STICHTING
MATHEMATISCH CENTRUM
2e BOERHAAVESTRAAT 49
AMSTERDAM

DR 37

Criteria for the Design of a Language.

B.A. Galler

and

A.J. Perlis

1966

Criteria for the Design of a Language

B.A. Galler and A.J. Perlis

Because of the rapid development and dissemination of ideas in programming, the exact source of important concepts is difficult to identify. Most of the ideas outlined here did not originate with the authors of this paper, or even the papers in which they came to the authors' attention -- nor does it matter that this is so. However, the immediate stimulus for this paper came from ALGOL X proposals described in lectures at the Mathematisch Centrum, Amsterdam, by A. van Wijngaarden.

We propose here some criteria for language design. These criteria are used as guidelines (although not always explicitly) in specifying a part of an ALGOL-like language. The criteria are:

(1) Minimization of the number of concepts.

(2) Uniformity of treatment of concepts, when possible.

(3) Scope of application of operators, declarations, etc., determined dynamically rather than lexicographically.

(4) The order in which things happen must be known and taken into account; concurrent computation occurs only when explicitly indicated.

(5) In place of any construction its name may be used; and conversely.

(6) Wherever a name occurs, it does so as a variable, for which an appropriate set of operations and a structure for its value should be provided.

(7) The attributes of a variable shall be available to the program and shall be interpreted dynamically when necessary.

(8) The description of the language should distinguish carefully between temporal and lexicographic sequencing. We shall use "before", "after", "precede", "follow", "predecessor", "successor" for temporal sequencing; and "above", "below", "left", "right" and "next" for lexicographic sequencing.

The basic idea is that execution of a program is just the evaluation of an expression via evaluation of its subexpressions. During the evaluation of one expression a postponement of an activity may occur while another (and yet another) expression is evaluated. Thus, the language must provide for "pushing" and "popping" the status of certain variables when needed. It is also necessary to have a rule indicating the next expression to be evaluated whenever one such evaluation is completed. The computation ceases only when no successor can be found.

It is usual to distinguish between two kinds of computation: (1) manipulation of values of variables introduced in the program, and (2) manipulation of variables known only to the system, such as the type of declared variables, or storage allocation and accessing information. It is clearly undesirable to continue this distinction, since it adds greatly to the power of the programming language if the "program" can access the system variables. The use of declarations has enforced this distinction by suppressing all names for system variables. Declarations have also been treated as quite static relative to the computation; although they may be invoked dynamically in ALGOL, they remain in force over the life of a block and may not be changed in the block. We assume instead the existence of a set of system variables (known as "bar variables" since they all have the form | n, where n is an integer), which are interpreted by the system and computable by the program.

Related to the bar variables (in that they are interpreted by the system) are the "attributes" of declared variables. A variable is "declared" by assigning to the system variable new (which could be one of the bar variables) a list of names. If a declaration, such as real, precedes a name, it shall be an initial assignment for that attribute for all variables to the right of it in the list, until another value for that attribute occurs in the list. With each declaration any previous values and/or attributes of the variables on the list are "pushed down", and a new set of values and attributes is created for each. They will have initial "default" assignments, but these may be manipulated by references to type of x, etc. The program may create

additional attributes of its own by using the operators attr 1, attr 2, etc. These may be assigned arbitrary values of type real, provided the variable so referenced has been declared new. All variables declared new since a given "push" are "popped" (along with all values and attributes) when the corresponding "pop" occurs.

Variables have a type: real, Boolean, complex, integer, form, string, etc., and a structure, such as individual, row, array, etc. Default values of type and structure are real and individual, resp. The default value of a variable is undefined. Expressions will be constructed as in ALGOL from these types and structures. In addition, we define a class of unary operators: x: = and $x[\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n]$: = , where x is any variable and ε_i is an arithmetic expression. These unary operators are to have a very low precedence ranking relative to all other operators. Thus, the parsing a + (b: = (c + (d: = (e + f)))) is implied on the expression

a + b: = c + d: = e + f

and the value of the expression is a + c + e + f, while during the evaluation the values e + f and c + e + f would be stored in d and b, respectively. Components of a row-structured variable x may be selected by the use of subscript expressions, using any combination of the two notations: x[i], x[i,j], [i] of x, [j] of [i] of x, etc. The combination ([j] of i of x[k,l])[m] is equivalent to writing x[k,l,i,j,m]. (The symbol of may also be omitted.) An explicit row-structured expression has the form

<type symbol> $(\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n)$.

