command encounters a proof with a `?` in place of a proof step, the statement is identified as not proved.

When I first started creating Metamath proofs, I would write down on a piece of paper the complete formal proof as it would appear in a `show proof` command; see the display of `show proof th1 /lemmon /renumber` above as an example. After you get used to using the Proof Assistant you may get to a point where you can "see" the proof in your mind and let the Proof Assistant guide you in filling in the details, at least for simpler proofs, but until you gain that experience you may find it very useful to write down all the details in advance. Otherwise you may waste a lot of time as you let it take you down a wrong path. However, others do not find this approach as helpful. For example, Thomas Brendan Leahy finds that it is more helpful to him to interactively work backward from a machine-readable statement. David A. Wheeler writes down a general approach, but develops the proof interactively by switching between working forwards (from hypotheses and facts likely to be useful) and backwards (from the goal) until the forwards and backwards approaches meet. In the end, use whatever approach works for you.

A proof is developed with the Proof Assistant by working backwards, starting with the theorem to be proved, and assigning each unknown step with a theorem or hypothesis until no more unknown steps remain. The Proof Assistant will not let you make an assignment unless it can be "unified" with the unknown step. This means that a substitution of variables exists that will make the assignment match the unknown step. On the other hand, in the middle of a proof, when working backwards, often more than one unification (set of substitutions) is possible, since there is not enough information available at that point to uniquely establish it. In this case you can tell Metamath which unification to choose, or you can continue to assign unknown steps until enough information is available to make the unification unique.

We will assume you have entered Metamath and read in the database as described above. The following dialog shows how the proof was developed. For more details on what some of the commands do, refer to Section 5.6.

```
MM> prove th1
Entering the Proof Assistant.  Type HELP for help, EXIT
to exit.  You will be working on the proof of statement th1:
  $p |- t = t
Note:  The proof you are starting with is already complete.
MM-PA>
```

The `MM-PA>` prompt means we are inside the Proof Assistant. Most of the regular Metamath commands (`show statement`, etc.) are still available if you need them.

```
MM-PA> delete all
The entire proof was deleted.
```

We have deleted the whole proof so we can start from scratch.

```
MM-PA> show new_proof/lemmon/all
1 ?              $? |- t = t
```

The `show new_proof` command is like `show proof` except that we don't specify a statement; instead, the proof we're working on is displayed.

```
MM-PA> assign 1 mp
To undo the assignment, DELETE STEP 5 and INITIALIZE, UNIFY
if needed.
3   min=?  $? |- $2
4   maj=?  $? |- ( $2 -> t = t )
```

The `assign` command above means "assign step 1 with the statement whose label is `mp`." Note that step renumbering will constantly occur as you assign steps in the middle of a proof; in general all steps from the step you assign until the end of the proof will get moved up. In this case, what used to be step 1 is now step 5, because the (partial) proof now has five steps: the four hypotheses of the `mp` statement and the `mp` statement itself. Let's look at all the steps in our partial proof:

```
MM-PA> show new_proof/lemmon/all
1 ?              $? wff $2
2 ?              $? wff t = t
3 ?              $? |- $2
4 ?              $? |- ( $2 -> t = t )
5 1,2,3,4 mp     $a |- t = t
```

The symbol `$2` is a temporary variable that represents a symbol sequence not yet known. In the final proof, all temporary variables will be eliminated. The general format for a temporary variable is `$` followed by an integer. Note that `$` is not a legal character in a math symbol (see Section 4.2.1, p. 117), so there will never be a naming conflict between real symbols and temporary variables.

Unknown steps 1 and 2 are constructions of the two wffs used by the modus ponens rule. As you will see at the end of this section, the Proof Assistant can usually figure these steps out by itself, and we will not have to worry about them. Therefore from here on we will display only the "essential" hypotheses, i.e. those steps that correspond to traditional formal proofs.

```
MM-PA> show new_proof/lemmon
3 ?              $? |- $2
4 ?              $? |- ( $2 -> t = t )
5 3,4 mp         $a |- t = t
```

Unknown steps 3 and 4 are the ones we must focus on. They correspond to the minor and major premises of the modus ponens rule. We will assign them as follows. Notice that because of the step renumbering that takes place after an assignment, it is advantageous to assign unknown steps in reverse order, because earlier steps will not get renumbered.

```
MM-PA> assign 4 mp
To undo the assignment, DELETE STEP 8 and INITIALIZE, UNIFY
if needed.
3   min=?  $? |- $2
6     min=?  $? |- $4
7     maj=?  $? |- ( $4 -> ( $2 -> t = t ) )
```

We are now going to describe an obscure feature that you will probably never use but should be aware of. The Metamath language allows empty symbol sequences to be substituted for variables, but in most formal systems this feature is never used. One of the few examples where is it used is the MIU-system described in Appendix D. But such systems are rare, and by default this feature is turned off in the Proof Assistant. (It is always allowed for `verify proof`.) Let us turn it on and see what happens.

```
MM-PA> set empty_substitution on
Substitutions with empty symbol sequences is now allowed.
```

With this feature enabled, more unifications will be ambiguous in the middle of a proof, because substitution of variables with empty symbol sequences will become an additional possibility. Let's see what happens when we make our next assignment.

```
MM-PA> assign 3 a2
There are 2 possible unifications.  Please select the correct
    one or Q if you want to UNIFY later.
Unify:  |- $6
 with:  |- ( $9 + 0 ) = $9
Unification #1 of 2 (weight = 7):
  Replace "$6" with "( + 0 ) ="
  Replace "$9" with ""
  Accept (A), reject (R), or quit (Q) <A>? r
```

The first choice presented is the wrong one. If we had selected it, temporary variable `$6` would have been assigned a truncated wff, and temporary variable `$9` would have been assigned an empty sequence (which is not allowed in our system). With this choice, eventually we would reach a point where we would get stuck because we would end up with steps impossible to prove. (You may want to try it.) We typed `r` to reject the choice.

```
Unification #2 of 2 (weight = 21):
  Replace "$6" with "( $9 + 0 ) = $9"
  Accept (A), reject (R), or quit (Q) <A>? q
To undo the assignment, DELETE STEP 4 and INITIALIZE, UNIFY
if needed.
 7     min=?  $? |- $8
 8     maj=?  $? |- ( $8 -> ( $6 -> t = t ) )
```

The second choice is correct, and normally we would type `a` to accept it. But instead we typed `q` to show what will happen: it will leave the step with an unknown unification, which can be seen as follows:

```
MM-PA> show new_proof/not_unified
 4   min    $a |- $6
        =a2  = |- ( $9 + 0 ) = $9
```

Later we can unify this with the `unify all/interactive` command.

The important point to remember is that occasionally you will be presented with several unification choices while entering a proof, when the program determines that there is not enough information yet to make an unambiguous choice automatically (and this can happen even with `set empty_substitution` turned off). Usually it is obvious by inspection which choice is correct, since incorrect ones will tend to be meaningless fragments of wffs. In addition, the correct choice will usually be the first one presented, unlike our example above.

Enough of this digression. Let us go back to the default setting.

```
MM-PA> set empty_substitution off
The ability to substitute empty expressions for variables
has been turned off.  Note that this may make the Proof
Assistant too restrictive in some cases.
```

If we delete the proof, start over, and get to the point where we digressed above, there will no longer be an ambiguous unification.

```
MM-PA> assign 3 a2
To undo the assignment, DELETE STEP 4 and INITIALIZE, UNIFY
if needed.
 7     min=?  $? |- $4
 8     maj=?  $? |- ( $4 -> ( ( $5 + 0 ) = $5 -> t = t ) )
```

Let us look at our proof so far, and continue.

```
MM-PA> show new_proof/lemmon
 4 a2            $a |- ( $5 + 0 ) = $5
 7 ?             $? |- $4
```

```
 8 ?            $? |- ( $4 -> ( ( $5 + 0 ) = $5 -> t = t ) )
 9 7,8 mp       $a |- ( ( $5 + 0 ) = $5 -> t = t )
10 4,9 mp       $a |- t = t
MM-PA> assign 8 a1
To undo the assignment, DELETE STEP 11 and INITIALIZE, UNIFY
if needed.
 7     min=?  $? |- ( t + 0 ) = t
MM-PA> assign 7 a2
To undo the assignment, DELETE STEP 8 and INITIALIZE, UNIFY
if needed.
MM-PA> show new_proof/lemmon
 4 a2           $a |- ( t + 0 ) = t
 8 a2           $a |- ( t + 0 ) = t
12 a1           $a |- ( ( t + 0 ) = t -> ( ( t + 0 ) = t ->
                                                t = t ) )
13 8,12 mp      $a |- ( ( t + 0 ) = t -> t = t )
14 4,13 mp      $a |- t = t
```

Now all temporary variables and unknown steps have been eliminated from the "essential" part of the proof. When this is achieved, the Proof Assistant can usually figure out the rest of the proof automatically. (Note that the `improve` command can occasionally be useful for filling in essential steps as well, but it only tries to make use of statements that introduce no new variables in their hypotheses, which is not the case for `mp`. Also it will not try to improve steps containing temporary variables.) Let's look at the complete proof, then run the `improve` command, then look at it again.

```
MM-PA> show new_proof/lemmon/all
 1 ?            $? wff ( t + 0 ) = t
 2 ?            $? wff t = t
 3 ?            $? term t
 4 3 a2         $a |- ( t + 0 ) = t
 5 ?            $? wff ( t + 0 ) = t
 6 ?            $? wff ( ( t + 0 ) = t -> t = t )
 7 ?            $? term t
 8 7 a2         $a |- ( t + 0 ) = t
 9 ?            $? term ( t + 0 )
10 ?            $? term t
11 ?            $? term t
12 9,10,11 a1   $a |- ( ( t + 0 ) = t -> ( ( t + 0 ) = t ->
                                                t = t ) )
13 5,6,8,12 mp  $a |- ( ( t + 0 ) = t -> t = t )
14 1,2,4,13 mp  $a |- t = t

MM-PA> improve all
```

```
A proof of length 1 was found for step 11.
A proof of length 1 was found for step 10.
A proof of length 3 was found for step 9.
A proof of length 1 was found for step 7.
A proof of length 9 was found for step 6.
A proof of length 5 was found for step 5.
A proof of length 1 was found for step 3.
A proof of length 3 was found for step 2.
A proof of length 5 was found for step 1.
Steps 1 and above have been renumbered.
CONGRATULATIONS!  The proof is complete.  Use SAVE
NEW_PROOF to save it.  Note:  The Proof Assistant does
not detect $d violations.  After saving the proof, you
should verify it with VERIFY PROOF.
```

The `save new_proof` command will save the proof in the database. Here we will just display it in a form that can be clipped out of a log file and inserted manually into the database source file with a text editor.

```
MM-PA> show new_proof/normal
---------Clip out the proof below this line:
      tt tze tpl tt weq tt tt weq tt a2 tt tze tpl tt weq
      tt tze tpl tt weq tt tt weq wim tt a2 tt tze tpl tt
      tt a1 mp mp $.
---------The proof of 'th1' to clip out ends above this line.
```

There is another proof format called "compressed" that you will see in databases. It is not important to understand how it is encoded but only to recognize it when you see it. Its only purpose is to reduce storage requirements for large proofs. A compressed proof can always be converted to a normal one and vice-versa, and the Metamath `show proof` commands work equally well with compressed proofs. The compressed proof format is described in Appendix B.

```
MM-PA> show new_proof/compressed
---------Clip out the proof below this line:
      ( tze tpl weq a2 wim a1 mp ) ABCZADZAADZAEZJJKFLIA
      AGHH $.
---------The proof of 'th1' to clip out ends above this line.
```

Now we will exit the Proof Assistant. Since we made changes to the proof, it will warn us that we have not saved it. In this case, we don't care.

```
MM-PA> exit
Warning:  You have not saved changes to the proof.
Do you want to EXIT anyway (Y, N) <N>? y
```

```
Exiting the Proof Assistant.
Type EXIT again to exit Metamath.
```

The Proof Assistant has several other commands that can help you while creating proofs. See Section 5.6 for a list of them.

A command that is often useful is `minimize_with */brief`, which tries to shorten the proof. It can make the process more efficient by letting you write a somewhat "sloppy" proof then clean up some of the fine details of optimization for you (although it can't perform miracles such as restructuring the overall proof).

2.5 A Note About Editing a Database File

Once your source file contains proofs, there are some restrictions on how you can edit it so that the proofs remain valid. Pay particular attention to these rules, since otherwise you can lose a lot of work. It is a good idea to periodically verify all proofs with `verify proof *` to ensure their integrity.

If your file contains only normal (as opposed to compressed) proofs, the main rule is that you may not change the order of the mandatory hypotheses of any statement referenced in a later proof. For example, if you swap the order of the major and minor premise in the modus ponens rule, all proofs making use of that rule will become incorrect. The `show statement` command will show you the mandatory hypotheses of a statement and their order.

If a statement has a compressed proof, you also must not change the order of *its* mandatory hypotheses. The compressed proof format makes use of this information as part of the compression technique. Note that swapping the names of two variables in a theorem will change the order of its mandatory hypotheses.

The safest way to edit a statement, say `mytheorem`, is to duplicate it then rename the original to `mytheoremOLD` throughout the database. Once the edited version is re-proved, all statements referencing `mytheoremOLD` can be updated in the Proof Assistant using `minimize_with mytheorem /allow_growth`.

Chapter 3

Abstract Mathematics Revealed

3.1 Logic and Set Theory

Set theory can be viewed as a form of exact theology.

Rudy Rucker[1]

Despite its seeming complexity, all of standard mathematics, no matter how deep or abstract, can amazingly enough be derived from a relatively small set of axioms or first principles. The development of these axioms is among the most impressive and important accomplishments of mathematics in the 20th century. Ultimately, these axioms can be broken down into a set of rules for manipulating symbols that any technically oriented person can follow.

We will not spend much time trying to convey a deep, higher-level understanding of the meaning of the axioms. This kind of understanding requires some mathematical sophistication as well as an understanding of the philosophy underlying the foundations of mathematics and typically develops over time as you work with mathematics. Our goal, instead, is to give you the immediate ability to follow how theorems are derived from the axioms and from other theorems. This will be similar to learning the syntax of a computer language, which lets you follow the details in a program but does not necessarily give you the ability to write non-trivial programs on your own, an ability that comes with practice. For now don't be alarmed by abstract-sounding names of the axioms; just focus on the rules for manipulating the symbols, which follow the simple conventions of the Metamath language.

[1] [3], p. 31.

The axioms that underlie all of standard mathematics consist of axioms of logic and axioms of set theory. The axioms of logic are divided into two subcategories, propositional calculus (sometimes called sentential logic) and predicate calculus (sometimes called first-order logic or quantifier theory). Propositional calculus is a prerequisite for predicate calculus, and predicate calculus is a prerequisite for set theory. The version of set theory most commonly used is Zermelo–Fraenkel set theory with the axiom of choice, often abbreviated as ZFC.

Here in a nutshell is what the axioms are all about in an informal way. The connection between this description and symbols we will show you won't be immediately apparent and in principle needn't ever be. Our description just tries to summarize what mathematicians think about when they work with the axioms.

Logic is a set of rules that allow us determine truths given other truths. Put another way, logic is more or less the translation of what we would consider common sense into a rigorous set of axioms. Suppose φ, ψ, and χ (the Greek letters phi, psi, and chi) represent statements that are either true or false, and x is a variable ranging over some group of mathematical objects (sets, integers, real numbers, etc.). In mathematics, a "statement" really means a formula, and ψ could be for example "$x = 2$." Propositional calculus allows us to use variables that are either true or false and make deductions such as "if φ implies ψ and ψ implies χ, then φ implies χ." Predicate calculus extends propositional calculus by also allowing us to discuss statements about objects (not just true and false values), including statements about "all" or "at least one" object. For example, predicate calculus allows to say, "if φ is true for all x, then φ is true for some x." The logic used in `set.mm` is standard classical logic (as opposed to other logic systems like intuitionistic logic).

Set theory has to do with the manipulation of objects and collections of objects, specifically the abstract, imaginary objects that mathematics deals with, such as numbers. Everything that is claimed to exist in mathematics is considered to be a set. A set called the empty set contains nothing. We represent the empty set by $\varnothing$. Many sets can be built up from the empty set. There is a set represented by $\{\varnothing\}$ that contains the empty set, another set represented by $\{\varnothing, \{\varnothing\}\}$ that contains this set as well as the empty set, another set represented by $\{\{\varnothing\}\}$ that contains just the set that contains the empty set, and so on ad infinitum. All mathematical objects, no matter how complex, are defined as being identical to certain sets: the integer 0 is defined as the empty set, the integer 1 is defined as $\{\varnothing\}$, the integer 2 is defined as $\{\varnothing, \{\varnothing\}\}$. (How these definitions were chosen doesn't matter now, but the idea behind it is that these sets have the properties we expect of integers once suitable operations are defined.) Mathematical operations, such as addition, are defined in terms of operations on sets—their union, intersection, and so on—operations you may have used in elementary school

when you worked with groups of apples and oranges.

With a leap of faith, the axioms also postulate the existence of infinite sets, such as the set of all non-negative integers $(0, 1, 2, \ldots$, also called "natural numbers"). This set can't be represented with the brace notation we just showed you, but requires a more complicated notation called "class abstraction." For example, the infinite set $\{x|$"x is a natural number"$\}$ means the "set of all objects x such that x is a natural number" i.e. the set of natural numbers; here, "x is a natural number" is a rather complicated formula when broken down into the primitive symbols.[2] Actually, the primitive symbols don't even include the brace notation. The brace notation is a high-level definition, which you can find in Section 3.4.

Interestingly, the arithmetic of integers and rationals can be developed without appealing to the existence of an infinite set, whereas the arithmetic of real numbers requires it.

Each variable in the axioms of set theory represents an arbitrary set, and the axioms specify the legal kinds of things you can do with these variables at a very primitive level.

Now, you may think that numbers and arithmetic are a lot more intuitive and fundamental than sets and therefore should be the foundation of mathematics. What is really the case is that you've dealt with numbers all your life and are comfortable with a few rules for manipulating them such as addition and multiplication. Those rules only cover a small portion of what can be done with numbers and only a very tiny fraction of the rest of mathematics. If you look at any elementary book on number theory, you will quickly become lost if these are the only rules that you know. Even though such books may present a list of "axioms" for arithmetic, the ability to use the axioms and to understand proofs of theorems (facts) about numbers requires an implicit mathematical talent that frustrates many people from studying abstract mathematics. The kind of mathematics that most people know limits them to the practical, everyday usage of blindly manipulating numbers and formulas, without any understanding of why those rules are correct nor any ability to go any further. For example, do you know why multiplying two negative numbers yields a positive number? Starting with set theory, you will also start off blindly manipulating symbols according to

[2]The statement "x is a natural number" is formally expressed as "$x \in \omega$," where $\in$ (stylized epsilon) means "is in" or "is an element of" and ω (omega) means "the set of natural numbers." When "$x \in \omega$" is completely expanded in terms of the primitive symbols of set theory, the result is $\neg$ ($\neg$ ($\forall$ z ($\neg$ $\forall$ w ($z \in w \rightarrow \neg$ $w \in x$) $\rightarrow z \in x$) $\rightarrow$ ($\forall$ z ($\neg$ ($\forall$ w ($w \in z \rightarrow w \in x$) $\rightarrow \forall$ $w \neg$ $w \in z$) $\rightarrow \neg \forall$ w ($w \in z \rightarrow \neg \forall$ v ($v \in z \rightarrow \neg$ $v \in w$))) $\rightarrow \neg \forall$ $z \forall$ w ($\neg$ ($z \in x \rightarrow \neg$ $w \in x$) $\rightarrow$ ($\neg$ $z \in w \rightarrow$ ($\neg$ $z = w \rightarrow$ $w \in z$))))) $\rightarrow \neg \forall$ y ($\neg$ ($\neg$ ($\forall$ z ($\neg \forall$ w ($z \in w \rightarrow \neg$ $w \in y$) $\rightarrow z \in y$) $\rightarrow$ ($\forall$ z ($\neg$ ($\forall$ w ($w \in z \rightarrow w \in y$) $\rightarrow \forall$ $w \neg$ $w \in z$) $\rightarrow \neg \forall$ w ($w \in z \rightarrow \neg \forall$ v ($v \in z \rightarrow \neg$ $v \in w$))) $\rightarrow \neg \forall$ $z \forall$ w ($\neg$ ($z \in y \rightarrow \neg$ $w \in y$) $\rightarrow$ ($\neg$ $z \in w \rightarrow$ ($\neg$ $z = w \rightarrow$ $w \in z$)))) $\rightarrow$ ($\forall$ $z \neg$ $z \in y \rightarrow \neg \forall$ w ($\neg$ ($w \in y \rightarrow \neg \forall$ z ($w \in z \rightarrow \neg$ $z \in y$)) $\rightarrow \neg$ ($\neg \forall$ z ($w \in z \rightarrow \neg$ $z \in y$) $\rightarrow$ $w \in y$)))) $\rightarrow$ $x \in y$))). Section 3.4 shows the hierarchy of definitions that leads up to this expression.

the rules we give you, but with the advantage that these rules will allow you, in principle, to access *all* of mathematics, not just a tiny part of it.

Of course, concrete examples are often helpful in the learning process. For example, you can verify that $2 \cdot 3 = 3 \cdot 2$ by actually grouping objects and can easily "see" how it generalizes to $x \cdot y = y \cdot x$, even though you might not be able to rigorously prove it. Similarly, in set theory it can be helpful to understand how the axioms of set theory apply to (and are correct for) small finite collections of objects. You should be aware that in set theory intuition can be misleading for infinite collections, and rigorous proofs become more important. For example, while $x \cdot y = y \cdot x$ is correct for finite ordinals (which are the natural numbers), it is not usually true for infinite ordinals.

3.2 The Axioms for All of Mathematics

In this section, we will show you the axioms for all of standard mathematics (i.e. logic and set theory) as they are traditionally presented. The traditional presentation is useful for someone with the mathematical experience needed to correctly manipulate high-level abstract concepts. For someone without this talent, knowing how to actually make use of these axioms can be difficult. The purpose of this section is to allow you to see how the version of the axioms used in the standard Metamath database `set.mm` relates to the typical version in textbooks, and also to give you an informal feel for them.

3.2.1 Propositional Calculus

Propositional calculus concerns itself with statements that can be interpreted as either true or false. Some examples of statements (outside of mathematics) that are either true or false are "It is raining today" and "The United States has a female president." In mathematics, as we mentioned, statements are really formulas.

In propositional calculus, we don't care what the statements are. We also treat a logical combination of statements, such as "It is raining today and the United States has a female president," no differently from a single statement. Statements and their combinations are called well-formed formulas (wffs). We define wffs only in terms of other wffs and don't define what a "starting" wff is. As is common practice in the literature, we use Greek letters to represent wffs.

Specifically, suppose φ and ψ are wffs. Then the combinations $\varphi \to \psi$ ("φ implies ψ," also read "if φ then ψ") and $\neg\varphi$ ("not φ") are also wffs.

The three axioms of propositional calculus are all wffs of the following form:[3]

[3] A remarkable result of C. A. Meredith squeezes these three axioms into the single axiom $((((\varphi \to \psi) \to (\neg\chi \to \neg\theta)) \to \chi) \to \tau) \to ((\tau \to \varphi) \to (\theta \to \varphi))$ [44], which is believed to be the shortest possible.

$$\varphi \to (\psi \to \varphi)$$
$$(\varphi \to (\psi \to \chi)) \to ((\varphi \to \psi) \to (\varphi \to \chi))$$
$$(\neg\varphi \to \neg\psi) \to (\psi \to \varphi)$$

These three axioms are widely used. They are attributed to Jan Łukasiewicz (pronounced woo-kah-SHAY-vitch) and was popularized by Alonzo Church, who called it system P2. (Thanks to Ted Ulrich for this information.)

There are an infinite number of axioms, one for each possible wff of the above form. (For this reason, axioms such as the above are often called "axiom schemes.") Each Greek letter in the axioms may be substituted with a more complex wff to result in another axiom. For example, substituting $\neg(\varphi \to \chi)$ for φ in the first axiom yields $\neg(\varphi \to \chi) \to (\psi \to \neg(\varphi \to \chi))$, which is still an axiom.

To deduce new true statements (theorems) from the axioms, a rule called "modus ponens" is used. This rule states that if the wff φ is an axiom or a theorem, and the wff $\varphi \to \psi$ is an axiom or a theorem, then the wff ψ is also a theorem.

As a non-mathematical example of modus ponens, suppose we have proved (or taken as an axiom) "Bob is a man" and separately have proved (or taken as an axiom) "If Bob is a man, then Bob is a human." Using the rule of modus ponens, we can logically deduce, "Bob is a human."

From Metamath's point of view, the axioms and the rule of modus ponens just define a mechanical means for deducing new true statements from existing true statements, and that is the complete content of propositional calculus as far as Metamath is concerned. You can read a logic textbook to gain a better understanding of their meaning, or you can just let their meaning slowly become apparent to you after you use them for a while.

It is actually rather easy to check to see if a formula is a theorem of propositional calculus. Theorems of propositional calculus are also called "tautologies." The technique to check whether a formula is a tautology is called the "truth table method," and it works like this. A wff $\varphi \to \psi$ is false whenever φ is true and ψ is false. Otherwise it is true. A wff $\neg\varphi$ is false whenever φ is true and false otherwise. To verify a tautology such as $\varphi \to (\psi \to \varphi)$, you break it down into sub-wffs and construct a truth table that accounts for all possible combinations of true and false assigned to the wff metavariables:

φ	ψ	$\psi \to \varphi$	$\varphi \to (\psi \to \varphi)$
T	T	T	T
T	F	T	T
F	T	F	T
F	F	T	T

If all entries in the last column are true, the formula is a tautology.

Now, the truth table method doesn't tell you how to prove the tautology from the axioms, but only that a proof exists. Finding an actual proof

(especially one that is short and elegant) can be challenging. Methods do exist for automatically generating proofs in propositional calculus, but the proofs that result can sometimes be very long. In the Metamath `set.mm` database, most or all proofs were created manually.

Section 3.4.1 discusses various definitions that make propositional calculus easier to use. For example, we define:

- $\varphi \vee \psi$ is true if either φ or ψ (or both) are true (this is disjunction aka logical OR).
- $\varphi \wedge \psi$ is true if both φ and ψ are true (this is conjunction aka logical AND).
- $\varphi \leftrightarrow \psi$ is true if φ and ψ have the same value, that is, they are both true or both false (this is the biconditional).

3.2.2 Predicate Calculus

Predicate calculus introduces the concept of "individual variables," which we will usually just call "variables." These variables can represent something other than true or false (wffs), and will always represent sets when we get to set theory. There are also three new symbols $\forall$, $=$, and $\in$, read "for all," "equals," and "is an element of" respectively. We will represent variables with the letters x, y, z, and w, as is common practice in the literature. For example, $\forall x \varphi$ means "for all possible values of x, φ is true."

In predicate calculus, we extend the definition of a wff. If φ is a wff and x and y are variables, then $\forall x\, \varphi$, $x = y$, and $x \in y$ are wffs. Note that these three new types of wffs can be considered "starting" wffs from which we can build other wffs with $\rightarrow$ and $\neg$. The concept of a starting wff was absent in propositional calculus. But starting wff or not, all we are really concerned with is whether our wffs are correctly constructed according to these mechanical rules.

A quick aside: To prevent confusion, it might be best at this point to think of the variables of Metamath as "metavariables," because they are not quite the same as the variables we are introducing here. A (meta)variable in Metamath can be a wff or an individual variable, as well as many other things; in general, it represents a kind of place holder for an unspecified sequence of math symbols.

Unlike propositional calculus, no decision procedure analogous to the truth table method exists (nor theoretically can exist) that will definitely determine whether a formula is a theorem of predicate calculus. Much of the work in the field of automated theorem proving has been dedicated to coming up with clever heuristics for proving theorems of predicate calculus, but they can never be guaranteed to work always.

Section 3.4.2 discusses various definitions that make predicate calculus easier to use. For example, we define $\exists x \varphi$ to mean "there exists at least one possible value of x where φ is true."

We now turn to looking at how predicate calculus can be formally represented.

Common Axioms

There is a new rule of inference in predicate calculus: if φ is an axiom or a theorem, then $\forall x\, \varphi$ is also a theorem. This is called the rule of "generalization." This is easily represented in Metamath.

In standard texts of logic, there are often two axioms of predicate calculus:

$\forall x\, \varphi(x) \rightarrow \varphi(y)$, where "$y$ is properly substituted for x."
$\forall x(\varphi \rightarrow \psi) \rightarrow (\varphi \rightarrow \forall x\, \psi)$, where "$x$ is not free in φ."

Now at first glance, this seems simple: just two axioms. However, conditional clauses are attached to each axiom describing requirements that may seem puzzling to you. In addition, the first axiom puts a variable symbol in parentheses after each wff, seemingly violating our definition of a wff; this is just an informal way of referring to some arbitrary variable that may occur in the wff. The conditional clauses do, of course, have a precise meaning, but as it turns out the precise meaning is somewhat complicated and awkward to formalize in a way that a computer can handle easily. Unlike propositional calculus, a certain amount of mathematical sophistication and practice is needed to be able to easily grasp and manipulate these concepts correctly.

Predicate calculus may be presented with or without axioms for equality. We will require the axioms of equality as a prerequisite for the version of set theory we will use. The axioms for equality, when included, are often represented using these two axioms:

$$x = x$$

$x = y \rightarrow (\varphi(x, x) \rightarrow \varphi(x, y))$ where "$\varphi(x, y)$ arises from $\varphi(x, x)$ by replacing some, but not necessarily all, free occurrences of x by y, provided that y is free for x in $\varphi(x, x)$."

The first equality axiom is simple, but again, the condition on the second one is somewhat awkward to implement on a computer.

Tarski System S2

Of course, we are not the first to notice the complications of these predicate calculus axioms when being rigorous.

Well-known logician Alfred Tarski published in 1965 a system he called system S2[69, p. 77]. Tarski's system is *exactly equivalent* to the traditional textbook formalization, but (by clever use of equality axioms) it eliminates

the latter's primitive notions of "proper substitution" and "free variable," replacing them with direct substitution and the notion of a variable not occurring in a formula (which we express with distinct variable constraints).

In advocating his system, Tarski wrote, "The relatively complicated character of [free variables and proper substitution] is a source of certain inconveniences of both practical and theoretical nature; this is clearly experienced both in teaching an elementary course of mathematical logic and in formalizing the syntax of predicate logic for some theoretical purposes"[69, p. 61].

Developing a Metamath Representation

The standard textbook axioms of predicate calculus are somewhat cumbersome to implement on a computer because of the complex notions of "free variable" and "proper substitution." While it is possible to use the Metamath language to implement these concepts, we have chosen not to implement them as primitive constructs in the `set.mm` set theory database. Instead, we have eliminated them within the axioms by carefully crafting the axioms so as to avoid them, building on Tarski's system S2. This makes it easy for a beginner to follow the steps in a proof without knowing any advanced concepts other than the simple concept of replacing variables with expressions.

In order to develop the concepts of free variable and proper substitution from the axioms, we use an additional Metamath statement type called "disjoint variable restriction" that we have not encountered before. In the context of the axioms, the statement `$d` $x\,y$ simply means that x and y must be distinct, i.e. they may not be simultaneously substituted with the same variable. The statement `$d` $x\,\varphi$ means variable x must not occur in wff φ. For the precise definition of `$d`, see Section 4.2.4.

Metamath representation

The Metamath axiom system for predicate calculus defined in set.mm uses Tarski's system S2. As noted above, this has a different representation than the traditional textbook formalization, but it is *exactly equivalent* to the textbook formalization, and it is *much* easier to work with. This is reproduced as system S3 in Section 6 of Megill's formalization [41].

There is one exception, Tarski's axiom of existence, which we label as axiom ax-6. In the case of ax-6, Tarski's version is weaker because it includes a distinct variable proviso. If we wish, we can also weaken our version in this way and still have a metalogically complete system. Theorem ax6 shows this by deriving, in the presence of the other axioms, our ax-6 from Tarski's weaker version ax6v. However, we chose the stronger version for our system because it is simpler to state and easier to use.

Tarski's system was designed for proving specific theorems rather than more general theorem schemes. However, theorem schemes are much more efficient than specific theorems for building a body of mathematical knowledge, since they can be reused with different instances as needed. While Tarski does derive some theorem schemes from his axioms, their proofs require concepts that are "outside" of the system, such as induction on formula length. The verification of such proofs is difficult to automate in a proof verifier. (Specifically, Tarski treats the formulas of his system as set-theoretical objects. In order to verify the proofs of his theorem schemes, a proof verifier would need a significant amount of set theory built into it.)

The Metamath axiom system for predicate calculus extends Tarski's system to eliminate this difficulty. The additional "auxilliary" axiom schemes (as we will call them in this section; see below) endow Tarski's system with a nice property we call metalogical completeness [41, Remark 9.6]. As a result, we can prove any theorem scheme expressable in the "simple metalogic" of Tarski's system by using only Metamath's direct substitution rule applied to the axiom system (and no other metalogical or set-theoretical notions "outside" of the system). Simple metalogic consists of schemes containing wff metavariables (with no arguments) and/or set (also called "individual") metavariables, accompanied by optional provisos each stating that two specified set metavariables must be distinct or that a specified set metavariable may not occur in a specified wff metavariable. Metamath's logic and set theory axiom and rule schemes are all examples of simple metalogic. The schemes of traditional predicate calculus with equality are examples which are not simple metalogic, because they use wff metavariables with arguments and have "free for" and "not free in" side conditions.

A rigorous justification for this system, using an older but exactly equivalent set of axioms, can be found in [41].

This allows us to take a different approach in the Metamath database `set.mm`. We do not directly use the primitive notions of "free variable" and "proper substitution" at all as primitive constructs. Instead, we use a set of axioms that are almost as simple to manipulate as those of propositional calculus. Our axiom system avoids complex primitive notions by effectively embedding the complexity into the axioms themselves. As a result, we will end up with a larger number of axioms, but they are ideally suited for a computer language such as Metamath. (Section 3.3 shows these axioms.)

We will not elaborate further on the "free variable" and "proper substitution" concepts here. You may consult [21, ch. 3–4] (as well as many other books) for a precise explanation of these concepts. If you intend to do serious mathematical work, it is wise to become familiar with the traditional textbook approach; even though the concepts embedded in their axioms require a higher level of sophistication, they can be more practical to deal with on an everyday, informal basis. Even if you are just developing Metamath proofs, familiarity with the traditional approach can help you arrive at a

proof outline much faster, which you can then convert to the detail required by Metamath.

We do develop proper substitution rules later on, but in set.mm they are defined as derived constructs; they are not primitives.

You should also note that our system of predicate calculus is specifically tailored for set theory; thus there are only two specific predicates $=$ and $\in$ and no functions or constants unlike more general systems. We later add these.

3.2.3 Set Theory

Traditional Zermelo–Fraenkel set theory with the Axiom of Choice has 10 axioms, which can be expressed in the language of predicate calculus. In this section, we will list only the names and brief English descriptions of these axioms, since we will give you the precise formulas used by the Metamath set theory database `set.mm` later on.

In the descriptions of the axioms, we assume that x, y, z, w, and v represent sets. These are the same as the variables in our predicate calculus system above, except that now we informally think of the variables as ranging over sets. Note that the terms "object," "set," "element," "collection," and "family" are synonymous, as are "is an element of," "is a member of," "is contained in," and "belongs to." The different terms are used for convenience; for example, "a collection of sets" is less confusing than "a set of sets." A set x is said to be a "subset" of y if every element of x is also an element of y; we also say x is "included in" y.

The axioms are very general and apply to almost any conceivable mathematical object, and this level of abstraction can be overwhelming at first. To gain an intuitive feel, it can be helpful to draw a picture illustrating the concept; for example, a circle containing dots could represent a collection of sets, and a smaller circle drawn inside the circle could represent a subset. Overlapping circles can illustrate intersection and union. Circles that illustrate the concepts of set theory are frequently used in elementary textbooks and are called Venn diagrams.

1. Axiom of Extensionality: Two sets are identical if they contain the same elements.

2. Axiom of Pairing: The set $\{x, y\}$ exists.

3. Axiom of Power Sets: The power set of a set (the collection of all of its subsets) exists. For example, the power set of $\{x, y\}$ is $\{\varnothing, \{x\}, \{y\}, \{x, y\}\}$ and it exists.

4. Axiom of the Null Set: The empty set $\varnothing$ exists.

5. Axiom of Union: The union of a set (the set containing the elements of its members) exists. For example, the union of $\{\{x, y\}, \{z\}\}$ is $\{x, y, z\}$ and it exists.

6. Axiom of Regularity: Roughly, no set can contain itself, nor can there be membership "loops," such as a set being an element of one of its members.

7. Axiom of Infinity: An infinite set exists. An example of an infinite set is the set of all integers.

8. Axiom of Separation: The set exists that is obtained by restricting x with some property. For example, if the set of all integers exists, then the set of all even integers exists.

9. Axiom of Replacement: The range of a function whose domain is restricted to the elements of a set x, is also a set. For example, there is a function from integers (the function's domain) to their squares (its range). If we restrict the domain to even integers, its range will become the set of squares of even integers, so this axiom asserts that the set of squares of even numbers exists. Technical note: In general, the "function" need not be a set but can be a proper class.

10. Axiom of Choice: Let x be a set whose members are pairwise disjoint (i.e, whose members contain no elements in common). Then there exists another set containing one element from each member of x. For example, if x is $\{\{y, z\}, \{w, v\}\}$, where y, z, w, and v are different sets, then a set such as $\{z, w\}$ exists (but the axiom doesn't tell us which one). (Actually the Axiom of Choice is redundant if the set x, as in this example, has a finite number of elements.)

The Axiom of Choice is usually considered an extension of ZF set theory rather than a proper part of it. It is sometimes considered philosophically controversial because it specifies the existence of a set without specifying what the set is. Constructive logics, including intuitionistic logic, do not accept the axiom of choice. Since there is some lingering controversy, we often prefer proofs that do not use the axiom of choice (where there is a known alternative), and in some cases we will use weaker axioms than the full axiom of choice. That said, the axiom of choice is a powerful and widely-accepted tool, so we do use it when needed. ZF set theory that includes the Axiom of Choice is called Zermelo–Fraenkel set theory with choice (ZFC).

When expressed symbolically, the Axiom of Separation and the Axiom of Replacement contain wff symbols and therefore each represent infinitely many axioms, one for each possible wff. For this reason, they are often called axiom schemes.

It turns out that the Axiom of the Null Set, the Axiom of Pairing, and the Axiom of Separation can be derived from the other axioms and are therefore unnecessary, although they tend to be included in standard texts for various reasons (historical, philosophical, and possibly because some authors may not know this). In the Metamath set theory database, these redundant axioms are derived from the other ones instead of truly being considered axioms. This is in keeping with our general goal of minimizing the number of axioms we must depend on.

3.2.4 Other Axioms

Above we qualified the phrase "all of mathematics" with "essentially." The main important missing piece is the ability to do category theory, which requires huge sets (inaccessible cardinals) larger than those postulated by the ZFC axioms. The Tarski–Grothendieck Axiom postulates the existence of such sets. Note that this is the same axiom used by Mizar for supporting category theory. The Tarski–Grothendieck axiom can be viewed as a very strong replacement of the Axiom of Infinity, the Axiom of Choice, and the Axiom of Power Sets. The `set.mm` database includes this axiom; see the database for details about it. Again, we only use this axiom when we need to. You are only likely to encounter or use this axiom if you are doing category theory, since its use is highly specialized, so we will not list the Tarsky-Grothendieck axiom in the short list of axioms below.

Can there be even more axioms? Of course. Gödel showed that no finite set of axioms or axiom schemes can completely describe any consistent theory strong enough to include arithmetic. But practically speaking, the ones above are the accepted foundation that almost all mathematicians explicitly or implicitly base their work on.

3.3 The Axioms in the Metamath Language

Here we list the axioms as they appear in `set.mm` so you can look them up there easily. Incidentally, the `show statement /tex` command was used to typeset them.

3.3.1 Propositional Calculus

Axiom of Simplification.
`ax-1 $a` $\vdash (\varphi \rightarrow (\psi \rightarrow \varphi))$
Axiom of Distribution.
`ax-2 $a` $\vdash ((\varphi \rightarrow (\psi \rightarrow \chi)) \rightarrow ((\varphi \rightarrow \psi) \rightarrow (\varphi \rightarrow \chi)))$
Axiom of Contraposition.
`ax-3 $a` $\vdash ((\neg \varphi \rightarrow \neg \psi) \rightarrow (\psi \rightarrow \varphi))$
Rule of Modus Ponens.
`min $e` $\vdash \varphi$
`maj $e` $\vdash (\varphi \rightarrow \psi)$
`ax-mp $a` $\vdash \psi$

3.3.2 Axioms of Predicate Calculus with Equality—Tarski's S2

Rule of Generalization.
`ax-g.1 $e` $\vdash \varphi$
`ax-gen $a` $\vdash \forall x \varphi$
Axiom of Quantified Implication.
`ax-4 $a` $\vdash (\forall x (\forall x \varphi \rightarrow \psi) \rightarrow (\forall x \varphi \rightarrow \forall x \psi))$
Axiom of Distinctness.
`ax-5 $a` $\vdash (\varphi \rightarrow \forall x \varphi)$ *where* \$*d* x φ (x *does not occur in* φ)
Axiom of Existence.
`ax-6 $a` $\vdash (\forall x (x = y \rightarrow \forall x \varphi) \rightarrow \varphi)$
Axiom of Equality.
`ax-7 $a` $\vdash (x = y \rightarrow (x = z \rightarrow y = z))$
Axiom of Left Equality for Binary Predicate.
`ax-8 $a` $\vdash (x = y \rightarrow (x \in z \rightarrow y \in z))$
Axiom of Right Equality for Binary Predicate.
`ax-9 $a` $\vdash (x = y \rightarrow (z \in x \rightarrow z \in y))$

3.3.3 Axioms of Predicate Calculus with Equality—Auxiliary

Axiom of Quantified Negation.
`ax-10 $a` $\vdash (\neg \forall x \neg \forall x \varphi \rightarrow \varphi)$
Axiom of Quantifier Commutation.
`ax-11 $a` $\vdash (\forall x \forall y \varphi \rightarrow \forall y \forall x \varphi)$
Axiom of Substitution.
`ax-12 $a` $\vdash (\neg \forall x \; x = y \rightarrow (x = y \rightarrow (\varphi \rightarrow \forall x (x = y \rightarrow \varphi))))$
Axiom of Quantified Equality.
`ax-13 $a` $\vdash (\neg \forall z \; z = x \rightarrow (\neg \forall z \; z = y \rightarrow (x = y \rightarrow \forall z \; x = y)))$

3.3.4 Set Theory

In order to make the axioms of set theory a little more compact, there are several definitions from logic that we make use of implicitly, namely, "logical AND," "logical equivalence," and "there exists."

$$\begin{array}{rcl} (\varphi \wedge \psi) & \text{stands for} & \neg(\varphi \rightarrow \neg\psi) \\ (\varphi \leftrightarrow \psi) & \text{stands for} & ((\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)) \\ \exists x \, \varphi & \text{stands for} & \neg\forall x \neg\varphi \end{array}$$

In addition, the axioms of set theory require that all variables be distinct,[4] thus we also assume:

[4]Set theory axioms can be devised so that *no* variables are required to be distinct, provided we replace `ax-c16` with an axiom stating that "at least two things exist," thus making `ax-5` the only other axiom requiring the `$d` statement. These axioms are unconventional and are not presented here, but they can be found on the `http://metamath.org` web site. See also the Comment on p. 125.

`$d` $x\,y\,z\,w$

Axiom of Extensionality.
`ax-ext $a` $\vdash (\forall x (x \in y \leftrightarrow x \in z) \rightarrow y = z)$
Axiom of Replacement.
`ax-rep $a` $\vdash (\forall w \exists y \forall z (\forall y \varphi \rightarrow z = y) \rightarrow \exists y \forall z (z \in y \leftrightarrow \exists w (w \in x \wedge \forall y \varphi)))$
Axiom of Union.
`ax-un $a` $\vdash \exists x \forall y (\exists x (y \in x \wedge x \in z) \rightarrow y \in x)$
Axiom of Power Sets.
`ax-pow $a` $\vdash \exists x \forall y (\forall x (x \in y \rightarrow x \in z) \rightarrow y \in x)$
Axiom of Regularity.
`ax-reg $a` $\vdash (\exists x \; x \in y \rightarrow \exists x (x \in y \wedge \forall z (z \in x \rightarrow \neg z \in y)))$
Axiom of Infinity.
`ax-inf $a` $\vdash \exists x (y \in x \wedge \forall y (y \in x \rightarrow \exists z (y \in z \wedge z \in x)))$
Axiom of Choice.
`ax-ac $a` $\vdash \exists x \forall y \forall z ((y \in z \wedge z \in w) \rightarrow \exists w \forall y (\exists w ((y \in z \wedge z \in w) \wedge (y \in w \wedge w \in x)) \leftrightarrow y = w))$

3.3.5 That's It

There you have it, the axioms for (essentially) all of mathematics! Wonder at them and stare at them in awe. Put a copy in your wallet, and you will carry in your pocket the encoding for all theorems ever proved and that ever will be proved, from the most mundane to the most profound.

3.4 A Hierarchy of Definitions

The axioms in the previous section in principle embody everything that can be done within standard mathematics. However, it is impractical to accomplish very much by using them directly, for even simple concepts (from a human perspective) can involve extremely long, incomprehensible formulas. Mathematics is made practical by introducing definitions. Definitions usually introduce new symbols, or at least new relationships among existing symbols, to abbreviate more complex formulas. An important requirement for a definition is that there exist a straightforward (algorithmic) method for eliminating the abbreviation by expanding it into the more primitive symbol string that it represents. Some important definitions included in the file `set.mm` are listed in this section for reference, and also to give you a feel for why something like ω (the set of natural numbers 0, 1, 2,...) becomes very complicated when completely expanded into primitive symbols.

