A Proposal for Definitions in ALGOL

by

B.A. Galler[1)]

University of Michigan and Mathematisch Centrum, Amsterdam

and

A.J. Perlis[2)]

Carnegie Institute of Technology and Mathematisch Centrum, Amsterdam

1. Introduction.

It has long been clear that no matter what notational devices and conveniences are defined as primitives of a language, its users will inevitably want others which are equally fundamental for their particular needs. This can not be taken as a criticism of the language designers, since it is impossible to forsee and provide in one language every notational device anyone could ever want. These inevitable omissions can be mitigated by providing a basic set on which there is general agreement (such as the arithmetic and relational operators, and so on), and by providing also a framework within which new devices can be defined into the language by a user for his particular needs.

A complete definition facility requires the capability to dynamically define the syntax of the language, as well as the accompanying semantics (as embodied in the processor or interpreter). Such a system, which provides a mechanism for changing the syntax and semantics, has been in use at Carnegie Institute of Technology since 1964, under the name Formal Semantics Language [1].

1) The work presented here was supported in part by the National Science Foundation (GP - 4538) and the Air Force Office of Scientific Research (AF - AFOSR - 1017-66).

2) The work presented here was supported in part by the Advanced Research Project Agency of the Office of the Secretary of Defense (SD - 146).

A more limited form of definition facility (but still extremely flexible and powerful in use) has been available for several years in the MAD [2] language at the University of Michigan. In the latter, for example, definition packages are now available to introduce into the language vector and matrix arithmetic, complex number arithmetic, and multiple precision arithmetic. The importance of the facility is not that these specific packages are now available, but that any user is free to develop his own within a broad framework that creates no difficulty in their implementation in the MAD processor and is simple to use.

The definition facility described in the following is similar in many ways to that which is available in MAD, and is intended as a proposal to extend ALGOL 60. It is not as complete a proposal as one might make, since the underlying syntax of the language may not be arbitrarily modified.

A strong feature of this proposal is that each new operator, or each new interpretation for an existing operator, is invoked via an implicit macro call; in fact, by the insertion of in-line code. The advantages here over explicit calls on procedure (which may already be written in ALGOL), lie in the notational convenience of composing operations into expressions using infix notation, i.e. with operators appearing between their operands instead of as nested procedure calls, together with a simple means of producing efficient ALGOL code.

The basic proposal presents means by which new operators and/or new data types may be introduced within programs. For each new operator one must show how to interpret its action on new or old data types. For each new data type one must show how it behaves under the action of new or old operators.

Moreover, it often is necessary to determine dynamically whether a variable has been assigned a given type. We therefore postulate the generation for each declared type q of a Boolean - valued procedure q(x) which is true if x has the type q, otherwise false. Examples would be real(x), Boolean(x), etc. Such procedures would presumably need dynamic access to a generalized symbol table carrying type information, among other things.

In order to introduce a new operator into the official ALGOL syntax specifications, it would be necessary to (1) specify how it would appear

among other syntatic units, and (2) relate it to existing operators by syntax forms which in effect establish its "precedence" in the extended set of operators. A short study of the treatment of "addition operator" and "multiplying operator" in the ALGOL 60 Report [3] would suffice to show how this specification process is achieved. We shall summarize the insertion of a new operator's syntax form by the declaration:

$$\left\{\begin{matrix}\underline{\text{unary}}\\ \\ \underline{\text{binary}}\end{matrix}\right\}\ \underline{\text{name}},\ \underline{\text{precedence}}\ \left\{\begin{matrix}\underline{\text{above}}\\ \underline{\text{same as}}\\ \underline{\text{below}}\end{matrix}\right\}\ \underline{\text{op}}$$

where one choice is made in each set of braces, and name is the symbol introduced for the new operator. The symbol op is the name of any operator already available (whether originally in ALGOL or already declared in this way), such as +, v, ÷ or ↑. The interpretation of above and below is exactly that obtained by carrying out the syntax modification suggested above. In other words, name is inserted into the syntax "just above" op, and below any other operators already above op. For example, a typical operator declaration such as

binary cons , precedence above ×

modifies the syntax of arithmetic expressions so that

<term>:: = <factor>|<term><mult.operator><factor>

is replaced by:

<term1>:: = <factor>|<term1>cons<factor>

<term>:: = <term1>|<term><mult.operator><term1>.

Similar replacement rules hold for unary operators.

In order to describe the behavior of any operator on any particular data type (and thus include all of the cases mentioned above), we introduce the context definitions, which specify those and only those contexts in which an operator or data type may legally occur, and the interpretations to be applied when these contexts are encountered.

