Foundations of Programming

Jacques Arsac

Ecole Normale Supérieure
Paris, France

Translated by Fraser Duncan

Foundations of Programming

Jacques Arsac

Ecole Normale Supérieure
Paris, France

Translated by Fraser Duncan

1985

ACADEMIC PRESS

(Harcourt Brace Jovanovich, Publishers)
London Orlando San Diego New York
Toronto Montreal Sydney Tokyo

ACADEMIC PRESS INC. (LONDON) LTD.
24–28 Oval Road
LONDON NW1 7DX

United States Edition published by
ACADEMIC PRESS, INC.
Orlando, Florida 32887

British Library Cataloguing in Publication Data

Arsac, Jacques
Foundations of programming.–(APIC studies in data processing)
1. Electronic digital computers–Programming
I. Title II. Les bases de la programmation.
English III. Series
000.64'2 QA76.6

Library of Congress Cataloging in Publication Data

Arsac, Jacques.
Foundations of programming.

(A.P.I.C. studies in data processing)
Translation of: Les bases de la programmation.
Includes index.
1. Electronic digital computers--Programming.
I. Title. II. Series.
QA76.6.A75813 1985 001.64'2 84-14436
ISBN 0–12–064460–6 (alk. paper)

PRINTED IN THE UNITED STATES OF AMERICA

85 86 87 88 9 8 7 6 5 4 3 2 1

FOUNDATIONS OF PROGRAMMING. Translated from the original French edition entitled LES BASES DE LA PROGRAMMATION, published by Dunod, Paris, © BORDAS 1983.

Preface

Informatics (or computer science or information processing) is too young a science to have become the subject of many historical studies or philosophical analyses. To me, however, its history seems quite exceptional. It is normal for a science to be born out of the observation of nature. When a sufficient number of duly observed facts have been recorded, the outline of a model can be sketched, and this in its turn allows the prediction of new patterns of behaviour which are then subject to experimental verification.

Technology comes later; the model being confirmed, to a certain degree of precision, by experiment, possible applications are conceived and come to birth. The discovery of electricity is typical of such an evolution. The electric motor and incandescent lamp did not appear until well after the development of the first theoretical models of electrostatics and of electromagnetism.

Much more rarely, technology has come before science. The steam engine was built long before the science of thermodynamics enabled its output to be calculated. In informatics we have an extreme example of this class.

The computer was engendered by the necessity for computation experienced during the Second World War. It was not the brainchild of some brilliant dabbler, but the product of a technology founded on a well-established science, electronics, and supported by a no less well-founded mathematical model, Boolean algebra. There is nothing unusual about this. But it was followed by wild speculation on the part of industrialists who believed that such monsters might have applications in everyday life and who engineered these machines for the market. This theme is to be found developed in a book by Rene Moreau [MOR].

From then on we have witnessed an incredible development of the means of computation, requiring the support of numerous technicians—operators, programmers, ... This in turn led to the need to establish

numerous new educational programmes, which began to be satisfied in about 1960. But the universities which took responsibility for this realised very quickly that they were not dealing simply with an application of the science of electronics aided by a very old branch of mathematics, numerical analysis. We were face to face with something fundamental, with a very old activity of the human mind, but one whose significance we were not able to grasp because there was no technical means of allowing it to take full flight.

In the same way as astronomy began with the telescopes of Galileo, but was unable fully to develop until better means of observation became technically possible, so informatics, the science of information processing, will not be able to show itself clearly until the proper means are found to exercise it [AR]. It would not be right to allow the reader to believe that what is said here is an absolute and proven truth, recognised by universal consensus. There are very many who believe in the existence of a science of informatics, as is shown, for example, by academic instances in every country: in all the great universities there are departments of "computer science"; the French Academy defines "l'informatique" as a science, and the French Academy of Sciences has two headings "informatique" in its proceedings. But there are just as many again who would restrict its scope, and deny it all claim to a character of its own. In this sense, the position of the Academy of Sciences is ambiguous: there is not one unique classification for informatics, as would be the case for a well-established discipline, but rather two, one for theoretical information science, a branch of mathematics concerned with the problems posed by information processing, and the other at the level of "science for the engineer." Many professionals see in informatics nothing more than a technique, sufficiently complicated to require a specific training, but of narrow breadth. Bruno Lussato has become a champion of this idea, asserting that informatics can be learnt in three weeks [LUS].

