LOGIC

FOR

COMPUTER SCIENCE

Steve Reeves
and
Mike Clarke

Department of Computer Science
Queen Mary and Westfield College
University of London
U.K.

Department of Computer Science
University of Waikato
New Zealand

The contents of this book were first published in 1990 by Addison-Wesley Publishers Ltd.

Preface to 1990 edition

Aims

The aim of this book is to give students of computer science a working knowledge of the relevant parts of logic. It is not intended to be a review of applications of logic in computer science, neither is it primarily intended to be a first course in logic for students of mathematics or philosophy, although we believe that mush of the material will be increasingly relevant to both of these groups as computational ideas pervade their syllabuses.

Most controversial perhaps will be our decision to include modal and intuitionistic logic in an introductory text, the inevitably cost being a rather more summary treatment of some aspects of classical predicate logic. We believe, however, that a glance at the wide variety of ways in which logic is used in computer science fully justifies this approach. Certainly classical predicate logic is the basic tool of sequential program verification, but modal and temporal logics are increasingly being used for distributed and concurrent systems and intuitionistic logic provides a basis for expressing specifications and deriving programs. Horn clause logic and resolution underlie the very widespread use of logic programming, while algorithms for automated theorem proving have long been of interest to computer scientists for both their intrinsic interest and the applications in artificial intelligence.

One major (and deliberate) omission is the standard development of the logical basis of set theory and arithmetic. These theories are so well covered in a number of excellent and widely available texts (many of which are referenced in the text or are among the sources we acknowledge at the end of this preface) that we preferred to use the space for less well-exposed topics. Of course, the need to formalize arithmetic and set theory has led to major developments in logic and computer science and we have tried to give the historical perspective, while referring readers elsewhere for the detail.

Different disciplines have different motivations for studying logic and correspondingly different conventions of notation and rigour. To keep the within reasonable bounds we have decided to omit some of the lengthier explanations and proofs found in traditional logic texts in favour of introducing topics considered more 'advanced', that are central to modern computer science. In many cases, where proof methods have been specified by non-deterministic sets of rules, we have been more precise than usual by giving algorithms and programs; in other cases we have relied on the background of our students to keep routine formal development to a minimum.

Another major departure is that we present many of the definitions and algorithms as computer programs in, not just one but, two programming languages. We have chosen Prolog and SML, partly because they are both highly succinct and suitable languages for the procedures we want to express, but also because they have their roots, respectively in logic and the λ–calculus, two of the most important theoretical developments that underlie computer science and the theory of computability. Either of the languages is sufficient, but a student who carefully studies the programs in both languages will learn a lot about the theory and technique of declarative programming as well as about the logical definitions and algorithms that the programs express.

In Appendix A we give a brief introduction to the philosophy and facilities of both languages, but this is aimed to some extent at teachers and students with a considerable background in computer science. The less sophisticated will need access to one or other of the introductory texts that we recommend. That being said, the programs are designed to be readable ad should relay some message even to non-programmers, perhaps helping them to realize that much of logic is inseparable from the notion of an effective algorithm, and even encourage them to start programming.

Overall, our aim has been to show how computer science and logic are closely linked. We hope that students will see that what they might have considered a dry subject without obvious applications is being put to good use and vigorously developed by computer scientists.

Readership

Much of the material has been tested in a course given to first-year undergraduate students in computer science who, at that stage, have had an introductory course in discrete mathematics and a first programming course that emphasizes recursion, inductive proof and scope of definition. So they already have a fair grasp at a semiformal level of notions such as set, function, relation, formal

language, free and bounds variable and mathematical induction, and we believe that such a background will, if not already standard, soon become so for first-year students of computer science. The students, we are glad to say, bear out our conviction that an introductory logic course can successfully go beyond what is usually considered tot he be the appropriate level. They are able to actually do proofs using the methods we teach and are surprised and challenged by the idea of several logics. We feel that this is because computer science, properly taught, makes the student of logic easier, and vice versa. The activity of constructing and reasoning about programs in not all that different from the activity of constructing and reasoning about proofs.

