THE LOGIC OF PROGRAMMING

ERIC C. R. HEHNER

THE LOGIC OF PROGRAMMING

ERIC C. R. HEHNER

Englewood Cliffs, New Jersey London New Delhi Rio de Janeiro
Singapore Sydney Tokyo Toronto Wellington

CONTENTS

PREFACE

A student who wished to learn computer programming asked me for advice on which of two courses to take. One was more mathematical than the other, and he did not like Mathematics. Since he was a pianist, I asked him which course one should take to become a composer, if one does not like music.

This book introduces the subject of computer programming as a rigorous, mathematical discipline. Its level of exposition makes it suitable for an advanced undergraduate university course, or introductory graduate course. As much as possible, it is self-contained, relying on mathematical ability but not on specific mathematical knowledge past the secondary school level. People whose interest is more in the practical aspects of programming than in the theoretical aspects will want to skip lightly over Chapters 2, 5, 11, and 12, and to concentrate on Chapters 6 through 10. Of course, those with the reverse interest will want to make the reverse concentration. Sprinkled throughout the book are paragraphs clearly marked "Aside". For beginners to the subject of programming, the text is complete without these paragraphs, and they can safely be ignored. They are intended for people who already have programming experience, to relate the material of this book to what they already know.

With the buzzwords "structured programming", standard texts typically suggest that programs should be composed in such a way that they can be seen to be correct, but these texts usually do not provide the means of doing so. Instead, they introduce programming constructs only by describing storage of values and flow of control, thus providing only the means to trace particular executions. This book differs by teaching the mathematics necessary for correctness concerns, in addition to providing operational models. It is not a survey of programming

methodology, but a thorough exposition of one method of programming.

Any book whose title or preface suggests that it will make programming easy is a fraud. Any book that avoids the "difficult" topics of formal semantics and analysis of efficiency avoids teaching programming, and should itself be avoided. Such books actually make programming more difficult by denying the would-be programmer the necessary intellectual tools. Avoiding difficult topics may serve universities whose goal is to maximize enrolment, but it is to the detriment of students. It is unfair to students who cannot ultimately succeed to delay the realization that programming is difficult for them, and it is unfair to good students to delay their education and the enjoyment of mastering the subject. We have plenty of poor programmers now; we need a few good ones.

The programming language used in this book is Pro. The language is introduced herein; no previous knowledge of it is assumed. Its design criteria were that it have a simple, elegant, formal definition (both syntax and semantics), and that it facilitate a mathematical style of program composition. No standard language was designed on this basis, and none meets these criteria. Programming languages are not natural objects that drop from the sky, and should not be studied as though they are. When they are inadequate or too complex, they should be criticized and changed. The Pro Language is simpler than most popular languages, but it is not a toy. It is presented in complete detail, and constitutes one theme of the book: that programming practice and programming language have a great influence on each other. Of course, students will want their knowledge to be useful when they leave university and join industry. They will still use the concepts and the notations learned here, even when their programs are to be coded in a standard language. And they will be better able to see the strengths, and especially the weaknesses, of those languages, after their exposure to Pro.

Having available a Pro implementation has some advantages, but also some disadvantages. It is fun to use computing equipment, to see one's program be executed (or at least to see the result of execution), to test one's creation. This fun should not be underestimated as a motivator; indeed, it has played a part in attracting many of us to the computing field. But the unfortunate trend in programming courses has been to encourage the attitude "try it and see if it works" as a method of program composition. Advanced language processors, with sophisticated error repair, which are intended for the laudable purposes of teaching syntax and allowing compilation to continue, have a bad side

effect: students come to think that the computer will correct their mistakes. Problem assignments seem to demand a running program (with no apparent errors) rather than a correct one (apparent that there are no errors). Graders ease their burden by grading the output from an execution of the program, rather than grading the program. In this book, the running of programs, for those who have a Pro implementation, is relegated to an appendix. Computers can usefully aid in the development of a program by providing "word processing" or "editing" facilities, by checking the syntax of the program, by checking proofs, by storing and retrieving pieces of a program, and by reminding the programmer which pieces still need to be written. These facilities, though helpful, are not necessary; pencil and paper still work. As much fun as pushing buttons may be, the real fun and the content of this book is the intellectual part of programming.

