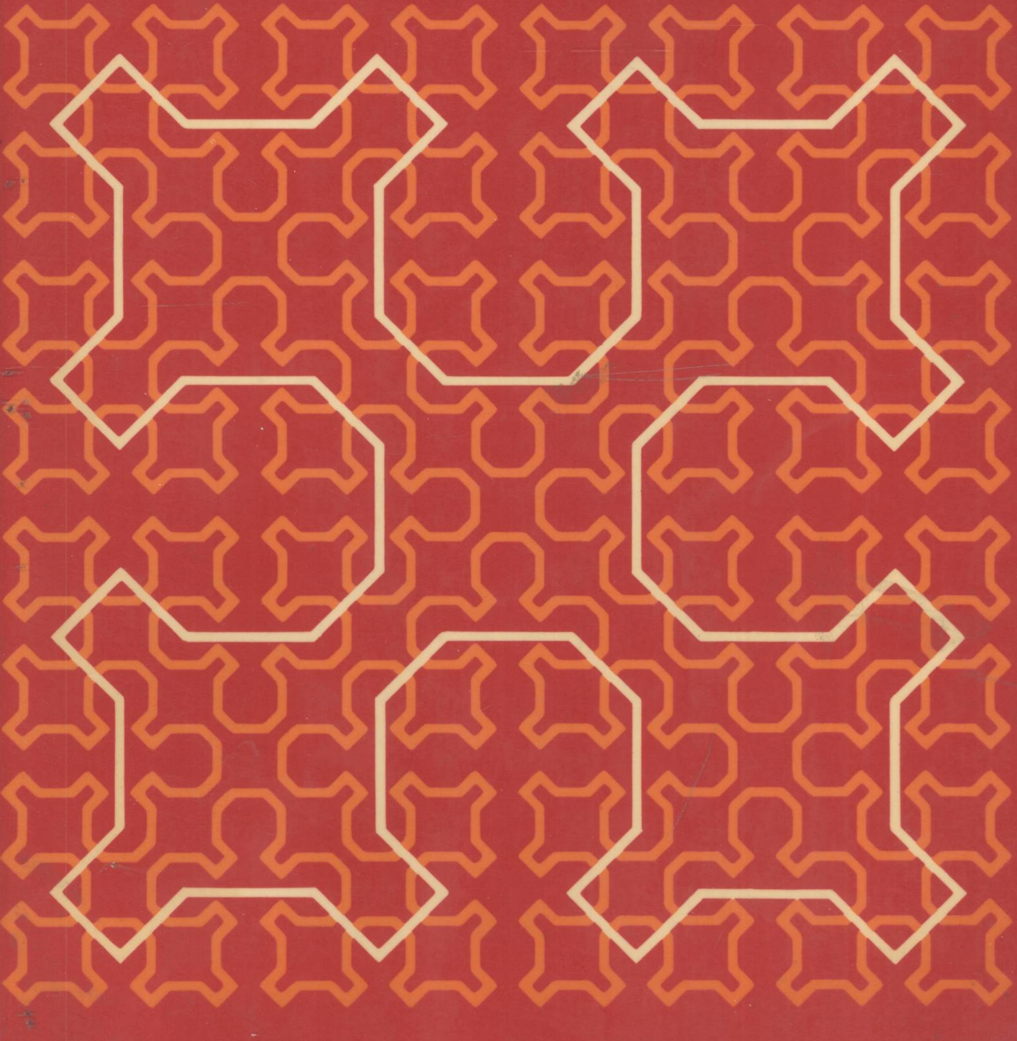
Cambridge Computer Science Texts 22

Program Construction

R.G.STONE & D.J.COOKE



22 Cambridge Computer Science Texts

Program construction

R. G. Stone and D. J. Cooke
Department of Computer Studies, Loughborough University of Technology

Cambridge University Press
Cambridge
London New York New Rochelle
Melbourne Sydney

Published by the Press Syndicate of the University of Cambridge
The Pitt Building, Trumpington Street, Cambridge CB2 1RP
32 East 57th Street, New York, NY 10022, USA
10 Stamford Road, Oakleigh, Melbourne 3166, Australia

First published 1987

Printed in Great Britain by Billing & Sons Ltd, Worcester

British Library cataloguing in publication data

Stone, R.G.
Program construction. – (Cambridge computer science texts; 22)
1. Electronic digital computers – Programming
I. Title II. Cooke, D.J.
005.1 QA76.6

