STICHTING
MATHEMATISCH CENTRUM
2e BOERHAAVESTRAAT 49
AMSTERDAM

REKENAFDELING

Rapport M.R.34

On the Design of Machine Independent Programming Languages

by

Dr. E.W. Dijkstra

October 1961

Introduction.

In the light of the subject matter of this report it is not surprising that a number of problems will be discussed here that also turn up regularly in connection with the so-called "ALGOL Maintenance". In order to avoid misunderstanding, however, I should like to stress that this report does not deal with ALGOL Maintenance. For instance, the semantic definition of ALGOL 60 does not specify in which order the primaries of an expression are to be evaluated; in consequence, syntactically admissible but semantically ambiguous expressions may be written down. When, in the following, I express a marked preference for semantic definitions in which, amongst others, the order in which primaries are to be evaluated is fixed uniquely, this must not be regarded as a proposal for the ALGOL Maintenance to supplement the official ALGOL 60 Report to this effect. In my opinion it is really too late for this now, when one thinks of the considerable amount of time and energy that has already gone into the construction of ALGOL 60 translators.

Instead of discussing in detail all sorts of proposals for the improvement of ALGOL 60 -or let us rather say: proposals for new languages- and judging them on their merits, I would prefer to pose (and try to answer) the question what the standards should be in judging these language proposals. This report has been written in order that we shall have in mind as clearly as possible what we are aiming at when we create a new machine independent programming language, and by which ways we could reach these goals.

I shall restrict myself to programming languages that, like ALGOL 60, are intended for the description of numerical processes. As the most important application I regard the possibility of using such a language to formulate processes in such a way that they can then be executed by an automatic computer. Naturally, such a language can be used fruitfully in the lecture room and in publications, but I prefer to regard those as secondary fields of application. I do so because the language requirements that seem to be dictated by "human consumption of texts" can easily lead us astray: I am fully aware that an algorithm of some complexity, published in ALGOL 60, is utterly unreadable to most people, but this fact does not shock me. Such unreadability is in no way caused by the usually emphasized "defects and shortcomings" of ALGOL 60; it is rather due to the not unusual (and often very sensible) superficiality of the reader, who

would prefer to be spared the overpowering number of particulars. It therefore seems a wise thing to focus our attention on the "mechanical consumption of texts". We shall mainly regard the programming language as a means of communication between man and machine, more precisely: with man as the "speaker" and machine as the "listener".

On good use of a machine.

Now, if we regard a programming language primarily as a means of feeding problems into a machine, the quality of a programming language becomes dependent on the degree in which it promotes "good use of a machine". Having an opinion on the quality of a programming language thus implies an opinion on what should be esteemed "good use of a machine". As long as our ideas on this matter diverge we shall never reach an agreement on an ideal programming language and I therefore propose to scrutinize our opinions on good machine use.

For a large group of people good use of a machine is synonymous with efficient use of a machine. And the only two criteria by which they judge the quality of a program or of a programming system are requirements of "time and space". I have a suspicion, however, that in forming their judgement they restrict themselves to these two criteria, not because they are so much more important than other possible criteria, but because they are so much easier to apply on account of their quantitative nature.

Some quotations may show that the sacrosanctity of these two criteria is a widespread phenomenon. Thus, Prof.dr. Bruno Thüring writes in "Einführung in die Methoden der Programmierung", page 65:

"Raum - Sparen bedingt Zeitverlust, Zeit-Sparen bedingt Raumverlust. Wir wollen dieses Gesetz als das "Reziprozitäts - Gesetz der Programmierung" bezeichnen. Dass die Gültigkeit dieses Gesetzes nicht auf das Beispiel des § 26 beschränkt ist (womit es ja kein "Gesetz" mehr wäre) geht aus dem rein logischen Umstande hervor, dass...."etc.etc.

