A Proposal for Definitions in ALGOL

by

36

B.A. Galler and A.J. Perlis

Abstract

An extension to ALGOL is proposed for adding new data types and operators to the language. Definitions may occur in any block heading and terminate with the block. They are an integral part of the program and are not fixed in the language. Even the behavior of existing operators may be redefined. The processing of text containing defined contexts features a "replacement rule" that eliminates unnecessary iterations and temporary storage. Examples of definition sets are given for real and complex matrices, complex numbers, file processing, and list manipulation.

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 ...

A Proposal for Definitions in ALGOL

by

B.A. Galler *)

University of Michigan and Mathematisch Centrum, Amsterdam

and

A.J. Perlis **)

Carnegie Institute of Technology and Mathematisch Centrum, Amsterdam

1. Introduction

This paper describes a technique for programming definitions within an ALGOL-like programming language. Though the approach is applicable to most languages which possess compilers, this paper describes in detail the technique for one language, ALGOL C, a variant of ALGOL 60 [1]. The variation, as described in section 2, does no violence to the important basic concepts of ALGOL 60.

It is believed that the inclusion of definition making in programming is a natural development to be expected from the increasing sophistication of language design. In any event, programming of definitions is to be preferred to the current pastime of tediously redesigning programming languages to handle newly arisen, apparently unforeseen,operators and operands.

Curiously, definitions in some programming languages have been in use for some time, e.g., macro systems in assembly languages, and

*) 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).

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

through the fluently applicable but weak device of procedures in all languages. But there has not been any general exploitation of the macro concept in algebraic languages.

The MAD language [2] permits the definition of new operators with unary and binary "contexts", but requires their definition to be in assembly language, and provides little interaction between definitions. CDC Fortran 63 [3] allows up to three additional types of arithmetic with existing operators, but the definitions merely produce calls on procedures. This paper attempts a more general exploitation of the macro concept, albeit an incomplete one, since it has nothing to contribute to the problem of diversification of control through definitions.

As is customary in other macro systems, the processing of definitions follows the simple pattern: Take a language x, extend its syntax to a language x' to include that syntax necessary for phrasing and using definitions, and then reduce a text in the extended syntax to an "equivalent" one in language x. We beg the reader's forgiveness for giving in this paper neither a precise definition of "equivalence" nor even an informal proof that given such a reasonable definition the reduction process does yield an equivalent program. But such has not yet been given for the translators that map ALGOL programs into machine code, either!

As mentioned above, it is convenient to look upon definitions as altering the syntax of a programming language, i.e., certain syntax units are assumed variable. While a range of choice is possible, this paper limits its treatment to variability of the syntax units: <block head>, <type>, <assignment statement>, <arithmetic expression>, and <Boolean expression>. This level of variability permits the definition of new operators and new data structures. For example, one who works extensively with matrix arithmetic writes $K = c \times \underline{\underline{T}}((A \times \underline{\underline{T}}B) \times \underline{\underline{T}}(G + D))$ where $\underline{\underline{T}}$ is the transpose operator, c is a scalar, A and B are n × n matrices, and D, G, and K are m × n matrices. The level of detail, i.e., iterations and subscription sequences, needed to carry out this computation on the elements of the matrices should be generated by the processor. Another person might wish to redefine real multiplication so as to force a non-standard rounding.

Furthermore, an implementation of the techniques described in this paper should be possible within the general framework of any ALGOL 60 translator. Analogous techniques can obviously be given for other algebraic languages.

A set of definitions assigns a syntax to the above-mentioned five syntactic units and defines, much as ALGOL 60 and ALGOL C are defined, the syntax of a language called ALGOL D. Thus there are many ALGOL D's. In order to give each an interpretation, the technique developed here gives a reduction of any of them to one fixed language, ALGOL C. The scope of definition of an ALGOL D is dynamically controlled by block boundaries in the following way.

The definitions occur only in a <block head> 1) and fix the five variable syntax units for the block having that heading, i.e., define an ALGOL D over that <block>.

On exit from a <block>, the syntax of the five variable syntax units reverts to that which was in effect on entry to the <block>, i.e., a reversion to a previous ALGOL D. The interaction of new definitions with those in effect on <block> entry will be described in section 3. It will be seen that only one ALGOL D is in effect; i.e., defined, at any one time in the processing of a program.

The most important feature of the implementation described here is that both the number of iterations and the amount of temporary storage needed for intermediate results in the evaluation of expressions are drastically reduced by the use of a replacement rule which avoids any unnecessary introduction into the code of iterations and "workspace" arrays. This is particularly important with

1) Throughout this paper the <> notation, such as in <block head>, is used to assign a technical intent to the bracketed words "block head", i.e., as one of a set of legal strings (not <string>s) defined by the syntax precisely as in ALGOL 60.

arrays, for example, and is difficult to achieve when working with collections of individual <procedure>s. This effect will be demonstrated by the examples in section 5.

Section 2 describes ALGOL C by listing its differences from ALGOL 60. Section 3 describes the syntax common to all ALGOL D's, concentrating primarily on the new constructions needed to write definitions. This discussion of syntax is followed by a description of the processing to be carried out on each of the new constructions and the generation of a new ALGOL D from the results of this processing. This process, when applied to the definitions of Figure 1, generates the syntax of <arithmetic expression>, <Boolean expression>, and <assignment statement> of ALGOL C. Once a new ALGOL D is generated, the process of determining the "legal" expressions in this language is then discussed. Section 4 is concerned with the processing of declarations and text once an ALGOL D has been defined. A "replacement rule" is given which reduces ALGOL D text to ALGOL C text. In section 5 several examples are given of definition sets: matrix arithmetic, file processing, and list processing. In the matrix set, the strategy for obtaining a useful set of definitions is presented, as well as the detailed application of all of the major features of this paper. Section 6 deals with the generalization possible with sets of definitions. Complex arithmetic definitions are introduced here, leading to a combined set of definitions for complex and real matrix arithmetic.

2. Description of ALGOL C

ALGOL C coincides with ALGOL 60 (revised) [1] except for the following:

(1) The addition of six standard functions:

<u>name</u>	<u>argument type</u>	<u>value type</u>
name of	<procedure identifier>, <array identifier>, <u>real</u>, <u>Boolean</u>, <u>integer</u>	<u>integer</u>

rc of [1]	<u>integer</u>	<u>real</u>
Bc of	<u>integer</u>	<u>Boolean</u>
ic of	<u>integer</u>	<u>integer</u>
pic of	<u>integer</u>	<procedure identifier>
aic of	<u>integer</u>	<array identifier>

The value name of(x) will always be a positive integer and may be interpreted as the index of x in a vector consisting of some segment of storage. The arguments of the five other functions must always be positive integers, since they are intended to apply to values which could be produced by the procedure name of.

(2) The addition of <structured expression>:

It will be required of ALGOL C that <arithmetic expression>s and <Boolean expression>s be able to have an array of values. Accordingly, the value of a function designator may need to be an array of values, also.

Syntax:

```
<program expression (p.e.) 2)>::= <type><program>[<bound pair list>]
          |<program>[<bound pair list>]|<type><program>|<program>
<function expression (f.e.)>::= <function designator>|(pic of(<arithmetic
          expression>))
<list of arithmetic expressions (l.a.e.)>::= <arithmetic expression>|
          <l.a.e.>, <arithmetic expression>
```

[1] These represent real contents of, Boolean contents of, integer contents of, procedure identifier contents of, and array identifier contents of, resp.

[2] Abbreviations introduced in syntax definitions will be used in other syntax definitions, but not in text.

```
<list of Boolean expressions (l.B.e.)>::= <Boolean expression>|
          <l.B.e.>, <Boolean expression>
<arithmetic type>::= integer | real
<enumerated arithmetic expression (e.a.e.)>::= <arithmetic type>
          (<l.a.e.>)[<bound pair list>]|<arithmetic type>(<arithmetic
          expression>)|(<arithmetic expression>)|(<l.a.e.>)[<bound pair list>]
<enumerated Boolean expression (e.B.e.)>::= Boolean (<l.B.e.>)
          [<bound pair list>]|Boolean (<Boolean expression>)
<enumerated expression (e.e.)>::= <e.a.e.>|<e.B.e.>
<structured expression (s.e.)>::= <p.e.>|<function designator>|(aic of
          (<arithmetic expression>))
```

(3) The revision of <procedure declaration>:

```
<procedure declaration>::= <prologue><procedure heading><procedure
          body>
```

and the addition of:

```
<prefix>::= <type>|<type> array[<bound pair list>]|array[<bound pair
          list>]
<prologue>::= procedure|<prefix> procedure
```

(4) The revision of <function designator>, <actual parameter>, <subscripted variable>:

```
<function designator>::= <procedure identifier><actual parameter part>|
          (pic of(<arithmetic expression>))<actual parameter part>
<actual parameter>::= <string>|<expression>|<array identifier>|<switch
          identifier>|<procedure identifier>|<s.e.>
<subscripted variable>::= <array identifier>[<subscript list>]|<s.e.>
          [<subscript list>]
```

(6) The revision of <left part> in <assignment statement>:

<left part>::= <variable>:= |<procedure identifier>:= |<procedure identifier>[<subscript list>]:= |r := |r[<subscript list>] :=

The value assigned to a <program expression> is the value on exit from that <program expression> of a variable r which occurs in it. In each <program expression> Q there must be exactly one occurrence of the variable r exterior to any other <program expression> contained in Q. This occurrence may be subscripted and must be the only variable in the <left part list> of an <assignment statement>.

If the <type> of a <structured expression> is omitted, it is real. The omission of the <bound pair list> allocates to the value of a <program expression> the structure of an individual variable of its <type>. Thus the <bound pair list> and the <type> specify the amount and organization of the storage to be allocated to the value of the <program expression> each time it is computed.

Example: (i) real(x,y)[1:2][j] may represent the real or imaginary part of the complex number x + iy.
(ii) {Program to compute the elements of the kth power of a real n × n matrix A}[1:n,1:n][i,j].

