cand and cor before and then or else in ADA;

David Gries

TR79-402

Department of Computer Science
Cornell University
Ithaca, New York 14853

cand and cor before and then or else in ADA;

David Gries
Computer Science Department
Cornell University
November 9, 1979

This note concerns the use in ADA of the "and then or else control device" instead of the more conventional operators cand and cor. I consider this to be a serious mistake. I will argue why the "rationale" for the ADA choice is faulty and I will attempt to point out the inadequacies and confusion that arise because of this choice.

The logical operators are not, and, or and xor, and it is explicitly stated in the ADA manual that all operands will be evaluated. This means that evaluation of

(1) $a \neq 0$ and $b/a = c$

will cause an exception to occur -- that is, an error will be recognized -- even though the value of the expression, true, is known as soon as the first operand $a \neq 0$ is evaluated.

The traditional (whatever that means in a field so young) solution to allow the programmer the flexibility of writing expressions like (1) without incurring the penalty of error during execution is to introduce the operators cand and cor, which are defined by the following truth table (the symbol "U" denotes undefined):

(2)

b	c	b cand c	b cor c
T	T	T	T
T	F	F	F
T	U	U	T
F	T	F	T
F	F	F	F
F	U	F	U
U	-	U	U

In other words, the operations denoted by the two operators can be defined by

$$b \text{ cand } c = \text{if } b \text{ then } c \text{ else } F$$
$$b \text{ cor } c = \text{if } b \text{ then } T \text{ else } c$$

These operators are used in ALGOL W, in EUCLID, and in a 4 year old research monograph on programming -- Dijkstra's A Discipline of Programming. They can be added to a language at little, almost trivial, cost -- a few extra lines to describe them and a simple implementation (see Gries, Compiler Construction

Digital Computers, 1971). Note that a language could have both and and or, for which both operands would always be evaluated, and cand and cor, for which the second operand would be evaluated fif (if and only if) necessary, as indicated in the above description.

Instead of choosing cand and cor to allow a certain flexibility and style of programming, the ADA designers have chosen to introduce the condition, with syntax

```
(3) condition ::=  boolean expression  {and then  boolean expression}
                 | boolean expression  {or  else  boolean expression}
```

Thus, (1) could be written in ADA to avoid evaluation of b/a=c as

(4) if a $\neq$ 0 and then b/c = a then ...

But it is important to realize that a condition is not an expression in ADA; it has no type and it cannot be written wherever one might expect a Boolean expression. It is best thought of as a "control device". This leads to inadequacy and confusion. It is a serious mistake.

The rationale behind the ADA choice

The rationale behind the ADA choice seems to rest completely on **three** sentences found at the beginning of Section 3.3 (page 3.4) of Part B of the ADA documentation, the"Rationale for the ...". I quote (the numbers preceding sentences are mine):

> (1) The purpose of an expression is the computation of a value. (2) The evaluation of an expression is conceptually an indivisible operation. (3) Hence it must be assumed that all of the constituent expressions will be evaluated and that any exception condition that can arise in any part of the expression cannot be avoided.

I have no quarrel with sentence 1, although, strictly speaking, one must look at the context surrounding an expression to determine its "purpose". For example, an expression within an assertion in a proof of correctness is used only to help define a set of states, and it is never evaluated.

I also have nothing to say about sentence 2. But I do not, can not, draw conclusion 3 from sentence 2. It is a non sequitur. The need to evaluate all subexpressions simply does not follow logically from the indivisiblity of expression evaluation.

I think that the designers of ADA meant to say that expressions should be defined so that one could always unambiguously determine whether evaluation of an expression in a given state is well defined. This is certainly reasonable, but it is a far cry from requiring all subexpressions to be evaluated.

The inadequacy of the and then or else control device

One can easily see by the syntax of conditions (3) that the following cannot be easily translated into ADA:

(5) if a[p] $\neq$ 0 cand (l[p] $\neq$ 0 cor m[l[p]] = 0) then ...

This is because and then and or else may not appear in the same expression! The ADA report, Part B, Section 3.7, attempts to explain the reason for this, but ends up with another non sequitur. On page 3.7 we find

> Since the rules of evaluation of and then clauses and or else clauses are contradictory, the two forms of clauses cannot be mixed in the same condition.

But why should the "contradictory" nature of their evaluations mean they cannot be mixed? Would it not suffice to be able to tell from the context whether a clause is an and then or or else clause? And isn't this always obvious from the context?

Please note that (5) above was not cooked up simply to show the inadequacy of the and then or else control device; it is the simplest and most elegant way to express part of a version of an algorithm for marking nodes of a directed graph. The expression a[p]$\neq$0 means "node p is not an atom", l[p]$\neq$0 means "the left successor of node p is nonempty" and m[l[p]]=0 means "the left successor of node p is unmarked." This shows that conditions with more complicated syntax than allowed in ADA are necessary and useful.