(In translation: "Economizing on space implies loss of time, economizing on time implies loss of space. We shall call this law the "Reciprocity Law of Programming". That the validity of this law is not restricted to the example of § 26 (in which case it would not be a "law" at all) follows from the purely logical

circumstance that..."etc.) All of this sounds most impressive, yet it is really nothing but disguising, by pompous terminology, a triviality as a scientific theory. As soon as space and time are the only two criteria and there are two competing programs for the same problem, then they can only compete with each other if the one program is better in one respect and the other better in the other respect. Of course. The whole paragraph is equivalent to "There are two possibilities, because.... I have not thought of any others." And, alas, this narrowness of outlook is not confined to Thuring: in a book by Prof. E. Billeter, published in 1961, we find this "Reciprocity Law" quoted with full approval. In 1961 the "University Mathematical Laboratory Technical Memorandum no. 61/5: Some proposals for improving the efficiency of ALGOL 60" was published, written by nobody less than C. Strachey and M.V. Wilkes, in which they write "Our concern is largely with the production of efficient object programs. It is in this respect that an automatic programming system will ultimately be judged." And they mention their standards for efficiency explicitly in the next sentence:"....when the shoe begins to pinch, by way either of machine speed or of storage limitation,....". All in all, there is sufficient reason to call for some attention to the more imponderable aspects of the quality of a program or of a programming system.

By way of introduction, I should like to draw attention to the not unknownfact that it is impossible to prove a mathematical theorem completely, because when one thinks that one has done so, one still has the duty to prove that the first proof was flawless, and so on, ad infinitum. So much for human fallibility. 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. And let us be honest: even extremely plausible. We achieve this high degree of plausibility by a means specially designed for this purpose, viz. theorems. On the one hand, so many people have, each in their own way, derived these theorems, that there is a non-negligible probability that they do indeed follow from the axioms, on the other hand, the pretended conclusions are subject to conditions so orderly that the user's task of showing that he has applied the theorem correctly is not too cumbersome.

The programmer is in exactly the same position, since it is not possible for him to prove the correctness of his programs. And yet the correctness of the programs is of vital importance: everybody working with an automatic computer knows from sad experience that it is very easy to produce an awful lot of numbers, but he also knows that they are worthless if their correctness is subject to doubt. Instead of only staring with envy at the fabulously convincing power of the proofs in pure mathematics, it seems more fruitful to me to inquire whether we can learn from the way the pure mathematician works. He has theorems, we have subroutines. A theorem, however, is (see above) only useful if we can apply it under a minimum number of clear conditions. In the same way the usefulness of a subroutine (or, in a language, a grammatical construction) increases as the chance decreases, that it will be used incorrectly. From this point of view we should aim at a programming language consisting of a small number of concepts, the more general the better, the more systematic the better, in short: the more elegant the better.

In particular I would require of a programming language that it should facilitate the work of the programmer as much as possible, especially in the most difficult aspects of his task, such as creating confidence in the correctness of his program. This is already difficult in the case of a specific program that must produce a finite set of results. But then the programmer only has the duty to show (afterwards) that if there were any flaws in his program they apparently did not matter (e.g. when the converging of his process is not quaranteed beforehand). The duty of verification becomes much more difficult once the programmer sets himself the task of constructing algorithms with the pretence of general applicability. But the publication of such algorithms is precisely one of the important fields of application for a generally accepted machine independent programming language. In this connection, the dubious quality of many of the ALGOL 60 algorithms published so far is a warning not to be ignored.

I am convinced that these problems will prove to be much more urgent than, for example, the exhaustive exploitation of specific machine features, if not now, then at any rate in the near future.

In order to get as clear a picture as possible of the real needs of the programmer, I intend to pay, for a while, no attention

to the well-known criteria "space and time". Those who on the ground of this remark now doubt the honest fervour with which the following is written, should remember that, in the last instance, a machine serves one of its highest purposes when its activities significantly contribute to our comfort.

On the needs of the user.

