Advance Access publication on 11 December 2014

doi:10.1093/comjnl/bxu145

On the Origin of Recursive Procedures

GAUTHIER VAN DEN HOVE*

CWI, SWAT, Science Park 123, 1098 XG Amsterdam, The Netherlands
**Corresponding author: ghe@cwi.nl*

We investigate the origin of recursive procedures in imperative programming languages. We attempt to set the record straight, and to identify the trend that led to recursive procedures, by means of an analysis of the related concepts and of the most reliable available documents, as far as known to us. We show that not all of those who were involved in defining these concepts in these documents were fully aware of the implications of their proposals. Our aim is not primarily historical, but to contribute to a clarification of some of the concepts related to recursion. In particular, we demonstrate that recursive procedure declarations and recursive procedure activations are logically disjoint concepts.

Keywords: procedure; recursion; declaration and activation; self-application; call by name

Received 27 June 2014; revised 17 October 2014
Handling editor: Manfred Broy

1. INTRODUCTION

It is well-known that the first programming language in which recursive procedures were included is J. McCarthy's LISP. However, he did not consider them explicitly in his first drafts of the language, and it is only around January 1959, that is, two or three months before the first implementation of LISP was completed, that he added, before the *define* primitive, a *label* primitive to the language, on a suggestion from N. Rochester [1, p. 179]. Something similar seems to have happened, about one year later, for the other major programming language designed in the late 1950s, namely ALGOL 60. McCarthy, now convinced of the importance of recursion, suggests in August 1959 to introduce it explicitly in the language, arguing that this feature has 'proved convenient' in LISP [2], but his proposal does not draw much attention. A few months later, in January 1960, during the discussions to finalize the language, a proposal from the American members of the committee to add a *recursive* declarator to the language is rejected [3, p. 112]. Finally, in February 1960, P. Naur, editor of the *Report*, decides, on a suggestion from A. van Wijngaarden and E. W. Dijkstra, but without consulting the other committee members, to add a sixteen-word sentence to it, to make it clear that the declaration of recursive procedures is explicitly allowed. We have two written accounts of that event, by its primary actors. The first one was written by Naur two decades later [3, pp. 112–113]:

> The last substantial change of language concept was the admission of recursive procedure activations. This took place as follows. [...] [O]n about 1960 February 10, [...] I had a telephone call from A. van Wijngaarden, speaking also for E. W. Dijkstra. They pointed to an important lack of definition in the draft report, namely the meaning, if any, of an occurrence of a procedure identifier inside the body of the declaration other than in the left part of an assignment. They also made it clear that preventing recursive activations through rules of the description would be complicated because of the possibilities of indirect activations through procedures and their parameters. They proposed to clarify the matter by adding a sentence to section 5.4.4: 'Any other occurrence of the procedure identifier within the procedure body denotes activation of the procedure'. I got charmed with the boldness and simplicity of this suggestion and decided to follow it in spite of the risk of subsequent trouble over the question.

The second one is from Dijkstra, and was written four decades later [4, p. 5]:

> A major milestone of ALGOL 60 was its introduction of recursion into imperative programming. [...] Its inclusion was almost an accident and certainly a coup. When the ALGOL 60 Report was nearing completion and circulated for final comments, it was discovered that recursion was nowhere explicitly excluded, and, just to be sure that it would be in, one innocent sentence was added at the end of the section 5.4.4.

At first sight, these two accounts seem to tell the same story: recursion had been rejected by the majority of the ALGOL 60 committee members, and has finally been included, surreptitiously, by the sole will of a few people; J. McCarthy was surprised that it had been included [5, p. 160], and F. L. Bauer even considered that it had been added to the language because of an 'Amsterdam plot' [3, p. 130], and did not implement it [6, p. 6]. One could then comment on this 'plot', by criticizing van Wijngaarden and Dijkstra's undemocratic attitude which resulted in an additional burden for those who implemented the language, or by interpreting it as a typical example of their supposedly theoretically inclined minds, or by explaining that it was prophetic because recursion later proved to be an important tool.

On a closer look, however, there are a number of differences between these two records. The first one is that Dijkstra's comments are about 'recursion', whereas Naur speaks of 'recursive procedure activations'. The second one, which seems even more important, is that Naur explains that the added sentence introduced a 'substantial change', whereas Dijkstra claims that it was an 'innocent sentence'. In other words, these two accounts apparently contradict each other. To understand what actually happened, it is first necessary to define a few terms carefully, that is, without introducing a *petitio principii*.