Acknowledgements

Our colleagues, also, have made a big contribution to the development of the course, and subsequently the book. We would single out for special mention, in no particular order, Peter Burton, Wilfrid Hodges, Doug Goldson, Peter Landin, Sarah Lloyd-Jones, Mike Hopkins, Keith Clarke, Richard Bornat, Steve Sommerville, Dave Saunders, Mel Slater, John Bell and Mark Christian, and well as those further away—Alan Bundy, Dov Gabbay—who have influenced our views on the more advanced topics.

Finally, it will be obvious that we have been strongly influenced, and greatly helped, by many other texts whose development fo the subject we have studied, and in many instances borrowed. These have included Hodges (1977), *Logic*, Hamilton (1978), *Logic for Mathematicians*, Boolos and Jeffrey (1980), *Computability and Logic*, Scott et al. (1981), *Foundations of Logic Programming*, and Martin-Löf (1985), *Constructive Mathematics and Computer Programming.*

Steve Reeves
Mike Clarke

QMW, University of London
November, 1989

Preface to 2003 edition

Since 1990 much has changed in our subject and many further chapters could be added to the book Mike and I wrote in 1989-1990. However, I think it is good to be able to say that all of the things we wrote about then are still relevant and being used in many areas of computer science today, which is something not many authors

of computer science texts looking back over 13 years from 2003 could say—we clearly chose well.

However, there are two reasons why the book has not changed. One is that no company, today, thinks it worth publishing (well, not Addison-Wesley anyhow—now part of Pearson). To some extent you can't blame them—computer science has become more and more a ticket to a good job rather than an intellectual undertaking (that is likely to lead to a good job) taught and studied by people who are interested in it. (many of our current students are not interested in the subject, or are not very good at it, so I hate to think what their working lives, in terms of self-fulfillment, are going to be like). The publishers look around at all the courses which teach short-term skills rather than lasting knowledge and see that logic has little place, and see that a book on logic for computer science does not represent an opportunity to make monetary profits.

Why, then, has the book re-appeared? Because of repeated demands from around the world (but mainly from the USA) for copies of it! There are no longer any (new) copies for sale, so given the demand something had to be done. Hence this ersatz edition. It's not as high quality as AW's was, but then I'm not a type-setter, printer, bookbinder, designer etc. It was produced from the original Word files we gave to AW (from which, after much re-typing and re-design, they produced the 1990 edition). Those files were written using a couple of Macintosh SEs. The files have traveled around the world with me, moving from computer to computer until the 2003 version has been produced on an eMac and a Titanium Powerbook. There is one constant in Word—it still crashes reliably about once a day!

The other reason the book has not been re-written is that Mike Clarke died in 1994, so the version before you stands as a memorial to him—he was a friend and a mentor, and you can't be more than that.

Steve Reeves

University of Waikato
January 2003

CONTENTS

CHAPTER ONE
Introduction

1.1. Aims and Objectives

This book will differ from most others with similar titles because we aim to give you not one or two ways of looking at Logic, but many. The forms of reasoning that are fundamental to Computer Science are not necessarily those most familiar from a study of Mathematics and this gives us the opportunity to develop the subject along two dimensions, looking not only at different methods for implementing one particular mode of reasoning, but also at different ways of formalizing the process of reasoning itself.

There are many reasons why a computer scientist should need to study logic. Not only has it historically formed the roots of computer science, both Church's and Turing's work being motivated by the decision problem for first-order logic, but nowadays we are finding conversely that computer science is generating an explosion of interest in logic, with the desire to automate reasoning and the necessity to prove programs correct.

Basically, logic is about formalizing language and reasoning, and computer science addresses similar problems with the extra task, having formalized them, of *expressing* those formalizations, in the technical sense of producing mechanisms which follow the rules that they lay down. This, indeed, has led to the recent use of computer science for investigating logics in an experimental way, exploring some of them much more thoroughly than was possible when the 'computer' was a person rather than a machine.