The type symbol is omitted in declarations.

Subscripts may be used on explicitly written row expressions, since any row-structured quantity must allow component selection. Since an explicit row-structured expression is preceded by a type symbol, any operation to be performed on the expression other than subscription is not performed component-wise, but according to the appropriate treatment of a quantity of that type. Thus, the complex number 3 + 4i would be written complex (3,4), as in:

complex w,z; w:= z + **complex** (3,4)

Evaluation of Expressions

Expressions not containing semi-colons, commas, or periods are evaluated according to the precedence ranking imposed by the ALGOL syntax specifications. Evaluation of a parenthesized expression is considered a postponement of activity, as is any computation needed to produce a value for an identifier. The postponement ends when a value is produced, and at that time any lists assigned as values to the variable **new** during the postponement are "popped" so that the value of **new** is the same as it was before the postponement. If, during a postponement, a sub-expression is completely evaluated, then if there is a period or right parenthesis to its right, the (most recent) postponement is terminated, returning as value the value of the last sub-expression evaluated. If it is followed by a semi-colon, the next sub-expression to be evaluated is that next to the right. In evaluating an explicit row-structured expression, such as <type symbol> $(\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n)$, the evaluation of ε_1 will be followed by the evaluation of ε_2, etc. In this way n values will be produced, as the n components of the row-structured expression.

Iteration

The construction

(1) **while** ε_1 **do** ε_2

where ε_2 contains no "zero-level" semi-colons, will be interpreted as follows: The symbols **while** and **do** act as left and right parentheses for ε_1, respectively. First, ε_1 is evaluated to produce a Boolean value. If **true**, ε_2 is evaluated, and then ε_1 is evaluated again, and so on. If ε_1 ever produces the value **false**, the evaluation is terminated and the successor of the **while** construction is determined as if the construction had been enclosed in parentheses. If ε_1 is a row-structured expression, say $(\varepsilon_{11}, \varepsilon_{12}, \ldots, \varepsilon_{1n})$, then the construction above is an

abbreviation for

$$(\underline{\text{while}}\ \varepsilon_{11}\ \underline{\text{do}}\ \varepsilon_2,\ \underline{\text{while}}\ \varepsilon_{12}\ \underline{\text{do}}\ \varepsilon_2, \ldots,\ \underline{\text{while}}\ \varepsilon_{1n}\ \underline{\text{do}}\ \varepsilon_2)$$

The construction

(2) $$\underline{\text{until}}\ \varepsilon_1\ \underline{\text{do}}\ \varepsilon_2$$

is equivalent to

$$\underline{\text{while}}\ \neg\varepsilon_1\ \underline{\text{do}}\ \varepsilon_2$$

The constructions

(3) $$\underline{\text{for}}\ n := \varepsilon_1\ \underline{\text{step}}\ \varepsilon_2 \left\{\begin{matrix}\underline{\text{until}}\\ \underline{\text{while}}\end{matrix}\right\} \varepsilon_3\ \underline{\text{do}}\ \varepsilon_4$$

where n is the name (possibly subscripted) of a variable, are equivalent to

$$n = \varepsilon_1; \left\{\begin{matrix}\underline{\text{until}}\\ \underline{\text{while}}\end{matrix}\right\} \varepsilon_3\ \underline{\text{do}}\ (\varepsilon_4;\ n := n + \varepsilon_2);\ n$$

Parts of these iterated constructions may be omitted, with the following default interpretations:

	Omitted	Interpretation
In (1) or (2):	$\left\{\begin{matrix}\underline{\text{while}}\\ \underline{\text{until}}\end{matrix}\right\} \varepsilon_1$	ε_2
	$\underline{\text{do}}\ \varepsilon_2$	$\left\{\begin{matrix}\underline{\text{while}}\\ \underline{\text{until}}\end{matrix}\right\} \varepsilon_1$ do;
In (3):	$n :=$	$\underline{\text{for}}\ k := \varepsilon_1\ \underline{\text{step}}\ \varepsilon_2 \left\{\begin{matrix}\underline{\text{until}}\\ \underline{\text{while}}\end{matrix}\right\} \varepsilon_3\ \underline{\text{do}}\ \varepsilon_4$ where k is an internal temporary variable.
	$:= \varepsilon_1$	$\underline{\text{for}}\ n := n\ \underline{\text{step}} \ldots$
	$\underline{\text{step}}\ \varepsilon_2$	$\underline{\text{for}}\ n := \varepsilon_1\ \underline{\text{step}}\ 0 \ldots$
	$\left\{\begin{matrix}\underline{\text{until}}\\ \underline{\text{while}}\end{matrix}\right\} \varepsilon_3$	$\ldots\ \underline{\text{step}}\ \varepsilon_2\ \underline{\text{until}}\ \underline{\text{false}}\ \underline{\text{do}} \ldots$
	$\underline{\text{for}}\ \ \underline{\text{do}}\ \varepsilon_4$	for $n := \varepsilon_1\ \underline{\text{step}}\ \varepsilon_2 \left\{\begin{matrix}\underline{\text{until}}\\ \underline{\text{while}}\end{matrix}\right\} \varepsilon_3$ do

Conditional

The construction

$$\underline{\text{if}}\ \varepsilon_1\ \underline{\text{then}}\ \varepsilon_2\ \underline{\text{else}}\ \varepsilon_3$$

where ε_1 is of type Boolean and ε_2 contains no "zero-level" conditional, is interpreted as if it had been written

$$[\underline{\text{first of}}\ (\varepsilon_1, \neg\varepsilon_1)]\ \underline{\text{of}}\ (\varepsilon_2, \varepsilon_3)$$

where first of is a unary operator which applies to Boolean-valued row-structured variables to produce the index of the first component (in order of listing from left to right) which is true.

Simultaneous Elaboration

The construction

$$\varepsilon_1\ \underline{\text{also}}\ \varepsilon_2\ \underline{\text{also}}\ \ldots\ \underline{\text{also}}\ \varepsilon_n$$

where ε_n is free of "zero-level" semi-colons, is interpreted as follows: The simultaneous evaluation of $\varepsilon_1, \ldots, \varepsilon_n$ is carried out. The value of the construction is that of ε_1, and the successor is that which would be the normal successor if the construction were contained in parentheses. Passage to the successor is made only after all of $\varepsilon_1, \ldots, \varepsilon_n$ have been evaluated. If the evaluation of more than one of the ε_i references the same variable, no interference will result, but no provision is made for knowing the order in which these references are made. We do postulate that:

1) Two accesses to the value of a variable can not occur simultaneously.
2) There is a procedure get-and-set (x,y) such that if the access to the value of x is by this procedure, then access to x other than from within the procedure is inhibited until the procedure terminates, returning the value of x and assigning the value of y to x.

The Prime Notation

In any expression various subexpressions may be "primed" (with one or more primes). An expression containing primed subexpressions is called a form. A variable of type form may be assigned form values or real values, and in the assignment process values are used for all unprimed sub-expressions, and simplifications are made when possible according to the indicated expression structure. No substitution of values is made for primed subexpressions, although unprimed components within them do receive values. An explicit row-structured form need not be preceded by the type symbol form. Removal of primes is accomplished by application of the eval operator. This has the explicit form:

a into f eval ε

where ε contains no "zero-level" semi-colons, and a and f are row-structured variables (with the same number of components) such that the values of a are forms, and the values of f are names of variables which are primed in ε. The action of eval in this case is: (i) Declare the elements of f to be new, (ii) assign them their correspondents in a, (iii) remove one prime from each sub-expression in ε, (iv) evaluate the resulting form.

