Comments on "Woodenman" HOL Requirements for the DoD.

(This is a sequel to EWD514 "On a language proposal for the Department of Defense", written after reading "Strawman". Many of those comments on that earlier document are still applicable to its successor "Woodenman". I here confine myself to some further comments.)

1. Repercussions on future hardware design have not been taken into account.

If the DoD accepts as a standard a common HOL, its sheer buying power will ensure that efficient implementability of tha HOL will become an important design criterion for computer manufactures. This will then happen, if the DoD so desires or not. This implies that the DoD cannot restrict its responsibility to just serving its own software needs as it now sees them: the choice will have much wider consequences of (almost certainly) long duration, affecting the whole market place. These responsibilities have not been taken into account, and that is a serious omission.

The old view of programming was that it was the purpose of our programs to instruct our machines. With the software problems --and as a result: the software costs-- skyrocketing, a new view of that relation has emerged, viz. that it is the purpose of our machines to execute our programs! The latter, more modern and more appropriate view is hardly reflected in Woodenman, in which existing hardware --with the report's insistence on efficient implementation by existing techniques-- is too much taken for granted. Unless the HOL is designed for an ideal machine --or better: machines as we would like them to be--, the whole effort may in the long run cause great harm.

2. Simplicity and ease to learn the new language.

The report contains a discussion of simplicity versus complexity that itself is so simplified as to be misleading. I mention in connection the requirement that the new language should be "easy to learn".

It is perfectly clear that any unnecessary training burden should be avoided: programmers should not learn to fight problems that should not be there in the first place. And the report is fully justified when it points out the tremendous costs induced by grotesquely baroque programming languages. Hence the cry for a simple programming language, which, in my view, is fully justified.

To conclude that, therefore, the language must be "easy to learn", however, is a rash conclusion, for cruelly stated, that "easiness" suggests the desire to continue to design software with hardly educated programmers! And then our problems will remain with us, no matter which HOL is adapted.

The training of programmers cannot be "easy",for programming is and will remain difficult. The point is that "learning the language" should, indeed, be a minimal affair, and the major attention should be given to mastering an orderly <u>discipline</u> how to use it.

As long as we discuss the whole problem in terms of simplicity versus complexity, we may end up with the conclusion that machine code --or perhaps even a Turing machine!-- is the most "simple" code.The major point is that machine codes, the Turing machine and the most baroque higher programming languages, suffer from the <u>same</u> shortcoming, viz. the absence of such an orderly discipline for their use.

When designing a programming language the existence of a sound, orderly discipline for its use --a discipline that, by necessity shall be one of a rather mathematical nature-- should be one of one's major concerns. I stress this for immensely practical reasons. The existence of such a discipline is a prerequisite for the design of high-quality programs; it is also a very effective criterion for the decision what to include and what to exclude from one's language design! When considering a "feature", one cannot find a way of avoiding the puzzles and the conflicts, omit it... The resulting language will be "small" and "simple" as a consequence: there are not so many "features" with the necessary (mathematical) properties.

Besides the existing population of machines, also the existing population of programmers is taken too much for granted. When I read --page 37-- "Most programmers are not used to origin 0 and find it inconvenient or unnatural.", I conclude that "inconvenient" has --again!-- been confused with "unconventional", and that the report is too much oriented towards the past and the present and too little towards the future.

3. Are so many "specialized capabilities needed"?

On a number of places --e.g. pages 11 and 31-- the report accepts the not uncommon assumption that, because applications are varied and the language must be geared to the application, a great number of specialized features or capabilities will be needed. Is this true to the extent that the report suggests? I seriously challenge this opinion because
a) I do not believe it myself
b) the report does not motivate it and takes it just for granted
c) historical evidence in inconclusive, as the observable variation can also be interpreted as the obvious result of the absence of the discipline referred to above.

It will not surprise the reader that I consider the discussion on pages 11-12 "Generality versus Specificity" as rather unconclusive; page 31 even strikes me as misleading. When I read there:

> "A common language must have capability for growth. It should contain all the power necessary to satisfy all the aplications and the ability tp specialize that power to the particular application task. A language with defining facilities for data and operations will make it possible to add new application-oriented structures and to use new programming techniques and mechanisms using descriptions written entirely within the language."

