On the YELLOW Language submitted to the DoD.

Studying the Preliminary Design Phase Report and Language Specification was not a pleasure: I found the documentation poorly written and chaotic. Sometimes the English is just wrong, as in "a record representation with two alternate (sic) formats" (p.E-7; I refer the authors for instance to "The Complete Plain Words" by Sir Ernest Gowers). Sometimes a remark is just (p.F-8) amazing, such as "The User Manual will be based on the structure and content (sic) of the language definition report". On what else could it be based! On the Bible? It is often very carelessly written, such as in the sample of "errors" that an implementation should detect (L-115); sometimes they state the violation of a rule, sometimes the rule violated, and we find in succession:

"Illegal symbol.
Index type must be scalar."

a sloppiness I find intolerable for people claiming to design a language!

Another reason for finding the text unpleasant to read was that I found the text written "down to the programmer" (is that a correct expression?). I mean the following. Every tool is always designed with a model or picture of its users in mind --in the same way as every text is written with some type of reader in mind-- . Trying to reconstruct from the yellow text the programmer its authors have been aiming at, I --as a programmer-- find the text pretty offensive, such as "The Language Definition Report can be used as a reference by programmers, but most will find the User's Manual more helpful." (p.F-7) or "... did not compensate (sic) for the severe changes in programming style that would result" (p.D-30) or "in view of the intended audience for the language [...] it would be too radical a step to entirely prohibit them" (p.D-24). I very much doubt whether the attitude displayed is in accordance with the "spirit of Ironman" --if such a thing exists!-- in which I read an honest effort to do justice to the intellectual challenge presented by the programming task. I found this shift towards greater recognition of the difficulty of programming one of the more refreshing aspects of the Revised Ironman Requirements, and in this respect the YELLOW Language seems to set the clock back by several years.

Upon closer inspection I found all sorts of unclarities (as is, in view of the above, only to be expected). I had a few problems with the example

on the top of p.D-39, which defines the module template

TEMPLATE StackTemplate(i, R) .

My first problem was that in the remaining text the formal parameter R does not occur. I have tried to remedy this by replacing (p.D-39, line + 6)

ARRAY(INTEGER[1..i] OF BOOLEAN);

by ARRAY(INTEGER[1..i] OF R); .

But this was not the end of my problems, for the text continues

"Within the scope of this declaration, the user requiring a stack of up to 100 INTs need only write

INSTANCE StackTemplate(100,INT)"

The last line quoted, however, does not contain a new identifier, and I am at a loss when trying to see how the user can ever refer to the stack of up to 100 INTs he has just acquired.

* * *

My next complaint is a conditional one: it is under the proviso that I have understood The Language Report correctly. (The text is so rambling that I am not sure....) From the section on Arrays I quote (p.L-28):

"Recall that the only representational qualifier appropriate to scalars and integers is range qualification; this range qualification specifies the bounds of an array and is therefore properly included in its type."

Range qualification is <u>not</u> defined in The Language Report! The only illumination of the notion is given in the form of a single example (p.L-20):

"For example,

INTEGER [5..9]

represents the range of integer values from 5 through 9."

Appendix C (p.L-115) gives some more information: among "the errors that a compiler for our language must detect" we find

"Low bound exceeds high bound."

The three quoted sentences strongly suggest to me that the YELLOW Language requires that in the array declaration the low bound for each index does not exceed its high bound, and that hence it is impossible to declare an empty array! (This interpretation is subject to doubt. On the one hand we find the quoted error among those "a compiler for our language must detect" and it is not mentioned under the sample of traps and exceptions "for which compiled code and runtime systems must provide" (p.L-116); on the other hand the text refers to "Arrays whose index ranges are determined only on entry to the scope of their declaration" (p.F-12).)

Now the exclusion of the empty array is absolutely silly. One of the great improvements of ALGOL 60 over FORTRAN at that time was that in FORTRAN's DO loop the repeatable statement had to be executed at least once, whereas in ALGOL 60's for statement zero executions of the repeatable statement was permissible. ALGOL 60 failed to introduce the same generality for arrays, but I find it unbelievable to see that 18 years later that mistake is still faithfully reproduced, and to see the empty set --500 years after the introduction of the digit zero in the Western world-- still treated as a second class citizen. (For the pragmatists who are insensitive to the verdict of mathematical immaturity: to allow the empty array as well would have simplified the language and its implementation, as it would have reduced the number of error messages by one.)

* * *

In the section on Records (p.L-29) we read:

"The type of a record is determined by the types, modifiability attributes, and selectors of its components."

