Chapter 1.1

Programming, Programming Languages and Programming Methods

C. Pair

Centre de Recherche en Informatique de Nancy, BP 239, 54506 Vandœuvre, France

Abstract

A 'program' has meant many diverse things at different times. The oldest view is that programming is describing calculations; starting with the early languages such as Fortran and Basic, improved languages have been developed, in which the difficult GOTO constructions have been avoided and a method of top-down structured programming has been supported. The top-down method has certainly increased the safety of programs, but it has never given a clear description of how calculations should be broken down into smaller problems. A second view presents programming as defining functions. A program is a chain of functions which each build an object intermediary between input and output, and the act of programming becomes a matter of progressively enlarging a library of functions. A number of versions of functional programming are sketched. In the most recent view, programming is presented as defining and treating objects, which combines the two previous views: breaking down the calculations to obtain an algorithm, and representing intermediary objects. A program defines inter-relationships between objects, and one new style of programming is progressively enriching not just an unstructured library of functions, but a universe of objects with well-defined relationships.

Psychology of Programming
ISBN 0-12-350772-3

1 Introduction

At first sight programming seems a very straightforward activity; however it is always deceptively so. Debugging programs takes time, there are always errors and it is a painstaking task to track them down and correct them. Programmers have all experienced this; if they have not found any solutions as such to the problem, they have nevertheless worked out ways of playing it safe. The psychology of programming (Hoc, 1982) studies these strategies.

The computing community also has its own collective history on this subject. At the end of the 1960s, a crisis emerged in software, programming and languages. Methodology, top-down design, structured programming, functional programming were mentioned for the first time. Twenty years later these topics continue to be discussed. New types of languages – new compared to Fortran or Basic – have appeared or have been rediscovered: Lisp, Prolog, Smalltalk, etc. Will they solve all the problems? That remains to be seen. If one is to come to terms with the issues at hand, it is necessary to define what a program is. Indeed, the idea of what a program is inevitably affects programming, the more so as it is reflected in the languages as well as in the methods that are available and are in use in education, and in the world of programming aids.

This chapter aims to point out the prevailing conceptions of the notion of 'program' in computer science, the way they are worked out, and the languages and the methods they have generated.

2 What is a program?

The first answer is *syntactical*: a program is a text constructed according to certain grammar rules. This point of view has long prevailed both in teaching (teaching the forms of instructions one after the other, instructing where not to forget a comma, and so on) and in research (the first well-written chapter of computer science was probably the theory of languages, on which Chomsky had a more lasting influence than in the study of natural languages; the compilers are organized around the syntactical analysis, etc.). In fact, the syntactical viewpoint remains predominant for beginners – it is a necessary stage, and cheaper training ends there.

It is obviously not enough. It is useless to know a language if one does not understand what it expresses, the meaning of its sentences, its *semantics*. So the question is: what does a program express?

An opinion poll conducted on this subject would probably reveal that in the eyes of most people a program describes a calculation. Programming, we know, came into existence at the same time as the computer, and the one aim of that tool is to calculate. Of course the word 'calculate' has to be taken in a wider sense: printing, drawing graphs, interpreting data transmitted by sensors, giving orders to a robot, consulting a dictionary or a file, all these are forms of calculation. It is therefore possible to define a calculation as a sequence of changes in the state of the machine (and particularly of its memory).

In fact, it is not correct to say that a program describes *one* calculation; in the majority of cases the calculation worked out is based on external inputs. So a program generally expresses a whole *set of calculations*, most often infinite – or else a function linking a calculation to each possible input.

But of what use are these calculations, and what do they express in themselves? It is sometimes the case that the calculation is the end in itself, for example, if it controls cartoons, or a game, or (more rarely) a robot; i.e. one is interested in all or some of the stages through which the machine, or a mechanism it controls, passes. But mostly it is not the calculation that is important but its result. The person using the computer is not really interested in the stages; what he is interested in is the outcome.

From the customer's point of view, the program leads from inputs to results; it expresses – to use a mathematical expression – a function : input $\rightarrow$ result.

This then is a second model of the notion of program. If the first one – the set of calculations – can be called *imperative* or *procedural* (because it expresses *how*) this second model can be called *declarative* or *definitional* (because it expresses *what* – what one expects, what has been asked for or specified).