2. DEFINITIONS

Some definitions can be given without a detailed discussion: they are not problematic for our discussion, and do not lie at its heart. A *function* is a mathematical concept: it defines a mapping between the value of a number of arguments and a value. A *procedure*, or algorithm, is a computational concept: in its most general sense, it is a program text, that can be called with a number of arguments, and that eventually returns to the point immediately following that from which it was called. It specifies a number of steps by which a certain task, for example the calculation of the value of a function, can be performed mechanically. An *activation* of a procedure is an execution instance of that procedure: it is created when the execution of the procedure begins, and terminated when the end of the procedure is reached. If the procedure is executed on a computer, its activation uses a portion of the computer's memory to record the data used by the procedure.

The word *recursion* is, like any word, ambiguous.[1] Its original meaning stems from the word 'recur', which comes from the Latin word 'recurrere' meaning 'to run (currere) again (re-)'. With this meaning, any iterative process can be called 'recursive', which explains that some of the early ACM algorithms (for instance, 10–13 and 44) were described as being 'recursive' although they are iterative. This is obviously not the meaning that is used in the above discussion.

Its second meaning was introduced by H. Grassmann in 1861,[2] and formalized by R. Dedekind, who called it 'recursion', in 1888,[3] and by G. Peano in 1889.[4] It designates a definition in which (a) at least one of the *designata* (the defined objects) is explicitly given, and in which (b) the *definiendum* (the defined symbol) is used in the *definiens* (the definition), with the meaning of the *definiendum*. Giving a particular instance of the defined objects has long been considered as an improper way of defining,[5] and circular or self-referential definitions have also long been considered incorrect,[6] but by combining both methods with a conditional, it becomes possible to build effective definitions. This meaning is now well established:[7] recursion is a method of definition that corresponds to the method of induction for proofs, itself introduced by F. Maurolico in 1557[8] and further formalized by B. Pascal in 1654.[9] (Earlier occurrences of both of these methods can be found,[10] but they are of less interest because they did not influence later mathematicians.)

A third meaning of the word has been introduced by K. Gödel in 1931 in the study of decidable functions [21, pp. 179–180], and he generalized it in 1934, on a suggestion from J. Herbrand, to include functions such as the famous one designed by W. Ackermann [22, pp. 26–27]: a function is said to be 'recursive' if it is possible to define it by using a certain given set of simple rules, one of them involving recursion in the second sense of the word. This definition formalizes the fact that it is possible to arrange the calculation of the function value in such a way that each step comes closer to the result, and became one of the ways to give a precise meaning to the intuitive notion of 'computable'.[11] This is clearly not the appropriate meaning for the above discussion: any procedure is 'recursive' in that sense.

A fourth meaning of the word 'recursion' designates a certain thought process by which a procedure is designed in such a way that its execution is restarted on a subset of its input data, and eventually returns to the point following that from which it was restarted. This restarted execution proceeds independently of any previously started executions of the procedure, and when it returns only its results are kept. It is that thought process that C. A. R. Hoare used to design the Quicksort algorithm, before he knew about the possibility of defining procedures that are recursive in the second sense. That mental process is thus facilitated by, but does not require, the possibility of defining recursive procedures [23, p. 76]. It is also because of this meaning that Quicksort is said to be a recursive algorithm, even when it is implemented without

[1] See [7], especially pp. 309–311.

[2] [8], for example Sections 2.7, 2.15 and 2.18, 4.56 and 4.57.

[3] [9], p. ix, and Sections 71 and 126.

[4] [10], for example Section 1, axioms 1 and 6, definition 18, and Section 4, definitions 1 and 2.

[5] See, for example, [11], 287e–289d, [12], 5d–e.

[6] See, for example, [13], 79b–d, [14], 209d–210a, [15], b. 6, c. 4, 142a33–b5.

[7] See, for example, [16], p. 217, or [17], p. 131.

[8] [18], for example p. 7, propositio 15^a.

[9] [19], p. 7, Consequence douziesme.

[10] See, for example, in 1321, [20].

[11] [7] demonstrates p. 307 that neither Gödel nor Turing used the work 'recursive' to mean 'computable' after the discovery of Turing machines in 1936, and argues pp. 312–315 that this third meaning of the word 'recursive' should be abandoned, in favor of the word 'computable'.

explicit recursion, for example for efficiency reasons. Again, this fourth meaning is evidently not the meaning that should be used to interpret the above quotes, because they are concerned with the way in which procedures are expressed, and not with the way in which they are designed. We can therefore conclude that the correct meaning is, in this case, the second one: a *recursive procedure declaration* is a procedure that is defined recursively, in the second sense of the word.

