TEXTS AND MONOGRAPHS IN COMPUTER SCIENCE

A PRACTICAL THEORY OF PROGRAMMING

Eric C.R. Hehner

Springer-Verlag

A Practical Theory of Programming

Eric C.R. Hehner

Springer Science+Business Media, LLC

Eric C.R. Hehner
Department of Computer Science
University of Toronto
Toronto, Ontario M5S 1A4
Canada

Series Editors:

David Gries
Department of Computer Science
Cornell University
Upson Hall
Ithaca, NY 14853-7501
USA

Fred B. Schneider
Department of Computer Science
Cornell University
Upson Hall
Ithaca, NY 14853-7501
USA

With ten illustrations.

Library of Congress Cataloging-in-Publication Data
Hehner, Eric C. R.
A practical theory of programming / Eric C.R. Hehner.
p. cm. — (Texts and monographs in computer science)
Includes bibliographical references and index.
ISBN 978-1-4612-6444-6 ISBN 978-1-4419-8596-5 (eBook)
DOI 10.1007/978-1-4419-8596-5
1. Programming (Electronic computers). I. Title. II. Series.
QA76.6.H428 1993
005.1—dc20 93-5269

Printed on acid-free paper.

Camera-ready copy prepared from the author's PostScript files using MacWrite.

9 8 7 6 5 4 3 2 1

ISBN 978-1-4612-6444-6

Contents

0 Preface

Introduction

What good is a theory of programming? Who wants one? Thousands of programmers program every day without any theory. Why should they bother to learn one? The answer is the same as for any other theory. For example, why should anyone learn a theory of motion? You can move around perfectly well without one. You can throw a ball without one. Yet we think it important enough to teach a theory of motion in high school.

One answer is that a mathematical theory gives a much greater degree of precision by providing a method of calculation. It is unlikely that we could send a rocket to Jupiter without a mathematical theory of motion. And even baseball pitchers are finding that their pitch can be improved by hiring an expert who knows some theory. Similarly a lot of mundane programming can be done without the aid of a theory, but the more difficult programming is very unlikely to be done correctly without a good theory. The software industry has an overwhelming experience of buggy programs to support that statement. And even mundane programming can be improved by the use of a theory.

Another answer is that a theory provides a kind of understanding. Our ability to control and predict motion changes from an art to a science when we learn a mathematical theory. Similarly programming changes from an art to a science when we learn to understand programs in the same way we understand mathematical theorems. With a scientific outlook, we change our view of the world. We attribute less to spirits or chance, and increase our understanding of what is possible and what is not. It is a valuable part of education for anyone.

Professional engineering maintains its high reputation in our society by insisting that, to be a professional engineer, one must know and apply the relevant theories. A civil engineer must know and apply the theories of geometry and material stress. An electrical engineer must know and apply electromagnetic theory. Software engineers, to be worthy of the name, must know and apply a theory of programming.

The subject of this book sometimes goes by the name "programming methodology", sometimes "science of programming", "logic of programming", "theory of programming", "formal methods of program development", or "verification". It concerns those aspects of programming that are amenable to mathematical proof. A good theory helps us to write precise specifications, and to design programs whose executions provably satisfy the specifications. We will be considering the state of a computation, the time of a computation, and the interactions with a computation. There are other important aspects of software design and production that are not touched by this book: the management of people, the user interface, documentation, and testing.

There are several theories of programming. The first usable theory, often called "Hoare's Logic", is still probably the most widely known. In it, a specification is a pair of predicates: a precondition and postcondition (these and all technical terms will be defined in due course). Another popular and closely related theory by Dijkstra uses the weakest precondition predicate transformer, which is a function from programs and postconditions to preconditions. Jones's Vienna Development Method has been used to advantage in some industries; in it, a specification is a pair of predicates (as in Hoare's Logic), but the second predicate is a relation. Temporal Logic is yet another formalism that introduces some special operators and quantifiers to describe some aspects of computation.

The theory in this book is simpler than any of those just mentioned. In it, a specification is just a boolean expression. Refinement is just ordinary implication. This theory is also more general than those just mentioned, applying to both terminating and nonterminating computation, to both sequential and parallel computation, to both stand-alone and interactive computation. And it includes time bounds, both for algorithm classification and for tightly constrained real-time applications.

End of Introduction

Quick Tour

All technical terms used in this book are explained in this book. Each new term that you should learn is underlined. As much as possible, the terminology is descriptive rather than honorary (notable exception: "boolean"). There are no abbreviations, acronyms, or other obscurities of language to annoy you. No specific previous mathematical knowledge or programming experience is assumed. However, the preparatory material on booleans, numbers, lists, and functions in Chapters 1, 2, and 3 is brief, and previous exposure might be helpful.

The following chart shows the dependence of each chapter on previous chapters.

$$\begin{tikzcd}
 & & & & 5 & \\
1 \arrow[r] & 2 \arrow[r] & 3 \arrow[r] & 4 \arrow[ur] \arrow[r] \arrow[dr] & 6 \arrow[r] & 7 \\
 & & & & 8 \arrow[r] & 9
\end{tikzcd}$$

Chapter 4, Program Theory, is the heart of the book. After that, chapters may be selected or omitted according to interest and the chart. The only deviations from the chart are that Chapter 9 uses variable declaration (**var**) presented in the first section of Chapter 5, and a small optional section within Chapter 9 depends on Chapter 6.

Chapter 10 consists entirely of exercises grouped according to the chapter in which the necessary theory is presented. All the exercises in the section "Program Theory" can be done according to the methods presented in Chapter 4; however, as new notations and methods are presented in later chapters, those same exercises can be redone taking advantage of the later material.

At the back of the book, Chapter 11 contains reference material. The first section, "Justifications", answers questions about earlier chapters, such as: why was this presented that way? why was this presented at all? why wasn't something else presented instead? It may be of interest to teachers and researchers who already know enough theory of programming to ask such questions. It is probably not of interest to students who are meeting formal methods for the first time. If you find yourself asking such questions, don't hesitate to consult the Justifications.