A typical context definition is:

list a sum list b := real sum(a,b)

which says that whenever sum occurs with two operands of type list, the code used is a call for a real procedure named sum, with the actual operands of type list becoming the actual parameters in the procedure call. (The

typed identifier expression appearing as the left side of a context definition is known as a type context.) As will be seen in the detailed syntax and semantics description given below for the proposed definition facility, this procedure call form is merely a special case of the explicit array form stipulated after the := in the syntax of the context definition. The explicit array form will normally contain a block expression involving the variables mentioned in the type context and local variables declared in the block containing the declaration, and it will usually produce a value for a result variable r.[1)]
An explicit array form becomes a block expression when its formal parameters (those specified in the type context) are replaced by the actual expressions through which this context has been recognized as applicable. It is clear that the context declaration not only defines the meaning of the context; it also acts as a restriction on the allowable combinations of operators and data types. Advantage can be taken of this to produce desired code constructs from a set of possibilities.

Another structure in the language which needs an appropriate notation is the enumerated array, representing explicitly an instance of a new data type. Since it is intended that existing ALGOL shall be a subset of what is proposed here, existing forms of expressions are acceptable without change. However, a new form of expression may also be written; viz., a list of expressions preceded by a type symbol. An example would be complex (3, x + z) representing the complex number usually written as 3 + (x + z)i. The representation as a sequence of components assumes a particular internal representation, i.e., as a real array. Multi-dimensional arrays are linearized by varying the last subscript first.

In this definition facility we assume that data structures can be allocated storage capable of description in terms of ALGOL arrays.

---

1) As ably demonstrated in CPL [5], block expressions are so useful that it is assumed that they have already been made available in ALGOL.

The declaration of this storage allocation is called the primitive representation below. Thus, a list is assumed, in one of the examples below, to be stored as a one-dimensional ALGOL array with two elements: the name of the first list item (stored as the first element of the array), and the name of the rest of the list (considered also a list), stored as the second element. We also assume that primitive procedures called name of and contents of are available which produce real numbers representing the name (or machine location) of the value of a variable, or vice versa.

The code created from an occurrence of a context involving new data types will often produce a result which has again one of the new data types. In particular, it may very well require an array for its storage, and the code needed to produce all of the elements of this array must be provided. In particular, when a block expression is executed whose result has a new data type which requires an array for its storage, it is necessary that this storage be automatically and dynamically supplied.

In a larger context, however, the separate parts of the computation for the general element of a result array can often be combined inside the outermost iteration structures, thus eliminating much unnecessary red-tape and intermediate storage computation. Thus, the code for summing three n x n arrays (d = a + b + c):

```
begin integer i,j; for i:=1 step 1 until n do
                       for j:=1 step 1 until n do
                           d[i,j] := a[i,j] + b[i,j] + c[i,j]
                       end;
```

is much preferred over the code produced by treating the sum as two consecutive binary sums, as shown below. The much better code given first results from the replacement rule given below, provided the explicit array form in the context definition is written appropriately. (The matrix arithmetic package exhibited below does produce the first form.)

```
begin integer i,j;
      for i:=1 step 1 until n do
            for j:=1 step 1 until n do
                  d[i,j] := a[i,j] + b[i,j];
      for i:=1 step 1 until n do
            for j:=1 step 1 until n do
                  d[i,j] := d[i,j] + c[i,j] end;
```

In order to be sure that the best code is produced in a given situation, it may be necessary to assign more than one type to a variable. Thus, for a repeated matrix product $A \times B \times C$, we shall find it useful in the examples below to declare B and C to have type row of columns (r/s), while A is declared to be of type column of rows (c/s). Since $B \times A \times C$ could occur in the same program, we need to be able to declare B of type c/s and A of type r/s, also. In other words, a variable may have several types, and it will have to follow from the context in which the variable occurs which is intended. Should more than one context definition apply, the first one is used which results in a data type which then enters into legitimate larger contexts, etc. Should a choice of context definition lead eventually to an undefined context, an earlier choice will have to be changed. In general, the largest context applicable is used whenever possible.

To summarize what is being proposed here as extensions to ALGOL 60, we have new data types (and procedures which test for these types), and declarations introducing new operators. Explicit instances of these data types may be written, but more often variables which have been assigned these types will be involved in expressions. In either case, the storage requirements of new data types are explicitly declared in terms of ALGOL arrays (by the primitive representation). By providing type contexts with corresponding explicit array forms, instances of contexts involving new operators and data types may be replaced by "open code" generated from the explicit array forms under the replace-

ment rule given below. This rule is such as to eliminate excessive iteration control computation whenever possible.