At the top of my list of people to thank are my parents; they are responsible for a large part of my training, and I hope they are not too displeased with the result. My wife, Barbara, provided the right combination of patience and impatience to help me complete this work. For my programming education and many of the ideas in this book, I thank the members of IFIP Working Group 2.3 on Programming Methodology, particularly Edsger Dijkstra, David Gries, Tony Hoare, Jim Horning, Cliff Jones, Bill McKeeman, John Reynolds, and Wlad Turski. Thanks also go to Christian Lengauer, Bob Tennent, Roland Backhouse, David Elliott, Nigel Horspool, Hugh Redelmeier, and Jean-Raymond Abrial for reading and criticizing the first draft. I thank the class to whom I first presented this material, for pointing out problems and ways to improve it. Inge Weber worked hard with a difficult text, word processor, and author, to produce a readable manuscript. Oscar Nierstrasz produced the final camera-ready copy.

This book is dedicated to my son Joshua, who has shown me that people (at least one of them) begin logical, and are urged and pressed to conform to an illogical and inconsistent world; may he continue to resist.

0 LOGIC

When airplanes were new, pilots flew by the feeling in the seat of their pants. Natural ability, bravery, daring, and luck were the necessary qualifications. One could hardly expect the early pilots to be well-trained, there being no previous generation of pilots whose experience would provide the basis for training. But today, of course, we do not entrust commercial aircraft and passengers' lives to natural ability alone, and though pilots may be brave and daring, we hope they are not often required to use those qualities.

The programmers of today, like those early pilots, use intuition and cunning; they defy all odds to make their programs fly. That illogical, corny line "It's so crazy it just might work!" describes their *modus operandi.* But the odds usually win: by far, most of the programs written today do not work! They work enough of the time that governments, financial institutions, and industries rely on them, but fail enough of the time to provide thousands of amusing stories of "computer errors" (at least, the stories are amusing when they happen to other people).

It is now time to replace the "seat of the pants" programmer by someone who is well-trained in reliable methods of program composition. The methods presented in this book are sufficient for writing any program, they are helpful, and they lead to correct programs. Intelligence is required to use them properly, and there is no guarantee against making a mistake. But they provide guidance, and for those who are careful, the way to avoid mistakes.

Does that take the excitement out of programming? To some people, debugging a buggy program is as exciting as solving a good murder mystery or puzzle. The computer industry is reserving a special

place and name for these people: they are the home computer enthusiasts, or "hobbyists". But surely the prospect of creating programs that work and that perform useful and previously unaccomplished tasks is more exciting than the fumblings of an untrained hobbyist.

The study of Informatics (Computer Science) requires a familiarity with Logic. This background is needed for digital system design, computability, data base modeling, automatic theorem proving, and other branches of Informatics. It is especially needed for the most basic part: programming. So let us begin.

0.0 Sentential Logic

What is Mathematics? The easy answer, and a useful one, is that Mathematics is what mathematicians do. They think a lot (at least the good ones do), and they write things down; they scratch their heads; sometimes they smile at what they've written, or even jump for joy; sometimes they frown and then usually they erase whatever it was that made them frown. Anyone who wants to become a mathematician must learn what to write, when to smile, and when to frown. The rules saying what to write are called the *syntax* of Mathematics; the rules saying when to smile or frown are called the *semantics*. Syntax is also known as "form", and the things written according to the syntactic rules are called *formulas*, or synonymously, *expressions*. Semantics is also known as "meaning".

The first piece of Mathematics we need is Sentential Logic; its older name is Propositional Calculus; it is sometimes also called Boolean Algebra, after its inventor George Boole (in 1854), although that term can be applied to a more general class of algebraic structures.

0.0.0 Syntax

Here is the syntax of Sentential Logic.

0. **true** and **false** are formulas.
1. If a is a formula, then $(\neg a)$ is a formula.
2. If a and b are formulas, then the following are all formulas.

$$(a \wedge b)$$
$$(a \vee b)$$
$$(a \Rightarrow b)$$
$$(a = b)$$

For the moment, there are no other ways of forming formulas. In rule 1, the formula is called negation, and the symbol "$\neg$" is pronounced "not". In rule 2, the formulas are called (respectively) conjunction, disjunction, implication, and equation, and the symbols are pronounced "and", "or", "implies", and "equals". In a conjunction $(a \wedge b)$, the two subformulas a and b are called conjuncts; in a disjunction $(a \vee b)$, a and b are called disjuncts; in an implication $(a \Rightarrow b)$, a is called the antecedent, and b the consequent; in an equation $(a = b)$, there are no special names for a and b except "left side" and "right side", which apply to the other formulas also.