Finally, a *recursive procedure activation* is, like the procedure activation concept, a computational concept. It designates a procedure activation that is created at a moment when one or more activations of the same procedure exist, that is, when one or more executions of that procedure are not completed. Put another way, a recursive procedure activation does not designate an activation of a recursive procedure (if a recursive procedure has, in a given computation, only one activation, then it is not a recursive procedure activation), but a recursive activation of a procedure. The word 'recursive' in this case is thus not derived from the second meaning of the word 'recursion' (circular definition), but from the first one: the procedure 'runs again', where 'again' should here be understood in its cumulative sense rather than in its iterative sense. This could well be a fifth meaning of the word 'recursion', similar to but nevertheless different from the first one.

The problem posed by these two definitions is that of the relation between recursive procedure *declarations* and *activations*: are these concepts related, and if so, how?

3. DIRECTLY AND INDIRECTLY RECURSIVE PROCEDURE DECLARATIONS

The definitions given in the previous section are clearly not *ad hoc* definitions: they are precisely what the sentence added in last minute to the *Report* designates, incidentally without using the words 'recursion' or 'recursive procedure activation'. It reads: 'Any other occurrence of the procedure identifier within the procedure body denotes activation of the procedure'.[12] In this sentence, the word 'other' means 'other than in the left part of an assignment statement', the words 'occurrences of the procedure identifier within the procedure body' refer to the circular references to the procedure identifier in the procedure declaration, and the words 'activation of the procedure' refer to the (recursive) procedure activations that correspond, during the execution of the program, to these procedure identifiers. Let us consider for example the following definition:

$$\begin{aligned}&\mathit{fibonacci}\ (n) =\\ &\quad (n = 0 \vee n = 1) \rightarrow n,\\ &\quad \textbf{true} \rightarrow \mathit{fibonacci}\ (n-1) + \mathit{fibonacci}\ (n-2)\end{aligned}$$

[12][24], Section 5.4.4 *in fine*.

The added sentence states that it is possible to express this definition in a straightforward fashion by the following procedure declaration:

```
integer procedure fibonacci (n); integer n;
fibonacci := if n = 0 ∨ n = 1 then n
     else fibonacci (n − 1) + fibonacci (n − 2);
```

In this procedure declaration, only the two last occurrences of the identifier *fibonacci* have the same meaning as the identifier present in the procedure heading: the first one, before the assignment symbol, is used merely to indicate to which function procedure the calculated value should be associated; this is useful when a procedure sets the value of a function procedure in which it is declared (as in the next example procedure). Dynamically, the evaluation of *fibonacci* (n) creates $2 \times \mathit{fibonacci}\ (n+1) - 1$ procedure activations, of which only one, namely the first one, is a non-recursive procedure activation.

However, we can readily note that, as is well-known, recursive procedure declarations do not necessarily imply recursive procedure activations. This can be demonstrated with the following procedure declaration, in which the inner procedure is 'tail recursive':

```
integer procedure fibonacci (n); integer n;
begin
     procedure aux (m, a1, a2);
          value m, a1, a2; integer m, a1, a2;
     if m = 0 then fibonacci := a1
          else aux (m − 1, a2, a1 + a2);
     aux (n, 0, 1)
end;
```

The procedure *aux* is recursive, but its parameters are passed by value, and the recursive call is not followed by any other operations, which means that nothing remains to be done by the calling activation after the end of the called activation. It is therefore possible, and in some languages even required, to terminate the calling activation before entering the recursive one, that is to say, to translate the procedure *aux* into a loop. This leaves us with the question: do recursive procedure activations imply recursive procedure declarations?

Let us now conduct a thought experiment, and let us assume from now on that the event described in the introduction did not take place. In other words, let us consider the *Report* without the contentious sentence. Let us also assume that its absence means that procedure declarations like the previous ones are not allowed. This restriction can easily be circumvented with 'mutual recursion'. A set of procedures $p_1, \dots, p_n$, with $n \geq 2$, is mutually recursive if each procedure p_i can indirectly call any of the procedures $p_1, \dots, p_n$. A simple example of this kind of definition, with $n = 2$, is to translate the above definition into the following procedure declaration:

```
integer procedure fibonacci (n); integer n;
begin
    integer procedure aux (n); integer n;
    aux := if n = 0 ∨ n = 1 then n
        else xua (n − 1) + xua (n − 2);
    integer procedure xua (n); integer n;
    xua := aux (n);
    fibonacci := aux (n)
end;
```

