STICHTING
MATHEMATISCH CENTRUM
2e BOERHAAVESTRAAT 49
AMSTERDAM

MR 57

Generalized ALGOL

A. van Wijngaarden

1963

Reprinted from
ANNUAL REVIEW IN AUTOMATIC PROGRAMMING Volume III
PERGAMON PRESS
OXFORD · LONDON · NEW YORK · PARIS
1963

Generalized ALGOL

A. VAN WIJNGAARDEN
Mathematisch Centrum, Amsterdam, Netherlands

THE title 'Generalized ALGOL' of this paper needs an explanation. The word ALGOL is used because of the fact that many of the concepts of the language to be described can be found, partially at least, in ALGOL. On the other hand the generalization goes to such an extent that the connection with ALGOL can only be appreciated by those who know ALGOL quite well.

The main idea in constructing a general language, I think, is that the language should not be burdened by syntactical rules which define meaningful texts. On the contrary, the definition of the language should be the description of an automatism, a set of axioms, a machine or whatever one likes to call it that reads and interprets a text or program, any text for that matter, i.e. produces during the reading another text, called the value of the text so far read. This value is a text which changes continuously during the process of reading and intermediate stages are just as important to know as the final value. Indeed, this final value may be empty.

In order that such a language be powerful and elegant it should not contain many concepts and it should not be defined with many words. On the contrary by saying less one can say more, at least say more general things. Each definition in the language may restrict the set of meaningful texts. Without any definitions, however, one can only be absolutely silent in full generality. Of course, some compromise must be made in practice. This compromise has been made in ALGOL in a certain way. There are other ways, however, by which a better defined and more general language can be obtained using fewer concepts. In this short paper not a complete technical description will be given, but only some features will be described.

Let us first discuss the way in which such a syntax-free language might be described. Logically, the best way is to give the precise axioms or the precise description of the machine. However, such a definition would give little insight perhaps into the way in which one has to write a program in order to obtain a wanted result. Also in the definition of the language there might be a distinction between fundamental concepts and useful but logically unnecessary conventions. Hence, we rather see the language as a machine *M0* which is fed with the program at one end and produces the value at the other end. The rules of the language, i.e. a rough description of the working of *M0* is printed on the lid of the machine so that the user can easily see how to use the language. This description is quite simple, quite elegant in a way and will suffice in many cases, taking into account that the user will often take for granted that in the language certain expressions like $x + y \times z$ will stand for what he himself understands to be the meaning of $x + y \times z$. If, however, the user does not trust his intuition or does not understand what the short description on the lid implies in a particular case, he can open the machine to inspect the precise working. To his surprise, he finds that there are actually two machines inside, named *P1* and *M1*. The working of the machines is explained in much more detail on the lids of the machines. The machine *P1* is a so-called preprocessor, which chews the offered text and produces another text in a more basic language which is evaluated by the processor, i.e. machine *M1*. In the text offered to *P1* use is made of those conventions mentioned above, which are easy for the programmer but logically unnecessary. *P1* recognizes this use and translates the text into one in which those conventions are not used. This can be done before the evaluation of the text by *M1* is performed and the description of *M0* by splitting these two tasks is considerably simplified. Since the description of the action of *P1* and *M1* on the lids is much more basic it is also less easy to grasp its implications but it will settle many uncertainties left by the description of *M0*. Of course, this description of the action of *P1* and *M1* being in some language or another goes again only some way. In order to know what happens in cases which are still felt to be dubious one has to inspect the interior only to find that inside one finds again two machines, a preprocessor which translates a text into one written in a language with fewer concepts and a processor which processes this translated text. The functioning of these machines is described on the lids. It is again more primitive and it is harder to understand what it leads to but many more uncertainties are settled. Proceeding in this way one hits eventually machines which cannot be opened. Their working cannot be better explained than by the wording on the lid. If one does not understand it, that is a pity but one cannot go further than that. It uses the most primitive notions that one presupposes to be understood without further explanation. If we now describe a language defined by such a process, we start by describing it in very general terms and refer for a more detailed description to a forthcoming publication elsewhere.

