On the reliability of programs.

All speakers at the lecture series have received very strict instructions as how to arrange their speach; as a result I expect all speaches to be similar to each other. Mine will not differ, I adhere to the instructions. They told us: first tell what you are going to say, then say it and finally XXXXXXXXX summarize what you have said.

My story consists of four points.
1) I shall argue that our programs should be correct
2) I shall argue that debigging is an inadequate means for achieving that goal and that we must prove the correctness of programs
3) I shall argue that we must tailor our programs to the proof requirements
4) I shall argue that programming will become more and more an activity of mathematical nature.

The starting point of my considerations is to be found at the "software failure". A few years ago the wide-spread existence of this regrettable phenomenon was established beyond doubt; as far as my information tells me, the software failure is still there as vigorous as ever and its effects are sufficiently alarming to justify our concern and attention. What, however, is it?

Depending on the specific instance of failure one chooses, it can be described in many ways. One of the most common forms starts with an exciting project, but as it proceeds, deadlines are violated and what started as a fascinating thriller slowly turns into a drama, to be played ny an ever increasing number of actors, the majority of which know perhaps their own part but have certainly lost their grasp on the meaning of the performance as a whole. At last the curtain falls, only because it is too late, but not because anything has really been completed, for the final piece of software is still full of bugs and will remain so for the rest of its days. There are other forms, but they all have in common, that it turns out to be very, very difficult to get the whole program working with an acceptable degree of reliability.

When we try to explain the present as the natural outcome of the recent past, we get a better understanding of what has happened. In the past ten, fifteen years, the power of commonly available computers has increased by a factor of a thousand. The ambition of society to apply these wonderful pieces of equipment has grown in proportion and the ppor programmer, with his duties in this field of tension between equipment and goals, finds his task exploded in size, scope and sophistication. And the poor programmer just has not caught up. Looking backwards we must conclude that the difficulty of the tasks ahead have been grossly underestimated in the past. Extrapolations concerning the number and power of computers to be installed have been made, but society's preparation for this oncoming wave of machinery has been the call for more and more programmers, rather than for more capable ones, who derive their greater capability from a better understanding of the nature of the programming task.

It is alarming to see how little the average programmer's attitude towards his work has changed. The reason is twofold: many programmers have been attracted towards the profession a long time ago, and among them many did not have the intellectual growth potential needed to keep up with the changing profession. Besides that I am afraid that in many organizations young people are attracted [illegible] as programmers, who are selected on account of obsolete aptitude tests; I still often hear that a succesful programmer should be "puzzle-minded" whereas I have the feeling that a [illegible] ~~essential~~ clear and systematic mind is more essential. A modern, competent programmer should not be puzzle-minded, he should not revel in tricks, he should be humble and should avoid clever solutions like the plague. [illegible] In the period under discussion higher level programming languages have been accepted as general programming tools and they have been hailed as a tremendous step forward. OK, but is it sufficient? Higher level programming languges enable us perhaps to cope with a factor of 10 in scope but not with a factor of 1000. In the old days, programs of one or two thousand assembly code instructions were horrors, but in the mean time highler level language programmers produce -admittedly- larger programs of exactly the same degree of unreadability and unreliability, in which the role of old machine-code tricks has been taken over by cunning higher level language tricks. Truly, I cannot see the difference... The conclusion is that, in spite of the factor thousand in scope, present day programming tries to solve its problems with the same old methods. And therefore, if we want to improve matters, we should make it our serious business to minimize the usage of what is now by far our scarcest resource, viz. brainpower. The burning question is "Can we get a better understanding of the nature of the programming task, so that by virtue of this understanding, programming becomes an order of magnitude easier, so that our ability to compose reliable programs is increased by a similar order of magitude?".

The fact that program reliability becomes a key issue is not only shown by the evidence around us, it is also quite easy to see why. A very large program is, by necessity, composed from a large number, say N, individual components and the fact the N is large implies that the individual program components must be produced with a very high confidence level. If for each individual component the probability of being right equals p, for the whole program the probability P of being right will satisfy $P \leq p^N$

and if we want P to differ appreciable from zero, p must be very close to one, because N is so large. And we shall never be able to exploit the power of computers unless we can cope with the case of very large N.

A common approach to get a program correct is called "debugging" and when the most patent bugs have been found and removed one tries to raise the confidence level further by subjecting the program to numerous test cases. From the failures around us we can derive ample evidence that this approach is inadequate. To remedy the situation it has been suggested that what we really need now are "automatic test case generators" by which the pieces of program to be validated can be exercised still more extensively. But will this really help? I don't think so.