What we hope then to show is that computer science has grown out of logic. It is helping to suggest new ideas for logical analysis and these logical ideas are, in turn, allowing computer science to develop further. The two subjects have each contributed to the growth of the other and still are, and in combination they form an exciting and rapidly growing field of study.

1.2. Background history

In the middle of the last century Boole laid down what we now see as the mathematical basis for computer hardware and propositional logic, but the logics that we are going to look at really started towards the end of the century with the work of Gottlob Frege, a German mathematician working in relative obscurity. Frege aimed to derive all of mathematics from logical principles, in other words pure reason, together with some self-evident truths about sets. (Such as 'sets are identical if they have the same members' or 'every property determines a set'). In doing this he introduced new notation and language which forms the basis of the work that we shall be covering. Until Boole and Frege, logic had not fundamentally changed since Aristotle!

Frege's huge work was (terminally) criticized at its foundations by Bertrand Russell who found a basic flaw in it stemming from one of the 'self-evident' truths upon which the whole enterprise was based. However, Russell developed the work further by suggesting ways of repairing the damage. He also introduced Frege's work to the English-speaking mathematicians since not many of them, at that time, read German. Russell, who did read German, saw that the work was important and so publicized it.

1.3. Background terminology

We are going to be doing what is usually known as 'mathematical Logic' or 'symbolic Logic' or 'formal Logic'. That is, we are going to use ordinary, but careful, mathematical methods to study a branch of mathematics called Logic. Before we start to look at what Logic actually is we shall try to make the context in which we are working a bit clearer. To make the discussion concrete we can think in terms of the typical introductory programming course that you may have followed.

Such a programming course not only teaches you how to use the constructs of the language to produce the effects that you want when the program is executed, but it also teaches you the distinction between the language that you write programs in and the meaning of the statements of that language in terms of the effect that they have when executed by a computer. If the course was a good one, it will also have taught

you how to reason about programs - perhaps to show that two apparently different programs are equivalent. Logic is the study of formal (i.e. symbolic) systems of reasoning and of methods of attaching meaning to them. So there are strong parallels between formal computer science and logic. Both involve the study of formal systems and ways of giving them meaning (semantics). However in Logic you study a wider variety of formal systems than you do in Computer Science, so wide and so fundamental that Logic is used not only as one of the mathematical tools for studying programming, but also as a foundation for mathematics itself. This ought to set the alarm bells ringing, because we have already said that we were going to use mathematics to study Logic, so there is an apparent circularity here. It is certainly the case that circular or "self-referential" discussion like this is very easy to get wrong but the notion of self-reference is a central one in Computer Science and, in fact, is exploited rather than avoided.

In Logic we deal with the issue by putting the logic we are going to study in one compartment and the logic we are going to do the studying with in another. These compartments are realized by using different languages. The logic that is the object of our study will be expressed in one particular language that we call the object language. Our study of this logic and language is carried out in another language which we call the observer's language. (You might also see the word metalanguage for this.)

The idea should already be familiar to you from studying foreign or ancient languages. In this case Latin, for example, might be the object language and your native language, in which you might have discussed the details of Latin syntax or the meaning of particular Latin sentences, is the observer's language. In mathematics, the symbolism of calculus, set theory, graph theory and so on, provide the object language and again your native language, augmented perhaps with some specialised mathematical vocabulary, is used as the observer's language. In programming, the object language is a 'programming' language such as Pascal, Lisp or Miranda and the observer's language is again your native language augmented with the appropriate mathematical and operational notions.