This procedure declaration evidently implies recursive procedure activations: the evaluation of *fibonacci* (n) creates $2 \times$ *fibonacci* $(n + 1) - 1$ activations of the procedure *aux*, of which only one is a non-recursive activation, and $2 \times$ *fibonacci* $(n + 1) - 2$ activations of the procedure *xua*, of which only two are non-recursive activations. It seems, given the definition of 'recursive procedure declaration' given above, that these two procedures are not recursively declared, because the identifier *aux* does not appear in the declaration of the procedure *aux*, and likewise for the procedure *xua*. However, the definition of 'recursion' given above does not specify that the *definiendum* has to appear explicitly in the *definiens*, only that it should be used in it, which means that if one of the elements of the *definiens* is itself defined with the *definiendum*, the definition is recursive, albeit indirectly so. For instance, if one defines a number as 'something that represents a quantity', and a quantity as 'a number of things', then these two definitions, in combination with each other, are circular. All this implies that the two procedures above do not only imply recursive procedure activations, but are indeed also recursively declared.

One could now object that, given that the declaration of directly recursive procedures is, in our thought experiment, nowhere explicitly allowed in the *Report*, it is at the very least unclear whether the declaration of indirectly recursive ones is possible. All in all, the existence of mutual recursion does not seem to be a totally convincing reason to consider that Naur and Dijkstra are both wrong when they claim respectively that 'recursive procedure activations' and 'recursion' have been added in last minute to ALGOL 60. We will therefore strengthen the conditions of our thought experiment, and we will assume that recursion, that is, both direct and indirect recursion, is not permitted in ALGOL 60. This leaves us with a new question: is it possible to have recursive procedure activations without having recursive procedure declarations?

4. THE POSSIBILITY OF SELF-APPLICATION

Besides mutual recursion, another way to evade the restriction just stated is to make use of procedural parameters. Let us consider for example the following procedure declaration:

```
integer procedure fibonacci (n); integer n;
begin
    integer procedure aux (n, p);
        integer n; integer procedure p;
    aux := if n = 0 ∨ n = 1 then n
        else p (n − 1, p) + p (n − 2, p);
    fibonacci := aux (n, aux)
end;
```

This procedure uses 'self-application', which means that a procedure is passed as argument to itself. Because the procedural argument is used inside the procedure and again passed to the two formal procedure calls, the evaluation of *fibonacci* (n) will also create $2 \times$ *fibonacci* $(n + 1) - 1$ activations of the procedure *aux*, of which only one is a non-recursive activation. What makes this example especially interesting is that the procedure *aux* is, according to the definition above, not recursively declared: the identifier *aux* appears nowhere in the body of the procedure, neither directly nor indirectly. The procedure *aux* is defined in terms of a certain procedural parameter p, which happens, but does not need, to be equal to *aux* in the subsequent procedure call. The most convincing argument that can be given to show that nothing requires that p be identical to *aux*, and that the declaration of *aux* is therefore not recursive, is to demonstrate that it can make sense to replace the parameter *aux* with another procedural parameter:

```
integer procedure lucas (n); integer n;
begin
    integer procedure aux (n, p);
        integer n; integer procedure p;
    aux := if n = 0 ∨ n = 1 then n
        else p (n − 1, p) + p (n − 2, p);
    integer procedure xua (n, p);
        integer n; integer procedure p;
    xua := (if even (n) then 1 else −1) ×
        2 ↑ (n + 1);
    lucas := aux (n, xua)
end;
```

This procedure, in which the Boolean procedure *even* is assumed to be defined, calculates the values of the Lucas sequence $U(-2, 0) = 0, 1, -2, 4, -8, 16, -32, 64, -128, \ldots$, without ever activating the procedures *aux* and *xua* recursively; this would clearly not have been possible if the procedure *aux* were declared recursively. We can thus conclude that non-recursive procedure declarations may imply recursive procedure activations, and that recursive procedure activations do not imply recursive procedure declarations.