Whe faced with a mechanism -be it hardware or software- one can ask oneself "How can I convince myself of its being correct?" As long as we regard the mechanism as a black box, the only thing we can do is to subject it to all possible inputs and checking whether it produces the correct outputs. But for the kind of mechanisms we are considering this is absolutely out of the question. I have a pet example to demonstrate this. At my University we have a machine XXXX and one should for instance like to know, whether the fixed point multiplication instruction works properly. The machine has a rather short word length of 27 bits, as a result their are only 2^{54} XXXXXXXX different fixed point multiplications possible. So, why not try them all? With 2^{14} multiplications per second, 2^{54} multiplications = 2^{40} sec = 10^{12} sec. = 10^{7} days = 30.000 years! It takes 30.000 years to have all possible multiplications performed just once. One of the consequences of this number is that in XXXXXXXXXXXXXXX the whole life time of the machine, the number of fixed point multiplications actually performed by our machine is a truly negligeable fraction of the set of possible multiplications. XXXXXXXXXXXX XXXXXXXXXXXX From a simple-minded point of view we are only interested in the correct execution of tiny set of multiplication the machine is actually called
the
XXXXXXXXX

to perform. But because in programming we think not in terms of numerical values but in terms of variables, we have abstracted from the values actually processed by the arithmetic unit and we are only allowed to make this abstraction when the multiplier would do <u>any</u> multiplication correctly. I make this point because it as often not realised that the at first sight extreme and ridiculous reliability requirements imposed by us on the hardware is a direct consequence of the fact that without it we could not afford this vital abstraction. Another consequence of the number of 30.000 years that sampling testing is hopelessly inadequate to convince ourselves of the correctness even of a simple piece of equipment as a multiplier: whole classes of in some sense critical cases can and will be missed! Al this apllies a fortiori to programs that claim to cope with many more cases and take more time for eacht single execution. The first moral of the story is that program testing can be used very effectively to show the presence of bugs but never to show their absence.

But as long as we regard the mechanism as a black box, testing is the only thing we can do. The conclusion is that we cannot afford to regard the mechanism as a black box, i.e. we have to take its internal structure into account. One studies its internal structure and on account of this analysis one convinces oneself that if such and such cases work "all others must work as well". That is, the internal structure is exploited to reduce the number of still necessary testcases, for all the other ones (the vast majority) one tries to convince oneself by reasoning, the only problem being that the amount of reasoning often becomes excessive, with the sad result that bugs remain.

This function of the mechanism's internal structure opens a new way to attack the reliability problem. Once we have seen that the confidence level can only be reached by virtue of the structure of the mechanism, that the extent to which the program correctness is not purely a function of its external specifications and behaviour, but depends critically upon its internal structure, then we can invert the question and ask ourselves "What forms of program structuring can we find, what elements of programming style and what forms of discipline, all for the benefit of the confidence level of our final product?".

Instead of trying to XXXXXXXXX devise methods to establish the correctness of arbitrary, given programs, we are now looking for the subclass of what I would like to call "intellectually manageable programs", which can be understood and for which we can justify our belief in their proper operation under all circumstances without excessive amounts of reasoning. This is done in aorder to reduce the number of testcases needed; in the case of software I see no reason at all why this approach could not be so effective that the number of testcases needed is eventually reduced to zero, i.e. that correctness can be shown a priori. Already now, debugging strikes me as putting the cart before the horse: instead of looking for more elaborate debugging aids, I would rather try to identify and to remove the more productive bug-generators!

In short, I suggest that the programmer should continue to understand what he is doing, that his growing product femains firmsly within his intellectual grip. It is my sad experience that this suggestion is repulsive to the average experienced programmer, who clearly derives a major part of his professional excitement from _not_ quite understanding what he is doing. In this treamlined age, one of our most undernourished psychological needs is the craving for Black Magic and apparently the automatic computer can satisfy this need for the professional software engineer, who is secretly enthralled by the gigantic risks he takes in his daring irresponsibility. For his frustrations I have no remedy......

We return to our question "Can the programmer arrange his activity in such a way that his growing product remains firmly in his intellectual grip, that he continues to understand what he is doing?" Well, let me state my firm belief: yes, he can, it is possible to increase our programming ability by an order of magnitude. And as a corrolary: there is no other way, for when the programmer looses his intellectual grip on his product, he will never get the program in such a state that we can rely upon it. I will now try to convey the quintessence of the considerations upon which this confidence is founded. The reliable design of a highly sophisticated program is anyway a very difficult task and we could place our question in a much broader perspective and ask ourselves "How does the human mind invent something very intricate, how does the human mind think difficult thoughts?" Fascinating as those general questions are, I shall not touch upon them, I shall restrict myself to the more limited field of programming, because already in the programming field we can distinguish five elements of mental discipline, each of which is a great help in keeping our programs understandable. In order to be able to talk about them, I must name them, I have called them sequencing discipline, operational abstraction, representational abstraction, configurational abstraction and textual encapsulation. In aorder to avoid false hopes: all these elements are known, all higher level programming languages cater more or less succesfully for some of them.