I should now like to investigate those needs of the user that are not a direct consequence of his own specific problems. Somebody who only integrates ordinary differential equations will in all probability not be very interested in matrix operations but somebody else might perfectly well want to operate on quaternions, For yet another person it may be vital to be able to exercise control on the precision in which (parts of the) computation should be performed. I would like to focus my attention on the linguistic demands he may make, irrespective of "the representative problem" that always underlies the design of a particular language.

When I speak of the user of the language I mean the man that programs. Unfortunately I feel obliged to mention this explicitly, as there is a tendency to design programming languages so that they are easily readable for a semi - professional, semi - interested reader. (Symptoms of this tendency are languages the vocabulary of which includes a wild variety of English words to be used in a nearly normal sense, and some translators that even allow a steadily expanding list of synonyms and misspellings for these words. Particularly, languages designed under commercial pressure have suffered seriously from this tendency.) It looks so attractive: "Everybody can understand it immediately." But giving a plausible semantic interpretation to a text which one assumes to be correct and meaningful, is one thing; writing down such a text in accordance with all the syntactical rules and expressing exactly what one wishes to say, may be quite a different matter!

For purposes of clarification let us consider ordinary English as language, in the use of which, however, certain additional rules must be obeyed. The simplest of these may be of the following nature: words of more than 15 letters are forbidden; the total number of letters of three consecutive words may not be greater than 40; sentences of more than 60 words are not allowed; in one and the same sentance the same word may not be used twice as a subject;

furthermore a list of, say 2000, words is given, that are so rarely used that they have been forbidden for the sake of convenience, etc.etc... There is no reason to assume that these extra conditions will be detrimental to the readability of the text, and what is more, one can read and understand such a text just as well without knowing of the existence of these restrictions. But if the number of such restrictions is sufficiently large and particularly if they impose highly implicit conditions, it becomes almost impossible to construct a correct text. In the extreme case one would need a large computer with a complicated program to check whether one's text does not violate the rules!

Of course, this example was an exaggeration, but it clearly shows us the direction which we must definitely not take. We must make it as easy as possible for the user to master the language. And we can immediately indicate two ways of making this difficult. In the first place, implicit conditions for which it is difficult to check whether a given text satisfies them or not, in the second place conditions that forbid a construction with a straightforward semantic implication. A language which neatly caters for algebraic expressions but, for example, restricts the number of enclosing bracket pairs to eight, is one which I would discard. The requirement is too implicit to my taste and I do not like to burden the programmer with the extra task of counting to see whether he has exceeded the maximum depth, and this really is an extra duty because a priori there is nothing to prevent him from writing more than eight nested bracket pairs: for even then the semantic interpretation is perfectly clear.

From this one should not draw the conclusion that I am an ardent supporter of so deeply nested bracket pairs. On the contrary, for the correspondence between opening and closing bracket becomes increasingly difficult to see at a glance when the depth increases. If, however, someone wishes, under certain circumstances, to write down such a perfectly sound expression, I see no acceptable reasons from the point of view of the user to disallow this.

In exactly the same way I have not the slightest inclination to forbid, as some people suggest, type procedures with so-called "side effects" in ALGOL 60. Under certain circumstances such procedures can be very useful and perfectly natural constructions,

and I completely fail to see how it can serve the user to impose such an extra condition on his language, thereby restricting his power of expression.

On semantic definition and the need for conversation.

As already mentioned, I do not regard the supposed readability for a general reader as a valid criterion. I have good reasons for this. In human communication the "unpredictability" of those we address plays a fundamental role. If we now apply the norms of human communication to an artificial language, in which we wish to address a computer, then we ignore one of the most essential characteristics of the automatic computer, viz. the "predictability" of its behaviour.

When I ask myself what my words actually mean, i.e. when I ask for the semantics of my language, I can say nothing about it without considering the listener. Without listeners -e.g. when I deliver a monologue on an otherwise uninhabited isle- it makes remarkably little difference whether I speak nonsense or not, so little, in fact, that under these circumstances "meaning" becomes an empty concept. My utterances can only have meaning by virtue of a listener, and what is more, the reaction of my listener determines what my utterances mean.

