Programming Considered as a Human Activity.

by

Edsger W.Dijkstra

1. Introduction.

By way of introduction I should like to start this talk with a story and a quotation.

The story is about the physicist Ludwig Boltzmann, who was willing to reach his goals by lengthy computations. Once somebody complained about the ugliness of his methods, upon which complaint Boltzmann defended his way of working by stating that "elegance was the concern of taylors and shoemakers", implying that he refused to be troubled by it.

In contrast I should like to quote another famous nineteenth century scientist, George Boole. In the middle of his book "An Investigation of the Laws of Thought" in a chapter titled "Of the Conditions of a Perfect Method." he writes:"I do not here speak of that perfection only which consists in power, but of that also which is founded in the conception of what is fit and beautiful. It is probable that a careful analysis of this question would conduct us to some such conclusion as the following, viz., that a perfect method should not only be an efficient one, as respects the accomplishment of the objects for which it is designed, but should in all its parts and processes manifest a certain unity and harmony." A difference in

attitude one can hardly fail to notice.

Our unconscious association of elegance with luxury
may be one of the origins of the not unusual tacit assumption
that it costs to be elegant. To show that it also pays to
be elegant is one of my prime purposes. It will give us
a clearer understanding of the true nature of the quality
of programs and the way in which they are expressed, viz.
the programming language. From this insight we shall try
to derive some clues as to which programming language
features are most desirable. Finally we hope to convince
you that the different aims are less conflicting with one
another than they might thought to be at first sight.

2. On the quality of the results.

Even under the assumption of flawlessly working
machines we should ask ourselves the questions: "When
an automatic computer produces results, why do we trust
them, if we do so?" and after that:"What measures can we
take to increase our confidence that the results produced
are indeed the results intended?"

How urgent the first question is might be illustrated
by a simple, be it somewhat simplified example. Suppose
that a mathematician interested in number theory has at
his disposal a machine with a program to factorize numbers.
This process may end in one of two ways: either it gives
a factorization of the number given or it answers that the
number given is prime. Suppose now that our mathematician

wishes to subject to this process a, say, 20 decimal number, while he has strong reasons to suppose that it is a prime number. If the machine confirms this expectation, he will be happy; if it finds a factorization, the mathematician may be disappointed because his intuition has fooled him again, but, when doubtful, he can take a desk machine and can multiply the factors produced in order to check whether the product reproduces the original number. The situation is drastically changed, however, if he expects the number given to be non-prime: if the machine now produces factors he finds his expectations confirmed and moreover he can check the result by multiplying. If, however, the machine comes back with the answer that the number given is, contrary to his expectations and warmest wishes, alas a prime number, why on earth should he believe this?

Our example shows that even in completely discrete problems the computation of a result is not a well-defined job, well-defined in the sense that one can say:"I have done it." without paying attention to the convincing power of the result, viz. to its "quality".

The programmer's situation is closely analogous to that of the pure mathematician, who develops a theory and proves results. For a long time pure mathematicians have thought -and some of them still think- that a theorem can be proved completely, that the question whether a supposed proof for a theorem �▇▇▇▇▇▇▇▇▇▇▇▇▇ is sufficient or not, admits an absolute answer "yes" or "no". But this is

an illusion, for as soon as one thinks that one has proved
something, one has still the duty to prove that the first
proof was flawless, and so on, ad infinitum! One can never
guarantee that a proof is correct, the best one can say, is:
"I have not discovered any mistakes." We sometimes flatter
ourselves with the idea of giving watertight proofs, but
in fact we do nothing but make the correctness of our
conclusions plausible. So extremely plausible, that the
analogy may serve as a great source of inspiration.

In spite of all its deficiencies, mathematical
reasoning presents an outstanding model of how to grasp
extremely complicated structures with a brain of limited
capacity. And it seems worth-while to investigate to what
extent these proven methods can be transplanted to the
art of computer usage. In the design of programming
languages one can let oneself be guided primarily by
considering "what the machine can do". Considering, however,
that the programming language is the bridge between the
user and the machine -that it can, in fact, be regarded
as his tool- it seems just as important to take into con-
sideration "what Man can think". It is in this vein that
we shall continue our investigations.

3. On the structure of convincing programs.

The technique of mastering complexity is known since
ancient times: "Divide et impera."("Divide and rule."). The
analogy between proof construction and program construction
is, again, striking. In both cases the available starting

points are given (axioms and existing theory versus primitives
and available library programs), in both cases the goal is
given (the theorem to be proved versus the desired perfor-
mance), in both cases the complexity is tackled by division
into parts (lemmas versus subprograms and procedures).

I assume the programmer's genius matched to the dif-
ficulty of his problem and assume that he has arrived at
a suitable subdivision of the task. He then proceeds in
the usual manner in the following stages:
I)   he makes the complete specifications of the individual
      parts
II)   he satisfies himself that the total problem is solved
      provided he had at his disposal program parts meeting
      the various specifications
III) he constructs the individual parts, satisfying the
      specifications, but independent of one another and
      the further context in which they will be used.