Library of Congress cataloguing in publication data

Stone, R. G., 1950–
Program construction.
(Cambridge computer science texts; 22)
Bibliography: p.
Includes index.
1. Electronic digital computers – Programming.
I. Cooke, D. J. (Derek John), 1947–
II. Title. III. Series.
QA76.6S767 1987 005.1 86-12954

ISBN 0 521 26823 0 hard covers
ISBN 0 521 31883 1 paperback

MP

Preface

This text promotes the disciplined construction of procedural programs from formal specifications. As such it can be used in conjunction with any of the more conventional programming texts which teach a mixture of 'coding' in a specific language and *ad hoc* algorithm design.

The awareness of the need for a more methodical approach to program construction is epitomised by the use of phrases such as 'software engineering', 'mathematical theory of programming', and 'science of programming'. The hitherto all-too-familiar practices of 'designing' a program 'as you write it' and 'patching' wrong programs being more appropriate to a cottage industry rather than a key activity in the current technological revolution.

The cost of producing hardware is decreasing while the production of software (programs) is becoming more expensive by the day. The complexity and importance of programs is also growing phenomenally, so much so that the high cost of producing them can only be justified when they are reliable and do what they are supposed to do – when they are correct.

No methodology can exist by which we can produce a program to perform an arbitrary task. Consequently that is **not** the aim of the book. What we **shall** do is to show how, by using a Program Design Language and templates for your chosen target language, you can develop programs from certain forms of specification.

Although programming is essentially a practical activity, the degree of formality adopted throughout the development process means that sufficient information is available to enable correctness proofs to be investigated if and when required. Moreover, the structured programming forms used throughout the text are all supported by verification rules derived from their total correctness proofs – the notion of correctness never being far from our thoughts.

The material presented has grown out of courses presented to first year Computer Science undergraduates, to 'conversion' M.Sc. students and in industrial short courses in software engineering, and as such has been under development since 1980.

During the evolution of the teaching material included herein we have been influenced by many sources. Of particular note is the work of Cliff Jones (now at Manchester University but previously at Oxford PRG and various IBM research establishments), on Specification; and the work of John Darlington (now at Imperial College, London and previously at Edinburgh University), on Program Transformations. At a more tangible level we wish to record our thanks to Terry Butland of UKAEE Winfrith and Morry van Ments of Loughborough's Centre for Extension Studies for their help in organising our industrial courses, to our colleagues, Mike Woodward and Dave Gittins, who helped modify earlier drafts of the text. to Jacqui Bonsor, Carole Hill and Deborah Harrison who produced the bulk of the typescript and to Ernest Kirkwood of Cambridge University Press for his encouragement and patience.

R. G. Stone
D. J. Cooke
Loughborough, 1985

Contents

1

A modern approach to computing

1.1 An appraisal of the current situation. Is anything the matter?

In the early days of computing the machines were not very powerful, there were not many of them and few people had high expectations of them. All that has changed. Computers seem to have become an essential part of everyday society and large numbers of people are employed in supporting existing computer systems and creating new ones.

Although the use of computers is widespread the public image of computers and the computing profession is in need of improvement. Everyone has their own story to tell of the time when their enquiry was rejected with the excuse that 'it's not possible since we installed the computer system'. There have been some well-publicised disasters with new computing systems.

Yes, something is the matter!

What is wrong with computing today?

Is it the machines? Well they are cheaper, smaller, faster and more reliable than they used to be. No, they do not seem to be the problem.

Is it the programs then? Software today is more expensive, more complex, but no more reliable than it used to be.

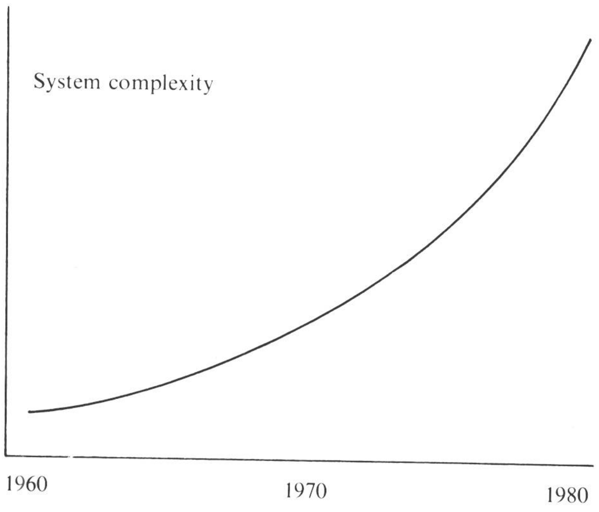
Why should this be? Is it the fault of the programmer teams? Are they not as clever as they used to be? No, they have been asked to do the impossible. It is like asking a child who has built toy houses out of Lego bricks to design and build tower blocks for people to live in. Using another analogy, it is like asking people who have discovered how to cross streams by stepping-stones and planks of wood to build a suspension bridge over an estuary. This is the scale of the increase in complexity that has faced programmers in recent years.