Whether I explain something to a six-year-old or to his father, has a marked influence on the choice of my words. The limited vocabulary of the boy imposes definite restrictions on the choice of my words: there are many words that are "meaningless" as far as he is concerned and if I do not respect these restrictions my explanation will very soon (and in a very real sense) become "meaningless".

A more striking example of how the listener defines the semantics of my language will perhaps be given by a somewhat more artificial setting, viz. the writing of an article. When it has been completed one reads it over to see whether it actually says what one wanted to say. For this purpose one tries to read it as if one had not written it oneself. one invents an "average reader" and tries to play the role of this imaginary person as well as possible. And if, in reading, this imaginary person is startled by a rash conclusion, one alters the paragraph! The way in which the imaginary person reacts becomes one's norm: he determines whether something is clear or not, he defines the meaning of our text, i.e. the semantics of our language.

In this light we only know what we have said, when we have seen how our listener reacted to it; we only know what the things we are going to say will mean in as far as we can predict his reaction. However, we only know other people up to a (low!) point and in human communication every message is therefore to a high degree a trial, a gamble to see whether the other will understand us as we had hoped. As we do not master the behaviour of the other, we badly need in speaking the feed back, known as "conversation". (Testing a program is in a certain sense conversation with a machine, but for other purposes. We have to test our programs in order to guard ourselves against mistakes, which is something else than imperfect knowledge of the machine. If a program error shows up, one has learnt nothing new about the machine -as in real conversation-, one just says to oneself "Stupid!".)

We can fully master, however, the way in which a computer reacts and this is precisely the reason why addressing an automatic computer presents us with undreamt-of linguistic possibilities. Mastery of the reaction of the computer must not only be a theoretical possibility but a real, practical one, if one is to able to make full use of those linguistic possibilities. It is therefore mandatory that our machine be not prohibitively complicated. (From this point of view the way in which ALGOL 60 is defined is rather alarming. "Pure ALGOL 60" is defined by the official "Report on the Algebraic Language ALGOL 60", edited by Peter Naur, but reasonably speaking one cannot expect a user of the language to know this Report by heart. Specific implementations of the language are defined by translators etc. of a couple of thousand machine instructions, a quantity which exceeds our powers of comprehension even further.)

As the aim of a programming language is to describe processes, I regard the definition of its semantics as the design, the description of a machine that has as reaction to an arbitrary process description in this langauge the actual execution of this process. One could also give the semantic definition of the language by stating all the rules according to which one could execute a process, given its description in the language. Fundamentally, there is nothing against this, provided that nothing is left to my imagination as regards the way and the order in which these rules are to be applied. But if nothing is left to

my imagination I would rather use the metaphor of the machine that by its very structure defines the semantics of the language. In the design of a language this concept of the "defining machine" should help us to ensure the unambiguity of semantic interpretation of texts.

When we have thus defined our language, its semantics are completely fixed and its syntax - I owe this remark to Prof.dr.ir. A.van Wijngaarden- does not have a definig function anymore: we can do without the syntax as it is merely a summary of "admissible constructions", i.e. all constructions to which the machine does not produce the uninteresting reaction "Meaningless". (Such a possibility of escape is very useful for our machine, when we remember that we may feed it with an entirely arbitrary sequence of symbols. We shall return to this subject later.)

At this moment it is very definitely not my intention to give any suggestions for the design of this defining machine (i.e. for the design of a next programming language); I would rather direct the reader's attention to some properties of this machine that seem desirable to me if it is to serve its purpose.

For the sake of uniqueness I would prefer a strictly sequential machine, i.e. a machine for which at every (discrete) moment there is not the slightest doubt as to what is happening. I feel on the safest ground if this machine is conceived as consisting of a finite arithmetic unit coupled to a store that is, by definition, sufficiently large. In particular: whenever an operation has to process an arbitrarily great amount of information, it should do so in finite portions and in a well defined order.