Chapter 11 also contains an index of terminology and a complete list of all laws used in the book. To a serious student of programming, these laws should become friends, on a first name basis. The final pages list all the notations used in the book. You are not expected to know these notations before reading the book; they are all explained as we come to them. You are welcome to invent new notations if you explain their use. Sometimes the choice of notation makes all the difference in our ability to solve a problem.

Transparency masters and solutions to exercises are available to course instructors from the author.

End of Quick Tour

Acknowledgments

For inspiration and guidance I thank Working Group 2.3 (Programming Methodology) of the International Federation for Information Processing, particularly Edsger Dijkstra, David Gries, John Guttag, Tony Hoare, Jim Horning, Cliff Jones, Bill McKeeman, Carroll Morgan, Greg Nelson, John Reynolds, Wlad Turski. I thank my graduate students and teaching assistants from whom I have learned so much, especially Ray Blaak, Lorene Gupta, Chris Lengauer, Andrew Malton, Theo Norvell, Alan Rosenthal. For their critical and helpful reading of the first draft I am most grateful to Wim Hesselink, Jim Horning, Jan van de Snepscheut. For good ideas I thank Ralph Back, Eike Best, Jo Ebergen, Wim Feijen, Netty van Gasteren, Nicolas Halbwachs, Gilles Kahn, Alain Martin, Martin Rem, Pierre-Yves Schobbens, Mary Shaw, Bob Tennent, Jan Tijmen Udding. For reading the draft and suggesting improvements I thank Jules Desharnais, Andy Gravell, Peter Lauer, Ali Mili, Bernhard Möller, Helmut Partsch, Jørgen Steensgaard-Madsen, Norbert Völker. I thank my class for finding errors.

End of Acknowledgments

This book is dedicated to my daughter, Amanda Susan, who at age 3 already shows signs of the organization and attention to detail of a good programmer.

End of Preface

1 Basic Theories

Boolean Theory

Boolean Theory, also known as logic, was designed as an aid to reasoning, and we will use it to reason about computation. The expressions of Boolean Theory are called boolean expressions. We divide boolean expressions into two classes; those in one class are called theorems, and those in the other are called antitheorems.

The expressions of Boolean Theory can be used to represent statements about the world; the theorems represent true statements, and the antitheorems represent false statements. That is the original application of the theory, the one it was designed for, and the one that supplies most of the terminology. Another application for which Boolean Theory is perfectly suited is digital circuit design. In that application, boolean expressions represent circuits; theorems represent circuits with high voltage output, and antitheorems represent circuits with low voltage output.

The two simplest boolean expressions are $\top$ and $\bot$. The first one, $\top$, is a theorem, and the second one, $\bot$, is an antitheorem. When Boolean Theory is being used for its original purpose, we pronounce $\top$ as “true” and $\bot$ as “false” because the former represents an arbitrary true statement and the latter represents an arbitrary false statement. When Boolean Theory is being used for digital circuit design, we pronounce $\top$ and $\bot$ as “high voltage” and “low voltage”, or as “power” and “ground”. They are sometimes called the “boolean values”; they may also be called the “nullary boolean operators”, meaning that they have no operands.

There are four unary (one operand) boolean operators, of which only one is interesting. Its symbol is $\neg$, pronounced “not”. It is a prefix operator (placed before its operand). An expression of the form $\neg x$ is called a negation. If we negate a theorem we obtain an antitheorem; if we negate an antitheorem we obtain a theorem. This is depicted by the following truth table.

	$\top$	$\bot$
$\neg$	$\bot$	$\top$

Above the horizontal line, $\top$ means that the operand is a theorem, and $\bot$ means that the operand is an antitheorem. Below the horizontal line, $\top$ means that the result is a theorem, and $\bot$ means that the result is an antitheorem.

There are sixteen binary (two operand) boolean operators. Mainly due to tradition, we will use only six of them, though they are not the only interesting ones. These operators are infix (placed between their operands). Here are the symbols and some pronunciations; for each symbol, the first pronunciation is the preferred one.

$\wedge$	"and"
$\vee$	"or"
$\Rightarrow$	"implies", "is as strong as", "is equal to or stronger than"
$\Leftarrow$	"follows from", "is implied by", "is as weak as", "is weaker than or equal to"
$=$	"equals", "if and only if"
$\neq$	"differs from", "is unequal to", "exclusive or", "boolean plus"

An expression of the form $x \wedge y$ is called a conjunction, and the operands x and y are called conjuncts. An expression of the form $x \vee y$ is called a disjunction, and the operands are called disjuncts. An expression of the form $x \Rightarrow y$ is called an implication, x is called the antecedent, and y is called the consequent. An expression of the form $x \Leftarrow y$ is also called an implication, but now x is the consequent and y is the antecedent. An expression of the form $x=y$ is called an equation, and the operands are called the left side and the right side. An expression of the form $x \neq y$ is called an unequation, and again the operands are called the left side and the right side.

The following truth table shows how the classification of boolean expressions formed with binary operators can be obtained from the classification of the operands. Above the horizontal line, the pair $\top\top$ means that both operands are theorems; the pair $\top\bot$ means that the left operand is a theorem and the right operand is an antitheorem; and so on. Below the horizontal line, $\top$ means that the result is a theorem, and $\bot$ means that the result is an antitheorem.