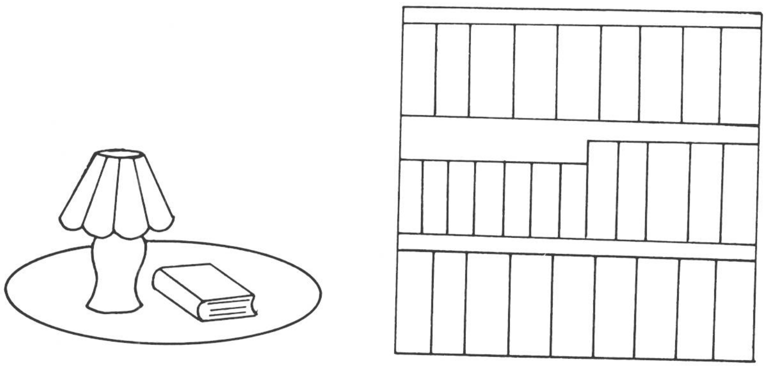
The increase in complexity is graphically illustrated by Figure 1.1.

What is being done?

Well until recently, not a lot. It has taken a long time to obtain widespread acknowledgement within the computing community that a problem exists. Now that this is established progress is being made – albeit slowly.

Fig. 1.1





Documentation

There were high hopes that 'Structured Programming' would be the solution. This was only marginally successful but crucially important for creating the idea that training and retraining computer professionals was possible and necessary.

The construction industry

Pursuing the bridge building analogy a little further it is quite clear that real bridges are not built by dumping ballast, concrete, etc., into the middle of the water on day one. In fact a prolonged process of surveying, designing, costing, model building and testing is performed before any construction is begun. This bridge construction process is entirely appropriate for the permanent large scale structures capable of supporting road or rail traffic.

(Of course this is not an appropriate solution for the hiker who simply wants to cross a stream to get to his destination before sunset. He will use only the immediately available materials such as stones, branches, etc., and he will experiment — e.g. to see if it will hold his weight.)

The fault with program construction in comparison to bridge construction is that no equivalent of the detailed drawings of the design of a bridge is in general use. That is not to say that there are no diagrams – there are – but they tend to be used in a 'cavalier' way, not as part of a methodical process.

The world of the artist

Let's move away from the construction industry and consider a possible comparison between program construction and oil painting.

Because it is possible to overpaint any colour with any other using oils, we could say that it does not matter if mistakes are made – they can always be painted over until we get it right.

This is a useful analogy with computing because apart from a small percentage of control programs (notably space shuttle landing programs) the consequences of errors in programs are not disastrous. They are frustrating, cost time and money to put right but are not disastrous, so why bother to get the program right first time?

The snag is that the *ability* to overpaint does not in itself make the person holding the paintbrush into a master artist. In fact the greater the artist the less likely their need to overpaint!

The detection of errors

It is symptomatic of the state of computing that errors are still known as 'bugs', in an attempt to pass off the blame onto some unnamed interfering force that spoils our otherwise perfect programs. (The term 'bug' originates from the days when computers contained large numbers of electromechanical relays into which insects could, and occasionally did, penetrate thus preventing normal operation – those days have passed but the term is still with us!)

Testing, whether performed by the originators of software or specialist teams (or the customer!), may reveal some errors which can be 'corrected'. What then? Who knows if there are as yet undetected errors in the original code or new errors that have been introduced with the 'corrections'?

If you were an astronaut, would you be satisfied with the statement that 'all known bugs in the shuttle landing program have been eliminated'? You would want *proof* that the program would not fail! Alright, but what does that mean? Proofs are available as a tool only in mathematics – we will have to find a way of discussing programs as mathematical objects.