We also include name of and contents of as primitives, since they are useful in generating list structures as data types. It was observed that results of expressions may be assigned new data types, and thus require temporary storage in the form of arrays. This use of temporary storage is considerably reduced, however, by the elemination of excessive iterations [4]. Multiple type assignments for variables are introduced to allow the best context recognition strategy. All of these concepts will appear in the examples below.

Certain features, which are an important part of existing macro systems, are missing from this treatment, e.g., conditional expansions and "expansion time" variables, with the implications to scope that they provide. This treatment, here limited to expressions, could with the aid of the above features, be expanded to provide a facility capable of handling contexts at the statement level, or even at the program level.

2. Syntax

The syntax additions for the proposed extension to ALGOL 60 follow. In each case of conflict with the ALGOL 60 report [3] (indicated by a bold-face period (.) in the left margin), the form given here is to be used. Forms already defined in the ALGOL 60 report are not repeated here.

<boldface character>::= a|b|...|z|1|...|9|0|+|...|↑|≤|...|≠|
≡|...|¬|,|...|⊗|(|...|'|;

<boldface symbol>::=<boldface character>|<boldface symbol>
<boldface character>

<new operator>::=<boldface symbol>

<new type>::=<boldface symbol>|<new type><typed identifier list>

- <operator>::=<arithmetic operator>|<relational operator>|
  <logical operator>|<sequential operator>|
  <new operator>
- <type>::= real|integer|Boolean|<new type>

<type declaration list>::=<type declaration>|<type declaration list>,
<type declaration>

<open set>::=<type>|<operator>|<open set>,<type>|<open set>,
<operator>

<set>::= (<open set>)

<set name>::=<boldface symbol>

<set declaration>::=<set name>:=<set>

<primitive representation>::=<type declaration list>means
<array declaration>

<typed identifier>::=<type><identifier>

<typed identifier list>::=<typed identifier>|<typed identifier list>,
<typed identifier>

<type context>::=<any arithmetic, Boolean or designational expression built out of typed identifiers and set names in the sane way that they are normally built out of identifiers.>[2),3)]

<type context list>::=<type context>|<type context list>,
<type context>

---

2) This is a short description of what could be (and should be considered to be) a formal syntatic statement.

3) In this treatment := is taken to be a binary operator whose precedence is below all arithmetic or Boolean operators in ALGOL. The semantics of := as an operator are obvious.

<expression list> ::=<expression>|<expression list>,<expression>

<enumerated array>::=<type><expression>|<type>(<expression list>)

<generated array>::=<type><expression>

<explicit array>::=<enumerated array>|<generated array>

<primary>::=<unsigned number>|<variable>|<function designator>
|<explicit array>|(<arithmetic expression>)

<explicit array form>::=<generated array>|<enumerated array>

<context definition>::=<type context list>::=<explicit array form>

<adjective>::= unary|binary

<precedence phrase>::= above|same as|below

<operator definition>::=<adjective><new operator>, precedence
<precedence phrase><operator>

3. Semantics

(1) A context, set, or operator definition is local to the block containing it and must precede its use.

(2) Operators become defined in the lexicographic order in which their definitions are written.

(3) An operator definition must involve only an operator for which a definition has already been encountered.

(4) The expression on the right side of a context definition must involve only operators which have already appeared in operator definitions.

(5) An explicit array form becomes an expression only after certain replacements have been made for some of the variables which appear in it. This is explained as part of the replacement rule given below.

(6) A context definition is interpreted as if it had been a sequence of separate definitions, each one containing one item from the type context list on the left, and always the explicit array form on the right.

(7) The array declarations after the means in a primitive representation may not involve new types. This declaration is required

whenever an instance of a new data type requires an array for its storage, and it applies to each of the types involved in the declaration.

(8) The primitive representation implies a storage allocation to be invoked whenever one of the types so declared occurs in a procedure or block heading.

(9) Whenever an explicit array needs to have storage set aside for it, such as for the result of a block expression or for an enumerated array, it is assumed to have the array structure declared for its type in a primitive representation. Multi-dimensional arrays will be linearized to accept enumerated arrays by varying the final subscript first.

(10) In a block expression the value produced by the block is (by convention) the value of the variable r on exit from the block. This variable is always local to the smallest block containing it.