What is the motivation for definitions, aside from allowing complicated expressions to be expressed more simply? In the case of ω, one goal is to provide a basis for the theory of natural numbers. Before set theory was invented, a set of axioms for arithmetic, called Peano's postulates, was

devised and shown to have the properties one expects for natural numbers. Now anyone can postulate a set of axioms, but if the axioms are inconsistent contradictions can be derived from them. Once a contradiction is derived, anything can be trivially proved, including all the facts of arithmetic and their negations. To ensure that an axiom system is at least as reliable as the axioms for set theory, we can define sets and operations on those sets that satisfy the new axioms. In the `set.mm` Metamath database, we prove that the elements of ω satisfy Peano's postulates, and it's a long and hard journey to get there directly from the axioms of set theory. But the result is confidence in the foundations of arithmetic. And there is another advantage: we now have all the tools of set theory at our disposal for manipulating objects that obey the axioms for arithmetic.

What are the criteria we use for definitions? First, and of utmost importance, the definition should not be *creative*, that is it should not allow an expression that previously qualified as a wff but was not provable, to become provable. Second, the definition should be *eliminable*, that is, there should exist an algorithmic method for converting any expression using the definition into a logically equivalent expression that previously qualified as a wff.

In almost all cases below, definitions connect two expressions with either $\leftrightarrow$ or $=$. Eliminating[5] such a definition is a simple matter of substituting the expression on the left-hand side (*definiendum* or thing being defined) with the equivalent, more primitive expression on the right-hand side (*definiens* or definition).

Often a definition has variables on the right-hand side which do not appear on the left-hand side; these are called *dummy variables*. In this case, any allowable substitution (such as a new, distinct variable) can be used when the definition is eliminated. Dummy variables may be used only if they are *effectively bound*, meaning that the definition will remain logically equivalent upon any substitution of a dummy variable with any other *qualifying expression*, i.e. any symbol string (such as another variable) that meets the restrictions on the dummy variable imposed by `$d` and `$f` statements. For example, we could define a constant $\bot$ (inverted tee, meaning logical "false") as $(\varphi \wedge \neg\varphi)$, i.e. "phi and not phi." Here φ is effectively bound because the definition remains logically equivalent when we replace φ with any other wff. (It is actually `df-fal` in `set.mm`, which defines $\bot$.)

There are two cases where eliminating definitions is a little more complex. These cases are the definitions `df-bi` and `df-cleq`. The first stretches the concept of a definition a little, as in effect it "defines a definition;" however, it meets our requirements for a definition in that it is eliminable and does not strengthen the language. Theorem `bii` shows the substitution needed to

[5]Here we mean the elimination that a human might do in his or her head. To eliminate them as part of a Metamath proof we would invoke one of a number of theorems that deal with transitivity of equivalence or equality; there are many such examples in the proofs in `set.mm`.

eliminate the $\leftrightarrow$ symbol.

Definition `df-cleq` extends the usage of the equality symbol to include "classes" in set theory. The reason it is potentially problematic is that it can lead to statements which do not follow from logic alone but presuppose the Axiom of Extensionality, so we include this axiom as a hypothesis for the definition. We could have made `df-cleq` directly eliminable by introducing a new equality symbol, but have chosen not to do so in keeping with standard textbook practice. Definitions such as `df-cleq` that extend the meaning of existing symbols must be introduced carefully so that they do not lead to contradictions. Definition `df-clel` also extends the meaning of an existing symbol ($\in$); while it doesn't strengthen the language like `df-cleq`, this is not obvious and it must also be subject to the same scrutiny.

Exercise: Study how the wff $x \in \omega$, meaning "x is a natural number," could be expanded in terms of primitive symbols, starting with the definitions `df-clel` on p. 78 and `df-om` on p. 82 and working your way back. Don't bother to work out the details; just make sure that you understand how you could do it in principle. The answer is shown in the footnote on p. 61. If you actually do work it out, you won't get exactly the same answer because we used a few simplifications such as discarding occurrences of $\neg\neg$ (double negation).

In the definitions below, we have placed the ASCII Metamath source below each of the formulas to help you become familiar with the notation in the database. For simplicity, the necessary `$f` and `$d` statements are not shown. If you are in doubt, use the `show statement` command in the Metamath program to see the full statement. A selection of this notation is summarized in Appendix A.

To understand the motivation for these definitions, you should consult the references indicated: Takeuti and Zaring [67], Quine [55], Bell and Machover [5], and Enderton [18]. Our list of definitions is provided more for reference than as a learning aid. However, by looking at a few of them you can gain a feel for how the hierarchy is built up. The definitions are a representative sample of the many definitions in `set.mm`, but they are complete with respect to the theorem examples we will present in Section 3.6. Also, some are slightly different from, but logically equivalent to, the ones in `set.mm` (some of which have been revised over time to shorten them, for example).

3.4.1 Definitions for Propositional Calculus

The symbols φ, ψ, and χ represent wffs.

Our first definition introduces the biconditional connective[6] (also called

[6]The term "connective" is informally used to mean a symbol that is placed between two variables or adjacent to a variable, whereas a mathematical "constant" usually indicates a symbol such as the number 0 that may replace a variable or metavariable. From Metamath's point of view, there is no distinction between a connective and a constant; both are constants in the Metamath language.

logical equivalence). Unlike most traditional developments, we have chosen not to have a separate symbol such as "Df." to mean "is defined as." Instead, we will use the biconditional connective for this purpose, as it lets us use logic to manipulate definitions directly. Here we state the properties of the biconditional connective with a carefully crafted `$a` statement, which effectively uses the biconditional connective to define itself. The $\leftrightarrow$ symbol can be eliminated from a formula using theorem `bii`, which is derived later.

Define the biconditional connective.

`df-bi $a` $\vdash \neg (((\varphi \leftrightarrow \psi) \rightarrow \neg ((\varphi \rightarrow \psi) \rightarrow \neg (\psi \rightarrow \varphi))) \rightarrow \neg (\neg ((\varphi \rightarrow \psi) \rightarrow \neg (\psi \rightarrow \varphi)) \rightarrow (\varphi \leftrightarrow \psi)))$

```
df-bi $a |- -. ( ( ( ph <-> ps ) -> -. ( ( ph -> ps ) -> -.
         ( ps -> ph ) ) ) -> -. ( -. ( ( ph -> ps ) -> -. (
         ps -> ph ) ) -> ( ph <-> ps ) ) ) $.
```

This theorem relates the biconditional connective to primitive connectives and can be used to eliminate the $\leftrightarrow$ symbol from any wff.

`bii $p` $\vdash ((\varphi \leftrightarrow \psi) \leftrightarrow \neg ((\varphi \rightarrow \psi) \rightarrow \neg (\psi \rightarrow \varphi)))$

```
bii $p |- ( ( ph <-> ps ) <-> -. ( ( ph -> ps ) -> -. ( ps
       -> ph ) ) ) $= ... $.
```

Define disjunction (OR).

`df-or $a` $\vdash ((\varphi \vee \psi) \leftrightarrow (\neg \varphi \rightarrow \psi))$

```
df-or $a |- ( ( ph \/ ps ) <-> ( -. ph -> ps ) ) $.
```

Define conjunction (AND).

`df-an $a` $\vdash ((\varphi \wedge \psi) \leftrightarrow \neg (\varphi \rightarrow \neg \psi))$

```
df-an $a |- ( ( ph /\ ps ) <-> -. ( ph -> -. ps ) ) $.
```

Define disjunction (OR) of 3 wffs.

`df-3or $a` $\vdash ((\varphi \vee \psi \vee \chi) \leftrightarrow ((\varphi \vee \psi) \vee \chi))$

```
df-3or $a |- ( ( ph \/ ps \/ ch ) <-> ( ( ph \/ ps ) \/ ch )
          ) $.
```

Define conjunction (AND) of 3 wffs.

`df-3an $a` $\vdash ((\varphi \wedge \psi \wedge \chi) \leftrightarrow ((\varphi \wedge \psi) \wedge \chi))$

```
df-3an $a |- ( ( ph /\ ps /\ ch ) <-> ( ( ph /\ ps ) /\ ch )
          ) $.
```

3.4.2 Definitions for Predicate Calculus

The symbols x, y, and z represent individual variables of predicate calculus. In this section, they are not necessarily distinct unless it is explicitly mentioned.

Define existential quantification. The expression $\exists x \varphi$ means "there exists an x where φ is true."

```
df-ex $a ⊢ ( ∃ x φ ↔ ¬ ∀ x ¬ φ )

df-ex $a |- ( E. x ph <-> -. A. x -. ph ) $.
```

Define proper substitution. In our notation, we use $[y/x]\varphi$ to mean "the wff that results when y is properly substituted for x in the wff φ."[7] For example, $[y/x]z \in x$ is the same as $z \in y$. One way to remember this notation is to notice that it looks like division and recall that $(y/x) \cdot x$ is y (when $x \neq 0$). The notation is different from the notation $\varphi(x|y)$ that is sometimes used, because the latter notation is ambiguous for us: for example, we don't know whether $\neg\varphi(x|y)$ is to be interpreted as $\neg(\varphi(x|y))$ or $(\neg\varphi)(x|y)$.[8] Other texts often use $\varphi(y)$ to indicate our $[y/x]\varphi$, but this notation is even more ambiguous since there is no explicit indication of what is being substituted. Note that this definition is valid even when x and y are the same variable. The first conjunct is a "trick" used to achieve this property, making the definition look somewhat peculiar at first.

```
df-sb $a ⊢ ( [ y / x ] φ ↔ ( ( x = y → φ ) ∧ ∃ x ( x = y ∧ φ ) ) )

df-sb $a |- ( [ y / x ] ph <-> ( ( x = y -> ph ) /\ E. x ( x
        = y /\ ph ) ) ) $.
```

Define existential uniqueness ("there exists exactly one"). Note that y is a variable distinct from x and not occurring in φ.

```
df-eu $a ⊢ ( ∃! x φ ↔ ∃ y ∀ x ( φ ↔ x = y ) )

df-eu $a |- ( E! x ph <-> E. y A. x ( ph <-> x = y ) ) $.
```

[7] This can also be described as substituting x with y, y properly replaces x, or x is properly replaced by y.

[8] Because of the way we initially defined wffs, this is the case with any postfix connective (one occurring after the symbols being connected) or infix connective (one occurring between the symbols being connected). Metamath does not have a built-in notion of operator binding strength that could eliminate the ambiguity. The initial parenthesis effectively provides a prefix connective to eliminate ambiguity. Some conventions, such as Polish notation used in the 1930's and 1940's by Polish logicians, use only prefix connectives and thus allow the total elimination of parentheses, at the expense of readability. In Metamath we could actually redefine all notation to be Polish if we wanted to without having to change any proofs!

3.4.3 Definitions for Set Theory

The symbols x, y, z, and w represent individual variables of predicate calculus, which in set theory are understood to be sets. However, using only the constructs shown so far would be very inconvenient.

To make set theory more practical, we introduce the notion of a "class." A class is either a set variable (such as x) or an expression of the form $\{x|\varphi\}$ (called an "abstraction class"). Note that sets (i.e. individual variables) always exist (this is a theorem of logic, namely $\exists y\, y = x$ for any set x), whereas classes may or may not exist (i.e. $\exists y\, y = A$ may or may not be true). If a class does not exist it is called a "proper class." Definitions `df-clab`, `df-cleq`, and `df-clel` can be used to convert an expression containing classes into one containing only set variables and wff metavariables.

The symbols A, B, C, D, F, G, and R are metavariables that range over classes. A class metavariable A may be eliminated from a wff by replacing it with $\{x|\varphi\}$ where neither x nor φ occur in the wff.

The theory of classes can be shown to be an eliminable and conservative extension of set theory. The **eliminability** property shows that for every formula in the extended language we can build a logically equivalent formula in the basic language; so that even if the extended language provides more ease to convey and formulate mathematical ideas for set theory, its expressive power does not in fact strengthen the basic language's expressive power. The **conservation** property shows that for every proof of a formula of the basic language in the extended system we can build another proof of the same formula in the basic system; so that, concerning theorems on sets only, the deductive powers of the extended system and of the basic system are identical. Together, these properties mean that the extended language can be treated as a definitional extension that is **sound**.

A rigorous justification, which we will not give here, can be found in Levy [38, pp. 357-366] supplementing his informal introduction to class theory on pp. 7-17. Two other good treatments of class theory are provided by Quine [55, pp. 15-21] and also [67, pp. 10-14]. Quine's exposition (he calls them virtual classes) is nicely written and very readable.

In the rest of this section, individual variables are always assumed to be distinct from each other unless otherwise indicated. In addition, dummy variables on the right-hand side of a definition do not occur in the class and wff metavariables in the definition.

The definitions we present here are a partial but self-contained collection selected from several hundred that appear in the current `set.mm` database. They are adequate for a basic development of elementary set theory.

Define the abstraction class. x and y need not be distinct. Definition 2.1 of Quine, p. 16. This definition may seem puzzling since it is shorter than the expression being defined and does not buy us anything in terms of brevity. The reason we introduce this definition is because it fits in neatly with the

extension of the $\in$ connective provided by `df-clel`.

`df-clab $a` $\vdash (x \in \{ y \mid \varphi \} \leftrightarrow [x / y] \varphi)$

```
df-clab $a |- ( x e. { y | ph } <-> [ x / y ] ph ) $.
```

Define the equality connective between classes. See Quine or Chapter 4 of Takeuti and Zaring for its justification and methods for eliminating it. This is an example of a somewhat "dangerous" definition, because it extends the use of the existing equality symbol rather than introducing a new symbol, allowing us to make statements in the original language that may not be true. For example, it permits us to deduce $y = z \leftrightarrow \forall x(x \in y \leftrightarrow x \in z)$ which is not a theorem of logic but rather presupposes the Axiom of Extensionality, which we include as a hypothesis so that we can know when this axiom is assumed in a proof (with the `show trace_back` command). We could avoid the danger by introducing another symbol, say $=\!\!\!=$, in place of $=$; this would also have the advantage of making elimination of the definition straightforward and would eliminate the need for Extensionality as a hypothesis. We would then also have the advantage of being able to identify exactly where Extensionality truly comes into play. One of our theorems would be $x =\!\!\!= y \leftrightarrow x = y$ by invoking Extensionality. However in keeping with standard practice we retain the "dangerous" definition.

`df-cleq.1 $e` $\vdash (\forall x (x \in y \leftrightarrow x \in z) \rightarrow y = z)$
`df-cleq $a` $\vdash (A = B \leftrightarrow \forall x (x \in A \leftrightarrow x \in B))$

```
df-cleq.1 $e |- ( A. x ( x e. y <-> x e. z ) -> y = z ) $.


df-cleq $a |- ( A = B <-> A. x ( x e. A <-> x e. B ) ) $.
```

Define the membership connective between classes. Theorem 6.3 of Quine, p. 41, which we adopt as a definition. Note that it extends the use of the existing membership symbol, but unlike `df-cleq` it does not extend the set of valid wffs of logic when the class metavariables are replaced with set variables.

`df-clel $a` $\vdash (A \in B \leftrightarrow \exists x (x = A \wedge x \in B))$

```
df-clel $a |- ( A e. B <-> E. x ( x = A /\ x e. B ) ) $.?
```

Define inequality.

`df-ne $a` $\vdash (A \neq B \leftrightarrow \neg A = B)$

```
df-ne $a |- ( A =/= B <-> -. A = B ) $.
```

Define restricted universal quantification. Enderton, p. 22.

`df-ral $a` $\vdash (\forall x \in A \, \varphi \leftrightarrow \forall x (x \in A \rightarrow \varphi))$

```
df-ral $a |- ( A. x e. A ph <-> A. x ( x e. A -> ph ) ) $.
```

Define restricted existential quantification. Enderton, p. 22.

`df-rex $a` $\vdash (\exists x \in A \, \varphi \leftrightarrow \exists x \, (x \in A \wedge \varphi))$

```
df-rex $a |- ( E. x e. A ph <-> E. x ( x e. A /\ ph ) ) $.
```

Define the universal class. Definition 5.20, p. 21, of Takeuti and Zaring.

`df-v $a` $\vdash \mathrm{V} = \{ x \mid x = x \}$

```
df-v $a |- _V = { x | x = x } $.
```

Define the subclass relationship between two classes (called the subset relation if the classes are sets i.e. are not proper). Definition 5.9 of Takeuti and Zaring, p. 17.

`df-ss $a` $\vdash (A \subseteq B \leftrightarrow \forall x \, (x \in A \rightarrow x \in B))$

```
df-ss $a |- ( A C_ B <-> A. x ( x e. A -> x e. B ) ) $.
```

Define the union of two classes. Definition 5.6 of Takeuti and Zaring, p. 16.

`df-un $a` $\vdash (A \cup B) = \{ x \mid (x \in A \vee x \in B) \}$

```
df-un $a ( A u. B ) = { x | ( x e. A \/ x e. B ) } $.
```

Define the intersection of two classes. Definition 5.6 of Takeuti and Zaring, p. 16.

`df-in $a` $\vdash (A \cap B) = \{ x \mid (x \in A \wedge x \in B) \}$

```
df-in $a |- ( A i^i B ) = { x | ( x e. A /\ x e. B ) } $.
```

Define class difference. Definition 5.12 of Takeuti and Zaring, p. 20. Several notations are used in the literature; we chose the $\setminus$ convention instead of a minus sign to reserve the latter for later use in, e.g., arithmetic.

`df-dif $a` $\vdash (A \setminus B) = \{ x \mid (x \in A \wedge \neg x \in B) \}$

```
df-dif $a ( A \ B ) = { x | ( x e. A /\ -. x e. B ) } $.
```

Define the empty or null set. Compare Definition 5.14 of Takeuti and Zaring, p. 20.

`df-nul $a` $\vdash \varnothing = (\mathrm{V} \setminus \mathrm{V})$

```
df-nul $a |- (/) = ( _V \ _V ) $.
```

Define power class. Definition 5.10 of Takeuti and Zaring, p. 17, but we also let it apply to proper classes. (Note that `~P` is the symbol for calligraphic P, the tilde suggesting "curly;" see Appendix A.)

`df-pw $a` $\vdash \mathcal{P}\, A = \{\, x \mid x \subseteq A \,\}$

```
df-pw $a |- ~P A = { x | x C_ A } $.
```

Define the singleton of a class. Definition 7.1 of Quine, p. 48. It is well-defined for proper classes, although it is not very meaningful in this case, where it evaluates to the empty set.

`df-sn $a` $\vdash \{\, A \,\} = \{\, x \mid x = A \,\}$

```
df-sn $a |- { A } = { x | x = A } $.
```

Define an unordered pair of classes. Definition 7.1 of Quine, p. 48.

`df-pr $a` $\vdash \{\, A , B \,\} = (\{\, A \,\} \cup \{\, B \,\})$

```
df-pr $a |- { A , B } = ( { A } u. { B } ) $.
```

Define an unordered triple of classes. Definition of Enderton, p. 19.

`df-tp $a` $\vdash \{\, A , B , C \,\} = (\{\, A , B \,\} \cup \{\, C \,\})$

```
df-tp $a |- { A , B , C } = ( { A , B } u. { C } ) $.
```

Kuratowski's ordered pair definition. Definition 9.1 of Quine, p. 58. For proper classes it is not meaningful but is well-defined for convenience. (Note that `<.` stands for $\langle$ whereas `<` stands for $<$, and similarly for `>.` .)

`df-op $a` $\vdash \langle\, A , B \,\rangle = \{\, \{\, A \,\} , \{\, A , B \,\} \,\}$

```
df-op $a |- <. A , B >. = { { A } , { A , B } } $.
```

Define the union of a class. Definition 5.5, p. 16, of Takeuti and Zaring.

`df-uni $a` $\vdash \bigcup A = \{\, x \mid \exists y \,(\, x \in y \wedge y \in A \,)\,\}$

```
df-uni $a |- U. A = { x | E. y ( x e. y /\ y e. A ) } $.
```

Define the intersection of a class. Definition 7.35, p. 44, of Takeuti and Zaring.

`df-int $a` $\vdash \bigcap A = \{\, x \mid \forall y \,(\, y \in A \rightarrow x \in y \,)\,\}$

```
df-int $a |- |^| A = { x | A. y ( y e. A -> x e. y ) } $.
```

Define a transitive class. This should not be confused with a transitive relation, which is a different concept. Definition from p. 71 of Enderton, extended to classes.

`df-tr $a` $\vdash (\, \mathrm{Tr}\, A \leftrightarrow \bigcup A \subseteq A \,)$

```
df-tr $a |- ( Tr A <-> U. A C_ A ) $.
```

Define a notation for a general binary relation. Definition 6.18, p. 29, of Takeuti and Zaring, generalized to arbitrary classes. This definition is well-defined, although not very meaningful, when classes A and/or B are proper. The lack of parentheses (or any other connective) creates no ambiguity since we are defining an atomic wff.

```
df-br $a ⊢ ( A R B ↔ ⟨ A , B ⟩ ∈ R )
```

```
df-br $a |- ( A R B <-> <. A , B >. e. R ) $.
```

Define an abstraction class of ordered pairs. A special case of Definition 4.16, p. 14, of Takeuti and Zaring. Note that z must be distinct from x and y, and z must not occur in φ, but x and y may be identical and may appear in φ.

```
df-opab $a ⊢ { ⟨ x , y ⟩ | φ } = { z | ∃ x ∃ y ( z = ⟨ x , y ⟩ ∧ φ ) }
```

```
df-opab $a |- { <. x , y >. | ph } = { z | E. x E. y ( z = <.
          x , y >. / ph ) } $.
```

Define the epsilon relation. Similar to Definition 6.22, p. 30, of Takeuti and Zaring.

```
df-eprel $a ⊢ E = { ⟨ x , y ⟩ | x ∈ y }
```

```
df-eprel $a |- _E = { <. x , y >. | x e. y } $.
```

Define a founded relation. R is a founded relation on A iff (if and only if) each nonempty subset of A has an "R-minimal element." Similar to Definition 6.21, p. 30, of Takeuti and Zaring.

```
df-fr $a ⊢ ( R Fr A ↔ ∀ x ( ( x ⊆ A ∧ ¬ x = ∅ ) → ∃ y ( y ∈ x ∧ ( x ∩ { z | z
          R y } ) = ∅ ) ) )
```

```
df-fr $a |- ( R Fr A <-> A. x ( ( x C_ A /\ -. x = (/) ) ->
         E. y ( y e. x /\ ( x i^i { z | z R y } ) = (/) ) )
         ) $.
```

Define a well-ordering. R is a well-ordering of A iff it is founded on A and the elements of A are pairwise R-comparable. Similar to Definition 6.24(2), p. 30, of Takeuti and Zaring.

```
df-we $a ⊢ ( R We A ↔ ( R Fr A ∧ ∀ x ∀ y ( ( x ∈ A ∧ y ∈ A ) → ( x R y ∨
          x = y ∨ y R x ) ) ) )
```

```
df-we $a ( R We A <-> ( R Fr A /\ A. x A. y ( ( x e. A /\ y
         e. A ) -> ( x R y \/ x = y \/ y R x ) ) ) ) $.
```

Define the ordinal predicate, which is true for a class that is transitive and is well-ordered by the epsilon relation. Similar to definition on p. 468, Bell and Machover.

`df-ord $a` $\vdash (\mathrm{Ord}\ A \leftrightarrow (\mathrm{Tr}\ A \wedge E\ \mathrm{We}\ A))$

```
df-ord $a |- ( Ord A <-> ( Tr A /\ E We A ) ) $.
```

Define the class of all ordinal numbers. An ordinal number is a set that satisfies the ordinal predicate. Definition 7.11 of Takeuti and Zaring, p. 38.

`df-on $a` $\vdash \mathrm{On} = \{ x \mid \mathrm{Ord}\ x \}$

```
df-on $a |- On = { x | Ord x } $.
```

Define the limit ordinal predicate, which is true for a non-empty ordinal that is not a successor (i.e. that is the union of itself). Compare Bell and Machover, p. 471 and Exercise (1), p. 42 of Takeuti and Zaring.

`df-lim $a` $\vdash (\mathrm{Lim}\ A \leftrightarrow (\ \mathrm{Ord}\ A \wedge \neg\ A = \varnothing \wedge A = \bigcup A))$

```
df-lim $a |- ( Lim A <-> ( Ord A /\ -. A = (/) /\ A = U. A )
          ) $.
```

Define the successor of a class. Definition 7.22 of Takeuti and Zaring, p. 41. Our definition is a generalization to classes, although it is meaningless when classes are proper.

`df-suc $a` $\vdash \mathrm{suc}\ A = (A \cup \{ A \})$

```
df-suc $a |- suc A = ( A u. { A } ) $.
```

Define the class of natural numbers. Compare Bell and Machover, p. 471.

`df-om $a` $\vdash \omega = \{ x \mid (\mathrm{Ord}\ x \wedge \forall y (\mathrm{Lim}\ y \rightarrow x \in y)) \}$

```
df-om $a |- om = { x | ( Ord x /\ A. y ( Lim y -> x e. y ) )
         } $.
```

Define the Cartesian product (also called the cross product) of two classes. Definition 9.11 of Quine, p. 64.

`df-xp $a` $\vdash (A \times B) = \{ \langle x , y \rangle \mid (x \in A \wedge y \in B) \}$

```
df-xp $a |- ( A X. B ) = { <. x , y >. | ( x e. A /\ y e. B)
         } $.
```

Define a relation. Definition 6.4(1) of Takeuti and Zaring, p. 23.

`df-rel $a` $\vdash (\mathrm{Rel}\ A \leftrightarrow A \subseteq (\mathrm{V} \times \mathrm{V}))$

```
df-rel $a |- ( Rel A <-> A C_ ( _V X. _V ) ) $.
```

Define the domain of a class. Definition 6.5(1) of Takeuti and Zaring, p. 24.

`df-dm $a` $\vdash \mathrm{dom}\, A = \{ x \mid \exists y \langle x, y \rangle \in A \}$

```
df-dm $a |- dom A = { x | E. y <. x , y >. e. A } $.
```

Define the range of a class. Definition 6.5(2) of Takeuti and Zaring, p. 24.

`df-rn $a` $\vdash \mathrm{ran}\, A = \{ y \mid \exists x \langle x, y \rangle \in A \}$

```
df-rn $a |- ran A = { y | E. x <. x , y >. e. A } $.
```

Define the restriction of a class. Definition 6.6(1) of Takeuti and Zaring, p. 24.

`df-res $a` $\vdash (A \restriction B) = (A \cap (B \times \mathrm{V}))$

```
df-res $a |- ( A |` B ) = ( A i^i ( B X. _V ) ) $.
```

Define the image of a class. Definition 6.6(2) of Takeuti and Zaring, p. 24.

`df-ima $a` $\vdash (A \mathbin{``} B) = \mathrm{ran}\, (A \restriction B)$

```
df-ima $a |- ( A " B ) = ran ( A |` B ) $.
```

Define the composition of two classes. Definition 6.6(3) of Takeuti and Zaring, p. 24.

`df-co $a` $\vdash (A \circ B) = \{ \langle x, y \rangle \mid \exists z (\langle x, z \rangle \in B \wedge \langle z, y \rangle \in A) \}$

```
df-co $a |- ( A o. B ) = { <. x , y >. | E. z ( <. x , z >.
          e. B /\ <. z , y >. e. A ) } $.
```

Define a function. Definition 6.4(4) of Takeuti and Zaring, p. 24.

`df-fun $a` $\vdash (\mathrm{Fun}\, A \leftrightarrow (\mathrm{Rel}\, A \wedge \forall x \exists z \forall y (\langle x, y \rangle \in A \rightarrow y = z)))$

```
df-fun $a |- ( Fun A <-> ( Rel A / A. x E. z A. y ( <. x , y
          >. e. A -> y = z ) ) ) $.
```

Define a function with domain. Definition 6.15(1) of Takeuti and Zaring, p. 27.

`df-fn $a` $\vdash (A \mathrm{\ Fn\ } B \leftrightarrow (\mathrm{Fun}\, A \wedge \mathrm{dom}\, A = B))$

```
df-fn $a |- ( A Fn B <-> ( Fun A /\ dom A = B ) ) $.
```

Define a function with domain and co-domain. Definition 6.15(3) of Takeuti and Zaring, p. 27.

`df-f $a` $\vdash (F : A \longrightarrow B \leftrightarrow (F \mathrm{\ Fn\ } A \wedge \mathrm{ran}\, F \subseteq B))$

```
df-f $a |- ( F : A --> B <-> ( F Fn A /\ ran F C_ B ) ) $.
```

Define a one-to-one function. Compare Definition 6.15(5) of Takeuti and Zaring, p. 27.

`df-f1 $a` $\vdash (F : A \xrightarrow{1\text{-}1} B \leftrightarrow (F : A \longrightarrow B \wedge \forall y \exists z \forall x (\langle x , y \rangle \in F \rightarrow x = z)))$

```
df-f1 $a |- ( F : A -1-1-> B <-> ( F : A --> B /\ A. y E. z
         A. x ( <. x , y >. e. F -> x = z ) ) ) $.
```

Define an onto function. Definition 6.15(4) of Takeuti and Zaring, p. 27.

`df-fo $a` $\vdash (F : A \xrightarrow[\text{onto}]{} B \leftrightarrow (F \text{ Fn } A \wedge \text{ran } F = B))$

```
df-fo $a |- ( F : A -onto-> B <-> ( F Fn A / ran F = B ) ) $.
```

Define a one-to-one, onto function. Compare Definition 6.15(6) of Takeuti and Zaring, p. 27.

`df-f1o $a` $\vdash (F : A \xrightarrow[\text{onto}]{1\text{-}1} B \leftrightarrow (F : A \xrightarrow{1\text{-}1} B \wedge F : A \xrightarrow[\text{onto}]{} B))$

```
df-f1o $a |- ( F : A -1-1-onto-> B <-> ( F : A -1-1-> B?
          /\ F : A -onto-> B ) ) $.?
```

Define the value of a function. This definition applies to any class and evaluates to the empty set when it is not meaningful. Note that F‘A means the same thing as the more familiar $F(A)$ notation for a function’s value at A. The F‘A notation is common in formal set theory.

`df-fv $a` $\vdash (F \text{‘} A) = \bigcup \{ x \mid (F \text{“} \{ A \}) = \{ x \} \}$

```
df-fv $a |- ( F ‘ A ) = U. { x | ( F " { A } ) = { x } } $.
```

Define the result of an operation. Here, F is an operation on two values (such as $+$ for real numbers). This is defined for proper classes A and B even though not meaningful in that case. However, the definition can be meaningful when F is a proper class.

`df-opr $a` $\vdash (A \; F \; B) = (F \text{‘} \langle A , B \rangle)$

```
df-opr $a |- ( A F B ) = ( F ‘ <. A , B >. ) $.
```

3.5 Tricks of the Trade

In the `set.mm` database our goal was usually to conform to modern notation. However in some cases the relationship to standard textbook language may be obscured by several unconventional devices we used to simplify the development and to take advantage of the Metamath language. In this section we will describe some common conventions used in `set.mm`.

- The turnstile symbol, $\vdash$, meaning "it is provable that," is the first token of all assertions and hypotheses that aren't syntax constructions. This is a standard convention in logic. (We mentioned this earlier, but this symbol is bothersome to some people without a logic background. It has no deeper meaning but just provides us with a way to distinguish syntax constructions from ordinary mathematical statements.)

- A hypothesis of the form

  `$e` $\vdash (\varphi \rightarrow \forall x \, \varphi)$

  should be read "assume variable x is (effectively) not free in wff φ." Literally, this says "assume it is provable that $\varphi \rightarrow \forall x \, \varphi$." This device lets us avoid the complexities associated with the standard treatment of free and bound variables. The footnote on p. 198 discusses this further.

- A statement of one of the forms

  `$a` $\vdash (\neg \forall x \; x = y \rightarrow \ldots)$
  `$p` $\vdash (\neg \forall x \; x = y \rightarrow \ldots)$

  should be read "if x and y are distinct variables, then..." This antecedent provides us with a technical device to avoid the need for the `$d` statement early in our development of predicate calculus, permitting symbol manipulations to be as conceptually simple as those in propositional calculus. However, the `$d` statement eventually becomes a requirement, and after that this device is rarely used.

- The statement

  `$d` $x \; y$

  should be read "assume x and y are distinct variables."

- The statement

  `$d` $x \; \varphi$

  should be read "assume x does not occur in φ."

- The statement

`$d` x A

should be read "assume variable x does not occur in class A."

- The restriction and hypothesis group

  `$d` x A
  `$d` x ψ
  `$e` $\vdash (x = A \rightarrow (\varphi \leftrightarrow \psi))$

  is frequently used in place of explicit substitution, meaning "assume ψ results from the proper substitution of A for x in φ." Sometimes "`$e` $\vdash (\psi \rightarrow \forall x \, \psi)$" is used instead of "`$d` $x \, \psi$," which requires only that x be effectively not free in φ but not necessarily absent from it. The use of implicit substitution is partly a matter of personal style, although it may make proofs somewhat shorter than would be the case with explicit substitution.

- The hypothesis

  `$e` $\vdash A \in \mathrm{V}$

  should be read "assume class A is a set (i.e. exists)." This is a convenient convention used by Quine.

- The restriction and hypothesis

  `$d` x y
  `$e` $\vdash (y \in A \rightarrow \forall x \; y \in A)$

  should be read "assume variable x is (effectively) not free in class A."

3.6 A Theorem Sampler

In this section we list some of the more important theorems that are proved in the `set.mm` database, and they illustrate the kinds of things that can be done with Metamath. While all of these facts are well-known results, Metamath offers the advantage of easily allowing you to trace their derivation back to axioms. Our intent here is not to try to explain the details or motivation; for this we refer you to the textbooks that are mentioned in the descriptions. (The `set.mm` file has bibliographic references for the text references.) Their proofs often embody important concepts you may wish to explore with the Metamath program (see Section 3.10). All the symbols that are used here are defined in Section 3.4. For brevity we haven't included the `$d` restrictions or `$f` hypotheses for these theorems; when you are uncertain consult the `set.mm` database.

We start with `syl` (principle of the syllogism). In *Principia Mathematica* Whitehead and Russell call this "the principle of the syllogism... because...

the syllogism in Barbara is derived from them" [74, quote after Theorem *2.06 p. 101]. Some authors call this law a "hypothetical syllogism." As of 2019 `syl` is the most commonly referenced proven assertion in the `set.mm` database.[9]

Theorem syl (principle of the syllogism).

`syl.1 $e` $\vdash (\varphi \rightarrow \psi)$
`syl.2 $e` $\vdash (\psi \rightarrow \chi)$
`syl $p` $\vdash (\varphi \rightarrow \chi)$

The following theorem is not very deep but provides us with a notational device that is frequently used. It allows us to use the expression "$A \in V$" as a compact way of saying that class A exists, i.e. is a set.

Two ways to say "A is a set": A is a member of the universe V if and only if A exists (i.e. there exists a set equal to A). Theorem 6.9 of Quine, p. 43.

`isset $p` $\vdash (A \in \mathrm{V} \leftrightarrow \exists x \; x = A)$

Next we prove the axioms of standard ZF set theory that were missing from our axiom system. From our point of view they are theorems since they can be derived from the other axioms.

Axiom of Separation (Aussonderung) proved from the other axioms of ZF set theory. Compare Exercise 4 of Takeuti and Zaring, p. 22.

`inex1.1 $e` $\vdash A \in \mathrm{V}$
`inex $p` $\vdash (A \cap B) \in \mathrm{V}$

Axiom of the Null Set proved from the other axioms of ZF set theory. Corollary 5.16 of Takeuti and Zaring, p. 20.

`0ex $p` $\vdash \varnothing \in \mathrm{V}$

The Axiom of Pairing proved from the other axioms of ZF set theory. Theorem 7.13 of Quine, p. 51.

`prex $p` $\vdash \{ A , B \} \in \mathrm{V}$

Next we will list some famous or important theorems that are proved in the `set.mm` database. None of them except `omex` require the Axiom of Infinity, as you can verify with the `show trace_back` Metamath command.

The resolution of Russell's paradox. There exists no set containing the set of all sets which are not members of themselves. Proposition 4.14 of Takeuti and Zaring, p. 14.

`ru $p` $\vdash \neg \exists x \; x = \{ y \mid \neg y \in y \}$

[9]The Metamath program command `show usage` shows the number of references. On 2019-04-29 (commit 71cbbbdb387e) `syl` was directly referenced 10,819 times. The second most commonly referenced proven assertion was `eqid`, which was directly referenced 7,738 times.

Cantor's theorem. No set can be mapped onto its power set. Compare Theorem 6B(b) of Enderton, p. 132.

`canth.1 $e` $\vdash A \in \mathrm{V}$
`canth $p` $\vdash \neg\, F : A \underset{\text{onto}}{\longrightarrow} \mathcal{P}\, A$

The Burali-Forti paradox. No set contains all ordinal numbers. Enderton, p. 194. (Burali-Forti was one person, not two.)

`onprc $p` $\vdash \neg\, \mathrm{On} \in \mathrm{V}$

Peano's postulates for arithmetic. Proposition 7.30 of Takeuti and Zaring, pp. 42–43. The objects being described are the members of ω i.e. the natural numbers 0, 1, 2,.... The successor operation suc means "plus one." `peano1` says that 0 (which is defined as the empty set) is a natural number. `peano2` says that if A is a natural number, so is $A + 1$. `peano3` says that 0 is not the successor of any natural number. `peano4` says that two natural numbers are equal if and only if their successors are equal. `peano5` is essentially the same as mathematical induction.

`peano1 $p` $\vdash \varnothing \in \omega$

`peano2 $p` $\vdash (A \in \omega \to \mathrm{suc}\, A \in \omega)$

`peano3 $p` $\vdash (A \in \omega \to \neg\, \mathrm{suc}\, A = \varnothing)$

`peano4 $p` $\vdash ((A \in \omega \wedge B \in \omega) \to (\mathrm{suc}\, A = \mathrm{suc}\, B \leftrightarrow A = B))$

`peano5 $p` $\vdash ((\varnothing \in A \wedge \forall x \in \omega\, (x \in A \to \mathrm{suc}\, x \in A)) \to \omega \subseteq A)$

Finite Induction (mathematical induction). The first hypothesis is the basis and the second is the induction hypothesis. Theorem Schema 22 of Suppes, p. 136.

`findes.1 $e` $\vdash [\varnothing \,/\, x]\, \varphi$
`findes.2 $e` $\vdash (x \in \omega \to (\varphi \to [\mathrm{suc}\, x \,/\, x]\, \varphi))$
`findes $p` $\vdash (x \in \omega \to \varphi)$

Transfinite Induction with explicit substitution. The first hypothesis is the basis, the second is the induction hypothesis for successors, and the third is the induction hypothesis for limit ordinals. Theorem Schema 4 of Suppes, p. 197.

`tfindes.1 $e` $\vdash [\varnothing \,/\, x]\, \varphi$
`tfindes.2 $e` $\vdash (x \in \mathrm{On} \to (\varphi \to [\mathrm{suc}\, x \,/\, x]\, \varphi))$
`tfindes.3 $e` $\vdash (\mathrm{Lim}\, y \to (\forall x \in y\, \varphi \to [y \,/\, x]\, \varphi))$
`tfindes $p` $\vdash (x \in \mathrm{On} \to \varphi)$

Principle of Transfinite Recursion. Theorem 7.41 of Takeuti and Zaring, p. 47. Transfinite recursion is the key theorem that allows arithmetic of ordinals to be rigorously defined, and has many other important uses as well. Hypotheses `tfr.1` and `tfr.2` specify a certain (proper) class F. The complicated definition of F is not important in itself; what is important is

that there be such an F with the required properties, and we show this by displaying F explicitly. `tfr1` states that F is a function whose domain is the set of ordinal numbers. `tfr2` states that any value of F is completely determined by its previous values and the values of an auxiliary function, G. `tfr3` states that F is unique, i.e. it is the only function that satisfies `tfr1` and `tfr2`. Note that f is an individual variable like x and y; it is just a mnemonic to remind us that A is a collection of functions.

`tfr.1 $e` $\vdash A = \{ f \mid \exists x \in \mathrm{On} \, (f \, \mathrm{Fn} \, x \wedge \forall y \in x \, (f \, ` \, y ) = ( G \, ` \, (f \restriction y))) \}$

`tfr.2 $e` $\vdash F = \bigcup A$

`tfr1 $p` $\vdash F \, \mathrm{Fn} \, \mathrm{On}$

`tfr2 $p` $\vdash (z \in \mathrm{On} \rightarrow (F \, ` \, z ) = ( G \, ` \, (F \restriction z)))$

`tfr3 $p` $\vdash ((B \, \mathrm{Fn} \, \mathrm{On} \wedge \forall x \in \mathrm{On} \, (B \, ` \, x ) = ( G \, ` \, (B \restriction x))) \rightarrow B = F)$

The existence of omega (the class of natural numbers). Axiom 7 of Takeuti and Zaring, p. 43. (This is the only theorem in this section requiring the Axiom of Infinity.)

`omex $p` $\vdash \omega \in \mathrm{V}$

3.7 Axioms for Real and Complex Numbers

This section presents the axioms for real and complex numbers, along with some commentary about them. Analysis textbooks implicitly or explicitly use these axioms or their equivalents as their starting point. In the database `set.mm`, we define real and complex numbers as (rather complicated) specific sets and derive these axioms as *theorems* from the axioms of ZF set theory, using a method called Dedekind cuts. We omit the details of this construction, which you can follow if you wish using the `set.mm` database in conjunction with the textbooks referenced therein.

Once we prove those theorems, we then restate these proven theorems as axioms. This lets us easily identify which axioms are needed for a particular complex number proof, without the obfuscation of the set theory used to derive them. As a result, the construction is actually unimportant other than to show that sets exist that satisfy the axioms, and thus that the axioms are consistent if ZF set theory is consistent. When working with real numbers you can think of them as being the actual sets resulting from the construction (for definiteness), or you can think of them as otherwise unspecified sets that happen to satisfy the axioms. The derivation is not easy, but the fact that it works is quite remarkable and lends support to the idea that ZFC set theory is all we need to provide a foundation for essentially all of mathematics.

3.7.1 The Axioms for Real and Complex Numbers Themselves

For the axioms we are given (or postulate) 8 classes: $\mathbb{C}$ (the set of complex numbers), $\mathbb{R}$ (the set of real numbers, a subset of $\mathbb{C}$), 0 (zero), 1 (one), i (square root of -1), $+$ (plus), $\cdot$ (times), and $<_{\mathbb{R}}$ (less than for just the real numbers). Subtraction and division are defined terms and are not part of the axioms; for their definitions see `set.mm`.

Note that the notation $(A + B)$ (and similarly $(A \cdot B)$) specifies a class called an *operation*, and is the function value of the class $+$ at ordered pair $\langle A, B \rangle$. An operation is defined by statement `df-opr` on p. 84. The notation $A <_{\mathbb{R}} B$ specifies a wff called a *binary relation* and means $\langle A, B \rangle \in <_{\mathbb{R}}$, as defined by statement `df-br` on p. 81.

Our set of 8 given classes is assumed to satisfy the following 22 axioms (in the axioms listed below, $<$ really means $<_{\mathbb{R}}$).

1. The real numbers are a subset of the complex numbers.
   `ax-resscn $p` $\vdash \mathbb{R} \subseteq \mathbb{C}$
2. One is a complex number.
   `ax-1cn $p` $\vdash 1 \in \mathbb{C}$
3. The imaginary number i is a complex number.
   `ax-icn $p` $\vdash i \in \mathbb{C}$
4. Complex numbers are closed under addition.
   `ax-addcl $p` $\vdash ((A \in \mathbb{C} \wedge B \in \mathbb{C}) \rightarrow (A + B) \in \mathbb{C})$
5. Real numbers are closed under addition.
   `ax-addrcl $p` $\vdash ((A \in \mathbb{R} \wedge B \in \mathbb{R}) \rightarrow (A + B) \in \mathbb{R})$
6. Complex numbers are closed under multiplication.
   `ax-mulcl $p` $\vdash ((A \in \mathbb{C} \wedge B \in \mathbb{C}) \rightarrow (A \cdot B) \in \mathbb{C})$
7. Real numbers are closed under multiplication.
   `ax-mulrcl $p` $\vdash ((A \in \mathbb{R} \wedge B \in \mathbb{R}) \rightarrow (A \cdot B) \in \mathbb{R})$
8. Multiplication of complex numbers is commutative.
   `ax-mulcom $p` $\vdash ((A \in \mathbb{C} \wedge B \in \mathbb{C}) \rightarrow (A \cdot B) = (B \cdot A))$
9. Addition of complex numbers is associative.
   `ax-addass $p` $\vdash ((A \in \mathbb{C} \wedge B \in \mathbb{C} \wedge C \in \mathbb{C}) \rightarrow ((A + B) + C) = (A + (B + C)))$
10. Multiplication of complex numbers is associative.
    `ax-mulass $p` $\vdash ((A \in \mathbb{C} \wedge B \in \mathbb{C} \wedge C \in \mathbb{C}) \rightarrow ((A \cdot B) \cdot C) = (A \cdot (B \cdot C)))$
11. Multiplication distributes over addition for complex numbers.
    `ax-distr $p` $\vdash ((A \in \mathbb{C} \wedge B \in \mathbb{C} \wedge C \in \mathbb{C}) \rightarrow (A \cdot (B + C)) = ((A \cdot B) + (A \cdot C)))$
12. The square of i equals -1 (expressed as i-squared plus 1 is 0).
    `ax-i2m1 $p` $\vdash ((i \cdot i) + 1) = 0$
13. One and zero are distinct.
    `ax-1ne0 $p` $\vdash 1 \neq 0$

14. One is an identity element for real multiplication.
`ax-1rid $p` $\vdash (A \in \mathbb{R} \to (A \cdot 1) = A)$
15. Every real number has a negative.
`ax-rnegex $p` $\vdash (A \in \mathbb{R} \to \exists x \in \mathbb{R} (A + x) = 0)$
16. Every nonzero real number has a reciprocal.
`ax-rrecex $p` $\vdash (A \in \mathbb{R} \to (A \neq 0 \to \exists x \in \mathbb{R} (A \cdot x) = 1))$
17. A complex number can be expressed in terms of two reals.
`ax-cnre $p` $\vdash (A \in \mathbb{C} \to \exists x \in \mathbb{R} \exists y \in \mathbb{R} A = (x + (y \cdot i)))$
18. Ordering on reals satisfies strict trichotomy.
`ax-pre-lttri $p` $\vdash ((A \in \mathbb{R} \wedge B \in \mathbb{R}) \to (A < B \leftrightarrow \neg (A = B \vee B < A)))$
19. Ordering on reals is transitive.
`ax-pre-lttrn $p` $\vdash ((A \in \mathbb{R} \wedge B \in \mathbb{R} \wedge C \in \mathbb{R}) \to ((A < B \wedge B < C) \to A < C))$
20. Ordering on reals is preserved after addition to both sides.
`ax-pre-ltadd $p` $\vdash ((A \in \mathbb{R} \wedge B \in \mathbb{R} \wedge C \in \mathbb{R}) \to (A < B \to (C + A) < (C + B)))$
21. The product of two positive reals is positive.
`ax-pre-mulgt0 $p` $\vdash ((A \in \mathbb{R} \wedge B \in \mathbb{R}) \to ((0 < A \wedge 0 < B) \to 0 < (A \cdot B)))$
22. A non-empty, bounded-above set of reals has a supremum.
`ax-pre-sup $p` $\vdash ((A \subseteq \mathbb{R} \wedge A \neq \varnothing \wedge \exists x \in \mathbb{R} \forall y \in A \; y < x) \to \exists x \in \mathbb{R} (\forall y \in A \neg x < y \wedge \forall y \in \mathbb{R} (y < x \to \exists z \in A \; y < z)))$

This completes the set of axioms for real and complex numbers. You may wish to look at how subtraction, division, and decimal numbers are defined in `set.mm`, and for fun look at the proof of $2 + 2 = 4$ (theorem `2p2e4` in `set.mm`) as discussed in section 3.8.

