<u>On the BLUE Language submitted to the DoD</u>.

The diary entry that records my initial reactions to the Revised "Ironman", July 1977, starts with the following introduction:

(Quotation from my diary)

Let me remark, right at the start, that it is much better than the earlier versions I have seen. The iteration process has not only more or less converged, during the process it has also become more realistic. So much for the praise. But we have still a long way to go! To quote Niklaus Wirth from

"Programming languages: what to demand and how to assess them."
Bericht 17 des Instituts fuer Informatik ETH Zuerich, March 1976

"I believe that there will be no real progress until programmers learn to distinguish clearly between a language (definition) and its implementation in terms of compiler and computer. The former must be understood without knowledge of the latter. And we can only expect programmers to understand this vital distinction, if language designers take the lead [...]. Hence we conclude that the first criterion that any future programming language must satisfy, and that the customers must ask for, is a <u>complete definition without reference to compiler or computer</u>."

I happen to share Niklaus's belief, to say the least. (I would like to go even one step further: the definition should be independent of any underlying computational model.) But even in comparison to Niklaus's more modest goal, the crew of IRONMAN is still <u>absolutely nowhere</u>! (End of quotation from my diary.)

I had been disappointed in particular by the opening sentence of Requirement 1D : "The language design should aid <u>the production of efficient object programs</u>." (my underlining) instead of "The language design should aid efficient implementation." I was disappointed to see that the phrasing of that sentence still betrayed the old misconception that a programmer should produce object programs and that a compiler is a "programming tool" for that purpose, instead of viewing the object code as an implementation detail to be ignored by the programmer.

Upon careful reading the remainder of the Revised "Ironman" reveals, I am happy to say, a more subtle attitude. It is true that sentences such as (from 3C) "Type definitions shall be processed entirely at translation time.", (from 3-3D) "The number of dimensions for each array [...] shall be determinable at translation time.", (from 4E) "Constant valued expressions (i.e. expressions whose values are determinable at translation time) shall be allowed wherever constants of the type are allowed. Such expressions shall be evaluated before execution time.", (from 6C) "Only the selected branch shall be compiled when the selected case for a conditional statement is determinable at translation time.", (from 7D) "The result type for each function [...] shall be determinable at translation time. [...] its size must be determinable at translation time.", and (from 7H) "The number of dimensions for formal array parameters [...] shall be determinable at translation time." all received from me in margine the same comment "Mixture!". As I now read them they are either direct requirements for the implementation --and as such out of place in a document stating requirements for the programming language-- or indirect language requirements --in the sense that the language should be such that such and such an implementation technique suffices-- . The quoted sentences require an awareness of the implementation process from the language designer, <u>but not from the programmer using the language</u>! This is an enlightened attitude (that I failed to notice on first reading). Compared with that enlightened attitude the BLUE Language seems to me to be a step backwards.

In Section 1.3 (LS 1-10) I encountered --and was puzzled by-- the "manifest-integer-expression", in Section 3.1.4.1 (LS 3-14) I found the (not very illuminating) syntactical definition

```
"[3-14]   manifest-expression          ::= expression"
```

but eventually, still in Section 3.1.4.1 (LS 3-18), I retrieved the intention: "A manifest-expression is an expression whose value is computable at translation time." The text continues with "Such expressions are defined in more detail for each of the predefined types", a promise that I haven't seen fulfilled. (Its fulfillment may be hidden somewhere, but without an index I couldn't find it.) It is my impression --but I am always willing to be corrected if my impression is wrong-- that what are manifest-expression's is <u>not</u> defined by the BLUE Language itself, but is left to the translator. This impression is confirmed by Section 3.5.6.2 (LS 3-80) where the function "IS_MANIFEST" is introduced with the semantics "This function returns the value TRUE if the expression is a manifest expression; otherwise it returns FALSE. [...]". In the justification we find the comment (J3.5-4): "This function is required by the IRONMAN. It was probably intended to be used to control conditional compilation for INLINE routines." The only expression of doubt is the word "probable". I could not find IRONMAN's requirement of that function --perhaps I cannot read-- and definitely missed the remark that it is absolutely superfluous if the language definition itself settles whether an expression is manifest or not. (Note that the further remark in the semantics for IS MANIFEST (LS 3-80): "This function is a manifest-expression." does not exclude that the notion of manifestness is translator dependent!)