(11) In any block expression which appears in an explicit array form there must be exactly one occurrence of the result r, possibly subscripted, and that occurrence must be on the left side of an assignment statement. A block expression nested within this block expression will have its own single occurrence of its own r.

(12) In type declarations and in type contexts one may use a new type with or without a bracketed list adjoined. (A bracketed list may not contain bracketed lists.) If a match is to be made with a type context which also contains a list, then the entries in the lists must agree in type and number.

If the match is made (according to the replacement rule below), then the list entries declared for the variable in question are substituted for their corresponding entries in the explicit array form. If the type context contains the new type without a list, a match may be made without reference to the declared list of constants, and no substitutions result from their presence.

(13) A set may appear in only one set declaration.

(14) A type context containing one or more set names is equivalent to the type context list obtained by substituting all combinations of

representatives from those sets for the set names.

(15) All primitive representations, set, operator, and context definitions are part of the block heading.

4. The Replacement Rule

The rule for the replacement of defined constructions by ALGOL code is as follows: We assign a type to each expression on the basis of the types of its sub-expressions. Identifiers and constants are assigned types on the basis of declaration and form. Wherever there is a choice of types, the type is chosen by the following rule:

(1) Defined contexts have priority over ALGOL syntax.

(2) Among the defined contexts, priority is assigned in order of listing.

When using a context definition, assuming that the sub-expression matches the type context in form, the type assigned to the sub-expression is that which appears to the right of the := in the context definition.

Either of two possibilities may now occur: (1) all expressions are assigned types, or (2) an expression has been encountered for which no type can be assigned. In case (2), either (2.1) this expression has the form E[<subscript list>], where E has the form and type occurring in a context definition, or (2.2) it doesn't. In the latter case (2.2), an alternate choice of types must be made where such a choice was possible, and the entire process must be repeated. If no further choice can be made, the original text was not syntactically correct. In the former case (2.1), consider the explicit array form in the context definition associated with E. Either (2.1.1) it contains a single pair of boldface parentheses, or (2.1.2) it does not. In the latter case (2.1.2) it must be an enumerated array, and the subscript list must consist of a single integer constant i_0. Then select the i_0-th element of the enumerated array. In this expression form substitute the corresponding sub-expression from E. The resulting ex-

pression is then substituted into the text for $E[i_0]$, and the type assignment is performed on the new text.

If the explicit array form does contain a pair of boldface parentheses (case 2.1.1), select the text bounded by these parentheses. This text must contain a single occurrence of r (the result of the block expression containing this text), and this occurrence must be of the form r[<subscript list>]:=F, where F is an expression. These formal subscripts must agree in number and type with those occurring with E, and then the formal subscripts are replaced in the text by the correponding actual subscripts. From this resulting expression form an expression E' is obtained by substitution of the corresponding sub-expressions from E. Then the "arithmetic context is moved inside," as follows: Substitute F for E[<subscript list>] in the maximal arithmetic expression G containing E, to obtain G'. Substitute G' for r[<subscript list>]:=F in E', obtaining E". (Note that if G follows a then or else in a conditional expression, we may extend G even further by considering the arithmetic context of the conditional expression to be the arithmetic context of each of the arithmetic expressions within the conditional expression.) We now replace G in the original text by E", and the type assignment process is performed on the new text.

We now consider the case (1) in which all expressions have been assigned types. Either (1.1) there are still defined constructions, or (1.2) there are not In case (1.1), for each maximal expression E whose type and form correspond to a context definition: (i) form an expression E' by substituting corresponding sub-expressions from E into the explicit array form associated with E, and (ii) substitute E' for E in the text. In this substitution, if E' consists of a block expression, move the arithmetic context of E inside, just as was done in case (2.1.1). The replacement rule is applied to the resulting text.

If no defined constructions exist (case 1.1), then all boldface parentheses (together with any immediately preceding type symbols that may accompany them) are deleted, and declarations of new types

are dropped after replacing them, when necessary, with appropriate array declarations obtained from the primitive representations. The resulting text is either a syntactically correct ALGOL 60 program, or the original program was syntactically incorrect.

The examples which follow were chosen to illustrate the power and versatility proposed here. Each "package" shows the strategy employed in choosing appropriate definitions.

5. Examples

A. Matrix Arithmetic Package

(1) r/s a, c/s a means array a[1:n, 1:n];

(2) col a means array a[1:n];

(3) row a means array a[1:n];

(4) unary T, precedence above ↑;

(5) unary I, precedence above ↓;[4]

(6) c/s a:= c/s b i → n do row (r[i]:= a[i]) e;[5),6)]

(7) r/s a:= r/s b j → n do col (r[j]:= a[j]) e;

(8) row a:= row b j → n do real (r[j]:= a[j]) e;

(9) col a:= col b i → n do real (r[i]:= T a[i]) e;

(10) c/s a[integer k][integer l]:= real a[k,l];

---

4) ↓ is the symbol used here for subscription.