	$\top\top$	$\top\bot$	$\bot\top$	$\bot\bot$
$\wedge$	$\top$	$\bot$	$\bot$	$\bot$
$\vee$	$\top$	$\top$	$\top$	$\bot$
$\Rightarrow$	$\top$	$\bot$	$\top$	$\top$
$\Leftarrow$	$\top$	$\top$	$\bot$	$\top$
$=$	$\top$	$\bot$	$\bot$	$\top$
$\neq$	$\bot$	$\top$	$\top$	$\bot$

Infix operators make some expressions ambiguous. For example, $\bot \wedge \top \vee \top$ might be read as the conjunction $\bot \wedge \top$, which is an antitheorem, disjoined with $\top$, resulting in a theorem. Or it might be read as $\bot$ conjoined with the disjunction $\top \vee \top$, resulting in an antitheorem. To say which is meant, we can use parentheses: either $(\bot \wedge \top) \vee \top$ or $\bot \wedge (\top \vee \top)$. To prevent a clutter of parentheses, we employ a table of precedence levels, listed on the final page of the book. In the table, $\wedge$ can be found on level 9, and $\vee$ on level 10; that means, in the absence of parentheses, apply $\wedge$ before $\vee$. The example $\bot \wedge \top \vee \top$ is therefore a theorem.

Each of the operators $= \Rightarrow \Leftarrow$ appears twice in the precedence table. The large versions $= \Longrightarrow \Longleftarrow$ on level 14 are applied after all other operators. Except for precedence, the small versions and large versions of these operators are identical. Used with restraint, these duplicate operators can sometimes improve readability by reducing the parenthesis clutter still further. But a word of caution: a few well-chosen parentheses, even if they are unnecessary according to precedence, can help us see structure. Judgment is required.

There are 256 ternary (three operand) operators, of which we show only one. It is called conditional composition, and written **if** x **then** y **else** z . Here is its truth table.

	$\top\top\top$	$\top\top\bot$	$\top\bot\top$	$\top\bot\bot$	$\bot\top\top$	$\bot\top\bot$	$\bot\bot\top$	$\bot\bot\bot$
if then else	$\top$	$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\top$	$\bot$

For every natural number n , there are 2^{2^n} operators of n operands, but we now have quite enough.

When we stated earlier that a conjunction is an expression of the form $x \wedge y$, we were using $x \wedge y$ to stand for all expressions obtained by replacing the variables x and y with arbitrary boolean expressions. For example, we might replace x with $(\bot \Rightarrow \neg(\bot \vee \top))$ and replace y with $(\bot \vee \top)$ to obtain the conjunction

$$(\bot \Rightarrow \neg(\bot \vee \top)) \wedge (\bot \vee \top)$$

Replacing a variable with an expression is called substitution or instantiation. With the understanding that variables are there to be replaced, we admit variables into our expressions, being careful of the following three points.

• We sometimes have to insert parentheses around expressions that are replacing variables in order to maintain the precedence of operators. In the example of the preceding paragraph, we replaced a conjunct x with an implication $\bot \Rightarrow \neg(\bot \vee \top)$; since conjunction comes before implication in the precedence table, we had to enclose the implication in parentheses. We also replaced a conjunct y with a disjunction $\bot \vee \top$, so we had to enclose the disjunction in parentheses.

• When the same variable occurs more than once in an expression, it must be replaced by the same expression at each occurrence. From $x \wedge x$ we can obtain $\top \wedge \top$, but not $\top \wedge \bot$.

• We are free to replace different variables by the same expression. From $x \wedge y$ we can obtain $\top \wedge \top$.

As we present other theories, we will introduce new boolean expressions that make use of the expressions of those theories, and classify the new boolean expressions. For example, when we present Number Theory we will introduce the number expressions $1+1$ and 2 , and the boolean expression $1+1=2$, and we will classify it as a theorem. We never intend to classify a boolean expression as both a theorem and an antitheorem. A statement about the world cannot be both true and (in the same sense) false; a circuit's output cannot be both high and low voltage. If, by accident, we do classify a boolean expression both ways, we have made a serious error. But it is perfectly legitimate to leave a boolean expression unclassified. For example, $0/0=5$ will be neither a theorem nor an antitheorem. An unclassified boolean expression may correspond to a statement whose truth or falsity we do not know or do not care about, or to a circuit whose output we cannot predict. A theory is called consistent if no boolean expression is both a theorem and an antitheorem, and inconsistent if some boolean expression is both a theorem and an antitheorem. A theory is called complete if every fully instantiated boolean expression is either a theorem or an antitheorem, and incomplete if some fully instantiated boolean expression is neither a theorem nor an antitheorem.

Axioms and Proof Rules

To prove that a boolean expression is a theorem, or to prove that it is an antitheorem, we must follow the five rules of proof. We state them first, then discuss them after.

Axiom Rule	If a boolean expression is an axiom, then it is a theorem. If a boolean expression is an antiaxiom, then it is an antitheorem.
Evaluation Rule	If all the boolean subexpressions of a boolean expression are classified, then it is classified according to the truth tables.
Completion Rule	If a boolean expression contains unclassified boolean subexpressions, and all ways of classifying them place it in the same class, then it is in that class.
Consistency Rule	If a classified boolean expression contains boolean subexpressions, and at most one way of classifying them is consistent, then they are classified that way.
Instance Rule	If a boolean expression is classified, then all its instances have that same classification.

We present a theory by saying what its expressions are, and what its theorems and antitheorems are. An axiom is a boolean expression that is stated to be a theorem. An antiaxiom is similarly a boolean expression stated to be an antitheorem. The only axiom of Boolean Theory is $\top$ and the only antiaxiom is $\bot$. So, by the Axiom Rule, $\top$ is a theorem and $\bot$ is an antitheorem.

Before the invention of formal logic, the word "axiom" was used for a statement whose truth was supposed to be obvious. In modern mathematics, an axiom is part of the design and presentation of a theory. Different axioms may yield different theories, and different theories may have different applications. When we design a theory, we can choose any axioms we like, but a bad choice can result in a useless theory.

The first entry in the truth table for the binary operators does not say $\top\wedge\top = \top$. It says that the conjunction of any two theorems is a theorem. To prove that $\top\wedge\top = \top$ is a theorem requires the boolean axiom (to prove that $\top$ is a theorem), the first entry in the truth table (to prove that $\top\wedge\top$ is a theorem), and the first entry on the $=$ row of the truth table (to prove that $\top\wedge\top = \top$ is a theorem).