This chase was started by Example 5 in Section 3.9.1 (LS 3-120). My first reaction to that example was that in the BLUE Language the question of type identity could pose not only difficult problems --note that because of the declaration "VAR X: INTEGER [1:50] := 50", which declares X as a variable and not as a constant, the establishement of the type of STRING_X already requires a control flow analysis!-- but easily unsoluble problems, because I could make the question of type identity between two variables dependent on input. That counter-example, however, turned out to be invalid when I reached Section 3.12.1 (LS 3-149) that states "Type-arguments [...] of type-specs in declaring type-formals must be manifest." This constraint seems to me to undo most of the potential advantages of parameterized type-declarations. I came to the conclusion that the whole notion of the manifest-expression --which I cannot see as an IRONMAN requirement, but only as an invention of the BLUE Language-- is a mistake ("a manifest mistake" if you prefer that qualification). It has a whole cascade of nasty consequences:

1) The manual must be burdened with a precise definition of what manifest-expressions are (and teaching the language becomes more expensive in proportion)

2) The compiler must be burdened with tracing the consequences of these rules; as this implies at least control flow analysis, it will be expensive in translation time.

3) A program for which a number of constants is declared at the beginning can possibly only be made into a procedure for which these local constants are replaced by constant parameters by making it an INLINE routine (see Section 4.3.1

LS 4-13), requiring recompilation over and over again if the flexibility is needed.

4) As a consequence of the above the notion "INLINE" becomes a necessary language ingredient, instead of an (optional) directive for the compiler by means of which the trade-off between program length and computation time can be influenced.

5) It forces the notion "INLINE" to belong to the routine instead of to its individual invocation points, not all of which are necessarily time-critical.

The appropriate tool seems to me to control type identity by the chain that introduced the types and identifying the result of a substitution, and to call two differently identified results of two substitutions different types, even when by accident the results of the two substitutions are identical. The whole analysis should not be more complicated than the following of scope rules. The authors of the BLUE Language have first been seduced by the accidentally equal results of different substitutions, and thereafter found themselves forced to introduce the notion of manifest-expression in order to be able to settle the matter.

The design has had more struggles in the same vein. If we take Section 3.9.5 (LS 3-136) with the record-assignment it says in the Constraints "The type of the record-expression and the variable must be the same." and three lines further "It should be noted that, as mentioned in Sec.3.9.1, two record objects containing constant fields are of the same type <u>if</u> <u>and</u> <u>only</u> <u>if</u> (my underlining) the <u>value</u> of the constant fields are the same." I did not find the requirement that the values with which constant fields are initialized are manifest-expressions; as a result assignment compatibility between two variables of type record has now become a run-time matter. They have similar problems with the array-assignment, as described in Section 3.8.5 (LS 3-113) where the DIMENSIONS and EXTENT attributes of the variable and the expression must be identical. I was very much amazed to see that the BOUNDS attributes are permitted to differ; was this because they propose to save on the test? The justification J3.8-11 is meagre: "Restricting assignments of arrays to arrays with matching bounds seems overly restrictive. It seems reasonable to require only that the number of dimensions and extents match." It means that after the assignment A:= B it is possible that A[i] $\neq$ B[i] , and that seems hardly an attractive feature. Here it seems to me that the opportunistic implementor has had a bigger say than the cautious programmer.

* * *

In Section 4.3.2 (LS 4-17) I discovered another horror. They describe a two-stage matching process: the first phase tests for type identity, and if more than one match is found, an error is detected, if one match is found, the search process stops, if no match is found, the second search stage is initiated, in which some of the type identities are weakened as "assignment compatible"; if in the second stage a single match is found the algorithm stops again successfully, no match is an error (for lack of a third stage), and two matches is again an error, because it points to the same sort of ambiguity as a double match in the first stage. In the case of a single match in the first stage and a single match in the second stage, no ambiguity is detected, because the second stage isn't even initiated. This must be intentional, for otherwise we could have started immediately with the stage requiring assignment compatibility only. I am flabbergasted. Even if the translator could establish assignment compatibility --after the above I am not so certain anymore-- it would be a horror: what is left from requirement 1C "The language [...] should emphasize clarity ..." ? The justification document is silent.....

