A macro-generator for ALGOL

by H. LEROY

Compagnie Bull—General Electric
Paris, France

INTRODUCTION

The concept of macro-facility is ambiguous, when applied to higher level languages.

For some authors\(^1,2\), a macro-facility is essentially a means to define, inside a program, local extensions to the language in which the program is written. It is usually understood that these definitions are interpreted before execution, but this is an implementation feature rather than a language feature, and, in this sense of the concept of macro-facility, the procedure mechanism of ALGOL must logically be regarded as a macro-facility.

In the macro-generator defined in this paper, there is no facility for defining new linguistic entities in ALGOL. It is a mechanism of generation of ALGOL programs, programmed in a special language—macro-ALGOL, and the fact that an ALGOL program is completely generated before being executed is an essential linguistic feature of macro-ALGOL.

The motivation here is not, therefore, the comfort of the programmer. The macro-generator is intended to be useful in situations where the production of an ALGOL program by human programming power would be too expensive or nearly impossible, or where its mechanical production would mean a substantial reduction of cost. Examples of such situations are:

- The production of a very big and complex program with relatively simple specifications, the complexity of the program arising from the fact that conceptually distinct specifications must be inextricably mixed up in the program.
- The production of many programs with small differences in their specifications, these differences implying, however, substantial differences in their structures to preserve efficiency.
- In heuristic programming, where many algorithms must be tested before one is found whose behavior is satisfactory. In this case, the ease of modification can only be achieved if each version of the algorithm can be programmed mechanically from a convenient specification.

Our system is therefore intended to be a system of computer aided ALGOL programming, and its expected advantages are those of any mechanization. Just as a program is better executed by a machine than by a man, the production of a program or a set of programs can involve a lot of repetitive thinking, and that part of the programming work can best be done by a machine.

Characteristics

It is clear that, whatever the system used, the programmer who instructs a macro-generator to produce a program must think at a level higher in abstraction, at least, than when he writes a program with his own hand. Macro-programming is therefore unavoidably more difficult than direct programming. It is suggested that a macro-generator cannot be significantly useful if it is not very powerful and flexible. For this to be achieved, we assume that the following requirements must be met, at least.

1. Full power of the macro-generator in general purpose information processing

This requirement might look strange at first sight. It is quite natural, on the contrary: for example, the generation of a polynomial approximation requires the computation of the numerical coefficients, and maybe the degree, of the polynomial, and this computation may be of any complexity.

We will give this requirement a more precise form: a generator of ALGOL programs must be able to do anything that can be programmed in ALGOL.

This immediately suggests that macro-ALGOL, the programming language of the macro-generator, must be a true extension of ALGOL.

2. Conceptual economy in the definition of the generating mechanism

In other words: the extension that makes ALGOL into macro-ALGOL must be defined with as few ad-
ditional notions as possible. This means that macro-ALGOL must be a natural and straightforward extension of ALGOL or, more precisely, that most generating mechanisms of macro-ALGOL be natural and straightforward generalizations of the mechanisms already present in ALGOL.

3. Syntax independence

The programmer is interested in the semantical contents of programs, and not in their representations as strings of basic symbols. This implies that the macro-generator must not be a generator of arbitrary strings of basic symbols. The ability of generating strings which would not conform to the ALGOL syntax can be of no use, and if it did exist, then the macro-programmer would be entirely responsible for the generated string obeying the rules of ALGOL syntax. Since these rules are defined once and for all, the macro-generator can, and therefore must take care of them.

It is not even necessary to assume that the generated program will always appear in the form of a string in the ALGOL syntax. If it must be immediately interpreted by a mechanism, then another form would probably be preferable.

The operation of the macro-generator can therefore be defined in terms of the abstract structure of the generated program. If this abstract structure must be translated into a string, then the implementor of the macro-generator, and not the macro-programmer, is left the choice of one string among the strings which represent this structure.