I believe this to be a consequence of the very peculiar development of the discipline. As had been said, the technology came first, founded as always on a science, solid-state physics, but remote from its proper concern, the treatment of information. Since then we have spent much energy trying to master techniques for which, the need not having been foreseen, we were never adequately prepared. There has hardly been time to think. The history of programming, within the history of informatics, is in this respect very revealing.

In the beginning there were the first computers, for which we had to write programs in the only language which they could interpret, their own, the machine language. This was extremely tedious, with a very high risk of error; and so machine languages were very soon replaced by assembly languages, maintaining in their forms the structures of the machine languages, and offering no more distinct operations than the machines could cope with by

hardware. It very soon became clear that the fragmentation of thought imposed by the very restricted set of these primitive operations was a handicap, and the first developed language, Fortran, appeared about 1955. This was a decisive step. The choice of structures for this language and the manner of writing of its compiler were to leave a deep mark on the history of programming.

In fact, the language was conceived fundamentally as an abbreviation of assembly languages, and maintained two of their principal characteristics:

—the assignment instruction, or modification of the contents of a unit storage in the computer,

—the idea of the instruction sequence, broken by the GO TO instruction, conditional or unconditional.

Very many languages developed rapidly from this common origin. The principal languages still in use today, Algol 60, Cobol, Basic, . . . , appeared over a period of at most five years. Lisp is of the same period, but it deserves special mention; it was not constructed on the same universal primitives, having no concept of assignment, no loop, no instruction sequence, but it was based on the recursive definition of functions.

This language explosion, this modern Babel, has caused much dissipation of energy in the writing of compilers, and in heated argument over a false problem—that of the comparative merits of the different languages. Unhappily, this game has not yet stopped. Take any assembly, anywhere, of sleepy computer scientists, and turn the discussion on to a programming language. They will all wake up and throw themselves into the debate. If you think about this, it is worrying, indeed deeply disturbing. All these languages in fact have been built upon the same universal primitives, as has already been pointed out; and the principal problems flow from this fact. Two of the chief difficulties in programming practice arise from the assignment instruction and the branch instruction. Why such rivalry between such close cousins?

As long as we work ourselves into such excitement over these languages, we are apt to forget that they are only a means of expression and that the essential thing is to have something to say, not how to say it.

> Ce que l'on concoit bien s'énonce clairement et les mots pour le dire arrivent aisement [BOI].

["That which is well conceived can be expressed clearly, and the words for saying it will come easily"—Nicolas Boileau, 1636–1711.]

The means for the construction of programs were rudimentary, indeed non-existent. The literature of the period abounds in works of the type "Programming in Basic" or "Programming in Fortran IV." Teaching was no

more than the presentation of a language, a trivial matter for these languages are remarkably poor. After that, the programming apprentice was left to his own devices. Consequently, each one became self-taught, and no transmission of competence was possible. The programmer of the eighties is no better off than the programmer of the sixties, and the accumulated mistakes of twenty years are there to prevent him avoiding the many programming traps into which he will fall.

All this was done in an entirely empirical fashion. (I hardly know why I use the past tense. If it is true that a few universities now operate differently and that a small number of industrial centres have at least understood the significance of new methods and adopted them, how many more practitioners go on in the same old way! More seriously, how many teachers, not to say recent authors, remain faithful to what cannot in any way be described by the name of "method.") The programmer sketches out the work to be done and makes a schematic representation of his program, setting out all the branch instructions, in the form of a flow-chart. Then he writes out the instructions, and begins the long grind of program testing. The program is put into the computer, and a compiler detects the syntactic errors. Some sophisticated compilers now even detect semantic errors (chiefly errors concerned with data types). The program is corrected and eventually accepted by the compiler, but that does not yet mean that it is correct. It still has to be made to run with test data. If the results appear correct, the program is declared to be correct. Otherwise, the program is examined for any apparent abnormalities, and modified to prevent their unfortunate consequences. It is the method called "trial and error" or, in the professional jargon, "suck it and see." I have seen it as practiced by a research worker in biology. His Fortran program was causing problems, and he had, by program-tracing (printing instructions as executed, with intermediate values), finally isolated his error. He then told me with great satisfaction how he had put it right. His program began with the declaration of two arrays:

```
DIMENSION A(1000),  B(1000)
```