5) i → n will be used here as an abbreviation for: integer i; for i:=1 step 1 until n

6) b and e are begin and end, resp.

```
(11) T real a[integer k][integer l] := real a[l,k];
(12) T c/s a:= r/s T a;
(13) I c/s a:= c/s inv(a);
(14) I I c/s a:= c/s a;
(15) c/s a + c/s b := c/s b i → n do row (r[i]:= a[i] + b[i]) e;
(16) r/s a + r/s b := r/s b j → n do col (r[j]:= a[j] + b[j]) e;
(17) c/s a + c/s b := r/s (T a + T b);
(18) row a + row b := row b j → n do real (r[j]:= a[j] + b[j]) e;
(19) c/s a × r/s b := c/s b i → n do row (r[i]:= b row t;
                    t := a[i]; r := t × b e) e;
(20) row a × r/s b := row b j → n do real (r[j]:= a × b[j]) e;
(21) row a × col b := real b integer j; real t; t:=0; for j:=1 step
        1 until n do t:=t + a[j]× T b[j]; r := t e;
(22) c/s a := c/s b := c/s b i → n do row (r[i]:= a[i]:= b[i]) e;
(23) row a := row b := row b j → n do real (r[j]:= a[j]:= b[j]) e;
(24) col a := col b := col (T a := T b);
(25) c/s a × r/s b + c/s a × r/s c := c/s a × (b+c);
```

Statements 1 to 3 give the primitive representations for the new data structures being created, in terms of basic ALGOL arrays. Statements 4 and 5 introduce the two new operators T (the transpose) and I (the inverse, implement by a procedure named inv). Statements 6 through 9 show how each level of structure is defined in terms of the next lower level. Statement 10 provides the accessing function needed to actually reach a value. (Here it is a trivial mapping, but in the file maintenance package below, for example, it is more complicated.) Statements 11 to 14 define the behavior of the two new operators. Note that the cancellation of pairs of applications I or T will be

effected during the reduction to ALGOL code, and thus lead to no unnecessary code. Statements 15 to 21 define the action of + and × on the various combinations of data types that may occur, while statements 22 to 24 define the "store" operation for the new data structures. Statement 25 takes advantage of an identity appropriate to matrix arithmetic; e.g. the distributive law, to effect a simplification before any other substitutions can occur, thus leading to better code at the highest level. In view of the action of the replacement rule, statement 25 will automatically extend to more than two terms, so that

$$a \times b + a \times c + a \times d + a \times e$$

will first be transformed to

$$a \times (b + c + d + e)$$

with a dramatic reduction in the amount of code subsequently generated.

This set of context definitions does not provide correct ALGOL code for expressions where an array appears on both side of the := and on the right side not only as the left most factor in a term. Additional definitions can be added, at the expense of efficiency, should this case have to be handled.

Example of use:

```
                real r/s c/s x,y,z,w; w := (x+y) × z;
(22)            b i → n do w[i]:= ((x+y) × z)[i]e;
(19)            b i → n do b row t; t:= (x+y)[i]; w[i]:= t × z e e;
(6,15,23)       b i → n do b row t; t:= x[i] + y[i]; b j → n do w[i][j]:=
                                        = (t × z)[j]e e e;
(6,7,8,10,      b i → n do b row t; b j → n do t[j]:= (x[i] + y[i])[j]e;
  20,23)                  b j → n do w[i,j]:= t  ×  z[j]e e e;
(6,7,18,21)     b i → n do b row t; b j → n do t[j]:= x[i][j] + y[i][j]e;
                  b j → n do b integer k, real s;s:=0, for k:=1 step
                                                              1 until n
                  do s:= s+t[j]× T z[j][k]; w[i,j]:= s e e e e;
```

```
(6,7,8,9,      b i → n do b row t; b j → n do t[j] := x[i,j] + y[i,j] e;
   10,11)      b j → n do b integer k, real s; s:=0, for k:=1 step
                                                    1 until n do
               s :=s + t[j] × z[k,j];w[i,j] := s e e e e;
(1,3)          real array x,y,z,w[1:n, 1:n];
               b i → n do b array t[1:n]; b j → n do t[j] := x[i,j] +
                                                    + y[i,j] e;
               b j → n do integer k; real s; s:=0; for k:=1 step 1
                                                    until n do
               s :=s + t[j] × z[k,j]; w[i,j] := s e e e e;
```

In this example the numbers to the left of each line show the definition declarations which were invoked to obtain that line. The first statement of the example, containing the type declarations, was suppressed during the intermediate steps of the expansion. The use of k and s (instead of j and t) was introduced for clarity; the block structure of ALGOL would make this unnecessary in practice. Note that in the application of (22) and (23), the r[i] := was deleted. This is in accord with the natural interpretation of the use of block expressions.