In `set.mm` we define the non-negative integers $\mathbb{N}$, the integers $\mathbb{Z}$, and the rationals $\mathbb{Q}$ as subsets of $\mathbb{R}$. This leads to the nice inclusion $\mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{R} \subseteq \mathbb{C}$, giving us a uniform framework in which, for example, a property such as commutativity of complex number addition automatically applies to integers. The natural numbers $\mathbb{N}$ are different from the set ω we defined earlier, but both satisfy Peano's postulates.

3.7.2 Complex Number Axioms in Analysis Texts

Most analysis texts construct complex numbers as ordered pairs of reals, leading to construction-dependent properties that satisfy these axioms but are not stated in their pure form. (This is also done in `set.mm` but our axioms are extracted from that construction.) Other texts will simply state that $\mathbb{R}$ is a "complete ordered subfield of $\mathbb{C}$," leading to redundant axioms when this phrase is completely expanded out. In fact I have not seen a text with the axioms in the explicit form above. None of these axioms is unique

individually, but this carefully worked out collection of axioms is the result of years of work by the Metamath community.

3.7.3 Eliminating Unnecessary Complex Number Axioms

We once had more axioms for real and complex numbers, but over years of time we (the Metamath community) have found ways to eliminate them (by proving them from other axioms) or weaken them (by making weaker claims without reducing what can be proved). In particular, here are statements that used to be complex number axioms but have since been formally proven (with Metamath) to be redundant:

- $\mathbb{C} \in V$. At one time this was listed as a "complex number axiom." However, this is not properly speaking a complex number axiom, and in any case its proof uses axioms of set theory. Proven redundant by Mario Carneiro on 17-Nov-2014 (see `axcnex`).
- $((A \in \mathbb{C} \wedge B \in \mathbb{C}) \rightarrow (A + B) = (B + A))$. Proved redundant by Eric Schmidt on 19-Jun-2012, and formalized by Scott Fenton on 3-Jan-2013 (see `addcom`).
- $(A \in \mathbb{C} \rightarrow (A + 0) = A)$. Proved redundant by Eric Schmidt on 19-Jun-2012, and formalized by Scott Fenton on 3-Jan-2013 (see `addid1`).
- $(A \in \mathbb{C} \rightarrow \exists x \in \mathbb{C}(A + x) = 0)$. Proved redundant by Eric Schmidt and formalized on 21-May-2007 (see `cnegex`).
- $((A \in \mathbb{C} \wedge A \neq 0) \rightarrow \exists x \in \mathbb{C}(A \cdot x) = 1)$. Proved redundant by Eric Schmidt and formalized on 22-May-2007 (see `recex`).
- $0 \in \mathbb{R}$. Proved redundant by Eric Schmidt on 19-Feb-2005 and formalized 21-May-2007 (see `0re`).

We could eliminate 0 as an axiomatic object by defining it as $((i \cdot i) + 1)$ and replacing it with this expression throughout the axioms. If this is done, axiom ax-i2m1 becomes redundant. However, the remaining axioms would become longer and less intuitive.

Eric Schmidt's paper analyzing this axiom system [61] presented a proof that these remaining axioms, with the possible exception of ax-mulcom, are independent of the others. It is currently an open question if ax-mulcom is independent of the others.

3.8 Two Plus Two Equals Four

Here is a proof that $2+2 = 4$, as proven in the theorem `2p2e4` in the database `set.mm`. This is a useful demonstration of what a Metamath proof can look

like. This proof may have more steps than you're used to, but each step is rigorously proven all the way back to the axioms of logic and set theory. This display was originally generated by the Metamath program as an HTML file.

In the table showing the proof "Step" is the sequential step number, while its associated "Expression" is an expression that we have proved. "Ref" is the name of a theorem or axiom that justifies that expression, and "Hyp" refers to previous steps (if any) that the theorem or axiom needs so that we can use it. Expressions are indented further than the expressions that depend on them to show their interdependencies.

Table 3.1: Two plus two equals four

Step	**Hyp**	**Ref**	**Expression**
1		df-2	$\vdash 2 = 1 + 1$
2	1	oveq2i	$\vdash (2 + 2) = (2 + (1 + 1))$
3		df-4	$\vdash 4 = (3 + 1)$
4		df-3	$\vdash 3 = (2 + 1)$
5	4	oveq1i	$\vdash (3 + 1) = ((2 + 1) + 1)$
6		2cn	$\vdash 2 \in \mathbb{C}$
7		ax-1cn	$\vdash 1 \in \mathbb{C}$
8	6,7,7	addassi	$\vdash ((2 + 1) + 1) = (2 + (1 + 1))$
9	3,5,8	3eqtri	$\vdash 4 = (2 + (1 + 1))$
10	2,9	eqtr4i	$\vdash (2 + 2) = 4$

Step 1 says that we can assert that $2 = 1 + 1$ because it is justified by `df-2`. What is `df-2`? It is simply the definition of 2, which in our system is defined as being equal to $1 + 1$. This shows how we can use definitions in proofs.

Look at Step 2 of the proof. In the Ref column, we see that it references a previously proved theorem, `oveq2i`. It turns out that theorem `oveq2i` requires a hypothesis, and in the Hyp column of Step 2 we indicate that Step 1 will satisfy (match) this hypothesis. If we looked at `oveq2i` we would find that it proves that given some hypothesis $A = B$, we can prove that $(CFA) = (CFB)$. If we use `oveq2i` and apply step 1's result as the hypothesis, that will mean that $A = 2$ and $B = (1 + 1)$ within this use of `oveq2i`. We are free to select any value of C and F (subject to syntax constraints), so we are free to select $C = 2$ and $F = +$, producing our desired result, $(2 + 2) = (2 + (1 + 1))$.

Step 2 is an example of substitution. In the end, every step in every proof uses only this one substitution rule. All the rules of logic, and all the axioms, are expressed so that they can be used via this one substitution rule. So once you master substitution, you can master every Metamath proof, no exceptions.

Each step is clear and can be immediately checked. In the HTML display

you can even click on each reference to see why it is justified, making it easy to see why the proof works.

3.9 Deduction

Strictly speaking, a deduction (also called an inference) is a kind of statement that needs some hypotheses to be true in order for its conclusion to be true. A theorem, on the other hand, has no hypotheses. Informally we often call both of them theorems, but in this section we will stick to the strict definitions.

It sometimes happens that we have proved a deduction of the form $\varphi \Rightarrow \psi$ (given hypothesis φ we can prove ψ) and we want to then prove a theorem of the form $\varphi \rightarrow \psi$.

Converting a deduction (which uses a hypothesis) into a theorem (which does not) is not as simple as you might think. The deduction says, "if we can prove φ then we can prove ψ," which is in some sense weaker than saying "φ implies ψ." There is no axiom of logic that permits us to directly obtain the theorem given the deduction.[10]

This is in contrast to going the other way. If we have the theorem $(\varphi \rightarrow \psi)$, it is easy to recover the deduction $(\varphi \Rightarrow \psi)$ using modus ponens (`ax-mp`; see section 3.3.1).

In the following subsections we first discuss the standard deduction theorem (the traditional but awkward way to convert deductions into theorems) and the weak deduction theorem (a limited version of the standard deduction theorem that is easier to use and was once widely used in `set.mm`). In section 3.9.3 we discuss deduction style, the newer approach we now recommend in most cases. Deduction style uses "deduction form," a form that prefixes each hypothesis (other than definitions) and the conclusion with a universal antecedent ("$\varphi \rightarrow$"). Deduction style is widely used in `set.mm`, so it is useful to understand it and *why* it is widely used. Section 3.9.4 briefly discusses our approach for using natural deduction within `set.mm`, as that approach is deeply related to deduction style. We conclude with a summary of the strengths of our approach, which we believe are compelling.

3.9.1 The Standard Deduction Theorem

It is possible to make use of information contained in the deduction or its proof to assist us with the proof of the related theorem. In traditional logic books, there is a metatheorem called the Deduction Theorem, discovered independently by Herbrand and Tarski around 1930. The Deduction Theorem,

[10]The conversion of a deduction to a theorem does not even hold in general for quantum propositional calculus, which is a weak subset of classical propositional calculus. It has been shown that adding the Standard Deduction Theorem (discussed below) to quantum propositional calculus turns it into classical propositional calculus!

which we often call the Standard Deduction Theorem, provides an algorithm for constructing a proof of a theorem from the proof of its corresponding deduction. See, for example, [39, p. 56]. To construct a proof for a theorem, the algorithm looks at each step in the proof of the original deduction and rewrites the step with several steps wherein the hypothesis is eliminated and becomes an antecedent.

In ordinary mathematics, no one actually carries out the algorithm, because (in its most basic form) it involves an exponential explosion of the number of proof steps as more hypotheses are eliminated. Instead, the Standard Deduction Theorem is invoked simply to claim that it can be done in principle, without actually doing it. What's more, the algorithm is not as simple as it might first appear when applying it rigorously. There is a subtle restriction on the Standard Deduction Theorem that must be taken into account involving the axiom of generalization when working with predicate calculus (see the literature for more detail).

One of the goals of Metamath is to let you plainly see, with as few underlying concepts as possible, how mathematics can be derived directly from the axioms, and not indirectly according to some hidden rules buried inside a program or understood only by logicians. If we added the Standard Deduction Theorem to the language and proof verifier, that would greatly complicate both and largely defeat Metamath's goal of simplicity. In principle, we could show direct proofs by expanding out the proof steps generated by the algorithm of the Standard Deduction Theorem, but that is not feasible in practice because the number of proof steps quickly becomes huge, even astronomical. Since the algorithm of the Standard Deduction Theorem is driven by the proof, we would have to go through that proof all over again—starting from axioms—in order to obtain the theorem form. In terms of proof length, there would be no savings over just proving the theorem directly instead of first proving the deduction form.

3.9.2 Weak Deduction Theorem

We have developed a more efficient method for proving a theorem from a deduction that can be used instead of the Standard Deduction Theorem in many (but not all) cases. We call this more efficient method the Weak Deduction Theorem.[11] Unlike the Standard Deduction Theorem, the Weak Deduction Theorem produces the theorem directly from a special substitution instance of the deduction, using a small, fixed number of steps roughly proportional to the length of the final theorem.

If you come to a proof referencing the Weak Deduction Theorem `dedth` (or one of its variants `dedthxx`), here is how to follow the proof without getting into the details: just click on the theorem referenced in the step just

[11] There is also an unrelated "Weak Deduction Theorem" in the field of relevance logic, so to avoid confusion we could call ours the "Weak Deduction Theorem for Classical Logic."

before the reference to `dedth` and ignore everything else. Theorem `dedth` simply turns a hypothesis into an antecedent (i.e. the hypothesis followed by $\rightarrow$ is placed in front of the assertion, and the hypothesis itself is eliminated) given certain conditions.

The Weak Deduction Theorem eliminates a hypothesis φ, making it become an antecedent. It does this by proving an expression $\varphi \rightarrow \psi$ given two hypotheses: (1) $(A = \text{if}(\varphi, A, B) \rightarrow (\varphi \leftrightarrow \chi))$ and (2) χ. Note that it requires that a proof exists for φ when the class variable A is replaced with a specific class B. The hypothesis χ should be assigned to the inference. You can see the details of the proof of the Weak Deduction Theorem in theorem `dedth`.

The Weak Deduction Theorem is probably easier to understand by studying proofs that make use of it. For example, let's look at the proof of `renegcl`, which proves that $\vdash (A \in \mathbb{R} \rightarrow -A \in \mathbb{R})$:

Step	**Hyp**	**Ref**	**Expression**
1		negeq	$\vdash (A = \text{if} (A \in \mathbb{R} , A , 1) \rightarrow \text{-} A = \text{-} \text{if} (A \in \mathbb{R} , A , 1))$
2	1	eleq1d	$\vdash (A = \text{if} (A \in \mathbb{R} , A , 1) \rightarrow (\text{-} A \in \mathbb{R} \leftrightarrow \text{-} \text{if} (A \in \mathbb{R} , A , 1) \in \mathbb{R}))$
3		1re	$\vdash 1 \in \mathbb{R}$
4	3	elimel	$\vdash \text{if}(A \in \mathbb{R}, A, 1) \in \mathbb{R}$
5	4	renegcli	$\vdash \text{-if}(A \in \mathbb{R}, A, 1) \in \mathbb{R}$
6	2,5	dedth	$\vdash (A \in \mathbb{R} \rightarrow \text{-}A \in \mathbb{R})$

The somewhat strange-looking steps in `renegcl` before step 5 are technical stuff that makes this magic work, and they can be ignored for a quick overview of the proof. To continue following the "important" part of the proof of `renegcl`, you can look at the reference to `renegcli` at step 5.

That said, let's briefly look at how `renegcl` uses the Weak Deduction Theorem (`dedth`) to do its job, in case you want to do something similar or want understand it more deeply. Let's work backwards in the proof of `renegcl`. Step 6 applies `dedth` to produce our goal result $\vdash (A \in \mathbb{R} \rightarrow -A \in \mathbb{R})$. This requires on the one hand the (substituted) deduction `renegcli` in step 5. By itself `renegcli` proves the deduction $\vdash A \in \mathbb{R} \Rightarrow \vdash -A \in \mathbb{R}$; this is the deduction form we are trying to turn into theorem form, and thus `renegcli` has a separate hypothesis that must be fulfilled. To fulfill the hypothesis of the invocation of `renegcli` in step 5, it is eventually reduced to the already proven theorem $1 \in \mathbb{R}$ in step 3. Step 4 connects steps 3 and 5; step 4 invokes `elimel`, a special case of `elimhyp` that eliminates a membership hypothesis for the weak deduction theorem. On the other hand, the equivalence of the conclusion of `renegcl` ($-A \in \mathbb{R}$) and the substituted conclusion of `renegcli` must be proven, which is done in steps 2 and 1.

The weak deduction theorem has limitations. In particular, we must be

able to prove a special case of the deduction's hypothesis as a stand-alone theorem. For example, we used $1 \in \mathbb{R}$ in step 3 of `renegcl`.

We used to use the weak deduction theorem extensively within `set.mm`. However, we now recommend applying "deduction style" instead in most cases, as deduction style is often an easier and clearer approach. Therefore, we will now describe deduction style.

3.9.3 Deduction Style

We now prefer to write assertions in "deduction form" instead of writing a proof that would require use of the standard or weak deduction theorem. We call this appraoch "deduction style."

It will be easier to explain this by first defining some terms:

- **closed form**: A kind of assertion (theorem) with no hypotheses. Typically its label has no special suffix. An example is `unss`, which states: $\vdash ((A \subseteq C \wedge B \subseteq C) \leftrightarrow (A \cup B) \subseteq C)$
- **deduction form**: A kind of assertion with one or more hypotheses where the conclusion is an implication with a wff variable as the antecedent (usually φ), and every hypothesis ($e statement) is either (1) an implication with the same antecedent as the conclusion or (2) a definition. A definition can be for a class variable (this is a class variable followed by "=") or a wff variable (this is a wff variable followed by $\leftrightarrow$); class variable definitions are more common. In practice, a proof in deduction form will also contain many steps that are implications where the antecedent is either that wff variable (normally φ) or is a conjunction (...$\wedge$...) including that wff variable (φ). If an assertion is in deduction form, and other forms are also available, then we suffix its label with "d." An example is `unssd`, which states[12]: $\vdash (\varphi \rightarrow A \subseteq C) \quad \& \quad \vdash (\varphi \rightarrow B \subseteq C) \quad \Rightarrow \quad \vdash (\varphi \rightarrow (A \cup B) \subseteq C)$
- **inference form**: A kind of assertion with one or more hypotheses that is not in deduction form (e.g., there is no common antecedent). If an assertion is in inference form, and other forms are also available, then we suffix its label with "i." An example is `unssi`, which states: $\vdash A \subseteq C \quad \& \quad \vdash B \subseteq C \quad \Rightarrow \quad \vdash (A \cup B) \subseteq C$

When using deduction style we express an assertion in deduction form. This form prefixes each hypothesis (other than definitions) and the conclusion with a universal antecedent ("$\varphi \rightarrow$"). The antecedent (e.g., φ) mimics the

[12]For brevity we show here (and in other places) a & between hypotheses and a $\Rightarrow$ between the hypotheses and the conclusion. This notation is technically not part of the Metamath language, but is instead a convenient abbreviation to show both the hypotheses and conclusion.

context handled in the deduction theorem, eliminating the need to directly use the deduction theorem.

Once you have an assertion in deduction form, you can easily convert it to inference form or closed form:

- To prove some assertion Ti in inference form, given assertion Td in deduction form, there is a simple mechanical process you can use. First take each Ti hypothesis and insert a `T.` $\rightarrow$ prefix ("true implies") using `a1i`. You can then use the existing assertion Td to prove the resulting conclusion with a `T.` $\rightarrow$ prefix. Finally, you can remove that prefix using `trud`, resulting in the conclusion you wanted to prove.

- To prove some assertion T in closed form, given assertion Td in deduction form, there is another simple mechanical process you can use. First, select an expression that is the conjunction (...$\wedge$...) of all of the consequents of every hypothesis of Td. Next, prove that this expression implies each of the separate hypotheses of Td in turn by eliminating conjuncts (there are a variety of proven assertions to do this, including `simpl`, `simpr`, `3simpa`, `3simpb`, `3simpc`, `simp1`, `simp2`, and `simp3`). If the expression has nested conjunctions, inner conjuncts can be broken out by chaining the above theorems with `syl` (see section 3.6).[13] As your final step, you can then apply the already-proven assertion Td (which is in deduction form), proving assertion T in closed form.

We can also easily convert any assertion T in closed form to its related assertion Ti in inference form by applying modus ponens (see section 3.3.1).

The deduction form antecedent can also be used to represent the context necessary to support natural deduction systems, so we will now discuss natural deduction.

3.9.4 Natural Deduction

Natural deduction (ND) systems, as such, were originally introduced in 1934 by two logicians working independently: Jaśkowski and Gentzen. ND systems are supposed to reconstruct, in a formally proper way, traditional ways of mathematical reasoning (such as conditional proof, indirect proof, and proof by cases). As reconstructions they were naturally influenced by previous work, and many specific ND systems and notations have been developed since their original work.

There are many ND variants, but Indrzejczak [27, p. 31-32] suggests that any natural deductive system must satisfy at least these three criteria:

[13] There are actually many theorems (labeled simp* such as `simp333`) that break out inner conjuncts in one step, but rather than learning them you can just use the chaining we just described to prove them, and then let the Metamath program command `minimize_with` figure out the right ones needed to collapse them.

- "There are some means for entering assumptions into a proof and also for eliminating them. Usually it requires some bookkeeping devices for indicating the scope of an assumption, and showing that a part of a proof depending on eliminated assumption is discharged.

- There are no (or, at least, a very limited set of) axioms, because their role is taken over by the set of primitive rules for introduction and elimination of logical constants which means that elementary inferences instead of formulae are taken as primitive.

- (A genuine) ND system admits a lot of freedom in proof construction and possibility of applying several proof search strategies, like conditional proof, proof by cases, proof by reductio ad absurdum etc."

The Metamath Proof Explorer (MPE) as defined in `set.mm` is fundamentally a Hilbert-style system. That is, MPE is based on a larger number of axioms (compared to natural deduction systems), a very small set of rules of inference (modus ponens), and the context is not changed by the rules of inference in the middle of a proof. That said, MPE proofs can be developed using the natural deduction (ND) approach as originally developed by Jaśkowski and Gentzen.

The most common and recommended approach for applying ND in MPE is to use deduction form and apply the MPE proven assertions that are equivalent to ND rules. For example, MPE's `jca` is equivalent to ND rule $\wedge$-I (and-insertion). We maintain a list of equivalences that you may consult. This approach for applying an ND approach within MPE relies on Metamath's wff metavariables in an essential way, and is described in more detail in the presentation "Natural Deductions in the Metamath Proof Language" by Mario Carneiro [11].

In this style many steps are an implication, whose antecedent mimics the context (Γ) of most ND systems. To add an assumption, simply add it to the implication antecedent (typically using `simpr`), and use that new antecedent for all later claims in the same scope. If you wish to use an assertion in an ND hypothesis scope that is outside the current ND hypothesis scope, modify the assertion so that the ND hypothesis assumption is added to its antecedent (typically using `adantr`). Most proof steps will be proved using rules that have hypotheses and results of the form $\varphi \rightarrow ...$

An example may make this clearer. Let's show theorem 5.5 of [34, p. 18] along with a line by line translation using the usual translation of natural deduction (ND) in the Metamath Proof Explorer (MPE) notation (this is proof `ex-natded5.5`). The proof's original goal was to prove $\neg\psi$ given two hypotheses, $(\psi \rightarrow \chi)$ and $\neg\chi$. We will translate these statements into MPE deduction form by prefixing them all with $\varphi \rightarrow$. As a result, in MPE the goal is stated as $(\varphi \rightarrow \neg\psi)$, and the two hypotheses are stated as $(\varphi \rightarrow (\psi \rightarrow \chi))$ and $(\varphi \rightarrow \neg\chi)$.

The following table shows the proof in Fitch natural deduction style and its MPE equivalent. The # column shows the original numbering, *MPE#* shows the number in the equivalent MPE proof (which we will show later), *ND Expression* shows the original proof claim in ND notation, and *MPE Translation* shows its translation into MPE as discussed in this section. The final columns show the rationale in ND and MPE respectively.

#	MPE#	ND Expression	MPE Translation	ND Rationale	MPE Rationale
1	2;3	$(\psi \to \chi)$	$(\varphi \to (\psi \to \chi))$	Given	\$e; `adantr` to put in ND hypothesis
2	5	$\neg\chi$	$(\varphi \to \neg\chi)$	Given	\$e; `adantr` to put in ND hypothesis
3	1	... | ψ	$(\varphi \to \psi)$	ND hypothesis assumption	`simpr`
4	4	... χ	$((\varphi \wedge \psi) \to \chi)$	$\to$E 1,3	`mpd` 1,3
5	6	... $\neg\chi$	$((\varphi \wedge \psi) \to \neg\chi)$	IT 2	`adantr` 5
6	7	$\neg\psi$	$(\varphi \to \neg\psi)$	$\wedge$I 3,4,5	`pm2.65da` 4,6

The original used Latin letters; we have replaced them with Greek letters to follow Metamath naming conventions and so that it is easier to follow the Metamath translation. The Metamath line-for-line translation of this natural deduction approach precedes every line with an antecedent including φ and uses the Metamath equivalents of the natural deduction rules. To add an assumption, the antecedent is modified to include it (typically by using `adantr`; `simpr` is useful when you want to depend directly on the new assumption, as is shown here).

In Metamath we can represent the two given statements as these hypotheses:

- ex-natded5.5.1 $\vdash (\varphi \to (\psi \to \chi))$
- ex-natded5.5.2 $\vdash (\varphi \to \neg\chi)$

Here is the proof in Metamath as a line-by-line translation:

Step	Hyp	Ref	Expression
1		simpr	$\vdash ((\varphi \wedge \psi) \rightarrow \psi)$
2		ex-natded5.5.1	$\vdash (\varphi \rightarrow (\psi \rightarrow \chi))$
3	2	adantr	$\vdash ((\varphi \wedge \psi) \rightarrow (\psi \rightarrow \chi))$
4	1, 3	mpd	$\vdash ((\varphi \wedge \psi) \rightarrow \chi)$
5		ex-natded5.5.2	$\vdash (\varphi \rightarrow \neg\chi)$
6	5	adantr	$\vdash ((\varphi \wedge \psi) \rightarrow \neg\chi)$
7	4, 6	pm2.65da	$\vdash (\varphi \rightarrow \neg\psi)$

Only using specific natural deduction rules directly can lead to very long proofs, for exactly the same reason that only using axioms directly in Hilbert-style proofs can lead to very long proofs. If the goal is short and clear proofs, then it is better to reuse already-proven assertions in deduction form than to start from scratch each time and using only basic natural deduction rules.

3.9.5 Strengths of Our Approach

As far as we know there is nothing else in the literature like either the weak deduction theorem or Mario Carneiro's natural deduction method. In order to transform a hypothesis into an antecedent, the literature's standard "Deduction Theorem" requires metalogic outside of the notions provided by the axiom system. We instead generally prefer to use Mario Carneiro's natural deduction method, then use the weak deduction theorem in cases where that is difficult to apply, and only then use the full standard deduction theorem as a last resort.

The weak deduction theorem does not require any additional metalogic but converts an inference directly into a closed form theorem, with a rigorous proof that uses only the axiom system. Unlike the standard Deduction Theorem, there is no implicit external justification that we have to trust in order to use it.

Mario Carneiro's natural deduction method also does not require any new metalogical notions. It avoids the Deduction Theorem's metalogic by prefixing the hypotheses and conclusion of every would-be inference with a universal antecedent ("$\varphi \rightarrow$") from the very start.

We think it is impressive and satisfying that we can do so much in a practical sense without stepping outside of our Hilbert-style axiom system. Of course our axiomatization, which is in the form of schemes, contains a metalogic of its own that we exploit. But this metalogic is relatively simple, and for our Deduction Theorem alternatives, we primarily use just the direct substitution of expressions for metavariables.

3.10 Exploring the Set Theory Database

At this point you may wish to study the `set.mm` file in more detail. Pay particular attention to the assumptions needed to define wffs (which are not included above), the variable types (`$f` statements), and the definitions that are introduced. Start with some simple theorems in propositional calculus, making sure you understand in detail each step of a proof. Once you get past the first few proofs and become familiar with the Metamath language, any part of the `set.mm` database will be as easy to follow, step by step, as any other part—you won't have to undergo a "quantum leap" in mathematical sophistication to be able to follow a deep proof in set theory.

Next, you may want to explore how concepts such as natural numbers are defined and described. This is probably best done in conjunction with standard set theory textbooks, which can help give you a higher-level understanding. The `set.mm` database provides references that will get you started. From there, you will be on your way towards a very deep, rigorous understanding of abstract mathematics.

The Metamath program can help you peruse a Metamath database, whether you are trying to figure out how a certain step follows in a proof or just have a general curiosity. We will go through some examples of the commands, using the `set.mm` database provided with the Metamath software. These should help get you started. See Chapter 5 for a more detailed description of the commands. Note that we have included the full spelling of all commands to prevent ambiguity with future commands. In practice you may type just the characters needed to specify each command keyword unambiguously, often just one or two characters per keyword, and you don't need to type them in upper case.

First run the Metamath program as described earlier. You should see the `MM>` prompt. Read in the `set.mm` file:

```
MM> read set.mm
Reading source file "set.mm"... 34554442 bytes
34554442 bytes were read into the source buffer.
The source has 155711 statements; 2254 are $a and 32250 are $p.
No errors were found.  However, proofs were not checked.
Type VERIFY PROOF * if you want to check them.
```

As with most examples in this book, what you will see will be slightly different because we are continuously improving our databases (including `set.mm`).

Let's check the database integrity. This may take a minute or two to run if your computer is slow.

```
MM> verify proof *
0 10%  20%  30%  40%  50%  60%  70%  80%  90% 100%
..............................................
All proofs in the database were verified in 2.84 s.
```

No errors were reported, so every proof is correct.

You need to know the names (labels) of theorems before you can look at them. Often just examining the database file(s) with a text editor is the best approach. In `set.mm` there are many detailed comments, especially near the beginning, that can help guide you. The `search` command in the Metamath program is also handy. The `comments` qualifier will list the statements whose associated comment (the one immediately before it) contain a string you give it. For example, if you are studying Enderton's *Elements of Set Theory* [18] you may want to see the references to it in the database. The search string `enderton` is not case sensitive. (This will not show you all the database theorems that are in Enderton's book because there is usually only one citation for a given theorem, which may appear in several textbooks.)

```
MM> search * "enderton" / comments
12067 unineq $p "... Exercise 20 of [Enderton] p. 32 and ..."
12459 undif2 $p "...Corollary 6K of [Enderton] p. 144. (C..."
12953 df-tp $a "...s. Definition of [Enderton] p. 19. (Co..."
13689 unissb $p ".... Exercise 5 of [Enderton] p. 26 and ..."
```

(etc.)

Or you may want to see what theorems have something to do with conjunction (logical AND). The quotes around the search string are optional when there's no ambiguity.

```
MM> search * conjunction / comments
120 a1d $p "...be replaced with a conjunction ( ~ df-an )..."
662 df-bi $a "...viated form after conjunction is introdu..."
1319 wa $a "...ff definition to include conjunction ('and')."
1321 df-an $a "Define conjunction (logical 'and'). Defini..."
1420 imnan $p "...tion in terms of conjunction. (Contribu..."
```

(etc.)

Now we will start to look at some details. Let's look at the first axiom of propositional calculus (we could use `sh st` to abbreviate `show statement`).

```
MM> show statement ax-1/full
Statement 19 is located on line 881 of the file "set.mm".
"Axiom _Simp_.  Axiom A1 of [Margaris] p. 49.  One of the 3
axioms of propositional calculus.  The 3 axioms are also
        ...
19 ax-1 $a |- ( ph -> ( ps -> ph ) ) $.
Its mandatory hypotheses in RPN order are:
  wph $f wff ph $.
  wps $f wff ps $.
```

```
The statement and its hypotheses require the variables:  ph
      ps
The variables it contains are:  ph ps


Statement 49 is located on line 11182 of the file "set.mm".
Its statement number for HTML pages is 6.
"Axiom _Simp_.  Axiom A1 of [Margaris] p. 49.  One of the 3
axioms of propositional calculus.  The 3 axioms are also
given as Definition 2.1 of [Hamilton] p. 28.
...
49 ax-1 $a |- ( ph -> ( ps -> ph ) ) $.
Its mandatory hypotheses in RPN order are:
  wph $f wff ph $.
  wps $f wff ps $.
The statement and its hypotheses require the variables:
  ph ps
The variables it contains are:  ph ps
```

Compare this to `ax-1` on p. 70. You can see that the symbol `ph` is the ASCII notation for φ, etc. To see the mathematical symbols for any expression you may typeset it in LaTeX (type `help tex` for instructions) or, easier, just use a text editor to look at the comments where symbols are first introduced in `set.mm`. The hypotheses `wph` and `wps` required by `ax-1` mean that variables `ph` and `ps` must be wffs.

Next we'll pick a simple theorem of propositional calculus, the Principle of Identity, which is proved directly from the axioms. We'll look at the statement then its proof.

```
MM> show statement id1/full
Statement 116 is located on line 11371 of the file "set.mm".
Its statement number for HTML pages is 22.
"Principle of identity.  Theorem *2.08 of [WhiteheadRussell]
p. 101.  This version is proved directly from the axioms for
demonstration purposes.
...
116 id1 $p |- ( ph -> ph ) $= ... $.
Its mandatory hypotheses in RPN order are:
  wph $f wff ph $.
Its optional hypotheses are:  wps wch wth wta wet
      wze wsi wrh wmu wla wka
The statement and its hypotheses require the variables:  ph
These additional variables are allowed in its proof:
      ps ch th ta et ze si rh mu la ka
The variables it contains are:  ph
```

The optional variables `ps`, `ch`, etc. are available for use in a proof of this statement if we wish, and were we to do so we would make use of optional hypotheses `wps`, `wch`, etc. (See Section 4.2.5 for the meaning of "optional hypothesis.") The reason these show up in the statement display is that statement `id1` happens to be in their scope (see Section 4.2.8 for the definition of "scope"), but in fact in propositional calculus we will never make use of optional hypotheses or variables. This becomes important after quantifiers are introduced, where "dummy" variables are often needed in the middle of a proof.

Let's look at the proof of statement `id1`. We'll use the `show proof` command, which by default suppresses the "non-essential" steps that construct the wffs. We will display the proof in "lemmon' format (a non-indented format with explicit previous step number references) and renumber the displayed steps:

```
MM> show proof id1 /lemmon/renumber
1 ax-1           $a |- ( ph -> ( ph -> ph ) )
2 ax-1           $a |- ( ph -> ( ( ph -> ph ) -> ph ) )
3 ax-2           $a |- ( ( ph -> ( ( ph -> ph ) -> ph ) ) ->
                     ( ( ph -> ( ph -> ph ) ) -> ( ph -> ph )
                                                        ) )
4 2,3 ax-mp      $a |- ( ( ph -> ( ph -> ph ) ) -> ( ph -> ph
                                                        ) )
5 1,4 ax-mp      $a |- ( ph -> ph )
```

If you have read Section 2.3, you'll know how to interpret this proof. Step 2, for example, is an application of axiom `ax-1`. This proof is identical to the one in Hamilton's *Logic for Mathematicians* [21, p. 32].

You may want to look at what substitutions are made into `ax-1` to arrive at step 2. The command to do this needs to know the "real" step number, so we'll display the proof again without the `renumber` qualifier.

```
MM> show proof id1 /lemmon
 9 ax-1          $a |- ( ph -> ( ph -> ph ) )
20 ax-1          $a |- ( ph -> ( ( ph -> ph ) -> ph ) )
24 ax-2          $a |- ( ( ph -> ( ( ph -> ph ) -> ph ) ) ->
                     ( ( ph -> ( ph -> ph ) ) -> ( ph -> ph )
                                                        ) )
25 20,24 ax-mp   $a |- ( ( ph -> ( ph -> ph ) ) -> ( ph -> ph
                                                        ) )
26 9,25 ax-mp    $a |- ( ph -> ph )
```

The "real" step number is 20. Let's look at its details.

```
MM> show proof id1 /detailed_step 20
Proof step 20:  min=ax-1 $a |- ( ph -> ( ( ph -> ph ) -> ph )
```

```
  )
This step assigns source "ax-1" ($a) to target "min" ($e).
The source assertion requires the hypotheses "wph" ($f, step
18) and "wps" ($f, step 19).  The parent assertion of the
target hypothesis is "ax-mp" ($a, step 25).
The source assertion before substitution was:
    ax-1 $a |- ( ph -> ( ps -> ph ) )
The following substitutions were made to the source
assertion:
    Variable  Substituted with
     ph        ph
     ps        ( ph -> ph )
The target hypothesis before substitution was:
    min $e |- ph
The following substitution was made to the target hypothesis:
    Variable  Substituted with
     ph        ( ph -> ( ( ph -> ph ) -> ph ) )
```

This shows the substitutions made to the variables in `ax-1`. References are made to steps 18 and 19 which are not shown in our proof display. To see these steps, you can display the proof with the `all` qualifier.

Let's look at a slightly more advanced proof of propositional calculus. Note that `/\` is the symbol for $\wedge$ (logical AND, also called conjunction).

```
MM> show statement prth/full
Statement 1791 is located on line 15503 of the file "set.mm".
Its statement number for HTML pages is 559.
"Conjoin antecedents and consequents of two premises.  This
is the closed theorem form of ~ anim12d .  Theorem *3.47 of
[WhiteheadRussell] p. 113.  It was proved by Leibniz,
and it evidently pleased him enough to call it
_praeclarum theorema_ (splendid theorem).
...
1791 prth $p |- ( ( ( ph -> ps ) /\ ( ch -> th ) ) -> ( ( ph
      /\ ch ) -> ( ps /\ th ) ) ) $= ... $.
Its mandatory hypotheses in RPN order are:
  wph $f wff ph $.
  wps $f wff ps $.
  wch $f wff ch $.
  wth $f wff th $.
Its optional hypotheses are:  wta wet wze wsi wrh wmu wla wka
The statement and its hypotheses require the variables:  ph
      ps ch th
These additional variables are allowed in its proof:  ta et
      ze si rh mu la ka
```

```
The variables it contains are:  ph ps ch th


MM> show proof prth /lemmon/renumber
1 simpl          $p |- ( ( ( ph -> ps ) /\ ( ch -> th ) ) ->
                                                ( ph -> ps ) )
2 simpr          $p |- ( ( ( ph -> ps ) /\ ( ch -> th ) ) ->
                                                ( ch -> th ) )
3 1,2 anim12d    $p |- ( ( ( ph -> ps ) /\ ( ch -> th ) ) ->
                           ( ( ph /\ ch ) -> ( ps /\ th ) ) )
```

There are references to a lot of unfamiliar statements. To see what they are, you may type the following:

```
MM> show proof prth /statement_summary
Summary of statements used in the proof of "prth":

Statement simpl is located on line 14748 of the file
"set.mm".
"Elimination of a conjunct.  Theorem *3.26 (Simp) of
[WhiteheadRussell] p. 112. ..."
  simpl $p |- ( ( ph /\ ps ) -> ph ) $= ... $.

Statement simpr is located on line 14777 of the file
"set.mm".
"Elimination of a conjunct.  Theorem *3.27 (Simp) of
[WhiteheadRussell] ..."
  simpr $p |- ( ( ph /\ ps ) -> ps ) $= ... $.

Statement anim12d is located on line 15445 of the file
"set.mm".
"Conjoin antecedents and consequents in a deduction.
..."
  anim12d.1 $e |- ( ph -> ( ps -> ch ) ) $.
  anim12d.2 $e |- ( ph -> ( th -> ta ) ) $.
  anim12d $p |- ( ph -> ( ( ps /\ th ) -> ( ch /\ ta ) ) )
      $= ... $.
```

(etc.)

Of course you can look at each of these statements and their proofs, and so on, back to the axioms of propositional calculus if you wish.

The `search` command is useful for finding statements when you know all or part of their contents. The following example finds all statements containing `ph -> ps` followed by `ch -> th`. The `$*` is a wildcard that matches anything; the `$` before the `*` prevents conflicts with math symbol

token names. The * after SEARCH is also a wildcard that in this case means "match any label."

```
MM> search * "ph -> ps $* ch -> th"
1791 prth $p |- ( ( ( ph -> ps ) /\ ( ch -> th ) ) -> ( ( ph
    /\ ch ) -> ( ps /\ th ) ) )
2455 pm3.48 $p |- ( ( ( ph -> ps ) /\ ( ch -> th ) ) -> ( (
    ph \/ ch ) -> ( ps \/ th ) ) )
117859 pm11.71 $p |- ( ( E. x ph /\ E. y ch ) -> ( ( A. x (
    ph -> ps ) /\ A. y ( ch -> th ) ) <-> A. x A. y ( ( ph /\
    ch ) -> ( ps /\ th ) ) ) )
```

Three statements, prth, pm3.48, and pm11.71, were found to match.

To see what axioms and definitions prth ultimately depends on for its proof, you can have the program backtrack through the hierarchy of theorems and definitions.

```
MM> show trace_back prth /essential/axioms
Statement "prth" assumes the following axioms ($a
statements):
  ax-1 ax-2 ax-3 ax-mp df-bi df-an
```

Note that the 3 axioms of propositional calculus and the modus ponens rule are needed (as expected); in addition, there are a couple of definitions that are used along the way. Note that Metamath makes no distinction between axioms and definitions. In set.mm they have been distinguished artificially by prefixing their labels with ax- and df- respectively. For example, df-an defines conjunction (logical AND), which is represented by the symbol /\. Section 4.5 discusses the philosophy of definitions, and the Metamath language takes a particularly simple, conservative approach by using the $a statement for both axioms and definitions.

You can also have the program compute how many steps a proof has if we were to follow it all the way back to $a statements.

```
MM> show trace_back prth /essential/count_steps
The statement's actual proof has 3 steps.  Backtracking, a
total of 79 different subtheorems are used.  The statement
and subtheorems have a total of 274 actual steps.  If
subtheorems used only once were eliminated, there would be a
total of 38 subtheorems, and the statement and subtheorems
would have a total of 185 steps.  The proof would have 28349
steps if fully expanded back to axiom references.  The
maximum path length is 38.  A longest path is:  prth <-
anim12d <- syl2and <- sylan2d <- ancomsd <- ancom <- pm3.22
<- pm3.21 <- pm3.2 <- ex <- sylbir <- biimpri <- bicomi <-
bicom1 <- bi2 <- dfbi1 <- impbii <- bi3 <- simprim <- impi <-
```

```
con1i <- nsyl2 <- mt3d <- con1d <- notnot1 <- con2i <- nsyl3
<- mt2d <- con2d <- notnot2 <- pm2.18d <- pm2.18 <- pm2.21 <-
pm2.21d <- a1d <- syl <- mpd <- a2i <- a2i.1 .
```

This tells us that we would have to inspect 274 steps if we want to verify the proof completely starting from the axioms. A few more statistics are also shown. There are one or more paths back to axioms that are the longest; this command ferrets out one of them and shows it to you. There may be a sense in which the longest path length is related to how "deep" the theorem is.

We might also be curious about what proofs depend on the theorem `prth`. If it is never used later on, we could eliminate it as redundant if it has no intrinsic interest by itself.

```
MM> show usage prth
Statement "prth" is directly referenced in the proofs of 18
statements:
  mo3 mo0LD 2mo 2mo0LD euind reuind reuss2 reusv3i opelopabt
  wemaplem2 rexanre rlimcn2 o1of2 o1rlimmul 2sqlem6 spanuni
  heicant pm11.71
```

Thus `prth` is directly used by 18 proofs. We can use the `/recursive` qualifier to include indirect use:

```
MM> show usage prth /recursive
Statement "prth" directly or indirectly affects the proofs of
24214 statements:
  mo3 mo mo30LD eu2 mo0LD eu20LD eu30LD mo4f mo4 eu4 mopick
...
```

3.10.1 A Note on the "Compact" Proof Format

The Metamath program will display proofs in a "compact" format whenever the proof is stored in compressed format in the database. It may be be slightly confusing unless you know how to interpret it. For example, if you display the complete proof of theorem `id1` it will start off as follows:

```
MM> show proof id1 /lemmon/all
 1 wph           $f wff ph
 2 wph           $f wff ph
 3 wph           $f wff ph
 4 2,3 wi    @4: $a wff ( ph -> ph )
 5 1,4 wi    @5: $a wff ( ph -> ( ph -> ph ) )
 6 @4            $a wff ( ph -> ph )
```

etc.

Step 4 has a "local label," `@4`, assigned to it. Later on, at step 6, the label `@1` is referenced instead of displaying the explicit proof for that step. This technique takes advantage of the fact that steps in a proof often repeat, especially during the construction of wffs. The compact format reduces the number of steps in the proof display and may be preferred by some people.

If you want to see the normal format with the "true" step numbers, you can use the following workaround:

```
MM> save proof id1 /normal
The proof of "id1" has been reformatted and saved internally.
Remember to use WRITE SOURCE to save it permanently.
MM> show proof id1 /lemmon/all
 1 wph           $f wff ph
 2 wph           $f wff ph
 3 wph           $f wff ph
 4 2,3 wi        $a wff ( ph -> ph )
 5 1,4 wi        $a wff ( ph -> ( ph -> ph ) )
 6 wph           $f wff ph
 7 wph           $f wff ph
 8 6,7 wi        $a wff ( ph -> ph )
```

etc.

Note that the original 6 steps are now 8 steps. However, the format is now the same as that described in Chapter 2.

Chapter 4

The Metamath Language

> *Thus mathematics may be defined as the subject in which we never know what we are talking about, nor whether what we are saying is true.*
>
> Bertrand Russell[1]

Probably the most striking feature of the Metamath language is its almost complete absence of hard-wired syntax. Metamath does not understand any mathematics or logic other than that needed to construct finite sequences of symbols according to a small set of simple, built-in rules. The only rule it uses in a proof is the substitution of an expression (symbol sequence) for a variable, subject to a simple constraint to prevent bound-variable clashes. The primitive notions built into Metamath involve the simple manipulation of finite objects (symbols) that we as humans can easily visualize and that computers can easily deal with. They seem to be just about the simplest notions possible that are required to do standard mathematics.

This chapter serves as a reference manual for the Metamath language. It covers the tedious technical details of the language, some of which you may wish to skim in a first reading. On the other hand, you should pay close attention to the defined terms in **boldface**; they have precise meanings that are important to keep in mind for later understanding. It may be best to first become familiar with the examples in Chapter 2 to gain some motivation for the language.

If you have some knowledge of set theory, you may wish to study this chapter in conjunction with the formal set-theoretical description of the Metamath language in Appendix C.

We will use the name "Metamath" to mean either the Metamath computer language or the Metamath software associated with the computer language. We will not distinguish these two when the context is clear.

[1] [59, p. 84].

The next section contains the complete specification of the Metamath language. It serves as an authoritative reference and presents the syntax in enough detail to write a parser and proof verifier. The specification is terse and it is probably hard to learn the language directly from it, but we include it here for those impatient people who prefer to see everything up front before looking at verbose expository material. Later sections explain this material and provide examples. We will repeat the definitions in those sections, and you may skip the next section at first reading and proceed to Section 4.2 (p. 116).

4.1 Specification of the Metamath Language

Sometimes one has to say difficult things, but one ought to say them as simply as one knows how.