The technique of self-application is now well-known, but one could argue that we are confusing today's evidences and yesterday's evidences. We can observe for example that at about the same time McCarthy was unaware of this technique, and that he realized only later that the *label* primitive, whereby an identifier is associated with an uninterpreted function body which can therefore contain that identifier, was actually not necessary in LISP, because the same effect could have been achieved by using the so-called 'Y combinator' [25, p. 346], which uses that technique [1, p. 179]. It is therefore possible that those who wrote the *Report* were also unaware of this possibility. However, in November 1959, two committee members and two non-members proposed to modify the syntax of procedure headings, to specify the kinds of arguments of procedural parameters by means of extra commas and parentheses [26]. A procedure *proc* taking one integer parameter n and one procedural parameter p, in which the actual procedure parameter should take two integer parameters and one procedural parameter, itself taking a single integer parameter, would then have the following heading:

```
procedure proc (n, p (, , ()));
    integer n; procedure p;
```

They noted that 'this prevents automatically that [the procedure identifier] is inserted in place of [the actual procedure parameter] in the corresponding procedure statement'. In other words, this rule renders self-application syntactically impossible. In the example above, the procedure *aux* would have to be specified with the following infinite string:

```
integer procedure aux (n, p (,(,(,(,(...))))));
    integer n; integer procedure p;
```

The motivation given by the authors of the proposal, namely to avoid that a procedure 'calls itself indefinitely', is however not very clear, and could make us doubt that it was self-application that they had in mind. Further, it would have been possible to escape this limitation with indirect self-application, a construction analogous to mutual recursion:

```
integer procedure fibonacci (n); integer n;
begin
    integer procedure aux (n, p ());
        integer n; integer procedure p;
    aux := if n = 0 ∨ n = 1 then n
        else p (n − 1) + p (n − 2);
    integer procedure xua (n); integer n;
    xua := aux (n, xua);
    fibonacci := aux (n, xua)
end;
```

In this example, the declaration of the procedure *aux* is again, for the same reasons as those given above, not recursive: it is defined in terms of a certain procedural parameter p, which does not need to be equal to *xua* and to refer to *aux* itself. On the contrary, because it explicitly refers to *aux* and indirectly to itself through the procedural parameter *xua*, the declaration of the procedure *xua* is recursive; it must therefore, in our thought experiment in which recursive procedure declarations are disallowed (see Section 3), be rejected.

The fact that the authors of the proposal apparently did not see that it was possible to circumvent the limitation they wanted to introduce shows that the two mechanisms we have discussed, namely mutual recursion and self-application, were not clearly envisioned by all the authors of the *Report*. We must thus conclude that they were not immediately obvious for everyone, at the sole reading of the *Report*. Moreover, one could also object that self-application is still a disguised form of indirect recursion, because it consists in dynamically creating, with a procedure that is not formally recursive in the program text, another procedure that becomes recursive during the program execution. All this leads us to strengthen the conditions of our thought experiment a second time, and to also disallow self-application, as for example in PASCAL, in which it is not possible to use direct self-application because of a syntax constraint similar to the above proposal. We are then confronted with a stronger question: is it possible to have recursive procedure activations without recursive procedure declarations and without self-application?

5. THE CONSEQUENCES OF THE CALL BY NAME MECHANISM

Let us consider the following innocent-looking function procedure declaration, which does not use any ambiguous features of ALGOL 60, and is in particular not recursive:[13]

```
integer procedure gips (i, n, var, val);
    integer i, n, var, val;
begin
    for i := 1 step 1 until n do var := val;
    gips := 1
end;
```

It is possible, using only this procedure and a single assignment statement, to calculate the Fibonacci numbers:

```
integer procedure fibonacci (n); integer n;
begin integer i, j, k, l;
    gips (i, (n + 1) ÷ 2 + 1, k, if i = gips (j,
        i − 1, l, if j = 1 then 1 else (l × (n
        − 2 × i + j + 2)) ÷ (j − 1)) then 0
        else k + l);
    fibonacci := k
end;
```

The inner workings of this procedure are admittedly a bit obscure (the numbers are calculated with a sum of binomial coefficients $\sum_{i=0}^{\lfloor (n+1)/2 \rfloor -1} \binom{n-1-i}{i}$), but it is clear that it does not use any feature of the language that is not documented in black and white in the *Report*. What it does use is the call by name mechanism, whose semantics are specified as follows: 'any formal parameter [called by name] is replaced, throughout the procedure body, by the corresponding actual parameter' [24, Section 4.7.3.2]. Given that the procedure identifier *gips* appears in the fourth parameter passed by name to the procedure *gips*, and given that this parameter is evaluated for each possible value of n passed to *fibonacci*, it is clear that

[13]It is inspired by the 'general problem solver' in [27, p. 271].

the procedure *gips* is activated recursively: for any $n \geq 0$, the procedure *gips* is activated two times at the same moment, at least once.