If one wished to take advantage of special properties of arrays, e.g. triangularity, symmetry, or to organize their store by using a column vector of row names to speed up accessing (hereafter denoted as a column-array), then new contexts must be appropriately inserted so as to cause their application before those of the original set, such as:

```
1.1:   column-array a means array a[1:n];
9.9:   c/s [column-array a]  a1 [integer i][integer j] :=
           real contents of (a[i] + j).
14.1:  column-array [c/s a1] a := c/s  a1;
```

In any use an actual declaration would be like

c/s [X] Y; column-array [Y] X;

One may also incorporate into the original set of definitions an explicit dependence on the row length n by using bracketed type lists. Then modified storage mappings could be introduced to accomodate symmetric and triangular arrays by making them depend in the same way on the row index i. In these cases one would also introduce alternate access functions just before statement 10, so that the new storage mappings could be effected. Details are left for the reader.

B. File Maintenance Package

```
B := (∧,v); A := (+, −, x, /,÷, :=);
R := (=, ≠, <, ≤, >, ≥); rat := (real, attr);

(1)   TV a, file a means array a[1:m];
(2)   rec a means array a[1:n];
(3)   file a := file b i → m do rec (r[i] := a[i]) e;
(4)   rec a := rec b j → n do real (r[j] := a[j]) e;
(5)   TV a := TV b i → m do Boolean (r[i] := a[i]) e;
(6)   file a[integer i][integer j] := real contents of (a[i] + j-1);
(7)   unary count of, precedence below v;
(8)   binary on, precedence above count of;
(9)   count of TV a := real b real c; integer i; c :=0;
        for i:=1 step 1 until m do
        c := c + if a[i] then 1 else 0; r := c e;
(10)  Boolean a on file x := TV b i → m do Boolean (r[i]:= a on x[i])e;
(11a) Boolean a B Boolean b on rec c := Boolean a on c B b on c;
```

```
(11b)  ¬ Boolean b on rec c := Boolean b on c;

(12)   rat a R rat b on rec c := Boolean a on c R b on c;

(13)   attr [integer j] a on rec c := real c[j];

(14)   real a on rec c := real a;

(15)   real a on file x, attr a on file x := file b i → m do rec
                                              (r[i] := a on x[i])e;

(16)   rat a A rat b on rec c := rec a on c A b on c;

(17a)  if Boolean a then real b on file x := file b i → m do rec
          (r[i] := if a on x[i] then b on x[i]) e;

(17b)  if Boolean a then real b else real c on file x:=file b i → m
          do rec(r[i] := if a on x[i]then b on x[i] else c on x[i])e;

(18)   rat a R rat b := Boolean a R b;

(19)   rat a A rat b := real a A b;

(20)   file a:= file b := file b i → m do rec (r[i] := a[i] := b[i])e;

(21)   rec a := rec b := rec b j → m do real (r[j] := a[j] := b[j])e;

(22)   TV a := TV b := TV b i → m do Boolean (r[i] := a[i] := b[i])e;
```

Statements 1 and 2 define the new data types that need storage allocation. Statements 3 , 4 and 5 show how each type is constructed out of the others. Statement (6) gives the explicit access for these data types. Statements 7 and 8 define two new operators, while 9 through 17b define their behavior. Statements 18 and 19 are used to assign types to subexpressions while looking for a match. Statements 20 through 22 define the behavior of := with the new data types.