Furthermore, the implementor is left free to take advantage of obvious cases of semantical equivalence: for example, he can detect and erase all unlabelled dummy statements, replace ‘a+O’ by ‘a’ and ‘-(-a)’ by ‘a’.

The idea of syntax independence has been inspired by the paper of P. J. Landin.

4. Mechanical generation of macro-programs

It may be necessary to perform the generation of an ALGOL program in several successive steps. This is possible if the macro-generator can generate any program in macro-ALGOL, that is in its own language. We can now concentrate on macro-ALGOL for itself, as a programming language in its own right, with a property of macro-generation. We can even define a macro-language, without reference to any existing language, thus:

'A macro-language is a programming language such that any program in this language, when executed, possesses a value, which is a program in the same language.'

'A macro-generator is the processor of some macro-language.'

(The value of a program is obviously what we called the generated program.)

Macro-ALGOL contains a distinguished subset, namely ALGOL. This subset has the following property:

'The value of a program in ALGOL is always equivalent to a dummy statement, i.e., a program whose execution has no effect, besides producing another dummy statement as value.'

We can now envision the generation of an ALGOL program, say in 2 steps:

- A program is written, by hand, in macro-ALGOL.
- This program is executed by the processor of macro-ALGOL (the ALGOL macro-generator), with a set of input data. This yields various results, known as output data, and a distinguished output, which is the value of the executed program.
- The value just obtained is executed, as a program, by the macro-ALGOL processor, with another set of input data, which yields another value, which turns out to be a program in ALGOL.
- The value just obtained is executed, as a program, by the macro-ALGOL processor. The value obtained is a dummy statement.

Further executions would have no other effect than producing dummy statements as values.

5. Efficient implementation

It must be pointed out that, until now, no assumption has been made on the implementation of macro-ALGOL. We have considered the macro-generator as an ideal machine whose machine language is macro-ALGOL. However, any practical implementation of macro-ALGOL will make use of another machine, with a compiler of macro-ALGOL built for this machine. Therefore, what we called the execution of a program in macro-ALGOL will take place in two steps:

- compilation of the program, i.e., its translation into the language of a machine M;
- execution, by machine M, of the result of the translation. This execution involves, in general, ordinary input-output activities, and always the output of a program in macro-ALGOL.

Furthermore, this implementation must be efficient. More precisely:

'The object program produced by the compilation of an ALGOL program by the macro-ALGOL compiler is as efficient as the object program produced, for the same machine, by an ordinary ALGOL compiler.'
Outline

The extension that makes ALGOL into macro­ALGOL is the introduction of a new class of identi­fiers, namely symbolic identifiers. Whether an identifier is symbolic or not is determined by its declaration.

A non symbolic identifier is treated exactly like in ALGOL.

A symbolic identifier is treated in the following way:

If it is a variable identifier, the variable is never assigned a value. When it occurs as the left part of an assignment statement, the assignment is not performed, but an assignment statement is produced, with the same identifier as left part, and put in the generated program. If it occurs in an expression, it appears in an expression which is produced as value of this expression, and put in the generated program.

For example, if $n$ is a symbolic identifier, the statement

$$n := n + 1$$

has no other effect than producing the statement

$$n := n + 1$$

If $a$ is an ordinary variable whose current value is 5, the statement:

$$n := n + a$$

produces the statement

$$n := n + 5;$$

If it is a label identifier, a go to pointing to it is not executed and produces a go to pointing to the same label;

If it is a procedure identifier, any call to this procedure produces another call to the same procedure.

The declaration of a symbolic identifier produces a declaration of an identifier, which may again be symbolic or not. The declaration of an ordinary identifier produces no declaration in the generated program.