The boolean expression

$$\top \vee x$$

contains an unclassified boolean subexpression, so we cannot use the Evaluation Rule to tell us which class it is in. If x were a theorem, the Evaluation Rule would say that the whole expression is a theorem. If x were an antitheorem, the Evaluation Rule would again say that the whole expression is a theorem. We can therefore conclude by the Completion Rule that the whole expression is indeed a theorem. The Completion Rule also says that

$$x \vee \neg x$$

is a theorem, and when we come to Number Theory, that

$$0/0 = 5 \ \vee \ \neg\ 0/0 = 5$$

is a theorem. We do not need to know that a subexpression is unclassified to use the Completion Rule. If we are ignorant of the classification of a subexpression, and we suppose it to be unclassified, any conclusion we come to by the use of the Completion Rule will still be correct.

In a classified boolean expression, if it would be inconsistent to place a boolean subexpression in one class, then the Consistency Rule says it is in the other class. For example, suppose we know that *expression0* is a theorem, and that *expression0* ⇒ *expression1* is also a theorem. Can we determine what class *expression1* is in? If *expression1* were an antitheorem, then by the Evaluation Rule *expression0* ⇒ *expression1* would be an antitheorem, and that would be inconsistent. So, by the Consistency Rule, *expression1* is a theorem. This use of the Consistency Rule is traditionally called "detachment" or "modus ponens". As another example, if ¬*expression* is a theorem, then the Consistency Rule says that *expression* is an antitheorem.

Thanks to the negation operator and the Consistency Rule, we never need to talk about antitheorems. Instead of saying that *expression* is an antitheorem, we can say that ¬*expression* is a theorem. But a word of caution: if a theory is incomplete, it is possible that neither *expression* nor ¬*expression* is a theorem. Thus "antitheorem" is not the same as "not a theorem". Our preference for theorems over antitheorems encourages some shortcuts of speech. We sometimes state a boolean expression, such as $1+1=2$, without saying anything about it; when we do so, we mean that it is a theorem. We sometimes say we will prove something, meaning we will prove it is a theorem.

————————————————————End of Axioms and Proof Rules

With our two axioms ($\top$ and $\neg\bot$) and five proof rules we can now prove theorems. Some theorems are useful enough to be given a name and be memorized, or at least be kept in a handy list. Such a theorem is called a law. Some laws of Boolean Theory are listed at the back of the book. Laws concerning $\Leftarrow$ have not been included, but any law that uses $\Rightarrow$ can be easily rearranged into one using $\Leftarrow$. All of them can be proven using the Completion Rule, classifying the variables in all possible ways, and evaluating each way. When the number of variables is more than about 2, this kind of proof is quite inefficient. It is much better to prove new laws by making use of already proven old laws. In the next subsection we see how.

Expression and Proof Format

The precedence table on the final page of this book tells how to parse an expression in the absence of parentheses. To help the eye group the symbols properly, it is a good idea to leave space for absent parentheses. Consider the following two ways of spacing the same expression.

$a{\wedge}b \;\;\vee\;\; c$

$a \;\;\wedge\;\; b{\vee}c$

According to our rules of precedence, the parentheses belong around $a{\wedge}b$, so the first spacing is helpful and the second misleading.

An expression that is too long to fit on one line must be broken into parts. There are several reasonable ways to do it; here is one suggestion. A long expression in parentheses can be broken at its main connective, which is placed under the opening parenthesis. For example,

(*first part*
$\wedge$ *second part*)

A long expression without parentheses can be broken at its main connective, which is placed under where the opening parenthesis belongs. For example,

first part
= *second part*

Attention to format makes a big difference in our ability to understand a complex expression.

A proof may be written in the following format.

	expression0	short hint 0
=	*expression1*	short hint 1
=	*expression2*	short hint 2
=	*expression3*	short hint 3

On the left side of the page is a continuing equation. If we did not use equations in this continuing fashion, we would have to write

expression0 = *expression1*
$\wedge$ *expression1* = *expression2*
$\wedge$ *expression2* = *expression3*

We intend it to be clear that this continuing equation is a theorem. The hints on the right side of the page are used, when necessary, to help make it clear. The "short hint 0" is supposed to make it clear that *expression0* = *expression1* is a theorem. The "short hint 1" is supposed to make it clear that *expression1* = *expression2* is a theorem. And so on. If the theorem to be proven is *expression0* = *expression3* , then there is no "short hint 3", and the theorem to be proven follows from the transitivity of = . If the theorem to be proven is *expression0* , then "short hint 3" is supposed to make it clear that *expression3* is a theorem, and the theorem to be proven follows from the transitivity of = and the Consistency Rule.

Here is an example. Suppose we want to prove the first Law of Portation

$$a \land b \Rightarrow c \;=\; a \Rightarrow (b \Rightarrow c)$$

Here is a proof.

	$a \land b \Rightarrow c$	Material Implication
$=$	$\neg(a \land b) \lor c$	Duality
$=$	$\neg a \lor \neg b \lor c$	Material Implication
$=$	$a \Rightarrow \neg b \lor c$	Material Implication
$=$	$a \Rightarrow (b \Rightarrow c)$	

From the first line of the proof, we are told to use "Material Implication", which is the first of the Laws of Inclusion. This law says that an implication can be changed to a disjunction if we also negate the antecedent. Doing so, we obtain the second line of the proof. The hint now is "Duality", and we see that the third line is obtained by replacing $\neg(a \land b)$ with $\neg a \lor \neg b$ in accordance with the first of the Duality Laws. By not using parentheses on the third line, we silently use the Associative Law of disjunction, in preparation for the next step. The next hint is again "Material Implication"; this time it is used in the opposite direction, to replace the first disjunction with an implication. And once more, "Material Implication" is used to replace the remaining disjunction with an implication. Therefore, by transitivity of $=$, we conclude that the first Law of Portation is a theorem.

Here is the same proof again, but using the proof format the other way.