G. H. HARDY[2]

4.1.1 Preliminaries

A Metamath **database** is built up from a top-level source file together with any source files that are brought in through file inclusion commands (see below). The only characters that are allowed to appear in a Metamath source file are the 94 non-whitespace printable ASCII characters, which are digits, upper and lower case letters, and the following 32 special characters:

```
! " # $ % & ' ( ) * + , - . / :
; < = > ? @ [ \ ] ^ _ ` { | } ~
```

plus the following characters which are the "white space" characters: space (a printable character), tab, carriage return, line feed, and form feed. We will use `typewriter` font to display the printable characters.

A Metamath database consists of a sequence of three kinds of **tokens** separated by **white space** (which is any sequence of one or more white space characters). The set of **keyword** tokens is `$(`, `$)`, `$c`, `$v`, `$f`, `$e`, `$d`, `$a`, `$p`, `$.`, `$=`, `$(`, `$)`, `$[`, and `$]`. The last four are called **auxiliary** or preprocessing keywords. A **label** token consists of any combination of letters, digits, and the characters hyphen, underscore, and period. A **math symbol** token may consist of any combination of the 93 printable standard ASCII characters other than space or `$` . All tokens are case-sensitive.

[2]As quoted in [16], p. 273.

4.1.2 Preprocessing

The token `$(` begins a **comment** and `$)` ends a comment. Comments may contain any of the 94 non-whitespace printable characters and white space, except they may not contain the 2-character sequences `$(` or `$)` (comments do not nest). Comments are ignored (treated like white space) for the purpose of parsing, e.g., `$( $[ $)` is a comment. See p. 140 for comment typesetting conventions; these conventions may be ignored for the purpose of parsing.

A **file inclusion command** consists of `$[` followed by a file name followed by `$]`. It is only allowed in the outermost scope (i.e., not between `${` and `$}`) and must not be inside a statement (e.g., it may not occur between the label of a `$a` statement and its `$.`). The file name may not contain a `$` or white space. The file must exist. The case-sensitivity of its name follows the conventions of the operating system. The contents of the file replace the inclusion command. Included files may include other files. Only the first reference to a given file is included; any later references to the same file (whether in the top-level file or in included files) cause the inclusion command to be ignored (treated like white space). A verifier may assume that file names with different strings refer to different files for the purpose of ignoring later references. A file self-reference is ignored, as is any reference to the top-level file (to avoid loops). Included files may not include a `$(` without a matching `$)`, may not include a `$[` without a matching `$]`, and may not include incomplete statements (e.g., a `$a` without a matching `$.`). It is currently unspecified if path references are relative to the process' current directory or the file's containing directory, so databases should avoid using pathname separators (e.g., "/") in file names.

Like all tokens, the `$(`, `$)`, `$[`, and `$]` keywords must be surrounded by white space.

4.1.3 Basic Syntax

After preprocessing, a database will consist of a sequence of **statements**. These are the scoping statements `${` and `$}`, along with the `$c`, `$v`, `$f`, `$e`, `$d`, `$a`, and `$p` statements.

A **scoping statement** consists only of its keyword, `${` or `$}`. A `${` begins a **block** and a matching `$}` ends the block. Every `${` must have a matching `$}`. Defining it recursively, we say a block contains a sequence of zero or more tokens other than `${` and `$}` and possibly other blocks. There is an **outermost block** not bracketed by `${ $}`; the end of the outermost block is the end of the database.

A **`$v` or `$c` statement** consists of the keyword token `$v` or `$c` respectively, followed by one or more math symbols, followed by the `$.` token. These statements **declare** the math symbols to be **variables** or **constants** respectively. The same math symbol may not occur twice in a given `$v` or `$c` statement.

A math symbol becomes **active** when declared and stays active until the end of the block in which it is declared. A variable may not be declared a second time while it is active, but it may be declared again (as a variable, but not as a constant) after it becomes inactive. A constant must be declared in the outermost block and may not be declared a second time.

A **$f statement** consists of a label, followed by `$f`, followed by its typecode (an active constant), followed by an active variable, followed by the `$.` token. A **$e statement** consists of a label, followed by `$e`, followed by its typecode (an active constant), followed by zero or more active math symbols, followed by the `$.` token. A **hypothesis** is a `$f` or `$e` statement. The type declared by a `$f` statement for a given label is global even if the variable is not (e.g., a database may not have `wff P` in one local scope nd `class P` in another).

A **simple $d statement** consists of `$d`, followed by two different active variables, followed by the `$.` token. A **compound $d statement** consists of `$d`, followed by three or more variables (all different), followed by the `$.` token. The order of the variables in a `$d` statement is unimportant. A compound `$d` statement is equivalent to a set of simple `$d` statements, one for each possible pair of variables occurring in the compound `$d` statement. Henceforth in this specification we shall assume all `$d` statements are simple. A `$d` statement is also called a **disjoint** (or **distinct**) **variable restriction**.

A **$a statement** consists of a label, followed by `$a`, followed by its typecode (an active constant), followed by zero or more active math symbols, followed by the `$.` token. A **$p statement** consists of a label, followed by `$p`, followed by its typecode (an active constant), followed by zero or more active math symbols, followed by `$=`, followed by a sequence of labels, followed by the `$.` token. An **assertion** is a `$a` or `$p` statement.

A `$f`, `$e`, or `$d` statement is **active** from the place it occurs until the end of the block it occurs in. A `$a` or `$p` statement is **active** from the place it occurs through the end of the database. There may not be two active `$f` statements containing the same variable. Each variable in a `$e`, `$a`, or `$p` statement must exist in an active `$f` statement.[3]

Each label token must be unique, and no label token may match any math symbol token.[4]

The set of **mandatory variables** associated with an assertion is the set of (zero or more) variables in the assertion and in any active `$e` statements. The (possibly empty) set of **mandatory hypotheses** is the set of all active `$f` statements containing mandatory variables, together with all active `$e` statements. The set of **mandatory $d statements** associated with an assertion are those active `$d` statements whose variables are both among the assertion's mandatory variables.

[3]This requirement can greatly simplify the unification algorithm (substitution calculation) required by proof verification.

[4]This restriction was added on June 24, 2006. It is not theoretically necessary but is imposed to make it easier to write certain parsers.

4.1.4 Proof Verification

The sequence of labels between the `$=` and `$.` tokens in a `$p` statement is a **proof**. Each label in a proof must be the label of an active statement other than the `$p` statement itself; thus a label must refer either to an active hypothesis of the `$p` statement or to an earlier assertion.

An **expression** is a sequence of math symbols. A **substitution map** associates a set of variables with a set of expressions. It is acceptable for a variable to be mapped to an expression containing it. A **substitution** is the simultaneous replacement of all variables in one or more expressions with the expressions that the variables map to.

A proof is scanned in order of its label sequence. If the label refers to an active hypothesis, the expression in the hypothesis is pushed onto a stack. If the label refers to an assertion, a (unique) substitution must exist that, when made to the mandatory hypotheses of the referenced assertion, causes them to match the topmost (i.e. most recent) entries of the stack, in order of occurrence of the mandatory hypotheses, with the topmost stack entry matching the last mandatory hypothesis of the referenced assertion. As many stack entries as there are mandatory hypotheses are then popped from the stack. The same substitution is made to the referenced assertion, and the result is pushed onto the stack. After the last label in the proof is processed, the stack must have a single entry that matches the expression in the `$p` statement containing the proof.

A proof may contain a `?` in place of a label to indicate an unknown step (Section 4.4.6). A proof verifier may ignore any proof containing `?` but should warn the user that the proof is incomplete.

A **compressed proof** is an alternate proof notation described in Appendix B; also see references to "compressed proof" in the Index. Compressed proofs are a Metamath language extension which a complete proof verifier should be able to parse and verify.

Verifying Disjoint Variable Restrictions

Each substitution made in a proof must be checked to verify that any disjoint variable restrictions are satisfied, as follows.

If two variables replaced by a substitution exist in a mandatory `$d` statement of the assertion referenced, the two expressions resulting from the substitution must satisfy the following conditions. First, the two expressions must have no variables in common. Second, each possible pair of variables, one from each expression, must exist in an active `$d` statement of the `$p` statement containing the proof.

This ends the specification of the Metamath language; see Appendix E for its syntax in Extended Backus–Naur Form (EBNF).

4.2 The Basic Keywords

Our expository material begins here.

Like most computer languages, Metamath takes its input from one or more **source files** which contain characters expressed in the standard ASCII (American Standard Code for Information Interchange) code for computers. A source file consists of a series of **tokens**, which are strings of non-whitespace printable characters (from the set of 94 shown on p. 112) separated by **white space** (spaces, tabs, carriage returns, line feeds, and form feeds). Any string consisting only of these characters is treated the same as a single space. The non-whitespace printable characters that Metamath recognizes are the 94 characters on standard ASCII keyboards.

Metamath has the ability to join several files together to form its input (Section 4.4.4). We call the aggregate contents of all the files after they have been joined together a **database** to distinguish it from an individual source file. The tokens in a database consist of **keywords**, which are built into the language, together with two kinds of user-defined tokens called **labels** and **math symbols**. (Often we will simply say **symbol** instead of math symbol for brevity). The set of **basic keywords** is `$c`, `$v`, `$e`, `$f`, `$d`, `$a`, `$p`, `$=`, `$.`, `${`, and `$}`. This is the complete set of syntactical elements of what we call the **basic language** of Metamath, and with them you can express all of the mathematics that were intended by the design of Metamath. You should make it a point to become very familiar with them. Table 4.1 lists the basic keywords along with a brief description of their functions. For now, this description will give you only a vague notion of what the keywords are for; later we will describe the keywords in detail.

Table 4.1: Summary of the basic Metamath keywords

Keyword	*Description*
`$c`	Constant symbol declaration
`$v`	Variable symbol declaration
`$d`	Disjoint variable restriction
`$f`	Variable-type ("floating") hypothesis
`$e`	Logical ("essential") hypothesis
`$a`	Axiomatic assertion
`$p`	Provable assertion
`$=`	Start of proof in `$p` statement
`$.`	End of the above statement types
`${`	Start of block
`$}`	End of block

There are some additional keywords, called **auxiliary keywords** that

help make Metamath more practical. These are part of the **extended language**. They provide you with a means to put comments into a Metamath source file and reference other source files. We will introduce these in later sections. Table 4.2 summarizes them so that you can recognize them now if you want to peruse some source files while learning the basic keywords.

Table 4.2: Auxiliary Metamath keywords

Keyword	*Description*
`$(`	Start of comment
`$)`	End of comment
`$[`	Start of included source file name
`$]`	End of included source file name

Unlike those in some computer languages, the keywords are short two-character sequences rather than English-like words. While this may make them slightly more difficult to remember at first, their brevity allows them to blend in with the mathematics being described, not distract from it, like punctuation marks.

4.2.1 User-Defined Tokens

As you may have noticed, all keywords begin with the `$` character. This mundane monetary symbol is not ordinarily used in higher mathematics (outside of grant proposals), so we have appropriated it to distinguish the Metamath keywords from ordinary mathematical symbols. The `$` character is thus considered special and may not be used as a character in a user-defined token. All tokens and keywords are case-sensitive; for example, `n` is considered to be a different character from `N`. Case-sensitivity makes the available ASCII character set as rich as possible.

Math Symbol Tokens

Math symbols are tokens used to represent the symbols that appear in ordinary mathematical formulas. They may consist of any combination of the 93 non-whitespace printable ASCII characters other than `$` . Some examples are `x`, `+`, `(`, `|-`, `!%@?&`, and `bounded`. For readability, it is best to try to make these look as similar to actual mathematical symbols as possible, within the constraints of the ASCII character set, in order to make the resulting mathematical expressions more readable.

In the Metamath language, you express ordinary mathematical formulas and statements as sequences of math symbols such as `2 + 2 = 4` (five

symbols, all constants).[5] They may even be English sentences, as in `E is closed and bounded` (five symbols)—here `E` would be a variable and the other four symbols constants. In principle, a Metamath database could be constructed to work with almost any unambiguous English-language mathematical statement, but as a practical matter the definitions needed to provide for all possible syntax variations would be cumbersome and distracting and possibly have subtle pitfalls accidentally built in. We generally recommend that you express mathematical statements with compact standard mathematical symbols whenever possible and put their English-language descriptions in comments. Axioms and definitions (`$a` statements) are the only places where Metamath will not detect an error, and doing this will help reduce the number of definitions needed.

You are free to use any tokens you like for math symbols. Appendix A recommends token names to use for symbols in set theory, and we suggest you adopt these in order to be able to include the `set.mm` set theory database in your database. For printouts, you can convert the tokens in a database to standard mathematical symbols with the LaTeX typesetting program. The Metamath command `open tex` *filename* produces output that can be read by LaTeX. The correspondence between tokens and the actual symbols is made by `latexdef` statements inside a special database comment tagged with `$t`. You can edit this comment to change the definitions or add new ones. Appendix A describes how to do this in more detail.

Label Tokens

Label tokens are used to identify Metamath statements for later reference. Label tokens may contain only letters, digits, and the three characters period, hyphen, and underscore:

```
. - _
```

A label is **declared** by placing it immediately before the keyword of the statement it identifies. For example, the label `axiom.1` might be declared as follows:

```
axiom.1 $a |- x = x $.
```

Each `$e`, `$f`, `$a`, and `$p` statement in a database must have a label declared for it. No other statement types may have label declarations. Every label must be unique.

A label (and the statement it identifies) is **referenced** by including the label between the `$=` and `$.` keywords in a `$p` statement. The sequence of labels between these two keywords is called a **proof**. An example of a statement with a proof that we will encounter later (Section 4.3) is

[5] To eliminate ambiguity with other expressions, this is expressed in the set theory database `set.mm` as `|- ( 2 + 2 ) = 4` , whose LaTeX equivalent is $\vdash (2 + 2) = 4$. The $\vdash$ means "is a theorem" and the parentheses allow explicit associative grouping.

```
wnew $p wff ( s -> ( r -> p ) )
     $= ws wr wp w2 w2 $.
```

You don't have to know what this means just yet, but you should know that the label `wnew` is declared by this `$p` statement and that the labels `ws`, `wr`, `wp`, and `w2` are assumed to have been declared earlier in the database and are referenced here.

4.2.2 Constants and Variables

An **expression** is any sequence of math symbols, possibly empty.

The basic Metamath language has two kinds of math symbols: **constants** and **variables**. In a Metamath proof, a constant may not be substituted with any expression. A variable can be substituted with any expression. This sequence may include other variables and may even include the variable being substituted. This substitution takes place when proofs are verified, and it will be described in Section 4.3. The `$f` statement (described later in Section 4.2.5) is used to specify the **type** of a variable (i.e. what kind of variable it is) and give it a meaning typically associated with a "metavariable"[6] in ordinary mathematics; for example, a variable may be specified to be a wff or well-formed formula (in logic), a set (in set theory), or a non-negative integer (in number theory).

4.2.3 The `$c` and `$v` Declaration Statements

Constants are introduced or **declared** with `$c` statements, and variables are declared with `$v` statements. A **simple** declaration statement introduces a single constant or variable. Its syntax is one of the following:

`$c` *math-symbol* `$.`
`$v` *math-symbol* `$.`

The notation *math-symbol* means any math symbol token.

Some examples of simple declaration statements are:

```
$c + $.
$c -> $.
$c ( $.
$v x $.
$v y2 $.
```

The characters in a math symbol being declared are irrelevant to Metamath; for example, we could declare a right parenthesis to be a variable,

[6] A metavariable is a variable that ranges over the syntactical elements of the object language being discussed; for example, one metavariable might represent a variable of the object language and another metavariable might represent a formula in the object language.

```
$v ) $.
```

although this would be unconventional.

A **compound** declaration statement is a shorthand for declaring several symbols at once. Its syntax is one of the following:

`$c` *math-symbol* $\cdots$ *math-symbol* `$.`
`$v` *math-symbol* $\cdots$ *math-symbol* `$.`

Here, the ellipsis (. . .) means any number of *math-symbol*s.

An example of a compound declaration statement is:

```
$v x y mu $.
```

This is equivalent to the three simple declaration statements

```
$v x $.
$v y $.
$v mu $.
```

There are certain rules on where in the database math symbols may be declared, what sections of the database are aware of them (i.e. where they are "active"), and when they may be declared more than once. These will be discussed in Section 4.2.8 and specifically on p. 134.

4.2.4 The `$d` Statement

The `$d` statement is called a **disjoint-variable restriction**. The syntax of the **simple** version of this statement is

`$d` *variable variable* `$.`

where each *variable* is a previously declared variable and the two *variable*s are different. (More specifically, each *variable* must be an **active** variable, which means there must be a previous `$v` statement whose **scope** includes the `$d` statement. These terms will be defined when we discuss scoping statements in Section 4.2.8.)

In ordinary mathematics, formulas may arise that are true if the variables in them are distinct, but become false when those variables are made identical. For example, the formula in logic $\exists x\, x \neq y$, which means "for a given y, there exists an x that is not equal to y," is true in most mathematical theories (namely all non-trivial theories, i.e. those that describe more than one individual, such as arithmetic). However, if we substitute y with x, we obtain $\exists x\, x \neq x$, which is always false, as it means "there exists something that is not equal to itself."[7] The `$d` statement allows you to specify a

[7] If you are a logician, you will recognize this as the improper substitution of a free variable with a bound variable. Metamath makes no inherent distinction between free and bound variables; instead, you let Metamath know what substitutions are permissible by using `$d` statements in the right way in your axiom system.

restriction that forbids the substitution of one variable with another. In this case, we would use the statement

```
$d x y $.
```

to specify this restriction.

The order in which the variables appear in a `$d` statement is not important. We could also use

```
$d y x $.
```

The `$d` statement is actually more general than this, as the "disjoint" in its name suggests. The full meaning is that if any substitution is made to its two variables (during the course of a proof that references a `$a` or `$p` statement associated with the `$d`), the two expressions that result from the substitution must have no variables in common. In addition, each possible pair of variables, one from each expression, must be in a `$d` statement associated with the statement being proved. (This requirement forces the statement being proved to "inherit" the original disjoint variable restriction.)

For example, suppose `u` is a variable. If the restriction

```
$d A B $.
```

has been specified for a theorem referenced in a proof, we may not substitute `A` with `a + u` and `B` with `b + u` because these two symbol sequences have the variable `u` in common. Furthermore, if `a` and `b` are variables, we may not substitute `A` with `a` and `B` with `b` unless we have also specified `$d a b` for the theorem being proved; in other words, the `$d` property associated with a pair of variables must be effectively preserved after substitution.

The `$d` statement does *not* mean "the two variables may not be substituted with the same thing," as you might think at first. For example, substituting each of `A` and `B` in the above example with identical symbol sequences consisting only of constants does not cause a disjoint variable conflict, because two symbol sequences have no variables in common (since they have no variables, period). Similarly, a conflict will not occur by substituting the two variables in a `$d` statement with the empty symbol sequence.

The `$d` statement does not have a direct counterpart in ordinary mathematics, partly because the variables of Metamath are not really the same as the variables of ordinary mathematics but rather are metavariables ranging over them (as well as over other kinds of symbols and groups of symbols). Depending on the situation, we may informally interpret the `$d` statement in different ways. Suppose, for example, that `x` and `y` are variables ranging over numbers (more precisely, that `x` and `y` are metavariables ranging over variables that range over numbers), and that `ph` (φ) and `ps` (ψ) are variables (more precisely, metavariables) ranging over formulas. We can make the following interpretations that correspond to the informal language of ordinary mathematics:

`$d x y $.` means "assume x and y are distinct variables."
`$d x ph $.` means "assume x does not occur in φ."
`$d ph ps $.` means "assume φ and ψ have no variables in common."

Compound `$d` Statements

The **compound** version of the `$d` statement is a shorthand for specifying several variables whose substitutions must be pairwise disjoint. Its syntax is:

`$d` *variable* $\cdots$ *variable* `$.`

Here, *variable* represents the token of a previously declared variable (specifically, an active variable) and all *variable*s are different. The compound `$d` statement is internally broken up by Metamath into one simple `$d` statement for each possible pair of variables in the original `$d` statement. For example,

```
$d w x y z $.
```

is equivalent to

```
$d w x $.
$d w y $.
$d w z $.
$d x y $.
$d x z $.
$d y z $.
```

Two or more simple `$d` statements specifying the same variable pair are internally combined into a single `$d` statement. Thus the set of three statements

```
$d x y $. $d x y $. $d y x $.
```

is equivalent to

```
$d x y $.
```

Similarly, compound `$d` statements, after being internally broken up, internally have their common variable pairs combined. For example the set of statements

```
$d x y A $. $d x y B $.
```

is equivalent to

```
$d x y $. $d x A $. $d y A $. $d x y $. $d x B $. $d y B $.
```

which is equivalent to

```
$d x y $. $d x A $. $d y A $. $d x B $. $d y B $.
```

Metamath automatically verifies that all `$d` restrictions are met whenever it verifies proofs. `$d` statements are never referenced directly in proofs (this is why they do not have labels), but Metamath is always aware of which ones must be satisfied (i.e. are active) and will notify you with an error message if any violation occurs.

To illustrate how Metamath detects a missing `$d` statement, we will look at the following example from the `set.mm` database.

```
$d x z $.  $d y z $.
$( Theorem to add distinct quantifier to atomic formula. $)
ax17eq $p |- ( x = y -> A. z x = y ) $=...
```

This statement has the obvious requirement that z must be distinct from x in theorem `ax17eq` that states $x = y \to \forall z\, x = y$ (well, obvious if you're a logician, for otherwise we could conclude $x = y \to \forall x\, x = y$, which is false when the free variables x and y are equal).

Let's look at what happens if we edit the database to comment out this requirement.

```
$( $d x z $. $) $d y z $.
$( Theorem to add distinct quantifier to atomic formula. $)
ax17eq $p |- ( x = y -> A. z x = y ) $=...
```

When it tries to verify the proof, Metamath will tell you that `x` and `z` must be disjoint, because one of its steps references an axiom or theorem that has this requirement.

```
MM> verify proof ax17eq
ax17eq ?Error at statement 1918, label "ax17eq", type "$p":
      vz wal wi vx vy vz ax-13 vx vy weq vz vx ax-c16 vx vy
                                                 ^^^^^
There is a disjoint variable ($d) violation at proof step 29.
Assertion "ax-c16" requires that variables "x" and "y" be
disjoint.  But "x" was substituted with "z" and "y" was
substituted with "x".  The assertion being proved, "ax17eq",
does not require that variables "z" and "x" be disjoint.
```

We can see the substitutions into `ax-c16` with the following command.

```
MM> show proof ax17eq / detailed_step 29
Proof step 29:  pm2.61dd.2=ax-c16 $a |- ( A. z z = x -> ( x =
  y -> A. z x = y ) )
This step assigns source "ax-c16" ($a) to target "pm2.61dd.2"
($e).  The source assertion requires the hypotheses "wph"
```

```
($f, step 26), "vx" ($f, step 27), and "vy" ($f, step 28).
The parent assertion of the target hypothesis is "pm2.61dd"
($p, step 36).
The source assertion before substitution was:
    ax-c16 $a |- ( A. x x = y -> ( ph -> A. x ph ) )
The following substitutions were made to the source
assertion:
    Variable  Substituted with
     x         z
     y         x
     ph        x = y
The target hypothesis before substitution was:
    pm2.61dd.2 $e |- ( ph -> ch )
The following substitutions were made to the target
hypothesis:
    Variable  Substituted with
     ph        A. z z = x
     ch        ( x = y -> A. z x = y )
```

The disjoint variable restrictions of `ax-c16` can be seen from the `show statement` command. The line that begins "`Its mandatory disjoint variable pairs are:...`" lists any `$d` variable pairs in brackets.

```
MM> show statement ax-c16/full
Statement 3033 is located on line 9338 of the file "set.mm".
"Axiom of Distinct Variables. ..."
  ax-c16 $a |- ( A. x x = y -> ( ph -> A. x ph ) ) $.
Its mandatory hypotheses in RPN order are:
  wph $f wff ph $.
  vx $f setvar x $.
  vy $f setvar y $.
Its mandatory disjoint variable pairs are:  <x,y>
The statement and its hypotheses require the variables:  x y
      ph
The variables it contains are:  x y ph
```

Since Metamath will always detect when `$d` statements are needed for a proof, you don't have to worry too much about forgetting to put one in; it can always be added if you see the error message above. If you put in unnecessary `$d` statements, the worst that could happen is that your theorem might not be as general as it could be, and this may limit its use later on.

On the other hand, when you introduce axioms (`$a` statements), you must be very careful to properly specify the necessary associated `$d` statements since Metamath has no way of knowing whether your axioms are correct. For example, Metamath would have no idea that `ax-c16`, which we are telling it

is an axiom of logic, would lead to contradictions if we omitted its associated `$d` statement.

You may wonder if it is possible to develop standard mathematics in the Metamath language without the `$d` statement, since it seems like a nuisance that complicates proof verification. The `$d` statement is not needed in certain subsets of mathematics such as propositional calculus. However, dummy variables and their associated `$d` statements are impossible to avoid in proofs in standard first-order logic as well as in the variant used in `set.mm`. In fact, there is no upper bound to the number of dummy variables that might be needed in a proof of a theorem of first-order logic containing 3 or more variables, as shown by H. Andréka [50]. A first-order system that avoids them entirely is given in [41]; the trick there is simply to embed harmlessly the necessary dummy variables into a theorem being proved so that they aren't "dummy" anymore, then interpret the resulting longer theorem so as to ignore the embedded dummy variables. If this interests you, the system in `set.mm` obtained from `ax-1` through `ax-c14` in `set.mm`, and deleting `ax-c16` and `ax-5`, requires no `$d` statements but is logically complete in the sense described in [41]. This means it can prove any theorem of first-order logic as long as we add to the theorem an antecedent that embeds dummy and any other variables that must be distinct. In a similar fashion, axioms for set theory can be devised that do not require distinct variable provisos, as explained at `http://us.metamath.org/mpeuni/mmzfcnd.html`. Together, these in principle allow all of mathematics to be developed under Metamath without a `$d` statement, although the length of the resulting theorems will grow as more and more dummy variables become required in their proofs.

4.2.5 The `$f` and `$e` Statements

Metamath has two kinds of hypotheses, the `$f` or **variable-type** hypothesis and the `$e` or **logical** hypothesis.[8] The letters `f` and `e` stand for "floating" (roughly meaning used only if relevant) and "essential" (meaning always used) respectively, for reasons that will become apparent when we discuss frames in Section 4.2.7 and scoping in Section 4.2.8. The syntax of these are as follows:

label `$f` *typecode variable* `$.`
label `$e` *typecode math-symbol* $\cdots$ *math-symbol* `$.`

A hypothesis must have a *label*. The expression in a `$e` hypothesis consists of a typecode (an active constant math symbol) followed by a sequence of zero or more math symbols. Each math symbol (including *constant* and *variable*) must be a previously declared constant or variable. (In addition,

[8] Strictly speaking, the `$d` statement is also a hypothesis, but it is never directly referenced in a proof, so we call it a restriction rather than a hypothesis to lessen confusion. The checking for violations of `$d` restrictions is automatic and built into Metamath's proof-checking algorithm.

each math symbol must be active, which will be covered when we discuss scoping statements in Section 4.2.8.) You use a `$f` hypothesis to specify the nature or **type** of a variable (such as "let x be an integer") and use a `$e` hypothesis to express a logical truth (such as "assume x is prime") that must be established in order for an assertion requiring it to also be true.

A variable must have its type specified in a `$f` statement before it may be used in a `$e`, `$a`, or `$p` statement. There may be only one (active) `$f` statement for a given variable. ("Active" is defined in Section 4.2.8.)

In ordinary mathematics, theorems are often expressed in the form "Assume P; then Q," where Q is a statement that you can derive if you start with statement P.[9] In the Metamath language, you would express mathematical statement P as a hypothesis (a `$e` Metamath language statement in this case) and statement Q as a provable assertion (a `$p` statement).

Some examples of hypotheses you might encounter in logic and set theory are

```
stmt1 $f wff P $.
stmt2 $f setvar x $.
stmt3 $e |- ( P -> Q ) $.
```

Informally, these would be read, "Let P be a well-formed-formula," "Let x be an (individual) variable," and "Assume we have proved $P \to Q$." The turnstile symbol $\vdash$ is commonly used in logic texts to mean "a proof exists for."

To summarize:

- A `$f` hypothesis tells Metamath the type or kind of its variable. It is analogous to a variable declaration in a computer language that tells the compiler that a variable is an integer or a floating-point number.
- The `$e` hypothesis corresponds to what you would usually call a "hypothesis" in ordinary mathematics.

Before an assertion (`$a` or `$p` statement) can be referenced in a proof, all of its associated `$f` and `$e` hypotheses (i.e. those `$e` hypotheses that are active) must be satisfied (i.e. established by the proof). The meaning of "associated" (which we will call **mandatory** in Section 4.2.7) will become clear when we discuss scoping later.

Note that after any `$f`, `$e`, `$a`, or `$p` token there is a required *typecode*. The typecode is a constant used to enforce types of expressions. This will

[9] A stronger version of a theorem like this would be the *single* formula $P \to Q$ (P implies Q) from which the weaker version above follows by the rule of modus ponens in logic. We are not discussing this stronger form here. In the weaker form, we are saying only that if we can *prove* P, then we can *prove* Q. In a logician's language, if x is the only free variable in P and Q, the stronger form is equivalent to $\forall x(P \to Q)$ (for all x, P implies Q), whereas the weaker form is equivalent to $\forall x P \to \forall x Q$. The stronger form implies the weaker, but not vice-versa. To be precise, the weaker form of the theorem is more properly called an "inference" rather than a theorem.

become clearer once we learn more about assertions (`$a` and `$p` statements). An example may also clarify their purpose. In the `set.mm` database, the following typecodes are used:

- `wff` : Well-formed formula (wff) symbol (read: "the following symbol sequence is a wff").
- $\vdash$: Turnstile (read: "the following symbol sequence is provable" or "a proof exists for").
- `setvar` : Individual set variable type (read: "the following is an individual set variable"). Note that this is *not* the type of an arbitrary set expression, instead, it is used to ensure that there is only a single symbol used after quantifiers like for-all ($\forall$) and there-exists ($\exists$).
- `class` : An expression that is a syntactically valid class expression. All valid set expressions are also valid class expression, so expressions of sets normally have the `class` typecode. Use the `class` typecode, *not* the `setvar` typecode, for the type of set expressions unless you are specifically identifying a single set variable.

4.2.6 Assertions (`$a` and `$p` Statements)

There are two types of assertions, `$a` statements (**axiomatic assertions**) and `$p` statements (**provable assertions**). Their syntax is as follows:

label `$a` *typecode math-symbol ... math-symbol* `$.`
label `$p` *typecode math-symbol ... math-symbol* `$=` *proof* `$.`

An assertion always requires a *label*. The expression in an assertion consists of a typecode (an active constant) followed by a sequence of zero or more math symbols. Each math symbol, including any *constant*, must be a previously declared constant or variable. (In addition, each math symbol must be active, which will be covered when we discuss scoping statements in Section 4.2.8.)

A `$a` statement is usually a definition of syntax (for example, if P and Q are wffs then so is $(P \rightarrow Q)$), an axiom of ordinary mathematics (for example, $x = x$), or a definition of ordinary mathematics (for example, $x \neq y$ means $\neg x = y$). A `$p` statement is a claim that a certain combination of math symbols follows from previous assertions and is accompanied by a proof that demonstrates it.

Assertions can also be referenced in (later) proofs in order to derive new assertions from them. The label of an assertion is used to refer to it in a proof. Section 4.3 will describe the proof in detail.

Assertions also provide the primary means for communicating the mathematical results in the database to people. Proofs (when conveniently displayed) communicate to people how the results were arrived at.

The $a Statement

Axiomatic assertions (`$a` statements) represent the starting points from which other assertions (`$p` statements) are derived. Their most obvious use is for specifying ordinary mathematical axioms, but they are also used for two other purposes.

First, Metamath needs to know the syntax of symbol sequences that constitute valid mathematical statements. A Metamath proof must be broken down into much more detail than ordinary mathematical proofs that you may be used to thinking of (even the "complete" proofs of formal logic). This is one of the things that makes Metamath a general-purpose language, independent of any system of logic or even syntax. If you want to use a substitution instance of an assertion as a step in a proof, you must first prove that the substitution is syntactically correct (or if you prefer, you must "construct" it), showing for example that the expression you are substituting for a wff metavariable is a valid wff. The `$a` statement is used to specify those combinations of symbols that are considered syntactically valid, such as the legal forms of wffs.

Second, `$a` statements are used to specify what are ordinarily thought of as definitions, i.e. new combinations of symbols that abbreviate other combinations of symbols. Metamath makes no distinction between axioms and definitions. Indeed, it has been argued that such distinction should not be made even in ordinary mathematics; see Section 4.5, which discusses the philosophy of definitions. Section 3.4 discusses some technical requirements for definitions. In `set.mm` we adopt the convention of prefixing axiom labels with `ax-` and definition labels with `df-`.

The results that can be derived with the Metamath language are only as good as the `$a` statements used as their starting point. We cannot stress this too strongly. For example, Metamath will not prevent you from specifying $x \neq x$ as an axiom of logic. It is essential that you scrutinize all `$a` statements with great care. Because they are a source of potential pitfalls, it is best not to add new ones (usually new definitions) casually; rather you should carefully evaluate each one's necessity and advantages.

Once you have in place all of the basic axioms and rules of a mathematical theory, the only `$a` statements that you will be adding will be what are ordinarily called definitions. In principle, definitions should be in some sense eliminable from the language of a theory according to some convention (usually involving logical equivalence or equality). The most common convention is that any formula that was syntactically valid but not provable before the definition was introduced will not become provable after the definition is introduced. In an ideal world, definitions should not be present at all if one is to have absolute confidence in a mathematical result. However, they are necessary to make mathematics practical, for otherwise the resulting formulas would be extremely long and incomprehensible. Since the nature of definitions (in the most general sense) does not permit them to automatically

be verified as "proper," the judgment of the mathematician is required to ensure it. (In `set.mm` effort was made to make almost all definitions directly eliminable and thus minimize the need for such judgment.)

If you are not a mathematician, it may be best not to add or change any `$a` statements but instead use the mathematical language already provided in standard databases. This way Metamath will not allow you to make a mistake (i.e. prove a false result).

4.2.7 Frames

We now introduce the concept of a collection of related Metamath statements called a frame. Every assertion (`$a` or `$p` statement) in the database has an associated frame.

A **frame** is a sequence of `$d`, `$f`, and `$e` statements (zero or more of each) followed by one `$a` or `$p` statement, subject to certain conditions we will describe. For simplicity we will assume that all math symbol tokens used are declared at the beginning of the database with `$c` and `$v` statements (which are not properly part of a frame). Also for simplicity we will assume there are only simple `$d` statements (those with only two variables) and imagine any compound `$d` statements (those with more than two variables) as broken up into simple ones.

A frame groups together those hypotheses (and `$d` statements) relevant to an assertion (`$a` or `$p` statement). The statements in a frame may or may not be physically adjacent in a database; we will cover this in our discussion of scoping statements in Section 4.2.8.

A frame has the following properties:

1. The set of variables contained in its `$f` statements must be identical to the set of variables contained in its `$e`, `$a`, and/or `$p` statements. In other words, each variable in a `$e`, `$a`, or `$p` statement must have an associated "variable type" defined for it in a `$f` statement.

2. No two `$f` statements may contain the same variable.

3. Any `$f` statement must occur before a `$e` statement in which its variable occurs.

The first property determines the set of variables occurring in a frame. These are the **mandatory variables** of the frame. The second property tells us there must be only one type specified for a variable. The last property is not a theoretical requirement but it makes parsing of the database easier.

For our examples, we assume our database has the following declarations:

```
$v P Q R $.
$c -> ( ) |- wff $.
```

The following sequence of statements, describing the modus ponens inference rule, is an example of a frame:

```
wp  $f wff P $.
wq  $f wff Q $.
maj $e |- ( P -> Q ) $.
min $e |- P $.
mp  $a |- Q $.
```

The following sequence of statements is not a frame because `R` does not occur in the `$e`'s or the `$a`:

```
wp  $f wff P $.
wq  $f wff Q $.
wr  $f wff R $.
maj $e |- ( P -> Q ) $.
min $e |- P $.
mp  $a |- Q $.
```

The following sequence of statements is not a frame because `Q` does not occur in a `$f`:

```
wp  $f wff P $.
maj $e |- ( P -> Q ) $.
min $e |- P $.
mp  $a |- Q $.
```

The following sequence of statements is not a frame because the `$a` statement is not the last one:

```
wp  $f wff P $.
wq  $f wff Q $.
maj $e |- ( P -> Q ) $.
mp  $a |- Q $.
min $e |- P $.
```

Associated with a frame is a sequence of **mandatory hypotheses**. This is simply the set of all `$f` and `$e` statements in the frame, in the order they appear. A frame can be referenced in a later proof using the label of the `$a` or `$p` assertion statement, and the proof makes an assignment to each mandatory hypothesis in the order in which it appears. This means the order of the hypotheses, once chosen, must not be changed so as not to affect later proofs referencing the frame's assertion statement. (The Metamath proof verifier will, of course, flag an error if a proof becomes incorrect by doing this.) Since proofs make use of "Reverse Polish notation," described in Section 4.3, we call this order the **RPN order** of the hypotheses.

Note that `$d` statements are not part of the set of mandatory hypotheses, and their order doesn't matter (as long as they satisfy the fourth property for a frame described above). The `$d` statements specify restrictions on variables that must be satisfied (and are checked by the proof verifier) when expressions are substituted for them in a proof, and the `$d` statements themselves are never referenced directly in a proof.

A frame with a `$p` (provable) statement requires a proof as part of the `$p` statement. Sometimes in a proof we want to make use of temporary or dummy variables that do not occur in the `$p` statement or its mandatory hypotheses. To accommodate this we define an **extended frame** as a frame together with zero or more `$d` and `$f` statements that reference variables not among the mandatory variables of the frame. Any new variables referenced are called the **optional variables** of the extended frame. If a `$f` statement references an optional variable it is called an **optional hypothesis**, and if one or both of the variables in a `$d` statement are optional variables it is called an **optional disjoint-variable restriction**. Properties 2 and 3 for a frame also apply to an extended frame.

The concept of optional variables is not meaningful for frames with `$a` statements, since those statements have no proofs that might make use of them. There is no restriction on including optional hypotheses in the extended frame for a `$a` statement, but they serve no purpose.

The following set of statements is an example of an extended frame, which contains an optional variable `R` and an optional hypothesis `wr`. In this example, we suppose the rule of modus ponens is not an axiom but is derived as a theorem from earlier statements (we omit its presumed proof). Variable `R` may be used in its proof if desired (although this would probably have no advantage in propositional calculus). Note that the sequence of mandatory hypotheses in RPN order is still `wp`, `wq`, `maj`, `min` (i.e. `wr` is omitted), and this sequence is still assumed whenever the assertion `mp` is referenced in a subsequent proof.

```
wp  $f wff P $.
wq  $f wff Q $.
wr  $f wff R $.
maj $e |- ( P -> Q ) $.
min $e |- P $.
mp  $p |- Q $= ... $.
```

Every frame is an extended frame, but not every extended frame is a frame, as this example shows. The underlying frame for an extended frame is obtained by simply removing all statements containing optional variables. Any proof referencing an assertion will ignore any extensions to its frame, which means we may add or delete optional hypotheses at will without affecting subsequent proofs.

The conceptually simplest way of organizing a Metamath database is as a sequence of extended frames. The scoping statements `${` and `$}` can be used

to delimit the start and end of an extended frame, leading to the following possible structure for a database.

($v *and* $c *statements*)
$\{
  extended frame
$\}
$\{
  extended frame
$\}
⋮

In practice, this structure is inconvenient because we have to repeat any $f, $e, and $d statements over and over again rather than stating them once for use by several assertions. The scoping statements, which we will discuss next, allow this to be done. In principle, any Metamath database can be converted to the above format, and the above format is the most convenient to use when studying a Metamath database as a formal system(Appendix C). In fact, Metamath internally converts the database to the above format. The command `show statement` in the Metamath program will show you the contents of the frame for any $a or $p statement, as well as its extension in the case of a $p statement.

During our discussion of scoping statements, it may be helpful to think in terms of the equivalent sequence of frames that will result when the database is parsed. Scoping (other than the limited use above to delimit frames) is not a theoretical requirement for Metamath but makes it more convenient.

4.2.8 Scoping Statements ($\{ and $\})

The **scoping** statements, $\{ (**start of block**) and $\} (**end of block**), provide a means for controlling the portion of a database over which certain statement types are recognized. The syntax of a scoping statement is very simple; it just consists of the statement's keyword:

$\{
$\}

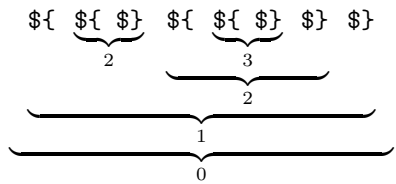
For example, consider the following database where we have stripped out all tokens except the scoping statement keywords. For the purpose of the discussion, we have added subscripts to the scoping statements; these subscripts do not appear in the actual database.

$\{_1 $\{_2 $\}_2 $\{_3 $\{_4 $\}_4 $\}_3 $\}_1

Each $\{ statement in this example is said to be **matched** with the $\} statement that has the same subscript. Each pair of matched scoping

statements defines a region of the database called a **block**. Blocks can be **nested** inside other blocks; in the example, the block defined by `${`$_4$ and `$}`$_4$ is nested inside the block defined by `${`$_3$ and `$}`$_3$ as well as inside the block defined by `${`$_1$ and `$}`$_1$. In general, a block may be empty, it may contain only non-scoping statements,[10] or it may contain any mixture of other blocks and non-scoping statements. (This is called a "recursive" definition of a block.)

Associated with each block is a number called its **nesting level** that indicates how deeply the block is nested. The nesting levels of the blocks in our example are as follows:



The entire database is considered to be one big block (the **outermost** block) with a nesting level of 0. The outermost block is *not* bracketed by scoping statements.[11]

All non-scoping Metamath statements become recognized or **active** at the place where they appear.[12] Certain of these statement types become inactive at the end of the block in which they appear; these statement types are:

`$c`, `$v`, `$d`, `$e`, and `$f`.

The other statement types remain active forever (i.e. through the end of the database); they are:

`$a` and `$p`.

Any statement (of these 7 types) located in the outermost block will remain active through the end of the database and thus are effectively "global" statements.

All `$c` statements must be placed in the outermost block. Since they are therefore always global, they could be considered as belonging to both of the above categories.

The **scope** of a statement is the set of statements that recognize it as active.

The concept of "active" is also defined for math symbols. Math symbols (constants and variables) become **active** in the `$c` and `$v` statements that

[10] Those statements other than `${` and `$}`.

[11] The language was designed this way so that several source files can be joined together more easily.

[12] To keep things slightly simpler, we do not bother to define the concept of "active" for the scoping statements.

declare them. A variable becomes inactive when its declaration statement becomes inactive. Because all `$c` statements must be in the outermost block, a constant will never become inactive after it is declared.

Redeclaration of Math Symbols

A variable may not be declared a second time while it is active, but it may be declared again after it becomes inactive. This provides a convenient way to introduce "local" variables, i.e. temporary variables for use in the frame of an assertion or in a proof without keeping them around forever. A previously declared variable may not be redeclared as a constant.

A constant may not be redeclared. And, as mentioned above, constants must be declared in the outermost block.

The reason variables may have limited scope but not constants is that an assertion (`$a` or `$p` statement) remains available for use in proofs through the end of the database. Variables in an assertion's frame may be substituted with whatever is needed in a proof step that references the assertion, whereas constants remain fixed and may not be substituted with anything. The particular token used for a variable in an assertion's frame is irrelevant when the assertion is referenced in a proof, and it doesn't matter if that token is not available outside of the referenced assertion's frame. Constants, however, must be globally fixed.

There is no theoretical benefit for the feature allowing variables to be active for limited scopes rather than global. It is just a convenience that allows them, for example, to be locally grouped together with their corresponding `$f` variable-type declarations.

Frames Revisited

Now that we have covered scoping, we will look at how an arbitrary Metamath database can be converted to the simple sequence of extended frames described on p. 132. This is also how Metamath stores the database internally when it reads in the database source. The method is simple. First, we collect all constant and variable (`$c` and `$v`) declarations in the database, ignoring duplicate declarations of the same variable in different scopes. We then put our collected `$c` and `$v` declarations at the beginning of the database, so that their scope is the entire database. Next, for each assertion in the database, we determine its frame and extended frame. The extended frame is simply the `$f`, `$e`, and `$d` statements that are active. The frame is the extended frame with all optional hypotheses removed.

An equivalent way of saying this is that the extended frame of an assertion is the collection of all `$f`, `$e`, and `$d` statements whose scope includes the assertion. The `$f` and `$e` statements occur in the order they appear (order is irrelevant for `$d` statements).

4.3 The Anatomy of a Proof

Each provable assertion (`$p` statement) in a database must include a **proof**. The proof is located between the `$=` and `$.` keywords in the `$p` statement.

In the basic Metamath language, a proof is a sequence of statement labels. This label sequence serves as a set of instructions that the Metamath program uses to construct a series of math symbol sequences. The construction must ultimately result in the math symbol sequence contained between the `$p` and `$=` keywords of the `$p` statement. Otherwise, the Metamath program will consider the proof incorrect, and it will notify you with an appropriate error message when you ask it to verify the proof.[13] Each label in a proof is said to **reference** its corresponding statement.

Associated with any assertion (`$p` or `$a` statement) is a set of hypotheses (`$f` or `$e` statements) that are active with respect to that assertion. Some are mandatory and the others are optional. You should review these concepts if necessary.

Each label in a proof must be either the label of a previous assertion (`$a` or `$p` statement) or the label of an active hypothesis (`$e` or `$f` statement) of the `$p` statement containing the proof. Hypothesis labels may reference both the mandatory and the optional hypotheses of the `$p` statement.

The label sequence in a proof specifies a construction in **reverse Polish notation** (RPN). You may be familiar with RPN if you have used older Hewlett–Packard or similar hand-held calculators. In the calculator analogy, a hypothesis label is like a number and an assertion label is like an operation (more precisely, an n-ary operation when the assertion has n `$e`-hypotheses). On an RPN calculator, an operation takes one or more previous numbers in an input sequence, performs a calculation on them, and replaces those numbers and itself with the result of the calculation. For example, the input sequence $2, 3, +$ on an RPN calculator results in 5, and the input sequence $2, 3, 5, \times, +$ results in $2, 15, +$ which results in 17.