The array[<bound pair list>] in the <prologue> of a <procedure declaration> specifies the amount and organization of the storage to be allocated to the value of the procedure. In general such declarations are to be used in the context of <function designator>s and a component of the value is accessed with a <function expression>. Along with the agreement required by the actual-formal correspondence, such procedures require an agreement of the number of subscripts in the <function expression> with that predicated by the <bound pair list> in the <procedure declaration>.

Example: **real** **array**[1:2] **procedure** rationaladd(x,y); **array** x,y;[1)]
b ... **e**;
...
... rationaladd(w, (t + 3,h - 7)[1:2])[1]...

3. Definitions in ALGOL D

Even though we have indicated that there will be many ALGOL D's, they are actually almost identical, the exception being the definitions of the five syntax units listed in 1) below. The common syntax of all ALGOL D's is:

1) that of ALGOL C with the syntax definitions of <block head>, <type>, <arithmetic expression>, <Boolean expression>, and <assignment statement> deleted,

and 2) the following additions:

a) The syntax for <block head>:

<declaration list>::= <declaration>|<declaration list>; <declaration>

<new syntax element>::= <primitive representation>|<context definition>|<set definition>|<new operator declaration>

<new syntax list>::= <new syntax element>|<new syntax list>; <new syntax element>

<block head>::= **begin** <new syntax list>; <declaration list>;|**begin** <declaration list>;|**begin** <new syntax list>;

b) The syntax for <type>:

<boldface character (b.ch.)>::= **a**|**b**|**c**|**d**|**e**|**f**|**g**|**h**|**i**|**j**|**k**|**l**|**m**|**n**|**o**|**p**|**q**|**r**|**s**|**t**|**u**|**v**|**w**|**x**|**y**|**z**|**A**|**B**|**C**|**D**|**E**|**F**|**G**|**H**|**I**|**J**|**K**|**L**|**M**|**N**|**O**|**P**|**Q**|**R**|**S**|**T**|**U**|**V**|**W**|**X**|**Y**|**Z**|**0**|**1**|**2**|**3**|**4**|**5**|**6**|**7**|**8**|**9**

<boldface symbol (b.s.)>::= ⊔<b.ch.>|<b.s.><b.ch.> [2)]

---

1) The symbols **b** and **e** are used throughout to represent **begin** and **end**, resp.

2) In this paper the leading character (⊔) will be suppressed where no ambiguity is possible.

```
<fixed type (f.t.)>::= real | Boolean | integer
<basic new type (b.n.t.)>::= <b.s.>|<f.t.>
<actual type parameter (a.t.p.)>::= <actual parameter>|<a.t.>
<actual type parameter list (a.t.p.l.)>::= <a.t.p.>|<a.t.p.l.>,
        <a.t.p.>
<actual type (a.t.)>::= <b.n.t.>|<b.s.>(<a.t.p.l.>)
```

c) The syntax for <new syntax element>:

(i) for <primitive representation>:

```
<new type (n.t.)>::= <b.s.>|<b.s.>(<a.t.p.l.>)
<representation>::= <a.t.>|<a.t.> array [<bound pair list>]
<primitive representation>::= <n.t.> means <representation>|
        <c.t.s.n.> means <representation>
```

Example: matrix(m,n) means array [1:m,1:n];

(ii) for <new operator declaration>:

```
<precedence relation (p.r.)>::= <|=|>
<operator (o.)>::= <b.s.>|<arithmetic operator>|<logical opera-
        tor>|<relational operator>|,
<current operator (c.o.)>::= <o.>
<new operator (n.o.)>::= <o.>
<new operator declaration (n.o.d.)>::= <n.o.><p.r.><c.o.>|<o.s.n.>
        <p.r.><c.o.>
```

Example: T > ×;

(iii) for <set definition>:

```
<operator set name (o.s.n.)>::= <b.s.>
<operator list element (o.l.e.)>::= <o.>| := |<o.s.n.>
<operator list (o.l.)>::= <o.l.e.>|<o.l.>, <o.l.e.>
<operator set definition (o.s.d.)>::= <o.s.n.>:= (<o.l.>)
<context type list (c.t.l.)>::= <c.t.>|<c.t.l.>, <c.t.>
<context type set name (c.t.s.n.)>::= <b.s.>
<context type set definition (c.t.s.d.)>:= <c.t.s.n.>:= (<c.t.l.>)
<set definition>::= <o.s.d.>|<c.t.s.d.>
```

Example: R := (=,≠,<,≤,>,≥);

(iv) for <context definition>:

```
<formal type parameter (f.t.p.)>::= <formal parameter>|<a.t.p.>
<formal type parameter list (f.t.p.l.)>::= <f.t.p.>|<f.t.p.l.>, <f.t.p.>
<context type (c.t.)>::= <b.n.t.>|<b.s.>(<f.t.p.l.>)|<c.t.s.n.>
<result type (r.t.)>::= <c.t.>
<context typed identifier (c.t.i.)>::= <c.t.><identifier>
<current context operator (c.c.o.)>::= <c.o.>|<o.s.n.>
<basic context (b.c.)>::= <c.t.i.>|<a.t.>[program][[bound pair list]]1)|
        (<c.>)
<subscripted context (s.c.)>::= <b.c.>|<b.c.>[<c.t.i.>]|<s.c.>[<c.t.i.>]
<context (c.)>::= <s.c.>|<c.c.o.><s.c.>|<s.c.><c.c.o.><s.c.>| if <c.>
        then <s.c.> else <c.>
<left side (l.s.)>::= <c.t.i.>|<l.s.>[<subscript list>]
<assignment context (a.c.)>::= <l.s.>:= <c.>
<context definition (c.d.)>::= <c.>:= <r.t.><string>|<c.>:= <r.t.>
        <a.c.>:= <r.t.><string>|<a.c.>:= <r.t.>
```

```
Example: row(u)a × row(u)b := real 'real b integer j;
           real s; s := 0; for j := 1 step 1 until u do
           s := s + a[j] × b[j]; r := s e';
```

The alphabet of ALGOL C contains certain characters which are, to say the least, difficult to distinguish from certain sequences of characters from the set of <boldface character>s; e.g., if and for. To remove this conflict, this paper will not contain any <boldface symbol> which resembles a character present in ALGOL 60.

In a <block> any <current context operator> occurring in a <basic

---

1) The boldfaced brackets indicate an ALGOL C syntax unit, for whose analysis the ALGOL C analyzer will have to be called upon.

context> must have already been defined either by appearing as a <new operator> in an earlier <new operator declaration> or by being an <arithmetic operator>, <relational operator>, or <logical operator> of ALGOL C. Any occurrence of an <actual type> appearing in a <representation> must have been preceded by its occurrence as a <new type> in a <primitive representation>, unless it is a <fixed type>. Any <context type> must appear as either a <new type> in a <primitive representation>, or as the <result type> in a <context definition>.

Naturally, it is not intended that the <string> appearing in a <context definition> be amorphous. As will be clear in section 4, it is a form which, after certain specific editing operations are performed, must be a "legal" <arithmetic expression>, <Boolean expression>, or <assignment statement> for some ALGOL D. In addition, there may appear in the <string> the following:

(i) A single pair of matching boldfaced parentheses such that (is immediately preceded by a <result type>, and

(ii) an assignment to a subscripted r contained between the matching boldfaced parentheses. (As will be seen, this structure will have disappeared before the <string> is processed with an ALGOL D syntax.)

(iii) An asterisk (*) preceding some occurrences of variables. [1] (A variable declared to be of a <new type> has two <type>s associated with it: its <new type> in ALGOL D and its ALGOL C <type> derived from the expansion of its <primitive representation>, described in detail later in this section. The programmer may indicate that an occurrence of a variable has its ALGOL C <type> in a context definition (i.e., in the <string>) by preceding that occurrence by an asterisk.)

[1] Strictly speaking, <delimiter> must also be redefined to include (,), *, and means. We ignore it since <separator> has no contact with any of the concepts treated here.

Furthermore, the <string> may not be perfectly general in the sense that, when parsed, no structures of the form <new syntax element> may be encountered, i.e., no definitions may contain other definitions. (This is a recognized limitation, and could be eliminated at the expense of some extra complexity in the performance of the replacement rule of section 4.)

```
Example:[1]'matrix(u,v)a + matrix(u,v)b := matrix(u,v)
           'matrix(u,v) b integer i; for i := 1 step 1 until u
           do row(v)(r[i] := a[i] + b[i]) e';
```

The relation of an ALGOL D to ALGOL C can now be made clear:

Upon entry to a block, two tables are assumed to exist:

1) a type table,

2) a context table containing two sub-tables: the table of expression contexts and the table of assignment contexts, whose lines are labelled by operators. The assignment context table contains only that line of the context table labelled by :=. The expression context table consists of the remaining lines.

The contents of the two tables determine a syntax for each of: <type>, <block head>, <assignment statement>, <arithmetic expression>, and <Boolean expression>. These, together with the common syntax of all ALGOL D's, yield the syntax of the ALGOL D which is in force throughout the processing of the <new syntax list> of the <block> just entered. When the <block> is entered, copies of these two tables are made, and the <new syntax list> of the new <block> may alter these copies. A general description of the processing of the <new syntax list> is given here; details will be found in Appendix A.

<Set definition>s are used to generate copies of other <new syntax element>s. Each generated copy contains a different representative from the set chosen in all possible ways. The <primitive representation> causes an entry to be made in the type table. These are later used to allocate storage for variables declared to have the <new type> involved, since each <primitive representation> describes its <new type> in terms of ALGOL C arrays. This may involve a chain of reductions, such as

[1] The examples used here are generally taken from the more complete examples in sections 5 and 6.

```
    matrix(complex,m,m) means complex array [1:m,1:m];
    complex means array [1:2];
```

which leads to the entries in the type table:

```
    matrix(complex,m,m) means array [1:m,1:m,1:2];
    complex means array [1:2];
```

Only <type>s in the type table are allowed in <declaration>s. A <new operator declaration> establishes a precedence for a <new operator> relative to existing operators, and it also labels a line in the context table with that operator. <Context definition>s involving that <new operator> as their principal operator will be placed in the line which it labels.