	$(a \land b \Rightarrow c \;=\; a \Rightarrow (b \Rightarrow c))$	Material Implication, 3 times
$=$	$(\neg(a \land b) \lor c \;=\; \neg a \lor (\neg b \lor c))$	Duality
$=$	$(\neg a \lor \neg b \lor c \;=\; \neg a \lor \neg b \lor c)$	Reflexivity of $=$

The final hint tells us that the final line is a theorem, hence each of the other lines is a theorem, and in particular, the first line is a theorem. It may be tempting to write one more line in this proof:

$= \quad \top$

so that every hint takes us from one line to the next. But once we see that we have a theorem, it is superfluous to write more. Indeed, it is always superfluous to equate a boolean expression to $\top$ just as it is to add 0 to a number expression. On the other hand, it doesn't hurt, and it makes proof the same as simplification, with the last line being the simplest expression that's equal to the first line. So we leave it as a matter of taste whether to add this line.

Sometimes it is clear enough how to get from one line to the next without a hint, and in that case no hint will be given. Hints are optional, to be used whenever they are helpful.

Sometimes a hint is too long to fit on the remainder of a line. When that is the case, the hint may be written in normal text form, between the lines of the proof. We may have

$$\begin{array}{lll} & \textit{expression0} & \text{short hint} \\ = & \textit{expression1} & \end{array}$$

and now a very long hint, written just as this is written, on as many lines as necessary, followed by

$$\begin{array}{ll} = & \textit{expression2} \end{array}$$

We cannot excuse an inadequate hint by the limited space on one line.

Our proof of the first Law of Portation was a continuing equation. A proof can also be a continuing implication, or a continuing mixture of these and other operators. As an example, here is a proof of the first Law of Confutation, which says

$$(a \Rightarrow b) \land (c \Rightarrow d) \implies a \land c \Rightarrow b \land d$$

The proof goes this way:

$$\begin{array}{lll} & a \land c \Rightarrow b \land d & \text{distribute } \Rightarrow \text{ over second } \land \\ = & (a \land c \Rightarrow b) \land (a \land c \Rightarrow d) & \text{antidistribution twice} \\ = & ((a\Rightarrow b) \lor (c\Rightarrow b)) \land ((a\Rightarrow d) \lor (c\Rightarrow d)) & \text{distribute } \land \text{ over } \lor \text{ twice} \\ = & (a\Rightarrow b)\land(a\Rightarrow d) \lor (a\Rightarrow b)\land(c\Rightarrow d) \lor (c\Rightarrow b)\land(a\Rightarrow d) \lor (c\Rightarrow b)\land(c\Rightarrow d) & \text{generalization} \\ \Longleftarrow & (a\Rightarrow b) \land (c\Rightarrow d) & \end{array}$$

From the mutual transitivity of $=$ and $\Longleftarrow$, we have proven

$$a \land c \Rightarrow b \land d \Longleftarrow (a\Rightarrow b) \land (c\Rightarrow d)$$

which can easily be rearranged to give the desired theorem.

A proof, or part of a proof, can make use of local assumptions. A proof may have the format

$$\begin{array}{ll} \textit{assumption} \Rightarrow (& \textit{expression0} \\ & = \textit{expression1} \\ & = \textit{expression2} \\ & = \textit{expression3}\ ) \end{array}$$

for example. The step $\textit{expression0} = \textit{expression1}$ can make use of the *assumption* just as though it were an axiom. So can the step $\textit{expression1} = \textit{expression2}$, and so on. Within the parentheses we have a proof; it can be any kind of proof including one that makes further local assumptions. We thus can have proofs within proofs, indenting appropriately. If the subproof is proving $\textit{expression0} = \textit{expression3}$, then the whole proof is proving

$$\textit{assumption} \Rightarrow (\textit{expression0} = \textit{expression3})$$

If the subproof is proving *expression0*, then the whole proof is proving

$$\textit{assumption} \Rightarrow \textit{expression0}$$

If the subproof is proving $\bot$, then the whole proof is proving

    assumption $\Rightarrow \bot$

which is equal to $\neg$*assumption* . This is called “proof by contradiction”.

We can also use **if then else** as a proof, or part of a proof, in a similar manner. The format is

```
if possibility
then ( first subproof
       assuming possibility
       as a local axiom )
else ( second subproof
       assuming ¬possibility
       as a local axiom )
```

If the first subproof proves *something* and the second proves *something else* , the whole proof proves

    if *possibility* **then** *something* **else** *something else*

If both subproofs prove the same thing, then by the Case Idempotent Law, so does the whole proof, and that is its most frequent use.

In this book, a proof is just a theorem, written with enough detail so that it is easily seen to be a theorem.

——End of Expression and Proof Format

Formalization

We use computers to solve problems, or to provide services, or just for fun. The desired computer behavior is usually described at first informally, in a natural language (like English), perhaps with some diagrams, perhaps with some hand gestures, rather than formally, using mathematical formulas (notations). In the end, the desired computer behavior is described formally as a program. A programmer must be able to translate informal descriptions to formal ones.

A statement in a natural language can be vague, ambiguous, or subtle, and can rely on a great deal of cultural context. This makes formalization difficult, but also necessary. We cannot possibly say how to formalize, in general; it requires a thorough knowledge of the natural language, and is always subject to argument. In this subsection we just point out a few pitfalls in the translation from English to boolean expressions.

The best translation may not be a one-for-one substitution of symbols for words. Also, the same word in different places may be translated to different symbols. And conversely, different words may be translated to the same symbol. The words “and”, “also”, “but”, “yet”, “however”, and "moreover" might all be translated as $\land$. Just putting things next to each other sometimes means $\land$. For example, "They're red, ripe, and juicy, but not sweet." becomes $red \land ripe \land juicy \land \neg sweet$.

The word "or" in English is sometimes best translated as $\lor$, and sometimes as $\ne$. For example, "They're either small or rotten." probably includes the possibility that they're both small and rotten, and should be translated as $small \lor rotten$. But "Either we eat them or we preserve them." probably excludes doing both, and is best translated as $eat \ne preserve$.