* * *

The BLUE Language is unacceptably complex. We could find an excuse in section 1.2 of the justification document (pg.1-17) "In this language design effort, we have attempted to satisfy all IRONMAN requirements as fully as possible, even thought it was clear from the beginning that a complex language would be the probable result. However, we felt that only by doing a thorough analysis of language features to support the requirements we would truly understand the cost of the requirements in language and/or implementation complexity. In this Section, we discuss some changes that, we feel, would lead to useful simplification of the language." That sounds alright, but the following three pages suggesting six changes are so meagre that they make one wonder whether the whole effort has been worth the trouble taken: one would have hoped that the designers of the BLUE Language would have learned much more!

We should perhaps give the authors the benefit of the doubt. The problem is that these documents are an inextricable mixture of technical documentation and salestalk. If we see a crazy argument, are we then allowed to conclude that the authors are idiots? It is questionable: perhaps the authors know full well that the argument is crazy, but it is possible that the argument has been included in an effort to win the favour of some general. Who knows? Because I am not intimately familiar with the military audience for which these documents have been written in the first place, and secondly it would be just political interpretation on my part, I have decided not to give authors such benefit of the doubt and to judge their texts as technical documents. But, again, I would like to state once more that such an analysis is possibly unfair.

By the time a group has designed a language of such an appalling complexity as the BLUE Language, it should, if of any competence at all, hate that design and abhor the requirements that has led it to such a monstrum. I said that the suggested changes are disappointing. It is stated "The fixed point data type would be considerably simplified if only radix 2 step sizes are permitted." (By the way: Section 3.2 and 3.2.1 (LS 3-23 and 3-24) would be considerably simplified if they would have been correct: replacing "R-S" in the text by R^{-S} helps a lot.) I agree. In view of the still remaining complexities I would have expected the suggestion to restrict S to $S = 0$; a further advantage is that then the value of R no longer matters. They also remark that "Variant records are quite complicated." I agree. They suggest the removal of "constant components"; that seems sensible. Furthermore they suggest "One simplification would be to require that functions not have any side-effects." I agree, but <u>not</u> with their motivation "Such a restriction would ensure that expressions invoking functions would be side-effect free and could be freely optimized." because this motivation is very one-sided: function routines with side-effects are for the programmer misleading tools! This orientation towards implementation is most strong in the first change that is suggested: "We recommend that aliasing detection be removed as a requirement." Motivation? "The most important practical problem with aliasing detection appears to be that it conflicts badly with separate compilation. Separate compilation is a concept of proven worth in embedded computer system development. Aliasing detection is a new and as yet untested idea...." Now this is amazing. If there is a conflict between aliasing detection and separate compilation, why sacrifice the aliasing detection? The established (and primitive) techniques for separate compilation have been pointed out as one of the major sources of today's software problems, and a significant step forward is presumably only possible when those techniques are reconsidered. But the designers of the BLUE Language are very conservative and propose to keep things as lousy as they are.... (A similarly backward attitude is reflected in J4.5-12 , (pg.4-13) where the suppression of exceptions is proposed as norm because that is usual today. They write "For example, JOVIAL [...] has no way of checking for OVERFLOW at all. The lack of such a capability has not been a problem noted by JOVIAL users. <u>Quite to the contrary</u>." (my underlining). Do we have to conclude that the absence of an OVERFLOW check is a positive convenience? When I read a sentence like that I feel back in the Middle Ages of Computing, when the ease of programming was confused with the latitude for making undetected errors.)

I think that the two suggestions --to restrict the radix to a power of 2 and to abolish the constant record components-- are but a meagre harvest from a chapter on TYPES AND OPERATORS that has grown to 160(!) pages.

Plataanstraat 5
5671 AL Nuenen
The Netherlands

prof.dr.Edsger W.Dijkstra
Burroughs Research Fellow