The distribution of effort

There is undeniably a sense of achievement to be had from 'getting a program to run'. People who have the ability to 'fix' problems in programs that have defeated everyone else are highly valued. But in the larger systems currently being built far too large a percentage of project time is being spent on this activity. Much more effort is needed in the early stages to minimise the need for 'testing and debugging'. This means getting the specification right at the beginning and sticking to it.

A powerful argument for getting things right at the beginning is that the cost of correcting errors increases dramatically the later the error is discovered during the production of a system (see Figure 1.2).

What is in a specification?

What does the specification of a bridge achieve? In order to be useful the design must be

concise	not full scale, no irrelevant detail, but still representing the intended bridge adequately
consistent	plan and elevation agree
precise	no ambiguities.

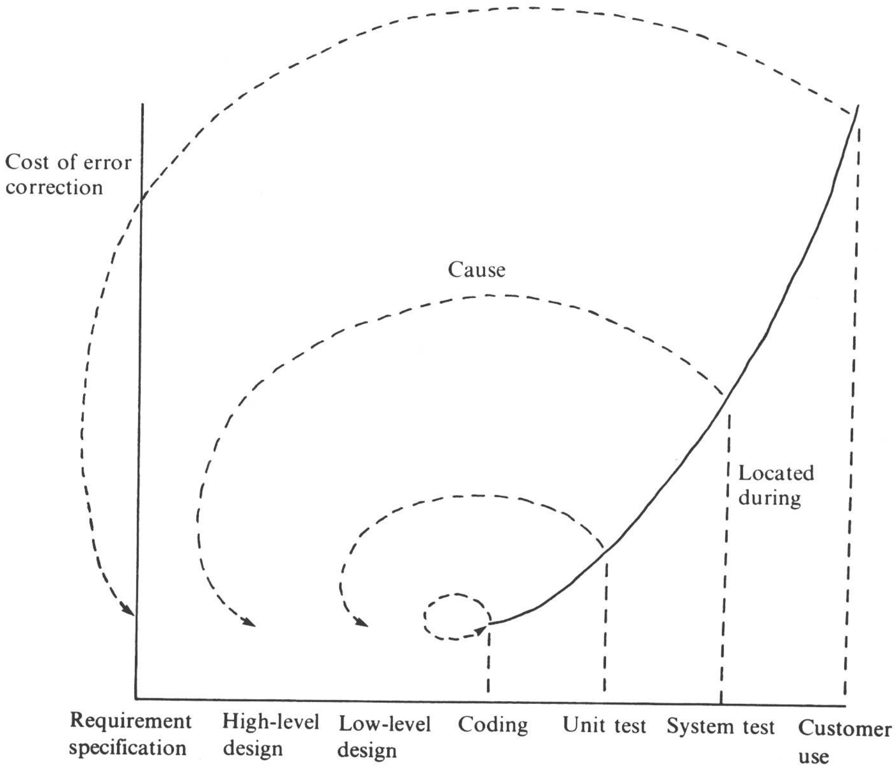
These are exactly the requirements for specifications of computer software only the medium for expressing the specification is different.

Flowcharts and other drawings have been tried and found wanting. The English language has been tried and found to be too ambiguous – or if not ambiguous then too verbose. The language that is known to be concise and precise that is currently being encouraged for use in specification is mathematics – which also has the advantage of allowing consistency proofs.

A revolution

The scenario sketched out up to now is that for a revolution in computing. There is a desperate need for formality, the ability to work with mathematical notation, and above all a desire to create high-quality correct software. The skills of coding ingenuity, optimisation and patching are becoming less important in favour of formal specification and systematic implementation with the backing of the rigour and precision of mathematics.

Fig. 1.2



1.2 A way ahead

So, there is a problem. Much Computer Science research over recent years has been directed at alleviating this 'software crisis' and the methods presented here incorporate some of the more tangible results to emanate from this research. We shall present a **practical** methodology for constructing procedural programs from formal specifications, in such a way that the individual steps can be justified (**mathematically** if necessary).

This book is not a course on algorithm design – such a course requires more detailed study relative to the specific problem domain, such as sorting, numerical analysis, file processing, etc., all proper subjects in their own right. Nor is it a book on 'writing programs in X' (name your own X!!) although coding in some specific language is necessary of course.