A <context> is an occurrence of one or more <operator>s, together with the <type>s of their operands. <Context>s are the natural unit of definition for <assignment statement>s and expressions, since it is always necessary to determine the <context> in which an operator occurs before one can replace it with "code". This is true even for ALGOL C, since there are quite different definitions for: real x/real y, real x/integer y, integer x/real y, and integer x/integer y. In a <context definition> a <result type> is given as the <type> of the value of the expression represented by the <context>, and the associated <string> is the "definition" of the <context>; i.e., text to replace an occurrence of the <context> during the reduction of ALGOL D text into ALGOL C text. (When no <string> is present, recognition of the <context> in a piece of text induces no change in the text.) The <string> need not be itself in the ALGOL C language; it merely reduces the ALGOL D text to something "closer" to ALGOL C. There may be <context>s in the <string> which will need replacement by other <string>s in turn.

During the processing of the <new syntax list> in a <block head>, any <context definition>s that are encountered are entered into a line of the context table determined by the relative precedence of the <context>'s principal operator among the set of <current context operator>s; i.e., among those operators which have appeared in ALGOL C or in <new operator declaration>s. Some <context definition>s generate others by means of the "boldface parentheses" notation, which is a device for implying subscription <context>s.[1] For example, the <context definition>:

```
matrix a + matrix b := matrix 'matrix
  b integer i; for i := 1 step 1 until n do
  row(r[i] := a[i] + b[i]) e';
```

would lead to two entries (which no longer contain (and)):

for subscription:

```
(matrix a + matrix b)[integer i] := row
              'r[i] := a[i] + b[i]';
```

for +:

```
matrix a + matrix b := matrix 'matrix b integer i;
              for i := 1 step 1 until n do
              br[i] := a[i] + b[i]  ee';
```

As indicated below, when the last declaration in the <new syntax list> has been examined, the two modified tables define the syntax of the five variable syntax entities which is to remain in force for the duration of the current block. A possibly new ALGOL D has been defined. Finally, on exit from this block, the tables are reconstituted as they were on entry, and the ALGOL D in effect on entry to the block is reinstituted as the current ALGOL D. Appendix B describes in detail how to generate the set of "legal" ALGOL D expressions and <assignment statement>s from the type and context tables. In general, these expressions and <assignment statement>s are just what one would expect to obtain from adding new operators to ALGOL C. They are generated by listing first those <context>s which are directly definable in ALGOL C, then those definable in terms of the <context>s already chosen, and so on. When the <context typed identifier>s are replaced in these <context>s by actual <identifier>s of the correct <type>, legal expressions in ALGOL D are obtained.

Initially, the context table contains four empty, but labelled lines:

1) Moreover, every context definition which has a string and result type> γ generates a <context definition>: γ[program][[bound pair list]] := γ.

(↓) [1]:

(if then else):

(program expression):

(no operator):

The type table contains real, integer, and Boolean. We assume the existence of an outermost block supplied by the "environment", which contains the definition set for ALGOL C. This is given in Figure 1.

```
arith := (real, integer); any := (arith, Boolean); M := (×,÷); M1 := (×,/);
A := (+,-); R1 := (>,=,≥,≤,≠); R := (R1,<); B := (∧,∨,⊃,≡);
⊕ > ↓; ↑ < ↓; × < ↑; ÷ = ×; / = ×; + < ×; - = +; < < +; R1 = <; ¬ < <;
∧ < ¬; ∨ < ∧; ⊃ < ∨; ≡ < ⊃; , < ≡;
any a ⊕ arith b := any;      any a ⊕ Boolean b := any;
any a ↓ arith b := any;
arith a ↑ arith b := real;
arith a M1 arith b := real;
integer a M integer b := integer;
arith a A arith b := real;
integer a A integer b := integer;
     A arith a := arith;
arith a R arith b := Boolean;
    ¬ Boolean a := Boolean;
Boolean a B Boolean b := Boolean;
if Boolean a then Boolean b else Boolean c := Boolean;
if Boolean a then arith b else arith c     := real;
if Boolean a then integer b else integer c := integer;
Boolean a := Boolean b                     := Boolean;
real a := arith b                          := real;
integer a := arith b                       := integer;
any a, any b                               := real;
(any a)                                    := any;
any [program] [[bound pair list]]          := any;
```

Figure 1 The Definition Set for ALGOL C.

[1] The symbol ↓ represents subscription, treated as a binary operator. Similarly, ⊕ represents function evaluation. The ALGOL 60 (implied) forms will also be acceptable.

4. The reduction of an ALGOL D program to an ALGOL C program

Since the syntax of an ALGOL D program is that of ALGOL C with the exception of <arithmetic expression>, <Boolean expression>, <assignment statement>, <block head>, and <type>, an ALGOL D program can be parsed by the syntax of ALGOL C, except for these five syntax units. For each of these units let $L_x(s)$ be the set of legal strings having syntax s in ALGOL x. To each s there corresponds an analyzer $A_D(s,\rho)$ which has the property that: if $\rho \varepsilon L_D(s)$ then $A_D(s,\rho) = t(\rho)$, where $t(\rho) \varepsilon L_C(s)$, or $t(\rho)$ is the null string. We wish to give the algorithm by which $t(\rho)$ is produced.

The processing in <block head> of <new syntax list> has already been described (section 3) and no ALGOL C text is produced from this processing. Following the parsing of <new syntax list>, the ALGOL D for the current block is completely defined. The processing of <declaration>s by ALGOL C is interrupted only by the necessaty of analyzing <type>s, when $A_D(\text{<type>},\rho)$ replaces an occurrence of an <actual type> by its <representation> as given in the type table. Otherwise <declaration>s are parsed by the ALGOL C parser. The analyzer $A_D(s,\rho)$, for an s which is one of <arithmetic expression>, <Boolean expression>, or <assignment statement> operates as follows:

Since an ALGOL D is defined, any text ρ of one of these three kinds can be parsed using the context tables for that ALGOL D. This parsing yields a tree representation for ρ. The terminals are identifiers and the nodes are operators and delimiters, such as <u>b</u>, $\downarrow$, $+$,   and instances of <for clause>s. The tree has either zero, one, two, or three branches [1] emanating from any node. Furthermore, each node and terminal element is labeled with an <actual type>. Similarly, any <context> in the context table can be parsed into a tree (called a "context tree") whose terminals are <context typed identifier>s.

[1] The conditional expression will be represented as a single node with three branches labelled <u>if</u>, <u>then</u>, and <u>else</u>, resp.

Suppose, then, that some text ρ has been encountered in the block which is an <assignment statement> or expression. The reduction of ρ to ALGOL C text consists of the following steps (given in detail in Appendix C):

(1) Parse ρ into a tree with labelled nodes, as described above.

(2) Select a sub-tree for which a matching context exists in the context table.

(3) Using the <string> associated with the selected <context>, construct its tree representation, and replace <context typed identifier>s in this tree by their correspondents (via the match) from ρ.

(4) Substitute the new tree for the sub-tree selected in (2).

Since each <context definition> thus invoked is written to provide a <string> which is "closer" to ALGOL C, repeated substitutions such as are described in steps (2) to (4) eventually lead to the desired t(ρ).

Although care must be taken in the details of "matching" trees with <context>s, the rule given in Appendix C would be much simpler if it were not also concerned with avoiding the introduction of unnecessary iterations and temporary storage allocations. As an example of the technique used for this, suppose that the text being processed at some stage is:

```
w[i] := x[i] + y[i] + (v × z)[i];
```

where w, x, y, v, and z are matrices, and w[i] is the i-th row of w. Suppose, also, that the following <context definition> is selected by the matching process:

```
(matrix a × matrix b)[integer i] := row 'row b row t;
t := a[i]; r := t × b e';
```

While a straightforward substitution produces:

```
w[i] := x[i] + y[i] + row b row t; t := v[i]; r := t × z e;
```

the replacement rule generates the following, eliminating the need for a row of temporary storage for r:

```
b row t; t := v[i]; w[i] := x[i] + y[i] + t × z e;
```

Additional examples will be found in section 5.

5. Examples of definition sets

1. A Matrix Arithmetic Definition Set

As an example of the development of a definition set, we consider the definitions necessary to allow arithmetic expressions and assignment statements involving matrices with dimensions m × m, m × n, n × m, and n × n (where m and n are non-local). The set is developed in stages, as follows:

I. The <primitive representation>s are given:

```
matrix(m,m) means array [1:m, 1:m];
matrix(m,n) means array [1:m, 1:n];
matrix(n,m) means array [1:n, 1:m];
matrix(n,n) means array [1:n, 1:n];
```

II. The <new operator>s are declared:

```
T > ×; I = T;
```

These represent the transpose and inverse operators.

