On the RED Language submitted to the DoD.

Having decided to study the languages in the alphabetic order --i.e. BLUE, GREEN, RED (or PINK), YELLOW-- the RED Language was number three. Was I getting tired or was it my allergy for pictorial representations? When I picked up the Informal Language Specification (ILS) of the RED Language I was initially repelled, and when I studied the RED proposal for the type FIXED, not even my great love of Euclid's Algorithm for the GCD could soften my feelings. It was only after several days of study when I observed --with some amazement, for I still remembered my initial reactions-- that a certain sympathy had been built up. Its authors had earned this sympathy by their obvious efforts to keep the language simple and systematic:

1) "REDL prohibits all instances of dangerous aliasing." (ILS p.7-14)
2) "At run-time the effect of an invocation of a routine declared with the <inline directive> is the same as if this <directive> were absent, except that the execution times and/or code space requirements for the invocations may differ." (ILS p.7-7)
3) "To prevent side effects from occurring in functions, ..." (ILS p.7-18)
4) "A fundamental principle of REDL's data facility is that the type of any data element is always named explicitly, i.e., it is always given by a single identifier, and distinct identifiers denote distinct types." (ILS p.5-4)

Questions to be asked, however, are "How successful have they been, and at what costs?" We may remark

ad 1) Because array elements can be passed as VAR parameters, the prohibition of aliasing in general requires run-time checks (ILS p.7-15), while the compile time checks seem to require access administration that could become pretty complicated.

ad 2) Regretfully, the <inline directive> refers to the routine and not to its invocations, not all of which need to be time-critical.

ad 3) Regretfully the restrictions on functions are so severe that they exclude the so-called "benevolent side effect" and I have some fear that the child has been thrown away with the bathwater. (Think about virtual storage implementations in which a simple access may have on a lower level the side effect of a rearrangement of primary store.)

4) The notion of "type" has been weakened --by the additional introduction

of attributes-- . It may be my fault, but I did not find the RED Language's discussion of types as clear as I had hoped on account of the quoted sentence. For instance:

> "REDL is strongly typed: each data object has a unique type, determinable at compile-time, which defines the possible values for the object and its set of behavioral properties." (ILS p.5-1)

but on the next page we read that the values of some attributes --and as a result: "the possible values for the object"!-- can only be determined at run-time!

At first I was totally puzzled by section 5.1.2 (ILS p.5-3) where assignment requires source and target to "have the same type" , whereas for VAR parameter passing they are required to have "both the same type and the same representation (i.e. the same values for each attribute)". It was only later --section 5.6.4.1 (ILS p.5-53)-- that I got the impression that the difference only deals with the attribute GROUPING . It was only still later --section 5.8 (ILS p.5-59)-- that I realized that I couldn't understand the attribute GROUPING and hence, failed to understand how variables can have "the same type" --in the sense of uniquely identified-- and yet may differ in their GROUPING attribute. I decided to try whether I could make sense out of the proposal after omission of the GROUPING attribute.

It was then that I encountered a strange inconsequence. Section 5.1.3 (ILS p.5-4) is quite explicit: there are no anonymous types

```
VAR V: ARRAY(1..16) OF BOOLEAN INIT ?;
```

is illegal, it should be:

```
TYPE V_TYPE: ARRAY(1..16) OF BOOLEAN;
VAR V: V_TYPE INIT ?;
```

and a variable W is only of the same type as V provided W is declared to be of V_TYPE: it is the repeated reference to the type identifier V_TYPE that does the trick. That seems sound. Note that in spite of the bounds to be supplied the declaration of an array type as such --section 5.5.2, (ILS p.5-30)-- is a <simple type declaration> and not a <parameterized type declaration> .

I then turned to Section 5.6 (ILS p.5-50) and found the declaration

```
TYPE SQUARE_MATRIX(N): ARRAY(1..N,1..N) OF FLOAT(5, -1.0..1.0);
```

and on the next page in terms of the above declared type the declaration of the variable X

```
VAR X: SQUARE_MATRIX(2) INIT etc.
```

I was very much amazed, because now the type of X again seems to be of the anonymous type that has been abolished by making

```
VAR V: ARRAY(1..16) OF BOOLEAN INIT ?;
```

illegal! My first impression was that it was a misprint, that the authors had forgotten to replace the above declaration of X by

```
TYPE TWO_BY_TWO: SQUARE MATRIX(2);
VAR X: TWO_BY_TWO INIT ?;
```

but according to the syntax diagram 11 --section5.3 (ILS p.5-10) my type declaration for type TWO_BY_TWO is illegal!

I may be very dumb, but I fail to understand why the authors haven't carried their principle of named types to its logical conclusion. Not having done so, they have to resort to the subsidiary condition of "equal values of the attributes" and, hence,

```
VAR Y: SQUARE_MATRIX(1+1) INIT ?;
```

introduces a variable Y of the same type as the X declared above.

A possible source of their confusion may be found in their position that the possible set of values ideally should be determinable at compile-time or, more precisely, that the type should be determinable at compile-time: only type identity needs to be established at compile time. For the time being I regard the fact that the notion of identified types has not been carried to its logical conclusion as a regrettable mistake that seems easy to remedy.

* * *

A more serious source of worry I encountered with the study of the "Multipath Facilities". In the Preliminary Design Phase Report (PDPR p.3-15) I read that semaphores were rejected because they were so "highly error-prone". Full of interest I continued to read and saw that the authors propose to introduce EVENTs instead, i.e. nonnegative integers that can be increased and decreased by 1 . What is in a name?

In two respects, I am afraid, I am not certain that the replacement of semaphores by EVENTs is such an improvement.