The choice between these two points of view definitely influences the activity of programming. Programming consists in transforming a *specification*, which describes a function, into a program, that is to say, a text which can be interpreted by a machine in order to calculate this function. Does proceeding from the specification of the problem to the program involve a mental image of all the calculations, a mental execution strategy (see Hoc, 1983; Chapters 3.1 and 3.3)? Or is it possible that only definitions of functions are involved?

It would seem, indeed, when one watches most beginners (at any rate), that one can answer the first question positively and, therefore, the second one negatively: possibly because the model is dictated by the tool; or possibly, on the contrary, because the programmer refers to the way a human being would work it out; or because the notion of function seems abstract. This is not without drawbacks and leads to errors, notably because it is not easy to have a mental image not only of *one* calculation but of a generally infinite *set* of calculations.

It is possible to react in two ways to this discovery, and this has been done by researchers and programming tutors:

i. to accept going through *all* the calculations and to see how best to help it run smoothly;

ii. or to refuse it and try to train people to avoid it.

In both cases tools are used: languages, methods, software aids.

3 Programming is describing calculations

Historically, this is the first way as it is the closest to the tool: a calculation is a sequence of changes of state. Each possible change of state will therefore be described by an instruction (modify the value of a memory word), and a calculation by a series of instructions. This is how 'machine languages' operate. In fact, it is not one calculation that has to be described, but a set of calculations which is generally infinite and which yet has to be described in a finite manner. To this end one invents the jump (or *go to*); it must be *conditional* to be able to give a good description of an (*infinite*) set of finite calculations and not a single infinite calculation.

The first algorithmic languages (Fortran, but also later Basic) are not founded on very different ideas, the only difference being that the memory words are represented by symbols: variables if they contain values, labels if they contain instructions.

Programming generated by these conceptions consists in considering the first instruction of the calculation, then the second, then the third. . . and in saying from time to time: at this point let's start again (see Chapter 2.4). It is programming at 'grass roots', setting everything at the same level; however, this is quite normal for beginners since it transposes the way of performing the task 'by hand'. A flowchart expresses it well.

Computer scientists came to realise, however, that jumps produced errors. It is easy enough to understand why: one is thinking of a calculation and, suddenly, there is a change in levels to designate a point in the program; what is more, the link with the described calculations becomes obscure as soon as the references cross (see Green, 1980). There is also collision in this case between the semantic aspects (calculations) and the syntactic ones (point in the program), and it is well known that such collisions are the source of misunderstandings and paradoxes; for instance, the one concerning 'the first word of the English language, in alphabetical order, which cannot be defined in under twenty words', but which has just been defined in eighteen.

In short, towards the end of the 1960s the programming reformers (Dijskstra, Hoare, Wirth *et al.*) rejected in a sometimes slightly dogmatic manner the *go to*; and the more modern algorithmic languages (Pascal, Ada, etc.) have turned it into an object of secondary importance by providing ways of dispensing with it.

Which ways? Ways which make it possible to describe a calculation, or rather a set of calculations, differently than at the 'grass roots' level, instruction after instruction. So the calculations have to be broken down into parts which are given names:

program: part 1; part 2; part 3

The next stage consists in breaking down the parts, on and on, until one reaches elements which are immediate to program: in this way a structured tree-shaped analysis is obtained. With this method (*top-down* or *structured programming*, see Chapter 3.3) details are only meant to be considered progressively. And within the program, each part can give rise to a procedure, particularly if the language allows for one procedure to be inserted into another.