In this respect our defining machine reflects one of the most important discoveries embodied in present day automatic computers, viz. that in the evaluation of arbitrarily complicated algebraic expressions one need not resort to an arbitrarily complicated arithmetic unit, but that this evaluation can always be performed by a finite arithmetic unit, provided that the anonymous intermediate results (now no longer produced simultaneously) can be stored until they are needed again. In other words: we can choose the strictly sequential machine without the slightes loss of generality. And as we shall require the concept of "sequencing" sooner or later anyway, I see no reason why we should

not introduce it right from the start.

Furthermore, we should be prepared to face the fact that our defining machine will become incredibly unpractical and unrealistic: it will be so wasteful of storage space and number of operations that it will hurt the eyes of every honest programmer. For, in how far does our defining machine differ from a real one that is provided with a good translator? This translator probably demands thousands of instructions and is therefore scarcely a realistic proposition as language definition. We should realize, however, that the size of the translator is largely due to the fact that the process has to be carried out as efficiently as possible (and furthermore by a machine not specially designed for this language). By disregarding all efficiency requirements and tailoring the machine to the language we can obtain a much simpler organization, so simple in fact, that it can very well be used as a means of language definition. (This must be possible; otherwise, how could we, poor humans, ever master the language?) If, on being confronted with our defining machine, a programmer now jumps up, protesting against this waste: "It can be done far more efficiently, if one..."etc., then we can be content. We have sown our seed in fruitful ground: he has accepted the challenge and has already started on the construction of his translator!

On unnecessary redundancy and optional information

There are two declarations in ALGOL 60 with a hybrid nature, viz. the switch declaration and the procedure declaration. Like all declaration, they reserve an identifier for a special sort of object but, besides, they immediately define this object and do so statically. In this respect they are comparable to the so-called "constant" declaration, which has been suggested for numerical quantities. We all know that by replacing static definitions by dynamic ones one can only gain in flexibility. Furthermore, ALGOL 60 includes the assignment statement that assigns a value dynamically but, alas, only in the case that this value is a logical value or a number. By extending the concept "assignment of a value" so that lists, statements etc. can also act as "assigned values", one can remove the value-defining function of the switch and procedure declaration. The declarators switch and procedure should then only be followed by a list of identifiers, to which suitable assignments should eventually be made. (I regard

such a modification as an improvement: the language then becomes more systematic and more powerful at the same time, as all value-relations have now become dynamic.)

If, as a next step, we regard the notorious logical expression "if B then C else D = E" as slip of the syntax, because the syntactical grouping of these symbols depends on the question whether the variable C is logical or not, then the type-declaration Boolean has become superfluous: whether it is a logical variable or not will become apparent from the way in which it is used. Finally we can omit all type indications in the declarations if we furthermore assume that there is no logical necessity to introduce the type integer (semantically it only plays a role in two minor cases, viz. in the definition of a↑b and in the implicit rounding off on assignment to an integer variable).

The array declaration is then left as the only odd case, as the subscript bounds must be specified there. Fortunately, however, the explicit specification of subscript bounds is logically speaking, not necessary: during the course of the computation if will transpire which array elements occur. We therefore omit the subscript bounds, since they can be regarded as redundant information.

Finally, we reduce the number of declarations to one; the function of this universal declarator is merely to introduce new identifiers local to the block in question.

In this way the programmer's powers of expression are increased considerably. There is no longer the slightest reason for an array to be rectangular, the triangular array, for instance, is automatically included in the language. It is no longer necessary that an array be homogeneous: some elements of an array may even be arrays again, or procedures or logical values, etc. Once the type of a variable is always defined dynamically, there is not even a reason for it to be constant in time. The power of expression is inceased as the language contains a smaller number of different kinds of elements and all kinds of artificial barriers have fallen away. An ordinary variable is nothing but a trivial example of a parameterless procedure. In short, the programmer now no longer needs to squeeze the relevant information into the rigid forms permitted by ALGOL 60.