Example 1.1

Consider the statement

```
times 0 do print* od = donothing
```

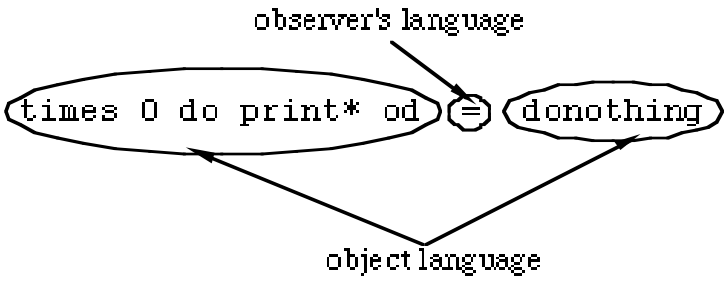

Figure 1

This, in fact, is a statement in the observer's language about the equivalence of two statements in one of our local programming languages. Although you may have guessed correctly, you have no means of saying with certainty which are the symbols of the object language and which are symbols of the observers language until the object language has been defined for you. In fact, the distinction is as shown in Figure 1.

Exercise 1.1 Now you are invited to use your linguistic, mathematical and programming experience to do a similar analysis of the following statements into observer and object languages.

(a) The sentence 'They feeds the cat' is ungrammatical in English.
(b) The French translation of the English phrase 'Thank you very much' is 'Merci beaucoup'
(c) The equation $E=mc^2$ holds in Special Relativity.
(d) There is no real value of x that satisfies $x^2 -2x + 2 = 0$
(e) There is no real value of foo that satisfies $x^2 -2x + 2 = 0$
(f) If $x^2 - 2x + 1 = 0$ then $x = 1$
(g) If $x^2 - 2x + 1 = 0$ then x must be 1
(h) If $x^2 - 2x + 1 = 0$ then x must be unity
(i) "$E=mc^2$ holds in Special Relativity" cannot be proved.
(j) The statements x:=x+1; x:=x+1; are equivalent to x:=x+2; in Pascal.
(k) "**if…then…else**" is a statement form in Pascal

You probably found that one or two of these exercises were borderline cases and caused you to point out, more or less forcibly, that it would be a lot easier if you

had an actual definition of the object language in front of you. This is the first thing we do when we embark on the development of propositional logic in Chapter 2.

1.4. Propositions, Beliefs and Declarative Sentences

The basic items that logic deals with are propositions. Philosophers have given a variety of answers to the question "What is a proposition?" but since we are dealing with the mathematics rather than the philosophy of logic it doesn't really matter for our purposes. One answer, however, is that a proposition is what is common to a set of declarative sentences in your native language that all say the same thing. Philosophers then have to argue about what "all say the same thing" means, but fortunately we don't.

Propositions communicate judgements or beliefs and since beliefs are themselves manifested as states of mind (it's hard to see what else they could be) the act of believing or the stating of propositions allows, with practice, the representation in your mind of complex objects both physical and abstract. We seem to be a long way from the limits of the human race, as a whole, in representing things mentally, and reasoning with them, and we are in the Stone Age when it comes to building ourselves tools for doing so. This is why the study of formal methods of manipulating propositions, Logic in other words, is so important.

Since Computer Science is one discipline in which the objects that we want to reason about are extraordinarily complex, and often abstract and purely formal, the need for Logic here is especially clear.

Of course, the fact that beliefs are states of mind means that we cannot directly manipulate them, neither can we manipulate propositions, since they are expressions of those states of mind. What we do is to manipulate sentences in some language which *map on to* propositions and beliefs.

The language with which we, the authors, are most familiar for this task is the natural language called "English". We use it to express our beliefs as propositions, for the purpose of transferring them to each other, testing them and so on. When one of us says to the other "I believe that Moto Guzzi manufacture the best motorcycles in the world" he conveys part of his current state of mind, in particular a part that expresses a certain relation between himself and motorcycles.

In general, then, we use English to express our beliefs. However, we need to refine this statement since English is rather complicated and not all of English is used for this purpose. There are only certain sentences in English that convey beliefs, i.e. express propositions, and these are the declarative sentences.

Definition 1.1

A declarative sentence is a grammatically correct English sentence that can be put in place of '…' in the sentence "Is it true that …?" with the effect that the resulting sentence is a grammatically correct English question.