III. The desired <context definition>s are listed:

```
(1) matrix(u,v) a := matrix(u,v) 'matrix(u,v) b integer i;
                     i → u do row(v) (r[i] := a[i]) e'; 1)

(2) matrix(u,v) a := matrix(u,v) b := matrix(u,v)
                     'b integer i; i → u do a[i] := b[i] e';

    A := (+,-);

(3) matrix(u,v) a A matrix(u,v) b := matrix(u,v) 'matrix(u,v)
                     b integer i; i → u do row(v) (r[i] :=
                     a[i] A b[i]) e';

    arith := (real, integer);
```

1) The notation i → u is an abbreviation for: for i := 1 step 1 until u.

```
(4) arith a × matrix(u,v)b := matrix(u,v) 'matrix(u,v)
                               b integer i; i → u do row(v)(r[i] :=
                               a × b[i]) e';

(5) matrix(u,v)a × matrix(v,w)b := matrix(u,w)
                               'matrix(u,w) b integer i; i → u do
                               row(w)(b row(v)t; t := a[i];
                               r[i] := t × b) e';

(6) T matrix(u,v)a := matrix(v,u);

(7) I matrix(u,u)a := matrix(u,u) 'inv(a,u)';
```

IV. The <string>s introduced in III make additional <context definition>s and <declaration>s necessary:

```
(8) row(v)a := row(v)b := row(v) 'b integer j; j → v do
                               a[j] := b[j] e';          (from (1)).

(9) row(v)a A row(v)b := row(v) 'row(v) b integer j;
                               j → v do real (r[j] := a[j] A b[j]) e';
                                                         (from (3))

(10) arith a × row(v)b := row(v) 'row(v) b integer j;
                               j → v do real (r[j] := a × b[j]) e';
                                                         (from (4))

     row(m) means array [1:m]; row(n) means array [1:n];

(11) row(u)a × matrix(u,v)b := row(v) 'row(v) b integer j;
                               j → v do real (r[j] := a × (Tb)[j]) e';
                                                         (from (5))
```

V. Additional <context definition>s arise from the <string>s in IV:

```
(12) row(v)a[integer j] := real '*a[j]';                 (from (8))

(13) row(v)a × row(v)b := real 'real b integer j; real s;
                        s := 0; j → v do s := s + a[j] × b[j]; r := s e';
                                                         (from (11))
```

VI. Additional <context>s arise when one <context> with <result type> γ is substituted for an occurrence of a <context typed identifier> with <context type> γ in another <context>. Only those need be listed for which a special <string> is desired:

```
(14) T (matrix(u,v)a A matrix(u,v)b) := matrix(v,u)
                                   'Ta A T b';       (from (3) into (6))

(15) T (matrix(u,v)a × matrix(v,w)b) := matrix(w,u)
                                   'Tb × Ta';        (from (5) into (6))

(16) T (T matrix(u,v)a) := matrix(u,v) 'a';          (from (6) into (6))

(17) T (I matrix(u,u)a) := matrix(u,u) 'I(Ta)';(from (7) into (6))

(18) I (I matrix(u,u)a) := matrix(u,u) 'a';          (from (7) into (7))

(19) matrix(u,v)a [integer i][integer j] := real '*a[i,j]';
                                                     (from (1) into (12))

(20) (T matrix(u,v)a) [integer i][integer j] := real 'a[j][i]';
```

This set of <context definition>s does not provide a correct ALGOL C program when a matrix appears on both sides of the := symbol, and on the right side in a context other than as the left-most factor in a product. For example, A := B := A × B × C would give incorrect results for A and B, but A := A × B × C would work correctly.

It is interesting to note that definition (5) could be written:

```
matrix(u,v)a × matrix(v,w)b := matrix(u,w)
          'matrix(u,w) b integer i; i → u do
          row(w)(r[i] := a[i] × b)';
```

This would eliminate the allocation of the temporary storage for t, but some elements would be computed several times in some expressions.

An example of the use of this definition set is:

```
real c; matrix(m,n)K, G, D; matrix(n,n)A, B;
K := c × T ((A × TB) × T (G + D));
```

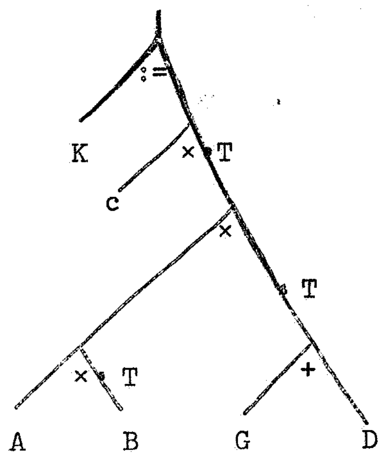
A straightforward ALGOL C program for this computation would be:

```
real c; array K, G, D [1:m, 1:n], A, B [1:n, 1:n];
b integer i, j, k; array P [1:n, 1:n], Q [1:m, 1:n]; real s;
     i → n do j → n do b s := 0; k → n do s:= s + A[i,k] × B[j,k];
          P[i,j] := s e;
     i → m do j → n do Q[i,j] := G[i,j] + D[i,j];
     i → m do j → n do b s := 0; k → n do
                    s := s + P[j,k] × Q[i,k]e;
                    K[i,j] := c × s e e
```

This program requires $n^2 + nm + 1$ locations for temporary storage. The expansion via the definition set, using the replacement rule, is:

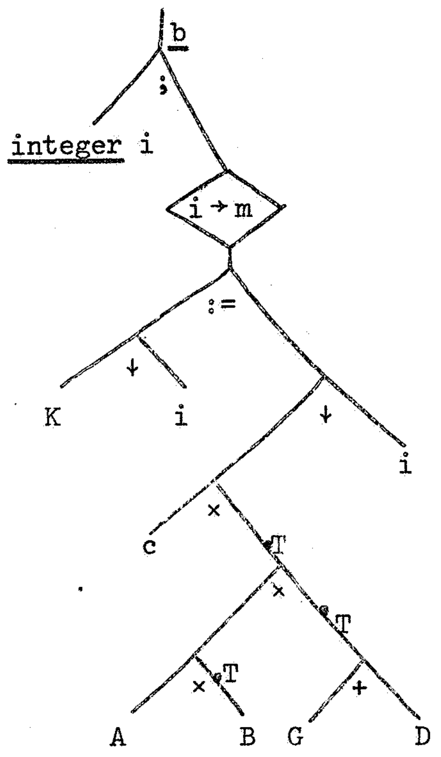
```
real c; array K, G, D [1:m, 1:n]; array A, B [1:n, 1:n];
b integer i; i → m do b array t [1:n];
                    b integer j; j → n do t[j] := G[i,j] + D[i,j] e;
                    b integer j; j → n do b array z [1:n];
                         b integer k; k → n do z[k] := A[j,k] e;
                         b integer k; real s; s := 0;
                              k → n do b integer h; real x; x := 0;
                                   h → n do x := x + z[h] × B[k,h];
                                   s := s + t[k] × x e;
                              K[i,j] := c × s e e e e e;
```

This program requires only 2n + 2 locations for temporary storage and will execute in less time. Some of the transitional states of the tree for this expansion were as follows (number in parentheses refer to definitions invoked):



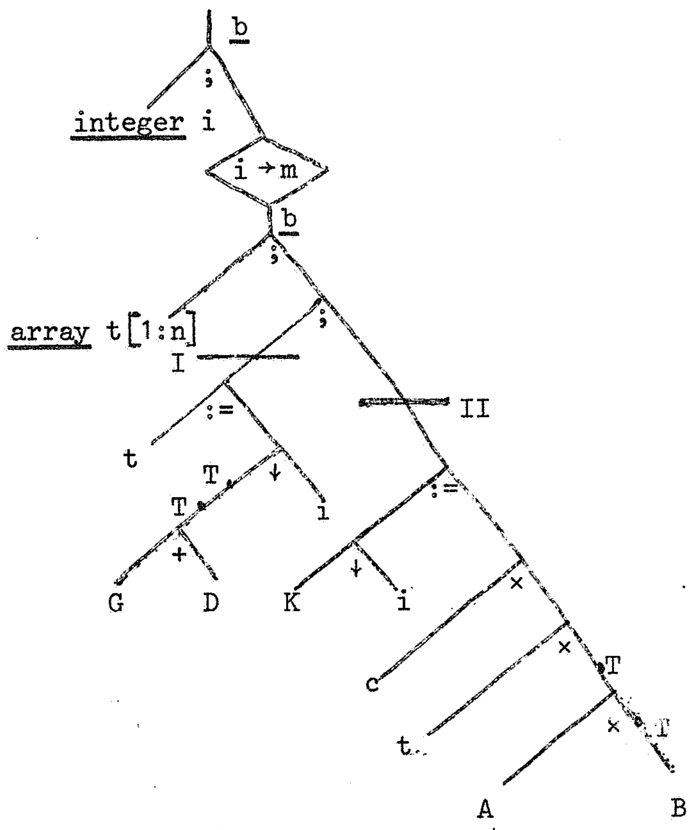
:=
K
×
T
c
×
T
×
T
+
A
B
G
D

(2):



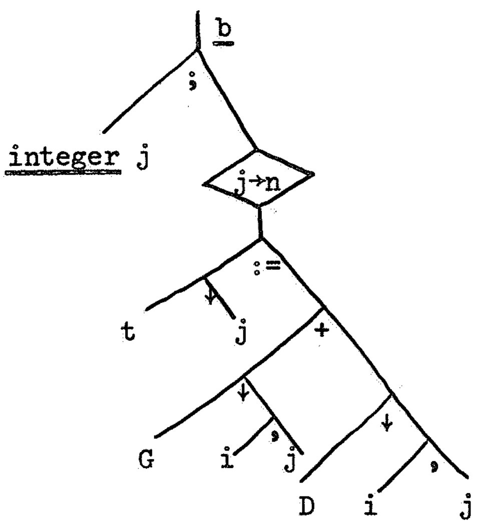
b
;
integer i
i → m
:=
↓
K
i
↓
i
c
×
T
×
T
×
T
+
A
B
G
D

(4) → (15) → (5):