> **Aside**. In other texts, $(\neg a)$ may be written $(\sim a)$ or (a') or $\bar{a}$; $(a \wedge b)$ may be written $(a \,\&\, b)$ or $(a \bullet b)$; $(a \vee b)$ may be written $(a \mid b)$ or $(a + b)$; $(a \Rightarrow b)$ may be written $(a \supset b)$ or $(a \rightarrow b)$; $(a = b)$ may be written $(a \equiv b)$ or $(a \leftrightarrow b)$. Formulas are sometimes called "well-formed formulas", suggesting that there are other kinds of formulas (ill-formed?). **End of Aside.**

Here is a formula.

$$((\mathbf{true} \wedge (\neg(\mathbf{false} \vee (\mathbf{true} \Rightarrow \mathbf{false})))) \Rightarrow (\mathbf{false} = \mathbf{true}))$$

To see that it is, we must see how to build it according to the rules; the activity of determining whether something is a formula, i.e. whether it follows the syntactic rules, is called "parsing".

When we move on to other parts of Mathematics and programming, we may need to distinguish the formulas we introduce there from the ones we have introduced here. The phrase "formula of Sentential Logic" is a little too long to use repeatedly; we can say more briefly "sentential formula", or "sentential expression", or "propositional formula", or "propositional expression". Even more briefly, we shall say "sentence", or "proposition", and still we mean a formula of Sentential Logic.

0.0.1 Semantics

Our present goal is to become mathematicians, not philosophers; accordingly, we shall not consider the question "What is truth?". (As this book goes to press, the latest report is that the philosophers have not yet settled that question.) We shall use the words "true" and "false" simply to identify two classes of sentences: the ones we call true are those that make mathematicians smile; the false ones are those

that evoke a frown. To give the semantics of Sentential Logic, we must place the sentences into these two classes. Let t and t' be any true sentences, f and f' be any false sentences, and a be any sentence.

true sentences	false sentences
true	**false**
$(\neg f)$	$(\neg t)$
$(t \wedge t')$	$(f \wedge a)$
$(t \vee a)$	$(a \wedge f)$
$(a \vee t)$	$(f \vee f')$
$(a \Rightarrow t)$	$(t \Rightarrow f)$
$(f \Rightarrow a)$	$(t = f)$
$(t = t')$	$(f = t)$
$(f = f')$	

Now we know what sentences mean. For a complicated sentence, it may not be obvious which class it belongs to, but we can find out by replacing parts of it by simpler sentences that mean the same thing. Our earlier example

$$((\textbf{true} \wedge (\neg(\textbf{false} \vee (\textbf{true} \Rightarrow \textbf{false})))) \Rightarrow (\textbf{false} = \textbf{true}))$$

means the same as

$$((\textbf{true} \wedge (\neg(\textbf{false} \vee \textbf{false}))) \Rightarrow \textbf{false})$$

because $(\textbf{true} \Rightarrow \textbf{false})$ means the same as **false** and $(\textbf{false} = \textbf{true})$ means the same as **false**. Continuing this process, we obtain

$$((\textbf{true} \wedge (\neg\textbf{false})) \Rightarrow \textbf{false})$$
$$((\textbf{true} \wedge \textbf{true}) \Rightarrow \textbf{false})$$
$$(\textbf{true} \Rightarrow \textbf{false})$$
$$\textbf{false}.$$

and so it is seen to be a false sentence.

The activity of determining what a sentence means is called "evaluating" the sentence. It uses a principle that is common to all mathematical formulas, but not to all natural languages: the "Denotational Principle", or "Transparency Principle".

Transparency Principle. If one formula is part of a larger formula, and the part is replaced by another formula with the same meaning, then the meaning of the larger formula is not changed.

Sentential Logic can be used to reason about the world, if it is supplied with the facts. We simply add sentences about the world to our logic, putting each into the true or false class as appropriate. Here are some examples.

new syntax	new semantics
(the Pope is Catholic)	true
(water is wet)	true
(the moon is made of green cheese)	false
(I'm a monkey's uncle)	false

Determining whether these sentences are true or false is no concern of Logic; but if we are given that information, we can use Logic to analyze compound sentences. English connectives and modifiers such as "not", "and", "or", "if", and "but" can all be translated into logical symbols. For example, the English sentence "The Pope is not Catholic." strongly suggests the logical sentence

(¬(the Pope is Catholic))

which is a false sentence. The English sentence "Water is wet and the moon is made of green cheese." should become the logical sentence

((water is wet) ∧ (the moon is made of green cheese))

which can be seen to be false by evaluating it:

(true ∧ false)
false

The English sentence "Either the Pope is Catholic, or the moon is made of green cheese." suggests the logical sentence

((the Pope is Catholic)
∨ (the moon is made of green cheese))

which is evaluated as follows:

(true ∨ false)
true

and is seen to be true. From the English sentence "If the moon is made of green cheese, then I'm a monkey's uncle." we obtain the logical sentence

((the moon is made of green cheese)
⇒ (I'm a monkey's uncle))

and evaluate it