The word "if" in English is sometimes best translated as $\Rightarrow$, and sometimes as $=$. For example, "If it rains, I'll get wet." probably leaves open the possibility that I might get wet for some other reason, and should be translated as $rain \Rightarrow wet$. But "If I get wet, I'll blame you for it." probably means "if and only if", and is best translated as $wet = blame$.

End of Formalization

End of Boolean Theory

Number Theory

Number Theory, also known as arithmetic, was designed to represent quantity. In the version we present, a number expression is formed in the following ways.

a sequence of one or more decimal digits

∞	"infinity"
$+x$	"plus x "
$-x$	"minus x "
$x+y$	" x plus y "
$x-y$	" x minus y "
$x \times y$	" x times y " (when unambiguous, $\times$ may be omitted)
x / y	" x divided by y "
x^y	" x to the power y "

if a **then** x **else** y

where x and y are any number expressions, and a is any boolean expression. The infinite number expression ∞ will be essential when we talk about the execution time of programs. We also introduce several new ways of forming boolean expressions:

$x<y$	" x is less than y ", " x is smaller than y "
$x \le y$	" x is less than or equal to y ", " x is as small as y "
$x>y$	" x is greater than y ", " x is bigger than y "
$x \ge y$	" x is greater than or equal to y ", " x is as big as y "
$x=y$	" x equals y ", " x is equal to y "
$x \ne y$	" x differs from y ", " x is unequal to y "

Here are the axioms of Bunch Theory. In these axioms, x and y are elements (elementary bunches), and A , B , and C are arbitrary bunches.

$x: y \;=\; x=y$	elementary axiom
$x: A,B \;=\; x: A \vee x: B$	compound axiom
$A,A = A$	idempotence
$A,B = B,A$	symmetry
$A,(B,C) = (A,B),C$	associativity
$A‘A = A$	idempotence
$A‘B = B‘A$	symmetry
$A‘(B‘C) = (A‘B)‘C$	associativity
$A,B: C \;=\; A: C \wedge B: C$	
$A: B‘C \;=\; A: B \wedge A: C$	
$A: A,B$	generalization
$A‘B: A$	specialization
$A: A$	reflexivity
$A: B \wedge B: A \;=\; A=B$	antisymmetry
$A: B \wedge B: C \Rightarrow A: C$	transitivity
$¢x = 1$	
$¢(A, B) + ¢(A‘B) = ¢A + ¢B$	
$\neg\, x: A \Rightarrow ¢(A‘x) = 0$	
$A: B \Rightarrow ¢A \le ¢B$	

From these axioms, many laws can be proven. Among them:

$A,(A‘B) = A$	absorption
$A‘(A,B) = A$	absorption
$A: B \Rightarrow C,A: C,B$	monotonicity
$A: B \Rightarrow C‘A: C‘B$	monotonicity
$A: B \;=\; A,B = B \;=\; A = A‘B$	inclusion
$A,(B,C) = (A,B),(A,C)$	distributivity
$A,(B‘C) = (A,B)‘(A,C)$	distributivity
$A‘(B,C) = (A‘B), (A‘C)$	distributivity
$A‘(B‘C) = (A‘B)‘(A‘C)$	distributivity
$A: B \wedge C: D \Rightarrow A,C: B,D$	confutation
$A: B \wedge C: D \Rightarrow A‘C: B‘D$	confutation

Here are several bunches that we will find useful:

null		the empty bunch
bool	$= \top, \bot$	the booleans
nat	$= 0, 1, 2, ...$	the natural numbers
int	$= ..., -2, -1, 0, 1, 2, ...$	the integer numbers

rat	$= 0, -1, 2/3, ...$	the rational numbers
$xnat$	$= nat, \infty$	the extended naturals
$xint$	$= -\infty, int, \infty$	the extended integers
$xrat$	$= -\infty, rat, \infty$	the extended rationals
$char$	$= ..., \grave{}a, \grave{}A, ...$	the characters

We define the empty bunch, $null$, with the axioms

$$null: A$$

$$\cent A = 0 \;\;=\;\; A = null$$

This gives us three more laws:

$A, null = A$	identity
$A \text{ ‘ } null = null$	base
$\cent\, null = 0$	

The bunch $bool$ is defined by the axiom $bool = \top, \bot$. The next six of these bunches (the number bunches) are infinite, and we have not yet defined them formally; the three dots are saying "guess what goes here". We define them formally in the chapter "Recursive Definition"; until then, we rely on your experience. In some books, particularly older ones, the natural numbers start at 1 ; we will use the term with its current and more useful meaning, starting at 0 . The bunch $char$ may or may not be infinite; we do not care to define it.

We also use the notation

$x,..y$ "x to y" (not "x through y")

where x and y are extended integers and $x \le y$. Its axiom is

$$i: x,..y \;\;=\;\; x \le i < y$$

The notation ,.. is asymmetric as a reminder that the left end of the interval is included and the right end is excluded. For example,

$$0,..\infty = nat$$

$$5,..5 = null$$

$$\cent(x,..y) = y - x$$

The operators , ‘ ¢ : = ≠ **if then else** apply to bunch operands according to the axioms already presented. Other operators can be applied to bunches with the understanding that they apply to the elements of the bunch. In other words, they distribute over bunch union. For example,

$$-null = null$$

$$-(A, B) = -A, -B$$

$$(A, B)+null = null$$

$$(A, B)+C = A+C, B+C$$

$$(A, B)+(C, D) = A+C, A+D, B+C, B+D$$

This makes it easy to express the positive naturals $(nat+1)$, the even naturals $(nat\times 2)$, the squares (nat^2) , the powers of two (2^{nat}) , and many other things. We will make great use of this distribution. (The operators that distribute over bunch union are listed on the final page.)

End of Bunch Theory

Except for a few brief mentions, we do not use sets in this book. After bunches, it is only a small step to define sets, and since they are so well-known, they may help to place the other theories in this chapter in perspective.