Example of use:

```
file a; attr [1] name, attr [2] sex; attr [3] height;
real cnt; cnt:= count of sex = 1 ∧ height ≥ 6 on x;
```

Expansion of assignment statement:

(9,10,17) b real c; integer i; c:=0; for i:=1 step 1 until m do

c := c + if

(sex = 1 ∧ height ≥ 6 on x)[i] then 1 else 0; cnt:= c e;

Using only the Boolean expression after if:

(10) sex = 1 ∧ height ≥ 6 on x[i]

(11a) (sex =1) on x[i] ∧ (height ≥ 6) on x[i]

(12) sex on x[i]= 1 on x[i] ∧ height on x[i] ≥ 6 on x[i]

(13,14) x[i][2] = 1 ∧ x[i][3] ≥ 6

(6) contents of (x[i] + 1) = 1 ∧ contents of (x[i] + 2) ≥ 6

The full expansion is than:

array x[1:m]; real cnt; b real c; integer i; c:=0;

for i:=1 step 1 until m do c:= c + if contents of

(x[i] + 1)=1 ∧ contents of (x[i] + 2) ≥ 6 then 1 else 0;

cnt := c e;

Example of use:

file y; y:= if height ≥ 6 then 1 else 0 on x;

Expansion of assignment statement:

(17b,20) b i → m do y[i]:=(if height ≥ 6 then 1 else 0 on x)[i] e;

Using only the right side of the assignment statement:

(17b,18) if height ≥ 6 on x[i] then 1 on x[i] else 0 on x[i]

(12,13,14) if x[i][3] ≥ 6 then 1 else 0

(6) if contents of (x[i] + 2) ≥ 6 then 1 else 0

The full expansion is:

array y[1:m]; b i → m do y[i]:= if contents of (x[i] + 2)≥ 6

then 1 else 0 e;

C. Complex Arithmetic Package

```
(0)  A := (+, -);
(1)  complex a means array a [1:2];
(2)  binary i, precedence above ×;
(3)  unary realpart, precedence below ↓;
(4)  unary imagpart, precedence same as realpart;
(5)  unary mag, precedence same as realpart;
(6)  unary arg, precedence same as realpart;
(7)  unary conj, precedence same as realpart;
(8)  realpart complex a:= real a[1];
(9)  imagpart complex a:= real a[2];
(10) real a i real b:= complex complex(a,b);
(11) complex a A complex b := complex complex(a[1] A b[1],
                                              a[2] A b[2]);
(12) complex a × complex b := complex complex(a[1] × b[1]-
          a[2] × b[2],a[2] × b[1] + a[1] × b[2]);
(13) complex a / complex b := complex
          (a × conj b)/(b[1] ↑ 2 + b[2] ↑ 2);
(14) real a × complex b := complex complex(a × b[1], a × b[2]);
(15) complex b × real a:= complex a × b;
(16) mag complex a:= real sqrt(a[1] ↑ 2 + a[2] ↑ 2);
(17) arg complex a:= real arctan(a[2]/a[1]);
(18) conj complex a:= complex complex (a[1],-a[2]);
(19) complex a= complex b:= Boolean a[1]= b[1] ∧ a[2]= b[2];
```

D. List Package (based on the LISP [6] primitives).

(1) list x means array x [1:2];

(2) binary cons, precedence same as x;

(3) unary car, precedence above cons;

(4) unary cdr, precedence same as car;

(5) real a cons real b := real name of list(a,b);

(6) car real a := real a[1];

(7) cdr real a := real a[2];

Then: car(a cons b) = car name of list(a,b) = a

and cdr(a cons b) = cdr name of list(a,b) = b

and car(a) cons cdr(a) = real name of list(a[1],a[2]) = b,

where equal(b,a) is true (in the LISP sense).

LISP Composition:

(8) op f ⊕ op g := op f cons g;[7]

(9) op f ⊕ real x := real procedure E(f,x); op f; real x;

E := if atom(x) then eval(f,x) else

E(f,car x) cons E(f,cdr x);

---

7) f⊕x means f(x), and this operator is now being extended to new contexts.

The procedure eval(f,x) is defined as follows:

```
real procedure eval(f,x); op f; real x;
if atom(f) then f(x) else eval(cdr f, (car f)(x));
```

Context definitions (8) and (9) provide an efficient rule for sequencing through a composition of operations on lists (such as f and g of type op), each one of which operates only on atoms to produce atoms or even lists. The sequencing is organized in this case (others are of course possible using different constructions) so that as each atom is encountered the remaining operators in the composition are applied to it. Thus the lists are not totally decomposed and composed for each successive operator.

Example of use:

```
begin atom g,h; op F,G;
real procedure subst(x,y,z); real x,z; atom y; subst :=
    if atom(z) then real b r := if eq(z,y) then x else z e
    else subst(x,y,car z) cons subst(x,y,cdr z);
real procedure F(x); real x; F:= subst(a,g,x);
real procedure G(x); real x; G:= subst(b,h,x);
c:= G ⊕ F ⊕ b end;
```

References

1. J.A. Feldman, A Formal Semantics for Programming Languages, Proc. I.F.I.P. (1965).

2. B.W. Arden, B.A. Galler, and R.M. Graham, "Michigan Algorithm Decoder", University of Michigan Press, Ann Arbor, Michigan, 1963.