It is just that by making these elements of discipline more explicit, that we can exploit them more consciously and that we can get more of a yardstick along which to compare the respective qualities of different programming languages.

The sequencing discipline first. We should be very well aware of the fact that although the written program is the final product that leaves the programmer's hand, the true subject matter of his trade is formed by the possible computations -the maing of which he leaves to the machine- that may be evoked by his program. Whenever we make a statement about the correctness of a program it is a statement about the corresponding computations that may be evoked by it. The subject matter is dynamic, it is the happening in time, while the last thing we can lay our hands upon is the static program text. It is on account of the latter that we must be able to make assertions about the former. It is therefore essential to bridge the conceptual gap between the static program text and the dynamic computations evolving in time as efffectively as possible. In order to do this I cannot recommend too heartily to abolish the goto statement and to restict oneself for sequencing purposes to the conventional conditional clauses, alternative clauses, repetitional clauses and recursion. The goto statement has been identified as one of the combinatorial bug generators we have been looking for. I draw attention to the fact that recursion is grouped under the class "sequencing discipline", I will return to this in a moment when we have mentioned operational abstraction.

Operational abstraction is embodied in most programming languages by the subroutine mechanism. It is an explicit recognition of the fact that a total net effect can be effectuated as the cumulative effect of a sequence of subactions and that this can be done by virtue of what these subactions do for us, as distinct from how these subactions work. In a main program calling subroutines we have a level of discourse in which we can -and should- regard the subroutines as available primitives whose net effect is known and it is in terms of this knwoledge that we can and must understand the main program. At that level of abstraction it should be of no concern how the subroutines work, that is only relevant on another, more detailed semantic level. The mixing of two such levels is one of the most common sources of program bugs. For the sake of completeness I mention that operational abstraction can also be recognized when the sequencing clauses are used, viz. in the relation between the whole construction and the statement controlled by the clause; in the latter case it is customary to represent the difference of level by indentation. And it is now clear, why recursion is treated as a sequencing discipline: in the definition of a recursive operator the level which describes how the operator works coincides with a level in which the operator is used and as a result we cannot distinguish the two different semantic levels.

Operational abstraction deals with the actions occurring in the computation and we have the two sides of the coin "What they do" versus "How they work". With respect to information stored inside the machine we have a similar coin with two sides. The only thing a computer can do for us is to manipulate but the only reason for manipulating symbols is that they stand for something else. Conceptually the program manipulates rather abstract objects, while in fact it does so in terms of a particular representation for them. It is a very common occurrence that alternative programs for the same job can be viewed as different refinements of the same algorithm, the same as long as the algorithm is expressed in terms of rather abstract values, only differing in the particular representations chosen to distinguish between these values. This is called representational abstraction; by virtue of it it is often possible that alternative programs for the same job -or as the case may be: programs for similar jobs- can share a large part of their correctness proofs. I have the feeling that representational abstraction is one of the most powerful techniques for understanding programs. Also, it seems a bit neglected. I do not know of any programming language -although it may exist- that caters for it in a convincing way. Records as structured data types do not answer its needs. With records we can introduce composite data types, OK, but the record structure chosen permeates the program using them. Also, much attention has been paid to the problem of proving formally the correctness of a program, but as far as I have seen these formal proofs, they were always tied down to the particular representation chosen in this program, because the formal proof did operate in terms of the primitive data elements. As a result changing the representation would require a completely new formal proof. It is this observation that may indicate the limits of applicability of highly formalized proof techniques. I have now mentioned two forms of abstraction; the inverse processes, viz. chosing an algorithm for an action or chosing a representation for a variable is called "refinement", and so we have operational refinement and representational refinement. We should mention that representational refinement is always accompanied by operational refinement. On the abstract level we have the unanalysed variables with actions operating upon them: when a particular representation is chosen, our original actions have to be translated in terms of algorithms operating on the components of the chosen representation.