First of all, the operations on semaphores had the advantage of being extremely cheap to implement; in the case of EVENTs we have in addition the rule of the priorities, compounded with the rule that within each priority the blocked processes will be served on a first-come-first-served basis. Add to this that while a path is being blocked, its priority may be changed: it seems vain to hope that such an elaborate scheme is equally simple to implement.

Secondly, EVENTs are not data types. "To avoid ad hoc and complex restrictions, we chose not to treat EXCEPTIONs and EVENTs as data types [...]" (PDPR p.2-5). As a result it is impossible to declare an array of EVENTs, and as a result --but again: I may be dumb-- I did not succeed in coding the first nontrivial monitor I tried to write in the RED Language. (I tried to write a monitor that would grant exclusive access to a single resource, but would schedule it on a last-come-first-served basis; the monitor should be able to serve N contenders.)

* * *

I had my next problem with the Compile Time Procedures, section 14.4 (ILS p.14-16 in particular). I had some problems with $DECLARE CAPSULE_ID on the tope line of p.14-16. When used it requires the introduction of the identifiers INT_STACK_CAPSULE and BOOL_STACK_CAPSULE , for which I see no use. This is probably not serious.

I was more worried by the fact that I did not see a way of introducing two integer stacks of different lengths without writing $CALL STACK_GENERATOR twice. I am perfectly willing to believe that supplying the type INTEGER as actual parameter requires something the designers of the RED Language would call a compile-time facility; is it unrealistic to expect something like a parameterized type declaration as the result, to which a parameter like stack size can be supplied at run time?

* * *

I was somewhat alarmed by the underlying attitude of the designers that

made them introduce the directives RECURSIVE and REENTRANT. The idea is clearly that nonrecursive, nonreentrant routines can be implemented more efficiently. (Personally I prefer machines in which this difference in efficiency is absent or can be ignored. They do exist!) The explicit introduction of these two directives requires from the implementation that it is checked that they are present where needed; hence in some stage of the compilation, a complete call graph has to be constructed in order to check the appropriateness of these directives. But, if that call graph is generated anyhow, we don't need the directives! The introduction of these directives forces all implementations to generate the call graph, even those in which we don't care at all whether a routine is used recursively or reentrantly or not. In short: the two directives are only an efficiency gain --but a risk increase!-- if the implementation omits the check.

It may even be a burden. Via EVENTs it is possible to program different paths of a <fork statement> in such a way that certain parts exclude each other in time; the language requires a routine called from a few of such paths to be given the directive REENTRANT, although it is absolutely irrelevant. (Note that a monitor is not always the proper vehicle to implement mutual exclusion.) Are all the routines in the library REENTRANT?

* * *

I wasn't pleased at all by the sentence (ILS p.7-5)

> "We note that a compiler can choose (for efficiency reasons) to implement CONST binding as a call "by reference" as opposed to "copy" provided it can guarantee that the effect of the routine is as though the parameter were passed by copy---viz., that the value of the actual parameter remains unchanged during the execution of the routine."

I didn't like it because, unless such an analysis is trivial and always gives the result "yes, OK", the language seems a very unsafe tool to use; but the design of the RED Language doesn't indicate why the analysis is trivial. On the contrary: the suggested REDL Language Support Facilities are of a frightening complexity (PDPR, section 4.4).

* * *

The documentation should be written in a way which is superior to the way in which the Informal Language Sepcifications have been written.

I refer the reader for example to section 12.4.3 (ILS p.12-13) that begins with

> "The effect of the <wait statement> is to perform the following actions, in an indivisible manner:"

(Presumably the authors meant something like "The effect of executing the <wait statement> is the effect of performing the following actions, in an indivisible manner:") .

Then follow five actions numbered (1), (2), (3), (4), and (5). Action (1) is empty (but refers to "step" (3)). Action (2) is obscure in its last sentence: "If the path executing the <wait statement> holds the named monitor, then it releases the monitor." As the previous sentence has postulated that the <wait statement> must be lexically contained in the named monitor, it is hard to see --at least for me-- how the path executing the <wait statement> can fail to hold the named monitor; the word "then" in the quoted sentence is superfluous; I failed to find the definition of "releasing" a monitor. Actions (3) and (4) exclude each other; besides that,in action (3) "execution continues in the scope which contains the <wait statement> " is wrong: because we have still actions (4) and (5), it should be something like "execution will continue after completion of action (5) etc.". Action (4) describes how the path will be suspended; in view of the fact that action (5) is still to follow, it is hard to reconcile that suspension with the usual interpretation of "in an indivisible manner". The description of action (5) is linguistically OK (but for another superfluous "then" and the fact that the last two sentences had better be separated by a weaker separator than the period.)

* * *

The reports dealing with the Red Language are as disturbingly inhomogeneous as those for the GREEN Language. Again one must fear that they have been produced by an incoherent team. While the design has tackled a number of fundamental design issues, in a way we should appreciate, we also encounter irrelevant remarks such as --section 4.3.1.2 PDPR p.4-21-- "The entire interpreter together with its tables will be resident in main memory."; a few lines lower the text refers to "overlays". I don't know what to do with such remarks; on the one hand I can skip them as irrelevant and suppose that some underling has imported them into the text without the real authors --due to lack of time-- really noticing it, on the other hand I can assume that the authors really think this remark important enough to be included. In the latter case it betrays an almost

medieval attitude towards programming that would justify the questioning of most of their design justifications that are based upon implementation considerations. The proposal is both advanced and backward in such an incongruous manner that I am baffled.

* * *

Have I failed to do the designers of the RED Language justice? Perhaps: the above has been written despite a mounting headache. Whether my feelings towards the RED Language are the result of the headache, or the headache is the result of the RED Language remains an open question.....

Plataanstraat 5
5671 AL NUENEN
The Netherlands

prof.dr.Edsger W.Dijkstra
Burroughs Research Fellow