Our intention is that you should be able to take a specification, written in a particular Specification Language, and, using a Program Design Language (PDL), extract a program plan which is subsequently encoded in an executable Target Language. Currently, specification languages are in a state of flux. We have chosen to base our presentation on VDM – the Vienna Definition Method, named after the IBM Vienna Laboratories where it was originated – which is the only such language to have appeared in a text book [20]. (Other systems gaining support but not yet generally seen outside of research journals and conference papers are 'Z' – see [17] for a very readable example of a Z specification – and languages variously called OBJ and CLEAR, etc., developed by Goguen and Birstall and their fellow workers [16]. The equational systems presented in our Chapter 10 closely follow the style of CLEAR.) At the other end of the spectrum, typical target languages are Pascal, FORTRAN and assemblers, but the choice here is almost limitless.

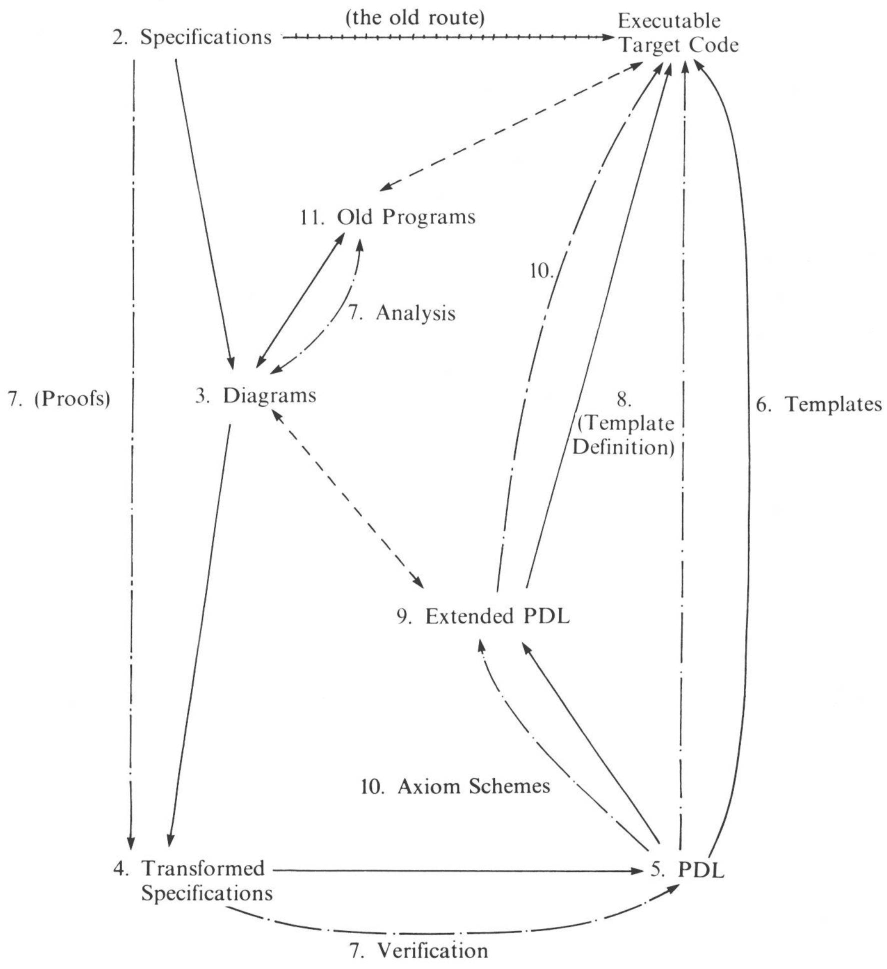
Once into PDL the remainder of the construction process is largely 'handle-turning' and hence may be automated; the earlier part cannot yet be treated in this way – there would be no need for conventional programmers if this were so.

Of necessity our specifications are formal – if a specification is to be translated into a program which causes a computer to react in a purely mechanistic way then the same level of formalisation must be inherent in the specification. Construction of the specification is non-trivial, it requires detailed knowledge of the application subject area and an understanding of how the user interfaces with the computer system. This is a problem of ergonomics and is not addressed here.

For reasons discussed at length in the body of the text, we shall restrict the way in which we interface with (real) target languages. Not to do this would necessitate extensive knowledge of the semantics of the particular language and its implementation. The approach adopted gives ample scope for object code optimisers to make the code more efficient; efficiency being a much lower priority than the correctness of the program as delivered by us.

The entire methodology is based firmly on formal specifications and the reader will be better equipped to appreciate how they are to be used after they have been introduced in Chapter 2. Nevertheless, in the hope of whetting the appetite we now attempt a brief overview of our *modus operandi*, our plan. In keeping with our philosophy of using diagrams as a legitimate aid to 'sorting things out' we shall use a diagram here.