The fourth is configurational abstraction. To a given machine we can add a standard library of subroutines. If we do so, we have in a sense rebuilt our machine, we have extended its instruction repertoire. But layers of standard software can rebuild a given machine so much more drastically that a separate name seems appropriate and I have chosen "configurational abstraction" for that purpose. I am referring to the functions of an operating system that can rebuild a single processor installation into a multiple processor installation, that can rebuild a hierarchy of storage levels into a virtual store etc. Ideally, this rebuilding of a given hardware configuration into a virtual machine serves a double purpose. Firstly it enables us to map different configurations into the same virtual machine, thereby providing a step towards program portability; secondly the virtual machine should be much more attractive to use than the original machines. To my great regret I get the impression that the second requirement is often overlooked, at least not met.

Finally we have textual encapsulation. If we consider any program of a sizeable complexity, this program should not be regarded as an object all by itself but as a member of a class of neighbouring programs, alternative programs for either the same or similar jobs. It is becoming recognized more and more that it is highly desirable that the transformation of ~~one program~~ a program into another one of the family should only affect well isolated portions of the program text and should not require adjustements scattered all through the program text. This is one of the main aspects of modularity. For some program changes this goal is very hard to achieve but the objective is so clear now that I expect it to become easier in the future. We may either invent programming languages more suited to this flexibility requirement or we shall devise more elaborate means of exploiting automatic equipment for the composition of program texts.

With the above I touched on five, what I called "elements of a programming discipline", all of them ways by which we can increase the understandability of programs. They have to do with the ease with which we can understand what a program is doing, they have to do with the feasibility of correctness proofs. It is to the pattern of such proofs that we now turn out attention.

Let me first relate them to patterns of proof for single sequential programs, because the proving techniques for sequential programs seem te be better understood than those for a bunch of parallel programs. If we have a program consisting of a sequence of statements to be executed in the order in which they are given the proving technique is fairly well understood. If the net effect of the execution of the constituent statements is given, the net effect of their successive execution can be established straightforwardly by what I have called "Enumerative reasoning", our main concern being that the amount of reasoning needed for each step does not get excessive. It is my impression that the amount of reasoning needed can be reduced when we can group subsequences into compound statements, the net effect of which allows a compact formulation. This is generally true: any sizeable piece of program, or even a complete program package, is only a useful tool that can be used in a reliable fashion, provided that the documentation pertinent for the user is much shorter than the program text. If any machine or system requires a very thick manual, its usefulness becomes for that very circumstance subject to doubt! Returning to the correctness proof: the compact formulation of the net effect of compound statements often requires the introduction of a new terminology and it is not unusual that this terminology is provided for by the mechanism of representational abstraction. So much for a piece of program text without any sequencing clauses. In the case of conditional clauses, alternative clauses, case constructions and repetitive clauses, the whole construction should be regarded as a single compound and its net effect should be formulated in such a way that it is no longer transparent that, internally, the construction is controlled by such a clause. In particular: if it is controlled by an alternative clause, the description of its net effect should be applicable regardless which of the two paths have been taken, if it is controlled by a repetitive clause, the description of its net effect should be applicable to all possible numbers of repetitions. This is because in the context in which these constructions occur, they occur on account of what they do for us and not on account of how they work. This is an application of operational abstraction possible thanks to our strict sequencing disciplines. We should make this operational abstraction very explicitly and very consciously: without it the number of cases to be distinguished between in our reasoning has a tendency to grow exponentially with the program text.

The conditional, the alternative and the case construction themselves can be understood by enumerative reasoning, the construction with the repetitive clause, however, requires a special pattern of reasoning for its understanding because it must cover all possible numbers of repetitions. Basically, mathematical induction is our only mental tool adequate for the purpose -and in the case of understanding a recursive subroutine mathematical induction is often applied explicitly- but in the case of loops the understanding often can be sppeded up by resorting to one or two theorems, thereby avoiding an explicit appeal to mathematical induction. They are invariance theorems. If it can be shown that a single execution of the repeatable statement will leave a relation between the values of some variables invariant -and this itself can be established by enumerative reasoning- and if furthermore it can be shown that this same relation is valid before entering the loop, then the said relation will also hold after termination of the repetition. This turns out to be a very powerful theorem.

I have mentioned the exploitation of invariant relations explicitly because it appears to be one of our more powerful tools when we wish to make assertions about the harmonious co-operation of a bunch a parallel programs. In a number of cases the harmonious co-operation between a number of parallel programs could be established by showing the invariance of a relation. In this relation two sorts of quantities occur, on the one hand variables, common to the programs, and on the other hand variables that are a function of the state of progress of the individual processes. The analysis of the requirements of such a proof, again, is very illuminating. Suppose that we want to establish the invariance of such a relation. As long as the variables occurring in the relation remain untouched, there values remain unchanged and the validity of the relation remains invariant. We therefore focus our attention upon all operators occurring in the totality of the bunch of parallel programs, that may influence the value of one or more of the variables concerned. Let there number be M. If all these operators leave the relation invariant, the relation will be invariant. This requires a study of M cases. But be careful! We are studying the net effect of a bunch of <u>parallel</u> programs, so in principle it is insufficient to establish the invariance of the relation under the execution of the individual M operators, but we should also establish it under the concurrent execution of all possible combinations, i.e. 2^M (-1) cases. Here again we are faced with exponential growth of the number of cases to be distinguished between in our reasoning.