Understanding how RPN is processed involves the concept of a **stack**, which can be thought of as a set of temporary memory locations that hold intermediate results. When Metamath encounters a hypothesis label it places or **pushes** the math symbol sequence of the hypothesis onto the stack. When Metamath encounters an assertion label, it associates the most recent stack entries with the *mandatory* hypotheses of the assertion, in the order where the most recent stack entry is associated with the last mandatory hypothesis of the assertion. It then determines what substitutions have to be made into the variables of the assertion's mandatory hypotheses to make them identical to the associated stack entries. It then makes those same substitutions into the assertion itself. Finally, Metamath removes or **pops** the matched

[13] To make the loading faster, the Metamath program does not automatically verify proofs when you `read` in a database unless you use the `/verify` qualifier. After a database has been read in, you may use the `verify proof *` command to verify proofs.

hypotheses from the stack and pushes the substituted assertion onto the stack.

For the purpose of matching the mandatory hypothesis to the most recent stack entries, whether a hypothesis is a `$e` or `$f` statement is irrelevant. The only important thing is that a set of substitutions[14] exist that allow a match (and if they don't, the proof verifier will let you know with an error message). The Metamath language is specified in such a way that if a set of substitutions exists, it will be unique. Specifically, the requirement that each variable have a type specified for it with a `$f` statement ensures the uniqueness.

We will illustrate this with an example. Consider the following Metamath source file:

```
$c ( ) -> wff $.
$v p q r s $.
wp $f wff p $.
wq $f wff q $.
wr $f wff r $.
ws $f wff s $.
w2 $a wff ( p -> q ) $.
wnew $p wff ( s -> ( r -> p ) ) $= ws wr wp w2 w2 $.
```

This Metamath source example shows the definition and "proof" (i.e., construction) of a well-formed formula (wff) in propositional calculus. (You may wish to type this example into a file to experiment with the Metamath program.) The first two statements declare (introduce the names of) four constants and four variables. The next four statements specify the variable types, namely that each variable is assumed to be a wff. Statement `w2` defines (postulates) a way to produce a new wff, `( p -> q )`, from two given wffs `p` and `q`. The mandatory hypotheses of `w2` are `wp` and `wq`. Statement `wnew` claims that `( s -> ( r -> p ) )` is a wff given three wffs `s`, `r`, and `p`. More precisely, `wnew` claims that the sequence of ten symbols `wff ( s -> ( r -> p ) )` is provable from previous assertions and the hypotheses of `wnew`. Metamath does not know or care what a wff is, and as far as it is concerned the typecode `wff` is just an arbitrary constant symbol in a math symbol sequence. The mandatory hypotheses of `wnew` are `wp`, `wr`, and `ws`; `wq` is an optional hypothesis. In our particular proof, the optional hypothesis is not referenced, but in general, any combination of active (i.e. optional and mandatory) hypotheses could be referenced. The proof of statement `wnew` is the sequence of five labels starting with `ws` (step 1) and ending with `w2` (step 5).

When Metamath verifies the proof, it scans the proof from left to right. We will examine what happens at each step of the proof. The stack starts

[14] In the Metamath spec (Section 4.1), we use the singular term "substitution" to refer to the set of substitutions we talk about here.

off empty. At step 1, Metamath looks up label `ws` and determines that it is a hypothesis, so it pushes the symbol sequence of statement `ws` onto the stack:

Stack location	Contents
1	`wff s`

Metamath sees that the labels `wr` and `wp` in steps 2 and 3 are also hypotheses, so it pushes them onto the stack. After step 3, the stack looks like this:

Stack location	Contents
3	`wff p`
2	`wff r`
1	`wff s`

At step 4, Metamath sees that label `w2` is an assertion, so it must do some processing. First, it associates the mandatory hypotheses of `w2`, which are `wp` and `wq`, with stack locations 2 and 3, *in that order*. Metamath determines that the only possible way to make hypothesis `wp` match (become identical to) stack location 2 and `wq` match stack location 3 is to substitute variable `p` with `r` and `q` with `p`. Metamath makes these substitutions into `w2` and obtains the symbol sequence `wff ( r -> p )`. It removes the hypotheses from stack locations 2 and 3, then places the result into stack location 2:

Stack location	Contents
2	`wff ( r -> p )`
1	`wff s`

At step 5, Metamath sees that label `w2` is an assertion, so it must again do some processing. First, it matches the mandatory hypotheses of `w2`, which are `wp` and `wq`, to stack locations 1 and 2. Metamath determines that the only possible way to make the hypotheses match is to substitute variable `p` with `s` and `q` with `( r -> p )`. Metamath makes these substitutions into `w2` and obtains the symbol sequence `wff ( s -> ( r -> p ) )`. It removes stack locations 1 and 2, then places the result into stack location 1:

Stack location	Contents
1	`wff ( s -> ( r -> p ) )`

After Metamath finishes processing the proof, it checks to see that the stack contains exactly one element and that this element is the same as the math symbol sequence in the `$p` statement. This is the case for our proof of `wnew`, so we have proved `wnew` successfully. If the result differs, Metamath will notify you with an error message. An error message will also result if the stack contains more than one entry at the end of the proof, or if the stack did not contain enough entries at any point in the proof to match all

of the mandatory hypotheses of an assertion. Finally, Metamath will notify you with an error message if no substitution is possible that will make a referenced assertion's hypothesis match the stack entries. You may want to experiment with the different kinds of errors that Metamath will detect by making some small changes in the proof of our example.

Metamath's proof notation was designed primarily to express proofs in a relatively compact manner, not for readability by humans. Metamath can display proofs in a number of different ways with the `show proof` command. The `/lemmon` qualifier displays it in a format that is easier to read when the proofs are short, and you saw examples of its use in Chapter 2. For longer proofs, it is useful to see the tree structure of the proof. A tree structure is displayed when the `/lemmon` qualifier is omitted. You will probably find this display more convenient as you get used to it. The tree display of the proof in our example looks like this:

```
1     wp=ws    $f wff s
2        wp=wr    $f wff r
3        wq=wp    $f wff p
4     wq=w2    $a wff ( r -> p )
5  wnew=w2  $a wff ( s -> ( r -> p ) )
```

The number to the left of each line is the step number. Following it is a **hypothesis association**, consisting of two labels separated by `=`. To the left of the `=` (except in the last step) is the label of a hypothesis of an assertion referenced later in the proof; here, steps 1 and 4 are the hypothesis associations for the assertion `w2` that is referenced in step 5. A hypothesis association is indented one level more than the assertion that uses it, so it is easy to find the corresponding assertion by moving directly down until the indentation level decreases to one less than where you started from. To the right of each `=` is the proof step label for that proof step. The statement keyword of the proof step label is listed next, followed by the content of the top of the stack (the most recent stack entry) as it exists after that proof step is processed. With a little practice, you should have no trouble reading proofs displayed in this format.

Metamath proofs include the syntax construction of a formula. In standard mathematics, this kind of construction is not considered a proper part of the proof at all, and it certainly becomes rather boring after a while. Therefore, by default the `show proof` command does not show the syntax construction. Historically `show proof` command *did* show the syntax construction, and you needed to add the `/essential` option to hide, them, but today `/essential` is the default and you need to use `/all` to see the syntax constructions.

When verifying a proof, Metamath will check that no mandatory `$d` statement of an assertion referenced in a proof is violated when substitutions are made to the variables in the assertion. For details see Section 4.1.4 or 4.2.4.

4.3.1 The Concept of Unification

During the course of verifying a proof, when Metamath encounters an assertion label, it associates the mandatory hypotheses of the assertion with the top entries of the RPN stack. Metamath then determines what substitutions it must make to the variables in the assertion's mandatory hypotheses in order for these hypotheses to become identical to their corresponding stack entries. This process is called **unification**. (We also informally use the term "unification" to refer to a set of substitutions that results from the process, as in "two unifications are possible.") After the substitutions are made, the hypotheses are said to be **unified**.

If no such substitutions are possible, Metamath will consider the proof incorrect and notify you with an error message.

The general algorithm for unification described in the literature is somewhat complex. However, in the case of Metamath it is intentionally trivial. Mandatory hypotheses must be pushed on the proof stack in the order in which they appear. In addition, each variable must have its type specified with a `$f` hypothesis before it is used and that each `$f` hypothesis have the restricted syntax of a typecode (a constant) followed by a variable. The typecode in the `$f` hypothesis must match the first symbol of the corresponding RPN stack entry (which will also be a constant), so the only possible match for the variable in the `$f` hypothesis is the sequence of symbols in the stack entry after the initial constant.

In the Proof Assistant, a more general unification algorithm is used. While a proof is being developed, sometimes not enough information is available to determine a unique unification. In this case Metamath will ask you to pick the correct one.

4.4 Extensions to the Metamath Language

4.4.1 Comments in the Metamath Language

The commenting feature allows you to annotate the contents of a database. Just as with most computer languages, comments are ignored for the purpose of interpreting the contents of the database. Comments effectively act as additional white space between tokens when a database is parsed.

A comment may be placed at the beginning, end, or between any two tokens in a source file.

Comments have the following syntax:

`$(` *text* `$)`

Here, *text* is a string, possibly empty, of any characters in Metamath's character set (p. 112), except that the character strings `$(` and `$)` may not

appear in *text*. Thus nested comments are not permitted:[15] Metamath will complain if you give it

```
$( This is a $( nested $) comment. $)
```

To compensate for this non-nesting behavior, I often change all `$`'s to `@`'s in sections of Metamath code I wish to comment out.

The Metamath program supports a number of markup mechanisms and conventions to generate good-looking results in LaTeX and HTML, as discussed below. These markup features have to do only with how the comments are typeset, and have no effect on how Metamath verifies the proofs in the database. The improper use of them may result in incorrectly typeset output, but no Metamath error messages will result during the `read` and `verify proof` commands. (However, the `write theorem_list` command will check for markup errors as a side-effect of its HTML generation.) Section 5.7 has instructions for creating LaTeX output, and section 5.8 has instructions for creating HTML output.

Headings

If the `$(` is immediately followed by a new line starting with a heading marker, it is a header. This can start with:

`####` - major part header

`#*#*` - section header

`=-=-` - subsection header

`-.-.` - subsubsection header

The line following the marker line will be used for the table of contents entry, after trimming spaces. The next line should be another (closing) matching marker line. Any text after that but before the closing `$`, such as an extended description of the section, will be included on the `mmtheoremsNNN.html` page.

For more information, run `help write theorem_list`.

Math mode

Inside of comments, a string of tokens enclosed in grave accents (`) will be converted to standard mathematical symbols during HTML or LaTeX output typesetting, according to the information in the special `$t` comment in the database (see section 4.4.2 for information about the typesetting comment, and Appendix A to see examples of its results).

[15] Computer languages have differing standards for nested comments, and rather than picking one it was felt simplest not to allow them at all, at least in the current version (0.177) of Metamath.

The first grave accent ` causes the output processor to enter **math mode** and the second one exits it. In this mode, the characters following the ` are interpreted as a sequence of math symbol tokens separated by white space. The tokens are looked up in the `$t` comment and if found, they will be replaced by the standard mathematical symbols that they correspond to before being placed in the typeset output file. If not found, the symbol will be output as is and a warning will be issued. The tokens do not have to be active in the database, although a warning will be issued if they are not declared with `$c` or `$v` statements.

Two consecutive grave accents `` are treated as a single actual grave accent (both inside and outside of math mode) and will not cause the output processor to enter or exit math mode.

Here is an example of its use:

```
$( Pierce's axiom, ` ( ( ph -> ps ) -> ph ) -> ph ` ,
                  is not very intuitive.  $)
```

becomes

$(Pierce's axiom, $((\varphi \to \psi) \to \varphi) \to \varphi$, is not very intuitive. $)

Note that the math symbol tokens must be surrounded by white space. White space should also surround the ` delimiters.

The math mode feature also gives you a quick and easy way to generate text containing mathematical symbols, independently of the intended purpose of Metamath. To do this, simply create your text with grave accents surrounding your formulas, after making sure that your math symbols are mapped to LaTeX symbols as described in Appendix A. It is easier if you start with a database with predefined symbols such as `set.mm`. Use your grave-quoted math string to replace an existing comment, then typeset the statement corresponding to that comment following the instructions from the `help tex` command in the Metamath program. You will then probably want to edit the resulting file with a text editor to fine tune it to your exact needs.

Label Mode

Outside of math mode, a tilde ~ indicates to Metamath's output processor that the token that follows (i.e. the characters up to the next white space) represents a statement label or URL. This formatting mode is called **label mode**. If a literal tilde is desired (outside of math mode) instead of label mode, use two tildes in a row to represent it.

When generating a LaTeX output file, the following token will be formatted in `typewriter` font, and the tilde removed, to make it stand out from the rest of the text. This formatting will be applied to all characters after the tilde up to the first white space. Whether or not the token is an actual

statement label is not checked, and the token does not have to have the correct syntax for a label; no error messages will be produced. The only effect of the label mode on the output is that typewriter font will be used for the tokens that are placed in the LaTeX output file.

When generating HTML, the tokens after the tilde *must* be a URL (either http: or https:) or a valid label. Error messages will be issued during that output if they aren't. A hyperlink will be generated to that URL or label.

Link to bibliographical reference

Bibliographical references are handled specially when generating HTML if formatted specially. Text in the form `[`*author*`]` is considered a link to a bibliographical reference. See `help html` and `help write bibliography` in the Metamath program for more information. See also Sections 4.4.2 and 5.8.2.

The `[`*author*`]` notation will also create an entry in the bibliography cross-reference file generated by `write bibliography` (Section 5.8.2) for HTML. For this to work properly, the surrounding comment must be formatted as follows:

keyword label noise-word `[`*author*`]` `p.` *number*

for example

```
Theorem 5.2 of [Monk] p. 223
```

The *keyword* is not case sensitive and must be one of the following:

```
theorem lemma definition compare proposition corollary
axiom rule remark exercise problem notation example
property figure postulate equation scheme chapter
```

The optional *label* may consist of more than one (non-*keyword* and non-*noise-word*) word. The optional *noise-word* is one of:

```
of in from on
```

and is ignored when the cross-reference file is created. The `write bibliography` command will perform error checking to verify the above format.

Parentheticals

The end of a comment may include one or more parenthicals, that is, statements enclosed in parentheses. The Metamath program looks for certain parentheticals and can issue warnings based on them. They are:

`(Contributed by` *NAME*`,` *DATE*`.)` - document the original contributor's name and the date it was created.

`(Revised by` *NAME*`,` *DATE*`.)` - document the contributor's name and creation date that resulted in significant revision (not just an automated minimization or shortening).

`(Proof shortened by` *NAME*`,` *DATE*`.)` - document the contributor's name and date that developed a significant shortening of the proof (not just an automated minimization).

`(Proof modification is discouraged.)` - Note that this proof should normally not be modified.

`(New usage is discouraged.)` - Note that this assertion should normally not be used.

The *DATE* must be in form YYYY-MMM-DD, where MMM is the English abbreviation of that month.

Other markup

There are other markup notations for generating good-looking results beyond math mode and label mode:

`_` (underscore) - Italicize text starting from *space*`_`*non-space* (i.e. `_` with a space before it and a non-space character after it) until the next *non-space*`_`*space*. Normal punctuation (e.g. a trailing comma or period) is ignored when determining *space*.

`_` (underscore) - *non-space*`_`*non-space-string*, where *non-space-string* is a string of non-space characters, will make *non-space-string* become a subscript.

`<HTML>...</HTML>` - do not convert "`<`" and "`>`" in the enclosed text when generating HTML, otherwise process markup normally. This allows direct insertion of HTML commands.

"`&ref;`" - insert an HTML character reference. This is how to insert arbitrary Unicode characters (such as accented characters). Currently only directly supported when generating HTML.

It is recommended that spaces surround any `~` and `‘` tokens in the comment and that a space follow the *label* after a `~` token. This will make global substitutions to change labels and symbol names much easier and also eliminate any future chance of ambiguity. Spaces around these tokens are automatically removed in the final output to conform with normal rules of punctuation; for example, a space between a trailing `‘` and a left parenthesis will be removed.

A good way to become familiar with the markup notation is to look at the extensive examples in the `set.mm` database.

4.4.2 The Typesetting Comment ($t)

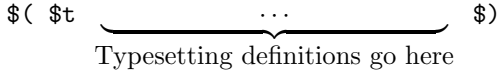
The typesetting comment `$t` in the input database file provides the information necessary to produce good-looking results. It provides LaTeX and HTML definitions for math symbols, as well supporting as some customization of the generated web page. If you add a new token to a database, you should also update the `$t` comment information if you want to eventually create output in LaTeX or HTML. See the `set.mm` database file for an extensive example of a `$t` comment illustrating many of the features described below.

Programs that do not need to generate good-looking presentation results, such as programs that only verify Metamath databases, can completely ignore typesetting comments and just treat them as normal comments. Even the Metamath program only consults the `$t` comment information when it needs to generate typeset output in LaTeX or HTML (e.g., when you open a LaTeX output file with the `open tex` command).

We will first discuss the syntax of typesetting comments, and then briefly discuss how this can be used within the Metamath program.

Typesetting Comment Syntax Overview

The typesetting comment is identified by the token `$t` in the comment, and the typesetting comment ends at the matching `$)`:



Typesetting definitions go here

There must be one or more white space characters, and only white space characters, between the `$(` that starts the comment and the `$t` symbol, and the `$t` must be followed by one or more white space characters (see section 4.1.1 for the definition of white space characters). The typesetting comment continues until the comment end token `$)` (which must be preceded by one or more white space characters).

In version 0.177 of the Metamath program, there may be only one `$t` comment in a database. This restriction may be lifted in the future to allow many `$t` comments in a database.

Between the `$t` symbol (and its following white space) and the comment end token `$)` (and its preceding white space) is a sequence of one or more typesetting definitions, where each definition has the form *definition-type arg arg ... ;*. Each of the zero or more *arg* values can be either a typesetting data or a keyword (what keywords are allowed, and where, depends on the specific *definition-type*). The *definition-type*, and each argument *arg*, are separated by one or more white space characters. Every definition ends in an unquoted semicolon; white space is not required before the terminating semicolon of a definition. Each definition should start on a new line.[16]

[16]This restriction of the current version of Metamath (0.177) may be removed in a future version, but you should do it anyway for readability.

For example, this typesetting definition:

```
latexdef "C_" as "\subseteq";
```

defines the token `C_` as the LaTeX symbol $\subseteq$ (which means "subset").

Typesetting data is a sequence of one or more quoted strings (if there is more than one, they are connected by `+`). Often a single quoted string is used to provide data for a definition, using either double (`"`) or single (`'`) quotation marks. However, *a quoted string (enclosed in quotation marks) may not include line breaks.* A quoted string may include a quotation mark that matches the enclosing quotes by repeating the quotation mark twice. Here are some examples:

Example	**Meaning**
`"a""b"`	`a"b`
`'c''d'`	`c'd`
`"e''f"`	`e''f`
`'g""h'`	`g""h`

Finally, a long quoted string may be broken up into multiple quoted strings (considered, as a whole, a single quoted string) and joined with `+`. You can even use multiple lines as long as a '+' is at the end of every line except the last one. The `+` should be preceded and followed by at least one white space character. Thus, for example,

```
"ab" + "cd" +
    'ef'
```

is the same as

```
"abcdef"
```

C-style comments `/*...*/` are also supported.

In practice, whenever you add a new math token you will often want to add typesetting definitions using `latexdef`, `htmldef`, and `althtmldef`, as described below. That way, they will all be up to date. Of course, whether or not you want to use all three definitions will depend on how the database is intended to be used.

Below we discuss the different possible *definition-kind* options. We will show data surrounded by double quotes (in practice they can also use single quotes and/or be a sequence joined by `+`s). We will use specific names for the *data* to make clear what the data is used for, such as *math-token* (for a Metamath math token, *latex-string* (for string to be placed in a LaTeX stream), HTML-*code* (for HTML code), and *filename* (for a filename).

Typesetting Comment - LaTeX

The syntax for a LaTeX definition is:

```
latexdef "math-token" as "latex-string";
```

The *token-string* and *latex-string* are the data (character strings) for the token and the LaTeX definition of the token, respectively,

These LaTeX definitions are used by the Metamath program when it is asked to product LaTeXoutput using the `write tex` command.

Typesetting Comment - HTML

The key kinds of HTML definitions have the following syntax:

`htmldef "`*math-token*`" as "`HTML-*code*`"; ...`
`althtmldef "`*math-token*`" as "`HTML-*code*`";`
...

Note that in HTML there are two possible definitions for math tokens. This feature is useful when an alternate representation of symbols is desired, for example one that uses Unicode entities and another uses GIF images.

There are many other typesetting definitions that can control HTML. These include:

`htmldef "`*math-token*`" as "`HTML-*code*`";`
`htmltitle "`HTML-*code*`";`
`htmlhome "`HTML-*code*`";`
`htmlvarcolor "`HTML-*code*`";`
`htmlbibliography "`*filename*`";`

The `htmltitle` is the HTML code for a common title, such as "Metamath Proof Explorer." The `htmlhome` is code for a link back to the home page. The `htmlvarcolor` is code for a color key that appears at the bottom of each proof. The file specified by *filename* is an HTML file that is assumed to have a `<A NAME=...>` tag for each bibiographic reference in the database comments. For example, if `[Monk]` occurs in the comment for a theorem, then `<A NAME='Monk'>` must be present in the file; if not, a warning message is given.

Associated with `althtmldef` are the statements

`htmldir "`*directoryname*`";`
`althtmldir "`*directoryname*`";`

giving the directories of the GIF and Unicode versions respectively; their purpose is to provide cross-linking between the two versions in the generated web pages.

When two different types of pages need to be produced from a single database, such as the Hilbert Space Explorer that extends the Metamath Proof Explorer, "extended" variables may be declared in the `$t` comment:

`exthtmltitle "`HTML-*code*`";`
`exthtmlhome "`HTML-*code*`";`
`exthtmlbibliography "`*filename*`";`

When these are declared, you also must declare

`exthtmllabel "`*label*`";`

that identifies the database statement where the "extended" section of the database starts (in our example, where the Hilbert Space Explorer starts). During the generation of web pages for that starting statement and the statements after it, the HTML code assigned to `exthtmltitle` and `exthtmlhome` is used instead of that assigned to `htmltitle` and `htmlhome`, respectively.

4.4.3 Additional Information Comment (`$j`)

The additional information comment, aka the `$j` comment, provides a way to add additional structured information that can be optionally parsed by systems.

The additional information comment is parsed the same way as the typesetting comment (`$t`) (see section 4.4.2). That is, the additional information comment begins with the token `$j` within a comment, and continues until the comment close `$)`. Within an additional information comment is a sequence of one or more commands of the form `command arg arg ... ;` where each of the zero or more `arg` values can be either a quoted string or a keyword. Note that every command ends in an unquoted semicolon. If a verifier is parsing an additional information comment, but doesn't recognize a particular command, it must skip the command by finding the end of the command (an unquoted semicolon).

A database may have 0 or more additional information comments. Note, however, that a verifier may ignore these comments entirely or only process certain commands in an additional information comment. The `mmj2` verifier supports many commands in additional information comments. We encourage systems that process additional information comments to coordinate so that they will use the same command for the same effect.

Examples of additional information comments with various commands (from the `set.mm` database) are:

- Define the syntax and logical typecodes, and declare that our grammar is unambiguous (verifiable using the KLR parser, with compositing depth 5).

```
$( $j
  syntax 'wff';
  syntax '|-' as 'wff';
  unambiguous 'klr 5';
$)
```

- Register $\neg$ and $\rightarrow$ as primitive expressions (lacking definitions).

```
$( $j primitive 'wn' 'wi'; $)
```

- There is a special justification for `df-bi`.

```
$( $j justification 'bijust' for 'df-bi'; $)
```

- Register $\leftrightarrow$ as an equality for its type (wff).

```
$( $j
  equality 'wb' from 'biid' 'bicomi' 'bitri';
  definition 'dfbi1' for 'wb';
$)
```

- Theorem `notbii` is the congruence law for negation.

```
$( $j congruence 'notbii'; $)
```

- Add `setvar` as a typecode.

```
$( $j syntax 'setvar'; $)
```

- Register $=$ as an equality for its type (`class`).

```
$( $j equality 'wceq' from 'eqid' 'eqcomi' 'eqtri'; $)
```

4.4.4 Including Other Files in a Metamath Source File

The keywords `$[` and `$]` specify a file to be included at that point in a Metamath source file. The syntax for including a file is as follows:

`$[` *file-name* `$]`

The *file-name* should be a single token with the same syntax as a math symbol (i.e., all 93 non-whitespace printable characters other than `$` are allowed, subject to the file-naming limitations of your operating system). Comments may appear between the `$[` and `$]` keywords. Included files may include other files, which may in turn include other files, and so on.

For example, suppose you want to use the set theory database as the starting point for your own theory. The first line in your file could be

```
$[ set.mm $]
```

All of the information (axioms, theorems, etc.) in `set.mm` and any files that *it* includes will become available for you to reference in your file. This can help make your work more modular. A drawback to including files is that if you change the name of a symbol or the label of a statement, you must also remember to update any references in any file that includes it.

The naming conventions for included files are the same as those of your operating system.[17] For compatibility among operating systems, you should keep the file names as simple as possible. A good convention to use is *file*`.mm` where *file* is eight characters or less, in lower case.

There is no limit to the nesting depth of included files. One thing that you should be aware of is that if two included files themselves include a common third file, only the *first* reference to this common file will be read in. This allows you to include two or more files that build on a common starting file without having to worry about label and symbol conflicts that would occur if the common file were read in more than once. (In fact, if a file includes itself, the self-reference will be ignored, although of course it would not make any sense to do that.) This feature also means, however, that if you try to include a common file in several inner blocks, the result might not be what you expect, since only the first reference will be replaced with the included file (unlike the include statement in most other computer languages). Thus you would normally include common files only in the outermost block.

4.4.5 Compressed Proof Format

The proof notation presented in Section 4.3 is called a **normal proof** and in principle is sufficient to express any proof. However, proofs often contain steps and subproofs that are identical. This is particularly true in typical Metamath applications, because Metamath requires that the math symbol sequence (usually containing a formula) at each step be separately constructed, that is, built up piece by piece. As a result, a lot of repetition often results. The **compressed proof** format allows Metamath to take advantage of this redundancy to shorten proofs.

The specification for the compressed proof format is given in Appendix B.

Normally you need not concern yourself with the details of the compressed proof format, since the Metamath program will allow you to convert from the normal format to the compressed format with ease, and will also automatically convert from the compressed format when proofs are displayed. The overall structure of the compressed format is as follows:

`$=` `(` *label-list* `)` *compressed-proof* `$.`

The first `(` serves as a flag to Metamath that a compressed proof follows. The *label-list* includes all statements referred to by the proof except the mandatory hypotheses. The *compressed-proof* is a compact encoding of the

[17] On the Macintosh, prior to Mac OS X, a colon is used to separate disk and folder names from your file name. For example, *volume*:*file-name* refers to the root directory, *volume*:*folder-name*:*file-name* refers to a folder in root, and *volume*:*folder-name*:...:*file-name* refers to a deeper folder. A simple *file-name* refers to a file in the folder from which you launch the Metamath application. Under Mac OS X and later, the Metamath program is run under the Terminal application, which conforms to Unix naming conventions.

proof, using upper-case letters, and can be thought of as a large integer in base 26. White space inside a *compressed-proof* is optional and is ignored.

It is important to note that the order of the mandatory hypotheses of the statement being proved must not be changed if the compressed proof format is used, otherwise the proof will become incorrect. The reason for this is that the mandatory hypotheses are not mentioned explicitly in the compressed proof in order to make the compression more efficient. If you wish to change the order of mandatory hypotheses, you must first convert the proof back to normal format using the `save proof` *`statement`* `/normal` command. Later, you can go back to compressed format with `save proof` *`statement`* `/compressed`.

During error checking with the `verify proof` command, an error found in a compressed proof may point to a character in *compressed-proof*, which may not be very meaningful to you. In this case, try to `save proof /normal` first, then do the `verify proof` again. In general, it is best to make sure a proof is correct before saving it in compressed format, because severe errors are less likely to be recoverable than in normal format.

4.4.6 Specifying Unknown Proofs or Subproofs

In a proof under development, any step or subproof that is not yet known may be represented with a single `?`. For the purposes of parsing the proof, the `?` will push a single entry onto the RPN stack just as if it were a hypothesis. While developing a proof with the Proof Assistant, a partially developed proof may be saved with the `save new_proof` command, and `?`'s will be placed at the appropriate places.

All `$p` statements must have proofs, even if they are entirely unknown. Before creating a proof with the Proof Assistant, you should specify a completely unknown proof as follows:

label `$p` *statement* `$= ? $.`

The `verify proof` command will check the known portions of a partial proof for errors, but will warn you that the statement has not been proved.

Note that partially developed proofs may be saved in compressed format if desired. In this case, you will see one or more `?`'s in the *compressed-proof* part.

4.5 Axioms vs. Definitions

The *basic* Metamath language and program make no distinction between axioms and definitions. The `$a` statement is used for both. At first, this may seem puzzling. In the minds of many mathematicians, the distinction is clear, even obvious, and hardly worth discussing. A definition is considered to be merely an abbreviation that can be replaced by the expression for which it

stands; although unless one actually does this, to be precise then one should say that a theorem is a consequence of the axioms *and* the definitions that are used in the formulation of the theorem [4, p. 20].

4.5.1 What is a Definition?

What is a definition? In its simplest form, a definition introduces a new symbol and provides an unambiguous rule to transform an expression containing the new symbol to one without it. The concept of a "proper definition" (as opposed to a creative definition) that is usually agreed upon is (1) the definition should not strengthen the language and (2) any symbols introduced by the definition should be eliminable from the language [49]. In other words, they are mere typographical conveniences that do not belong to the system and are theoretically superfluous. This may seem obvious, but in fact the nature of definitions can be subtle, sometimes requiring difficult metatheorems to establish that they are not creative.

A more conservative stance was taken by logician S. Leśniewski.

> Leśniewski regards definitions as theses of the system. In this respect they do not differ either from the axioms or from theorems, i.e. from the theses added to the system on the basis of the rule of substitution or the rule of detachment [modus ponens]. Once definitions have been accepted as theses of the system, it becomes necessary to consider them as true propositions in the same sense in which axioms are true [37].

Let us look at some simple examples of definitions in propositional calculus. Consider the definition of logical OR (disjunction): "$P \vee Q$ denotes $\neg P \to Q$ (not P implies Q)." It is very easy to recognize a statement making use of this definition, because it introduces the new symbol $\vee$ that did not previously exist in the language. It is easy to see that no new theorems of the original language will result from this definition.

Next, consider a definition that eliminates parentheses: "$P \to Q \to R$ denotes $P \to (Q \to R)$." This is more subtle, because no new symbols are introduced. The reason this definition is considered proper is that no new symbol sequences that are valid wffs (well-formed formulas) in the original language will result from the definition, since "$P \to Q \to R$" is not a wff in the original language. Here, we implicitly make use of the fact that there is a decision procedure that allows us to determine whether or not a symbol sequence is a wff, and this fact allows us to use symbol sequences that are not wffs to represent other things (such as wffs) by means of the definition. However, to justify the definition as not being creative we need to prove that "$P \to Q \to R$" is in fact not a wff in the original language, and this is more difficult than in the case where we simply introduce a new symbol.

What constitutes a definition versus an axiom is sometimes arbitrary in mathematical literature. For example, the connectives $\vee$ (OR), $\wedge$ (AND), and

↔ (equivalent to) in propositional calculus are usually considered defined symbols that can be used as abbreviations for expressions containing the "primitive" connectives → and ¬. This is the way we treat them in the standard logic and set theory database `set.mm`. However, the first three connectives can also be considered "primitive," and axiom systems have been devised that treat all of them as such. For example, [19, p. 35] presents one with 15 axioms, some of which in fact coincide with what we have chosen to call definitions in `set.mm`. In certain subsets of classical propositional calculus, such as the intuitionist fragment, it can be shown that one cannot make do with just → and ¬ but must treat additional connectives as primitive in order for the system to make sense.[18]

4.5.2 The Approach to Definitions in `set.mm`

In set theory, recursive definitions define a newly introduced symbol in terms of itself. The justification of recursive definitions, using several "recursion theorems," is usually one of the first sophisticated proofs a student encounters when learning set theory, and there is a significant amount of implicit metalogic behind a recursive definition even though the definition itself is typically simple to state.

Metamath itself has no built-in technical limitation that prevents multiple-part recursive definitions in the traditional textbook style. However, because the recursive definition requires advanced metalogic to justify, eliminating a recursive definition is very difficult and often not even shown in textbooks.

Direct definitions instead of recursive definitions

It is, however, possible to substitute one kind of complexity for another. We can eliminate the need for metalogical justification by defining the operation directly with an explicit (but complicated) expression, then deriving the recursive definition directly as a theorem, using a recursion theorem "in reverse." The elimination of a direct definition is a matter of simple mechanical substitution. We do this in `set.mm`, as follows.

In `set.mm` our goal was to introduce almost all definitions in the form of two expressions connected by either ↔ or =, where the thing being defined does not appear on the right hand side. Quine calls this form "a genuine or direct definition" [55, p. 174], which makes the definitions very easy to eliminate and the metalogic needed to justify them as simple as possible. Put another way, we had a goal of being able to eliminate all definitions with direct mechanical substitution and to verify easily the soundness of the definitions.

[18] Two nice systems that make the transition from intuitionistic and other weak fragments to classical logic just by adding axioms are given in [57].

Example of direct definitions

We achieved this goal in almost all cases in `set.mm`. Sometimes this makes the definitions more complex and less intuitive. For example, the traditional way to define addition of natural numbers is to define an operation called *successor* (which means "plus one" and is denoted by "suc"), then define addition recursively with the two definitions $n + 0 = n$ and $m + \mathrm{suc}\, n = \mathrm{suc}(m + n)$. Although this definition seems simple and obvious, the method to eliminate the definition is not obvious: in the second part of the definition, addition is defined in terms of itself. By eliminating the definition, we don't mean repeatedly applying it to specific m and n but rather showing the explicit, closed-form set-theoretical expression that $m + n$ represents, that will work for any m and n and that does not have a + sign on its right-hand side. For a recursive definition like this not to be circular (creative), there are some hidden, underlying assumptions we must make, for example that the natural numbers have a certain kind of order.

In `set.mm` we chose to start with the direct (though complex and nonintuitive) definition then derive from it the standard recursive definition. For example, the closed-form definition used in `set.mm` for the addition operation on ordinals (of which natural numbers are a subset) is

`df-oadd $a` $\vdash +_o = (x \in \mathrm{On} , y \in \mathrm{On} \mapsto (\mathrm{rec} ((z \in \mathrm{V} \mapsto \mathrm{suc}\, z) , x) \mathbin{`} y))$

which depends on rec.

Recursion operators

The above definition of `df-oadd` depends on the definition of rec, a "recursion operator" with the definition `df-rdg`:

`df-rdg $a` $\vdash \mathrm{rec} (F , I) = \mathrm{recs} ((g \in \mathrm{V} \mapsto \mathrm{if} (g = \varnothing , I , \mathrm{if} (\mathrm{Lim}\, \mathrm{dom}\, g , \bigcup \mathrm{ran}\, g , (F \mathbin{`} ( g \mathbin{`} \bigcup \mathrm{dom}\, g))))))$

which can be further broken down with definitions shown in Section 3.4.3.

This definition of rec defines a recursive definition generator on On (the class of ordinal numbers) with characteristic function F and initial value I. This operation allows us to define, with compact direct definitions, functions that are usually defined in textbooks with recursive definitions. The price paid with our approach is the complexity of our rec operation (especially when `df-recs` that it is built on is also eliminated). But once we get past this hurdle, definitions that would otherwise be recursive become relatively simple, as in for example `oav`, from which we prove the recursive textbook definition as theorems `oa0`, `oasuc`, and `oalim` (with the help of theorems `rdg0`, `rdgsuc`, and `rdglim2a`). We can also restrict the rec operation to define otherwise recursive functions on the natural numbers ω; see `fr0g` and `frsuc`. Our rec operation apparently does not appear in published literature, although closely related is Definition 25.2 of [Quine] p. 177, which he uses to "turn...a recursion into a genuine or direct definition" (p. 174). Note that the if operations (see `df-if`) select cases based on whether the domain of g

is zero, a successor, or a limit ordinal.

An important use of this definition rec is in the recursive sequence generator `df-seq` on the natural numbers (as a subset of the complex infinite sequences such as the factorial function `df-fac` and integer powers `df-exp`).

The definition of rec depends on recs. New direct usage of the more powerful (and more primitive) recs construct is discouraged, but it is available when needed. This defines a function recs(F) on On, the class of ordinal numbers, by transfinite recursion given a rule F which sets the next value given all values so far. Unlike `df-rdg` which restricts the update rule to use only the previous value, this version allows the update rule to use all previous values, which is why it is described as "strong," although it is actually more primitive. See `recsfnon` and `recsval` for the primary contract of this definition. It is defined as:

`df-recs $a` $\vdash \mathrm{recs}(F) = \bigcup \{ f \mid \exists x \in \mathrm{On} (f \mathrm{Fn} x \wedge \forall y \in x (f `y ) = ( F ` (f \restriction y))) \}$

Closing comments on direct definitions

From these direct definitions the simpler, more intuitive recursive definition is derived as a set of theorems. The end result is the same, but we completely eliminate the rather complex metalogic that justifies the recursive definition.

Recursive definitions are often considered more efficient and intuitive than direct ones once the metalogic has been learned or possibly just accepted as correct. However, it was felt that direct definition in `set.mm` maximizes rigor by minimizing metalogic. It can be eliminated effortlessly, something that is difficult to do with a recursive definition.

Again, Metamath itself has no built-in technical limitation that prevents multiple-part recursive definitions in the traditional textbook style. Instead, our goal is to eliminate all definitions with direct mechanical substitution and to verify easily the soundness of definitions.

4.5.3 Adding Constraints on Definitions

The basic Metamath language and the Metamath program do not have any built-in constraints on definitions, since they are just $a statements.

However, nothing prevents a verification system from verifying additional rules to impose further limitations on definitions. For example, the `mmj2` program supports various kinds of additional information comments (see section 4.4.3). One of their uses is to optionally verify additional constraints, including constraints to verify that definitions meet certain requirements. These additional checks are required by the continuous integration (CI) checks of the `set.mm` database. This approach enables us to optionally impose additional requirements on definitions if we wish, without necessarily imposing those rules on all databases or requiring all verification systems to implement them. In addition, this allows us to impose specialized constraints

tailored to one database while not requiring other systems to implement those specialized constraints.

We impose two constraints on the `set.mm` database via the `mmj2` program that are worth discussing here: a parse check and a definition soundness check.

First, we enable a parse check in `mmj2` (through its `SetParser` command) that requires that all new definitions must *not* create an ambiguous parse for a KLR(5) parser. This prevents some errors such as definitions with imbalanced parentheses.

Second, we run a definition soundness check specific to `set.mm` or databases similar to it. (through the `definitionCheck` macro). Some `$a` statements (including all ax-* statemnets) are excluded from these checks, as they will always fail this simple check, but they are appropriate for most definitions. This check imposes a set of additional rules:

1. New definitions must be introduced using $=$ or $\leftrightarrow$.
2. No `$a` statement introduced before this one is allowed to use the symbol being defined in this definition, and the definition is not allowed to use itself (except once, in the definiendum).
3. Every variable in the definiens should not be distinct
4. Every dummy variable in the definiendum are required to be distinct from each other and from variables in the definiendum. To determine this, the system will look for a "justification" theorem in the database, and if it is not there, attempt to briefly prove $(\varphi \rightarrow \forall x \varphi)$ for each dummy variable x.
5. Every dummy variable should be a set variable, unless there is a justification theorem available.
6. Every dummy variable must be bound (if the system cannot determine this a justification theorem must be provided).

4.5.4 Summary of Approach to Definitions

In short, when being rigorous it turns out that definitions can be subtle, sometimes requiring difficult metatheorems to establish that they are not creative.

Instead of building such complications into the Metamath language itself, the basic Metmath language and program simply treat traditional axioms and definitions as the same kind of `$a` statement. We have then built various tools to enable people to verify additional conditions as their creators believe is appropriate for those specific databases, without complicating the Metamath foundations.

Chapter 5

The Metamath Program

This chapter provides a reference manual for the Metamath program.

Current instructions for obtaining and installing the Metamath program can be found at the `http://metamath.org` web site. For Windows, there is a pre-compiled version called `metamath.exe`. For Unix, Linux, and Mac OS X (which we will refer to collectively as "Unix"), the Metamath program can be compiled from its source code with the command

```
gcc *.c -o metamath
```

using the `gcc` C compiler available on those systems.

In the command syntax descriptions below, fields enclosed in square brackets [ ] are optional. File names may be optionally enclosed in single or double quotes. This is useful if the file name contains spaces or slashes (`/`), such as in Unix path names, that might be confused with Metamath command qualifiers.

5.1 Invoking Metamath

Unix, Linux, and Mac OS X have a command-line interface called the *bash shell*. (In Mac OS X, select the Terminal application from Applications/Utilities.) To invoke Metamath from the bash shell prompt, assuming that the Metamath program is in the current directory, type

```
bash$ ./metamath
```

To invoke Metamath from a Windows DOS or Command Prompt, assuming that the Metamath program is in the current directory (or in a directory included in the Path system environment variable), type

```
C:\metamath>metamath
```

To use command-line arguments at invocation, the command-line arguments should be a list of Metamath commands, surrounded by quotes if they contain spaces. In Windows, the surrounding quotes must be double (not single) quotes. For example, to read the database file `set.mm`, verify all proofs, and exit the program, type (under Unix)

```
bash$ ./metamath 'read set.mm' 'verify proof *' exit
```

Note that in Unix, any directory path with /'s must be surrounded by quotes so Metamath will not interpret the / as a command qualifier. So if `set.mm` is in the `/tmp` directory, use for the above example

```
bash$ ./metamath 'read "/tmp/set.mm"' 'verify proof *' exit
```

For convenience, if the command-line has one argument and no spaces in the argument, the command is implicitly assumed to be `read`. In this one special case, /'s are not interpreted as command qualifiers, so you don't need quotes around a Unix file name. Thus

```
bash$ ./metamath /tmp/set.mm
```

and

```
bash$ ./metamath "read '/tmp/set.mm'"
```

are equivalent.

5.2 Controlling Metamath

The Metamath program was first developed on a VAX/VMS system, and some aspects of its command line behavior reflect this heritage. Hopefully you will find it reasonably user-friendly once you get used to it.

Each command line is a sequence of English-like words separated by spaces, as in `show settings`. Command words are not case sensitive, and only as many letters are needed as are necessary to eliminate ambiguity; for example, `sh se` would work for the command `show settings`. In some cases arguments such as file names, statement labels, or symbol names are required; these are case-sensitive (although file names may not be on some operating systems).

A command line is entered by typing it in then pressing the *return* (*enter*) key. To find out what commands are available, type `?` at the `MM>` prompt. To find out the choices at any point in a command, press *return* and you will be prompted for them. The default choice (the one selected if you just press *return*) is shown in brackets (`<>`).

You may also type `?` in place of a command word to force Metamath to tell you what the choices are. The `?` method won't work, though, if a

non-keyword argument such as a file name is expected at that point, because the program will think that `?` is the value of the argument.

Some commands have one or more optional qualifiers which modify the behavior of the command. Qualifiers are preceded by a slash (`/`), such as in `read set.mm / verify`. Spaces are optional around the `/`. If you need to use a space or slash in a command argument, as in a Unix file name, put single or double quotes around the command argument.

The `open log` command will save everything you see on the screen and is useful to help you recover should something go wrong in a proof, or if you want to document a bug.

If a command responds with more than a screenful, you will be prompted to `<return> to continue, Q to quit, or S to scroll to end.` `Q` or `q` (not case-sensitive) will complete the command internally but will suppress further output until the next `MM>` prompt. `s` will suppress further pausing until the next `MM>` prompt. After the first screen, you are also presented with the choice of `b` to go back a screenful. Note that `b` may also be entered at the `MM>` prompt immediately after a command to scroll back through the output of that command.

A command line enclosed in quotes is executed by your operating system. See Section 5.2.12.

Warning: Pressing CTRL-C will abort the Metamath program unconditionally. This means any unsaved work will be lost.

5.2.1 `exit` Command

Syntax: `exit` [`/force`]

This command exits from Metamath. If there have been changes to the source with the `save proof` or `save new_proof` commands, you will be given an opportunity to `write source` to permanently save the changes.

In Proof Assistant mode, the `exit` command will return to the `MM>` prompt. If there were changes to the proof, you will be given an opportunity to `save new_proof`.

The `quit` command is a synonym for `exit`.

Optional qualifier: `/force` - Do not prompt if changes were not saved. This qualifier is useful in `submit` command files (Section 5.2.4) to ensure predictable behavior.

5.2.2 `open log` Command

Syntax: `open log` *file-name*

This command will open a log file that will store everything you see on the screen. It is useful to help recovery from a mistake in a long Proof Assistant session, or to document bugs.

The log file can be closed with `close log`. It will automatically be closed upon exiting Metamath.

5.2.3 `close log` Command

Syntax: `close log`

The `close log` command closes a log file if one is open. See also `open log`.

5.2.4 `submit` Command

Syntax: `submit` *filename*

This command causes further command lines to be taken from the specified file. Note that any line beginning with an exclamation point (`!`) is treated as a comment (i.e. ignored). Also note that the scrolling of the screen output is continuous, so you may want to open a log file (see `open log`) to record the results that fly by on the screen. After the lines in the file are exhausted, Metamath returns to its normal user interface mode.

The `submit` command can be called recursively (i.e. from inside of a `submit` command file).

Optional command qualifier:

`/silent` - suppresses the screen output but still records the output in a log file if one is open.