This step-by-step breaking down procedure, however, only allows for the description of a set of calculations if a conditional statement is introduced:

if *condition* **then** *part 11* **else** *part 12*

To solve the problem of describing infinite calculations, two fairly equivalent tools are available:

* iteration, of the type:

  while *condition* **repeat** *action*

  (more powerful than Fortran's **do** where the number of repetitions is fixed);

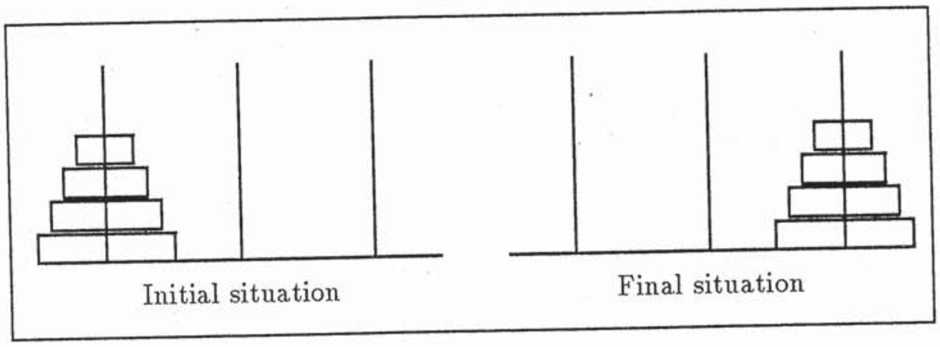

Figure 1:

* recursivity, which appears as a natural consequence of the top-down decomposition process: one stops decomposing not only when one reaches an 'elementary' part or an already known and programmed part, but also when one comes upon the very problem that one is programming.

When, however, this recursivity is presented in the context of the imperative top-down programming and of the algorithmic languages, it seldom goes down well and one is left with a feeling of uneasiness (see Chapter 2.4). One of the reasons is that one generally does not find the exact problem which is being analysed, but a similar one, with slightly different data. So the two problems need to be unified into a more general problem by the introduction of parameters. This makes the description more complex and changes its level.

Besides, when writing a recursive procedure, is one in fact still giving a description of calculations? Let us examine, for example, the well-known problem of the Hanoï towers: the goal is to move n pieces from a *starting* post to a *finishing* post using a third post as an *intermediary* (see Figure 1); at each step the top piece of one of the posts is moved, but may not be set onto a smaller one. The recursive analysis consists in writing that the goal can be reached by:

* passing $n-1$ pieces from the *starting* post to the *intermediary* post using the *finishing* post (similar problem);
* then moving the number n piece from the *starting* post to the *finishing* post;
* finally passing the remaining $n-1$ pieces from the *intermediary* post to the *finishing* post using the *starting* post.

The procedure obtained is the following:

```
procedure h (n, starting, finishing, intermediary);
            if n > 0 then h (n−1, starting, intermediary, finishing);
                          move (n, starting, finishing);
                          h (n−1, intermediary, finishing, starting)
```

But does this not amount to a definition of the set of calculations, by a property defined as a function of a certain number of arguments?

Even more so than for procedures, the question is crucial for the programming of functions; for instance, to calculate rapidly a power a^n, it is possible to use the fact that it is the square of $a^{n/2}$, at least if n is an even number; more precisely, one obtains the function:

```
function $p(a,\ n)$ : if $n = 0$ then $p := 1$
                      else if $even\ (n)$ then $p := p\ (a,\ n,\ div\ 2) \neq 2$
                      else $p := p\ (a,\ n,\ div\ 2) \neq 2 * a$
```

(*div* yields the integral quotient and $\neq$ stands for the raising to a power).

This function definition is an *equation* verified by the function p:

```
$p(a,\ n) =$ if $n = 0$ then $1$
            else if $even\ (n)$ then $p\ (a,\ n\ div\ 2) \neq 2$
            else $p\ (a, n\ div\ 2) \neq 2 * a$
```

In fact the notion of function in algorithmic languages is a hybrid: describing calculations or yielding a value? The fruit of this strange coupling is the famous 'side effect' which, during the computation of a function, also modifies the state of the memory.

Finally, it must be added that even if structured programming has represented an important step towards greater safety, it does not provide any lead concerning the central question: how does one break down a problem?

4 Programming is defining functions

In many cases, when one examines the breaking down of a problem in structured programming, one notices that each part builds an object, intermediary between input and result.

Consider the example of linear regression: input, values of n experimental measures for variables $x_1, ..., x_p$ (explaining variables) and y (explained variable); output, the coefficients $b_0, b_1, ..., b_p$ expressing by the least-squares method y as a linear combination $b_0 + b_1x_1 + ... + b_px_p$.