As a result it makes all the difference in the world, whether one has primitives at one's disposal by means of which mutual exclusion in time can be guaranteed. And I am not sure whether all designers of multiprocessor machines, of programming languages for process control etc. have paid enough attention to this observation.

Our knowledge of how to prove the correctness of programs is far from complete but it is growing. It will have an increasing influence on the programs that are going to be produced. It is irrealistic, however, to expect all potential benefits from this approach before quite a lot of cleaning up of programming languages has been established. That such forms of cleaning up are perfectly possible has been shown by professor Niklaus Wirth from Zurich who has designed the programming language PASCAL. I quote from his introduction "The desire for a new language for the purpose of teaching programming is due to my deep dissatisfaction with the presently used major languages whose features and constructs too often cannot be explained logically and convincingly and which too often represent an insult to minds trained in systematic reasoning. Along with this dissatisfaction goes my convinction that the language in which the student is taught to express his ideas profoundly influences his habits of thought and invention, and that the disorder governing these languages directly imposes itself onto the programming style of the students. I am inclined to think that the lack of discipline and structure in professional programming style is the major reason for the present appalling lack of reliability of practically all larger software products." End of quotation. We shall never learn to write good and convincing programs as long as we identify the task of programming with the task of writing programs in FORTRAN -a programming tool which, indeed, was a great step forward when it was conceived some fifteen years ago but which, by now, should be regarded as a lower level language, as a low grade coding device. This programming tool, and the thinking habits induced by it, have grown hopelessly inadequate. As a teacher it is my job to help programmers in clearing up their own thinking. Often this is a highly rewarding activity, equally delightful and instructive for both parties. But when talking to the produce as grown in what is getting known as "the pure FORTRAN environment", I am usually baffled, for unsuspected depths of misunderstanding open themselves before my very eyes. It is well known that we don't gain aitomatically from experience, on the contrary, that the wrong experience may easily corrput the soundness of our judgement. In the case of FORTRAN, it is my impression that it's intellectually degrading influence is not commonly recognized, that too few people realize that the sooner we can forget that it ever existed, the better, as it is now too inadequate, too difficult and therefore too expensive and too risky to use.

Ladies and gentlemen, let me come to my final conclusions. Automatic computers are with us for twenty years and in that oeriod of time they have proved to be extremely flexible and powerful tools, the usage of which seems to be changing the face of the earth (and the moon, for that matter!) In spite of their tremendous influence on nearly every activity whenever they are called to assist, it is my considered opinion that we underestimate the computer's significance for our culture as long as we only view them in their capacity of tools that can be used. In the long run that may turn out to be a mere ripple on the surface of our culture. They have taught us much more: they have taught us that programming any non-trivial performance is really very difficult and I expect a much more profound influence from the advent of the automatic computer in its capacity of a formidable intellectual challenge which is unequalled in the history of mankind. This opinion is meant as a very practical remark, for it means that unless the scope of this challenge is realized, unless we admit that the tasks ahead are so difficult that even the best of tools and methods will be hardly sufficient, the software failure will remain with us. We may continue to think that programming is not essentially difficult, that it can be done by accurate morons, provided you have enough of them, but then we continue to fool ourselves and no one can do so for a long time unpunished.

Finally,let me summarize as instructed. [illegible] Reliability concerns force us to restrict ourselves to intellectually manageable programs. This faces us with the questions "But how do we manage complex structure intellectually? What mental aids do we have, what patterns of thought are efficient? What are the intrisic limitations of the human mind that we had better respect?" Without knowledge and experience, such questions would be very hard to answer, but luckily enough, our culture harbours with a tradition of centuries an intellectual discipline whose main purpose it is to apply efficient structuring to otherwise intellectually unmanageable complexity. This discipline is called "Mathematics". If we take the existence of the impressive body of Mathematics as the experimental evidence for the opinion that for the human mind the mathematical method is, indeed, the most effective way to come tq grips with complexity, we have no choice any longer: we should reshape our field of programming in such a way that their methods of understanding become equally applicable, for there are no other means.

Thank you.