Set Theory

Let A be any bunch (anything). Then

$\{A\}$	"set containing A"

is a set. Thus $\{null\}$ is the empty set, and the set containing the first three natural numbers is expressed as $\{0, 1, 2\}$ or as $\{0,..3\}$. All sets are elements; not all bunches are elements; that is the difference between sets and bunches. We can form the bunch $1, \{3, 7\}$ consisting of two elements, and from it the set $\{1, \{3, 7\}\}$ containing two elements, and in that way we build a structure of nested sets.

The powerset operator ${}_2$ is a unary prefix operator that takes a set as operand and yields a set of sets as result. Here is an example.

$${}_2\{0, 1\} = \{\{null\}, \{0\}, \{1\}, \{0, 1\}\}$$

The inverse of set formation is also useful. If S is any set, then

$\sim S$	"contents of S"

is its contents. For example,

$$\sim\{0, 1\} = 0, 1$$

We "promote" the bunch operators to obtain the set operators $\$ \in \subseteq \cup \cap =$. Here are the axioms.

$\{A\} \neq A$	
$\sim\{A\} = A$	"contents"
$\$\{A\} = ¢A$	"size", "cardinality"
$A \in \{B\} \;=\; A: B$	"elements"
$\{A\} \subseteq \{B\} \;=\; A: B$	"subset"
$\{A\} \in {}_2\{B\} \;=\; A: B$	"powerset"
$\{A\} \cup \{B\} = \{A, B\}$	"union"
$\{A\} \cap \{B\} = \{A \text{ ‘ } B\}$	"intersection"
$\{A\} = \{B\} \;=\; A = B$	"equation"

End of Set Theory

Just as bunches and sets are, respectively, unpackaged and packaged collections, so strings and lists are, respectively, unpackaged and packaged sequences. There are sets of sets, and lists of lists, but there are neither bunches of bunches nor strings of strings.

String Theory

The simplest string is

nil the empty string

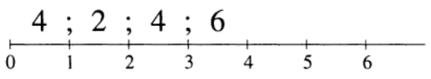
Any number, character, boolean, set (and later also list and function) is a one-item string, or item. For example, the number 2 is a one-item string, or item. Strings are catenated (joined) together by semicolons to make longer strings. For example,

4; 2; 4; 6

is a four-item string. The length of a string is the number of items, and is obtained by the # operator.

#(4; 2; 4; 6) = 4

We can measure a string by placing it along a string-measuring ruler, as in the following picture.



Each of the numbers under the ruler is called an index. When we are considering the items in a string from beginning to end, and we say we are at index n, it is clear which items have been considered and which remain because we draw the items between the indexes. (If we were to draw an item at an index, saying we are at index n would leave doubt as to whether the item at that index has been considered.)

The picture saves one confusion, but causes another: we must refer to the items by index, and two indexes are equally near each item. We adopt the convention that most often avoids the need for a "+1" or "–1" in our expressions: the index of an item is the number of items that precede it. In other words, indexing is from 0. Your life begins at year 0, a highway begins at mile 0, and so on. An index is not an arbitrary label, but a measure of how much has gone before. We refer to the items in a string as "item 0", "item 1", "item 2", and so on; we never say "the third item" due to the possible confusion between item 2 and item 3. When we are at index n, then n items have been considered, and item n will be considered next.

We obtain an item of a string by subscripting. For example,

$(3; 5; 7; 9)_2 = 7$

In general, S_n is item n of string S. We can even pick out a whole string of items, as in the following example.

$(3; 5; 7; 9)_{2;\,1;\,2} = 7; 5; 7$

If n is a natural and S is a string, then $n*S$ means n copies of S catenated together.

$$3*(0;\,1) \;=\; 0;\,1;\,0;\,1;\,0;\,1$$

Without any left operand, $*S$ means all strings formed by catenating any number of copies of S .

$$*(0;\,1) \;=\; nil\,,\;\; 0;1\,,\;\; 0;1;0;1\,,\;\ldots$$

Strings can be compared for equality and order. To be equal, strings must be of equal length, and have equal items at each index. The order of two strings is determined by the items at the first index where they differ. For example,

$$3;\,6;\,4;\,7 \;<\; 3;\,7;\,2$$

If there is no index where they differ, the shorter string comes before the longer one.

$$3;\,6;\,4 \;<\; 3;\,6;\,4;\,7$$

This ordering is known as lexicographic order; it is the ordering used in dictionaries.

Here is the syntax of strings. If i is an item, S and T are strings, and n is a natural number, then

nil	the empty string
i	an item
$S;T$	" S catenate T "
S_T	" S sub T "
$n*S$	" n copies of S "

are strings,

$*S$	"copies of S "

is a bunch of strings, and

$\#S$	"length of S "

is a natural number. The order operators $<\le>\ge$ apply to strings.

Here are the axioms of String Theory. In these axioms, S , T , and U are strings, i and j are items, and n is a natural number.

$nil;\,S \;=\; S;\,nil \;=\; S$	identity
$S;\,(T;\,U) \;=\; (S;\,T);\,U$	associativity
$\#nil \;=\; 0$	
$\#i \;=\; 1$	
$\#(S;\,T) \;=\; \#S + \#T$	
$S_{nil} \;=\; nil$	
$(S;\,i;\,T)_{\#S} \;=\; i$	
$S_{T;\,U} \;=\; S_T;\,S_U$	
$S_{(T_U)} \;=\; (S_T)_U$	

Let $L = [10;..15]$. Then

$2\to L3 \mid 3\to L2 \mid L = [10; 11; 13; 12; 14]$

The order operators $< \le > \ge$ apply to lists; the order is lexicographic, just like string order.

Here are the axioms. Let S and T be strings, let n be a natural number, and let i and j be items.