The mathematical study shows that the calculation can be done in the following manner:

construct the matrix M of the measurements $(x_{ij}y_i)$ with $x_{i0} = 1$

calculate the transposed T of the partial matrix obtained by removing the last column from M

calculate the product P of T by M

triangulate the linear system of matrix P

solve the system

write the coefficients

Or more precisely:

$M =$ *construction (input)*

$T =$ *transpart (M)*

$P =$ *product (T, M)*

$A =$ *triangulation (P)*

$B =$ *solution (A)*

result $=$ *writing (B)*

In fact these are *definitions* obtained by introducing *functions* which have to be described as well. It is actually possible to give these definitions in an arbitrary sequence (for example, by starting from the result and introducing the intermediaries which seem useful), since a calculation sequence automatically emerges from the linking of intermediaries; consequently it does not need to be specified. But the notion of calculation is altogether forgotten: the style obtained is purely algebraic. This can be referred to as *declarative* or *functional programming.*

The earliest attempt at promoting this style of programming through a language is Lisp (McCarthy *et al.*, 1962; McCarthy, 1978), a language from the same period as Fortran, and long before Pascal. Although knowledge of it for a long time was not widespread, perhaps because of its syntax and its rank obstinacy to do everything starting from a very small number of concepts, and also because of the few applications requiring at that time such programming; but it has now become, thanks in particular to its applications in artificial intelligence, one of the prominent programming languages. It must be mentioned, however, that pure Lisp did not survive for long and that variables, iterations, etc. were soon introduced again, but as elements of secondary importance.

Functional programming urges one, even more so than top-down imperative programming, to come back to functions that have already been programmed. What matters is less writing a large program than progressively enlarging a library of functions introduced in the most logical sequence possible: the distinction between programming and controlling a system then disappears.

Other attempts to combine the functional style of analysis with algorithmic programming (particularly concerning the notion of iteration) can be mentioned: e.g. Lucid (Ashcroft and Wadge, 1976) and deductive programming (Ducrin, 1984).

J. Backus's ideas are more radical: the idea is no longer as with Lisp to define a function by giving the expression of its result (see function p at the end of Section 3) but to consider it rather as constructed by various types of composition from bricks, i.e. elementary or predefined functions: function p, for example, would be defined as a conditional composition involving the predicate of equality, the constant functions 0 and 1, the *even* predicate, the *square, div, multiplication* functions. It should not come as a surprise (all one is doing is, in fact, composing a program starting from library programs), but it is nevertheless a shock to our present mathematical culture, for, if Lisp does away with the notion of the computing variable of Fortran and is satisfied with mathematical variables, Backus goes as far as abandoning this concept. It is in fact the point of view of combinatory logic (Curry and Feys, 1968).

Yet another way of seeing the definition of a function is to describe a relation between its data and its result. The greatest common divider of two integers, for example, is defined by

$$\begin{aligned} y = \textit{gcd}\ (a,\ b) \Leftrightarrow\ & \textit{divides}\ (y,\ a)\ \textit{and divides}\ (y,\ b) \\ & \textit{and}\ \forall z\ (\textit{divides}\ (z,\ a)\ \textit{and divides}\ (z,\ b) \Rightarrow \textit{divides}\ (z,\ x)) \end{aligned}$$

The language used here is that of mathematical logic, more precisely that of first-order predicate calculus. And it makes it possible to describe more general relations than functions:

$$\textit{uncle}\ (x,\ y) \Leftrightarrow \exists z\ (\textit{brother}\ (x,\ z)\ \textit{and parent}\ (z,\ y))$$

This type of language can be used for the specification of problems, as was done by Abrial in the Z language: in this sense, specification is the first step towards a program. But it is also possible to go no further than specification and to shift away from the description of a calculation by limiting oneself to defining, in this type of logical language, the relations between arguments and results, leaving it up to interpretation software to discover the calculations that will lead to the result. These calculations are in fact reasonings. This is what happens in expert systems. It is also the idea put into practice by the Prolog language (Roussel, 1975; Colmerauer *et al.*, 1983). In theory, the author of the program would not need to know anything about the manner in which the interpreter will draw the deduction which leads to the results. In theory at least – reality is not so straightforward.