Let us first define the concept of a name. There are basic symbols in the language, just as many as one likes. Some of them are peculiar, they are called ordinators. To them belong first the brackets, like ( ) [ ] ‘ ’, and, moreover, others like **if then else**, and maybe others. Then there are other distinguishable symbols like letters which have no inherent meaning but serve to build identifiers which may be used to denote variables. Also there are symbols which are used for very specific purposes, viz. for themselves like digits, logical values and operators. Which ones exist in the language is left to the programmer who is free to or rather has to define his own language in terms of some basic concepts which constitute the basic language. If, for instance, the programmer wants to use the symbols + and − he is free to do so but, of course, he has to define what he means. A sequence of letters and digits starting with a letter is an identifier, the simplest notation of a variable, i.e. a single entity or a linearly ordered set. The elements of such a set can be denoted by the variable followed by the number of the element in the brackets [ ]. Since the elements in themselves may again be such ordered sets one might have for example a, a[3], a[3][2]. If one does not like this notation but wants to denote the last example by a[3,2], then this is just a matter of convention language in which we are not interested here. One has only to instruct the preprocessor to replace a[3,2] by a[3][2] and then one can use that notation. Of course, in actually establishing a language internationally like ALGOL 60, one might wish to agree upon standard notations, but anyhow the definition of the basic machine should not be burdened by such unnecessarily complicated concepts as multiple indices in the bracket [ ].

There are also entities called constants, viz. sequences of symbols the meaning of which is defined by the sequences themselves. Constants are for instance digits, numbers, operators and strings. Open strings are sequences of strings and symbols other than ‘ and ’. A string is an open string enclosed in the bracket ‘ ’.

Again there are entities called metavariables, viz. sequences of letters possibly followed by digits all enclosed in the metabracket $\langle\ \rangle$, which denote sequences of none or more basic symbols.

Also there are entities called metaconstants viz. all basic symbols except the comma, strings, metaoperators as **value**, **in** and so on.

All these entities are examples of primaries. Simple names are formed by concatenation of primaries, e.g.

$x + y[2]$
$x := y[2]$
goto l

new *x*
if *a* **then** *x* **else** *y*
3 + 4 = 7
a **in** ⟨letter⟩

A simple name is a name. Also a name followed by a comma followed by a simple name is a name. A name enclosed in the bracket ( ) is a primary, another denotation of a set, the elements of which are the simple names which constitute the name and which are numbered from 1 onwards. Hence (*x* := *y*, **goto** L)[2] stands for **goto** L.

The fundamental concept of a program is now introduced. This is again a name. The value of the program is determined dynamically by the machine when reading the program. In order to find out how this value is to be found the machine examines regularly a sequence **V** consisting of truths separated by commas, i.e. a name, which is however precisely the value of the program as found so far and recorded by the machine! The examination proceeds as follows. Suppose the machine wants to determine the value of a certain name, **value** ⟨name 1⟩ say, in an obvious notation. It examines the simple names, the truths, which constitute **V** until it finds one which is applicable, i.e. which conveys information about ⟨name 1⟩, in order starting with the last one. If it finds such a truth, it applies it. Generally, the problem is not solved then since in the value so obtained the operator **value** occurs again, perhaps even more than once, which fact induces new evaluations until a name is obtained in which the operator **value** does no longer occur. As an example, suppose that **value** {*x* + *y*} had to be determined. By examining **V** the machine might find the truth

value {⟨sum 1⟩ + ⟨term 1⟩} = **value** {**value** ⟨sum 1⟩ + **value** ⟨term 1⟩}

In order to know whether this is applicable it would consult **V** to find out whether *x* **in** ⟨sum⟩ and *y* **in** ⟨term⟩ hold. It finds

x **in** ⟨letter⟩

hence it consults **V** in order to find out whether ⟨letter⟩ **in** ⟨sum⟩. It finds

⟨letter⟩ **in** ⟨identifier⟩

hence it consults **V** in order to find out whether ⟨identifier⟩ **in** ⟨sum⟩. Suppose it finds (we assume in this example that definitions analogous to those given in ALGOL are found in **V**)