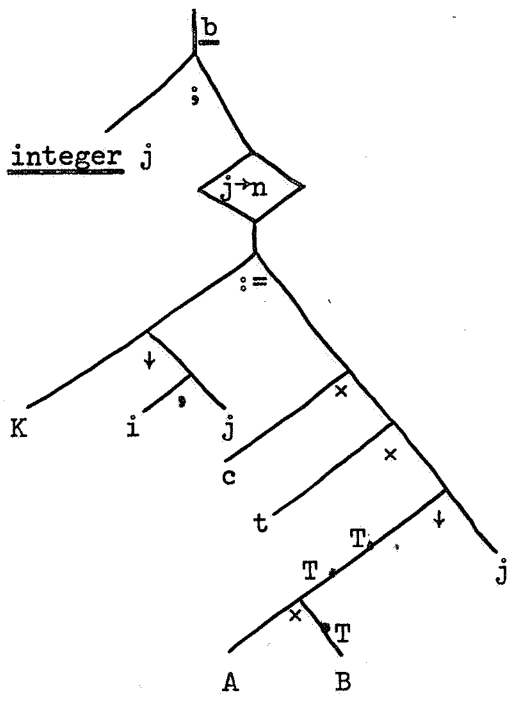


b
;
integer i
i → m
b
;
array t[1:n]
;
I
II
:=
t
T
↓
T
i
+
G
D
:=
↓
K
i
×
c
×
T
t
×
T
A
B

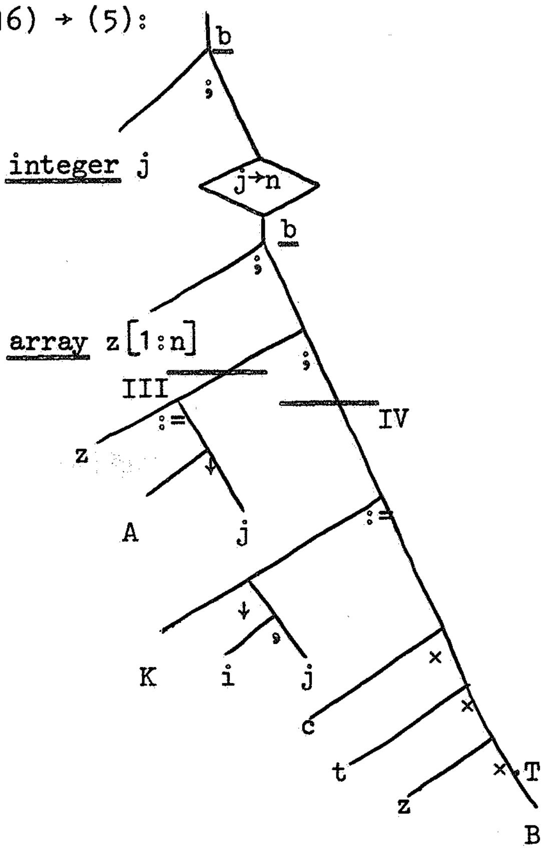
I (16) → (3) → (8) → (19) → (12):



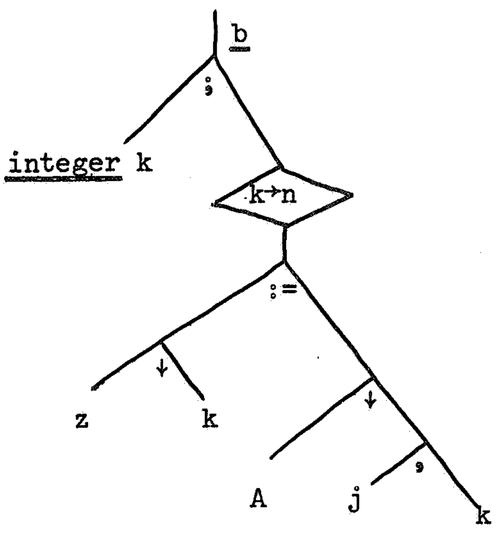
II (8) → (19) → (4) → (11):



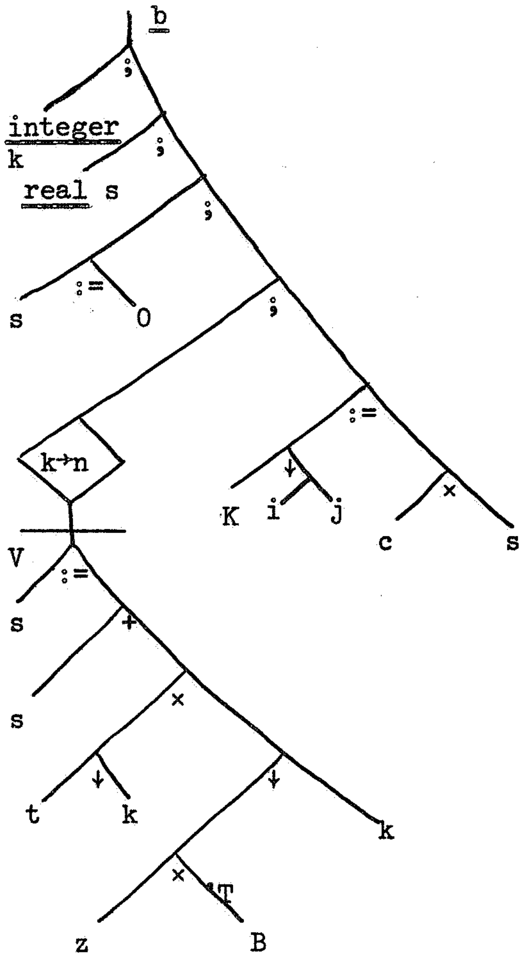
II (16) → (5):



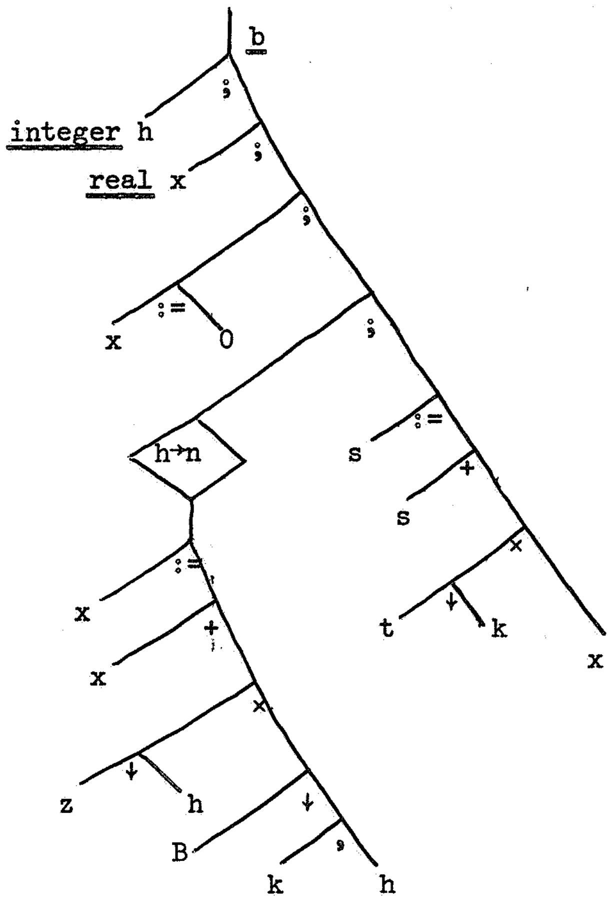
III (8) → (19) → (12):



IV (13):



V (11) → (16) → (12) → (13) → (19) → (12):



2. A File Processing Definition Set

```
B := (∧,∨); A := (+,-,×,/,÷,:=);
R := (=,≠,<,≤,>,≥);
arith := (real, integer); rat := (arith, attr(j));
file means array [1:m];
rec means array [1:n];
TV means Boolean array [1:m];
file a [integer i][integer j] := real 'rc of(a[i] + j - 1)';
file a [integer i] := rec;
rec a [integer j] := real;
TV a [integer i] := Boolean;
on = ↓;
countof < on;
countof . TV a := real 'real b real c; integer i; c := 0; i → m do
                          c := b + if a[i] then 1 else 0; r := c e';
Boolean a on file x := TV 'TV b integer i; i → m do
                          Boolean(r[i] := a on x[i]) e';
rat a on file x := file 'file b integer i; i → m do
                          rec(r[i] := a on x[i]) e';
(Boolean a B Boolean b) on rec c := Boolean 'a on c B b on c';
(¬Boolean b) on rec c := Boolean '¬(b on c)';
(rat a R rat b) on rec c := Boolean 'a on c R b on c';
(rat a A rat b) on rec c := rec 'a on c A b on c';
attr(j) a on rec c := real 'c[j]';
arith a on rec c := arith 'a';
(if Boolean a then rat b else rat c) on rec x :=
            real 'if a on x then b on x else c on x';
rat a R rat b := Boolean;
rat a A rat b := real;
file a := file b := file 'b integer i; i → m do a[i] := b[i] e';
rec a := rec b := rec 'b integer j; j → n do a[j] := b[j] e';
TV a := TV b := TV 'b integer i; i → m do a[i] := b[i] e';
```

The access function: rc of (a[i] + j - 1) and the <primitive representation>s used give a data structure of a vector of m record names. Each record is a vector of n entries. However, no other context depends on the choice of data structure.

```
Example: file x; array A[1:m, 1:n]; attr(1) name; attr(2) sex; attr(3) height;
                  real cnt;
         b integer i; for i → m do x[i] := name of (A[i,1]);
         ...
         cnt := countof(sex = 1∧height ≥ 6) on x;
         ...
```

The example expands to:

```
array x [1:m]; array A[1:m, 1:n];
b integer i; for i → m do x[i] := name of(A[i,1]);
...
b real c; integer i; c := 0;
       i → m do c := c + if rc of(x[i] + 2 - 1) = 1∧
             rc of(x[i] + 3 - 1) ≥ 6 then 1 else 0;
       cnt := c e;
       ...
```

It is assumed here that A is stored by rows.