One of the most powerful generation mechanisms is the procedure mechanism applied to non-symbolic procedures. When a non-symbolic procedure is called, the copy rule is applied, and the resulting statement is processed as if it occurred at the same place in the program being executed. This gives rise to various effects, e.g., input of external data, changing the values of ordinary variables, output of data), and to the production of a statement, if the statement resulting from the copy rule contained symbolic identifiers. If the procedure is recursive, the statement produced can have an infinity of forms, depending on the point from where it is called, and on external data. Conversely, the use of recursive procedures allows the production of any statement whose structure depends on external data, provided the rule of dependence is a computable function.

In the following formal definition, the processing of declarations, statements and expressions is completely defined. Besides other effects, a value is defined for each of them (which defines the value of a program as a special case of the value of a statement).

Furthermore, a generalized type is defined for each expression. Since the value of an expression is, in general, an expression containing identifiers, the types of ALGOL must be generalized. Just as, in ALGOL, the type integer is distinguished from the type real, the types integer, real, Boolean are distinguished, in macro-ALGOL, from the types integer expression, real expression, Boolean expression. In order to meet the efficiency requirement, the generalized types of expressions are defined in such a way that they can be determined at compile-time.

Declarations

Simple variables

An ordinary simple variable is declared and treated exactly like in ALGOL 60. Its declaration has no value.

A symbolic simple variable is declared like a simple variable in ALGOL 60, except that its identifier is immediately preceded by one or more occurrences of the basic symbol symbol. The declaration of a symbolic simple variable has a value, which is the declaration obtained by deleting one occurance of symbol.

Examples:

integer $x$
declares $x$ as an ordinary simple variable of type integer. This declaration has no value.

integer symbol $y$
declares $y$ as a symbolic simple variable of type integer. The value of this declaration is:

integer $y$

The declaration

integer symbol symbol $z$
declares $z$ to have the same properties as the above declared $y$. The value of this declaration is:

integer symbol $z$

Arrays

An ordinary array is declared and treated exactly like in ALGOL 60. Its declaration has no value.

A symbolic array is declared like in ALGOL 60, except that the array identifier is immediately preceded by one or more occurrences of symbol, and that any bound pair may be preceded by macro.
If the array identifier is preceded by more than one occurrence of symbol, the value of the declaration is the same declaration, where one occurrence of symbol has been deleted, and where all bound pairs are replaced by their values.

If the array identifier is preceded by exactly one occurrence of symbol, the value of the declaration is obtained by the following process:
1. The unique occurrence of symbol is deleted.
2. All expressions in bound pairs are replaced by their values.
3. If a bound pair is preceded by macro, the array identifier is replaced by a list of identifiers, one for each possible value of the corresponding subscript, and a note is taken of the correspondence. The bound pair with the symbol macro is deleted. The process is repeated until there remain no macro-bound pairs.
4. The result is the value of the initial declaration (with the reservation that, if no bound pair is left, the symbol array is deleted).

**Examples:**

real array A [1 : n] declares A as an ordinary array identifier, and has no value (note that, if n is a symbolic identifier, the effect of this declaration is undefined).

real array symbol A [1 : n] declares A as a symbolic array identifier. If n is a symbolic identifier, the value of the declaration is:
real array A [1 : n]
If n is an ordinary identifier whose current value is 3, the value is:
real array A [1 : 3]
Under the same hypothesis about n, the value of:
real array symbol A [macro 1 : n]
can be:
real A1, A2, A3 where the 3 identifiers are chosen distinct from any other declared identifier.

**Procedures**

If a procedure is declared with a type, the type may be real, integer, Boolean, real expression, integer expression, Boolean expression.

Besides this extension of type, an ordinary procedure is declared like in ALGOL 60. The declaration has no value.

A symbolic procedure is declared with one or more occurrences of symbol preceding the procedure identifier. The value of the declaration is a procedure declaration with the same procedure heading, and whose body is the value of the body of the initial declaration. In the evaluation of the procedure body, all formal parameters are treated as symbolic identifiers.