Variables of Type Form

A variable whose type is set to form acquires automatically two additional attributes of type form and structure row : formal and actual. The default values of both attributes are the empty list Λ. The notation L : ε will be an abbreviation for (i) a declaration at the beginning of the program new L; type of L := form; and (ii) replacement of L : ε by L := ε'. At any time the formal and actual attributes of L may be assigned values as described for a and f in the preceding section. The notation Lf : ε, where f is a row-structured variable (or explicit row-structured expression), is an abbreviation for L : (formal of L := f;ε). If the expression ε doesn't need a name,

perhaps because it appears in a list of forms to be used as actual parameters, but it needs its own formal parameters, one may omit the name L and write f : ε. An occurrence of L in an expression is an abbreviation for:

actual of L into formal of L eval L

If one writes explicitly:

a into f eval L

then this is an abbreviation for:

actual of L := a; formal of L := f; L

If either a or into f is omitted here, the corresponding assignment is omitted. Finally, the notation La is an abbreviation for

a eval L

Correspondence with ALGOL facilities

It is instructive to digress here and map some of the ALGOL facilities into these constructions. The correspondents of the iteration and conditional constructions in ALGOL should be clear. What needs further explanation is the treatment of procedures, and the lack of blocks, the go to statement, and a switch facility. The point of view adopted here unifies all of these under the dynamic assignment of values (including forms) to names of variables and their attributes, and the dynamic determinations of scope; i.e., the duration of such assignments. The function of the block in ALGOL was to delimit the scope of certain declarations. The procedure definition delimited the scope of parameter substitutions. All of these are now treated as dynamic assignments, capable of redefinition, and each determining - dynamically - its own scope. Thus, a go to statement is no longer needed, since one simply names the code to be evaluated next, say L. This is a postponement of activity at the point at which the name L is invoked, and one is free to continue on when the postponement terminates (writing "L;"), or indicate that nothing further is intended here (writing "L.". On the

other hand, L is the name of an expression, so that it yields a value when it returns, and one can embed the "call" for L into larger expressions. The three concepts: (1) go to, (2) function call (or procedure call) with parameters, (3) function call (or procedure call) without parameters, now appear as one.

Own Variables

At times it is necessary to declare a variable during the evaluation of an expression whose existence does not end (i.e., it is not "popped") at the end of the current postponement. It is usually intended that its scope of definition coincide with that of some other variable declared new earlier. This is accomplished by declaring the variable just as with new, but with own instead, together with the name of the controlling variable (in brackets). Thus, one would have:

(new := (x); ... (own[x] := (y); ...) ...)

and both x and y are "popped" at the same time.

String variables

Variables of type string will be manipulated by the usual operators, such as concatenation, sub-string identification, extraction, and deletion, and so on. In addition, rules for matching and substitution similar to those found in Markov algorithms, COMIT, or SNOBOL will be used. Such rules will be written:

Apply (L_1 : (Left → Right; L_2) . L_3 : (...) .) to x

where Left and Right are strings, as in SNOBOL, for example, to be matched against the contents of the string variable x. First, the evaluation of L_1 is begun. If Left matches in x, Right is substituted as usual, and the procedure L_2 is called. Sequencing is subject to the rules given earlier relative to the semi-colon, parentheses, and period. If Left does not match, the "pop" corresponding to the postponement caused by L_1 is effected, and L_3 is called next. (The evaluation of L_3

in this situation and the initial evaluation of L_1 are the primary effects of the Apply operator.) The evaluation of the Apply construction terminates in a normal way when some rule has no successor. The successor to the Apply construction is determined as if x were standing alone.

Definitions

A facility will be provided by means of which types and structures not yet provided for in the language may be defined by the program. Contexts in the program involving defined constructions will be expanded into ordinary constructions at appropriate times prior to their evaluation as expressions. The prime notation for indicating that such code conversion must occur dynamically (and thus in an interpretive way), or statically, so that the defined construction may be entirely replaced by its expanded definition.

Representation of code as data

For some purposes, such as the application of editting operators, it is necessary that expressions normally intended to be evaluated be accessible as data. Thus, we may wish to select particular subexpressions, perform substitutions, and replace them. Their representation as evaluatable code may be quite different from that needed by the editting operators, however. We may postulate one pair of operators to convert "zero-level" semi-colons to commas and back, thus allowing selection of sub-expressions by subscript. A further conversion of the text into some kind of string form is probably necessary, also, depending on the kind of modifications one anticipates. For instance, replacement of identifiers or sub-expressions with others without disturbing the existing expression structure may be accomplished within the prime notation - eval facility, and so on. No specifications are given here for these conversions, but the need must be recognized.