3. List Definition Set

The following set of definitions is based on the LISP [5] primitives. The basic LISP predicates "atom" and "eq" are assumed to have been defined as Boolean procedures:

```
Boolean procedure atom(x); list x;
     atom := cdr x = 0;
Boolean procedure eq(x,y); list x,y;
     eq := car x = car y ∧ atom(x) ∧ atom(y);
```

'NIL' in LISP is represented here by 0. The following definitions are used to organize lists as structures of names.

```
(1) list means integer array [1:2];
(2) cons = ×;
(3) car > cons;
(4) cdr = car;
(5) of < cons;
(6) list a cons list b := list 'list(a,b)';
(7) car list a := list 'a[1]';
(8) cdr list a := list 'a[2]';
(9) integer a := list b := integer 'a := name of(b)';
```

Note that the conditions usually taken as necessary for the internal consistency of car, cdr, and cons are satisfied here:

```
c = car a cons cdr a = list(car a, cdr a)
                     = list(a[1],a[2])
    so c = a; i.e., c[1] = a[1] and c[2] = a[2].
Also, car(a cons b) = (list(a,b))[1]
                    = a
and cdr(a cons b)   = b.
```

```
(10) op f(F) of list x := list 'E(list(name of(F),0),x)';
(11) op f(F) of op g(G) := list 'list(name of(F), name of(G))';
(12) list y of op f(F) := list 'list(y, name of(F))';
(13) list y of list x := list 'E(y,x)';
```

Context definitions (10) through (13) provide an efficient rule for sequencing through a composition of operations on lists, each one of which operates only on atoms to produce atoms or even lists. The procedure E is organized so that as each atom of data is encountered the remaining operators in the composition are applied to it. Thus the lists are not totally decomposed and composed for each succesive operator. In a <declaration> such as op(H), the <actual type parameter> H represents the <procedure> to be used to apply h to a list.

The block containing these list definitions must also contain the procedure E:

```
list procedure E(f,x); list f,x;
E := if atom(x) then (if atom(f) then (pic of(car f))(x)
     else E(car f; (pic of(cdr f)(x)))) else
     E(f, car x) cons E(f,cdr x);
```

Example:

```
     begin op(F)f; op(G)g; integer c; list a, b, d, h, k;
     ...
     integer procedure subst(x,y,z); list x,y,z; subst :=
          if atom(z) then (if eq(z,y) then x else z)
          else subst(x,y,car z) cons subst(x,y,cdr z);
     list procedure F(x); list x; F := subst(a,k,x);
     list procedure G(x); list x; G := subst(d,h,x);
     c := (f of g) of b end;
```

6. Programming Definitions

Suppose we have a definition set for some application, e.g., the arithmetic of reals. We wish now to construct a collection for an arithmetic which generalizes real arithmetic, e.g., complex arithmetic. We know that the field of real numbers can be mapped isomorphically into the field of complex numbers, and this provides a clue to one method of programming definitions:

(i) Give a <primitive representation> for a complex number.

(ii) Assume all reals are mapped into their corresponding complex representations.

(iii) Give the arithmetic as a set of definitions whose <context>s involve only complex quantities.

(iv) The definitions involve manipulations of complex numbers in terms of their <primitive representation>s, i.e., as ALGOL C reals.

(v) Define a predicate to determine whether any complex number is in fact real, so that, e.g., x > y can be given its real interpretation when x and y are real.

This approach has the virtue of simplicity. However, in applications where much of the arithmetic and/or the data is <u>real</u>, a grotesque inefficiency of execution time and storage use can occur. This method is thus inappropriate for programming.

Alternatively, define not only <u>complex</u> arithmetic but mixed arithmetic. Any identifier may now be declared to be either <u>complex</u> or <u>real</u>. Such a definition set is:

```
A := (+,-); Op := (arg, conj, magsq, itimes, isreal);
arith := (real, integer); complex means array [1:2];
mag > ↑;
Op = mag;
complex a [integer i] := real;
complex a A complex b := complex 'complex(a[1] A b[1],
                                          a[2] A b[2])';
complex a × complex b := complex 'complex(a[1] × b[1] - a[2] × b[2],
                                    a[2] × b[1] + a[1] × b[2])';
complex a/complex b := complex '(a × conj b)/magsq b';
magsq complex a := real 'a[1] ↑ 2 + a[2] ↑ 2';
arg complex a := real 'arg(a)';
conj complex a := complex 'complex(a[1], - a[2])';
itimes complex a := complex 'complex(-a[2], a[1])';
itimes arith a := complex 'complex(0,a)';
mag complex a := real 'sqrt(magsq a)';
arith a A complex b := complex 'complex(a A b[1], A b[2])';
complex b A arith a := complex 'complex(b[1] A a, b[2])';
arith a × complex b := complex 'complex(a × b[1], a × b[2])';
complex b ×arith a := complex 'a × b';
complex b/arith a := complex 'complex(b[1]/a, b[2]/a)';
arith a/complex b := complex '(a × conj b)/magsq b';
complex a ↑ integer b := complex 'ciexp(a,b)';
complex a ↑ real b := complex 'crexp(a,b)';
complex a ↑ complex b := complex 'ccexp(a,b)';
integer a ↑ complex b := complex 'icexp(a,b)';
real a ↑ complex b := complex 'rcexp(a,b)';
```

```
complex a := complex b := complex 'b a[1] := b[1];
                          a[2] := b[2] e';
complex b := arith a := complex 'b b[1] := a; b[2] := 0 e';
arith a := complex b := arith 'a := mag b';
complex a = complex b := Boolean 'a[1] = b[1]∧a[2] = b[2]';
isreal complex a := Boolean 'a[2] = 0';
```

Note that the number of contexts required is of the order of the square of those needed by the first approach. However, since <identifier>s of both types are permissible, better use of the computer will result. Figure 2 shows the type table and the initial part of the context table after the initial ALGOL C definitions and these complex arithmetic definitions have been processed.

Type Table

```
real;
integer;
Boolean;
complex means array [1:2];
```

Context Table

```
(⊕):            real a ⊕ real b := real; real a ⊕ integer b := real;
                integer a ⊕ real b := integer;
                        integer a ⊕ integer b := integer;
                Boolean a ⊕ real b := Boolean;
                Boolean a ⊕ integer b := Boolean;
                real a ⊕ Boolean b := real;
                integer a ⊕ Boolean b := integer;
                Boolean a ⊕ Boolean b := Boolean;

(↓):            real a ↓ real b := real; real a ↓ integer b := real;
                integer a ↓ real b := integer;
                        integer a ↓ integer b := integer;
                Boolean a ↓ real b := Boolean;
                Boolean a ↓ integer b := Boolean;
```

```
(mag, arg, conj,          mag complex a := real 'sqrt(magsq a)';
magsq, itimes, isreal):   arg complex a := real 'arg(a)';
                          conj complex a := complex 'complex(a[1],-a[2])';
                          magsq complex a := real 'a[1] ↑ 2 + a[2] ↑ 2';
                          itimes complex a := complex 'complex(-a[2],a[1])';
                          itimes real a := complex 'complex(0,a)';
                          isreal complex a := Boolean 'a[2] = 0';

(↑):                      real a ↑ real b := real; real a ↑ integer b :=
                                    real; integer a ↑ real b := real;
                          integer a ↑ integer b := real;
                                  complex a ↑ integer b := complex
                                  'ciexp(a,b)'; complex a ↑ real b :=
                                  complex 'crexp(a,b)';
                          complex a ↑ complex b := complex 'ccexp(a,b)';
                          integer a ↑ complex b := complex 'icexp(a,b)';
                          real a ↑ complex b := complex 'rcexp(a,b)';

(×,÷,/):                  real a × real b := real; real a × integer b := real;
                          integer a × real b := real;
                                  integer a × integer b := integer;
```

Figure 2. The type table and the initial part of the context table.

Suppose, now that both the real and complex sets are available and we wish to develop a collection for real and complex matrix arithmetic. Again several choices are open:

A. Construct a definition set for real matrices, as was done in section 5. Then represent a complexmatrix as:

```
complexmatrix(b,c) a := complexmatrix 'complexmatrix(b,c)';
matrix x,y; complexmatrix(x,y) z:
```

i.e., as a pair of real matrices. Unfortunately, it will turn out that iterations over the elements of each matrix will arise and will be executed separately. Thus, the replacement rule will not be as well employed as it might. Furthermore, no mention of the actual complexmatrix will occur in the final ALGOL C program.

B. Construct a definition set for matrices of complex elements. This definition set is very much like the real matrix definition set, and indeed is so because real and complex operations are instances of a set of operations over a field v. This points up the following still better approach.

C. Construct a definition set for a field v. Then let v be a set definition, e.g., v := (real, complex), and provide a <formal type parameter> p in <declaration>s whose <actual type parameter> would be real or complex. The v-matrix definition set would be:

```
     v := (real, complex);
     matrix(v,m,m) means v array [1:m, 1:m];
     matrix(v,m,n) means v array [1:m, 1:n];
     matrix(v,n,m) means v array [1:n, 1:m];
     matrix(v,n,n) means v array [1:n, 1:n];

     T > ×; I = T;

(1) matrix(p,u,v) a := matrix(p,u,v) 'matrix(p,u,v) b integer i;
                        i → u do row(p,v)(r[i] := a[i]) e';
(2) matrix(p,u,v) a := matrix(p,u,v) b := matrix(p,u,v)
                        'b integer i; i → u do a[i] := b[i] e';
    A := (+,-);
(3) matrix(p,u,v) a A matrix(p,u,v) b := matrix(p,u,v)
                        'matrix(p,u,v) b integer i; i → u do row(p,v)
                        (r[i] := a[i] A b[i]) e';
    arith := (v,integer);
(4) arith a × matrix(p,u,v) b := matrix(p,u,v) 'matrix(p,u,v)
                        b integer i; i → u do row(p,v)(r[i] := a × b[i]) e';
(5) matrix(p,u,v) a × matrix(p,v,w) b := matrix(p,u,w)
                        'matrix(p,u,w) b integer i; i → u do row(p,w)
                        (b row(p,v) t; t := a[i]; r[i] := t × b) e';
(6) T matrix(p,u,v) a := matrix(p,v,u);
```

```
 (8) row(p,v) a := row(p,v) b := row(p,v) 'b integer j;
                j → v do a[j] := b[j] e';
 (9) row(p,v) a A row(p,v) b := row(p,v) 'row(p,v) b integer j;
                j → v do p(r[j] := a[j] A b[j]) e';
(10) arith a × row(p,v) b := row(p,v) 'row(p,v) b integer j;
                j → v do p(r[j] := a × b[j]) e';
     row(v,m) means v array [1:m]; row(v,n) means v array [1:n];
(11) row(p,u) a × matrix(p,u,v) b := row(p,v) 'row(p,v) b integer j;
                j → v do p(r[j] := a × (Tb)[j]) e';
(13) row(p,v) a × row(p,v) b := p 'p b integer j; p s; s := 0;
                j → v do s := s + a[j] × b[j]; r := s e';
(14) T (matrix(p,u,v) a A matrix(p,u,v) b) := matrix(p,v,u) 'T a A T b';
(15) T (matrix(p,u,v) a × matrix(p,v,w) b) := matrix(p,w,u) 'T b × T a';
(16) T (T matrix(p,u,v) a) := matrix(p,u,v) 'a';
(17) T (I matrix(p,u,u) a) := matrix(p,u,u) 'I (T a)';
(18) I (I matrix(p,u,u) a) := matrix(p,u,u) 'a';
(20) (T matrix(p,u,v) a)[integer i][integer j] := p 'a[j][i]';
```