One might expect further restrictions here, though. The definition has a rather syntactic bias to it, and English is notoriously expressive. We cannot go into it fully here, but a good introductory discussion can be found in Hodges (1977).

Exercise 1.2 Decide whether the following are declarative sentences or not:

(a) What is your name?
(b) Close the door!
(c) Grass is green.
(d) Grass is red.
(e) It is wrong.
(f) I am honest.
(g) You must not cheat.
(h) It is false that grass is red.

1.5. Contradictions

By this stage you should have some feel for how beliefs are manipulated in natural language. But how are beliefs actually useful? What is their reason for existing? Basically beliefs give a description of the world as it is or might be. For example, if I have a system of beliefs about the laws of mechanics (a description of part of the world) I can generate beliefs about the Solar System without having to actually go out there and make measurements. If I have a system of beliefs about my friends, I can predict their behaviour in certain situations without the possible embarrassment of engineering and being in those situations. Again, I can have a set of beliefs about numbers and reason about the result of the sum 2+2, without actually creating two different sets of cardinality 2, amalgamating and counting them.

So systems of belief allow decisions to be made, facts to be conjectured; it seems that they can do anything. However, there is one limitation. You cannot simultaneously hold two different beliefs which you know contradict one another. Since we have said that Logic is important because it allows us to manipulate beliefs, it follows that a fundamental task of Logic is to be able to decide whether or not a set of beliefs is contradictory. In simple cases, as we shall see, Logic can do this in a

mechanical way. But there are inherent limitations and it may be that, ultimately, even the most ingeniously programmed machine cannot do as well at manipulating propositions as the most careful person. This belief has not yet been contradicted!

1.6. Formalization

Formalization is the process of constructing an object language together with rules for manipulating sentences in the language. One aim in doing this is to promote clarity of thought and eliminate mistakes.

Another equally important issue, one that gives rise to the term "formalization" itself, is that we provide a means of manipulating objects of interest without having to understand what we are doing. This sounds at first like a retrograde step, but to give an example: arithmetic arose out of the formalization of counting. So we now have a set of rules which we can follow to add together numbers correctly without needing to understand what numbers are or what adding up is. Of course, we can always go back to the original physical act and see that adding up comes from the process of counting up to a particular number n with reference to one group of objects and then starting from n+1 in counting a second group. The answer in this case is what we would formally call the sum of the numbers concretely represented by each group.

So the power of formalization is that, once formalized, an area of interest can be worked in without understanding. If the agent following the rules is a human being this might be a mixed blessing, since understanding at the intellectual level is a strong motivation for getting things done. But, if you want to write a computer program to reason, then formalization is essential. Equally essential, if the results are to be useful, is to be able to prove that, as long as the rules are correctly applied, the results will be correct.

For instance, a programmer employed in the financial sector may have, in the form of a set of beliefs that are related in complicated ways, an idea of how the Stock Exchange works. It is the abstract structure of these relationships which models the concrete structure of the Stock Exchange and forms a model of how the Stock Exchange works. The programmer will then formalize this model when writing a computer system to automatically deal in the Stock Exchange, say. Now, if you look at the program, it is clear that the names of the objects in the program do not matter. Nor does the language in which they are written. What matters is that the relationships in the real thing are faithfully and fully represented in the program. This

is the sense of formalization that we are concerned with: the program should model the form of the real thing in interaction between its parts.

In the next chapter we make a start by looking at a simple form of declarative sentence and we show how it can be used to formalize some basic instances of reasoning.

Summary

- Mathematical logic began towards the end of the last century when Frege developed what is now the predicate calculus.

- Mathematical logic involves applying standard mathematical methods to the study of systems that themselves can be used to formalize mathematics. The apparent circularity is overcome by distinguishing between the *object* language, in which the formal system is expressed, and the *observer's* language in which properties of the formal system are expressed and reasoned about.

- The basic items that logic deals with are *propositions*. Propositions are used to express beliefs. In natural language they are represented by *declarative sentences*.