**Examples:**

procedure P ; n : == n + 1 declares an ordinary procedure. (Note that n can be a symbolic identifier).

procedure symbol Q ; n : == n + a
If n is a symbolic identifier, and a an ordinary variable whose current value is 3, the value of this declaration is:

procedure Q ; n : == n + 3

**Labels**

A label declaration is the label identifier followed by a colon. The rules are the same as for simple variables.
Example: The value of:

symbol A : B : symbol symbol C :

is:

A : symbol C :
preceding the value of the statement following the initial declaration.

**Switches and own variables**

Switches and own variables are not considered here.

**Expressions**

We assume that a designational expression always reduces to a label.

The type of an expression other than designational can be:

integer : its value is an integral number;
real : its value is a real number;
Boolean : its value is a logical value;
integer expression, real expression,
Boolean expression : its value is an expression of the indicated type in which all identifiers are symbolic.

The type and value of an expression are defined recursively as follows:
1. The type and value of a number or logical value are as defined in ALGOL 60.
2. The type and value of an ordinary simple variable are as defined in ALGOL 60.
3. The value of a symbolic simple variable is the variable itself. Its type is:

   Boolean expression if the declared type of the variable is Boolean;
   integer expression if the declared type of the variable is integer;
real expression if the declared type of the variable is real.

3. The type and value of a subscripted variable with an ordinary array identifier are as defined in ALGOL 60.

The value of a subscripted variable with a symbolic array identifier is a subscripted or simple variable obtained by evaluating all subscript expressions, removing the macro-subscripts if any, and replacing the array identifier by the identifier corresponding to the values of the macro-subscripts. It is defined only when all expressions in macro-subscript positions are of type real or integer. Its type is determined the same way as the type of a simple variable.

4. The type of a function designator with an ordinary procedure identifier is the declared type of the procedure. Its value is the value of the expression obtained by application of the copy rule.

If the procedure identifier is symbolic, the value is a function designator with the same procedure identifier, and the values of the actual parameters as actual parameters. The type is:

Boolean expression if the type of the procedure is Boolean or Boolean expression etc... .

5. The value and type of an expression consisting of a unary operator applied to an expression are defined as follows:

- if the operand expression is of type Boolean, integer or real, the rules of ALGOL 60 are applied;
- if the operand expression is of type Boolean expression, integer expression or real expression, the type of the expression is the type of the operand.

The value is equivalent to the expression consisting of the same operator applied to the value of the operand.

6. The value and type of an expression consisting of a binary operator applied to two expressions are defined as follows:

- if both expressions are of type Boolean, integer or real, the rules of ALGOL are applied;
- if at least one of the two expressions is of type Boolean expression, integer expression or real expression, the value is an expression equivalent to the expression consisting of the same operator applied to the values of the operands. The type is always Boolean expression, integer expression or real expression, the first symbol being determined by the rules of ALGOL 60.

7. The value and type of a conditional expression are determined as follows: let

\[ \text{if } B \text{ then } E_1 \text{ else } E_2 \]

be the conditional expression;

- if \( B \) is of type Boolean, the value is the value of \( E_1 \) if the value of \( B \) is true, the value of \( E_2 \) if the value of \( B \) is false. If \( E_1 \) is a Boolean expression the type is the type of \( E_1 \) \( \lor \) \( E_2 \). If \( E_1 \) is an arithmetic expression, it is the type of \( E_1 + E_2 \).

- if \( B \) is of type Boolean expression, the value of the expression is:

\[ \text{if } V_B \text{ then } E_1 \text{ else } E_2 \]

where \( V_B, E_1, E_2 \) are the values of \( B, E_1, E_2 \). The type is determined as in the first case.

8. The value of an ordinary label is undefined. The value of a symbolic label is the label itself.

9. Remark: The value of an expression is defined up to an equivalence. It is therefore left to each implementor to decide whether, for example:

\[ +A \]

is to be replaced by \( A \);
\[ -(A) \]

is to be replaced by \( A \);
\[ A + 0 \]

is to be replaced by \( A \);
\[ 2 + 3 \]

is to be replaced by \( 5 \) (where 2 and 3 are the values of expressions of type integer expression), etc...

