A DISCIPLINE OF PROGRAMMING

EDSGER W. DIJKSTRA

A DISCIPLINE OF PROGRAMMING

EDSGER W. DIJKSTRA

Burroughs Research Fellow,
Professor Extraodinarius,
Technological University, Eindhoven

PRENTICE-HALL, INC.

ENGLEWOOD CLIFFS, N.J.

Library of Congress Cataloging in Publication Data

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

10 9 8 7 6 5 4 3 2 1

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*

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 nondeterministic 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

CONTENTS

0 EXECUTIONAL ABSTRACTION

Executional abstraction is so basic to the whole notion of "an algorithm" that it is usually taken for granted and left unmentioned. Its purpose is to map different computations upon each other. Or, to put it in another way, it refers to the way in which we can get a specific computation within our intellectual grip by considering it as a member of a large class of different computations; we can then abstract from the mutual differences between the members of that class and, based on the definition of the class as a whole, make assertions applicable to each of its members and therefore also to the specific computation we wanted to consider.

In order to drive home what we mean by "a computation" let me just describe a noncomputational mechanism "producing" —intentionally I avoid the term "computing"— say, the greatest common divisor of *111* and *259*. It consists of two pieces of cardboard, placed on top of each other. The top one displays the text "GCD(*111, 259*)="; in order to let the mechanism produce the answer, we pick up the top one and place it to the left of the bottom one, on which we can now read the text "*37*".

The simplicity of the cardboard mechanism is a great virtue, but it is overshadowed by two drawbacks, a minor one and a major one. The minor one is that the mechanism can, indeed, be used for producing the greatest common divisor of *111* and *259*, but for very little else. The major drawback, however, is that, no matter how carefully we inspect the construction of the mechanism, our confidence that it produces the correct answer can only be based on our faith in the manufacturer: he may have made an error, either in the design of his machine or in the production of our particular copy.

In order to overcome our minor objection we could consider on a huge piece of cardboard a large rectangular array of the grid points with the

integer coordinates x and y, satisfying $0 \leq x \leq 500$ and $0 \leq y \leq 500$. For all the points (x, y) with positive coordinates only, i.e. excluding the points on the axes, we can write down at that position the value of GCD(x, y); we propose a two-dimensional table with *250,000* entries. From the point of view of usefulness, this is a great improvement: instead of a mechanism able to supply the greatest common divisor for a single pair of numbers, we now have a "mechanism" able to supply the greatest common divisor for any pair of the *250,000* different pairs of numbers. Great, but we should not get too excited, for what we identified as our second drawback —"Why should we believe that the mechanism produces the correct answer?"— has been multiplied by that same factor of *250,000*: we now have to have a tremendous faith in the manufacturer!

So let us consider a next mechanism. On the same cardboard with the grid points, the only numbers written on it are the values *1* through *500* along both axes. Furthermore the following straight lines are drawn:

1. the vertical lines (with the equation $x =$ constant);
2. the horizontal lines (with the equation $y =$ constant);
3. the diagonals (with the equation $x + y =$ constant);
4. the "answer line" with the equation $x = y$.

In order to operate this machine, we have to follow the following instructions ("play the game with the following rules"). When we wish to find the greatest common divisor of two values X and Y, we place a pebble —also provided by the manufacturer— on the grid point with the coordinates $x = X$ and $y = Y$. As long as the pebble is not lying on the "answer line", we consider the smallest equilateral rectangular triangle with its right angle coinciding with the pebble and one sharp angle (either under or to the left of the pebble) on one of the axes. (Because the pebble is not on the answer line, this smallest triangle will have only one sharp angle on an axis.) The pebble is then moved to the grid point coinciding with the other sharp angle of the triangle. The above move is repeated as long as the pebble has not yet arrived on the answer line. When it has, the x-coordinate (or the y-coordinate) of the final pebble position is the desired answer.

What is involved when we wish to convince ourselves that this machine will produce the correct answer? If (x, y) is any of the *249,500* points not on the answer line and (x', y') is the point to which the pebble will then be moved by one step of the game, then either $x' = x$ and $y' = y - x$ or $x' = x - y$ and $y' = y$. It is not difficult to *prove* that GCD$(x, y) =$ GCD(x', y'). The important point here is that the *same* argument applies equally well to *each* of the *249,500* possible steps! Secondly —and again it is not difficult— we can *prove* for any point (x, y) where $x = y$ (i.e. such that (x, y) is one of the *500* points on the answer line) that GCD$(x, y) = x$. Again the important point

is that the *same* argument is applicable to *each* of the *500* points of the answer line. Thirdly —and again this is not difficult— we have to show that for any initial position (X, Y) a finite number of steps will indeed bring the pebble on the answer line, and again the important observation is that the same argument is equally well applicable to any of the *250,000* initial positions (X, Y). Three simple arguments, whose length is independent of the number of grid points: that, in a nutshell, shows what mathematics can do for us! Denoting with (x, y) any of the pebble positions during a game started at position (X, Y), our first theorem allows us to conclude that during the game the relation

$$\mathrm{GCD}(x, y) = \mathrm{GCD}(X, Y)$$

will always hold or —as the jargon says— "is kept invariant". The second theorem then tells us that we may interpret the x-coordinate of the final pebble position as the desired answer and the third theorem tells us that the final position exists (i.e. will be reached in a finite number of steps). And this concludes the analysis of what we could call "our abstract machine".