This increase in expressive power is a practical advantage; from the linguistic point of view I think it even more important that in this way the language can be made less redundant. For: the redundancy of the ALGOL 60 declarations has two undesirable effects (even apart from the duty of inserting a number of extra symbols). As the declarations are obligatory, the user is forced to state explicitly a number of properties of the remainder of the block: the declarations lay down conditions which the rest of the block must satisfy and as such they are highly implicit restrictions. In the second place, if the redundant information is to be a vital part of the language, the defining machine must take note of it, i.e. it must detect whether the rest of the program is in accordance with it and this makes the defining machine considerably more complicated. By excluding redundant information from the language, means of contradicting himself have been taken away from the user and language designers are spared the temptation of assigning (afterwards) a special meaning to a particular contradiction (as in ALGOL 60 in which "go to" followed by a switch element may, under certain circumstances, be equivalent to a dummy statement).

As I am probably not using the word "redundant" in its official, technical sense, I should like to insert some clarification of my point of view. Our defining machine should be so complete as to react in a well-defined way to every arbitrary string of symbols presented to it. The special signal "Meaningless" may be one of its possible reactions. The concept "redundancy" only has a right of existence as long as it is not our intention to provoke this special signal "Meaningless" as the machine's reaction: as soon as we include this reaction in the set of "intended reactions" no program can sin against the language rules anymore and we must therefore regard every arbitrary text as acceptable. I assume that evoking the reaction "Meaningless" will never be our intention and our language therefore remains redundant as long as the signal "Meaningless" belong to the set of possible reactions of the defining machine. In itself I have no objection to this, I only have objections to "unnecessary redundancy" i.e. language rules that I can regard as restrictions.

I hope that my distinction between "rules" and "restrictions" is not purely emotinal. Roughly speaking, a language rule enables

me to express something, whereas a restriction prevents me from doing so. The language definition consists of a number of rules of reaction; some of these rules may under certain circumstances prescribe the reaction "Meaningless". When, however, the reaction "Meaningless" is prescribed in a situation for which the remaining rules cater, then I speak of a restriction, of unnecessary redundancy. This in contrast to a rule that prescribes the reaction "Meaningless" in a case for which the other language do not cater. Then I do not regard this rule as an objectionable restriction: it is just a consequence of the fact that we can write down a string of symbols for which we will not take the trouble to define a meaningful reaction (at the cost of who knows how many complications of the defining machine).

On behalf of the user I envisage a not unnecessarily redundant language, the semantics of which have been completely fixed by our defining machine. But now it is time for us to remember that it was also our intention that the processes described should be executed by a real computer in a reasonably efficient way.

In this connection I should like to quote from the "University Mathematical Laboratory Cambridge Technical Memorandum no 61/2: Some reflections on Automatic Programming and on the design of Digital Computers." by M.V. Wilkes the following remarks, with which I wholeheartedly agree:

> "If a small machine is used for compiling, however, it is desirable for the programmer to be able to lighten the task of the compiler by providing extra information; much of the information given in the declarations in ALGOL is of this type. I believe that, in designing future automatic programming languages, a clear distinction should be made between the thread of essential information necessary to define the program and the additional information put in to help the compiler."

This paragraph expresses exactly my own sentiments.

From a linguistic point of view it may be very attractive to formulate our process in a not unnecessarily redundant language, thus only being obliged to give the absolute minimum that is needed to define the process. But what is the translator going to do with this? I assume that the structure of many a machine is such that it

is desirable that the translator thoroughly analyzes this program and tries to detect all kinds of "special cases" of our general concepts, for example whether an array has a regular form (rectangular, triangular etc.), whether an array is perhaps homogeneous, whether a variable is always simple and never defined in the form of a procedure, whether a procedure is used recursively or not, etc. In short, the translator will search for "unused generality"with the aim of gaining something. These analyses are no child's play and furthermore, as the analysis is carried out statically, the translator must always remain on the safe side. But we can hardly speak of "good use of a computer" when the translator spends a considerable amount of time and trouble in trying to come to discoveries that the programmer could have told it as well! It may be a nuisance that ALGOL 60 arrays must be rectangular, but we should not close our eyes to the fact that a rectangular array is a fairly common phenomenon, and that the user is usually aware of its rectangularity. It is undesirable that the programmer is forced to give this extra information, but it is unwise to prevent him from inserting such additional information "for the possible benefit of the translator". I would like to call this "optional information", optional in the sense that a correct and complete program remains when it is left out.