Statements

Since a statement has always a value, we shall speak indifferently of execution or evaluation of a statement.

1. Assignment statement

If all left part variables are ordinary variables, and if the right part expression is of type integer, real or Boolean, the statement is executed according to the rules of ALGOL 60. The value is a dummy statement.

If all left part variables are symbolic, the value is an assignment statement with the values of left part variables as left part variables, and the value of the expression as right part expression.

In all other cases, the effects and value of an assignment statement are undefined.

2. Go to statement

If the label is an ordinary label, the statement is executed as in ALGOL 60. Its value is a dummy statement.

If the label is symbolic, the value is a goto statement with the same label.

3. Conditional statement

If the Boolean expression is of type Boolean, the first statement or second statement (if any) is executed.

If the Boolean expression is of type Boolean expression, the components of the conditional statement are evaluated in sequence. The value is a statement equivalent to the conditional statement built with these components.

4. Procedure statement

The rules are the same as for a function designator, except that no type is defined.
5. For statement

Let
\[
\text{for } V := e_1, e_n \ldots e_n \text{ do } S
\]
be the for statement.

If \( V \) is a symbolic variable, all expressions in the for list and the statement \( S \) are evaluated. The value is a for statement with the values of these components as corresponding components.

If \( V \) is an ordinary variable, and if \( n > 1 \), the value is that of the statement:
\[
\text{begin } \text{for } V := e, \text{ do } S ; \ldots ; \text{for } := e_n \text{ do } S \text{ end}
\]

If \( n = 1 \), 3 cases are possible:

1. \( e_i \) is an arithmetic expression. The value is the value of the statement:
\[
\text{begin } V := e_i ; S \text{ end}
\]
2. \( e_i \) is of the form \( A \text{ step } B \text{ until } C \). The value is the value of the statement:
\[
\text{begin } V := A ;\quad L : \text{if } (V-C)x \text{ sign } (B) > 0 \text{ then goto } L1 ;\quad S \text{ ; } V := V + B ; \text{ goto } L1 ;\quad \text{Element exhausted:}
\]
3. \( e_i \) is of the form \( E \text{ while } F \). The value is the value of the statement:
\[
\text{begin } L3 : V := E ; \quad \text{if } ] \quad \text{then goto } L3 ;\quad S ; \text{ goto } L3 ;\quad \text{Element exhausted:}
\]

6. Dummy statement

The value is a dummy statement.

7. Compound statement

The statements of the sequence are evaluated from left to right, until a go to statement with an ordinary label is encountered, or the sequence is exhausted.

In the first case, if the label is declared inside the compound statement, the evaluations are resumed from the point where it is declared. If the label is declared outside, or in the second case, the evaluation of the compound statement is completed, and its value is equivalent to a compound statement made of the sequence of values obtained thus far.

8. Block

The declarations are evaluated. If there is at least one value declaration, the value is a block with the value declarations as declarations and the value of the statement sequence, treated like a compound statement, as statement sequence. If there is no value declaration, the value is equivalent to the value of the statement sequence treated like a compound statement.

Examples

1. Generation of a polynomial. We assume that the degree and coefficients of the polynomial are unknown when the generating program is written. The degree is the current value of the ordinary variable \( n \), the coefficients are the elements of the value of the ordinary array \( a \), and the variable is the symbolic identifier \( x \).

We write
\[
p (x, a, n)
\]
where we want the polynomial to be generated: the polynomial is the value of the above function designator.

