Copyright Notice

Programming as a discipline of mathematical nature.

by Edsger W.Dijkstra

In this article I intend to present programming as a mathematical activity without undertaking the arduous task of supplying a definition of "mathematics" pleasing all mathematicians, nor of defining "programming" in a way that is palatable to all programmers.

With respect to mathematics I believe, however, that most of us can heartily agree upon the following characteristics of most mathematical work:

1) Compared with other fields of intellectual activity, mathematical assertions tend to be unusually precise.

2) Mathematical assertions tend to be general in the sense that they are applicable to a large (often infinite) class of instances.

3) Mathematics embodies a discipline of reasoning allowing such assertions to be made with an unusually high confidence level.

The mathematical method derives its power from the combination of all these three characteristics; conversely, when an intellectual activity displays these three characteristics to a strong degree, I feel justified in calling it "an activity of mathematical nature", independent of the question whether its subject matter is familiar to most mathematicians. In other words, I grant the predicate "mathematical nature" rather on the "quo modo" than on the "quod".

A programmer designs algorithms, intended for mechanical execution, intended to control existing or conceivable computing equipment. These -usually electronic- devices derive their power from two basic characteristics. Firstly, the amount of information they can store and the amount of processing that they can perform in a reasonably short time are both large beyond imagination. And as a result, what computers could do for us has outgrown its basic triviality by several orders of magnitude. Secondly, as executors of algorithms, they are reliable and obedient, again beyond imagination: as a rule they indeed behave exactly as instructed.

This obedience makes heavy demands on the accuracy with which the

programmer has instructed the machine: if the instructions were to produce non-sense, the machine will produce non-sense. Inexperienced programmers often blame the machinery for its strict obedience, for the impossibility to appeal to the machine's "common sense"; more experienced programmers realize that it is exactly its strict obedience that enables us to use it reliably, to forge it into a sophisticated tool that we would never be able to build if the executor of our algorithm had the uncontrolled freedom to interpret our instructions in "the most plausible way". As a result, the competent programmer does not regard precision and accuracy as mean virtues: he knows that he could not work without them.

His work is also always "general" in the sense that each program is able to evoke as many different computations as we can supply it with different input **data.** Whenever we make an assertion about a program, it is an assertion about the whole class of possible computations that could be evoked under control of it and "designing an algorithm" is nothing more nor less than "designing a whole class of computations". So the programmer's work also shares the second characteristic.

Finally, what about the confidence level of his work? Well, it should be very high for two reasons. Firstly, a large sophisticated program can only be made by a careful application of the rule "Divide and Conquer" and as such consists of many components, say N; if, however, p is the probability of an individual component being correct, then the probability P of the whole aggregate being correct satisfies something like $P \leq p^N$. In other words, unless p is indistinguishable from 1, for large N the value P will be indistinguishable from zero! If we cannot design the components sufficiently well, it is vain to hope that their aggregate will work at all. Secondly, its confidence level should be very high if we, as society, would like to rely upon the performance of the algorithm. And we do, when we use machines for air traffic control, banking, patient care in hospitals or earth-quake **prediction.**

Now honesty compells me to admit that today, on the average, the confidence level reached by the programming profession is not yet what it should be, on the contrary!

From a historical point of view this sorry state of affairs is only

too understandable. The tradition of programming is very young and has still many traceable roots in the recent past, when machines were still rather small and programming was not yet such a problem. (Before we had machines, programming was no problem at all!) But in the last ten to fifteen years, the power of available machinery has grown at least with a factor of a thousand, thereby completely changing the scope of the programming profession. But old habits seldom die!

The old technique was to make a program and then to subject it to a number of testcases where the answer was known; and when the testruns produced the correct result, this was taken as a sufficient ground for believing the program to be correct.

But with growing sophistication, this assumption proved more and more to be unjustified until, some five years ago, it surfaced in the form of "the software crisis". One of the first considerations of what was later to emerge as "programming methodology" was this question of the confidence level: "How can we rely on our algorithms?"

An analysis of the situation quite forcibly showed that program testing can be used very convincingly to show the presence of bugs, but never to demonstrate their absence, because the number of cases one can actually try is absolutely negligible compared with the possible number of cases. The only way out was to prove the program to be correct.

The suggestion that the correctness of programs could and should be established by proof was met with a great amount of scepticism. (In the mean time, many older people had already accepted as a Law of Nature, that each program is bound to contain bugs!) The scepticism, however, was not without reason.

To start with, it was not clear what form such correctness proofs could have. You cannot build a proof on quicksand, you must have axioms, in this case an axiomatic definition of the semantics of the programming language in which the program has been expressed. It was only after a few efforts that a technique for semantic definition emerged that could serve as a possibly practical starting point for correctness proofs.

When people then tried to give correctness proofs for existing programs, the result of that effort was very disappointing: the proofs were so long, hairy and clumsy, that they failed to convince. And also this disappointment can still be traced in the scepticism of many. But three discoveries have changed the scene since then.