One loop involving the array A was running beyond the array bounds and was destroying the values of B. I asked him how he had found the error which caused the bounds to to exceeded, and how he had gone on to correct the loop. But his reply showed he had done neither. He had simply doubled the size of A:

```
DIMENSION A(2000),  B(1000)
```

How can anyone expect correct results from a program which is manifestly false? How can he go on to publish a paper, saying "It has been found by computation that. . ."?

It has been all too easy to denounce this lamentable state of the art [BOE]. Statistics for 1972, dealing with sums of millions of dollars, give the cost of writing as $75 per instruction, but after program testing $4000 per instruction. In its estimates at the beginning of its bid for support for ADA, the United States Department of Defense declared that if the cost of programming errors were to be reduced by only 1%, the total saving would be $25 million per annum. . . .

The answer has been sought in languages with tighter constraints. Increase control by having more data types. I must be permitted to say that I do not appreciate this line of reasoning. Let us take a comparison. Because certain reckless drivers take needless risks and so endanger the community of drivers and passengers, we need to multiply the methods of control and surveillance — traffic lights, automatic barriers at stop signs, weighing devices at the approaches to bridges, radar for speed control, ... All this translates into inconvenience for the reasonable driver, penalised for the brutishness of a few clods. Programming is going the same way. Just because my biologist does not know how to look after his subscripts and lets them exceed their declared bounds, at every call of a subscripted variable the run-time system will make tests to ensure that there is no error, and too bad for me if I have not made an error. These tests cost computing time, which costs money. I am paying for fools.

The other cure, the one in which I believe, lies in teaching. The programmer can be taught to work properly, to write correct programs, which he knows are right, and why.

It is difficult to say who first launched this idea. "Notes on Structured Programming" of Edsger Dijkstra is, in the view of many, the first important text in this domain [DI1]. "Testing a program can show only that it contains errors, never that it is correct. . . ."

Many research workers have addressed themselves to the problem of the creation of programs, and we shall not attempt here to give an exhaustive bibliography, being unable to do so and unwilling to risk being unfair. But two works stand out along the way. In 1968, Donald Knuth began the publication of a series of books entitled "The Art of Computer Programming" [KNU]. In 1981, David Gries published "The Science of Programming" [GRI]. From art to science in ten years! As far as we are concerned, the most important step has been the introduction of inductive assertions by R. Floyd [FLO] in 1967, the system axiomatised by Tony Hoare [HOA] in 1969. At first this was regarded as a tool for proving the correctness of programs. But experience has shown that it can be very difficult to prove a program written by someone else.

Assertions have been used, therefore, as the basis of a method of program construction [AR1] [AR2] [GRI]. This method will be recalled in the first

chapter, below. We presented it first in 1977, and again in a book in 1980, trying to avoid as far as possible all mathematical formulation; that book was intended primarily for school teachers, some of them certainly literate in the humanities, and it was essential that its language should discourage none of them. David Gries, on the other hand, regards it as essential to formulate the assertions in terms of first-order logic, and discusses the relationship between this and the logic expressible in natural language with AND and OR. We shall say a few words here about this point. Because we have no wish to write a book of mathematics, we shall as far as possible keep to assertions expressed in natural language.

The program construction method is founded on recurrence. "Suppose that I have been able to solve the problem up to a certain point. . . ." If that is the end, we stop. Otherwise, we move a step nearer the solution by going back to the recurrence hypothesis. Then we look for a way to begin. In other words, and in simplified, schematic fashion, suppose I have been able to compute $f(i)$. If $i = n$, and $f(n)$ is required, the work is finished. Otherwise, I compute $f(i + 1)$ in terms of $f(i)$, then I change $i + 1$ into i and start again. Finally I compute $f(0)$.