5 Programming is defining and treating objects

The above is flawed by a serious defect. Stressing the breaking down of calculations or the definition of functions and relations leads one to forget that calculations as well as functions deal with objects: objects mentioned in the specification of a problem; objects processed by the machine or, at a slightly higher level, directly accessible in the programming language. The latter can be numbers, strings, arrays, files; the former customers, parts, graphs, polyhedrons, logical formulæ. . . Programming is also the transformation from the latter to the former; in other words, their representation.

There are, therefore, two aspects to programming: breaking down the calculation to obtain an algorithm and representing objects. One way of dealing with this duality – the one generally adopted by beginners – consists in concentrating first and foremost on the representation of the objects, because this leads them back to a more familiar situation: treating a problem bearing on the objects of language. In fact this method is induced by most languages, and Lisp more than any other, since the types of objects handled are particularly poor: there is only one, *lists*, or more precisely, *trees*; it is true that starting from there it is possible to describe all other types, but it is not necessarily easy.

Taking successively into account the two dimensions of programming – the representation of the types then the breaking down of the calculation – does not provide any guide as to representation and one can hardly expect to achieve an efficient representation for the problem in hand. So it is better to start off with the breaking-down process, and then to choose representations which will make it possible to work out efficiently the functions and procedures brought to light by the breaking down, a process which can be repeated at several levels. For example, it is very important that the characteristics of an object which are useful in the algorithm be directly accessible in its representation: they can be fields of a record representing the object (Pair *et al.*, 1988).

This leads one to think that what matters most of all is not the objects as such, but the operations executed on them. This idea joins up with similar viewpoints in other areas: mathematics, epistemology, psychology, and even technology. In computer science, it has given rise to the notion of abstract data type: contrary to the viewpoint generated by the classical algorithmic languages, here a type is essentially characterized by its operations and to define a type is to give a packet of operations involving it; they will guide its representation. The first language that adopted this viewpoint must have been Clu (Liskov, 1975), and these ideas have since been incorporated into Ada (Ledgard, 1980).

Taking this approach, however, one soon realises that types of objects are related, particularly in a hierarchical manner: a rectangle is a kind of quadrilateral and a square is a kind of rectangle, which means that to define the class of rectangles, and the operations which will be carried out on them, it will be possible to refine the notion of quadrilateral without repeating everything that can characterize a quadrilateral. It should also be noticed that this hierarchy between types is not in itself different from that which connects one object with its type: a type shows properties valid for all its objects, each one of them being a particular variation of it. It is also possible to link it to psychological notions such as the prototype, i.e. an object of a type from which others can be obtained by modifying certain characteristics. This can be seen as a tendency to approximate a 'naive' logic, whereas functional programming leans more towards mathematics.

In programming, the hierarchy between types presents similarities with the hierarchy between problems and subproblems or between procedures. In the latter case, there exists a rule concerning the area of validity of the identifiers in the interleaved procedures, which allows for understatements. And the duality of viewpoints – processing and objects – allows two hierarchies, which are not generally identical, to coexist. An algorithmic language such as Pascal clearly favours the hierarchy of processing: in particular a type declaration is separated from the functions and procedures which should accompany it.

It is possible to adopt the opposite point of view and favour the notion of object. There, programming is seen as defining objects. It means enriching a universe of objects; and an object is described from other objects (e.g. its type) by specifying certain characteristics (a value, for instance).

This viewpoint generates yet another new style of programming: progressively enriching not only an unstructured library of functions, but a universe of objects in relation with one another.

Even if these ideas appear relatively new, they already have a long history – part of which can be found in Simula (Dahl and Nygaard, 1966), an adaptation for the simulation of the algorithmic language Algol 60 (Naur, 1960). They are the basis of the object-oriented languages of which Smalltalk (Kay and Goldberg, 1976) is the best known: 'object-oriented languages' not because others would disregard the objects, but because the programming process in this case is guided by the objects, their definitions and their relations.