The control device is inadequate in another way; conditions may not be written wherever a Boolean expression may, and this violates the principle of "syntactic uniformity" (which has not yet been fully explicated). Thus, although

if a $\neq$ 0 and then b/a = c then ...

is legal, the following is not:

t:= a $\neq$ 0 and then b/a = c

To understand this it is necessary to understand the ADA viewpoint: conditions are not expressions, and cannot appear wherever expressions can occur, but only in certain respected places. A condition is not an expression; it has no type and thus cannot be evaluated to yield a value. The rationale for this was given earlier.

Confusion reigns

The inadequacy of the control device leads to other confusions as well. For example, it is usual in a class to describe the case statement as an extension of the if statement. That is, any if statement can be rewritten as a case statement where a Boolean expression is being tested. The following should be equivalent:

```
if a ≠ 0 and then b/a=c             case a ≠ 0 and then b/a = c of
    then  S1                             when true  => S1
    else  S2                             when false => S2
end if                              end case
```

But they are not, for the second is illegal! The expression of a case may not be a condition. Such confusion and inconsistency does not belong in a modern, Pascal based language.

It would appear that the ADA designers are interested in proofs of correctness of programs to some extent, based on the fact that they have included an assert statement. Note that, in general, a condition is not thought of as an expression; it has no type. Nevertheless, the assert statement has the syntax

```
assert condition ;
```

and, in this particular place, the Report discusses evaluating the condition to yield the value true or false. Thus, in some places the condition is treated as a control device while, where it suits the designers needs, it turns into an expression. This smacks of hypocrisy. A spade should be called a spade and should be useable in any place that another spade is.

Note also that the following is not a legal statement:

```
assert a ≠ 0 and then (l(p) ≠ 0 or else m(l(p))=0) ;
```

This means that I cannot write certain assertions in ADA, and there is no other way for me to get the same effect.

Concluding remarks on the control device

The and then or else will only confuse the novice with its irregularities and inconsistencies and enrage the expert with its inadequacies -- especially since a better solution exists. It should be changed. The language will become a bit smaller, the manual will decrease in size by a good deal, several pages will be deleted from the Rationale, and the language will be much more consistent and clean.

I would also like to say that somebody outside the ADA group should be paid well to edit the ADA report for consistency, precision, and grammar. I am not faulting the ADA designers; they are just too close to the report to do a good job of it anymore. The non sequiturs in the Rationale, the language of the report, some of the inconsistencies in the notation and syntax of ADA itself, all these lead me to believe that someone else, a person of Peter Naur's stature, should be involved.

A final note on semicolons

The reader may have wondered what the semicolon was doing at the end of the title of this constructive criticism. It looks awkward, doesn't it? I feel the same way about the semicolon as a terminator in a language. I was educated to use the semicolon as a separator, or joiner, of adjacent clauses, and it was nice to see that tradition carried on by ALGOL 60. It was even nicer to see this standpoint backed up by theory. In terms of weakest pre-conditions, the semicolon is the operator for composition:

$$wp(\text{"S1; S2"}, R) = wp(S1, wp(S2,R)).$$

From the standpoint of tradition, grammar, theory, elegance and style, the semicolon deserves its place as a separator. To use the PL/I blunder of the semicolon as terminator is, in my mind, intolerable. The grounds for this are very weak: a badly done experiment and the fact that it is supposed to help the compiler recover after errors.

If the ADA designers do not like the semicolon as separator, then they should come up with a fresh, new solution. No separators at all. Or a different symbol as terminator. Or terminators only where necessary. But the current approach is intolerable.

Note that ADA blindly requires the semicolon even at the end of compound statements, which already have a terminating reserved word. All compound statements end in a reserved word, so the semicolon becomes unnessary. Why write end if; end case; end loop; end select; ?;

Don't repeat the crude, PL/I blunder.

Note that the use of semicolon as terminator has already caused confusion on the part of the ADA designers themselves. On page 2-4 of the Rationale is an example:

```
if  A ≠ 1 then
     S1;
elseif ...
```

But the semicolon does not belong after S1, because a semicolon belongs to S1. For example, substitution of the assignment A:=1; for S1 would yield the incorrect segment

```
if  A≠1 then
     S1;;
elseif ...
```

Final remarks

I have criticized two rather obvious points of ADA. The and then or else control device has a fair chance of being changed, since it is so obviously wrong. The other, the use of semicolon as terminator, I see little chance of having changed. Not because it should not be changed, bu rather because not enough people see the need for style and elegance. You may not be able to measure style and elegance well, to measure how much they matter, but in my opinion style and elegance are a necessary part of good programming. And beautiful programs just don't look beautiful in an ugly language.

These points are important because ADA may very likely become the language. It may be used and taught throughout computer science in five years. It is thus extremely important that even such points as the use of semicolon be analyzed and discussed to the fullest extent. Any small bad point in ADA will be magnified many times because of the heavy use the language may get. It is important that ADA be as small, clean, precise and consistent as possible.

I am sorry that I did not have time to study the ADA Report more carefully in order to be able to comment on other substantial matters.