It is remarkable that the same reasoning, with little or no modification, will produce a recursive procedure. We shall give in the second chapter a certain number of examples of procedure construction for recursive functions. They do not use the assignment instruction, and so their programming style is very different. It is nearer to axiomatic definition than to computing strategy. It is computed through a very complex execution mechanism, but also gives the possibility of deduction from the properties of the recursive definition. We shall give several examples of this, particularly in connection with the complexity or precision of computations.

If recurrence is the common base of iteration and recursion, we can build it into a true programming language. It uses recurrent sequences of depth 1, and the minimisation operator. In this sense, we can say that it is constructed on the same universal primitives as its contemporary Lucid [ASH], proposed by Ashcroft and Wadge. But we would not seek to develop a formal system, and we use numerical indices for the sequences, although in fact we would call only on the successor function of natural integers. The important thing is to have a simple means of expressing recurrence, which is at the centre of the whole of this study. We shall show how a recurrent algorithm can be interpreted and transformed into an iterative program. Our aim is much less to make of this a true programming language, endowed with good data structures, and compilable by a computer, than to be able to write recurrent algorithms in a recurrent language, and to have a simple method of extracting iterative programs from them. But why this diversion, when in the first chapter we have shown how to create the iterative program directly? One of

the reasons is that this recurrent language has no assignment instruction, and so makes possible substitution, and hence the whole of algebraic manipulation. This allows a recurrent algorithm to be transformed easily into the form best adapted for translation into a good iterative program. We shall give several examples of this.

This language proceeds at the same time from recursion, for like that it is not founded upon assignment, and from iteration, for like that it shows sequences of values to be computed to reach a result. It is thus not surprising that it can serve as an intermediary between recursion and iteration. We shall show how a recurrent program can be created from a recursive function definition. This will allow us to expose for examination the execution mechanism of a recursive procedure, to study in depth the concept of the stack, not laid down *a priori*, but deduced from the recurrent scheme, and to discuss the further significance of this. We shall show afterwards how stronger hypotheses allow simplication of the iterative form, making the stack redundant in some cases, and often leading finally to the reduction of a program to a single loop.

Because recursion and iteration are two expressions of the same recurrence, there must exist a simple transition from one form to the other. We shall show in Chapter 5 how we can pass directly form the recursive definition to an iterative program. A generalisation of the recursive definition gives the recurrence hypothesis on which the iterative program can be constructed. A substitution, a little algebraic manipulation, and then a unification lead to the body of the loop. This method is extremely powerful when it can be used. Thus it provides a true mechanism for program synthesis. Beginning with a recursive definition which describes the function, but says nothing as to how it can be computed, we obtain an iterative program which exhibits a computing strategy.

This method is valid only for recursively defined functions. For sub-programs, other tools are needed. Just as recurrent sequences have played their part as the hinge between recursion and iteration, so regular actions, presented in Chapter 6, take on this double aspect, both iterative and recursive. They can be interpreted as segments of a program which has branch instructions, but also as recursive procedures without formal parameters or local variables. In particular they are amenable to substitution (the replacement of a procedure name by the procedure body), to identification (an expanded form of identity between two actions), and to the transformation of their terminal recursion into iteration. We show how this can be used to formulate the program representing an automaton.

It can also be used for program transformations. We give in Chapter 7 three frequently used syntactic transformations, which do not depend on the program's semantics and which do not modify the sequence of computa-

tions defined by the program. But we may well need transformations acting on this sequence if we are to modify, principally to shorten, it. We give some transformations which depend only on local properties. With these tools, we can build up more complex transformations or operate upon programs, for example to make their termination become clear or to pass from one strategy to another.

Regular actions and syntactic or local semantic transformations allow operations on iterative programs. To operate on recursive sub-programs, we need a more powerful tool. Thus we introduce generalised actions, and show how they can be regularised. This allows us to pass from parameterless recursive procedures to iterative procedures, if necessary through the introduction of an integer variable. To operate upon procedures with formal parameters and local variables, we must first make semantic transformations replacing the formal parameters and local variables by global variables. We give a first example in Chapter 8.