- The notion of belief is a very general one; nevertheless there are some restrictions on the way beliefs can be manipulated in mental reasoning. For example you *cannot simultaneously hold contradictory beliefs* (at least without being aware that something is wrong).

- The importance of formalization is that once a particular area of mathematics or computer science has been formalized, reasoning in it can be carried out purely by symbol manipulation, without reference to meaning or understanding, and mathematical properties of the reasoning process can be clearly stated and proved.

CHAPTER TWO
Formalizing the Language

2.1 Informal Propositional Calculus

2.1.1 The language

We will use □ , □, , ¬ and □ as our standard symbols for the connectives. These symbols form part of the alphabet of the language of propositional logic. Other elements of the alphabet are the parentheses, *)* and *(*, and a set of propositional variables, for instance {*p, q, r, s*}. We can now give a proper definition of the conditions that a sequence (string) of symbols must satisfy to be a sentence of propositional logic.

Definition 2.1

If *P* is a set of propositional variables then:

1) a propositional variable from the set *P* is a sentence,

2) if $\mathcal{S}$ and $\mathcal{T}$ are sentences then so are $(\neg\mathcal{S})$, $(\mathcal{S} \square \mathcal{T})$, $(\mathcal{S}$ $\mathcal{T})$, $(\mathcal{S} \square \mathcal{T})$ and $(\mathcal{S} \square \mathcal{T})$,

3) no other sequences are sentences.

We can, for example, use the definition to show that *p*, *(p □ q)*, *(p □ (¬q))* and *((p □ q) □ r)* are sentences, because *p* is a sentence by clause 1, since *p* is a propositional variable, *(p □ q)* is a sentence because *p* and *q* are sentences by clause 1 and hence, by the fourth condition in clause 2, the whole string of symbols is a sentence. The other cases follow similarly.

In practice, to keep the number of brackets to a minimum, there is a convention that ¬ takes precedence over or, as it is sometimes put, "binds more tightly" than $\wedge$ and $\vee$, which in turn bind more tightly than $\rightarrow$ and $\leftrightarrow$. Also, outside parentheses can often be omitted without ambiguity. So $(p \wedge (\neg q))$ would usually be written as $p \wedge \neg q$ and $((p \wedge q) \rightarrow r)$ as $p \wedge q \rightarrow r$.

Furthermore, we can see that $\neg p)$, for instance, is not a sentence since, although we have that p is a sentence (clause 1), none of the other clauses makes $\neg p)$ a sentence so by clause 3 it is not a sentence.

The set of symbol sequences (strings) defined in this way is called the set of (well-formed) sentences or language of propositional logic. The form of the definition is important not only because it is the first of many that we shall be seeing, but also because it determines the property of "being a sentence of propositional logic" as being decidable. That is, the question "is this sequence of symbols, formed from the alphabet of propositional logic, a sentence of propositional logic?" can always be answered correctly either "yes" or "no". Later on we shall see some similarly structured questions which cannot be answered in all cases. These will be called undecidable questions.

There are two ways to be sure that this (or any) definition gives rise to a decidable property; you can either construct a proof that it is so or you can construct a program which always gives the correct answer to the question "is this string a sentence?". By "always gives the correct answer" here we mean that, whenever the question is asked, the program answers it correctly before it terminates - and it always terminates. Clearly, to implement the definition as a program involves much more work than only proving that the definition gives a decidable property, but for the extra work we gain a program that can always answer the question correctly without further work on our part.

With a suitably expressive programming language we can use the above definition, of sentences based on the set P of propositional variables, to give us almost directly a programmed decision procedure. First, though, we have to represent the forms of sentence defined by the grammar in the language that we will use.

In this book we use two programming languages, SML and Prolog, that are becoming widely used in computer science for implementing algebraic and logically based calculations. These languages are also theoretically interesting in their own right. SML is based on the λ-calculus and type inference, while Prolog is abed on the notion of logic programming (see Chapter 7). An introduction to each of the languages is given in Appendix A. Using the datatype feature of SML makes it easy