⟨identifier⟩ **in** ⟨simple variable⟩
and again
⟨simple variable⟩ **in** ⟨variable⟩
and again
⟨variable⟩ **in** ⟨primary⟩

and again
⟨primary⟩ **in** ⟨factor⟩
and again
⟨factor⟩ **in** ⟨term⟩
and again
⟨term⟩ **in** ⟨sum⟩

then it has verified that x **in** ⟨sum⟩ and the machine starts to investigate whether or not y **in** ⟨term⟩. If it finds

y **in** ⟨letter⟩

then we know from the history of x that it will deduce indeed that y **in** ⟨term⟩, and hence that our truth is applicable if x is substituted for ⟨sum 1⟩ and y for ⟨term 1⟩. Hence it has to determine **value** x by consulting **V**, where it finds, let us say

$x=z$

which it applies by a built-in mechanism in stating

value x = **value** z

and now looking for the value of z. Suppose it finds in **V**

$z = 3$

then it knows

value x = **value** 3

and it proceeds to find the value of 3. Nothing is found which is applicable until on the very bottom of **V** it finds

value ⟨name 1⟩ = ⟨name 1⟩

It verifies that 3 is a name and hence it finds

value x = 3

Since the operator **value** no longer appears on the right-hand side the evaluation of x is ended, and now **value** y has to be found. Suppose in some way or another it finds eventually

value y = 4

then it knows that

value $\{x+y\}$ = **value** $\{3+4\}$

and it starts to determine the value of $3 + 4$. Suppose it finds in **V** before it finds

value {⟨sum 1⟩ + ⟨term 1⟩} = **value** {**value** ⟨sum 1⟩ +
  value ⟨term 1⟩}

which we know already to exist in **V**, the truth

$3 + 4 = 7$

then the same mechanism yields

value $\{x + y\}$ = **value** $\{7\}$

which is in the same way as above leads to

value $\{x+y\}$ = 7

One observes that the fact that **V** is consulted in a prescribed order prevents the occurrence of any contradictions. For instance, the truth that was found at the botton of **V**, viz.

value ⟨name 1⟩ = ⟨name 1⟩

is not in contradiction with any of the other truths since it can only be applied when nothing else is applicable. It necessitates however that the truths are in the correct order. If, for instance 3 + 4 = 7 were lower down in **V** than

value {⟨sum 1⟩ + ⟨term 1⟩} = **value** {**value** ⟨sum 1⟩ + **value** ⟨term 1⟩}

then the process mentioned above would never end! If on the other hand no information concerning y could be found in **V** then the machine would have found

value $\{x+y\} = 3 + y$

One sees that the action of the machine is determined by its built in mechanism and further by the truths it finds in **V**. Some truths are in **V** to start with. This list starts as follows:

value ⟨name 1⟩ = ⟨name 1⟩,
⟨sequence of none or more symbols not containing 'or'⟩ **in** ⟨proper string⟩,
⟨proper string⟩ **in** ⟨open string⟩,
'⟨open string⟩' **in** ⟨open string⟩,
⟨open string⟩⟨open string⟩ **in** ⟨open string⟩,
value {'⟨open string 1⟩'} = ⟨open string 1⟩,
value {'⟨open string 1⟩', ⟨sequence of symbols 1⟩} = {⟨open string 1⟩ **value** ⟨sequence of symbols 1⟩},

The main point is that the value of a string is the stripped string and that the value of a sequence of simple names separated by commas is the corresponding sequence of the values of these simple names separated by commas. However, one does not know yet what a simple name is and therefore this last rule has been specialized to the case that one has a string followed by a comma, and followed again by some sequence of symbols. This is sufficient for our purpose since now the computer can read the program and add its value to the text already existing in **V**. If the program starts with all the additional rules that one wants to have in the language, i.e. also for instance the rules governing arithmetic, all separated by commas and all enclosed in the bracket ' ' followed by a comma and followed again by other information, then the first result will be that the set of language rules will be added to the few that existed already in **V**. In that way the language is fully available for the following part of the information, which is presumably more the *ad hoc* part of the program, but of course may also contain new language rules. In this way, moreover, most of the task of the precise definition of the language is left to the programmer. Of course, some suggestions can be made as to how rules can be chosen in such a way that an elegant language is obtained. A piece could run as follows:

value {⟨simple name 1⟩, ⟨sequence of symbols 1⟩} =
  {**value** ⟨simple name 1⟩, **value** ⟨sequence of symbols 1⟩},
if {⟨name 1⟩ = ⟨name 2⟩} **in V then value** ⟨name 1⟩ =
  value ⟨name 2⟩,
value {⟨variable 1⟩ := ⟨expression 1⟩} = {⟨variable 1⟩ =
  value ⟨expression 1⟩},
if {⟨variable 1⟩ = ⟨variable 2⟩} **in V then**
  value {⟨variable 1⟩ := ⟨expression 1⟩} = **value** {⟨variable 2⟩ :=
  ⟨expression 1⟩},

The third and fourth simple name in this segment define the main part of what is called in ALGOL the meaning of the assignment statement, the procedure declaration without parameters, the procedure statement without parameters and the formal actual substitution.

Let us first consider a simple name like $s := x + y$. Its value is, if we assume that nowhere in **V** a truth of the form $s = t$ appears, according to the segment of **V** above, $s =$ **value** $(x+y)$ which may give rise to $s = 7$ in **V**. If one wants to express what in ALGOL would be expressed by

real procedure s; $s: = x + y$

then $s :=$ '$x + y$' does the job. Indeed this gives rise to $s = x + y$ in **V**. The procedure concept, at least without parameters, is therefore no longer needed. A name replacement like in the substitution of actual parameters for formal parameters in ALGOL, is simply done by $s :=$ 't'. This gives rise to $s = t$ in **V**. Suppose this actually appears in **V**. Then the value of $s := x + y$ will be, according to the last rule **value** {t $:= x + y$}, which will give rise to $t = 7$ in **V**. One sees that the substitution is not actually performed but that just a note is left in **V** which will yield the desired result. Also in the case that the value of the actual name is not defined but required the scheme works. For instance the value of $x := s$ would be $x =$ **value** s which gives $x =$ **value** t according to the second rule of the segment.

Before investigating how the parameters of a procedure are dealt with the concept of locality will be introduced. In ALGOL a declaration serves three purposes: it introduces an identifier which is local to a block, it restricts the use of that identifier to a particular class of entities, e.g. **real** a, **array** A[1:n], or again it can completely define the meaning of an identifier as in procedure declarations and switch declarations. We have seen already that this last function is superfluous, but the concept of locality is useful. We shall not deal here with the concept **own**. The concept block in ALGOL as a sequence of statements, preceded by declarations and embraced in the bracket **begin end** is too special for our purpose since even the concept statement does not exist here. Hence we shall define that inside the bracket ( ) local identifiers can be introduced by the simple name **new** ⟨identifier⟩ with the following meaning. The value of an opening parenthesis (is, in **V**, {(⟨integer⟩,}. The integer is determined by consulting **V**. If no simple name of the same form is found, it is 1, else it is one more than the integer found in that simple name. The value of **new** ⟨identifier 1⟩ is **new** ⟨identifier 1⟩ ℘ ⟨integer 1⟩ where ⟨integer 1⟩ is the integer found following (by consulting **V**. Here ℘ is a letter, which is chosen as one which is not likely to be used by the programmer normally. Since it is a letter, however, the sequence ⟨identifier 1⟩ ℘ ⟨integer 1⟩ is again an identifier. The evaluation of a name is now redefined to the extent that the evaluation of a variable, the identifier of which, ⟨identifier 1⟩, does not end in ℘ ⟨integer⟩, causes first the identifier to be extended with such an ending. The extended identifier is found by consulting in **V** the simple name of the form **new** ⟨identifier 1⟩ ℘ ⟨integer 1⟩. The value of the closing parenthesis) is defined as follows. **V** is consulted until (⟨integer 1⟩ is found. The simple names in **V** are now scanned in the advancing order. If in a simple name ⟨identifier⟩ ℘ ⟨integer 1⟩ is found not inside a string then that simple name together with a separating comma is deleted. If this is not the case but if the simple name is a relation like x ℘ $14 = 3$ then a copy is inserted together with a separating comma directly after the last comma which precedes (⟨integer 1⟩ in **V** after which it is itself deleted together with a separating comma. After this process (⟨integer 1⟩ in **V** is replaced by (and the closing parenthesis) is added to **V**. If in this way **V** would end with ( ) then these symbols are deleted. This seemingly long definition of the value of the pair of parentheses has quite a lot of useful consequences. Let us first consider the concept function designator as it occurs in ALGOL. It assumes the existence of a procedure declaration in the body of which there occurs an assignment to the procedure identifier. A simple example is for instance given by the ALGOL declaration