One could argue that this example does not correspond to the definition of recursive procedure activations given above; one might think that it means, implicitly, that the number of simultaneous procedure activations varies from call to call and is unbounded. This understanding is, however, incorrect: recursive procedure activations are present as soon as there is more than one activation of a given procedure, and it is then in particular no longer possible to store the data used by the procedure at a fixed place in the computer memory, which was the standard way of implementing subroutines before 1960. By the way, if a program contains a call to *gips* with *fibonacci* in one of its parameters, or a call to *fibonacci* with a call to *fibonacci* in its parameter, the number of simultaneous activations of the procedure *gips* would increase.

We can now finally conclude that the answer to our question is, beyond any reasonable doubt, negative: recursive procedure activations do not necessarily imply recursive procedure declarations. Put another way, procedures can be activated recursively even with a very restricted subset of ALGOL 60, from which all the possibilities of recursion have been removed. This means that the sentence added in last minute in the *Report* did not add recursive procedure activations to the language, as Naur claimed, and consequently that it did not add any burden for those who had to implement it. This also means that it was not even possible to implement the preliminary language ALGOL 58, defined one-and-a-half years earlier, without implementing recursive procedure activations, because it used the call by name mechanism as its sole parameter passing mode [28, Section 2.4.9].

Perhaps the most tangible proof of this is to look at the subsets of ALGOL 60 that were defined with the aim of simplifying the implementation of the language, and in particular to allow compilers to establish a fixed correspondence between identifiers and memory locations.[14] If it were because of the contentious sentence that recursive procedure activations were present in ALGOL 60, it would have been enough to simply remove it. On the contrary, to define the subsets, one had to *add* a number of sentences to the *Report* to make sure that no recursive procedure activations are possible. These sentences disallow recursion by stating that a procedure identifier has no meaning before the end of the procedure declaration,[15] or disallow mutual recursion either by requiring that procedures are declared before they are used[16] or by limiting the occurrences of identifiers in block heads,[17] or disallow self-application by requiring that all the parameters of a formal procedure be called by value,[18] or simply disallow recursive procedure activations explicitly:

> No call of the procedure itself may occur during the execution of the statements of the body of any procedure, nor during the evaluation of those of its actual parameters, the corresponding formal parameters of which are called by name, nor during the evaluation of expressions occurring in declarations inside the procedure.[19]

> Recursive calls on procedures are not permitted. A *recursive call* is defined as one where you have begun to set up parameters for a call of a certain procedure or have entered it, and then the procedure is entered again before it is exited. For example, [...] $f(x, f(y, z))$ would be a recursive call on procedure f, even if both of its parameters were called by value, since f is entered a second time while the parameters are being set up for the first call of f.[20]

The extreme level of detail of these sentences shows clearly that it was, as we have shown, actually quite difficult to remove recursive procedure activations from ALGOL 60. The last one of these sentences also shows that the situation described above with the *gips* procedure might even occur for parameters passed by value, if the procedure activations are created before the values of their parameters are calculated. In this case however, the existence of recursive procedure activations is not necessary: an alternative implementation scheme can be chosen, in which that order is reversed.

If the sentence added in last minute did not add recursive procedure activations to the language, did it at least add recursion to the language? We have assumed so far, in our thought experiment, that the answer to that question was positive, but it is actually negative. The semantics of procedure statements are indeed defined as follows in the *Report* [24, Section 4.7.3]: (a) each of the parameters called by value becomes a local variable, and is initialized to the value of the corresponding actual parameter, (b) each occurrence of the formal identifiers of the parameters called by name is replaced by the text of the corresponding actual parameter (before this operation, the identifiers that appear in the procedure body must, if necessary, be renamed, to avoid unintended identifications of originally distinct identifiers), and (c) the modified procedure body is inserted in place of the procedure statement and executed.

These semantics define, in particular, by means of substitution rules and without any ambiguity, the execution of a procedure statement '$p(\ldots)$' in the body of a procedure q. There are no reasons whatsoever, except the (wrong) impression that recursive procedure declarations make ALGOL 60 harder to implement, to believe that these rules do not apply when p happens to be identical to q, as Naur claimed when he wrote that there was an 'important lack of definition' for these identifiers, because the result of the above

[14]See for example [29], modification to Section 5.2 and modification (e) to Section 5.4.

[15][30], p. 72, Section 8.2, rule 2, and p. 75, Section 11.1.

[16][29], modification (b) to Section 4.7.

[17][31], modification to Section 5.

[18][29], modification (d) to Section 4.7.