Additional <context definition>s must be provided for those which imply specific connections to the underlying field:

```
 (7) I matrix(real,u,u) a := matrix(real,u,u) 'inv(a,u)';
     I matrix(complex,u,u) a := matrix(complex,u,u) 'cinv(a,u)';
(12) row(real,u) a[integer j] := real '*a[j]';
     row(complex,u) a[integer j] := complex;
     row(complex,u) a[integer j][integer k] := '*a[j,k]';
(19) matrix(real,u,v) a[integer i][integer j] := real '*a[i,j]';
     matrix(complex,u,v) a[integer i][integer j] := complex;
     matrix(complex,u,v) a[integer i][integer j][integer k] :=
                                             real '*a[i,j,k]';
```

We assume that this definition set will be preceded by the definition set for complex arithmetic given above. Since real arithmetic is primitive in ALGOL C, it is not necessary to provide a real arithmetic definition set. Note that the first <primitive representation> will expand to include:

```
          matrix(complex,m,m) means real array [1:m, 1:m, 1:2];
```

As a final example, suppose that the <assignment statement> expanded above were written for complex matrices:

```
complex c; matrix(complex,m,n) K, G, D; matrix(complex,n,n) A, B;
K := c × T ((A × TB) × T (G + D));
```

The expansion will proceed as before, but the final tree of the expansion will have many nodes labelled complex, and these will induce further expansions. The fully expanded program now becomes:

```
array c [1:2]; array K, G, D [1:m, 1:n, 1:2]; array
A, B [1:n, 1:n, 1:2];
b integer i; i → m do b array t [1:n, 1:2];
    b integer j; j → n do b t[j,1] := G[i,j,1] + D[i,j,1];
                            t[j,2] := G[i,j,2] + D[i,j,2] e e;
    b integer j; j → n do b array z[1:n, 1:2];
        b integer k; k → n do b z[k,1] := A[j,k,1];
                                z[k,2] := A[j,k,2] e e;
        b integer k; array s[1:2]; b s[1] := 0; s[2] := 0 e;
            k → n do b integer h; array x[1:2];
                b x[1] := 0; x[2] := 0 e;
                h → n do b x[1] := x[1] + z[h,1] ×
                    B[k,h,1] - z[h,2] × B[k,h,2];
                    x[2] := x[2] + z[h,2] × B[k,h,1]
                      + z[h,1] × B[k,h,2] e;
                b s[1] := s[1] + t[k,1] × x[1] - t[k,2] ×
                   x[2]; s[2] := s[2] + t[k,2] × x[1]
                     + t[k,1] × x[2] e e;
                b K[i,j,1] := c[1] × s[1] - c[2] × s[2];
                  K[i,j,2] := c[2] × s[1] + c[1] × s[2] e e e e e e;
```

Summing up we see that the following different approaches can be used when one arithmetic α is included in another β:

(i) Represent α in β, provide for only β <context>s, but give the <primitive representation>s of β in terms of α and those of α in terms of ALGOL C.

(ii) Provide for operations over β in terms of those over α and include these <context>s with those for α. Provide <primitive representation>s for β as in (i).

(iii) Generalize the operations to work for an arithmetic $\underline{v}$ containing both α and β, and also provide for the <context>s linking elements of β to those of α.

7. Conclusions

In the previous sections we have described a technique for programming and processing definitions.

The technique, while complicated, does not seem to require programming methods beyond what are now commonplace in compiler construction. Although ALGOL C was the "supporting language", it should be clear that others having its capabilities could have served.

It is contended that every new language should be capable of being a supporting language; even more, it should permit the programming of definitions on at least the scale described in this paper.

While definition sets will enhance the use of "personal sub-languages", the most important consequence of a definition facility in a language will be the creation of a library of definition sets, whose programming is an algorithmic activity comparable in value to the creation of procedures. The definition facility and the generated libraries will fulfill the need for language change which all experience has shown is so necessary. In fact, a more rational basis for standardization will thereby result.

Acknowledgment: The authors acknowledge and appreciate very helpful criticism by the referee.

References

1. P. Naur ed.	"Revised Report on the Algorithmic Language ALGOL 60", Num. Math. 4(1963), p. 420-453, and Comm. A.C.M., vol. 6 (1963), p. 1-17.
2. B.W. Arden, B.A. Galler, and R.M. Graham	"Michigan Algorithm Decoder", University of Michigan Press, Ann Arbor, Michigan, 1965.
3.	"Fortran 63/General Information Manual, Control Data Corporation Publication No. 514, August 1962.
4. B.A. Galler and A.J. Perlis	"Compiling Matrix Operations", Comm. A.C.M., vol. 5 (1962), p. 590-594.
5. J. McCarthy, et al.	"LISP 1.5 Programmers Manual", MIT, Cambridge, Mass., 1962.

Appendix A. Processing the <new syntax list>

A. If a <new syntax element> other than a <set definition> contains one or more <operator set name>s or <context type set name>s, this <new syntax element> is replaced by the collection of <new syntax element>s obtained by replacing each occurrence by a representative from its set, chosen in all possible ways. Any <new syntax element> so obtained may itself lead to a collection if it contains set names. (It is assumed that sets are defined so that this process terminates.) A set name may appear to the right of the := in a <context definition> only if it appears exactly once in the <context>[1]; then the same set representative used in the <context> is also used to the right of the :=.

B. If a <primitive representation> Q occurs, then:
(i) In the <representation> of Q replace its <actual type> γ by the <representation> of γ as given in its own <primitive representation>. If both of the <representation>s of γ and Q have <bound pair list>s, then append that of γ to that of Q and delete one occurrence of array.
(ii) Reapply (i) to the resultant <primitive representation> until γ becomes real, integer, or Boolean; i.e., one of the ALGOL C <type>s.
(iii) Enter the <new type> and its expanded <representation> into the type table, replacing if necessary any previous entry for that <new type>.

C. If a <new operator declaration> occurs, then:
(i) If its <new operator> already labels a line in the context table, delete the original occurrence of the <new operator> there, and delete from the context table all contexts in which it occurs. (It is assumed that the <new operator> and the <current operator> are different.) Then the <new operator declaration> is treated as follows:
(ii) If its <new operator> does not label a line in the table, and if the <new operator> is declared to be (<,=,>) the <current operator>, then (label with <new operator> a new, adjoin <new operator> as a label to the, label with <new operator> a new) line in the context table (immediately below, which is, immediately above) the line labelled by the <current operator>.

1) This restriction could be relaxed by using well-known naming techniques.

D. If a <context definition> Q containing a <string> occurs with <context> P, then:

(i) If the <string> in Q contains matching boldfaced parentheses, they must occur in the context <result type>**(**<open string>**)**, and the <open string> must contain an assignment to **r** which has integer subscripts $j_1, j_2, \ldots, j_p$, $(p > 0)$. From this context a new <context definition> is created of the form:

(P)[**integer** j_1][**integer** j_2] ... [**integer** j_p] := <result type> '**b** <open string> **e**'. Furthermore, in Q itself this context is modified so that

a) the <result type> is deleted, and

b) **(** and **)** are replaced by **b** and **e**, respectively.

Now for each context defintion use the first one of the following rules which applies, even if there is no <string>.

(ii) If there is already an entry in the context table whose <context> is P, replace that entry by Q.

(iii) Represent P as a tree as determined by the syntax of <context> or <assignment context>. This is clearly unique. Enter Q on the line labelled by the operator attached to the root node of this tree (or the line labelled "no operator", if there is none). On that line enter Q anywhere before any existing entry whose <context> or <assignment context> is a sub-tree of the tree for P.

Appendix B. Legal Expressions in ALGOL D.

Each entry in the type table is an <actual type>. Let there be N entries: <actual type>$_1$,..., <actual type>$_N$. Then, for this ALGOL D: <type> ::= <actual type>$_1$|<actual type>$_2$|...|<actual type>$_N$. Furthermore, these are the only types which may appear in the <declaration list> for this block, if there is one.

The set of "legal" expressions in an ALGOL D may be defined as follows:

We shall construct two sequences of sets $\{G_i\}$ and $\{E_i\}$. The set G_i will contain those <context definition>s whose <string>s are parsable by the <context>s available in G_{i-1}. The set E_i will contain those expressions and <assignment statement>s which can be generated by the <context>s available in G_i. Let S be the set of <context definition>s within the context table (including assignment contexts). Let G_0 be that subset of S which has no <string>s or whose <string>s are legal ALGOL C expressions and/or <assignment statement>s.

We now construct from each set G_i, $i \geq 0$, the set E_i and the set G_{i+1}. A language ALGOL D_i is then obtained by using the type tables and the <arithmetic expression>s, <Boolean expression>s, and <assignment statement>s of E_i, together with the common syntax of all ALGOL D's.

