

a
discipline
of
programming
edsgar
w.
dijkstra

“ For a long time I have wanted to write a book somewhat along the lines of this one: on the one hand I knew that programs could have a compelling and deep logical beauty, on the other hand I was forced to admit that most programs are presented in a way fit for mechanical execution but, even if of any beauty at all, totally unfit for human appreciation. ”

A DISCIPLINE OF PROGRAMMING

EDSGER W. DIJKSTRA

*Burroughs Research Fellow,
Professor Extraordinarius,
Technological University, Eindhoven*

PRENTICE-HALL, INC.

ENGLEWOOD CLIFFS, N.J.

Dijkstra, Edsger Wybe.

A discipline of programming.

1. Electronic digital computers—Programming.

I. Title.

QA76.6.D54 001.6'42 75-40478

ISBN 0-13-215871-X

© 1976 by Prentice-Hall, Inc.
Englewood Cliffs, New Jersey

All rights reserved. No part of this book
may be reproduced in any form or by any means
without permission in writing
from the publisher.

10

Printed in the United States of America

PRENTICE-HALL INTERNATIONAL, INC., *London*
PRENTICE-HALL OF AUSTRALIA PTY. LIMITED, *Sydney*
PRENTICE-HALL OF CANADA, LTD., *Toronto*
PRENTICE-HALL OF INDIA PRIVATE LIMITED, *New Delhi*
PRENTICE-HALL OF JAPAN, INC., *Tokyo*
PRENTICE-HALL OF SOUTHEAST ASIA PTE. LTD., *Singapore*

CONTENTS

	FOREWORD	ix
	PREFACE	xiii
0	EXECUTIONAL ABSTRACTION	1
1	THE ROLE OF PROGRAMMING LANGUAGES	7
2	STATES AND THEIR CHARACTERIZATION	10
3	THE CHARACTERIZATION OF SEMANTICS	15
4	THE SEMANTIC CHARACTERIZATION OF A PROGRAMMING LANGUAGE	24
5	TWO THEOREMS	37
6	ON THE DESIGN OF PROPERLY TERMINATING CONSTRUCTS	41
7	EUCLID'S ALGORITHM REVISITED	45
8	THE FORMAL TREATMENT OF SOME SMALL EXAMPLES	51

9	ON NONDETERMINACY BEING BOUNDED	72
10	AN ESSAY ON THE NOTION: "THE SCOPE OF VARIABLES"	79
11	ARRAY VARIABLES	94
12	THE LINEAR SEARCH THEOREM	105
13	THE PROBLEM OF THE NEXT PERMUTATION	107
14	THE PROBLEM OF THE DUTCH NATIONAL FLAG	111
15	UPDATING A SEQUENTIAL FILE	117
16	MERGING PROBLEMS REVISITED	123
17	AN EXERCISE ATTRIBUTED TO R. W. HAMMING	129
18	THE PATTERN MATCHING PROBLEM	135
19	WRITING A NUMBER AS THE SUM OF TWO SQUARES	140
20	THE PROBLEM OF THE SMALLEST PRIME FACTOR OF A LARGE NUMBER	143
21	THE PROBLEM OF THE MOST ISOLATED VILLAGES	149
22	THE PROBLEM OF THE SHORTEST SUBSPANNING TREE	154

23	REM'S ALGORITHM FOR THE RECORDING OF EQUIVALENCE CLASSES	161
24	THE PROBLEM OF THE CONVEX HULL IN THREE DIMENSIONS	168
25	FINDING THE MAXIMAL STRONG COMPONENTS IN A DIRECTED GRAPH	192
26	ON MANUALS AND IMPLEMENTATIONS	201
27	IN RETROSPECT	209

FOREWORD

In the older intellectual disciplines of poetry, music, art, and science, historians pay tribute to those outstanding practitioners, whose achievements have widened the experience and understanding of their admirers, and have inspired and enhanced the talents of their imitators. Their innovations are based on superb skill in the practice of their craft, combined with an acute insight into the underlying principles. In many cases, their influence is enhanced by their breadth of culture and the power and lucidity of their expression.

This book expounds, in its author's usual cultured style, his radical new insights into the nature of computer programming. From these insights, he has developed a new range of programming methods and notational tools, which are displayed and tested in a host of elegant and efficient examples. This will surely be recognised as one of the outstanding achievements in the development of the intellectual discipline of computer programming.

C.A.R. HOARE

PREFACE

For a long time I have wanted to write a book somewhat along the lines of this one: on the one hand I knew that programs could have a compelling and deep logical beauty, on the other hand I was forced to admit that most programs are presented in a way fit for mechanical execution but, even if of any beauty at all, totally unfit for human appreciation. A second reason for dissatisfaction was that algorithms are often published in the form of finished products, while the majority of the considerations that had played their role during the design process and should justify the eventual shape of the finished program were often hardly mentioned. My original idea was to publish a number of beautiful algorithms in such a way that the reader could appreciate their beauty, and I envisaged doing so by describing the —real or imagined— design process that would each time lead to the program concerned. I have remained true to my original intention in the sense that the long sequence of chapters, in each of which a new problem is tackled and solved, is still the core of this monograph; on the other hand the final book is quite different from what I had foreseen, for the self-imposed task to present these solutions in a natural and convincing manner has been responsible for so much more, that I shall remain grateful forever for having undertaken it.