[19][31, 32], modification to Section 4.7.5.

[20][29], modification (c) to Section 4.7.

operations is no less defined in that case. If, on the contrary, the semantics of a program had been defined by the replacement of all the procedure statements by the corresponding procedure bodies *before* the program execution, then there would have been at least a reason to think that recursive procedure calls are not possible, namely because this replacement process would create an infinite program text; but such is not the case. We can therefore conclude that the sentence was not only 'innocent', but that it was even redundant with the rest of the *Report*. It is thus clear that the introduction of recursion in ALGOL 60 was not at all an 'accident' or a 'coup', as Dijkstra claimed. This conclusion has already been stated by K. Samelson, when he wrote that the sentence was 'superfluous' and that 'saying nothing about recursivity automatically introduced it in full', but he did not argument it, besides declaring that 'the meaning of procedure [. . .] identifiers in procedure bodies was always clear by syntactic position' [33, p. 133]. Naur apparently changed his mind after reading Samelson's remarks, and said that recursion was 'almost the obvious thing', while maintaining that the contentious sentence 'is the way [recursive procedure activations] came in' [5, p. 159].

It is clear that recursive procedures were, in 1960, a controversial notion, incidentally for the same reasons that high-level programming languages were, at the same moment, controversial. The principal of these reasons was efficiency: in both cases a certain loss of computing power is unavoidable. It is, however, equally clear that recursion was present in ALGOL 60 from its inception. While it has seemed to many of those who were involved in its design that it was necessary to struggle to get it included in the language, what actually happened is that it has been necessary to struggle to *exclude* it from the language; this became clear when attempts were made to define subsets in which it was prohibited. This leaves us with a last question: where did recursion in ALGOL 60 come from?

It appears that recursion and recursive procedure activations have been introduced in ALGOL 60 because of the influence of mathematics on the design of the language. Its first declared objective was to be 'as close as possible to standard mathematical notation' [28, Section 1], and indeed both recursion and the call by name mechanism, that cannot be implemented without recursive procedure activations, are part of 'standard mathematical notations'. This is obvious for recursion, and can be seen, for the call by name mechanism, in the natural way in which arguments are handled in mathematics, namely by replacing, in the function definition, the occurrences of the symbol corresponding to an argument by the symbols of the argument itself. The same mechanism is used in multiple summations for example, or in the calculation of integrals. It is tempting to think that ALGOL 60 was influenced, more specifically, by lambda-calculus, for example through McCarthy, who was part of the committee. A. Church developed lambda-calculus about two decades earlier, and it was defined with rules that are similar to those of ALGOL 60 [34, Section 6]. However, McCarthy admitted that at that time he 'didn't understand' lambda-calculus besides the lambda-notation [1, p. 176, 35, p. 190], and the first implementations of LISP did not conform to its semantics [1, p. 180]. There is, to the best of our knowledge, no evidence of a direct influence of lambda-calculus on ALGOL 60: it is, for example, never evoked in the documents of the preparatory discussions. What happened is, most probably, that both of them were independently influenced by 'standard mathematical notations'. What can thus be done, and what has already been done by P. J. Landin, is to establish a correspondence between them [36, 37].

6. CONCLUSION

Contrary to what their names seem to indicate, recursive procedure declarations and recursive procedure activations are two logically disjoint concepts: neither of them implies the other. Both of them were introduced in ALGOL, and from then on in imperative programming languages, because of the influence of mathematics. Recursive declarations are, in the end, an innate characteristic of a language in which declarations are defined by a recursive syntax, and recursive activations are an innate characteristic of a language whose semantics are defined by substitution rules.

ACKNOWLEDGEMENTS

The author would like to thank K. R. Apt, M. H. van Emden, P. Klint, B. Le Charlier and P. McJones for their comments and suggestions during the preparation of this article, and the anonymous reviewers for their insightful remarks and recommendations.

FUNDING

This research was supported by the NWO (Netherlands Organisation for Scientific Research) Free Competition Grant number 612.001.003.

REFERENCES

[1] McCarthy, J. (1981) History of LISP. In Wexelblat, R.L. (ed.), *History of Programming Languages*, pp. 173–185. Academic Press, New York.

[2] McCarthy, J. (1959) On conditional expressions and recursive functions. *Commun. ACM*, **2**(8), 2–3.

[3] Naur, P. (1981) The European Side of the Last Phase of the Development of ALGOL 60. In Wexelblat, R.L. (ed.), *History of Programming Languages*, pp. 92–139. Academic Press, New York.