Of course we must guess what is meant by "determined", but if I read it in the usual fashion, I conclude that two records for which "the types, modifiability attributes and selectors of its components" are the same, are thus of the same type. This to my taste rather unescapable conclusion, however, is in conflict with the text that follows "We now define when two declared objects have the same type" (p.L-39). On p.L-41 the variables H and I are stated to be of different type, the quoted sentence from p.L-29 would give them the same type (unless a highly unusual meaning is given to the word "determined").

That whole introduction of types and representations strikes me as an unresolved mess. The algorithm as given on pages L-39/40 is too complicated and should already make one highly suspicious; the examples of p.L-41 fully confirm this impression. It is totally unclear to me how the gentlemen propose to use this facility. We now have the possibility of introducing variables of the same type but of different representations. Now note that parameter passing is explicitly controlled by types and not by representations! (Note, for instance, (p.L-61, my underlining): "The REPLACES option, on the other hand, applies if the existing definition for a particular type n-tuple is to be made inapplicable in some scope." or, a few lines earlier: "extend a routine to apply to a new n-tuple of argument types".) So the single procedure can get parameters of the same type but of different representations. How is the procedure going to distinguish? Maybe I am missing something, but I am afraid that it won't work.

I have more problems with the examples on p.L-41. What is the purpose of the different types "Foot_Type" and "Inch_Type" and what are the consequences of the type difference thus introduced? With A and B having Foot_Type (not "FOOT_TYPE" as in the text) and F and G having Inch_Type, are we allowed to write A + F or A * F ? I don't know, because on pages L-49/50 the requirements on the operands are "same numeric type", and eventually A, B, F, and G are all INTEGER. If these expressions are not allowed, the language tries to introduce "baby dimension analysis" which is a silly endeavour (see EWD660); in this case it would be exceptionally silly, because the result of addition and multiplication is of the same numeric type, and in spite of the very strong interpretation of "same numeric type" an expression like A * A + B would be allowed. If expression like A + F and A * F are allowed, however, what was the purpose of introducing "Foot_Type" and "Inch_Type" in the first place?

So here we are: they pretend to have made the YELLOW Language "strongly typed" , but if you try to find out what is exactly meant by it, you discover that their proposal is terribly ambiguous; and, whatever guess you make to remove the ambiguity, you end up with something rather nonsensical.

In retrospect I am puzzled by the difference in weight they have given to different fundamental requirements. Concerning the absence of side-effects

they have gone further than Ironman --I believe (p.D-33)-- but have thrown away the child with the bathwater. On the prevention of aliasing they don't seem to be very keen. They suggest that that is "only an academic nicety" (p.D-29) and state --without source or other forms of support-- "it appears that aliasing errors, while troublesome in theory, are extremely rare in practice." (p.D-29). (I think no one is worried by their frequency, but everyone should be worried by the havoc they may create when they do occur: the mere possibility of an aliasing error could easily lead to the development of extremely expensive debugging aids!) But they don't seem to care: on p.D-30 they refer to the "untested benefits" and "the untested practical benefits" of alias checking. This "easy" attitude towards alias checking is in strong contrast to the vigour with which they have made their language "strongly typed". I am puzzled. Did they do so in the belief that the latter requirement was easier to meet in a meaningful way than the prevention of aliasing? I hardly dare to suppose such touching innocence....

* * *

When I studied the BLUE Language I was very much amazed how its authors ever could have made the mistake of introducing what they called "a manifest expression". In the YELLOW Language I encounter again the same idea with the same term "manifest". That cannot be an accident. Where has that term been introduced? PL/I? That would explain a lot.

* * *

I quote (p.D-44):

"We decided that block structure of the program is semantically well defined, does not conflict with efficient compilation, can easily be understood,[...]".

The question, however, is whether the designers of the YELLOW Language have understood it, in particular in combination with

1) the introduction of a new type at "scope entry time"
2) recursion.

It means for an implementation that the number of co-existent different types is as unbounded as the depth of recursion. The authors' awareness of this circumstance is not what I would call manifest. (See the extremely loose usage of the term "unique" in (p.L-15) "The language is strongly typed: every expression in a program has a unique type.)

* * *

To ease the evaluation of their design, the authors themselves have already enumerated its virtues as gauged in terms of compliance with the Revised Ironman Requirements, July 1977. If the scores mentioned (pages C-3/6) are honest, they score high for self-satisfaction. Particularly the score they gave their design for 1H. Formal Definition --viz. "Exact Compliance (with capital C! EWD) with the requirement, by means of the mechenism specified"-- strikes me as hilarious.

An unsalvageable mess. I am full of sympathy for their consultants who have been unable to prevent this.

Plataanstraat 5
5671 AL NUENEN
The Netherlands

prof.dr.Edsger W.Dijkstra
Burroughs Research Fellow