Example 1:

```
quadratic (a,b,c):(new := (x1,x2);
    if a ≠ 0 then (d := sqrt(b x b - 4 x a x c);
        if d < 0 then Print ("no sol").
        else x1 := (- b - d)/(2 x a); x2 := (- b + d)/(2 x a);
            Print (x1,x2)).
    else if c ≠ 0 then Print ("one root", -b/c).
    else if b ≠ 0 then Print ("one root", 0).
    else Print ("no eq"))
```

Example 2:

```
quadratic (a,b,c):
    [first of (a,c,b,1) ≠ 0] of
        (if d := sqrt(b x b - 4 x a x c) < 0 then Print ("no sol")
            else Print ((- b - d)/(2 x a), (- b - d)/(2 x a)),
        Print ("one root", -b/c),
        Print ("one root", 0),
        Print ("no eq"))
```

Example 3:

```
quadratic (a,b,c):
    (if d := sqrt(b x b - 4 x a x c) < 0 then Print ("no sol")
        else Print ((- b - d)/(2 x a), (- b + d)/(2 x a)),
    Print ("one root", -b/c), Print ("one root", 0),
    Print ("no eq")) [first of (a,c,b,1) ≠ 0]
```

Example 4:

The problem: Many procedures exist for computing (a) $\sum_{k=0}^{N} C_k X^k$ as an approximation to (b) $\sum_{k=0}^{\infty} d_k X^k$, for some integral N. It is correct to say that N is a variable of use or call of the procedure. It is interesting to design a procedure which prepares (a) from (b).

Example 4.1.

```
f(N) : (own [f] := (array C[1 : N']); new := (integer k; form U);
        U(r) : <computation of C_r>;
        for k := 0 step 1 until N' do C[k] := U(k);
        f(X) : (new := (real g, integer k); g := C[N'];
        for k := N' - 1 step -1 until 0 do g := C[k] + g * X'))
```

Then, a first call f(n) followed by n calls of the form f(ε), where ε is an expression, provides values of $\sum_{k=0}^{n} C_k \varepsilon^k$.

Example 4.2.

```
f : ((N) : (own [f] := (array C[1 : N']); new := (integer k; form U);
        U(r) : <computation of C_r>; own [f] := M; M := N';
        for k := 0 step 1 until N' do C[k] := U(k)),
    (X) : (new := (real g, integer k); g := C[M];
        for k := M - 1 step -1 until 0 do g := C[k] + g * X'))
```

The calls would be [1] of f(n) or f[1](n) and then successive calls [2] of f(ε) or [2]f(ε).

Example 4.3.

Suppose an n is not known, but an $\varepsilon > 0$ is provided such that (a) is prepared for the least $N > 0$ such that $|C_N - C_{N+1}| < \epsilon$.

```
f(ϵ) : (new := (row C, real h,g, integer i); own [f] := (integer k;form U);
       U(r) : <computation of C_r>; C := g := U(0);
       for k := 1 step 1 until abs ((h := U(k)) - g) < ϵ') do
           append (C,g := h); own [f] := array d[0 : k];
       for i := 0 step 1 until k do
           (d[i] := contents of car (C); C := cdr(C));
   f(X) : (new := (g,integer i); g := d[k];
       for i := k - 1 step -1 until 0 do g := d[i] + g * X'))
```

Note 1: Append is a built-in procedure for adding a component to a row variable (which we may think of as represented by a list). The operator contents of is needed because it is assumed that in the list representation names are used as list components.

Note 2: The algorithm creates the array d to improve both storage and time performance of subsequent executions of f. This is possible because k has been computed and is assumed constant for future calls of f. Note that the row C disappears after the first call of f, and its storage is thereby released. One could, of course, dispense with d and use C directly by:

1) Rewriting the initial declarations as:
   new := (h,g, integer k); own [f] := (row C);

2) Using prefix in place of append to produce the list in reverse order.

3) Writing the last assignment to f as

```
f(X) : (new := (g,h); g := contents of car (C); h := C;
       until h := cdr(h) = undefined do
           g := contents of car(h) + g * X')
```