Obviously, the construction of such an individual
part may again be a task of such a complexity, that
inside this part job, a further subdivision is required.

Some people might think the dissection technique just
sketched a rather indirect and tortuous way of reaching
ones goals. My own feelings are perhaps best described
by saying that I am perfectly aware that there is no Royal
Road to Mathematics, in other words, That I have only a
very small head and must live with it. I, therefore, see

the dissection technique as one of the rather basic patterns
of human understanding and think it worth-while to try to
create circumstances in which it can be most fruitfully
applied.

The assumption that the programmer had made a suitable
subdivision finds its reflection in the possibility to per-
form the first two stages: the specification of the parts
and the verification that they together do the job. Here
elegance, accuracy, clarity and a thorough understanding
of the problem at hand are prerequisite. But the whole
dissection techniques relies on something less outspoken,
viz. on what I should like to call "The priciple of non-
interference". In stage II it is assumed that the correct
working of the whole can be established by taking, of the
parts, into account their exterior specification only, and
not the particulars of their interior construction. In
stage III the principle of non-interference pops up again:
here it is assumed that the individual parts can be con-
ceived and constructed independently from one another.

This is perhaps the moment to mention that, provided
I interpret the signs of current attitudes towards the
ptoblems of language definition correctly, in some more
formalistic approaches the soundness of the dissection
technique is made subject to doubt. Their promotors argue
as follows: whenever you give of a mechanism such a two
stage definition, first what it should do, viz. its
specifications, and secondly how it works, you have,

at best, said twice the same thing, but in all probability

you have contradicted yourself. And statistically speaking,

I am sorry to say, this last remark is a strong point. The

only clean way towards language definition, they argue, is

by just defining the mechanisms, becuase what they then

will do will follow from this. My question:"How does this

follow?" is wisely left unanswered and I am afraid that

their neglection of the subtle, but sometimes formidable

difference between the concepts "defined" and "known" will

make their efforts an intellectual exercise leading into

another blind alley.

After this excursion we return to programming itself.

Everybody familiar with ALGOL 60 will agree that its procedure

concept satisfies to a fair degree our requirements of non-

interference, both in its static properties (e.g. in the

freedom in the choice of local identifiers) as in its

dynamic properties (e.g. the possibility to call a procedure,

directly or indirectly, from within itself).

Another striking example of increase of clarity through

non-interference, guaranteed by structure, is presented by

all programming languages in which algebraic expressions

are allowed. Evaluation of such expressions with a sequential

machine having an arithmetic unit of limited comolexity

will imply the use of temporary store for the intermediate

results. Their anonymity in the source language guarantees

the imoossibility that one of them will inadvertently be

destroyed before it is used, as would have been possible

if the computational process were described in a von Neumann

type machine code.

4. A comparison of some alternatives.

A broad comparison between a von Neumann type machine

code -well known for its lack of clarity- and different

types of algorithmic languages may be not out of order.

In all cases the execution of a program consists of

a repeated confrontation of two information streams, the

one (say "the program") constant in time, the other (say

"the data") varying. For many years it has been thought

one of the essential virtues of the von Neumann type code

that a program could modify its own instructions. In the

mean time we have discovered that exactly this facility

is to a great extent responsible for the lack of clarity

in machine code programs. Simultaneously its indispensability

has been questioned: all algebraic compilers I know produce

an object program that remains constant during its entire

execution phase.

This observation brings us to consider the status of

the variable information. Let us first confine our attention

to programming languages without assignment statements and

without goto statements. Provided that the spectrum of

admissible function values is sufficiently broad and the

concept of the conditional expression is among the available

primitives, one can write the output of every program as

the value of a big (recursive) function. For a sequential

machine this can be translated into a constant object
program, in which at run time a stack is used to keep
track of the current hierarchy of calls and the values
of the actual parameters supplied at these calls.

Despite its elegance a serious objection can be made
against such a programming language. Here the information
in the stack can be viewed as objects with nested life
times and with a constant value during their entire
life time Nowhere (except in the implicit increase of
the order counter which embodies the progress of time)
the value of an already existing named object is replaced
by another value. As a result the only way to store a
newly formed result is by putting it on top of the stack;
we have no way of expressing that an earlier value becomes
now obsolete and the latter's life time will be prolonged,
although void of interest. Summing up: it is elegant but
inadequate. A second objection -which is probably a direct
consequence of the first one- is that such programs become
after a certain, quickly attained degree of nesting,
terribly hard to read.

The usual remedy is the combined introduction of the
goto statement and the assignment statement. The goto
statement enables us with a backward jump to repeat a
piece of program, while the assignment statement can
create the necessary difference in status between the
successive repetotions.