The first discovery was that the amount of formal labour, needed to prove the correctness of a program could depend very heavily on the structure of the program.

The second discovery was that of a few useful theorems about program constructs and thanks to them we no longer needed to go all the way back to the axioms all the time.

The most drastic discovery, however, was the last one, that what we then tried, viz. to prove the correctness of a given program, was in a sense putting the cart before the horse. A much more promising approach turned out to be letting correctness proof and program grow hand in hand: with the choice of the structure of the correctness proof one designs a program for which this proof is applicable. The fact that then the correctness concerns turn out to act as an inspiring heuristic guidance is an added benefit.

If I ended this article with the above optimistic note I could create the wrong impression that now the intrinsic difficulties of programming have been solved, but this is not true: the best I can say is that now we have a better insight in the nature of the difficulties of the programming task than a few years before. In my closing paragraphs I hope to convey this nature, at the same time sketching the intellectual demands made upon the competent programmer.

A programmer must be able to express himself extremely well, both in a natural language and in formal systems. The need for exceptional mastery of a natural language is twofold. Firstly it is not uncommon that e.g. English is the language in which the problem is communicated to him and in which he must describe his interpretation or modification of the problem. This circumstance has been a source of many misunderstandings to the extent that there is a wide-spread belief that e.g. English by its very nature is

inadequate for that communication task. I don't believe it (although sloppy English certainly is!), on the contrary: I always have the feeling that our natural language is so intimitely tied with what we call understanding that we must be able to use it to express what we have understood. Secondly, we should not close **our** eyes for the fact that formalization, in a sense, is always "after the fact" and that therefore natural language is an indispensable tool for thinking, in particular when new concepts have to be introduced. And this is what a programmer has to do all the time: he has to introduce new concepts -not occurring in the original problem statement- in order to be able to find, to describe and to understand his own solution to the problem. For instance, when asked to construct a detector, analysing a string of characters for the occurence of an instance of the -nicely formally defined- syntactic category "sentence", he may find himself led to the introduction of a completely new syntactic category "proper begin of a sentence", i.e. a string of characters that is admissible as the opening substring of a sentence but not yet a complete sentence itself. After having established that this is indeed a useful concept for the characterisation of some intermediate [redacted] of the computational process, he will proceed by manipulating the given formal syntax in order to derive the formal definition of this new syntactic category. In other words, given the problem, the programmer has to develop (and formulate!) the theory necessary to justify his algorithm. In the course of this work he will often be forced to invent his own formalism.

Such demands, of course, are common to most mathematical work, but there are reasons to suppose that in programming they are more heavy than anywhere else. For besides the need of precision and explicitness, the programmer is faced with a problem of size that seems unique to the programming profession. When dealing with "mastered complexity", the idea of a hierarchy seems to be a key concept. But the notion of a hierarchy implies that what at one level is regarded as an unanalyzed unit, is regarded as a composite object at the next lower level of greater detail, for which the appropriate grain (say, of time or space) is an order of magnitude smaller than the corresponding grain appropriate at the next higher level. As a result the number of levels that can meaningfully be distinguished in a hierarchical composition is kind of proportional to the logarithm of the ratio between the largest and the smallest grain. In programming, where the total computation may take an hour, while the smallest time grain is

in the order of a microsecond, we have an environment in which this ratio can easily exceed 10^9 and I know of no other environment in which a single technology has to encompass so wide a span.

It seems to be the circumstance sketched in the above paragraph that gives programming as an intellectual activity some of its unique flavours. The concepts he introduces must be highly effective tools for bringing the necessary amount of reasoning down to an amount that can be done. And also the formalism he chooses must be such that his formulae do not explode in length, a regrettable phenomenon that is bound to occur unless the programmer pays conscious care to measures for avoiding that explosion. A final consequence of the hierarchical nature of his artefacts is the competent programmer's agility with which he switches back and forth between various semantic levels, between global and local considerations, between macroscopic and microscopic concerns, an ability that has been described as "a mental zoom lens". This agility is bewildering for those that are unaccustomed to it.

If -and I hope that I am fair- I confront this with my impression of "the standard mathematical curriculum" (whatever that may be), I come to the following differences in stress:

1) In the standard mathematical curriculum the student becomes familiar (sometimes even very familiar!) with a standard collection of mathematical concepts, he is less trained in introducing new concepts himself.

2) In the standard mathematical curriculum the student becomes familiar (sometimes even very familiar!) with a standard set of notational techniques, he is less trained in inventing his own notation when the need arises.

3) In the standard mathematical curriculum the student often only sees problems so "small" that they are dealt with at a single semantic level. As a result many students see mathematics rather as the art of organizing their symbols on their piece of paper than as the art of organizing their thoughts.

If I have given some of my readers the first germs of the feeling that to an inventive and effective mathematician the field of programming may provide the area par excellence in which to find his challenge and bring his abilities to bear, one of my dearest wishes has been fulfilled.

Eindhoven, Technological University 23rd May 1973