The procedure \( p \) has been declared thus:
\[
\text{real expression procedure } p (X, A, N);\quad \text{value } N;\quad \text{real expression } X;\quad \text{array } A;\quad \text{integer } N;
\]
\[
p := \begin{cases} \text{if } N = 0 \text{ then } A[0] \text{ else } p(X, A, N-1)X + A[N] \end{cases}
\]

If the current degree is 2, and if the current coefficients are 1, 2, 3, the application of the copy rule to \( p(x, a, n) \) will yield
\[
p := \begin{cases} \text{if } N = 0 \text{ then } A[0] \text{ else } p(x, a, n-1)X + a[n] \end{cases}
\]

(ince 'n = 0' is of type \text{Boolean}, and its value is \text{false}, the value of this expression is the value of:
\[
p (x, a, n-1)X + 1
\]
(since 1 is the value of \( a[2] \)).

The value will be:
\[
(3 X x + 2) X x + 1
\]

2. Generation of the statement:
\[
\begin{align*}
\text{if } B (1) \text{ then } S (1) \text{ else } \\
\text{if } B (2) \text{ then } S (2) \text{ else } \\
\end{align*}
\]
\[
\begin{align*}
\text{if } B (n) \text{ then } S (n) \text{ else error}
\end{align*}
\]

where \( n \) is unknown. We assume that \( B (n) \) and \( S (n) \) are computable and defined recursively. This statement is the value of the procedure statement:
\[
q (1, n)
\]

where \( q \) is declared thus:
\[
\text{procedure } q(k,n);\quad \text{value } k,n;\quad \text{integer } k, n;\quad \\
\text{if } k = n + 1 \text{ then error else if } B (k) \text{ then } S (k) \text{ else } q (k + 1, n)
\]

Comments

1. Implementation

The compiler of macro-ALGOL will produce machine-code for all executable parts of the macro-program, and calls to a subroutine for operations of symbolic arithmetic and program generation.

It is important to note that all symbolic results are always on top of the stack, since there are no variables with symbolic values.

The execution of the machine-code produced by the compiler never calls for a compilation process: this
is because the symbolic nature of an identifier never changes during execution of a program. This would not be the case with a language which would have a self-generation facility (i.e. the possibility, for a program, of generating itself during its execution).

2. Completeness

A macro-language is complete if any possible program generation algorithm can be programmed in this language.

As defined above, macro-ALGOL is certainly not complete. Its main weakness is in the generation of declarations, where the only non-trivial mechanism is that of macro-bound pairs, which provides only for the generation of an unknown number of simple variables, or arrays of one same dimensionality and size.

Part of the necessary extensions could be used in direct programming: these extensions would be general mechanisms of recursive definition of declarations, which would yield a means of declaration of arbitrary composite types.

From another point of view, the fact that variables of types integer expression, etc. . . , do not exist, might seem to make the language essentially incomplete. However, the introduction of such variables is probably not necessary. The reason for this is that a variable is fundamentally necessary only when a result must be used more than once. A symbolic result must appear, sooner or later, in the generated program, and it is not good practice, in general, to write a given expression or statement more than once in a program.

It remains that, in the first example above, the polynomial could be more efficiently generated by a for statement than by a recursive procedure. However, the auxiliary variable which would take on, as values, polynomials of increasing degrees, would not serve the general purpose of a variable: its value would always be used only once. Some kind of 'for expression,' leaving the iterated result anonymous, would be a better solution, and could be used to advantage in direct programming as well.

REFERENCES

1 T. E. CHEATHAM, JR.  
The introduction of definitional facilities into higher level programming languages  
AFIPS, 29, FJCC (1966)

2 B. M. LEAVENWORTH  
Syntax macros and extended translation  
Comm. A.C.M., 9, 11 (1966)

3 P. J. LANDIN  
The mechanical evaluation of expressions  
Computer Journal 6, 4 Jan. (1964)

4 J. W. BACKUS, et al.  
Revised report on the algorithmic language ALGOL 60  
Ed. P. Naur Comm. A.C.M. 6,1 (1963)