When starting on a book like this, one is immediately faced with the question: “Which programming language am I going to use?”, and this is *not* a mere question of presentation! A most important, but also a most elusive, aspect of any tool is its influence on the habits of those who train themselves in its use. If the tool is a programming language, this influence is —whether we like it or not— an influence on our thinking habits. Having analyzed that influence to the best of my knowledge, I had come to the conclusion that none of the existing programming languages, nor a subset of them, would suit my purpose; on the other hand I knew myself so unready for the design

of a new programming language that I had taken a vow not to do so for the next five years, and I had a most distinct feeling that that period had not yet elapsed! (Prior to that, among many other things, this monograph had to be written.) I have tried to resolve this conflict by only designing a mini-language suitable for my purposes, by making only those commitments that seemed unavoidable and sufficiently justified.

This hesitation and self-imposed restriction, when ill-understood, may make this monograph disappointing for many of its potential readers. It will certainly leave all those dissatisfied who identify the difficulty of programming with the difficulty of cunning exploitation of the elaborate and baroque tools known as “higher level programming languages” or —worse!— “programming systems”. When they feel cheated because I just ignore all those bells and whistles, I can only answer: “Are you quite sure that all those bells and whistles, all those wonderful facilities of your so-called “powerful” programming languages belong to the solution set rather than to the problem set?”. I can only hope that, in spite of my usage of a mini-language, they will study my text; after having done so, they may agree that, even without the bells and the whistles, so rich a subject remains that it is questionable whether the majority of the bells and the whistles should have been introduced in the first place. And to all readers with a pronounced interest in the design of programming languages, I can only express my regret that, as yet, I do not feel able to be much more explicit on that subject; on the other hand I hope that, for the time being, this monograph will inspire them and will enable them to avoid some of the mistakes they might have made without having read it.

During the act of writing—which was a continuous source of surprise and excitement—a text emerged that was rather different from what I had originally in mind. I started with the (understandable) intention to present my program developments with a little bit more formal apparatus than I used to use in my (introductory) lectures, in which semantics used to be introduced intuitively and correctness demonstrations were the usual mixture of rigorous arguments, handwaving, and eloquence. In laying the necessary foundations for such a more formal approach, I had two surprises. The first surprise was that the so-called “predicate transformers” that I had chosen as my vehicle provided a means for directly defining a relation between initial and final state, without any reference to intermediate states as may occur during program execution. I was very grateful for that, as it affords a clear separation between two of the programmer’s major concerns, the mathematical correctness concerns (viz. whether the program defines the proper relation between initial and final state—and the predicate transformers give us a formal tool for that investigation without bringing computational processes into the picture) and the engineering concerns about efficiency (of which it is now clear that they are only defined in relation to an implementation). It turned out to

be a most helpful discovery that the same program text always admits two rather complementary interpretations, the interpretation as a code for a predicate transformer, which seems the more suitable one for us, versus the interpretation as executable code, an interpretation I prefer to leave to the machines! The second surprise was that the most natural and systematic “codes for predicate transformers” that I could think of would call for non-deterministic implementations when regarded as “executable code”. For a while I shuddered at the thought of introducing nondeterminacy already in uniprogramming (the complications it has caused in multiprogramming were only too well known to me!), until I realized that the text interpretation as code for a predicate transformer has its own, independent right of existence. (And in retrospect we may observe that many of the problems multiprogramming has posed in the past are nothing else but the consequence of a prior tendency to attach undue significance to determinacy.) Eventually I came to regard nondeterminacy as the normal situation, determinacy being reduced to a—not even very interesting—special case.

After having laid the foundations, I started with what I had intended to do all the time, viz. solve a long sequence of problems. To do so was an unexpected pleasure. I experienced that the formal apparatus gave me a much firmer grip on what I was doing than I was used to; I had the pleasure of discovering that explicit concerns about termination can be of great heuristic value—to the extent that I came to regret the strong bias towards partial correctness that is still so common. The greatest pleasure, however, was that for the majority of the problems that I had solved before, this time I ended up with a more beautiful solution! This was very encouraging, for I took it as an indication that the methods developed had, indeed, improved my programming ability.