real procedure P ; **begin real** *s*; *s* := **if** *x* > 0 **then** *y* **else** *z*;
P := *s* × (*s*+1); *i* := *i*+1 **end**

where the procedure P uses the non-local variables *x*, *y*, *z* and *i* and has a side effect on *i*.

This would run in our new language

new P, . . ., P := '(**new** *s*, *s* := **if** *x* > 0 **then** *y* **else** *z*,
s × (*s*+1), *i* := *i*+1)'
. . ., *u* := 3 × P

where . . ., may either stand for empty or for a sequence of simple names separated by commas and opening parentheses. One may very well change the meaning of P by another assignment, since there is no logical distinction between the assignment above and, for instance P := 3.14. 'Assignment to the procedure identifier' is not necessary since the value of P is automatically delivered by the process described, enclosed in parentheses. The arithmetic in 3 × P does away with the parentheses. However, this value should probably occur only once between the parentheses since otherwise a set would be delivered like, for example (3, 4, 2). In itself such a set may of course quite well be the value of P, but it is not the same as the last value (2). The following example has no counterpart in ALGOL.

P := '(10, *n* := *n*+1, **if** n > 20 **then** P := 25)'

This has the effect that the value of P in an expression will be 10 until *n* has surpassed 20, from where onwards P has the value 25, but no counting and testing will be done anymore!

At last we shall show that the parameters in a procedure can just be dealt with by the following two simple preprocessing rules.

Replace
⟨identifier 1⟩ (⟨name 1⟩) := '(⟨name 2⟩)'
by
⟨identifier 1⟩ := '(⟨name 3⟩)'
where ⟨name 3⟩ is found by replacing in ⟨name 2⟩ each simple name which is identical with (⟨name 1⟩)[⟨integer 1⟩] by ℘ ⟨identifier 1⟩ [⟨integer 1⟩]

Replace in any other occurrence
⟨identifier 1⟩ (⟨name 1⟩)
by
(**new** ℘ ⟨identifier 1⟩, ℘ ⟨identifier 1⟩ := (⟨name 1⟩), ⟨identifier 1⟩)

This process is perhaps best illustrated by an example.

P(*u*,*v*,*w*) := '(⟨text containing the identifiers *u*,*v*,*w*⟩)'

will be replaced by

P := '(⟨text containing the identifiers ℘ P[1], ℘P[2], ℘P[3] instead of *u*,*v*,*w*⟩)'

The 'function designator' or 'procedure statement'

P(*a*, '*b*', *c*[*i*, '*j*'])

will be replaced by

(**new** ℘P, ℘P := (*a*, '*b*', *c*[*i*]['*j*']),P)

Effectively, those actual parameters which are not in quotes are called by value, those which are in quotes are called by name, whereas even mixed cases can occur as is shown in this example. This way of dealing with the call by name-value-concept, viz, taking the decision on the call side rather than on the declaration side like in ALGOL has great advantages. Often it is only clear on the call side whether or not it is necessary or advantageous to call by name but also one declaration will now do for many different uses.

As a matter of fact, in the presentation given here, some things have not been defined, although explicitly used under the assumption that either the reader would be willing to accept that a proper definition could be given and would give intuitively the desired interpretation or that he would not recognize the difficulty. It may seem clear, however, that an elegant, flexible, and powerful language can be defined with a great rigour using the techniques described above.