Select a <type> from the type table. Select any <context definition> Q in G_i which has the form:

P := γ<string>

where P is a <context> and γ is the selected <result type>. If Q is from the expression context table, replace each of the <context typed identifier>s β in P by either (a) an <identifier> of that <type>; or (b) a <context> from a <context definition> Q' in G_i whose <result type> is the <type> of β, and which comes from the expression context table. The <context> is to be enclosed in parentheses if it comes from a lower line in the table than Q, or if it comes from the same line as Q and β is the right operand of a binary <context>, or if β is preceded by <u>then</u> and Q' has an <u>if</u>-<u>then</u>-<u>else</u> context. If P is of the form <actual type>[program] [[bound pair list]], then any ALGOL D_i <program> and <bound pair list> may be substituted into P.

If Q is from the assignment context table, its left <context typed identifier> is to be replaced by (a) an <identifier> of the same <type>, or (b) the variable <u>r</u>, either possibly subscripted. The right <context typed identifier> is to be replaced by (a) another <context> from the assignment context table, or (b) a <context> from the expression context table.

Succesive replacements continue until no typed identifiers remain in the expansion of P, at which point the result is put in E_i. The set of all elements in E_i is obtained by applying the above rule to all <type>s, declared variables, and Q and Q' in G_i in all possible ways.

Suppose now that a subset G_i of S has been constructed, along with its expressions and/or <assignment statement>s E_i. The set G_{i+1} is obtained as the union of G_i with those <context>s T_i from S - G_i whose <string>s are in E_i. Let N be the smallest integer ≥ 0 such that T_N is empty. Then E_N is the set of "legal" ALGOL D expressions and/or <assignment statement>s. As an example, the $\{G_i\}$ (not the $\{E_i\}$!) are given for the matrix definition set of section 5:

G_0 = {all <context definition>s of Figure 1} $\cup$ {(6), (12), (19)}

$G_1 = G_0 \cup \{(8), (9), (10), (13), (20)\}$

$G_2 = G_1 \cup \{(1a)$ 1)$\}$

$G_3 = G_2 \cup \{(1b), (2), (3), (4), (11)\}$

$G_4 = G_3 \cup \{(5), (7), (14), (16), (18)\}$

$G_5 = G_4 \cup \{(15), (17)\}$

In this case, S - T_5 is empty, so there are no unusable <context>s.

Appendix C. The Replacement Rule.

To each sub-tree S we associate a unique sub-tree max(S) with the property of being the maximal sub-tree containing S such that it and all of its sub-trees represent ALGOL D expressions. If S is the tree of a conditional expression whose then and else branches are T and E, resp., then max(T) = max(E) = max(S). Define b(S) as the tree obtained by replacing S in max(S) by a special terminal character Σ.

1) (1a) refers to the <context definition> generated from (1) because of the presence of (and) in the <string>. (1b) refers to the <context definition> resulting from (1) when (and) are replaced by b and e, respectively.

A <context type> α and an <actual type> β are said to agree if (i) they are the same <actual type>, or (ii) they have the same <boldface symbol>, they have the same number of parameters, and the parameters of α which are <actual type parameter>s are identical as <string>s to the corresponding parameters in β.

A sub-tree T_1 is said to match a context tree T_2 if and only if: (i) The <type>s of their root nodes agree, and (ii) either the root of T_2 is a terminal, or both root nodes are labelled with the same operator and have the same number of branches and corresponding branches represent matching sub-trees. Throughout the entire test for a match, all occurrences of a specific <formal type parameter> must match the same <actual type parameter>. In finding a matching context for a sub-tree, the scan of the context table is from top to bottom, and the first context reached whose context tree matches the sub-tree is chosen.

The replacement rule now proceeds in the following steps:

(1) Let S be any maximal <program expression>; i.e., a <program expression> not contained in any other, such that S ≠ max(S). If there are none, carry out step (2). Let R be the sub-tree representing the <assignment statement> in S whose left side contains the result $\underline{r}$ of S, and whose right side is an expression ψ. Then do the following steps in turn: [1)]

(i) Replace R by b(S).

(ii) The Σ introduced by b(S) is replaced by ψ.

Repeat step (1) as many times as possible.

(2) If there is a sub-tree of the form $\underline{r}$:= ψ or $\underline{r}$[<subscript list>] := ψ, replace this sub-tree by the tree for ψ alone.

1) If S is a sub-tree of either the T or E branches of a conditional expression, say T, the other branch E is converted to a <program expression>, if necessary, by replacing it with the tree for the <program expression>: $\underline{b}$ $\underline{r}$:= E $\underline{e}$. In any case, steps (i) and (ii) are carried out on the two branches simultaneously.

(3) From the tree now under consideration, we select a sub-tree S to be replaced, as follows:

(i) If there is a maximal sub-tree T whose root node is the subscription operator, then (a) if there is a matching <context> with a <string> for T, then S is taken to be T. Otherwise, (b) we apply step (3) to T. Otherwise,

(ii) select any maximal sub-tree whose matching <context definition> contains a <string>. (If the replacement of some sub-tree causes no change in the tree (after any occurrence of r has been eliminated), that sub-tree is not to be selected for further replacement.)

(iii) If, by an application of (i)(b) just above, we have restricted consideration to a sub-tree T for which no matching <context> can be found, then (ii) is applied to the original tree.

(4) Let P be the matching context. Remove the outer string quotes from the <string> in P's <context definition>. Each occurrence of an identifier with * is assigned its ALGOL C <type>, and then all *'s are deleted. Any <declaration> involving a <new type> is replaced by the appropriate ALGOL C <declaration> obtained via its <primitive representation> in the type table.[1)] If the <open string> for a <context> P has the form of an <enumerated expression>, let its list of expressions (with <bound pair list>) be G. If, after appending the <bound pair list> of the <type> of G (as given through a <primitive representation>) to the <bound pair list> of G, the resultant <bound pair list> has only <integer> bounds, say $i_1 : j_1, \ldots, i_n : j_n$; and furthermore if the selected tree S occurs subscripted by an <integer> k, then

(i) extract the sub-sequence of <arithmetic expression>s or <Boolean expression>s in G from position $(k - 1) \times \prod_{m=2}^{n} (j_m - i_m + 1) + 1$ to position $k \times \prod_{m=2}^{n} (j_m - i_m + 1)$, inclusive.

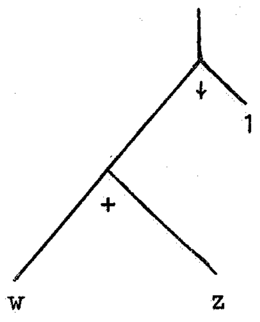
(ii) Delete the leading <bound pair> and append the resulting <bound pair list> to the sub-sequence. Call this G.

(iii) Replace S and its subscript k by S alone in the tree.

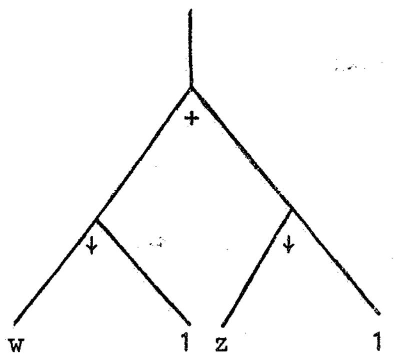
(iv) Repeat the process commencing with furthermore until it fails. Select G as the <open string> to be parsed.

1) This includes implied <declaration>s, such as complex(a,b), which becomes (a,b)[1:2].

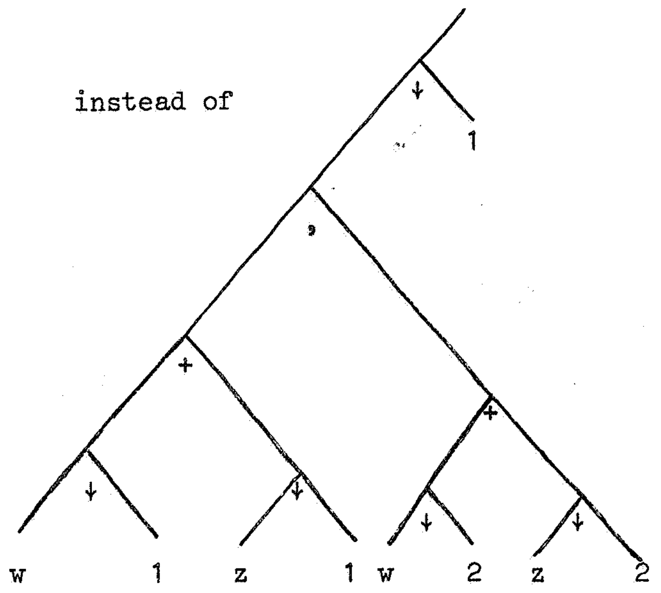
Example of step (4): For complex w,z represented as real array [1:2],



becomes



instead of



since w + z would normally be (w[1] + z[1], w[2] + z[2]). (See section 6).

(5) Parse the text as selected in (4) into a tree $\tilde{S}$ using the current ALGOL D syntax. If the <open string> is a <program expression>, only that part representing <program> is parsed. There is an obvious correspondence between the <context typed identifier>s of P and the sub-trees of S, which arises directly out of the matching process. (All identifiers declared in $\tilde{S}$ must be renamed so that no identifier in $\tilde{S}$ has the same name as an identifier occurring in max(S).) Using this correspondence, as well as the formal-actual correspondence between <actual type>s and <context type>s, we replace the occurrence of any <context typed identifier> in $\tilde{S}$ by its corresponding sub-tree from S. The resulting tree is substituted for S in the original tree.

(6) The replacement rule is applied to the resultant tree until no sub-trees remain whose matching contexts have <string>s in their <context definition>s; i.e., no sub-tree S can be selected. The resulting tree is the tree representation of an ALGOL C <arithmetic expression>, <Boolean expression>, or <assignment statement>.

The matrix example in section 5 shows in detail how this rule is applied.