For translator makers particularly I cannot stress enough that they actually have no right to this optional information: the whole concept is a concession to the weakness of the flesh. The quality of a translator naturally diminishes if it simply does not accept certain parts of the language or if it demands unconditionally certain forms of optional information -for then we have fallen back into the rigorous scheme of ALGOL 60-; it is also to the detriment of a translator, when the efficiency loss as result of omission of the optional information is so large that the user is virtually compelled -be it not "de jure"then "de facto"- to insert it. In this connection I should like to point out that the reactions to ALGOL 60 have shown that suggestions for so-called improvement of ALGOL 60 should fill us with great suspicion, especially if these suggestions come from unsuccesful translator makers.

The fact that this helpful information is kept outside the language improves the machine independence, because one machine will want to be helped in quite another way than another machine. The

second advantage could be that the language itself may remain up to date longer: information which is very helpful now may be of no interest at all in a number of years, when there may be more suitable machines and more sophisticated translators. It would be very sad if we were then bound by restrictions which can then no longer be justified (say the rectangularity of an array).

It is of course desirable that the possible forms of optional information be standardized. And for the making of proposals in this direction probably just as much tact and wisdom are required as for the design of the language itself. The general language may be very attractive logically and linguistically, but its practical merits may very well depend on the special cases for which we want to be able to give the translator a hint, as long as they are of interest but prohibitively difficult to detect automatically. In any case it is an advantage that the defining machine will provide a clear terminology in which we can express these special cases (in ALGOL 60, it is -see below- not clear, when a procedure is used recursively).

One final remark about the bearing of the semantic definition and the consequent task of a translator. Our defining machine incarnates a detailed prescription of how one can execute a given process as described by a text in the language, how one can compute the required result. By this we do not mean that every implementation should be an exact copy or detailed simulation of the defining machine. When, for example, the defining machine leaves no doubt about the order in which the primaries of an expression should be evaluated, then this is only with the intention of defining the answer uniquely as soon as it depends on this order. As long as it does not depend on this order, every implementer is free to change the order as he sees fit. I regard every implementation as a correct one as long as the answer is correct, i.e. undistinguishable from the answer that our defining machine would have given. In this sense, the "net semantics" of a language is only defined if we know what "the answer" is and we must include output statements as an essential factor in the semantic implication of a program. Regarded in this light, the net semantics of a program in pure ALGOL 60. which, as we know, contains no output statements, is empty. (The semantics of our language is defined by the reaction of our listener, but can we speak about his

reaction if no part of it reaches us?)

On some proposals by Strachey and Wilkes.

Those who have read the Technical Memorandum 61/5 by C. Strachey and M.V. Wilkes, quoted earlier, will not be surprised after all this, that the only one of their suggestions that attracts me is in the last section, in which the concept of optional information has been worked out in more detail. I will give a simple example. One of their proposals is:

> "Procedures shall be recursive if introduced by the declarator recursive procedure; otherwise they may be treated as non-recursive."

Considered in the light of the concept of the optional information introduced earlier by Wilkes (Technical Memorandum 61/2) it would have been more elegant to present the non-recursive procedure as the special, restricted case, and not the recursive one as the exception. A competing preposal would be:

> "In general all procedures may be used recursively. If the programmer, however, happens to know for certain that one of his procedures will not be used recursively in his program, he may state so, for the possible benefit of the translator, by inserting the prefix "nonrecursive" immediately in front of its declaration."