I feel like reading a misleading advertisement that would be rejected by the professional code of our advertisers.

* * *

A few further, minor points. *)

4. Equivalence for real numbers.

On page 38 I read:

> "The use of equivalence is not recommended for real numbers but resultion of what equivalence means for imprecise quantities is a problem of numerical analysis not language design."

This is not true, the problem should not be left to the numerical analysis, because language design requires --and dictates-- a very precise answer to

*) In retrospect not so minor at all! EWD

to that question. The point has been settled fifteen years ago.

When I implemented ALGOL 60, I thought that --because exact equality of floating point numbers seemed to much to ask for-- I would provide a service by delivering the value <u>true</u> for the boolean expression $a = b$, when a and b were floating point values only differing in the least significant bit of the mantissa. This was one of the gravest mistakes I ever made, because it proved to be an absolute disaster, and I had to remedy the situation very quickly. The point is that such a loose equality is to weak a criterion to be of any use; for instance, the loose equality is no longer transitive. One could find $a = b$, $b = c$ and $a \neq c$ all being true at the same time! Under those shaky circumstances it is very hard, if not impossible, to prove the correctness of a program manipulating real variables. If at that time, proving programming caorrectness had been normal practice, I would never have made the blunder in the first place.

I mention this point for two reasons: firstly it settles a point that the report has erroneously left open, secondly it illustrates my point, made above, that the requirement of an orderly discipline indeed settles design questions.

<u>5. The range</u>.

I read on page 43:
"The source language should require its users to individually specify the range of values for integer and floating point variables [...] Range [...] specifications should not be interpreted as defining new types."

The latter addition seems in contradiction with page 33 "By the type of a data object is meant the set of objects themselves, the essential properties of those objects and the set of operations which give access to and take advantage of those properties."

There seems to be a confusion, it is not clear what the implications are when the integer variable x has the range declared to be from 0 through 15. There are at least two possibilities:

a) any implementation has to check that the value of the intermediate variable x always lies within that range from 0 through 15.

b) no implementation needs to allocate more than 4 bits to the variable x ; any implementation has the right, <u>but never the obligation</u>, to signal an alarm when x is found to lie outside the range from 0 through 15, it has only the duty to signal an alarm when it has used the assumption that x would have such a value (by, for instance, only allocating 4 bits to it).

In interpretation (b) any implementation, therefore, may ignore the range specification, and may take it into account if it can do so at good advantage. In interpretation (a), each implementation <u>has</u> to take it into account; as this implies in general a run-time check, interpretation (a) is in conflict with some of the efficiency requirements. Interpretation (a) has the further dramatic disadvantage that "static type checking" is no longer possible. I call this "dramatic", because it introduces an intertwining of the semantics of the programming languages --describing the net effect to be effectuated by the programs written in it-- and their implementations --i.e. possible computational histories that could achieve that effect-- .

I have learned to appreciate axiomatic, non-operational definitions of programming language semantics, in which the program text defines the net effect to be established independent of the computational histories that may be invoked under control of the programs. In such an approach the program does not prescribe what has to happen during the execution, it only prescribes the answers; what happens during the execution of a program is only defined by the combination of program and implementation. But the implementation is not defined by the definition of the programming language, only constrained by the requirement that always the correct answer will be produced. I have found such a separation of concerns between what should be achieved and how it is to be achieved, absolutely essential. Interpretation (a) which refers to the computational history, would deny to me that separation of concerns; it is for _that_ reason --and not on account of the cost of run-time checks-- that I regard adoption of interpretation (a) as a disaster. The suggestion --page 80-- that formal definition of the semantics could be done via either the Vienna Definition Language or (a la LISP) by an interpreter --which are both operational definitions-- is one of which I cannot approve.

* * *

To Dr.John B.Goodenough's list of typographical errors I can add:

Page	Line	As Is	Should Be
78	7	inovation	innovation

3rd November 1975
Plataanstraat 5
NUENEN - 4565
The Netherlands

prof.dr.Edsger W.Dijkstra
Burroughs Research Fellow