[4] Dijkstra, E.W. (1999) Computing science: achievements and challenges. *ACM SIGAPP Appl. Comput. Rev.*, **7**, 2–9.

[5] Naur, P. (1981) Transcript of Presentation. In Wexelblat, R.L. (ed.), *History of Programming Languages*, pp. 147–161. Academic Press, New York.

[6] Naur, P. (1962) The replies to the ALGOL-Bulletin 14 questionnaire. *ALGOL-Bull.*, **15**, 3–51.

[7] Soare, R.I. (1996) Computability and recursion. *Bull. Symb. Logic*, **2**, 284–321.

[8] Grassmann, H. (1861) *Lehrbuch der Arithmetik für höhere Lehranstalten*. Enslin, Berlin.

[9] Dedekind, R. (1888) *Was sind und was sollen die Zahlen?* Vieweg, Braunschweig.

[10] Peano, G. (1889) *Arithmetices Principia—Nova Methodo Exposita*. Bocca, Rome.

[11] Plato (ante 347 B. C.) *Hippias Major*.

[12] Plato (ante 347 B. C.) *Euthyphro*.

[13] Plato (ante 347 B. C.) *Meno*.

[14] Plato (ante 347 B. C.) *Theaetetus*.

[15] Aristotle (ante 322 B. C.) *Topics*.

[16] Kleene, S.C. (1952) *Introduction to Metamathematics*. North-Holland, Amsterdam.

[17] Tennent, R.D. (1981) *Principles of Programming Languages*. Prentice-Hall, Engelwood.

[18] Maurolico, F. (1557) *Arithmeticorum Libri Duo*. Senensem, Venice (1575).

[19] Pascal, B. (1654) *Traité du Triangle Arithmetique*. Desprez, Paris (1665).

[20] ben Gerschom, L. (1321) *Sefer Maassei Choscheb*, ed. and tr. G. Lange (*Die Praxis des Rechners*). Golde, Frankfurt (1909).

[21] Gödel, K. (1931) Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, **38**, 173–198.

[22] Gödel, K. (1934) *On Undecidable Propositions of Formal Mathematical Systems*. Institute for Advanced Study, Princeton. Mimeographed lecture notes by S.C. Kleene and J.B. Rosser.

[23] Hoare, C.A.R. (1981) The emperor's old clothes. *Commun. ACM*, **24**, 75–83.

[24] Backus, J.W. *et al.* (1960) Report on the algorithmic language ALGOL 60. *Commun. ACM*, **3**, 299–314.

[25] Kleene, S.C. (1936) λ-definability and recursiveness. *Duke Math. J.*, **2**, 340–353.

[26] Rutishauser, H., Ehrling, G., Woodger, M. and Paul, M. (1959) Procedures (and functions) as input parameters. *ALGOL-Bull.*, **8**, 5–6.

[27] Knuth, D.E. and Merner, J.N. (1961) ALGOL 60 Confidential. *Commun. ACM*, **4**, 268–272.

[28] Perlis, A.J. *et al.* (1958) Preliminary report—international algebraic language. *Commun. ACM*, **1**(12), 8–22.

[29] Bachelor, G.A. *et al.* (1961) SMALGOL-61. *Commun. ACM*, **4**, 499–502.

[30] Baumann, R. *et al.* (1962) ALGOL-Manual der ALCOR-Gruppe, Teil 3. *Elektron. Rechenanl.*, **4**, 71–85.

[31] AA. VV. (1963) ECMA Subset of ALGOL 60. *Commun. ACM*, **6**, 595–597.

[32] AA. VV. (1964) Report on SUBSET ALGOL 60 (IFIP). *Commun. ACM*, **7**, 626–628.

[33] Samelson, K. (1981) Comments of 1978 December 1. In Wexelblat, R.L. (ed.), *History of Programming Languages*, pp. 131–134. Academic Press, New York. Appendix 7 of [3].

[34] Church, A. (1941) *The Calculi of Lambda-Conversion*. Princeton University Press, London.

[35] McCarthy, J. (1981) Transcript of Presentation. In Wexelblat, R.L. (ed.), *History of Programming Languages*, pp. 185–191. Academic Press, New York.

[36] Landin, P.J. (1965) A Correspondence between ALGOL 60 and Church's Lambda-Notation: part I. *Commun. ACM*, **8**, 89–101.

[37] Landin, P.J. (1965) A correspondence between ALGOL 60 and Church's Lambda-Notation: part II. *Commun. ACM*, **8**, 158–165.