But that is concerned with the most powerful tool we have for operating on programs, and its presentation leads us to develop some examples which are both longer and apparently more sophisticated. We delve more deeply into the study of the replacement of formal parameters and local variables by global variables. We try to spell out the choices which are made during the transformation process and to demonstrate their importance.

For we end up, in fact, with a new method of programming. We create a recursive procedure; by semantic transformations we pass to global variables; by transformations into regular actions, and their subsequent manipulation, we obtain an iterative program which often seems to have very little left in common with the initial procedure. During a meeting of an international working group at the University of Warwick in 1978, where I presented this mode of operation, Edsger Dijkstra strongly criticised it, comparing it with the analytical geometry of Descartes. For him, it is worse than useless to spend hours in covering pieces of paper with tedious computations, just to get in the end a simple program which a little reflective thought would have revealed directly. He thinks that it is a waste of time for me to teach methods of computation which apply to programs and that I would do better to teach people how to think and how to ponder.

I am sensitive to this criticism. It has truly become an obsession with me—how does one invent a new program? How do you get a simple idea which turns into a simple program? I have not for the moment any answer, and I am afraid there may not be one. Human beings have been confronted with the problem of creativity for thousands of years, and we still have not a chance of being able deliberately to create anything new. Nonetheless, the constraints of computer technology have thrust the assignment instruction on us, and obliged us to invent new forms of reasoning. Recurrence is an old

form of reasoning; iteration and recursion are creations of informatics. This book tries to shed some light on their relationships. Will this result in new ways in matters of program creation, and thence, in problem solving?

For me, analytical programming is a new path of discovery. From simple premises, and by computational methods which hardly change from one example to another, I can obtain a simple program in which what I am looking for—a simple strategy—is to be found. In his book, David Gries says that when we have obtained a simple program in this way, we should forget the twisted path by which we have come, and look for the straight road which leads directly to it [GRI]. It would indeed be a good thing to have only a simple way of producing a simple result. But I do not think that this would necessarily benefit the student or programming apprentice. He may well be filled with admiration for the master, but is it not a deceit to be made to believe that the master has an excess of inventive genius when, on the contrary, it is computation which has been the principal instrument of his success?

In this book, we should like to illustrate the mechanism for the creation of programs by computation, that is, "analytical programming." We have not hesitated, in developing the working, freely to jump over several clear but tedious steps from time to time. We immediately ask the reader on each of these occasions to do the computations himself, if he wishes to make the most of what is in front of him.

Henri Ledgard [LE1] has recognised good style with his programming proverbs. He might have included, from Boileau: "Qui ne sut se borner ne sut jamais écrire" ["No one who cannot limit himself has ever been able to write"]. A cruel proverb, and I have suffered a lot here. There are so many extraordinary examples of program transformations, revealing altogether unexpected strategies. . . . How to choose the best of them? Where to publish the others? I have tried to be content with examples not requiring too much computation and yet producing spectacular results. I have tried hard not to reproduce the examples of my previous book, still valid even if I have abandoned its notation as too unreadable and my methods of computation rather too sketchy [AR2]. The reader may wish to refer to it if he needs further examples.

If he learns from this book that recurrence is the foundation of methodical informatics, that recursion is an excellent method of programming, and that there are in general simple transitions from recursion to iteration, I shall already have achieved an important result. But I have been a little more ambitious: has the reader been convinced that computation is a reliable tool for the creation of programs?

I am anxious to warn the reader that this book revives my previous work [AR2], continuing much of its argument in terms which are, it is hoped,

more simple. Since it was written I have had much new experience, both in transforming programs and in teaching students. This new book brings all these things together. I hope the result is rather more readable. . . .

Finally, exercises have been given, because it is not the result which is significant, but the way in which it is obtained. This aspect needs to be developed completely—but to do that I shall have to write another book.

List of Programs Discussed in the Book

(Numbers in parentheses are chapter and section numbers.)

Exponentiation

Computation of x^n, x real, n natural integer: An incorrect program (1.1), discussed (1.4), corrected (1.4). Recursive forms (2.2.2), precision (2.5.2), complexity (2.5.3). Computation (2.3), transformation to iterative (4.2).