$\#[S] = \#S$	length
$[S]^{+}[T] = [S; T]$	catenation
$[S]\, n = S_n$	indexing
$[S]\,[T] = [S_T]$	composition
$(\#S) \to i \mid [S; j; T] = [S; i; T]$	modification
$[S] = [T] \quad = \quad S = T$	equation
$[S] < [T] \quad = \quad S < T$	order
$[S; T]\colon [S]$	inclusion

We can now prove a variety of theorems, such as for lists L , M , N , and natural n that

$(L\,M)\,n = L\,(M\,n)$	
$(L\,M)\,N = L\,(M\,N)$	associativity
$L\,(M^{+}N) = L\,M\,{}^{+}\,L\,N$	distributivity

(The proofs assume that each list has the form $[S]$.)

When a list is indexed by a list, we get a list of results. More generally, the index can be any structure, and the result will have the same structure.

$L\,null = null$

$L\,(A, B) = L\,A, L\,B$

$L\,\{A\} = \{L\,A\}$

$L\,nil = nil$

$L\,(S; T) = L\,S; L\,T$

$L\,[S] = [L\,S]$

Here is a fancy example. Let $L = [10; 11; 12]$. Then

$L\,[0, \{1, [2; 1]; 0\}] = [L\,0, \{L\,1, [L\,2; L\,1]; L\,0\}] = [10, \{11, [12; 11]; 10\}]$

The text notation is an alternative way of writing a list of characters. A text begins with a double-quote, continues with any natural number of characters (but a double-quote must be repeated), and concludes with a double-quote. Here is a text of length 15 .

"Don't say ""no""." = [`D; `o; `n; `'; `t; ` ; `s; `a; `y; ` ; `"; `n; `o; `"; `.]

Composing a text with a list of indexes we obtain a subtext. For example,

"abcdefghij" [3;..6] = "def"

Here is a self-describing expression (self-reproducing automaton).

"""[0;0;2*(0;..17)]"[0;0;2*(0;..17)]

Multidimensional Structures

Lists can be items in a list. For example, let

$$A = [\ [6; 3; 7; 0]\ ; \\ [4; 9; 2; 5]\ ; \\ [1; 5; 8; 3]\ ]$$

Then A is a 2-dimensional array, or more particularly, a 3×4 array. Formally, A: $[3*[4*nat]]$. Indexing A with one index gives a list

$$A\ 1 = [4; 9; 2; 5]$$

which can then be indexed again to give a number.

$$A\ 1\ 2 = 2$$

Warning: The notations $A(1,2)$ and $A[1,2]$ are used in several programming languages to index a 2-dimensional array. But in this book,

$$A\ (1, 2) = A\ 1, A\ 2 = [4; 9; 2; 5], [1; 5; 8; 3]$$
$$A\ [1, 2] = [A\ 1, A\ 2] = [\ [4; 9; 2; 5], [1; 5; 8; 3]\ ]$$

We have just seen a rectangular array, a very regular structure, which requires two indexes to give a number. Lists of lists can also be quite irregular in shape, not just by containing lists of different lengths, but in dimensionality. For example, let

$$B = [\ [2; 3]; 4; [5; [6; 7]\ ]\ ]$$

Now $B\ 0\ 0 = 2$ and $B\ 1 = 4$, and $B\ 1\ 1$ is undefined. The number of indexes needed to obtain a number varies. We can regain some regularity in the following way. Let L be a list, let n be an index, and let S and T be strings of indexes. Then

$$L@nil = L$$
$$L@n = L\,n$$
$$L@(S; T) = L@S@T$$

Now we can always "index" with a single string, called a pointer, obtaining the same result as indexing by the sequence of items in the string. In the example list,

$$B@(2; 1; 0) = B\ 2\ 1\ 0 = 6$$

We generalize the notation $S \to i \mid L$ to allow S to be a string of indexes. The axioms are

$$nil \to i \mid L = i$$
$$(S;T) \to i \mid L = S \to (T \to i \mid L@S) \mid L$$

Thus $S \to i \mid L$ is a list like L except that S points to item i . For example,

$$(0;1) \to 6 \mid [\ [0; 1; 2]\ ; \\ [3; 4; 5]\ ] = [\ [0; 6; 2]\ ; \\ [3; 4; 5]\ ]$$

End of Multidimensional Structures

End of List Theory

End of Basic Data Structures

3 Function Theory

We are always allowed to invent new syntax if we explain the rules for its use. A ready source of new syntax is names (identifiers), and the rules for their use are most easily given by some axioms. We might say something like "let $pi = 3.14$ ", meaning that we introduce the name pi and the axiom $pi = 3.14$. A similar example is "let x be an element such that x: *nat* ", or more briefly, "let x: *nat* ". We call pi a constant because the axiom constrains it to one value, and we call x a variable because the axiom allows it many possible values.

Here is an intermediate example: let p and q be numbers such that

p, q: *nat*

$p \times q = p+q$

From the axioms of Number Theory, plus *nat* induction (Chapter 6), plus these two axioms, we can prove $p=q=0 \vee p=q=2$. What are p and q ? They are not very variable, yet they are not completely constant. A theory is defined when we know what expressions we can write and how to prove theorems. Beyond that, it does not matter what we call anything. It no more matters whether p and q are called variables or constants than whether $<$ is called "less than" or "smaller than".

Usually when we introduce names and axioms we want them for some local purpose; we do not want to shout them to the world. The reader is supposed to understand their scope, the region where they apply, and not use them beyond it. Though the names and axioms are formal (expressions in our formalism), they were introduced informally in our examples (by an English sentence beginning with "let"), and the scope of informally introduced names and axioms is not always clear. In this chapter we present a formal notation for introducing a local name together with a local axiom to say what its possible values are.

Functions

Let v be a name, let D be a bunch of items (possibly using previously introduced names but not using v), and let b be any expression (possibly using previously introduced names and possibly using v). Then

$\lambda v: D \cdot b$ "map v in D to b ", "local v in D maps to b "

is a function of variable v with domain D and body b . The inclusion $v: D$ is a local axiom within the body b . For example,

λn: *nat*$\cdot$ $n+1$

is the successor function on the natural numbers. Here is a picture of it.