3. P. Naur ed. "Revised Report on the Algorithmic Language ALGOL 60", Num. Math. 4(1963), p. 420-453.

4. B.A. Galler and A.J. Perlis, "Compiling Matrix Operations", Comm. A.C.M., vol. 5(1962), p. 590-594.

5. D.W. Barron, J.N. Buxton, D.F. Hartley, E. Nixon, and C. Strackey "The main features of C.P.L.", The Comp. Journ. 6(1963), p. 134-142.

6. J. McCarthy, et al "LISP 1.5 Programmers Manual", MIT, Cambridge, Mass. 1962.

Errata for "A Proposal for Definitions in ALGOL"
by B.A. Galler and A.J. Perlis

Title page Insert as l.-1 March 29, 1966

p. 6 l.11 (<arithmetic expression>)) | <e.e.>

p. 7 l. 1 (5) The revision ...

l.-10 ... [i,j] represents $(A^k)_{ij}$.

l.-3 such <procedure>s ...

p. 9 l. 3 ... <actual parameter> | <a.t.s.p.>

Insert after l.6 <actual type set parameter (a.t.s.p.)> ::= <a.t.> | <c.t.s.n.>

l.10 ... <a.t.> | <a.t.s.p.>array ...

l.16 <operators(o.)> ::= ...

l.17 ... | <relational operator> |, |↓|⊕ [1]

Move footnote from page 15 to page 9.

p. 10 l. 9 ... <c.t.i.> | <a.t.s.p.> [program] ...

l.16 ... <r.t.><string> | <c.> := <r.t.> |

p. 11 l.-2 ... ignore it since <delimiter> has no ...

p. 14 l. 2 ... <context>s. [1] ...

l.-2 ... every <context definition> which has a <string> and <result

p. 15 l. 1 (↓):

Delete footnote here after moving it to p. 9.

p. 16 l. 6 ... units let L_x be the ... legal strings in

l. 7 ALGOL x. To each D there ... analyzer A_D which has ...

l. 8 ... that : if $\rho \varepsilon L_D$ then $A_D(\rho) = t(\rho)$, where $t(\rho) \varepsilon L_C$, or $t(\rho)$

l.14 ... necessity ... when A_D

l.17 ... The analyzer A_D, for one of <arithmetic ...

p. 21 l. 7 s := s + P[j,k] × Q[i,k];

l. 8 K[i,j] := c × s e e;

p. 21 l.-4,...,-1 This program requires only 2n+2 locations for temporary storage, but it takes more time. Should one wish to produce the faster code, definitions could be written to generate the full temporary storage required. The key definitions would reflect a "bottom-up" syntax analysis. For example:

```
matrix(u,v)a × matrix(v,w)b := matrix(u,w) 'matrix(u,w)
b array P[1:u,1:v], Q[1:v,1:w]; P := a; Q := b; b integer i,j,k;
real s ;   i → u do j → w do b s := 0; k → v do s :=
s + P[i,k] × Q[k,j]; r[i,j] := s e e e';
```

Some of the transitional states of the tree for the second expansion were as follows (numbers in parentheses refer to definitions invoked):

l.-2 ... follows (numbers in ...

p. 25 l.14 ... real'real b real c; ...

l.15 c := c + if a[i] ...

p. 27 l.18 (10) op (F)f of list x ...

l.19 (11) op (F)f of op (G)g := ...

l.20 (12) list y of op (F)f := ...

l.-4 such as op (H)h, the ...

p. 28 l. 3 else E(car f, (pic of ...

p. 29 l.-9 ... 'complex (a × b[1], a × b[2])';

p. 31 l.-5 tion 5. Then a <context definition> and <declaration> are:

l.-4 complexmatrix (b,c)a := complexmatrix 'b,c';

p. 33 l.-1 matrix (complex, m, m) means array [1:m, 1:m, 1:2];

p. 37 l.-1 ... naming techniques, as in COMIT and SNOBOL, for example.

p. 38 Insert after l.11 (ii) Create another <context definition> using the <result type> γ of Q:

γ[program] [[bound pair list]] := γ

l.12 Now for each <context definition> use ...

l.14 (iii) If there is already ...

l.16 (iv) Represent P ...

p. 39 l.-5 Successive replacements ...