5.2.5 `erase` Command

Syntax: `erase`

This command will reset Metamath to its starting state, deleting any database that was `read` in. If there have been changes to the source with the `save proof` or `save new_proof` commands, you will be given an opportunity to `write source` to permanently save the changes.

5.2.6 `set echo` Command

Syntax: `set echo on` or `set echo off`

The `set echo on` command will cause command lines to be echoed with any abbreviations expanded. While learning the Metamath commands, this feature will show you the exact command that your abbreviated input corresponds to.

5.2.7 `set scroll` Command

Syntax: `set scroll prompted` or `set scroll continuous`

The Metamath command line interface starts off in the `prompted` mode, which means that you will be prompted to continue or quit after each full screen in a long listing. In `continuous` mode, long listings will be scrolled without pausing.

5.2.8 `set width` Command

Syntax: `set width` *number*

Metamath assumes the width of your screen is 79 characters (chosen because the Command Prompt in Windows XP has a wrapping bug at column 80). If your screen is wider or narrower, this command allows you to change this default screen width. A larger width is advantageous for logging proofs to an output file to be printed on a wide printer. A smaller width may be necessary on some terminals; in this case, the wrapping of the information messages may sometimes seem somewhat unnatural, however. In LaTeX, there is normally a maximum of 61 characters per line with typewriter font. (The examples in this book were produced with 61 characters per line.)

5.2.9 `set height` Command

Syntax: `set height` *number*

Metamath assumes your screen height is 24 lines of characters. If your screen is taller or shorter, this command lets you to change the number of lines at which the display pauses and prompts you to continue.

5.2.10 `beep` Command

Syntax: `beep`

This command will produce a beep. By typing it ahead after a long-running command has started, it will alert you that the command is finished. For convenience, `b` is an abbreviation for `beep`.

Note: If `b` is typed at the `MM>` prompt immediately after the end of a multiple-page display paged with "`Press <return> for more...`" prompts, then the `b` will back up to the previous page rather than perform the `beep` command. In that case you must type the unabbreviated `beep` form of the command.

5.2.11 `more` Command

Syntax: `more` *filename*

This command will display the contents of an ASCII file on your screen. (This command is provided for convenience but is not very powerful. See Section 5.2.12 to invoke your operating system's command to do this, such as the `more` command in Unix.)

5.2.12 Operating System Commands

A line enclosed in single or double quotes will be executed by your computer's operating system if it has a command line interface. For example, on a VAX/VMS system, `MM> 'dir'` will print disk directory contents. Note that

this feature will not work on the Macintosh prior to Mac OS X, which does not have a command line interface.

For your convenience, the trailing quote is optional.

5.2.13 Size Limitations in Metamath

In general, there are no fixed, predefined limits on how many labels, tokens, statements, etc. that you may have in a database file. The Metamath program uses 32-bit variables (64-bit on 64-bit CPUs) as indices for almost all internal arrays, which are allocated dynamically as needed.

5.3 Reading and Writing Files

The following commands create new files: the `open` commands; the `write` commands; the `/html`, `/alt_html`, `/brief_html`, `/brief_alt_html` qualifiers of `show statement`, and `midi`. The following commands append to files previously opened: the `/tex` qualifier of `show proof` and `show new_proof`; the `/tex` and `/simple_tex` qualifiers of `show statement`; the `close` commands; and all screen dialog between `open log` and `close log`.

The commands that create new files will not overwrite an existing *filename* but will rename the existing one to *filename*`~1`. An existing *filename*`~1` is renamed *filename*`~2`, etc. up to *filename*`~9`. An existing *filename*`~9` is deleted. This makes recovery from mistakes easier but also will clutter up your directory, so occasionally you may want to clean up (delete) these `~`*n* files.

5.3.1 `read` Command

Syntax: `read` *file-name* [`/verify`]

This command will read in a Metamath language source file and any included files. Normally it will be the first thing you do when entering Metamath. Statement syntax is checked, but proof syntax is not checked. Note that the file name may be enclosed in single or double quotes; this is useful if the file name contains slashes, as might be the case under Unix.

If you are getting an "`?Expected VERIFY`" error when trying to read a Unix file name with slashes, you probably haven't quoted it.

If you are prompted for the file name (by pressing *return* after `read`) you should *not* put quotes around it, even if it is a Unix file name with slashes.

Optional command qualifier:

`/verify` - Verify all proofs as the database is read in. This qualifier will slow down reading in the file. See `verify proof` for more information on file error-checking.

See also `erase`.

5.3.2 `write source` Command

Syntax: `write source` *filename* [`/rewrap`] [`/split`] [`/keep_includes`] [`/no_versioning`]

This command will write the contents of a Metamath database into a file.

Optional command qualifiers:

`/rewrap` - Reformats statements and comments according to the convention used in the set.mm database. It unwraps the lines in the comment before each `$a` and `$p` statement, then it rewraps the line. You should compare the output to the original to make sure that the desired effect results; if not, go back to the original source. The wrapped line length honors the `set width` parameter currently in effect. Note: Text enclosed in `<HTML>...</HTML>` tags is not modified by the `/rewrap` qualifier. Proofs themselves are not reformatted; use `save proof * / compressed` to do that. An isolated tilde (~) is always kept on the same line as the following symbol, so you can find all comment references to a symbol by searching for ~ followed by a space and that symbol (this is useful for finding cross-references). Incidentally, `save proof` also honors the `set width` parameter currently in effect.

`/split` - Files included in the source using the expression $[*inclfile* $] will be written into separate files instead of being included in a single output file. The name of each separately written included file will be the *inclfile* argument of its inclusion command.

`/keep_includes` - If a source file has includes but is written as a single file by omitting `/split`, by default the included files will be deleted (actually just renamed with a ~1 suffix unless `/no_versioning` is specified) to prevent the possibly confusing source duplication in both the output file and the included file. The `/keep_includes` qualifier will prevent this deletion.

`/no_versioning` - Backup files suffixed with ~1 are not created.

5.4 Showing Status and Statements

5.4.1 `show settings` Command

Syntax: `show settings`

This command shows the state of various parameters.

5.4.2 `show memory` Command

Syntax: `show memory`

This command shows the available memory left. It is not meaningful on most modern operating systems, which have virtual memory.

5.4.3 `show labels` Command

Syntax: `show labels` *label-match* [`/all`] [`/linear`]

This command shows the labels of `$a` and `$p` statements that match *label-match*. A `*` in label-match matches zero or more characters. For example, `*abc*def` will match all labels containing `abc` and ending with `def`.

Optional command qualifiers:

`/all` - Include matches for `$e` and `$f` statement labels.

`/linear` - Display only one label per line. This can be useful for building scripts in conjunction with the utilities under the `tools` command.

5.4.4 `show statement` Command

Syntax: `show statement` *label-match* [*qualifiers* (see below)]

This command provides information about a statement. Only statements that have labels (`$f`, `$e`, `$a`, and `$p`) may be specified. If *label-match* contains wildcard (`*`) characters, all matching statements will be displayed in the order they occur in the database.

Optional qualifiers (only one qualifier at a time is allowed):

`/comment` - This qualifier includes the comment that immediately precedes the statement.

`/full` - Show complete information about each statement, and show all statements matching *label* (including `$e` and `$f` statements).

`/tex` - This qualifier will write the statement information to the LaTeX file previously opened with `open tex`. See Section 5.7.

`/simple_tex` - The same as `/tex`, except that LaTeX macros are not used for formatting equations, allowing easier manual edits of the output for slide presentations, etc.

`/html`, `/alt_html`, `/brief_html`, `/brief_alt_html` - These qualifiers invoke a special mode of `show statement` that creates a web page for the statement. They may not be used with any other qualifier. See Section 5.8 or `help html` in the program.

5.4.5 `search` Command

Syntax: search *label-match* "*symbol-match*" [`/all`] [`/comments`] [`/join`]

This command searches all `$a` and `$p` statements matching *label-match* for occurrences of *symbol-match*. A `*` in *label-match* matches any label character. A `$*` in *symbol-match* matches any sequence of symbols. The symbols in *symbol-match* must be separated by white space. The quotes surrounding *symbol-match* may be single or double quotes. For example, `search b* "-> $* ch"` will list all statements whose labels begin with `b` and contain the symbols `->` and `ch` surrounding any symbol sequence (including no symbol sequence). The wildcards `?` and `$?` are also available to match individual characters in labels and symbols respectively; see `help search` in the Metamath program for details on their usage.

Optional command qualifiers:

`/all` - Also search `$e` and `$f` statements.

`/comments` - Search the comment that immediately precedes each label-matched statement for *symbol-match*. In this case *symbol-match* is an arbitrary, non-case-sensitive character string. Quotes around *symbol-match* are optional if there is no ambiguity.

`/join` - In the case of a `$a` or `$p` statement, prepend its `$e` hypotheses for searching. The `/join` qualifier has no effect in `/comments` mode.

5.5 Displaying and Verifying Proofs

5.5.1 `show proof` Command

Syntax: `show proof` *label-match* [*qualifiers* (see below)]

This command displays the proof of the specified `$p` statement in various formats. The *label-match* may contain wildcard (`$*`) characters to match multiple statements. Without any qualifiers, only the logical steps will be shown (i.e. syntax construction steps will be omitted), in an indented format.

Most of the time, you will use `show proof` *label* to see just the proof steps corresponding to logical inferences.

Optional command qualifiers:

`/essential` - The proof tree is trimmed of all `$f` hypotheses before being displayed. (This is the default, and it is redundant to specify it.)

`/all` - the proof tree is not trimmed of all `$f` hypotheses before being displayed. `/essential` and `/all` are mutually exclusive.

`/from_step` *step* - The display starts at the specified step. If this qualifier is omitted, the display starts at the first step.

`/to_step` *step* - The display ends at the specified step. If this qualifier is omitted, the display ends at the last step.

`/tree_depth` *number* - Only steps at less than the specified proof tree depth are displayed. Sometimes useful for obtaining an overview of the proof.

`/reverse` - The steps are displayed in reverse order.

`/renumber` - When used with `/essential`, the steps are renumbered to correspond only to the essential steps.

`/tex` - The proof is converted to LaTeX and stored in the file opened with `open tex`. See Section 5.7 or `help tex` in the program.

`/lemmon` - The proof is displayed in a non-indented format known as Lemmon style, with explicit previous step number references. If this qualifier is omitted, steps are indented in a tree format.

`/start_column` *number* - Overrides the default column (16) at which the formula display starts in a Lemmon-style display. May be used only in conjunction with `/lemmon`.

`/normal` - The proof is displayed in normal format suitable for inclusion in a Metamath source file. May not be used with any other qualifier.

`/compressed` - The proof is displayed in compressed format suitable for inclusion in a Metamath source file. May not be used with any other qualifier.

`/statement_summary` - Summarizes all statements (like a brief `show statement`) used by the proof. It may not be used with any other qualifier except `/essential`.

`/detailed_step` *step* - Shows the details of what is happening at a specific proof step. May not be used with any other qualifier. The *step* is the step number shown when displaying a proof without the `/renumber` qualifier.

5.5.2 `show usage` Command

Syntax: `show usage` *label-match* [`/recursive`]

This command lists the statements whose proofs make direct reference to the statement specified.

Optional command qualifier:

`/recursive` - Also include statements whose proofs ultimately depend on the statement specified.

5.5.3 `show trace_back` Command

Syntax: `show trace_back` *label-match* [`/essential`] [`/axioms`] [`/tree`] [`/depth` *number*]

This command lists all statements that the proof of the `$p` statement(s) specified by *label-match* depends on.

Optional command qualifiers:

`/essential` - Restrict the trace-back to `$e` hypotheses of proof trees.

`/axioms` - List only the axioms that the proof ultimately depends on.

`/tree` - Display the trace-back in an indented tree format.

`/depth` *number* - Restrict the `/tree` trace-back to the specified indentation depth.

`/count_steps` - Count the number of steps the proof has all the way back to axioms. If `/essential` is specified, expansions of variable-type hypotheses (syntax constructions) are not counted.

5.5.4 `verify proof` Command

Syntax: `verify proof` *label-match* [`/syntax_only`]

This command verifies the proofs of the specified statements. *label-match* may contain wild card characters (`*`) to verify more than one proof; for example `*abc*def` will match all labels containing `abc` and ending with `def`. The command `verify proof *` will verify all proofs in the database.

Optional command qualifier:

`/syntax_only` - This qualifier will perform a check of syntax and RPN stack violations only. It will not verify that the proof is correct. This qualifier is useful for quickly determining which proofs are incomplete (i.e. are under development and have `?`'s in them).

Note: `read`, followed by `verify proof *`, will ensure the database is free from errors in the Metamath language but will not check the markup notation in comments. You can also check the markup notation by running `verify markup *` as discussed in Section 5.5.5; see also the discussion on generating HTML in Section 5.8.

5.5.5 `verify markup` Command

Syntax: `verify markup` *label-match* [`/date_skip`] [`/top_date_skip`] [`/file_skip`] [`/verbose`]

This command checks comment markup and other informal conventions we have adopted. It error-checks the latexdef, htmldef, and althtmldef statements in the `$t` statement of a Metamath source file. It error-checks any `‘...‘`, `~` *label*, and bibliographic markups in statement descriptions. It checks that `$p` and `$a` statements have the same content when their labels start with "ax" and "ax-" respectively but are otherwise identical, for example ax4 and ax-4. It also verifies the date consistency of "(Contributed by...)," "(Revised by...)," and "(Proof shortened by...)" tags in the comment above each `$a` and `$p` statement.

Optional command qualifiers:

`/date_skip` - This qualifier will skip date consistency checking, which is usually not required for databases other than `set.mm`.

`/top_date_skip` - This qualifier will check date consistency except that the version date at the top of the database file will not be checked. Only one of `/date_skip` and `/top_date_skip` may be specified.

`/file_skip` - This qualifier will skip checks that require external files to be present, such as checking GIF existence and bibliographic links to mmset.html or equivalent. It is useful for doing a quick check from a directory without these files.

`/verbose` - Provides more information. Currently it provides a list of axXXX vs. ax-XXX matches.

5.5.6 `save proof` Command

Syntax: `save proof` *label-match* [`/normal`] [`/compressed`]

The `save proof` command will reformat a proof in one of two formats and replace the existing proof in the source buffer. It is useful for converting between proof formats. Note that a proof will not be permanently saved until a `write source` command is issued.

Optional command qualifiers:

`/normal` - The proof is saved in the normal format (i.e., as a sequence of labels, which is the defined format of the basic Metamath language). This is the default format that is used if a qualifier is omitted.

`/compressed` - The proof is saved in the compressed format which reduces storage requirements for a database. See Appendix B.

5.6 Creating Proofs

Before using the Proof Assistant, you must add a `$p` to your source file (using a text editor) containing the statement you want to prove. Its proof should consist of a single `?`, meaning "unknown step." Example:

```
equid $p x = x $= ? $.
```

To enter the Proof assistant, type `prove` *label*, e.g. `prove equid`. Metamath will respond with the `MM-PA>` prompt.

Proofs are created working backwards from the statement being proved, primarily using a series of `assign` commands. A proof is complete when all steps are assigned to statements and all steps are unified and completely known. During the creation of a proof, Metamath will allow only operations that are legal based on what is known up to that point. For example, it will not allow an `assign` of a statement that cannot be unified with the unknown proof step being assigned.

Important: The Proof Assistant is *not* a tool to help you discover proofs. It is just a tool to help you add them to the database. For a tutorial read Section 2.4. To practice using the Proof Assistant, you may want to `prove` an existing theorem, then delete all steps with `delete all`, then re-create it with the Proof Assistant while looking at its proof display (before deletion). You might want to figure out your first few proofs completely and write them down by hand, before using the Proof Assistant, though not everyone finds that effective.

Important: The `undo` command if very helpful when entering a proof, because it allows you to undo a previously-entered step. In addition, we suggest that you keep track of your work with a log file (`open log`) and save it frequently (`save new_proof`, `write source`). You can use `delete` to reverse an `assign`. You can also do `delete floating_hypotheses`, then `initialize all`, then `unify all /interactive` to reinitialize bad unifications made accidentally or by bad `assign`s. You cannot reverse a `delete` except by a relevant `undo` or using `exit /force` then reentering the Proof Assistant to recover from the last `save new_proof`.

The following commands are available in the Proof Assistant (at the `MM-PA>` prompt) to help you create your proof. See the individual commands for more detail.

> `show new_proof` [`/all`,...] - Displays the proof in progress. You will use this command a lot; see `help show new_proof` to become familiar with its qualifiers. The qualifiers `/unknown` and `/not_unified` are useful for seeing the work remaining to be done. The combination `/all/unknown` is useful for identifying dummy variables that must be assigned, or attempts to use illegal syntax, when `improve all` is unable to complete the syntax constructions. Unknown variables are shown as `$1`, `$2`,...

`assign` *step label* - Assigns an unknown *step* number with the statement specified by *label*.

`let variable` *variable* `= "`*symbol sequence*`"` - Forces a symbol sequence to replace an unknown variable (such as `$1`) in a proof. It is useful for helping difficult unifications, and it is necessary when you have dummy variables that eventually must be assigned a name.

`let step` *step* `= "`*symbol sequence*`"` - Forces a symbol sequence to replace the contents of a proof step, provided it can be unified with the existing step contents. (I rarely use this.)

`unify step` *step* (or `unify all`) - Unifies the source and target of a step. If you specify a specific step, you will be prompted to select among the unifications that are possible. If you specify `all`, all steps with unique unifications, but only those steps, will be unified. `unify all /interactive` goes through all non-unified steps.

`initialize` *step* (or `all`) - De-unifies the target and source of a step (or all steps), as well as the hypotheses of the source, and makes all variables in the source unknown. Useful to recover from an `assign` or `let` mistake that resulted in incorrect unifications.

`delete` *step* (or `all` or `floating_hypotheses`) - Deletes the specified step(s). `delete floating_hypotheses`, then `initialize all`, then `unify all /interactive` is useful for recovering from mistakes where incorrect unifications assigned wrong math symbol strings to variables.

`improve` *step* (or `all`) - Automatically creates a proof for steps (with no unknown variables) whose proof requires no statements with `$e` hypotheses. Useful for filling in proofs of `$f` hypotheses. The `/depth` qualifier will also try statements whose `$e` hypotheses contain no new variables. *Warning:* Save your work (with `save new_proof` then `write source`) before using `/depth = 2` or greater, since the search time grows exponentially and may never terminate in a reasonable time, and you cannot interrupt the search. I have found that it is rare for `/depth = 3` or greater to be useful.

`save new_proof` - Saves the proof in progress in the program's internal database buffer. To save it permanently into the database file, use `write source` after `save new_proof`. To revert to the last `save new_proof`, `exit /force` from the Proof Assistant then re-enter the Proof Assistant.

`match step` *step* (or `match all`) - Shows what statements are possibilities for the `assign` statement. (This command is not very useful in its present form and hopefully will be improved eventually. In the

meantime, use the `search` statement for candidates matching specific math token combinations.)

`minimize_with` - After a proof is complete, this command will attempt to match other database theorems to the proof to see if the proof size can be reduced as a result. See `help minimize_with` in the Metamath program for its usage.

`undo` - Undo the effect of a proof-changing command (all but the `show` and `save` commands above).

`redo` - Reverse the previous `undo`.

The following commands set parameters that may be relevant to your proof. Consult the individual `help set...` commands.

`set unification_timeout`

`set search_limit`

`set empty_substitution` - note that default is `off`

Type `exit` to exit the `MM-PA>` prompt and get back to the `MM>` prompt. Another `exit` will then get you out of Metamath.

5.6.1 `prove` Command

Syntax: `prove` *label*

This command will enter the Proof Assistant, which will allow you to create or edit the proof of the specified statement. The command-line prompt will change from `MM>` to `MM-PA>`.

Note: In the present version (0.177) of Metamath, the Proof Assistant does not verify that `$d` restrictions are met as a proof is being built. After you have completed a proof, you should type `save new_proof` followed by `verify proof` *label* (where *label* is the statement you are proving with the `prove` command) to verify the `$d` restrictions.

See also: `exit`

5.6.2 `set unification_timeout` Command

Syntax: `set unification_timeout` *number*

(This command is available outside the Proof Assistant but affects the Proof Assistant only.)

Sometimes the Proof Assistant will inform you that a unification time-out occurred. This may happen when you try to `unify` formulas with many temporary variables (`$1`, `$2`, etc.), since the time to compute all possible unifications may grow exponentially with the number of variables. If you want Metamath to try harder (and you're willing to wait longer) you may increase this parameter. `show settings` will show you the current value.

5.6.3 `set empty_substitution` Command

Syntax: `set empty_substitution on` or `set empty_substitution off`

(This command is available outside the Proof Assistant but affects the Proof Assistant only.)

The Metamath language allows variables to be substituted with empty symbol sequences. However, in many formal systems this will never happen in a valid proof. Allowing for this possibility increases the likelihood of ambiguous unifications during proof creation. The default is that empty substitutions are not allowed; for formal systems requiring them, you must `set empty_substitution on`. (An example where this must be `on` would be a system that implements a Deduction Rule and in which deductions from empty assumption lists would be permissible. The MIU-system described in Appendix D is another example.) Note that empty substitutions are always permissible in proof verification (VERIFY PROOF...) outside the Proof Assistant. (See the MIU system in the Metamath book for an example of a system needing empty substitutions; another example would be a system that implements a Deduction Rule and in which deductions from empty assumption lists would be permissible.)

It is better to leave this `off` when working with `set.mm`. Note that this command does not affect the way proofs are verified with the `verify proof` command. Outside of the Proof Assistant, substitution of empty sequences for math symbols is always allowed.

5.6.4 `set search_limit` Command

Syntax: `set search_limit` *number*

(This command is available outside the Proof Assistant but affects the Proof Assistant only.)

This command sets a parameter that determines when the `improve` command in Proof Assistant mode gives up. If you want `improve` to search harder, you may increase it. The `show settings` command tells you its current value.

5.6.5 `show new_proof` Command

Syntax: `show new_proof` [*qualifiers* (see below)]

This command (available only in Proof Assistant mode) displays the proof in progress. It is identical to the `show proof` command, except that there is no statement argument (since it is the statement being proved) and the following qualifiers are not available:

`/statement_summary`
`/detailed_step`

Also, the following additional qualifiers are available:

`/unknown` - Shows only steps that have no statement assigned.

`/not_unified` - Shows only steps that have not been unified.

Note that `/essential`, `/depth`, `/unknown`, and `/not_unified` may be used in any combination; each of them effectively filters out additional steps from the proof display.

See also: `show proof`

5.6.6 `assign` Command

Syntax: `assign` *step label* [`/no_unify`]

and: `assign first` *label*

and: `assign last` *label*

This command, available in the Proof Assistant only, assigns an unknown step (one with `?` in the `show new_proof` listing) with the statement specified by *label*. The assignment will not be allowed if the statement cannot be unified with the step.

If `last` is specified instead of *step* number, the last step that is shown by `show new_proof /unknown` will be used. This can be useful for building a proof with a command file (see `help submit`). It also makes building proofs faster when you know the assignment for the last step.

If `first` is specified instead of *step* number, the first step that is shown by `show new_proof /unknown` will be used.

If *step* is zero or negative, the -*step*th from last unknown step, as shown by `show new_proof /unknown`, will be used. `assign -1` *label* will assign the penultimate unknown step, `assign -2` *label* the antepenultimate, and `assign 0` *label* is the same as `assign last` *label*.

Optional command qualifier:

`/no_unify` - do not prompt user to select a unification if there is more than one possibility. This is useful for noninteractive command files. Later, the user can `unify all /interactive`. (The assignment will still be automatically unified if there is only one possibility and will be refused if unification is not possible.)

5.6.7 `match` Command

Syntax: `match step` *step* [`/max_essential_hyp` *number*]

and: `match all` [`/essential`] [`/max_essential_hyp` *number*]

This command, available in the Proof Assistant only, shows what statements can be unified with the specified step(s). *Note:* In its current form, this command is not very useful because of the large number of matches it reports. It may be enhanced in the future. In the meantime, the `search` command can often provide finer control over locating theorems of interest.

Optional command qualifiers:

`/max_essential_hyp` *number* - filters out of the list any statements with more than the specified number of `$e` hypotheses.

`/essential_only` - in the `match all` statement, only the steps that would be listed in the `show new_proof /essential` display are matched.

5.6.8 `let` Command

Syntax: `let variable` *variable* = "*symbol-sequence*"

and: `let step` *step* = "*symbol-sequence*"

These commands, available in the Proof Assistant only, assign a temporary variable or unknown step with a specific symbol sequence. They are useful in the middle of creating a proof, when you know what should be in the proof step but the unification algorithm doesn't yet have enough information to completely specify the temporary variables. A "temporary variable" is one that has the form `$`*nn* in the proof display, such as `$1`, `$2`, etc. The *symbol-sequence* may contain other unknown variables if desired. Examples:

```
let variable $32 = "A = B"
let variable $32 = "A = $35"
let step 10 = '|- x = x'
let step -2 = "|- ( $7 = ph )"
```

Any symbol sequence will be accepted for the `let variable` command. Only those symbol sequences that can be unified with the step will be accepted for `let step`.

The `let` commands "zap" the proof with information that can only be verified when the proof is built up further. If you make an error, the command sequence `delete floating_hypotheses`, `initialize all`, and `unify all /interactive` will undo a bad `let` assignment.

If *step* is zero or negative, the -*step*th from last unknown step, as shown by `show new_proof /unknown`, will be used. The command `let step 0` = "*symbol-sequence*" will use the last unknown step, `let step -1` = "*symbol-sequence*" the penultimate, etc. If *step* is positive, `let step` may be used to assign known (in the sense of having previously been assigned a label with `assign`) as well as unknown steps.

Either single or double quotes can surround the *symbol-sequence* as long as they are different from any quotes inside a *symbol-sequence*. If *symbol-sequence* contains both kinds of quotes, see the instructions at the end of `help let` in the Metamath program.

5.6.9 `unify` Command

Syntax: `unify step` *step*

and: `unify all` [`/interactive`]

These commands, available in the Proof Assistant only, unify the source and target of the specified step(s). If you specify a specific step, you will be prompted to select among the unifications that are possible. If you specify `all`, only those steps with unique unifications will be unified.

Optional command qualifier for `unify all`:

`/interactive` - You will be prompted to select among the unifications that are possible for any steps that do not have unique unifications. (Otherwise `unify all` will bypass these.)

See also `set unification_timeout`. The default is 100000, but increasing it to 1000000 can help difficult cases. Manually assigning some or all of the unknown variables with the `let variable` command also helps difficult cases.

5.6.10 `initialize` Command

Syntax: `initialize step` *step*

and: `initialize all`

These commands, available in the Proof Assistant only, "de-unify" the target and source of a step (or all steps), as well as the hypotheses of the source, and makes all variables in the source and the source's hypotheses unknown. This command is useful to help recover from incorrect unifications that resulted from an incorrect `assign`, `let`, or unification choice. Part or all of the command sequence `delete floating_hypotheses`, `initialize all`, and `unify all /interactive` will recover from incorrect unifications.

See also: `unify` and `delete`

5.6.11 `delete` Command

Syntax: `delete step` *step*

and: `delete all` – *Warning: dangerous!*

and: `delete floating_hypotheses`

These commands are available in the Proof Assistant only. The `delete step` command deletes the proof tree section that branches off of the specified step and makes the step become unknown. `delete all` is equivalent to `delete step` *step* where *step* is the last step in the proof (i.e. the beginning of the proof tree).

In most cases the `undo` command is the best way to undo a previous step. An alternative is to salvage your last `save new_proof` by exiting and reentering the Proof Assistant. For this to work, keep a log file open to record your work and to do `save new_proof` frequently, especially before `delete`.

`delete floating_hypotheses` will delete all sections of the proof that branch off of `$f` statements. It is sometimes useful to do this before an `initialize` command to recover from an error. Note that once a proof step with a `$f` hypothesis as the target is completely known, the `improve` command can usually fill in the proof for that step. Unlike the deletion of logical steps, `delete floating_hypotheses` is a relatively safe command that is usually easy to recover from.

5.6.12 `improve` Command

Syntax: `improve` *step* [`/depth` *number*] [`/no_distinct`]
and: `improve first` [`/depth` *number*] [`/no_distinct`]
and: `improve last` [`/depth` *number*] [`/no_distinct`]
and: `improve all` [`/depth` *number*] [`/no_distinct`]

These commands, available in the Proof Assistant only, try to find proofs automatically for unknown steps whose symbol sequences are completely known. They are primarily useful for filling in proofs of `$f` hypotheses. The search will be restricted to statements having no `$e` hypotheses.

Note: If memory is limited, `improve all` on a large proof may overflow memory. If you use `set unification_timeout 1` before `improve all`, there will usually be sufficient improvement to easily recover and completely `improve` the proof later on a larger computer. Warning: Once memory has overflowed, there is no recovery. If in doubt, save the intermediate proof (`save new_proof` then `write source`) before `improve all`.

If `last` is specified instead of *step* number, the last step that is shown by `show new_proof /unknown` will be used.

If `first` is specified instead of *step* number, the first step that is shown by `show new_proof /unknown` will be used.

If *step* is zero or negative, the -*step*th from last unknown step, as shown by `show new_proof /unknown`, will be used. `improve -1` will use the penultimate unknown step, `improve -2` *label* the antepenultimate, and `improve 0` is the same as `improve last`.

Optional command qualifier:

`/depth` *number* - This qualifier will cause the search to include statements with `$e` hypotheses (but no new variables in the `$e` hypotheses), provided that the backtracking has not exceeded the specified depth. *Warning:* Try `/depth 1`, then `2`, then `3`, etc. in sequence because of possible exponential blowups. Save your work before trying `/depth` greater than `1`!

`/no_distinct` - Skip trial statements that have `$d` requirements. This qualifier will prevent assignments that might violate `$d` requirements but it also could miss possible legal assignments.

See also: `set search_limit`

5.6.13 `save new_proof` Command

Syntax: `save new_proof` *label* [`/normal`] [`/compressed`]

The `save new_proof` command is available in the Proof Assistant only. It saves the proof in progress in the source buffer. `save new_proof` may be used to save a completed proof, or it may be used to save a proof in progress in order to work on it later. If an incomplete proof is saved, any user assignments with `let step` or `let variable` will be lost, as will any ambiguous unifications that were resolved manually. To help make recovery

easier, it can be helpful to `improve all` before `save new_proof` so that the incomplete proof will have as much information as possible.

Note that the proof will not be permanently saved until a `write source` command is issued.

Optional command qualifiers:

`/normal` - The proof is saved in the normal format (i.e., as a sequence of labels, which is the defined format of the basic Metamath language). This is the default format that is used if a qualifier is omitted.

`/compressed` - The proof is saved in the compressed format, which reduces storage requirements for a database. (See Appendix B.)

5.7 Creating LaTeX Output

You can generate LaTeX output given the information in a database. The database must already include the necessary typesetting information (see section 4.4.2 for how to provide this information).

The `show statement` and `show proof` commands each have a special `/tex` command qualifier that produces LaTeX output. (The `show statement` command also has the `/simple_tex` qualifier for output that is easier to edit by hand.) Before you can use them, you must open a LaTeX file to which to send their output. A typical complete session will use this sequence of Metamath commands:

```
read set.mm
open tex example.tex
show statement a1i /tex
show proof a1i /all/lemmon/renumber/tex
show statement uneq2 /tex
show proof uneq2 /all/lemmon/renumber/tex
close tex
```

See Section 4.4.1 for information on comment markup and Appendix A for information on how math symbol translation is specified.

To format and print the LaTeX source, you will need the LaTeX program, which is standard on most Linux installations and available for Windows. On Linux, in order to create a PDF file, you will typically type at the shell prompt

```
$ pdflatex example.tex
```

5.7.1 `open tex` Command

Syntax: `open tex` *file-name* [`/no_header`]

This command opens a file for writing LaTeX source and writes a LaTeX header to the file. LaTeX source can be written with the `show proof`, `show new_proof`, and `show statement` commands using the `/tex` qualifier.

The mapping to LaTeX symbols is defined in a special comment containing a `$t` token, described in Appendix A.

There is an optional command qualifier:

`/no_header` - This qualifier prevents a standard LaTeX header and trailer from being included with the output LaTeX code.

5.7.2 `close tex` Command

Syntax: `close tex`

This command writes a trailer to any LaTeX file that was opened with `open tex` (unless `/no_header` was used with `open tex`) and closes the LaTeX file.

5.8 Creating HTML Output

You can generate HTML web pages given the information in a database. The database must already include the necessary typesetting information (see section 4.4.2 for how to provide this information). The ability to produce HTML web pages was added in Metamath version 0.07.30.

To create an HTML output file(s) for `$a` or `$p` statement(s), use

`show statement` *label-match* `/html`

The output file will be named *label-match*`.html` for each match. When *label-match* has wildcard (`*`) characters, all statements with matching labels will have HTML files produced for them. Also, when *label-match* has a wildcard (`*`) character, two additional files, `mmdefinitions.html` and `mmascii.html` will be produced. To produce *only* these two additional files, you can use `?*`, which will not match any statement label, in place of *label-match*.

There are three other qualifiers for `show statement` that also generate HTML code. These are `/alt_html`, `/brief_html`, and `/brief_alt_html`, and are described in the next section.

The command

`show statement` *label-match* `/alt_html`

does the same as `show statement` *label-match* `/html`, except that the HTML code for the symbols is taken from `althtmldef` statements instead of `htmldef` statements in the `$t` comment.

The command

```
show statement * /brief_html
```

invokes a special mode that just produces definition and theorem lists accompanied by their symbol strings, in a format suitable for copying and pasting into another web page (such as the tutorial pages on the Metamath web site).

Finally, the command

```
show statement * /brief_alt_html
```

does the same as `show statement * / brief_html` for the alternate HTML symbol representation.

A statement's comment can include a special notation that provides a certain amount of control over the HTML version of the comment. See Section 4.4.1 (p. 140) for the comment markup features.

The `write theorem_list` and `write bibliography` commands, which are described below, provide as a side effect complete error checking for all of the features described in this section.

5.8.1 `write theorem_list` Command

Syntax: `write theorem_list` [`/theorems_per_page` *number*]

This command writes a list of all of the `$a` and `$p` statements in the database into a web page file called `mmtheorems.html`. When additional files are needed, they are called `mmtheorems2.html`, `mmtheorems3.html`, etc.

Optional command qualifier:

`/theorems_per_page` *number* - This qualifier specifies the number of statements to write per web page. The default is 100.

Note: In version 0.177 of Metamath, the "Nearby theorems" links on the individual web pages presuppose 100 theorems per page when linking to the theorem list pages. Therefore the `/theorems_per_page` qualifier, if it specifies a number other than 100, will cause the individual web pages to be out of sync and should not be used to generate the main theorem list for the web site. This may be fixed in a future version.

5.8.2 `write bibliography` Command

Syntax: `write bibliography` *filename*

This command reads an existing HTML bibliographic cross-reference file, normally called `mmbiblio.html`, and updates it per the bibliographic links in the database comments. The file is updated between the HTML comment lines `<!-- #START# -->` and `<!-- #END# -->`. The original input file is renamed to *filename*`~1`.

A bibliographic reference is indicated with the reference name in brackets, such as `Theorem 3.1 of [Monk] p. 22`. See Section 5.8 (p. 177) for syntax details.

5.8.3 `write recent_additions` Command

Syntax: `write recent_additions` *filename* [`/limit` *number*]

This command reads an existing "Recent Additions" HTML file, normally called `mmrecent.html`, and updates it with the descriptions of the most recently added theorems to the database. The file is updated between the

HTML comment lines `<!-- #START# -->` and `<!-- #END# -->`. The original input file is renamed to *filename*`~1`.

Optional command qualifier:

`/limit` *number* - This qualifier specifies the number of most recent theorems to write to the output file. The default is 100.

5.9 Text File Utilities

5.9.1 `tools` Command

Syntax: `tools`

This command invokes an easy-to-use, general purpose utility for manipulating the contents of ASCII text files. Upon typing `tools`, the command-line prompt will change to `TOOLS>` until you type `exit`. The `tools` commands can be used to perform simple, global edits on an input/output file, such as making a character string substitution on each line, adding a string to each line, and so on. A typical use of this utility is to build a `submit` input file to perform a common operation on a list of statements obtained from `show label` or `show usage`.

The actions of most of the `tools` commands can also be performed with equivalent (and more powerful) Unix shell commands, and some users may find those more efficient. But for Windows users or users not comfortable with Unix, `tools` provides an easy-to-learn alternative that is adequate for most of the script-building tasks needed to use the Metamath program effectively.

5.9.2 `help` Command (in `tools`)

Syntax: `help`

The `help` command lists the commands available in the `tools` utility, along with a brief description. Each command, in turn, has its own help, such as `help add`. As with Metamath's `MM>` prompt, a complete command can be entered at once, or just the command word can be typed, causing the program to prompt for each argument.

Line-by-line editing commands:

`add` - Add a specified string to each line in a file.
`clean` - Trim spaces and tabs on each line in a file; convert characters.
`delete` - Delete a section of each line in a file.
`insert` - Insert a string at a specified column in each line of a file.
`substitute` - Make a simple substitution on each line of the file.
`tag` - Like `add`, but restricted to a range of lines.
`swap` - Swap the two halves of each line in a file.

Other file-processing commands:

`break` - Break up (tokenize) a file into a list of tokens (one per line).

`build` - Build a file with multiple tokens per line from a list.
`count` - Count the occurrences in a file of a specified string.
`number` - Create a list of numbers.
`parallel` - Put two files in parallel.
`reverse` - Reverse the order of the lines in a file.
`right` - Right-justify lines in a file (useful before sorting numbers).
`sort` - Sort the lines in a file with key starting at specified string.
`match` - Extract lines containing (or not) a specified string.
`unduplicate` - Eliminate duplicate occurrences of lines in a file.
`duplicate` - Extract first occurrence of any line occurring more than once in a file, discarding lines occurring exactly once.
`unique` - Extract lines occurring exactly once in a file.
`type` (10 lines) - Display the first few lines in a file. Similar to Unix `head`.
`copy` - Similar to Unix `cat` but safe (same input and output file allowed).
`submit` - Run a script containing `tools` commands.

Note: `unduplicate`, `duplicate`, and `unique` also sort the lines as a side effect.

5.9.3 Using `tools` to Build Metamath `submit` Scripts

The `break` command is typically used to break up a series of statement labels, such as the output of Metamath's `show usage`, into one label per line. The other `tools` commands can then be used to add strings before and after each statement label to specify commands to be performed on the statement. The `parallel` command is useful when a statement label must be mentioned more than once on a line.

Very often a `submit` script for Metamath will require multiple command lines for each statement being processed. For example, you may want to enter the Proof Assistant, `minimize_with` your latest theorem, `save` the new proof, and `exit` the Proof Assistant. To accomplish this, you can build a file with these four commands for each statement on a single line, separating each command with a designated character such as `@`. Then at the end you can `substitute` each `@` with `\n` to break up the lines into individual command lines (see `help substitute`).

5.9.4 Example of a `tools` Session

To give you a quick feel for the `tools` utility, we show a simple session where we create a file `n.txt` with 3 lines, add strings before and after each line, and display the lines on the screen. You can experiment with the various commands to gain experience with the `tools` utility.

```
MM> tools
Entering the Text Tools utilities.
Type HELP for help, EXIT to exit.
```

```
TOOLS> number
Output file <n.tmp>? n.txt
First number <1>?
Last number <10>? 3
Increment <1>?
TOOLS> add
Input/output file? n.txt
String to add to beginning of each line <>? This is line
String to add to end of each line <>? .
The file n.txt has 3 lines; 3 were changed.
First change is on line 1:
This is line 1.
TOOLS> type n.txt
This is line 1.
This is line 2.
This is line 3.
TOOLS> exit
Exiting the Text Tools.
Type EXIT again to exit Metamath.
MM>
```

Appendix A

Sample Representations

This Appendix provides a sample of ASCII representations, their corresponding traditional mathematical symbols, and a discussion of their meanings in the `set.mm` database. These are provided in order of appearance. This is only a partial list, and new definitions are routinely added. A complete list is available at `http://metamath.org`.

These ASCII representations, along with information on how to display them, are defined in the `set.mm` database file inside a special comment called a `$t` *comment* or *typesetting comment.* A typesetting comment is indicated by the appearance of the two-character string `$t` at the beginning of the comment. For more information, see Section 4.4.2, p. 144.

In the following table the "ASCII" column shows the ASCII representation, "Symbol" shows the mathematical symbolic display that corresponds to that ASCII representation, "Labels" shows the key label(s) that define the representation, and "Description" provides a description about the symbol. As usual, "iff" is short for "if and only if." In most cases the "ASCII" column only shows the key token, but it will sometimes show a sequence of tokens if that is necessary for clarity.

ASCII	Symbol	Labels	Description
`|-`	$\vdash$		"It is provable that..."
`ph`	φ	`wph`	The wff (boolean) variable phi, conventionally the first wff variable.
`ps`	ψ	`wps`	The wff (boolean) variable psi, conventionally the second wff variable.
`ch`	χ	`wch`	The wff (boolean) variable chi, conventionally the third wff variable.
`-.`	$\neg$	`wn`	Logical not. E.g., if φ is true, then $\neg\varphi$ is false.

ASCII	Symbol	Labels	Description
`->`	$\rightarrow$	`wi`	Implies, also known as material implication. In classical logic the expression $\varphi \rightarrow \psi$ is true if either φ is false or ψ is true (or both), that is, $\varphi \rightarrow \psi$ has the same meaning as $\neg\varphi \vee \psi$ (as proven in theorem `imor`).
`<->`	$\leftrightarrow$	`df-bi`	Biconditional (aka is-equals for boolean values). $\varphi \leftrightarrow \psi$ is true iff φ and ψ have the same value.
`\/`	$\vee$	`df-or`, `df-3or`	Disjunction (logical "or"). $\varphi \vee \psi$ is true iff φ, ψ, or both are true.
`/\`	$\wedge$	`df-an`, `df-3an`	Conjunction (logical "and"). $\varphi \wedge \psi$ is true iff both φ and ψ are true.
`A.`	$\forall$	`wal`	For all; the wff $\forall x\varphi$ is true iff φ is true for all values of x.
`E.`	$\exists$	`df-ex`	There exists; the wff $\exists x\varphi$ is true iff there is at least one x where φ is true.
`[ y / x ]`	$[y/x]$	`df-sb`	The wff $[y/x]\varphi$ produces the result when y is properly substituted for x in φ (y replaces x). For example, $[x/y]z \in y$ is the same as $z \in x$.
`E!`	$\exists!$	`df-eu`	There exists exactly one; $\exists!\, x\varphi$ is true iff there is at least one x where φ is true.
`{ y | phi }`	$\{y\|\varphi\}$	`df-clab`	The class of all sets where φ is true.
`=`	$=$	`df-cleq`	Class equality; $A = B$ iff A equals B.
`e.`	$\in$	`df-clel`	Class membership; $A \in B$ if A is a member of B.
`_V`	V	`df-v`	Class of all sets (not itself a set).
`C_`	$\subseteq$	`df-ss`	Subclass (subset); $A \subseteq B$ is true iff A is a subclass of B.
`u.`	$\cup$	`df-un`	$A \cup B$ is the union of classes A and B.
`i^i`	$\cap$	`df-in`	$A \cap B$ is the intersection of classes A and B.
`\`	$\setminus$	`df-dif`	$A \setminus B$ (set difference) is the class of all sets in A except for those in B.

ASCII	Symbol	Labels	Description
`(/)`	$\varnothing$	`df-nul`	$\varnothing$ is the empty set (aka null set).
`~P`	$\mathcal{P}$	`df-pw`	Power class.
`<. A , B >.`	$\langle A, B \rangle$	`df-op`	The ordered pair $\langle A, B \rangle$.
`( F ` ` A )`	$(F`A)$	`df-fv`	The value of function F when applied to A.
`_i`	i	`df-i`	The square root of negative one.
`x.`	$\cdot$	`df-mul`	Complex number multiplication; $2 \cdot 3 = 6$.
`CC`	$\mathbb{C}$	`df-c`	The set of complex numbers.
`RR`	$\mathbb{R}$	`df-r`	The set of real numbers.

Appendix B

Compressed Proofs

The proofs in the `set.mm` set theory database are stored in compressed format for efficiency. Normally you needn't concern yourself with the compressed format, since you can display it with the usual proof display tools in the Metamath program (`show proof...`) or convert it to the normal RPN proof format described in Section 4.3 (with `save proof` *label* `/normal`). However for sake of completeness we describe the format here and show how it maps to the normal RPN proof format.

A compressed proof, located between `$=` and `$.` keywords, consists of a left parenthesis, a sequence of statement labels, a right parenthesis, and a sequence of upper-case letters `A` through `Z` (with optional white space between them). White space must surround the parentheses and the labels. The left parenthesis tells Metamath that a compressed proof follows. (A normal RPN proof consists of just a sequence of labels, and a parenthesis is not a legal character in a label.)

The sequence of upper-case letters corresponds to a sequence of integers with the following mapping. Each integer corresponds to a proof step as described later.

$$\begin{aligned}
\texttt{A} &= 1 \\
\texttt{B} &= 2 \\
&\ldots \\
\texttt{T} &= 20 \\
\texttt{UA} &= 21 \\
\texttt{UB} &= 22 \\
&\ldots \\
\texttt{UT} &= 40 \\
\texttt{VA} &= 41 \\
\texttt{VB} &= 42 \\
&\ldots \\
\texttt{YT} &= 120
\end{aligned}$$

$$\texttt{UUA} = 121$$
$$\ldots$$
$$\texttt{YYT} = 620$$
$$\texttt{UUUA} = 621$$
$$\text{etc.}$$

In other words, `A` through `T` represent the least-significant digit in base 20, and `U` through `Y` represent zero or more most-significant digits in base 5, where the digits start counting at 1 instead of the usual 0. With this scheme, we don't need white space between these "numbers."

(In the design of the compressed proof format, only upper-case letters, as opposed to say all non-whitespace printable ASCII characters other than `$`, were chosen so as not to collide with most text editor searches, at the expense of a typical 20% compression loss. The base 5/base 20 grouping, as opposed to say base 6/base 19, was chosen by experimentally determining the grouping that resulted in best typical compression.)