But I have reasons to ask, whether the goto

statement as remedy is not worse than the defect it aimed

to cure. For instance, two programming department managers

from different countries and different back rounds -the

one mainly scientific, the other mainly commercial- have

communicated to me, independently of each other and on

their own initiative, their observation that the quality

of their programmers was inversely proportional to the

density of goto statements in their programs. This has

been an incentive to try to do away with the goto

statement.

The idea is, that what we know as "transfer of control",

i.e. replacement of the order counter value, is an operation

usually implied as part of more powerful notions: I mention

the transition to the next statement, the procedure call

and return, the conditional clauses and the for statement;

and it is the question whether the programmer is not rather

lead astray by giving him separate control over it.

I have done various programming experiments and

compared the ALGOL text with the text I got in modified

versions of ALGOL 60 in which the goto statement was

abolished and the for statement -being pompous and

over-elaborate- being replaced by a primitive repetition

clause. The latter versions were more difficult to make:

we are so familiar with the jump order that it requires

some effort to forget it! In all cases tried, however, the

program without the goto statements turned out to be

shorter and more lucid.

The origin of the increase in clarity is quite under-
standable. As is well known there exists no algorithm to
decide whether a given program ends or not. In other words:
each programmer who wants to produce a flawless program
must at least convince himself by inspection that his
program will indeed terminate. In a program, in which
unrestricted use of the goto statement XX has been made,
this analysis may be very hard on account of the great
variety of ways in which the program may fail to stop.
After the abolishment of the goto statement there are
only two ways in which a program may fail to stop: either
by infinite recursion -i.e. through the procedure mechanism-
or by the repetition clause. This simplifies the inspection
greatly.

The notion of repetition, so fundamental in programming,
has a further consequence. It is not unusual that inside a
sequence of statements to be repeated one or more subexpressions
occur, which do not change their value during the repetition.
If such a sequence is to be repeated many times, it would
be a regrettable waste of time if the machine had to
recompute these same values over and over again. One way
out of this is to delegate to the now optimizing translator
the discovery of such constant subexpressions in order that
it can take the computation of their values outside the
loop. Without an optimizing translator the obvious solution
is to invite the programmer to be somewhat more explicit
and he can do so by introducing as many additional variables
as there are constant subexpressions within the repetition

and by assigning the values to them before entering the

repetition. I should like to stress that both ways of

writing the program are equally misleading. In the first

case the translator is faced with the unnecessary puzzle

to discover the constancy, in the second case we have

introduced a variable, the only function of which is to

denote a constant value. This last observation shows

the way out of the difficulty: besides variables the pro-

grammer would be served by "local constants", i.e. identi-

fiable quantities with a finite life time, during which

they will have a constant value, that has been defined

at the moment of introduction of the quantity. Such

quantities are not new: the formal parameters of procedures

already display this property. The above is a plea to

recognize that the concept of the "local constant" has

its own right of existence. If I am well informed, this

has already been recognized in CPL, the programming

language designed in a joint effort around the Mathematical

Laboratory of the University of Cambridge, England.

5. The double gain of clarity.

I have discussed at length that the convincing power

of the results is greatly dependent on the clarity of the

program, on the degree in which it reflects the structure

of the process to be performed. For those who feel them-

selves mostly concerned with efficiency as measured in

the cruder units of storage space and machine time, I

should like to XXXXXXXXXXXX point out that increase of

efficiency always comes down to exploitation of structure
and for them I should like to stress that all structural
properties mentioned can be used to increase the efficiency
of an implementation. I shall review them briefly.

The life time relation satisfied by the local quantities
of procedures allows us to allocate them in a stack, thus
making very efficient use of available store; the anonymity
of the intermediate results enables us to minimize storage
references dynamically with the aid of an automatically
controled set of push down accumulators; the constancy of
program text under execution is of great help in machines
with different storage levels and reduces the complexity
of advanced control considerably; the repetition clause
eases the dynamic detection of endless looping and
finally, the local constant is a succesful candidate for
a write-slow-read-fast store, when available.

Let me conclude. When I became acquainted with the
notion of algorithmic languages I never challenged the
then prevailing opinion that the problems of language
design and implementation were mostly a question of
compromises: every new convenience for the user had to
be paid for by the implementation, either in the form
of increased trouble during translation, or during
execution or during both. Well, we are most certainly
not living in Heaven and I am not going to deny the
possibility of a conflict between convenience and efficiency,

but now I do protest when this conflict is presented as

a complete summing up of the situation. I am of the

opinion that it is worth-while to investigate to what

extent the needs of Man and Machine go hand in hand and

to see what techniques we can devise for the benefit of

all of us. I trust that this investigation will bear

fruits and if this talk made some of you share this

fervent hope, it has achieved its aim.

--- --- --- --- --- --- --- --- -- -- -- --

Prof.Dr.E.W.Dijkstra

Department of Mathematics

Technological University

P.O.Box 513

E I N D H O V E N

The Netherlands