Factorial

Computation of $n!$: recursive form (2.2.1), aberrant form (2.6.1). Variously derived iterative forms (4.6.1, 4.6.2).

String reversal

Mirror image of a character string, NOEL giving LEON: Recursive definition (2.2.3), its nature (2.4), complexity (2.6.2). Transformation to iteration (5.3.1), added strategy (5.3.1). Application to the representation of an integer in binary; conversion of an odd binary integer into the integer whose binary representation is the image of that of the given number; recursive and iterative forms (5.5).

Conversion of an integer to base b

Regarded as a character string: recursive definition (2.2.4), property (2.5.1). Regarded as a decimal number: beginning with $n = 5$ and $b = 2$, write 101 (the number one hundred and one, not the string one, zero, one). Recursive definition (4.8.2), transformation to recurrent (4.8.2), to iterative (5.3.2).

Fibonacci sequence

Dyadic recursive definition (2.2.5): computation (2.3.2), complexity (2.7), transformation to monadic recursive procedure (2.7), complexity (2.7), recurrent form (3.6.3).

Product of two integers

Dyadic recursive form (2.2.6): transformation to monadic (4.8.1), then to iterative (4.8.1). Other iterative forms (9.2). Recurrent form from another algorithm (3.6.1), associated iterative form (3.8.2).

Hamming sequence
Sequence, in increasing order of integers with no other prime factors than 2, 3, or 5:

2 3 4 5 6 8 9 10 13 15 16 18 20 24 25 27 · · ·

Recursive definition (2.7), iterative form (5.7).

"Function 91"
A dyadic recursive definition which computes a constant: Definition, value, complexity (2.8.1).

Christmas Day
Recurrent algorithm giving the day of the week on which Christmas Day falls in a given year (3.1).

Perpetual calendar
Gives the day of the week corresponding to a specified date (3.2).

Integral square root
Recurrent algorithm and improved forms, associated iterative forms (3.9). Iterative form operating in base b by subtraction of successive odd numbers (3.10).

Euclidean division of integers
Recursive form (4.1.1), associated iterative form (4.3).

Addition of integers by successor and predecessor functions only
Recursive and associated iterative forms (4.7.2, 4.7.3).

Longest common left-prefix of two integers in base *b*
In base b, the two integers are represented by character strings; the longest common prefix (leftmost part) of the two strings is taken as representing an integer in base b, the required result (5.4).

Greatest common divisor of two integers
Recursive form based on Euclid's algorithm and associated iterative form (5.4), automatic change of strategy (7.9).

Happy numbers
Sequence of integers obtained from a variant of the sieve of Eratosthenes, by crossing out at each stage only numbers which have not yet been crossed out. Complete elaboration of an iterative form, improvement, and comparison with another form (5.6).

Equality of two character strings apart from blanks
Automaton (6.1), change to an iterative program and improvement (6.8).

Notations

These are not intended to be learnt by heart, but only, should the need arise, to be consulted in cases of doubt.

*	Multiplication sign
:	Integer quotient (only for natural integers)
/	Real division
!	Sign for concatenation of character strings giving the string resulting from placing the operand strings end to end
↑	Exponentiation, $x \uparrow n = x^n$
$a[i \ldots j]$	Sequence $a[i]$, $a[i+1]$, . . . $a[j]$, empty if $j < i$
$a[i \ldots j] < c$	Equivalent to $a[i] < c$ AND $a[i+1] < c$ AND . . . $a[j] < c$
AND, and	Forms of the boolean operator giving the value TRUE if and only if the two operands both have the value TRUE
OR, or	Forms of the boolean operator giving the value FALSE if and only if the two operands both have the value FALSE
sbs(c, i, " ")	Sub-string taken from the string c beginning at the character in position i and finishing at the end of c
sbs(c, i, j)	Sub-string taken from the string c beginning at the character in position i and comprising j characters
sbs(c, i, i)	Special case of the foregoing: the character of c in position i
"<Sequence of characters not including a quotation mark>"	A constant string whose value is the string enclosed between the quotation marks
" "	Empty constant string
pos(c, i, d)	c and d are character strings; the value of this function is 0 if d does not occur within c with its