6 Conclusion

What about the future then? Convergences appear. Object-oriented languages are generally created starting from Lisp, which becomes a 'machine language' of functional programming, on which more sophisticated languages from the viewpoint of represented objects are built. Prolog also handles a single type of object – trees again – so it would be useful to improve it by diversifying the types; that is why a certain number of attempts are being made to bring logical programming into convergence with object-oriented languages. This is because artificial intelligence continues to play a driving role in the evolution of programming and of languages. However, there will be no true artificial intelligence if data bases are not included; and one discovers here a whole universe of objects and relations from which to carry out reasoning.

On the other hand, even if programming through a mental execution strategy appears theoretically as a useless detour and a source of errors, experience shows that it is difficult to do away with it completely, and for many people the notion of calculation seems more concrete than the mathematical languages to which the declarative viewpoint leads. Object-oriented programming may render this discussion useless by working on objects close enough to the problem set for the analysis to be very simple and then doing a series of representations.

One must make distinctions, not only according to the kinds of application, but – and this is new – according to the programmers. The language used is another dimension. Formerly one used to say that the language in which a program was written was not very important: all languages used were indeed once similar, all were algorithmic languages (except Lisp which was not well known). This is no longer the case nowadays.

The development of programming languages and methods, and the teaching of them, have up to now hardly been linked to a psychological study of the activity of programming, and this can account for certain failures. To be of any use, however, psychology must go beyond the procedural aspect of programming; it must take into account those other styles which, even if they are not new, are becoming more and more important nowadays due to the variety of applications and the training that programmers receive.

References

Ashcroft, E.A. and Wadge, W.W. (1976). Lucid, a formal system for writing and proving programs. *SIAM Journal of Computing* **5**, 336-354.

Backus J. (1978). Can programming be liberated from the von Neumann style ? A functional style and its algebra of programs. *Communications of the ACM,* **21**, 613-641.

Colmerauer, A., Kanoui, H. and Van Caneghem, M. (1983). Prolog, bases théoriques et développements actuels. *Technique et Science Informatiques* **2**, 271-311.

Curry, H. B. and Feys, R. (1968). *Combinatory Logic,* vol. 1. Amsterdam: North Holland.

Dahl, O.J. and Nygaard, K. (1966). Simula, an Algol based simulation language. *Communications of the ACM,* **9**, 671-678.

Dijkstra, E.W. (1976). *A Discipline of Programming.* Englewood Cliffs: Prentice Hall.

Ducrin, A. (1984). *Programmation.* Paris: Dunod.

Green, T.G.R. (1980). Ifs and thens: is nesting just for birds? *Software Practice and Experience,* **10**, 371-381.

Hoc, J.M. (1982). L'étude psychologique de l'activité de programmation: une revue de la question. *Technique et Science Informatiques,* **1**, 383-392.

Hoc, J.M. (1983). Analysis of beginner's problem-solving strategies in programming. *In* T.R.G. Green, S.J. Payne and G. van der Veer (Eds), *The Psychology of Computer Use.* London: Academic Press, pp. 143-158.

Kay, A. and Goldberg, A. (1976). *Smalltalk 72, Instruction Manual.* Palo Alto: Xerox Research Center.

Ledgard, H. (1980). *Ada, an Introduction, Ada Reference Manual.* New York, Heidelberg, Berlin: Springer-Verlag.

Liskov, B.H. (1975). An Introduction to Clu. *In* S.A. Schuman (Ed.), *New Directions in Algorithmic Languages.* Rocquencourt: INRIA, pp. 139-156.

Liskov, B.H. and Zilles, S.N. (1974). Programming with abstract data types. *SIGPLAN Notices,* **9**, 50-59.

McCarthy, J. (1978). History of Lisp. *SIGPLAN Notices,* **13**, 217-223.

McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P. and Levin, M.I. (1962). *Lisp 1.5. Programmer's Manual.* Cambridge, MA.: MIT Press.

Naur, P. (1960). Report on the Algorithmic Language Algol 60. *Nümerische Mathematik,* **2**, 106-137.

Pair, C., Mohr, R. and Schott, R. (1988). *Construire les algorithmes.* Paris: Dunod.

Roussel, P. (1975). *Prolog, manuel de référence et d'utilisation.* Groupe d'intelligence artificielle. Marseille: Université de Marseille.

Wirth, N. (1976). *Algorithms + Data Structures = Programs.* Englewood Cliffs: Prentice Hall.