Fig. 1.3



In this diagram the numbers refer to our chapters, the solid lines (—→) relate to the practical stages in developing code to satisfy a specification, the chained lines (–··–·· →) indicate where formal justification can be provided to ensure validity of these methods, and dashed lines (– – – –→) show other logical connections. The remaining arrow (⧺→) is only for completeness and indicates the old, insecure, link between the problem and an answer (not necessarily a solution!). Notice that it does not have an associated 'proof' arrow although one can be found by going via 4, 5 and 6 (or 4, 5 and 9). To take this logical route is to admit the possibility of a stepwise practical approach, *voila*!

As already noted, Chapter 2 sees the introduction of specifications. It is in the users interest to ensure that what is specified is exactly what he wants specified. This is where **logic programming languages** come into their own, programs in such languages being of similar structure to specifications. However implementations of such languages are too inefficient for the majority of 'final' systems. At their current stage of development logic programming languages are probably most suitable for prototyping (checking out specifications) and as such lie outside the scope of the text.

Diagrams can be used to represent the flow of data through a specification as well as control flow through conventional (procedural) programs. The disciplined use of diagrams is the subject addressed by Chapter 3 and this leads naturally to the specifications in Chapter 4 which presents a glimpse of how we may transform specifications. Ultimately we wish to move from a logical form to a functional one from which we can extract a procedural program. This aim – which is attainable for those tasks which are soluble by computer, although there are theoretical limitations – is a considerable challenge.

As an intermediate goal we introduce PDL in Chapter 5 and then, in Chapter 6, consider how to realise PDL in more familiar languages. Our Program Design Language is similar to Pascal and Algol 68, but is neither. It has simple semantics, which are discussed in Chapter 8, and can be extended by the addition of Abstract Data Types (ADTs) to create a higher level PDL in which the data types are oriented more to particular application areas. These ADTs are introduced in Chapter 9.

Formal questions relating to PDL and its possible extensions are discussed at some length in Chapters 7 and 10. Essentially these look at the requirements of correctness theorems. Knowledge that a correctness theorem **can** be proved for a given specification/program combination is enough, we don't need to prove it again. However, if such a proof is known then so is the program and we need not rewrite it; if any aspect of the problem is new then we ought to consider how to formally verify that our 'solution' **is** a solution. Details of such proofs can often be omitted or checked by an automated theorem prover but the program constructor will still be required to know **how** the specification, the program and the proof inter-relate.

Chapter 11 tackles the question of how to cope with large, existing, important programs; how to rationalise their existence. In a perfect world such potentially 'dodgy' programs would either not exist or could instantly be ejected and replaced by verified software. This is not so, and hence we have Chapter 11.

Finally, Chapter 12 includes a small case study. This is complete except for full formal proofs. Such proofs would probably double the size of this book.

In teaching courses based on this material, notation and terminology has always been a problem. In an undergraduate context, when timescales are much larger and a proper computer-oriented mathematics course is run in tandem, little difficulty is experienced by the student. In the case of industrial short courses or post-graduate 'conversion' courses time itself (or should we say, lack of it) is the main problem. What is required is a facility to treat topics in an abstract mathematical fashion. Mathematics here does not imply such topics as traditional calculus, which is totally irrelevant, but exposure to almost any kind of algebra would be beneficial. At Loughborough we use our own local text [8] and the Alvey directorate has funded the production of short-course material [36] aimed specifically as a pre-requisite for formal software engineering courses. But the use and availability of such material is outside of our dictate. Within the confines of this book we shall attempt to ease the introduction of notation by using two forms. For instance we may initially write IS_EQUAL_TO and later, when the reader is used to 'saying the words' and is getting tired of writing so much, replace it by '='. Consistent with using simple arithmetic examples from the beginning, we shall however presume that the reader can do simple 'sums' and is familiar with the symbols, $+$, $-$, $*$ (for multiplication), $\div$, $<$, $\leqslant$, $=$, etc. The only other symbol not properly introduced in the text is '$\square$' which is used to indicate the end of a proof; but this is only used in Chapters 7 and 10.

Our assumptions about computers and the readers' knowledge of computers are minimal. As viewed through programming languages they are devices for storing symbolic data, performing simple arithmetic and logical operations – one at a time – under the control of a list of commands, with the added facility that we can jump about within this list.