The letter `Z` identifies (tags) a proof step that is identical to one that occurs later on in the proof; it helps shorten the proof by not requiring that identical proof steps be proved over and over again (which happens often when building wff's). The `Z` is placed immediately after the least-significant digit (letters `A` through `T`) that ends the integer corresponding to the step to later be referenced.

The integers that the upper-case letters correspond to are mapped to labels as follows. If the statement being proved has m mandatory hypotheses, integers 1 through m correspond to the labels of these hypotheses in the order shown by the `show statement ... / full` command, i.e., the RPN order of the mandatory hypotheses. Integers $m+1$ through $m+n$ correspond to the labels enclosed in the parentheses of the compressed proof, in the order that they appear, where n is the number of those labels. Integers $m+n+1$ on up don't directly correspond to statement labels but point to proof steps identified with the letter `Z`, so that these proof steps can be referenced later in the proof. Integer $m+n+1$ corresponds to the first step tagged with a `Z`, $m+n+2$ to the second step tagged with a `Z`, etc. When the compressed proof is converted to a normal proof, the entire subproof of a step tagged with `Z` replaces the reference to that step.

For efficiency, Metamath works with compressed proofs directly, without converting them internally to normal proofs. In addition to the usual error-checking, an error message is given if (1) a label in the label list in parentheses does not refer to a previous `$p` or `$a` statement or a non-mandatory hypothesis of the statement being proved and (2) a proof step tagged with `Z` is referenced before the step tagged with the `Z`.

Just as in a normal proof under development (Section 4.4.6), any step or subproof that is not yet known may be represented with a single `?`. White space does not have to appear between the `?` and the upper-case letters (or other `?`'s) representing the remainder of the proof.

Appendix C

Metamath's Formal System

C.1 Introduction

Perfection is when there is no longer anything more to take away.

Antoine de Saint-Exupery[1]

This appendix describes the theory behind the Metamath language in an abstract way intended for mathematicians. Specifically, we construct two set-theoretical objects: a "formal system" (roughly, a set of syntax rules, axioms, and logical rules) and its "universe" (roughly, the set of theorems derivable in the formal system). The Metamath computer language provides us with a way to describe specific formal systems and, with the aid of a proof provided by the user, to verify that given theorems belong to their universes.

To understand this appendix, you need a basic knowledge of informal set theory. It should be sufficient to understand, for example, Ch. 1 of Munkres' *Topology* [48] or the introductory set theory chapter in many textbooks that introduce abstract mathematics. (Note that there are minor notational differences among authors; e.g. Munkres uses $\subset$ instead of our $\subseteq$ for "subset." We use "included in" to mean "a subset of," and "belongs to" or "is contained in" to mean "is an element of.") What we call a "formal" description here, unlike earlier, is actually an informal description in the ordinary language of mathematicians. However we provide sufficient detail so that a mathematician could easily formalize it, even in the language of Metamath itself if desired. To understand the logic examples at the end of this appendix, familiarity with an introductory book on mathematical logic would be helpful.

[1] [9, p. 3-25].

C.2 The Formal Description

C.2.1 Preliminaries[2]

By ω we denote the set of all natural numbers (non-negative integers). Each natural number n is identified with the set of all smaller numbers: $n = \{m | m < n\}$. The formula $m < n$ is thus equivalent to the condition: $m \in n$ and $m, n \in \omega$. In particular, 0 is the number zero and at the same time the empty set $\varnothing$, $1 = \{0\}$, $2 = \{0, 1\}$, etc. ${}^{B}A$ denotes the set of all functions on B to A (i.e. with domain B and range included in A). The members of ${}^{\omega}A$ are what are called *simple infinite sequences*, with all *terms* in A. In case $n \in \omega$, the members of ${}^{n}A$ are referred to as *finite n-termed sequences*, again with terms in A. The consecutive terms (function values) of a finite or infinite sequence f are denoted by $f_0, f_1, \ldots, f_n, \ldots$. Every finite sequence $f \in \bigcup_{n \in \omega} {}^{n}A$ uniquely determines the number n such that $f \in {}^{n}A$; n is called the *length* of f and is denoted by $|f|$. $\langle a \rangle$ is the sequence f with $|f| = 1$ and $f_0 = a$; $\langle a, b \rangle$ is the sequence f with $|f| = 2$, $f_0 = a$, $f_1 = b$; etc. Given two finite sequences f and g, we denote by $f \frown g$ their *concatenation*, i.e., the finite sequence h determined by the conditions:

$$\begin{aligned} |h| &= |f| + |g|; \\ h_n &= f_n \qquad \text{for } n < |f|; \\ h_{|f|+n} &= g_n \qquad \text{for } n < |g|. \end{aligned}$$

C.2.2 Constants, Variables, and Expressions

A formal system has a set of *symbols* denoted by *SM*. A precise set-theoretical definition of this set is unimportant; a symbol could be considered a primitive or atomic element if we wish. We assume this set is divided into two disjoint subsets: a set *CN* of *constants* and a set *VR* of *variables*. *CN* and *VR* are each assumed to consist of countably many symbols which may be arranged in finite or simple infinite sequences $c_0, c_1, \ldots$ and $v_0, v_1, \ldots$ respectively, without repeating terms. We will represent arbitrary symbols by metavariables α, β, etc.

> *Comment.* The variables $v_0, v_1, \ldots$ of our formal system correspond to what are usually considered "metavariables" in descriptions of specific formal systems in the literature. Typically, when describing a specific formal system a book will postulate a set of primitive objects called variables, then proceed to describe their properties using metavariables that range over them, never mentioning again the actual variables themselves. Our formal system does not mention these primitive variable objects at all but deals directly with metavariables, as its primitive objects, from the start. This is a subtle but key distinction you should keep in mind, and it makes our definition of "formal system" somewhat different from that typically found in the literature. (So, the α, β, etc. above are actually "metametavariables" when used to represent $v_0, v_1, \ldots$.)

[2]This section is taken mostly verbatim from Tarski [69, p. 63].

Finite sequences all terms of which are symbols are called *expressions*. EX is the set of all expressions; thus

$$EX = \bigcup_{n \in \omega} {}^{n}SM.$$

A *constant-prefixed expression* is an expression of non-zero length whose first term is a constant. We denote the set of all constant-prefixed expressions by $EX_C = \{e \in EX | (|e| > 0 \wedge e_0 \in CN)\}$.

A *constant-variable pair* is an expression of length 2 whose first term is a constant and whose second term is a variable. We denote the set of all constant-variable pairs by $EX_2 = \{e \in EX_C | (|e| = 2 \wedge e_1 \in VR)\}$.

> *Relationship to Metamath.* In general, the set SM corresponds to the set of declared math symbols in a Metamath database, the set CN to those declared with `$c` statements, and the set VR to those declared with `$v` statements. Of course a Metamath database can only have a finite number of math symbols, whereas formal systems in general can have an infinite number, although the number of Metamath math symbols available is in principle unlimited.
>
> The set EX_C corresponds to the set of permissible expressions for `$e`, `$a`, and `$p` statements. The set EX_2 corresponds to the set of permissible expressions for `$f` statements.

We denote by $\mathcal{V}(e)$ the set of all variables in an expression $e \in EX$, i.e. the set of all $\alpha \in VR$ such that $\alpha = e_n$ for some $n < |e|$. We also denote (with abuse of notation) by $\mathcal{V}(E)$ the set of all variables in a collection of expressions $E \subseteq EX$, i.e. $\bigcup_{e \in E} \mathcal{V}(e)$.

C.2.3 Substitution

Given a function F from VR to EX, we denote by σ_F or just σ the function from EX to EX defined recursively for nonempty sequences by

$$\begin{aligned} \sigma(<\alpha>) &= F(\alpha) && \text{for } \alpha \in VR; \\ \sigma(<\alpha>) &= <\alpha> && \text{for } \alpha \notin VR; \\ \sigma(g \frown h) &= \sigma(g) \frown \sigma(h) && \text{for } g, h \in EX. \end{aligned}$$

We also define $\sigma(\varnothing) = \varnothing$. We call σ a *simultaneous substitution* (or just *substitution*) with *substitution map* F.

We also denote (with abuse of notation) by $\sigma(E)$ a substitution on a collection of expressions $E \subseteq EX$, i.e. the set $\{\sigma(e) | e \in E\}$. The collection $\sigma(E)$ may of course contain fewer expressions than E because duplicate expressions could result from the substitution.

C.2.4 Statements

We denote by DV the set of all unordered pairs $\{\alpha, \beta\} \subseteq VR$ such that $\alpha \neq \beta$. DV stands for "distinct variables."

A *pre-statement* is a quadruple $\langle D, T, H, A\rangle$ such that $D \subseteq DV$, $T \subseteq EX_2$, $H \subseteq EX_C$ and H is finite, $A \in EX_C$, $\mathcal{V}(H \cup \{A\}) \subseteq \mathcal{V}(T)$, and $\forall e, f \in T\ \mathcal{V}(e) \neq \mathcal{V}(f)$ (or equivalently, $e_1 \neq f_1$) whenever $e \neq f$. The terms of the quadruple are called *distinct-variable restrictions*, *variable-type hypotheses*, *logical hypotheses*, and the *assertion* respectively. We denote by T_M (*mandatory variable-type hypotheses*) the subset of T such that $\mathcal{V}(T_M) = \mathcal{V}(H \cup \{A\})$. We denote by $D_M = \{\{\alpha, \beta\} \in D | \{\alpha, \beta\} \subseteq \mathcal{V}(T_M)\}$ the *mandatory distinct-variable restrictions* of the pre-statement. The set of *mandatory hypotheses* is $T_M \cup H$. We call the quadruple $\langle D_M, T_M, H, A\rangle$ the *reduct* of the pre-statement $\langle D, T, H, A\rangle$.

A *statement* is the reduct of some pre-statement. A statement is therefore a special kind of pre-statement; in particular, a statement is the reduct of itself.

Comment. T is a set of expressions, each of length 2, that associate a set of constants ("variable types") with a set of variables. The condition $\mathcal{V}(H \cup \{A\}) \subseteq \mathcal{V}(T)$ means that each variable occurring in a statement's logical hypotheses or assertion must have an associated variable-type hypothesis or "type declaration," in analogy to a computer programming language, where a variable must be declared to be say, a string or an integer. The requirement that $\forall e, f \in T\, e_1 \neq f_1$ for $e \neq f$ means that each variable must be associated with a unique constant designating its variable type; e.g., a variable might be a "wff" or a "set" but not both.

Distinct-variable restrictions are used to specify what variable substitutions are permissible to make for the statement to remain valid. For example, in the theorem scheme of set theory $\neg\forall x\, x = y$ we may not substitute the same variable for both x and y. On the other hand, the theorem scheme $x = y \rightarrow y = x$ does not require that x and y be distinct, so we do not require a distinct-variable restriction, although having one would cause no harm other than making the scheme less general.

A mandatory variable-type hypothesis is one whose variable exists in a logical hypothesis or the assertion. A provable pre-statement (defined below) may require non-mandatory variable-type hypotheses that effectively introduce "dummy" variables for use in its proof. Any number of dummy variables might be required by a specific proof; indeed, it has been shown by H. Andréka [50] that there is no finite upper bound to the number of dummy variables needed to prove an arbitrary theorem in first-order logic (with equality) having a fixed number $n > 2$ of individual variables. (See also the Comment on p. 125.) For this reason we do not set a finite size bound on the collections D and T, although in an actual application (Metamath database) these will of course be finite, increased to whatever size is necessary as more proofs are added.

Relationship to Metamath. A pre-statement of a formal system corresponds to an extended frame in a Metamath database (Section 4.2.7). The collections D, T, and H correspond respectively to the `$d`, `$f`, and `$e` statement collections in an extended frame. The expression A corresponds to the `$a` (or `$p`) statement in an extended frame.

A statement of a formal system corresponds to a frame in a Metamath database.

C.2.5 Formal Systems

A *formal system* is a triple $\langle CN, VR, \Gamma \rangle$ where Γ is a set of statements. The members of Γ are called *axiomatic statements*. Sometimes we will refer to a formal system by just Γ when CN and VR are understood.

Given a formal system Γ, the *closure*[3] of a pre-statement $\langle D, T, H, A \rangle$ is the smallest set C of expressions such that:

1. $T \cup H \subseteq C$; and
2. If for some axiomatic statement $\langle D'_M, T'_M, H', A' \rangle \in \Gamma$ and for some substitution σ we have

   a. $\sigma(T'_M \cup H') \subseteq C$; and

   b. for all $\{\alpha, \beta\} \in D'_M$, for all $\gamma \in \mathcal{V}(\sigma(\langle \alpha \rangle))$, and for all $\delta \in \mathcal{V}(\sigma(\langle \beta \rangle))$, we have $\{\gamma, \delta\} \in D$;

   then $\sigma(A') \in C$.

A pre-statement $\langle D, T, H, A \rangle$ is *provable* if $A \in C$ i.e. if its assertion belongs to its closure. A statement is *provable* if it is the reduct of a provable pre-statement. The *universe* of a formal system is the collection of all of its provable statements. Note that the set of axiomatic statements Γ in a formal system is a subset of its universe.

Comment. The first condition in the definition of closure simply says that the hypotheses of the pre-statement are in its closure.

Condition 2(a) says that a substitution exists that makes the mandatory hypotheses of an axiomatic statement exactly match some members of the closure. This is what we explicitly demonstrate in a Metamath language proof.

Condition 2(b) describes how distinct-variable restrictions in the axiomatic statement must be met. It means that after a substitution for two variables that must be distinct, the resulting two expressions must either contain no variables, or if they do, they may not have variables in common, and each pair of any variables they do have, with one variable from each expression, must be specified as distinct in the original statement.

Relationship to Metamath. Axiomatic statements and provable statements in a formal system correspond to the frames for `$a` and `$p` statements respectively in a Metamath database. The set of axiomatic statements is a subset of the set of provable statements in a formal system, although in a Metamath database a `$a` statement is distinguished by not having a proof. A Metamath language proof for a `$p` statement tells the computer how to explicitly construct a series of members of the closure ultimately leading to a demonstration that the assertion being proved is in the closure. The actual closure typically contains an infinite number of expressions. A formal system itself does not have an explicit object called a "proof" but rather the existence of a proof is implied indirectly by membership of an assertion in a provable statement's closure. We do this to make the formal system easier to describe in the language of set theory.

[3]This definition of closure incorporates a simplification due to Josh Purinton..

We also note that once established as provable, a statement may be considered to acquire the same status as an axiomatic statement, because if the set of axiomatic statements is extended with a provable statement, the universe of the formal system remains unchanged (provided that *VR* is infinite). In practice, this means we can build a hierarchy of provable statements to more efficiently establish additional provable statements. This is what we do in Metamath when we allow proofs to reference previous `$p` statements as well as previous `$a` statements.

C.3 Examples of Formal Systems

Relationship to Metamath. The examples in this section, except Example 2, are for the most part exact equivalents of the development in the set theory database `set.mm`. You may want to compare Examples 1, 3, and 5 to Section 3.3, Example 4 to Sections 3.4.1 and 3.4.2, and Example 6 to Section 3.4.3.

C.3.1 Example 1—Propositional Calculus

Classical propositional calculus can be described by the following formal system. We assume the set of variables is infinite. Rather than denoting the constants and variables by $c_0, c_1, \ldots$ and $v_0, v_1, \ldots$, for readability we will instead use more conventional symbols, with the understanding of course that they denote distinct primitive objects. Also for readability we may omit commas between successive terms of a sequence; thus $\langle \text{wff } \varphi \rangle$ denotes $\langle \text{wff}, \varphi \rangle$.

Let

$$CN = \{\text{wff}, \vdash, \rightarrow, \neg, (,)\}$$

$$VR = \{\varphi, \psi, \chi, \ldots\}$$

$T = \{\langle \text{wff } \varphi \rangle, \langle \text{wff } \psi \rangle, \langle \text{wff } \chi \rangle, \ldots\}$, i.e. those expressions of length 2 whose first member is wff and whose second member belongs to *VR*.[4] Then Γ consists of the axiomatic statements that are the reducts of the following pre-statements:

$$\langle \varnothing, T, \varnothing, \langle \text{wff } (\varphi \rightarrow \psi) \rangle \rangle$$
$$\langle \varnothing, T, \varnothing, \langle \text{wff } \neg\varphi \rangle \rangle$$
$$\langle \varnothing, T, \varnothing, \langle \vdash (\varphi \rightarrow (\psi \rightarrow \varphi)) \rangle \rangle$$
$$\langle \varnothing, T, \varnothing, \langle \vdash ((\varphi \rightarrow (\psi \rightarrow \chi)) \rightarrow ((\varphi \rightarrow \psi) \rightarrow (\varphi \rightarrow \chi))) \rangle \rangle$$

[4] For convenience we let T be an infinite set; the definition of a statement permits this in principle. Since a Metamath source file has a finite size, in practice we must of course use appropriate finite subsets of this T, specifically ones containing at least the mandatory variable-type hypotheses. Similarly, in the source file we introduce new variables as required, with the understanding that a potentially infinite number of them are available.

$$\langle \varnothing, T, \varnothing, \langle \vdash ((\neg\varphi \to \neg\psi) \to (\psi \to \varphi))\rangle\rangle$$
$$\langle \varnothing, T, \{\langle \vdash (\varphi \to \psi)\rangle, \langle \vdash \varphi\rangle\}, \langle \vdash \psi\rangle\rangle$$

(For example, the reduct of $\langle \varnothing, T, \varnothing, \langle \text{wff } (\varphi \to \psi)\rangle\rangle$ is

$$\langle \varnothing, \{\langle \text{wff } \varphi\rangle, \langle \text{wff } \psi\rangle\}, \varnothing, \langle \text{wff } (\varphi \to \psi)\rangle\rangle,$$

which is the first axiomatic statement.)

We call the members of *VR wff variables* or (in the context of first-order logic which we will describe shortly) *wff metavariables*. Note that the symbols ϕ, ψ, etc. denote actual specific members of *VR*; they are not metavariables of our expository language (which we denote with α, β, etc.) but are instead (meta)constant symbols (members of *SM*) from the point of view of our expository language. The equivalent system of propositional calculus described in [69] also uses the symbols ϕ, ψ, etc. to denote wff metavariables, but in [69] unlike here those are metavariables of the expository language and not primitive symbols of the formal system.

The first two statements define wffs: if φ and ψ are wffs, so is $(\varphi \to \psi)$; if φ is a wff, so is $\neg\varphi$. The next three are the axioms of propositional calculus: if φ and ψ are wffs, then $\vdash (\varphi \to (\psi \to \varphi))$ is an (axiomatic) theorem; etc. The last is the rule of modus ponens: if φ and ψ are wffs, and $\vdash (\varphi \to \psi)$ and $\vdash \varphi$ are theorems, then $\vdash \psi$ is a theorem.

The correspondence to ordinary propositional calculus is as follows. We consider only provable statements of the form $\langle \varnothing, T, \varnothing, A\rangle$ with T defined as above. The first term of the assertion A of any such statement is either "wff" or "$\vdash$". A statement for which the first term is "wff" is a *wff* of propositional calculus, and one where the first term is "$\vdash$" is a *theorem (scheme)* of propositional calculus.

The universe of this formal system also contains many other provable statements. Those with distinct-variable restrictions are irrelevant because propositional calculus has no constraints on substitutions. Those that have logical hypotheses we call *inferences* when the logical hypotheses are of the form $\langle \vdash \rangle \frown w$ where w is a wff (with the leading constant term "wff" removed). Inferences (other than the modus ponens rule) are not a proper part of propositional calculus but are convenient to use when building a hierarchy of provable statements. A provable statement with a nonsense hypothesis such as $\langle \to, \vdash, \neg\rangle$, and this same expression as its assertion, we consider irrelevant; no use can be made of it in proving theorems, since there is no way to eliminate the nonsense hypothesis.

> *Comment.* Our use of parentheses in the definition of a wff illustrates how axiomatic statements should be carefully stated in a way that ties in unambiguously with the substitutions allowed by the formal system. There are many ways we could have defined wffs—for example, Polish prefix notation would have allowed us to omit parentheses entirely, at the expense of readability—but we must define them in a way that is unambiguous. For example, if we had omitted parentheses from the definition of $(\varphi \to \psi)$, the wff $\neg\varphi \to \psi$ could be interpreted as either $\neg(\varphi \to \psi)$ or $(\neg\varphi \to \psi)$ and

would have allowed us to prove nonsense. Note that there is no concept of operator binding precedence built into our formal system.

C.3.2 Example 2—Predicate Calculus with Equality

Here we extend Example 1 to include predicate calculus with equality, illustrating the use of distinct-variable restrictions. This system is the same as Tarski's system $\mathfrak{S}_2$ in [69] (except that the axioms of propositional calculus are different but equivalent, and a redundant axiom is omitted). We extend *CN* with the constants $\{\text{var}, \forall, =\}$. We extend *VR* with an infinite set of *individual metavariables* $\{x, y, z, \ldots\}$ and denote this subset *Vr*.

We also join to *CN* a possibly infinite set *Pr* of *predicates* $\{R, S, \ldots\}$. We associate with *Pr* a function rnk from *Pr* to ω, and for $\alpha \in Pr$ we call $\text{rnk}(\alpha)$ the *rank* of the predicate α, which is simply the number of "arguments" that the predicate has. (Most applications of predicate calculus will have a finite number of predicates; for example, set theory has the single two-argument or binary predicate $\in$, which is usually written with its arguments surrounding the predicate symbol rather than with the prefix notation we will use for the general case.) As a device to facilitate our discussion, we will let *Vs* be any fixed one-to-one function from ω to *Vr*; thus *Vs* is any simple infinite sequence of individual metavariables with no repeating terms.

In this example we will not include the function symbols that are often part of formalizations of predicate calculus. Using metalogical arguments that are beyond the scope of our discussion, it can be shown that our formalization is equivalent when functions are introduced via appropriate definitions.

We extend the set T defined in Example 1 with the expressions $\{\langle \text{var } x\rangle, \langle \text{var } y\rangle, \langle \text{var } z\rangle, \ldots\}$. We extend the Γ above with the axiomatic statements that are the reducts of the following pre-statements:

$\langle \varnothing, T, \varnothing, \langle \text{wff } \forall x\, \varphi\rangle\rangle$

$\langle \varnothing, T, \varnothing, \langle \text{wff } x = y\rangle\rangle$

$\langle \varnothing, T, \{\langle \vdash \varphi\rangle\}, \langle \vdash \forall x\, \varphi\rangle\rangle$

$\langle \varnothing, T, \varnothing, \langle \vdash ((\forall x(\varphi \to \psi) \to (\forall x\, \varphi \to \forall x\, \psi)))\rangle\rangle$

$\langle \{\{x, \varphi\}\}, T, \varnothing, \langle \vdash (\varphi \to \forall x\, \varphi)\rangle\rangle$

$\langle \{\{x, y\}\}, T, \varnothing, \langle \vdash \neg\forall x \neg x = y\rangle\rangle$

$\langle \varnothing, T, \varnothing, \langle \vdash (x = z \to (x = y \to z = y))\rangle\rangle$

$\langle \varnothing, T, \varnothing, \langle \vdash (y = z \to (x = y \to x = z))\rangle\rangle$

These are the axioms not involving predicate symbols. The first two statements extend the definition of a wff. The third is the rule of generalization. The fifth states, in effect, "For a wff φ and variable x, $\vdash (\varphi \to \forall x\, \varphi)$, provided that x does not occur in φ." The sixth states "For variables x and y, $\vdash \neg\forall x \neg x = y$, provided that x and y are distinct." (This proviso is not

necessary but was included by Tarski to weaken the axiom and still show that the system is logically complete.)

Finally, for each predicate symbol $\alpha \in Pr$, we add to Γ an axiomatic statement, extending the definition of wff, that is the reduct of the following pre-statement:

$$\langle \varnothing, T, \varnothing, \langle \text{wff}, \alpha \rangle \frown Vs \restriction \text{rnk}(\alpha) \rangle$$

and for each $\alpha \in Pr$ and each $n < \text{rnk}(\alpha)$ we add to Γ an equality axiom that is the reduct of the following pre-statement:

$$\begin{aligned} \langle \varnothing, T, \varnothing, \langle \vdash, (, Vs_n, =, Vs_{\text{rnk}(\alpha)}, \rightarrow, (, \alpha \rangle \frown Vs \restriction \text{rnk}(\alpha) \\ \frown \langle \rightarrow, \alpha \rangle \frown Vs \restriction n \frown \langle Vs_{\text{rnk}(\alpha)} \rangle \\ \frown Vs \restriction (\text{rnk}(\alpha) \setminus (n+1)) \frown \langle),) \rangle \rangle \end{aligned}$$

where $\restriction$ denotes function domain restriction and $\setminus$ denotes set difference. Recall that a subscript on Vs denotes one of its terms. (In the above two axiom sets commas are placed between successive terms of sequences to prevent ambiguity, and if you examine them with care you will be able to distinguish those parentheses that denote constant symbols from those of our expository language that delimit function arguments. Although it might have been better to use boldface for our primitive symbols, unfortunately boldface was not available for all characters on the LaTeX system used to typeset this text.) These seemingly forbidding axioms can be understood by analogy to concatenation of substrings in a computer language. They are actually relatively simple for each specific case and will become clearer by looking at the special case of a binary predicate $\alpha = R$ where $\text{rnk}(R) = 2$. Letting Vs be the sequence $\langle x, y, z, \ldots \rangle$, the axioms we would add to Γ for this case would be the wff extension and two equality axioms that are the reducts of the pre-statements:

$$\begin{aligned} &\langle \varnothing, T, \varnothing, \langle \text{wff } Rxy \rangle \rangle \\ &\langle \varnothing, T, \varnothing, \langle \vdash (x = z \rightarrow (Rxy \rightarrow Rzy)) \rangle \rangle \\ &\langle \varnothing, T, \varnothing, \langle \vdash (y = z \rightarrow (Rxy \rightarrow Rxz)) \rangle \rangle \end{aligned}$$

Study these carefully to see how the general axioms above evaluate to them. In practice, typically only a few special cases such as this would be needed, and in any case the Metamath language will only permit us to describe a finite number of predicates, as opposed to the infinite number permitted by the formal system. (If an infinite number should be needed for some reason, we could not define the formal system directly in the Metamath language but could instead define it metalogically under set theory as we do in this appendix, and only the underlying set theory, with its single binary predicate, would be defined directly in the Metamath language.)

Comment. As we noted earlier, the specific variables denoted by the symbols $x, y, z, \ldots \in Vr \subseteq VR \subseteq SM$ in Example 2 are not the actual variables

of ordinary predicate calculus but should be thought of as metavariables ranging over them. For example, a distinct-variable restriction would be meaningless for actual variables of ordinary predicate calculus since two different actual variables are by definition distinct. And when we talk about an arbitrary representative $\alpha \in Vr$, α is a metavariable (in our expository language) that ranges over metavariables (which are primitives of our formal system) each of which ranges over the actual individual variables of predicate calculus (which are never mentioned in our formal system).

The constant called "var" above is called `setvar` in the `set.mm` database file, but it means the same thing. I felt that "var" is a more meaningful name in the context of predicate calculus, whose use is not limited to set theory. For consistency we stick with the name "var" throughout this Appendix, even after set theory is introduced.

C.3.3 Free Variables and Proper Substitution

Typical representations of mathematical axioms use concepts such as "free variable," "bound variable," and "proper substitution" as primitive notions. A free variable is a variable that is not a parameter of any container expression. A bound variable is the opposite of a free variable; it is a a variable that has been bound in a container expression. For example, in the expression $\forall x \varphi$ (for all x, φ is true), the variable x is bound within the for-all ($\forall$) expression. It is possible to change one variable to another, and that process is called "proper substitution." In most books, proper substitution has a somewhat complicated recursive definition with multiple cases based on the occurrences of free and bound variables. You may consult [21, ch. 3–4] (as well as many other texts) for more formal details about these terms.

Using these concepts as `primitives` creates complications for computer implementations.

In the system of Example 2, there are no primitive notions of free variable and proper substitution. Tarski [69] shows that this system is logically equivalent to the more typical textbook systems that do have these primitive notions, if we introduce these notions with appropriate definitions and metalogic. We could also define axioms for such systems directly, although the recursive definitions of free variable and proper substitution would be messy and awkward to work with. Instead, we mention two devices that can be used in practice to mimic these notions. (1) Instead of introducing special notation to express (as a logical hypothesis) "where x is not free in φ" we can use the logical hypothesis $\vdash (\varphi \rightarrow \forall x \, \varphi)$.[5] (2) It can be shown that the wff $((x = y \rightarrow \varphi) \wedge \exists x (x = y \wedge \varphi))$ (with the usual definitions of $\wedge$ and $\exists$; see Example 4 below) is logically equivalent to "the wff that results from proper substitution of y for x in φ." This works whether or not x and y are distinct.

[5]This is a slightly weaker requirement than "where x is not free in φ." If we let φ be $x = x$, we have the theorem $(x = x \rightarrow \forall x \, x = x)$ which satisfies the hypothesis, even though x is free in $x = x$. In a case like this we say that x is *effectively not free* in $x = x$, since $x = x$ is logically equivalent to $\forall x \, x = x$ in which x is bound.

C.3.4 Metalogical Completeness

In the system of Example 2, the following are provable pre-statements (and their reducts are provable statements):

$$\langle\{\{x, y\}\}, T, \varnothing, \langle\vdash \neg\forall x \neg x = y\rangle\rangle$$
$$\langle\varnothing, T, \varnothing, \langle\vdash \neg\forall x \neg x = x\rangle\rangle$$

whereas the following pre-statement is not to my knowledge provable (but in any case we will pretend it's not for sake of illustration):

$$\langle\varnothing, T, \varnothing, \langle\vdash \neg\forall x \neg x = y\rangle\rangle$$

In other words, we can prove "$\neg\forall x \neg x = y$ where x and y are distinct" and separately prove "$\neg\forall x \neg x = x$", but we can't prove the combined general case "$\neg\forall x \neg x = y$" that has no proviso. Now this does not compromise logical completeness, because the variables are really metavariables and the two provable cases together cover all possible cases. The third case can be considered a metatheorem whose direct proof, using the system of Example 2, lies outside the capability of the formal system.

Also, in the system of Example 2 the following pre-statement is not to my knowledge provable (again, a conjecture that we will pretend to be the case):

$$\langle\varnothing, T, \varnothing, \langle\vdash (\forall x\, \varphi \rightarrow \varphi)\rangle\rangle$$

Instead, we can only prove specific cases of φ involving individual metavariables, and by induction on formula length, prove as a metatheorem outside of our formal system the general statement above. The details of this proof are found in [28].

There does, however, exist a system of predicate calculus in which all such "simple metatheorems" as those above can be proved directly, and we present it in Example 3. A *simple metatheorem* is any statement of the formal system of Example 2 where all distinct variable restrictions consist of either two individual metavariables or an individual metavariable and a wff metavariable, and which is provable by combining cases outside the system as above. A system is *metalogically complete* if all of its simple metatheorems are (directly) provable statements. The precise definition of "simple metatheorem" and the proof of the "metalogical completeness" of Example 3 is found in Remark 9.6 and Theorem 9.7 of [41].

C.3.5 Example 3—Metalogically Complete Predicate Calculus with Equality

For simplicity we will assume there is one binary predicate R; this system suffices for set theory, where the R is of course the $\in$ predicate. We label

the axioms as they appear in [41]. This system is logically equivalent to that of Example 2 (when the latter is restricted to this single binary predicate) but is also metalogically complete.

Let

$$CN = \{\text{wff}, \text{var}, \vdash, \to, \neg, (,), \forall, =, R\}.$$

$$VR = \{\varphi, \psi, \chi, \ldots\} \cup \{x, y, z, \ldots\}.$$

$$T = \{\langle \text{wff } \varphi\rangle, \langle \text{wff } \psi\rangle, \langle \text{wff } \chi\rangle, \ldots\} \cup \{\langle \text{var } x\rangle, \langle \text{var } y\rangle, \langle \text{var } z\rangle, \ldots\}.$$

Then Γ consists of the reducts of the following pre-statements:

$$\langle \varnothing, T, \varnothing, \langle \text{wff } (\varphi \to \psi)\rangle\rangle$$
$$\langle \varnothing, T, \varnothing, \langle \text{wff } \neg\varphi\rangle\rangle$$
$$\langle \varnothing, T, \varnothing, \langle \text{wff } \forall x\, \varphi\rangle\rangle$$
$$\langle \varnothing, T, \varnothing, \langle \text{wff } x = y\rangle\rangle$$
$$\langle \varnothing, T, \varnothing, \langle \text{wff } Rxy\rangle\rangle$$

(C1′) $\langle \varnothing, T, \varnothing, \langle \vdash (\varphi \to (\psi \to \varphi))\rangle\rangle$

(C2′) $\langle \varnothing, T, \varnothing, \langle \vdash ((\varphi \to (\psi \to \chi)) \to ((\varphi \to \psi) \to (\varphi \to \chi)))\rangle\rangle$

(C3′) $\langle \varnothing, T, \varnothing, \langle \vdash ((\neg\varphi \to \neg\psi) \to (\psi \to \varphi))\rangle\rangle$

(C4′) $\langle \varnothing, T, \varnothing, \langle \vdash (\forall x(\forall x\, \varphi \to \psi) \to (\forall x\, \varphi \to \forall x\, \psi))\rangle\rangle$

(C5′) $\langle \varnothing, T, \varnothing, \langle \vdash (\forall x\, \varphi \to \varphi)\rangle\rangle$

(C6′) $\langle \varnothing, T, \varnothing, \langle \vdash (\forall x \forall y\, \varphi \to \forall y \forall x\, \varphi)\rangle\rangle$

(C7′) $\langle \varnothing, T, \varnothing, \langle \vdash (\neg\varphi \to \forall x \neg \forall x\, \varphi)\rangle\rangle$

(C8′) $\langle \varnothing, T, \varnothing, \langle \vdash (x = y \to (x = z \to y = z))\rangle\rangle$

(C9′) $\langle \varnothing, T, \varnothing, \langle \vdash (\neg\forall x\, x = y \to (\neg\forall x\, x = z \to (y = z \to \forall x\, y = z)))\rangle\rangle$

(C10′) $\langle \varnothing, T, \varnothing, \langle \vdash (\forall x(x = y \to \forall x\, \varphi) \to \varphi)\rangle\rangle$

(C11′) $\langle \varnothing, T, \varnothing, \langle \vdash (\forall x\, x = y \to (\forall x\, \varphi \to \forall y\, \varphi))\rangle\rangle$

(C12′) $\langle \varnothing, T, \varnothing, \langle \vdash (x = y \to (Rxz \to Ryz))\rangle\rangle$

(C13′) $\langle \varnothing, T, \varnothing, \langle \vdash (x = y \to (Rzx \to Rzy))\rangle\rangle$

(C15′) $\langle \varnothing, T, \varnothing, \langle \vdash (\neg\forall x\, x = y \to (x = y \to (\varphi \to \forall x(x = y \to \varphi))))\rangle\rangle$

(C16′) $\langle \{\{x, y\}\}, T, \varnothing, \langle \vdash (\forall x\, x = y \to (\varphi \to \forall x\, \varphi))\rangle\rangle$

(C5) $\langle \{\{x, \varphi\}\}, T, \varnothing, \langle \vdash (\varphi \to \forall x\, \varphi)\rangle\rangle$

(MP) $\langle \varnothing, T, \{\langle \vdash (\varphi \to \psi)\rangle, \langle \vdash \varphi\rangle\}, \langle \vdash \psi\rangle\rangle$

(Gen) $\langle \varnothing, T, \{\langle \vdash \varphi\rangle\}, \langle \vdash \forall x\, \varphi\rangle\rangle$

While it is known that these axioms are "metalogically complete," it is not known whether they are independent (i.e. none is redundant) in the metalogical sense; specifically, whether any axiom (possibly with additional non-mandatory distinct-variable restrictions, for use with any dummy

variables in its proof) is provable from the others. Note that metalogical independence is a weaker requirement than independence in the usual logical sense. Not all of the above axioms are logically independent: for example, C9′ can be proved as a metatheorem from the others, outside the formal system, by combining the possible cases of distinct variables.

C.3.6 Example 4—Adding Definitions

There are several ways to add definitions to a formal system. Probably the most proper way is to consider definitions not as part of the formal system at all but rather as abbreviations that are part of the expository metalogic outside the formal system. For convenience, though, we may use the formal system itself to incorporate definitions, adding them as axiomatic extensions to the system. This could be done by adding a constant representing the concept "is defined as" along with axioms for it. But there is a nicer way, at least in this writer's opinion, that introduces definitions as direct extensions to the language rather than as extralogical primitive notions. We introduce additional logical connectives and provide axioms for them. For systems of logic such as Examples 1 through 3, the additional axioms must be conservative in the sense that no wff of the original system that was not a theorem (when the initial term "wff" is replaced by "$\vdash$" of course) becomes a theorem of the extended system. In this example we extend Example 3 (or 2) with standard abbreviations of logic.

We extend *CN* of Example 3 with new constants $\{\leftrightarrow, \wedge, \vee, \exists\}$, corresponding to logical equivalence, conjunction, disjunction, and the existential quantifier. We extend Γ with the axiomatic statements that are the reducts of the following pre-statements:

$$\langle \varnothing, T, \varnothing, \langle \text{wff } (\varphi \leftrightarrow \psi) \rangle \rangle$$
$$\langle \varnothing, T, \varnothing, \langle \text{wff } (\varphi \vee \psi) \rangle \rangle$$
$$\langle \varnothing, T, \varnothing, \langle \text{wff } (\varphi \wedge \psi) \rangle \rangle$$
$$\langle \varnothing, T, \varnothing, \langle \text{wff } \exists x\, \varphi \rangle \rangle$$
$$\langle \varnothing, T, \varnothing, \langle \vdash ((\varphi \leftrightarrow \psi) \rightarrow (\varphi \rightarrow \psi)) \rangle \rangle$$
$$\langle \varnothing, T, \varnothing, \langle \vdash ((\varphi \leftrightarrow \psi) \rightarrow (\psi \rightarrow \varphi)) \rangle \rangle$$
$$\langle \varnothing, T, \varnothing, \langle \vdash ((\varphi \rightarrow \psi) \rightarrow ((\psi \rightarrow \varphi) \rightarrow (\varphi \leftrightarrow \psi))) \rangle \rangle$$
$$\langle \varnothing, T, \varnothing, \langle \vdash ((\varphi \wedge \psi) \leftrightarrow \neg(\varphi \rightarrow \neg\psi)) \rangle \rangle$$
$$\langle \varnothing, T, \varnothing, \langle \vdash ((\varphi \vee \psi) \leftrightarrow (\neg\varphi \rightarrow \psi)) \rangle \rangle$$
$$\langle \varnothing, T, \varnothing, \langle \vdash (\exists x\, \varphi \leftrightarrow \neg\forall x \neg\varphi) \rangle \rangle$$

The first three logical axioms (statements containing "$\vdash$") introduce and effectively define logical equivalence, "$\leftrightarrow$". The last three use "$\leftrightarrow$" to effectively mean "is defined as."

C.3.7 Example 5—ZFC Set Theory

Here we add to the system of Example 4 the axioms of Zermelo–Fraenkel set theory with Choice. For convenience we make use of the definitions in Example 4.

In the *CN* of Example 4 (which extends Example 3), we replace the symbol R with the symbol $\in$. More explicitly, we remove from Γ of Example 4 the three axiomatic statements containing R and replace them with the reducts of the following:

$$\langle \varnothing, T, \varnothing, \langle \text{wff } x \in y \rangle \rangle$$
$$\langle \varnothing, T, \varnothing, \langle \vdash (x = y \to (x \in z \to y \in z)) \rangle \rangle$$
$$\langle \varnothing, T, \varnothing, \langle \vdash (x = y \to (z \in x \to z \in y)) \rangle \rangle$$

Letting $D = \{\{\alpha, \beta\} \in DV | \alpha, \beta \in Vr\}$ (in other words all individual variables must be distinct), we extend Γ with the ZFC axioms, called Extensionality, Replacement, Union, Power Set, Regularity, Infinity, and Choice, that are the reducts of:

Ext $\langle D, T, \varnothing, \langle \vdash (\forall x(x \in y \leftrightarrow x \in z) \to y = z) \rangle \rangle$

Rep $\langle D, T, \varnothing, \langle \vdash \exists x(\exists y \forall z(\varphi \to z = y) \to \forall z(z \in x \leftrightarrow \exists x(x \in y \wedge \forall y\, \varphi))) \rangle \rangle$

Un $\langle D, T, \varnothing, \langle \vdash \exists x \forall y(\exists x(y \in x \wedge x \in z) \to y \in x) \rangle \rangle$

Pow $\langle D, T, \varnothing, \langle \vdash \exists x \forall y(\forall x(x \in y \to x \in z) \to y \in x) \rangle \rangle$

Reg $\langle D, T, \varnothing, \langle \vdash (x \in y \to \exists x(x \in y \wedge \forall z(z \in x \to \neg z \in y))) \rangle \rangle$

Inf $\langle D, T, \varnothing, \langle \vdash \exists x(y \in x \wedge \forall y(y \in x \to \exists z(y \in z \wedge z \in x))) \rangle \rangle$

AC $\langle D, T, \varnothing, \langle \vdash \exists x \forall y \forall z((y \in z \wedge z \in w) \to \exists w \forall y(\exists w((y \in z \wedge z \in w) \wedge (y \in w \wedge w \in x)) \leftrightarrow y = w)) \rangle \rangle$

C.3.8 Example 6—Class Notation in Set Theory

A powerful device that makes set theory easier (and that we have been using all along in our informal expository language) is *class abstraction notation*. The definitions we introduce are rigorously justified as conservative by Takeuti and Zaring [67] or Quine [55]. The key idea is to introduce the notation $\{x|—\}$ which means "the class of all x such that —" for abstraction classes and introduce (meta)variables that range over them. An abstraction class may or may not be a set, depending on whether it exists (as a set). A class that does not exist is called a *proper class*.

To illustrate the use of abstraction classes we will provide some examples of definitions that make use of them: the empty set, class union, and unordered pair. Many other such definitions can be found in the Metamath set theory database, `set.mm`.

We extend *CN* of Example 5 with new symbols { class, {, |, }, $\varnothing$, $\cup$, , } where the inner braces and last comma are constant symbols. (As before,

our dual use of some mathematical symbols for both our expository language and as primitives of the formal system should be clear from context.)

We extend *VR* of Example 5 with a set of *class variables* $\{A, B, C, \ldots\}$. We extend the T of Example 5 with $\{\langle \text{class } A\rangle, \langle \text{class } B\rangle, \langle \text{class } C\rangle, \ldots\}$.

To introduce our definitions, we add to Γ of Example 5 the axiomatic statements that are the reducts of the following pre-statements:

$\langle \varnothing, T, \varnothing, \langle \text{class } x\rangle\rangle$

$\langle \varnothing, T, \varnothing, \langle \text{class } \{x|\varphi\}\rangle\rangle$

$\langle \varnothing, T, \varnothing, \langle \text{wff } A = B\rangle\rangle$

$\langle \varnothing, T, \varnothing, \langle \text{wff } A \in B\rangle\rangle$

Ab $\langle \varnothing, T, \varnothing, \langle \vdash (y \in \{x|\varphi\} \leftrightarrow ((x = y \rightarrow \varphi) \wedge \exists x(x = y \wedge \varphi)))\rangle\rangle$

Eq $\langle \{\{x, A\}, \{x, B\}\}, T, \varnothing, \langle \vdash (A = B \leftrightarrow \forall x(x \in A \leftrightarrow x \in B))\rangle\rangle$

El $\langle \{\{x, A\}, \{x, B\}\}, T, \varnothing, \langle \vdash (A \in B \leftrightarrow \exists x(x = A \wedge x \in B))\rangle\rangle$

Here we say that an individual variable is a class; $\{x|\varphi\}$ is a class; and we extend the definition of a wff to include class equality and membership. Axiom Ab defines membership of a variable in a class abstraction; the right-hand side can be read as "the wff that results from proper substitution of y for x in φ."[6] Axioms Eq and El extend the meaning of the existing equality and membership connectives. This is potentially dangerous and requires careful justification. For example, from Eq we can derive the Axiom of Extensionality with predicate logic alone; thus in principle we should include the Axiom of Extensionality as a logical hypothesis. However we do not bother to do this since we have already presupposed that axiom earlier. The distinct variable restrictions should be read "where x does not occur in A or B." We typically do this when the right-hand side of a definition involves an individual variable not in the expression being defined; it is done so that the right-hand side remains independent of the particular "dummy" variable we use.

We continue to add to Γ the following definitions (i.e. the reducts of the following pre-statements) for empty set, class union, and unordered pair. They should be self-explanatory. Analogous to our use of "$\leftrightarrow$" to define new wffs in Example 4, we use "$=$" to define new abstraction terms, and both may be read informally as "is defined as" in this context.

$\langle \varnothing, T, \varnothing, \langle \text{class } \varnothing\rangle\rangle$

$\langle \varnothing, T, \varnothing, \langle \vdash \varnothing = \{x|\neg x = x\}\rangle\rangle$

$\langle \varnothing, T, \varnothing, \langle \text{class } (A \cup B)\rangle\rangle$

[6]Note that this definition makes unnecessary the introduction of a separate notation similar to $\varphi(x|y)$ for proper substitution, although we may choose to do so to be conventional. Incidentally, $\varphi(x|y)$ as it stands would be ambiguous in the formal systems of our examples, since we wouldn't know whether $\neg\varphi(x|y)$ meant $\neg(\varphi(x|y))$ or $(\neg\varphi)(x|y)$. Instead, we would have to use an unambiguous variant such as $(\varphi\, x|y)$.

$\langle\{\{x, A\}, \{x, B\}\}, T, \varnothing, \langle\vdash (A \cup B) = \{x|(x \in A \vee x \in B)\}\rangle\rangle$
$\langle\varnothing, T, \varnothing, \langle\text{class } \{A, B\}\rangle\rangle$
$\langle\{\{x, A\}, \{x, B\}\}, T, \varnothing, \langle\vdash \{A, B\} = \{x|(x = A \vee x = B)\}\rangle\rangle$

C.4 Metamath as a Formal System

This section presupposes a familiarity with the Metamath computer language.