How should this monograph be studied? The best advice I can give is to stop reading as soon as a problem has been described and to try to solve it yourself before reading on. Trying to solve the problem on your own seems the only way in which you can assess how difficult the problem is; it gives you the opportunity to compare your own solution with mine; and it may give you the satisfaction of having discovered yourself a solution that is superior to mine. And, by way of a priori reassurance: be not depressed when you find the text far from easy reading! Those who have studied the manuscript found it quite often difficult (but equally rewarding!); each time, however, that we analyzed their difficulties, we came together to the conclusion that not the text (i.e. the way of presentation), but the subject matter itself was “to blame”. The moral of the story can only be that a nontrivial algorithm is just nontrivial, and that its final description in a programming language is highly compact compared to the considerations that justify its design: the shortness of the final text should not mislead us! One of my assistants made the suggestion—which I faithfully transmit, as it could be a valuable one—

that little groups of students should study it together. (Here I must add a parenthetical remark about the “difficulty” of the text. After having devoted a considerable number of years of my scientific life to clarifying the programmer’s task, with the aim of making it intellectually better manageable, I found this effort at clarification to my amazement (and annoyance) repeatedly rewarded by the accusation that “I had made programming difficult”. But the difficulty has always been there, and only by making it visible can we hope to become able to design programs with a high confidence level, rather than “smearing code”, i.e., producing texts with the status of hardly supported conjectures that wait to be killed by the first counterexample. None of the programs in this monograph, needless to say, has been tested on a machine.)

I owe the reader an explanation why I have kept the mini-language so small that it does not even contain procedures and recursion. As each next language extension would have added a few more chapters to the book and, therefore, would have made it correspondingly more expensive, the absence of most possible extensions (such as, for instance, multiprogramming) needs no further justification. Procedures, however, have always occupied such a central position and recursion has been for computing science so much the hallmark of academic respectability, that some explanation is due.

First of all, this monograph has not been written for the novice and, consequently, I expect my readers to be familiar with these concepts. Secondly, this book is not an introductory text on a specific programming language and the absence of these constructs and examples of their use should therefore not be interpreted as my inability or unwillingness to use them, nor as a suggestion that anyone else who can use them well should not do so. The point is that I felt no need for them in order to get my message across, viz. how a carefully chosen separation of concerns is essential for the design of in all respects, high-quality programs: the modest tools of the mini-language gave us already more than enough latitude for nontrivial, yet very satisfactory designs.

The above explanation, although sufficient, is, however, not the full story. In any case I felt obliged to present repetition as a construct in its own right, as such a presentation seemed to me overdue. When programming languages emerged, the “dynamic” nature of the assignment statement did not seem to fit too well into the “static” nature of traditional mathematics. For lack of adequate theory mathematicians did not feel too easy about it, and, because it is the repetitive construct that creates the need for assignment to variables, mathematicians did not feel too easy about repetition either. When programming languages without assignments and without repetition —such as pure LISP— were developed, many felt greatly relieved. They were back on the familiar grounds and saw a glimmer of hope of making programming an activity with a firm and respectable mathematical basis. (Up to this very day

there is among the more theoretically inclined computing scientists still a widespread feeling that recursive programs “come more naturally” than repetitive ones.)

For the alternative way out, viz. providing the couple “repetition” and “assignment to a variable” with a sound and workable mathematical basis, we had to wait another ten years. The outcome, as is demonstrated in this monograph, has been that the semantics of a repetitive construct can be defined in terms of a recurrence relation between *predicates*, whereas the semantic definition of general recursion requires a recurrence relation between *predicate transformers*. This shows quite clearly why I regard general recursion as an order of magnitude more complicated than just repetition, and it therefore hurts me to see the semantics of the repetitive construct

“while *B* do *S*”

defined as that of the call

“*whiledo*(*B*, *S*)”

of the recursive procedure (described in ALGOL 60 syntax):

```

procedure whiledo (condition, statement);
begin if condition then begin statement;
                                     whiledo (condition, statement) end
end
    
```

Although correct, it hurts me, for I don't like to crack an egg with a sledgehammer, no matter how effective the sledgehammer is for doing so. For the generation of theoretical computing scientists that became involved in the subject during the sixties, the above recursive definition is often not only “the natural one”, but even “the true one”. In view of the fact that we cannot even define what a Turing machine is supposed to do without appealing to the notion of repetition, some redressing of the balance seemed indicated.

For the absence of a bibliography I offer neither explanation nor apology.

Acknowledgements. The following people have had a direct influence on this book, either by their willingness to discuss its intended contents or by commenting on (parts of) the finished manuscript: C. Bron, R.M. Burstall, W.H.J. Feijen, C.A.R. Hoare, D.E. Knuth, M. Rem, J.C. Reynolds, D.T. Ross, C.S. Scholten, G. Seegmüller, N. Wirth and M. Woodger. It is a privilege to be able to express in print my gratitude for their cooperation. Furthermore I am greatly indebted to Burroughs Corporation for providing me with the opportunity and necessary facilities, and thankful to my wife for her unfailing support and encouragement.

EDSGER W. DIJKSTRA

*Nuenen,
The Netherlands*