In passing -to underline my desire for rigorous, strictly sequential semantics- I should like to point out that I do not feel much inclined to support this proposal, not even in its mitigated form, because the question whether a procedure call gives rise to recursiveness in the object program is not answered by the language but by the implementation. Thus, in the ALGOL 60 translator developed by the Computation Department of the Mathematical Centre, Amsterdam, the call "sqrt (sqrt(x))" does not give rise to any recursiveness when the identifier "sqrt" refers to the undeclared standard procedure for the square root, but it does so in all other cases.

The authors' motivation for their proposal is very illuminating:

> "An example of unnecessary generality is provided by the requirement that all procedures should be recursive. In ordinary

> computing -as distinct from symbol manipulation- it will be found that the need for procedures to be recursive is the exception rather than the rule, and the requirement that all procedures should be recursive leads to inefficiency, since a recursive procedure is both longer and slower than a non-recursive one."

Let us assume that their observation is correct and not purely the result of the fact that until recently most programming systems did not cater for recursiveness. I hope to have made clear in the above that I regard such a statistical observation as insufficient grounds to justify the conclusion "unnecessary generality". Finally they make an appeal to the fact that "... a recursive procedure is both longer and slower than a non-recursive one." But the recursive procedure is such a neat and elegant concept that I can hardly imagine that it will not have a marked influence on the design of new machines in the near future. And this influence could quite easily be so considerable, that the possible gain in efficiency that can still be booked by excluding recursiveness, will become negligible. Personally, it will not surprise me if this will prove to be the case. To me the whole proposal shows too great a similarity to a proposal along the following lines: "As in most multiplications both factors are positive, we propose that the ordinary multiplication sign may only be used if both factors are indeed positive; for multiplication of factors without sign restriction the new operator "general mult" is introduced." Perhaps there are still machines in which a special multiplication of positive factors is executed faster than the general one; otherwise we can easily design such a machine.

The same sort of remarks can be made with regard to their proposal to abolish the "left to right precedence rule", a rule which they fortunately extend to the order of primary evaluation. I do not feel the slightest inclination to do this. The result of such shaky semantics is clearly shown at the end of the paragraph in question, where the authors write:

> "If, however, compilers become so sophisticated that they can rearrange whole sequences of statements in the interest of compiling efficient programs, it may be necessary to resort to a note ..... which, prefixed to a compund statement, would indicate that it was to be compiled in the order in which it was written."

If we read this carefully we see that it is suggested here, that the

advent of more sophisticated translators would give us the duty, under certain circumstances, of adding an extra "note" to the program, because otherwise the translator would translate something else. If the semantics of the language are well-defined, then, in my opinion, such a "sophisticated translator" is just plainly wrong. The paragraph quoted creates the impression that these authors have in mind a sort of floating semantics, that becomes more and more vague the translators should like to have more and more freedom. A disturbing picture for the future: a program being correct today, false tomorrow!

> Furthermore, these authors write:
> "The above restrictions appear to be sufficient to enable the terms of an expression to be evaluated in any order. We would, therefore, abolish the left to right precedence rule and, if further investigation shows that there are loop-holes that we have overlooked, we would seek to close them rather than re-introduce the precedence rule."

This is plain language: rather than closing the gap in the semantics they propose restrictions to prevent all circumstances in which this lack of definition matters, no matter how implicit these restrictions may prove to be. If these authors had their way, I should have few illusions left about the ease with which the eventual language could be used. Their proposals strike me as fighting the symptoms rather than the illness, as solving a minor problem at the expense of a major one.

Acknowledgement.

I should very much like to add that, wherever the opinions stated above should prove to make some sense, this could very well be the result of the numerous discussions I was privileged to have with the staff members of the Computation Department of the Mathematical Centre, Amsterdam, about these and allied subjects. They are, however, not in the slightest way responsible for the contents of this report.

It is a pleasure to express my sincere thanks to Mrs. J.M. Goldschmeding - Feringa, who assisted in the translation of this report.