Our theory describes formal systems and their universes. The Metamath language provides a way of representing these set-theoretical objects to a computer. A Metamath database, being a finite set of ASCII characters, can usually describe only a subset of a formal system and its universe, which are typically infinite. However the database can contain as large a finite subset of the formal system and its universe as we wish. (Of course a Metamath set theory database can, in principle, indirectly describe an entire infinite formal system by formalizing the expository language in this Appendix.)

For purpose of our discussion, we assume the Metamath database is in the simple form described on p. 132, consisting of all constant and variable declarations at the beginning, followed by a sequence of extended frames each delimited by `$\{` and `$\}`. Any Metamath database can be converted to this form, as described on p. 134.

The math symbol tokens of a Metamath source file, which are declared with `$c` and `$v` statements, are names we assign to representatives of *CN* and *VR*. For definiteness we could assume that the first math symbol declared as a variable corresponds to v_0, the second to v_1, etc., although the exact correspondence we choose is not important.

In the Metamath language, each `$d`, `$f`, and `$e` source statement in an extended frame (Section 4.2.7) corresponds respectively to a member of the collections D, T, and H in a formal system statement $\langle D_M, T_M, H, A\rangle$. The math symbol strings following these Metamath keywords correspond to a variable pair (in the case of `$d`) or an expression (for the other two keywords). The math symbol string following a `$a` source statement corresponds to expression A in an axiomatic statement of the formal system; the one following a `$p` source statement corresponds to A in a provable statement that is not axiomatic. In other words, each extended frame in a Metamath database corresponds to a pre-statement of the formal system, and a frame corresponds to a statement of the formal system. (Don't confuse the two meanings of "statement" here. A statement of the formal system corresponds to the several statements in a Metamath database that may constitute a frame.)

In order for the computer to verify that a formal system statement is provable, each `$p` source statement is accompanied by a proof. However, the proof does not correspond to anything in the formal system but is simply a way of communicating to the computer the information needed for its

verification. The proof tells the computer *how to construct* specific members of closure of the formal system pre-statement corresponding to the extended frame of the `$p` statement. The final result of the construction is the member of the closure that matches the `$p` statement. The abstract formal system, on the other hand, is concerned only with the *existence* of members of the closure.

As mentioned on p. 194, Examples 1 and 3–6 in the previous Section parallel the development of logic and set theory in the Metamath database `set.mm`. You may find it instructive to compare them.

Appendix D

The MIU System

The following is a listing of the file `miu.mm`. It is self-explanatory.

```
$( The MIU-system:  A simple formal system $)

$( Note:  This formal system is unusual in that it allows
empty wffs.  To work with a proof, you must type
SET EMPTY_SUBSTITUTION ON before using the PROVE command.
By default, this is OFF in order to reduce the number of
ambiguous unification possibilities that have to be selected
during the construction of a proof.  $)

$(
Hofstadter's MIU-system is a simple example of a formal
system that illustrates some concepts of Metamath.  See
Douglas R. Hofstadter, _Goedel, Escher, Bach:  An Eternal
Golden Braid_ (Vintage Books, New York, 1979), pp. 33ff. for
a description of the MIU-system.

The system has 3 constant symbols, M, I, and U.  The sole
axiom of the system is MI. There are 4 rules:
     Rule I:  If you possess a string whose last letter is I,
     you can add on a U at the end.
     Rule II:  Suppose you have Mx.  Then you may add Mxx to
     your collection.
     Rule III:  If III occurs in one of the strings in your
     collection, you may make a new string with U in place
     of III.
     Rule IV:  If UU occurs inside one of your strings, you
     can drop it.
Unfortunately, Rules III and IV do not have unique results:
```

```
strings could have more than one occurrence of III or UU.
This requires that we introduce the concept of an "MIU
well-formed formula" or wff, which allows us to construct
unique symbol sequences to which Rules III and IV can be
applied.
$)

$( First, we declare the constant symbols of the language.
Note that we need two symbols to distinguish the assertion
that a sequence is a wff from the assertion that it is a
theorem; we have arbitrarily chosen "wff" and "|-". $)
      $c M I U |- wff $. $( Declare constants $)

$( Next, we declare some variables. $)
     $v x y $.

$( Throughout our theory, we shall assume that these
variables represent wffs. $)
 wx   $f wff x $.
 wy   $f wff y $.

$( Define MIU-wffs.  We allow the empty sequence to be a
wff. $)

$( The empty sequence is a wff. $)
 we   $a wff $.
$( "M" after any wff is a wff. $)
 wM   $a wff x M $.
$( "I" after any wff is a wff. $)
 wI   $a wff x I $.
$( "U" after any wff is a wff. $)
 wU   $a wff x U $.

$( Assert the axiom. $)
 ax   $a |- M I $.

$( Assert the rules. $)
 ${
   Ia   $e |- x I $.
$( Given any theorem ending with "I", it remains a theorem
if "U" is added after it.  (We distinguish the label I_
from the math symbol I to conform to the 24-Jun-2006
Metamath spec.) $)
   I_    $a |- x I U $.
```

```
 $}
 ${
IIa  $e |- M x $.
$( Given any theorem starting with "M", it remains a theorem
if the part after the "M" is added again after it. $)
   II   $a |- M x x $.
 $}
 ${
   IIIa $e |- x I I I y $.
$( Given any theorem with "III" in the middle, it remains a
theorem if the "III" is replaced with "U". $)
   III  $a |- x U y $.
 $}
 ${
   IVa  $e |- x U U y $.
$( Given any theorem with "UU" in the middle, it remains a
theorem if the "UU" is deleted. $)
   IV   $a |- x y $.
  $}

$( Now we prove the theorem MUIIU.  You may be interested in
comparing this proof with that of Hofstadter (pp. 35 - 36).
$)
 theorem1  $p |- M U I I U $=
      we wM wU wI we wI wU we wU wI wU we wM we wI wU we wM
      wI wI wI we wI wI we wI ax II II I_ III II IV $.
```

The `show proof /lemmon/renumber` command yields the following display. It is very similar to the one in [26, pp. 35–36].

```
1 ax            $a |- M I
2 1 II          $a |- M I I
3 2 II          $a |- M I I I I
4 3 I_          $a |- M I I I I U
5 4 III         $a |- M U I U
6 5 II          $a |- M U I U U I U
7 6 IV          $a |- M U I I U
```

We note that Hofstadter's "MU-puzzle," which asks whether MU is a theorem of the MIU-system, cannot be answered using the system above because the MU-puzzle is a question *about* the system. To prove the answer to the MU-puzzle, a much more elaborate system is needed, namely one that models the MIU-system within set theory. (Incidentally, the answer to the MU-puzzle is no.)

Appendix E

Metamath Language EBNF

The following is a formal description of the basic Metamath language syntax (with compressed proofs and support for unknown proof steps). It is defined using the Extended Backus–Naur Form (EBNF) notation from W3C *Extensible Markup Language (XML) 1.0 (Fifth Edition)* (W3C Recommendation 26 November 2008) at `https://www.w3.org/TR/xml/#sec-notation`.

The `database` rule is processed until the end of the file (`EOF`). The rules eventually require reading whitespace-separated tokens. A token has an upper-case definition (see below) or is a string constant in a non-token (such as `'$a'`). We intend for this to be correct, but if there is a conflict the rules of section 4.1 govern. That section also discusses non-syntax restrictions not shown here (e.g., that each new label token defined in a `hypothesis-stmt` or `assert-stmt` must be unique).

```
database ::= outermost-scope-stmt*

outermost-scope-stmt ::=
  include-stmt | constant-stmt | stmt

/* File inclusion command; process file as a database.
   Databases should NOT have a comment in the filename. */
include-stmt ::= '$[' filename '$]'

/* Constant symbols declaration. */
constant-stmt ::= '$c' constant+ '$.'

/* A normal statement can occur in any scope. */
stmt ::= block | variable-stmt | disjoint-stmt |
  hypothesis-stmt | assert-stmt
```

```
/* A block. You can have 0 statements in a block. */
block ::= '${' stmt* '$}'

/* Variable symbols declaration. */
variable-stmt ::= '$v' variable+ '$.'

/* Disjoint variables. Simple disjoint statements have
   2 variables, i.e., "variable*" is empty for them. */
disjoint-stmt ::= '$d' variable variable variable* '$.'

hypothesis-stmt ::= floating-stmt | essential-stmt

/* Floating (variable-type) hypothesis. */
floating-stmt ::= LABEL '$f' typecode variable '$.'

/* Essential (logical) hypothesis. */
essential-stmt ::= LABEL '$e' typecode MATH-SYMBOL* '$.'

assert-stmt ::= axiom-stmt | provable-stmt

/* Axiomatic assertion. */
axiom-stmt ::= LABEL '$a' typecode MATH-SYMBOL* '$.'

/* Provable assertion. */
provable-stmt ::= LABEL '$p' typecode MATH-SYMBOL*
  '$=' proof '$.'

/* A proof. Proofs may be interspersed by comments.
   If '?' is in a proof it's an "incomplete" proof. */
proof ::= uncompressed-proof | compressed-proof
uncompressed-proof ::= (LABEL | '?')+
compressed-proof ::= '(' LABEL* ')' COMPRESSED-PROOF-BLOCK+

typecode ::= constant

filename ::= MATH-SYMBOL /* No whitespace or '$' */
constant ::= MATH-SYMBOL
variable ::= MATH-SYMBOL
```

A `frame` is a sequence of 0 or more `disjoint-stmt` and `hypotheses-stmt` statements (possibly interleaved with other non-`assert-stmt` statements) followed by one `assert-stmt`.

Here are the rules for lexical processing (tokenization) beyond the constant tokens shown above. By convention these tokenization rules have upper-case names. Every token is read for the longest possible length. Whitespace-separated tokens are read sequentially; note that the separating whitespace and `$( ... $)` comments are skipped.

If a token definition uses another token definition, the whole thing is considered a single token. A pattern that is only part of a full token has a name beginning with an underscore ("_"). An implementation could tokenize many tokens as a `PRINTABLE-SEQUENCE` and then check if it meets the more specific rule shown here.

Comments do not nest, and both `$(` and `$)` have to be surrounded by at least one whitespace character (`_WHITECHAR`). Technically comments end without consuming the trailing `_WHITECHAR`, but the trailing `_WHITECHAR` gets ignored anyway so we ignore that detail here. Metamath language processors are not required to support `$)` followed immediately by a bare end-of-file, because the closing comment symbol is supposed to be followed by a `_WHITECHAR` such as a newline.

```
PRINTABLE-SEQUENCE ::= _PRINTABLE-CHARACTER+

MATH-SYMBOL ::= (_PRINTABLE-CHARACTER - '$')+

/* ASCII non-whitespace printable characters */
_PRINTABLE-CHARACTER ::= [#x21-#x7e]

LABEL ::= ( _LETTER-OR-DIGIT | '.' | '-' | '_' )+

_LETTER-OR-DIGIT ::= [A-Za-z0-9]

COMPRESSED-PROOF-BLOCK ::= ([A-Z] | '?')+

/* Define whitespace between tokens. The -> SKIP
   means that when whitespace is seen, it is
   skipped and we simply read again. */
WHITESPACE ::= (_WHITECHAR+ | _COMMENT) -> SKIP

/* Comments. $( ... $) and do not nest. */
_COMMENT ::= '$(' (_WHITECHAR+ (PRINTABLE-SEQUENCE - '$)')*
  _WHITECHAR+ '$)' _WHITECHAR

/* Whitespace: (' ' | '\t' | '\r' | '\n' | '\f') */
_WHITECHAR ::= [#x20#x09#x0d#x0a#x0c]
```

Bibliography

[1] Donald J. Albers and G. L. Alexanderson, editors. *Mathematical People*. Contemporary Books, Inc., Chicago, 1985. [QA28.M37].

[2] Alan Ross Anderson and Nuel D. Belnap. *Entailment*, volume 1. Princeton University Press, Princeton, 1975. [QA9.A634 1975 v.1].

[3] John D. Barrow. *Theories of Everything: The Quest for Ultimate Explanation*. Oxford University Press, Oxford, 1991. [Q175.B225].

[4] H. Behnke, F. Backmann, K. Fladt, and W. Süss, editors. *Fundamentals of Mathematics*, volume I. The MIT Press, Cambridge, Massachusetts, 1974. [QA37.2.B413].

[5] J. L. Bell and M. Machover. *A Course in Mathematical Logic*. North-Holland, Amsterdam, 1977. [QA9.B3953].

[6] Andrea Blass. The interaction between category theory and set theory. In John Walter Gray, editor, *Mathematical Applications of Category Theory (Proceedings of the Special Session on Mathematical Applications Category Theory, 89th Annual Meeting of the American Mathematical Society, held in Denver, Colorado January 5–9, 1983)*, pages 5–29, Providence, Rhode Island, 1983. American Mathematical Society. [QA169.A47 1983].

[7] W. W. Bledsoe and D. W. Loveland, editors. *Automated Theorem Proving: After 25 Years (Proceedings of the Special Session on Automatic Theorem Proving, 89th Annual Meeting of the American Mathematical Society, held in Denver, Colorado January 5–9, 1983)*, Providence, Rhode Island, 1983. American Mathematical Society. [QA76.9.A96.S64 1983].

[8] George S. Boolos and Richard C. Jeffrey. *Computability and Logic*. Cambridge University Press, Cambridge, third edition, 1989. [QA9.59.B66 1989].

[9] John Campbell. *Programmer's Progress*. White Star Software, Box 51623, Palo Alto, CA 94303, 1991.

[10] Mario Carneiro. Conversion of HOL light proofs into metamath. *CoRR*, abs/1412.8091, 2014.

[11] Mario Carneiro. Natural deductions in the metamath proof language. 2014.

[12] Shang-Ching Chou. Proving elementary geometry theorems using Wu's algorithm. In W. W. Bledsoe and D. W. Loveland, editors, *Automated Theorem Proving: After 25 Years (Proceedings of the Special Session on Automatic Theorem Proving, 89th Annual Meeting of the American Mathematical Society, held in Denver, Colorado January 5–9, 1983)*, pages 243–286, Providence, Rhode Island, 1983. American Mathematical Society. [QA76.9.A96.S64 1983].

[13] Richard Courant and Herbert Robbins. Topology. In James R. Newman, editor, *The World of Mathematics, Volume One*, pages 573–590. Simon and Schuster, New York, 1956. [QA3.W67 1988].

[14] Haskell B. Curry. *Foundations of Mathematical Logic*. Dover Publications, Inc., New York, 1977. [QA9.C976 1977].

[15] Philip J. Davis and Reuben Hersh. *The Mathematical Experience*. Birkhäuser Boston, Boston, 1981. [QA8.4.D37 1982].

[16] Richard de Millo, Richard Lipton, and Alan Perlis. Social processes and proofs of theorems and programs. In Thomas Tymoczko, editor, *New Directions in the Philosophy of Mathematics*, pages 267–285. Birkhäuser Boston, Inc., Boston, 1986. [QA8.6.N48 1986].

[17] Robert E. Edwards. *A Formal Background to Mathematics*. Springer-Verlag, New York, 1979. [QA37.2.E38 v.1a].

[18] Herbert B. Enderton. *Elements of Set Theory*. Academic Press, Inc., San Diego, 1977. [QA248.E5].

[19] R. L. Goodstein. *Development of Mathematical Logic*. Springer-Verlag New York Inc., New York, 1971. [QA9.G6554].

[20] Michael Guillen. *Bridges to Infinity*. Jeremy P. Tarcher, Inc., Los Angeles, 1983. [QA93.G8].

[21] Alan G. Hamilton. *Logic for Mathematicians*. Cambridge University Press, Cambridge, revised edition, 1988. [QA9.H298].

[22] John Robert Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053. SRI Cambridge, Millers Yard, Cambridge, UK, 1995. Available on the Web as `http://www.cl.cam.ac.uk/users/jrh/papers/reflect.html`.

[23] John Robert Harrison. Theorem proving with the real numbers. Technical Report 408, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK, 1996. Author's PhD thesis, available on the Web at `http://www.cl.cam.ac.uk-/users/jrh/papers/thesis.html`.

[24] Horst Herrlich and George E. Strecker. *Category Theory: An Introduction.* Allyn and Bacon Inc., Boston, 1973. [QA169.H567].

[25] J. Roger Hindley and David Meredith. Principal type-schemes and condensed detachment. *The Journal of Symbolic Logic*, 55:90–105, 1990. [QA.J87].

[26] Douglas R. Hofstadter. *Gödel, Escher, Bach.* Basic Books, Inc., New York, 1979. [QA9.H63 1980].

[27] Andrzej Indrzejczak. Natural deduction, hybrid systems and modal logic. *Trends in Logic*, 30, 2010.

[28] D. Kalish and R. Montague. On Tarski's formalization of predicate logic with identity. *Archiv für Mathematische Logik und Grundlagenforschung*, 7:81–101, 1965. [QA.A673].

[29] J. A. Kalman. Condensed detachment as a rule of inference. *Studia Logica*, 42(4):443–451, 1983. [B18.P6.S933].

[30] Morris Kline. *Mathematical Thought from Ancient to Modern Times.* Oxford University Press, New York, 1972. [QA21.K516 1990 v.3].

[31] Morris Kline. *Mathematics, The Loss of Certainty.* Oxford University Press, New York, 1980. [QA21.K525].

[32] Oliver Knill. Some fundamental theorems in mathematics. 2018.

[33] Edna E. Kramer. *The Nature and Growth of Modern Mathematics.* Princeton University Press, Princeton, New Jersey, 1981. [QA93.K89 1981].

[34] Daniel Clemente Laboreo. *Introduction to natural deduction.* 2014.

[35] Edmund Landau. *Foundations of Analysis.* Chelsea Publishing Company, New York, second edition, 1960. [QA241.L2541 1960].

[36] Hugues Leblanc. On Meyer and Lambert's quantificational calculus FQ. *The Journal of Symbolic Logic*, 33:275–280, 1968. [QA.J87].

[37] Czeslaw Lejewski. On implicational definitions. *Studia Logica*, 8:189–208, 1958. [B18.P6.S933].

[38] Azriel Levy. *Basic Set Theory.* Dover Publications, Mineola, NY, 2002.

[39] Angelo Margaris. *First Order Mathematical Logic*. Blaisdell Publishing Company, Waltham, Massachusetts, 1967. [QA9.M327].

[40] Adrian R. D. Mathias. A term of length 4,523,659,424,929. *Synthese*, 133:75–86, 2002. [Q.S993].

[41] Norman D. Megill. A finitely axiomatized formalization of predicate calculus with equality. *Notre Dame Journal of Formal Logic*, 36:435–453, 1995. [QA.N914].

[42] Norman D. Megill and Martin W. Bunder. Weaker D-complete logics. *Journal of the IGPL*, 4:215–225, 1996. Available on the Web at `http://www.mpi-sb.mpg.de/igpl/Journal/V4-2/#Megill`.

[43] Elliott Mendelson. *Introduction to Mathematical Logic*. D. Van Nostrand Company, Inc., New York, second edition, 1979. [QA9.M537 1979].

[44] C. A. Meredith. Single axioms for the systems (C,N), (C,O) and (A,N) of the two-valued propositional calculus. *The Journal of Computing Systems*, 3:155–164, 1953.

[45] David Meredith. In memoriam Carew Arthur Meredith (1904-1976). *Notre Dame Journal of Formal Logic*, 18:513–516, 1977. [QA.N914].

[46] J. Donald Monk. Substitutionless predicate logic with identity. *Archiv für Mathematische Logik und Grundlagenforschung*, 7:103–121, 1965.

[47] A. W. Moore. *The Infinite*. Routledge, New York, 1989. [BD411.M59].

[48] James R. Munkres. *Topology: A First Course*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1975. [QA611.M82].

[49] E. Z. Nemesszeghy and E. A. Nemesszeghy. On strongly creative definitions: A reply to V. F. Rickey. *Logique et Analyse (N. S.)*, 20:111–115, 1977. [BC.L832].

[50] I. Németi. Algebraizations of quantifier logics, an overview. Version 11.4, preprint, Mathematical Institute, Budapest, 1994. A shortened version without proofs appeared in "Algebraizations of quantifier logics, an introductory overview," *Studia Logica*, 50:485–569, 1991 [B18.P6.S933].

[51] M. Pavičić. A new axiomatization of unified quantum logic. *International Journal of Theoretical Physics*, 31:1753 –1766, 1992. [QC.I626].

[52] Roger Penrose. *The Emperor's New Mind*. Oxford University Press, New York, 1989. [Q335.P415].

[53] Ivars Peterson. *The Mathematical Tourist*. W. H. Freeman and Company, New York, 1988. [QA93.P475].

[54] Jeremy George Peterson. An automatic theorem prover for substitution and detachment systems. *Notre Dame Journal of Formal Logic*, 19:119–122, 1978. [QA.N914].

[55] Willard Van Orman Quine. *Set Theory and Its Logic*. The Belknap Press of Harvard University Press, Cambridge, Massachusetts, revised edition, 1969. [QA248.Q7 1969].

[56] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12:23–41, 1965.

[57] T. Thacher Robinson. Independence of two nice sets of axioms for the propositional calculus. *The Journal of Symbolic Logic*, 33:265–270, 1968. [QA.J87].

[58] Rudy Rucker. *Infinity and the Mind: The Science and Philosophy of the Infinite*. Bantam Books, Inc., New York, 1982. [QA9.R79 1982].

[59] Bertrand Russell. Recent work on the principles of mathematics. *International Monthly*, 4:84, 1901.

[60] Bertrand Russell. *Mysticism and Logic, and Other Essays*. Barnes & Noble Books, Totowa, New Jersey, 1981. [B1649.R963.M9 1981].

[61] Eric Schmidt. Reductions in norman megill's axiom system for complex numbers. 2012.

[62] Joseph R. Shoenfield. *Mathematical Logic*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1967. [QA9.S52].

[63] Daniel Solow. *How to Read and Do Proofs: An Introduction to Mathematical Thought Process*. John Wiley & Sons, New York, 1982. [QA9.S577].

[64] Harold M. Stark. *An Introduction to Number Theory*. Markham Publishing Company, Chicago, 1970. [QA241.S72 1978].

[65] E. R. Swart. The philosophical implications of the four-color problem. *American Mathematical Monthly*, 87:697–707, November 1980. [QA.A5125].

[66] George G. Szpiro. *Poincaré's Prize: The Hundred-Year Quest to Solve One of Math's Greatest Puzzles*. Penguin Books Ltd, London, 2007. [QA43.S985 2007].

[67] Gaisi Takeuti and Wilson M. Zaring. *Introduction to Axiomatic Set Theory*. Springer-Verlag New York Inc., New York, second edition, 1982. [QA248.T136 1982].

[68] Alfred Tarski. What is elementary geometry. In Leon Henkin, Patrick Suppes, and Alfred Tarski, editors, *The Axiomatic Method, with Special Reference to Geometry and Physics (Proceedings of an International Symposium held at the University of California, Berkeley, December 26, 1957 — January 4, 1958)*, pages 16–29, Amsterdam, 1959. North-Holland Publishing Company.

[69] Alfred Tarski. A simplified formalization of predicate logic with identity. *Archiv für Mathematische Logik und Grundlagenforschung*, 7:61–79, 1965. [QA.A673].

[70] Thomas Tymoczko, editor. *New Directions in the Philosophy of Mathematics*. Birkhäuser Boston, Inc., Boston, 1986. [QA8.6.N48 1986].

[71] Hao Wang. Theory and practice in mathematics. In Thomas Tymoczko, editor, *New Directions in the Philosophy of Mathematics*, pages 129–152. Birkhäuser Boston, Inc., Boston, 1986. [QA8.6.N48 1986].

[72] Daniel Whalen. Holophrasm: a neural automated theorem prover for higher-order logic. *CoRR*, abs/1608.02644, 2016.

[73] Alfred North Whitehead. *An Introduction to Mathematics*, 1911.

[74] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica*. Cambridge University Press, Cambridge, second edition, 1927. (3 vols.) [QA9.W592 1927].

[75] Freek Wiedijk. The qed manifesto revisited. 2007.

[76] Stephen Wolfram. *Mathematica: A System for Doing Mathematics by Computer*. Addison-Wesley Publishing Co., Redwood City, California, second edition, 1991. [QA76.95.W65 1991].

[77] Larry Wos, Ross Overbeek, Ewing Lusk, and Jim Boyle. *Automated Reasoning: Introduction and Applications*. McGraw-Hill, Inc., New York, second edition, 1992. [QA76.9.A96.A93 1992].

Index

&, 97
⇒, 97
⇒, 94
`minimize_with` command, 98
`$(` and `$)` auxiliary keywords, 42, 113, 117, 139
`$.` keyword, 42, 43, 50, 116, 118
`$=` keyword, 43, 50, 116, 118, 127, 135, 149, 150
`$[` and `$]` auxiliary keywords, 113, 117, 148
`${` and `$}` keywords, 43, 116, 131–133
`$a` statement, 42, 43, 108, 114, 116, 118, 124, 127–129, 133, 135, 150, 164
`$c` statement, 42, 43, 113, 116, 119, 120, 133
`$d` statement, 66, 71, 115, 116, 120–122, 124, 125, 133, 138, 170, 175
  compound, 114
  simple, 114
`$e` statement, 42, 43, 114, 116, 118, 125, 126, 133, 135, 164, 166, 172, 175
`$f` statement, 42, 43, 49, 102, 114, 116, 118, 125, 126, 133, 135, 164, 165, 174, 175
`$j` comment, 147
`$p` statement, 42, 43, 114, 116, 118, 126–128, 133, 135, 137, 150, 164, 165
`$t` comment, 118, 140, 141, 144, 183
`$v` statement, 42, 43, 113, 116, 119, 120, 133
`[...]` inside comments, 146
`_` inside comments, 143
`~` inside comments, 140
`?` in command lines, 50
`?` inside proofs, 50, 150
‘ inside comments, 140
`althtmldef` statement, 146
`althtmldir` statement, 146
`assign` command, 52, 172
`beep` command, 161
`close log` command, 160
`close tex` command, 177
`delete` command, 174
`erase` command, 160
`exit` command, 49, 159
`exthtmlbibliography` statement, 146
`exthtmlhome` statement, 146
`exthtmllabel` statement, 147
`exthtmltitle` statement, 146
`help` command, 49
`htmlbibliography` statement, 146
`htmldef` statement, 146
`htmldir` statement, 146
`htmlhome` statement, 146
`htmltitle` statement, 146
`htmlvarcolor` statement, 146
`improve` command, 175
`initialize` command, 174
`latexdef` statement, 146
`let` command, 173
`match` command, 172
`minimize_with` command, 57, 170
`more` command, 161

`open log` command, 159
`open tex` command, 118, 176
`prove` command, 51, 170
`read` command, 45, 49, 50, 102, 162
`redo` command, 170
`save new_proof` command, 56, 150, 175
`save proof` command, 110, 150, 167
`search` command, 103, 108, 164
`set echo` command, 160
`set empty_substitution` command, 53, 171
`set height` command, 161
`set scroll` command, 160
`set search_limit` command, 171
`set unification_timeout` command, 170
`set width` command, 161
`show trace_back` command, 166
`show labels` command, 46, 163
`show memory` command, 163
`show new_proof` command, 52, 171
`show proof` command, 41, 46, 49, 51, 56, 105, 138, 165
`show settings` command, 163
`show statement` command, 43, 46, 57, 70, 74, 103, 104, 164
`show trace_back` command, 17, 108
`show usage` command, 109, 166
`submit` command, 49, 160
`syl`, 87
`tools` command, 164, 179
`undo` command, 170
`unify` command, 173
`verify markup` command, 167
`verify proof` command, 45, 135, 150, 166
`write bibliography` command, 178
`write recent_additions` command, 178
`write source` command, 50, 163
`write theorem_list` command, 178

abstract algebra, 3
abstraction class, 61, 77, 202
    of ordered pairs, 81
active math symbol, 114, 120, 133
active statement, 114, 133
addition, 154
    of ordinals, 153
additional information comment, 147
ambiguous unification, 53, 139, 171, 175
analysis, 3, 31
Anderson, Alan Ross, 17
Andréka, H., 125, 192
Appel, K., 23
artificial intelligence, xi
ASCII, 37, 112, 116
assertion, 43, 44, 114, 126, 127, 135
    in a formal system, 192
assertion label, 44, 135, 139
Auden, W. H., 5
Aussonderung, 87
automated proof verification, 19, 20
automated theorem proving, 21, 26, 27, 64
auxiliary keyword, 112, 116
axiom, ix, 2, 17, 19, 32, 34, 38–42, 59, 61, 108, 118, 127, 128, 150, 151
Axiom of Choice, 17, 69, 72, 202
Axiom of Extensionality, 68, 72, 74, 78, 202
Axiom of Infinity, 35, 69, 72, 89, 202
Axiom of Pairing, 68, 87
Axiom of Power Sets, 68, 72, 202
Axiom of Regularity, 68, 72, 202
Axiom of Replacement, 69, 72, 202
Axiom of Separation, 69, 87
Axiom of the Null Set, 35, 68, 87
Axiom of Union, 68, 72, 202
axiom scheme, 39, 63, 69
axiom vs. definition, 108, 128, 150, 151
axiomatic assertion, 43, 127
axiomatic statement
    in a formal system, 193
axioms for mathematics, 62

axioms of equality, 65
axioms of logic, 60
axioms of predicate calculus, 65, 71
axioms of predicate calculus - auxiliary, 71
axioms of propositional calculus, 62, 70
axioms of set theory, 68, 71

Barrow, John D., xiii
basic keyword, 116
basic language, 116, 119, 135, 167, 176
Behnke, H., 33, 151
Bell, J. L., 74
Bible, 16
biconditional ($\leftrightarrow$), 64, 71, 74, 75, 201
binary relation, 81, 90
Blass, Andrea, 34
Bledsoe, W. W., 27
block, 43, 113, 132, 133
  outermost, 113
Boolean algebra, 26
Boolos, George S., 15, 17
bound variable, 35, 120
Bourbaki, Nicolas, xiii, 15
brace notation, 61
Bunder, Martin, 17
Burali-Forti paradox, 88

Cantor's theorem, 88
Cantor, Georg, 23
cardinal, inaccessible, xv, 34
cardinal, transfinite, 23
cardinality, 23
Carneiro, Mario, xi, 30, 92, 99, 101
Cartesian product, 82
category theory, xv, 32, 34
certainty, 18
Chou, Shang-Ching, 25
citation, 142
class, 74, 77
  proper, 77, 202
class abstraction, 61, 77, 202
class difference, 79
class equality, 78
class membership, 78
class variable, 203
Clemente Laboreo, Daniel, 99
Clifford algebras, 24
closed form, 97
closure, 193
Cohen, Paul, 23
collection, 68
command keyword, 45, 49, 102
command line interface (CLI), 44, 49
command qualifier, 46
comment, 113, 139
comments
  markup notation, 139
compact proof, 109
composition, 83
compound declaration, 120
compressed proof, 44, 56, 115, 149, 150, 187
computer algebra system, 3, 21, 24
computer program bugs, x, xi, 8, 9, 12, 19, 21, 22, 25
computers and pure mathematics, 21
concatenation, 190
conclusion, 97
condensed detachment, 34
  and first-order logic, 34
conjunction ($\wedge$), 64, 71, 75, 106, 201
connective, 74
consistent theory, 18
constant, xii, 42, 44, 74, 113, 119, 133
  in a formal system, 190
  in predicate calculus, 68
constant declaration, 42, 119
constant-prefixed expression, 191
constant-variable pair, 191
constructive language, 20
constructivism, 17
continuous integration (CI), 154
continuum hypothesis, 23
Coq, 27
Courant, Richard, 23

Courier font, 37
Coxeter, H. S. M., 23
cranks, 13
creative definition, 73, 151
cross product, 82
Curry, Haskell B., 17

database, 42, 112, 116, 163
Davis, Phillip J., 1, 12, 14, 33
de Millo, Richard, 15
de Saint-Exupery, Antoine, 189
decidable theory, 25
decision procedure, 64
declaration, 113
deduction form, 97, 99
deduction style, 97
Deduction Theorem, 94, 101
deduction theorem, 11
definiendum, 73
definiens, 73
definition, 15, 72, 108, 118, 127, 128, 150, 151, 201
    creative, 73, 151
    eliminability, 73
    proper, 129, 151
df-3an, 75
`df-3or`, 75
`df-an`, 75
`df-or`, 75
disjoint sets, 69
disjoint variables, 66, 121
disjoint-variable restriction, 114
    in a formal system, 192
disjunction ($\vee$), 64, 75, 151, 201
distinct variables, 35, 66, 71, 120, 123
domain, 23, 83
dummy variable, 105, 131
    eliminating, 27, 125
    in definitions, 73

E prover, 27
EBNF, xvi, 115, 211
Edwards, Robert E., 14
effectively bound variable, 73
effectively not free, 85, 198
element, 68
empty domain, 35
empty set, 60, 79, 203
empty substitution, 121, 171
Enderton, Herbert B., 17, 23, 74, 103
epsilon relation, 81
equality ($=$), 64, 65, 74
error checking, 142, 167, 178
errors in proofs, 22, 24
essential hypothesis, 43, 125
Euclidean geometry, xiv, 23, 25
existential quantifier ($\exists$), 71, 76, 201
    restricted, 79
existential uniqueness quantifier ($\exists!$), 76
expression, 115, 119
    in a formal system, 191
Extended Backus–Naur Form, xvi, 115, 211
extended frame, 131
extended language, 117, 139

family, 68
Feferman, Solomon, xv
Fenton, Scott, 92
Fermat's Last Theorem, 13, 22, 25
file inclusion, 113, 148
file names
    Macintosh, 45, 149
    Unix, 45, 157, 162
finitary proof, 32, 33
finite n-termed sequence, 190
finite induction, 88
first-order logic, 11, 20, 26, 34, 60
    completeness, 11
floating hypothesis, 43, 125
formal logic, 18, 19, 128
formal proof, xiii, 12, 13, 15, 19, 20, 35, 38, 40, 41, 46, 48, 51, 52
formal system, ix, xi, 2, 19, 32, 171, 193, 207
formalism, 33
forms
    closed, 97

deduction, 97, 99
inference, 97
foundations of mathematics, 17
founded relation, 81
four-color theorem, 21, 23
frame, 129
frames and scoping statements, 134
free logic, 35
free variable, 35, 65–67, 120, 126, 198
free vs. bound variable, 120
Frege, Gottlob, 18
function, 23, 83
in predicate calculus, 68
function value, 84

Gödel's incompleteness theorem, xi, 6, 15, 18
gaps in proofs, 19
Ghilbert, xv, 28
global statement, 133
Goodstein, R. L., 152
grave accent (‘), 140, 141
Grothendieck, Alexander, xv
group theory, 3
Guillen, Michael, 5

Haken, W., 23
Halmos, Paul, 20
Hamilton, Alan G., 67, 105, 198
Hardy, G. H., 112
Harrison, John, 20, 25
Herrlich, Horst, 34
hierarchy, 13, 15, 19, 26, 108
higher-order logic, 20, 32
Higher-Order Logic (HOL) Explorer, 32
Higher-order Logic database (`hol.mm`), 32
Hilbert, David, 15, 16, 23, 32, 33
Hindley, J. Roger, 34
Hofstadter, Douglas R., 32, 40, 209
HOL, 27
HOL light, 27
HTML, xii, 140
HTML generation, 164
Hume, David, 22
Huntington, E. V., 26
hypotheses, 97
hypothesis, 43, 114, 125
hypothesis association, 138
hypothesis label, 44, 135

iff, 81, 183
image, 83
implication (→), 38, 62
implicit axiom, 35
included file, 113, 148
individual metavariable, 196
individual variable, 64
Indrzejczak, Andrzej, 98
inference, 126, 195
inference form, 97
inference rule, 39
infinite set, 61
infinity, 23
infix connective, 76
informal proof, 13, 15
integer, 60, 61
intersection, 79, 80
intuitionism, 17, 32, 152
intuitionistic logic database (`iset.mm`), 32
Intuitionistic Logic Explorer, 32
Isabelle, 27

Jubin, Benoît, xvii

Kalman, J. A., 34
Kempe, A. B., 23
keyword, 42, 112, 116, 117
Kline, Morris, 18, 33
Knill, Oliver, 29
Koch, K., 23
Kronecker, Leopold, 16
Kuratowski, Kazimierz, 80

label, 43, 46, 49, 112, 116, 118, 123, 125, 127, 128, 135, 138
label declaration, 118
label mode, 141

label reference, 118, 135
label sequence, 118, 135
labels in `set.mm`, 108
Landau, Edmund, 5
LaTeX, xii, 38, 104, 118, 140, 165, 176, 177
  characters per line, 161
LaTeX definitions, 146
LCF, 27
Leśniewski, S., 151
Leahy, Thomas Brendan, 51
Leblanc, Hugues, 35
Lejewski, Czeslaw, 151
Lemmon-style proof, 47
length of a sequence (| |), 190
Levien, Raph, xiv, 28
limit ordinal, 82
link to bibliographical reference, 142
local label, 110
local variable, 134
logic, 2
logical equivalence (↔), 71, 74, 75, 201
logical hypothesis, 43, 125
  in a formal system, 192
logical OR (∨), 75
logical AND (∧), 64, 71, 75, 106
logical OR (∨), 64
Lounesto, Pertti, 24

Mac Lane, Saunders, xv
machine learning, xi
Macintosh file names, 45, 149
MACSYMA, 3
mandatory `$d` statement, 138
mandatory disjoint-variable restriction, 114
  in a formal system, 192
mandatory hypothesis, 43, 46, 57, 114, 130, 135, 138, 139, 149
  in a formal system, 192
mandatory variable, 114, 129
mandatory variable-type hypothesis
  in a formal system, 192
Maple, 3, 25
mapping, 23
Margaris, Angelo, 95
markup notation, 139, 143
math mode, 140, 141
math symbol, 42, 43, 48, 49, 64, 112, 116–119, 133
Mathematica, 3, 24
Mathematica and proofs, 24
mathematical induction, 88
Mathias, Adrian R. D., 15
Megill, Norman, ix, 17, 27, 34, 35, 66, 67, 125, 199
member, 68
Mendelson, Elliot, 38, 40
Meredith, C. A., 34, 62
metalanguage, 2, 17, 32
metalogic, 152
metalogical completeness, 35, 199, 200
Metamath, ix–xii, xiv, 2–4, 15–17, 19, 22, 24, 26, 31–34, 38, 39, 41, 42, 44, 45, 49, 50, 59, 62–64, 66–69, 102, 111, 116–119, 123, 126, 128, 139, 141, 148–150
  as a formal system, 189
  bugs, 159
  commands, 157
  installation, 37
  limitations of version 0.177, 140, 144, 170, 178
  memory limits, 162
  memory usage, 163
  representation of numbers, 3
  self-description, 20
  specification, 112
  using as a math editor, 141
Metamath Language EBNF, 211
Metamath Proof Explorer, xi, xvi, 16, 30–32, 127, 154, 155
metamathematics, 33
metatheorem, 12, 35
metavariable, 38, 41, 42, 64, 119, 121
Millay, Edna, 16

MIU-system, 32, 53, 171, 207
Miyaoka, Yoichi, 23
Mizar, xv, 28
mmj2, xi, 154, 155
modal logic, 17, 32
model theory, 32
modus ponens, 39, 40, 63, 70, 94, 98
Monaco font, 37
Monk, J. Donald, 35
monospaced font, 37
Munkres, James R., 3, 189

natural deduction, 98, 101
natural number, 61, 72, 82, 89, 154
negation (¬), 62
Nemesszeghy, E. Z., 151
nested block, 133
nesting level, 133
neural networks, xi
New Foundations database (`nf.mm`), 32
New Foundations Explorer, 32
non-scoping statement, 133
non-trivial theory, 120
normal proof, 44, 56, 149
null set, 79
number theory, 2, 38

object, 68
object language, 17
Octave, 3
omega (ω), 61, 72, 82, 89
one-to-one function, 84
onto function, 84
operating system command, 161
operation, 84, 90
operator precedence, 39
optional disjoint-variable restriction, 131
optional hypothesis, 105, 131
optional variable, 105, 131
ordered pair, 80
ordinal addition, 153, 154
ordinal number, 82
ordinal predicate, 82
OTTER, 27
outermost block, 133, 149

pair, 80
parsing Metamath, 112
Pasch's axiom, xiv, 23
Pavičić, M., 17
Peano's postulates, 72, 88
Penrose, Roger, 25
Perelman, Grigory, 24
Peterson, Jeremy George, 34
Pierce's axiom, 141
plain text, 37
Poincaré conjecture, 24
Polish notation, 76
pop, 44, 135
postfix connective, 76
power class, 80
power set, 80
pre-statement
  in a formal system, 192
predicate calculus, 60, 64, 196
prefix connective, 76
Principia Mathematica, 33
printable character, 116
printers, 37
proof, 24, 39, 118, 135
  compressed, 44, 56, 115, 149, 150, 187
  Lemmon-style, 47
  Metamath, 115
  Metamath, description of, 135
  normal, 44, 56, 149
  tree-style, 47, 138
Proof Assistant, xi, 50–52, 55, 57, 139, 150, 159, 168, 170, 171, 173–175
proof length, 15, 21, 108
proof scheme, 39
proof step, 41
proof theory, 32
proper class, 77, 202
proper definition, 129, 151
proper substitution, 35, 66, 67, 76, 120, 198

propositional calculus, 34, 60, 62, 194
provable assertion, 43, 127
provable statement
in a formal system, 193
prover9, 27
Purinton, Josh, xiv, 193
push, 44, 135

QED project, 29
qualifying expression, 73
quantifier theory, 60
quantum logic, 3, 17, 32
quantum mechanics, 3
Quine, Willard Van Orman, 17, 74, 77, 152, 202

Rêgo, Eduardo, 24
range, 83
rational number, 61
real and complex numbers, 3, 31
axioms for, 89
real number, 61
recursion operator, 153
recursive definition, 133, 153, 154
redeclaration of symbols, 43, 114, 134
reduct
in a formal system, 192
reflection principle, 20
relation, 82
restriction, 83
reverse Polish notation (RPN), 39, 135
Robbins algebra, 26
Robbins, Herbert, 26
Robinson's resolution principle, 26, 34
Robinson, T. Thacher, 152
Rourke, Colin, 24
RPN order, 46, 130, 188
RPN stack, 44, 115, 135, 139
Rucker, Rudy, 6, 40, 59
rule, ix, 17, 34, 40, 63, 128
rule of generalization, 65, 71
Russell's paradox, 9, 10, 18, 87
Russell, Bertrand, 15, 33, 111

Schmidt, Eric, 92
Schröder–Bernstein theorem, 23
scope, 105, 120, 133
scoping statement, 113, 132
sentential logic, 60
set, 23, 68
set difference, 79
set intersection, 60
set theory, 2, 19, 31, 60, 68
set theory database (`set.mm`, 32
set theory database (`set.mm`), xi, xvi, 16, 27, 30–32, 34, 35, 37, 62, 64, 67, 70, 85, 94, 102, 127, 144, 152, 154, 155, 202, 205
Set theory without distinct variable provisos, 125
set union, 60
Shoenfield, Joseph R., 33
simple declaration, 119
simple infinite sequence, 190
simple metatheorem, 199
singleton, 80
Solovay, Robert, xv
Solow, Daniel, 8
source buffer, 50, 167, 175
source file, 116, 117, 148, 163
special characters, 112
stack, 44, 115, 135, 139
Standard Deduction Theorem, 94, 101
Stark, Harold M, 23
statement
in a formal system, 192
stylized epsilon ($\in$), 61, 64
subclass, 79
subset, 68, 79
substitution
implicit, 86
proper, 35, 66, 67, 76, 120, 198
variable, xii, 34, 41, 42, 44, 51, 53, 66, 105, 106, 115, 119, 135, 138, 139, 171, 191

substitution map, 115, 191
substitution theorem, 11
successor, 82, 88, 153
Swart, E. R., 21
Syllogism, 87
symbol, 116
    in a formal system, 190
syntax rules, 19, 38
Szpiro, George, 24

Takeuti, Gaisi, 74, 77, 202
Tarski, Alfred, 25, 26, 35, 66, 190
tautology, 63
temporary variable, 52, 170, 173
term, 38, 41, 190
text editor, 37
theorem, ix, 19, 24, 32, 34, 38–40, 42, 51, 59, 61, 63, 65, 126, 151
theorem scheme, 39
tilde (~), 141
token, 42, 49, 50, 112, 116–119, 139–141, 148, 162
topology, 3
transfinite recursion, 88
transitive class, 80
transitive set, 80
tree-style proof, 47, 138
trusting computers, 21
truth table, 63
turnstile ($\vdash$), 42, 85, 118, 126
Tymoczko, Thomas, 17, 21
type, 119, 126
typecode, 126
typesetting comment, 118, 140, 141, 144, 183

Ulam, Stanislaw, 22
unification, 34, 51, 139
    ambiguous, 53, 139, 171, 175
union, 79, 80, 203
universal class (V), 79
universal quantifier ($\forall$), 64
    restricted, 78
universe of a formal system, 193
Unix file names, 45, 157, 162
unordered pair, 80, 203
unordered triple, 80

variable, xii, 42, 44, 119, 121, 133
    in a formal system, 190
    in ordinary mathematics, 121
    in predicate calculus, 60, 64
    in set theory, 61, 68
    Metamath, 113
variable declaration, 42, 119
variable substitution, xii, 34, 41, 42, 44, 51, 53, 66, 105, 106, 115, 119, 135, 138, 139, 171, 191
variable type, 119, 126
variable-type hypothesis, 43, 125
    in a formal system, 192
Venn diagram, 68

W3C, 211
Wang, Hao, 14
Weak Deduction Theorem, 95, 101
weak logic, 17, 20, 32
well-formed formula (wff), ix, 19, 20, 38, 41, 62–65, 69, 102, 136, 151, 209
well-ordering, 81
Wen-tsün, Wu, 25
Whalen, Daniel, xi
Wheeler, David A., ix, xvi, 29, 51
white space, 112, 116, 139, 141, 150
Whitehead, Alfred North, 4, 33
Wiedijk, Freek, 29
Wiles, Andrew, 23
Williams, Anthony, xv
Word (Microsoft), 37
word processor, 37
Wos, Larry, 26, 27

Zermelo–Fraenkel set theory, 9, 32, 35, 60, 68
ZFC, 60
ZFC set theory, 69, 202