Our next duty is to verify that the board as supplied by the manufacturer is, indeed, a fair model. For this purpose we have to check the numbering along both axes and we have to check that all the straight lines have been drawn correctly. This is slightly awkward as we have to investigate a number of objects proportional to N if N (in our example *500*) is the length of the side of the square, but it is always better than N^2, the number of possible computations.

An alternative machine would not work with a huge cardboard but with two nine-bit registers, each capable of storing a binary number between *0* and *500*. We could then use one register to store the value of the x-coordinate and the other to store the value of the y-coordinate as they correspond to "the current pebble position". A move then corresponds to decreasing the contents of one register by the contents of the other. We could do the arithmetic ourselves, but of course it is better if the machine could do that for us. If we then want to believe the answer, we should be able to convince ourselves that the machine compares and subtracts correctly. On a smaller scale the history repeats itself: we derive once and for all, i.e. for any pair of n-digit binary numbers, the equations for the binary subtractor and then satisfy ourselves that the physical machine is a fair model of this binary subtractor.

If it is a parallel subtractor, the number of verifications —proportional to the number of components and their interactions— is proportional to $n = \log_2 N$. In a serial machine the trading of time against equipment is carried still one step further.

Let me try, at least for my own illumination, to capture the highlights of our preceding example.

Instead of considering the single problem of how to compute the GCD(*111*, *259*), we have generalized the problem and have regarded this as a specific instance of the wider class of problems of how to compute the GCD(*X*, *Y*). It is worthwhile to point out that we could have generalized the problem of computing GCD(*111*, *259*) in a different way: we could have regarded the task as a specific instance of a wider class of tasks, such as the computation of GCD(*111*, *259*), SCM(*111*, *259*), *111* * *259*, *111* + *259*, *111*/*259*, *111* − *259*, 111^{259}, the day of the week of the *111*th day of the *259*th year B.C., etc. This would have given rise to a "*111*-and-*259*-processor" and in order to let that produce the originally desired answer, we should have had to give the request "GCD, please" as its input! We have proposed a "GCD-computer" instead, that should be given the number pair "*111*, *259*" as its input if it is to produce the originally desired answer, and that is a quite different machine!

In other words, when asked to produce one or more results, it is usual to generalize the problem and to consider these results as specific instances of a wider class. But it is no good just to say that everything is a special instance of something more general! If we want to follow such an approach we have two obligations:

1. We have to be quite specific as to how we generalize, i.e. we have to choose that wider class carefully and to define it explicitly, because our argument has to apply to that whole class.
2. We have to choose ("invent" if you wish) a generalization that is helpful to our purpose.

In our example I certainly prefer the "GCD-computer" above the "*111*-and-*259*-processor" and a comparison between the two will give us a hint as to what characteristics make a generalization "helpful for our purpose". The machine that upon request can produce as answer the value of all sorts of funny functions of *111* and *259* becomes harder to verify as the collection of functions grows. This is in sharp contrast with our "GCD-computer".

The GCD-computer would have been equally bad if it had been a table with *250,000* entries containing the "ready-made" answers. Its unique feature is that it could be given in the form of a compact set of "rules of a game" that, when played according to those rules, will produce the answer.

The tremendous gain is that a single argument applied to these rules allows us to prove the vital assertions about the outcome of any of the games. The price to be paid is that in each of the *250,000* specific applications of these rules, we don't get our answer "immediately": each time the game has to be played according to the rules!

The fact that we could give such a compact formulation of the rules of the game such that a single argument allowed us to draw conclusions about

any possible game is intimately tied to the systematic arrangement of the *250,000* grid points. We would have been powerless if the cardboard had shown a shapeless, chaotic cloud of points without a systematic nomenclature! As things are, however, we could divide our pebble into two half-pebbles and move one half-pebble downward until it lies on the horizontal axis and the other half-pebble to the left until it lies on the vertical axis. Instead of coping with one pebble with *250,000* possible positions, we could also deal with two half-pebbles with only *500* possible positions each, i.e. only *1000* positions in toto! Our wealth of *250,000* possible states has been built up by the circumstance that any of the *500* positions of the one half-pebble can be combined with any of the *500* positions of the other half-pebble: the number of positions of the undivided pebble equals the product of the number of positions of the half-pebbles. In the jargon we say that "the total state space is regarded as the Cartesian product of the state spaces of the variables x and y".

The freedom to replace one pebble with a two-dimensional freedom of position by two half-pebbles with a one-dimensional freedom of position is exploited in the suggested two-register machine. From a technical point of view this is very attractive; one only needs to build registers able to distinguish between *500* different cases ("values") and by just combining two such registers, the total number of different cases is squared! This multiplicative rule enables us to distinguish between a huge number of possible total states with the aid of a modest number of components with only a modest number of possible states each. By adding such components the size of the state space grows exponentially but we should bear in mind that we may only do so provided that the argument justifying our whole contraption remains very compact; by the time that the argument grows exponentially as well, there is no point in designing the machine at all!

Note. A perfect illustration of the above can be found in an invention which is now more than ten centuries old: the decimal number system! This has indeed the fascinating property that the number of digits needed only grows proportional to the logarithm of the largest number to be represented. The binary number system is what you get when you ignore that each hand has five fingers. (*End of note.*)

In the above we have dealt with one aspect of multitude, viz. the great number of pebble positions (= possible states). There is an analogous multiplicity, viz. the large number of different games (= computations) that can be played according to our rules of the game: one game for each initial position to be exact. Our rules of the game are very general in the sense that they are applicable to any initial position. But we have insisted upon a compact justification for the rules of the game and this implies that the rules of the game themselves have to be compact. In our example this has been achieved by the following device: instead of enumerating "do this, do that" we have