Handling the growth by definition of mechanical languages*

by SAUL GORN
The Moore School of Electrical Engineering
University of Pennsylvania
Philadelphia, Pennsylvania

INTRODUCTION

It has for many years been my opinion that programming languages in particular, and mechanical languages in general, exhibit many phenomena generally thought to be characteristic of natural languages.

The concept of mechanical language, in my mind, includes all sorts of notational and signalling systems, whether one-dimensional or multi-dimensional, sequential or simultaneous acting, continuous or discrete, descriptive or prescriptive; it is also supposed to subsume a vast variety of recording media subjected to a vast variety of sensing devices (including biological as well as physical), with a vast variety of permanence characteristics, reaction times, storage arrangements in space-time, retrieval devices, coupling, coding, and translational processors.

Thus musical and choreographic notations are mechanical languages, as are structural formulae notations in chemistry, nucleotide chain diagram schematics in genetics, parsing tree and other phrase marking systems in linguistics, and mechanical drawing, wiring diagram, block diagram, production control charts and organization chart systems in industry. So are cataloguing, inventory and accounting systems, whether used in industry, commerce, war, government, or religion. I would even include notations and arrangement systems for monuments and their contents.

*The Research behind this paper was made possible by the joint support of the Army Research Office (DA-31-124-ARO(D)-98), the National Science Foundation (NSF-GP-5661), and the Public Health Service Research Grant (5 RO1 GM-13,494-02) from the National Institute of General Medical Sciences. It was presented in a preliminary report at the Systems and Computer Science Conference, University of Western Ontario, London, Ontario, September 10 and 11, 1965, and the proofs of the theorems and the presentation of the processors discussed here appear in the Proceedings of that conference, under the title "Explicit Definitions and Linguistic Dominoes."

The time scales and space scales therefore vary from milli-microseconds to millenia, from microns to light years; the adaptation of devices (from artifacts to specially educated humans) working in different time scales, different transfer rates, different reaction times, with different signalling languages, and the logic of their system assembly, even apart from adaptability in time, is part of the problem of the study of mechanical languages.

The more successful a mechanical language (system) is in its ease of specification for many different types of areas, and in its adaptability among different kinds and levels of artifacts and humans (i.e. processors), the denser will be its usage for communication among men, among machines, and between men and machines. It will therefore tend to 'grow,' to reach for 'universality' as a 'common language.'

An example of such a trend appears if one looks at the history of mathematical notation. As long as very few people operated with numbers (as abstracted from things) the notations for numbers were primitive, but stable; so stable that they lasted for millenia. Only a specialized historian would have been able to perceive their process of change. The urban revolution increased their usage, and therefore their rate of change; their stability was reduced from millenia, to centuries, to decades. The arithmetic notation expanded into the arabic number representation language, but the users could still deceive themselves into crediting the changes to ingenious individuals rather then recognizing them as part of a social phenomenon. This continued even into the development of algebraic notation and its growth into a notation for analytic expression. But when the notation grew to include symbolic logic and operations with sets, and the applications spread through all the sciences and professions, the contributions to the structure of the mechan-

ical language became too numerous to be anything but anonymous, and therefore social. Furthermore, the logical operations became devices by which the language itself could be examined; the human activity became self-conscious, and the language became self-referencing and capable of operating, in the same fashion, many meta-syntactic levels (I have called such, 'languages with unstratified control'). The tendency is now for this mechanical language with its descriptive precision and its ability to analyze so much one-dimensional symbol manipulation to fuse with programming languages with their prescriptive precision and their ability to synthesize so much of the processing of that same area of one-dimensional, and even multi-dimensional symbol manipulation.

Thus, different mechanical languages may therefore undergo *fusion* as time goes on.

But a particularly successful language will also tend to reflect the behavior of the human community by undergoing *fission* by splitting into dialects, even separate languages. This will surely happen if the using community splits into groups which seldom intercommunicate, and each of which restricts itself to a favorite subsystem of expressions; this will first be a jargon, then a dialect, then a separate language. In programming, these jargons are favorite systems of macro-instructions, introduced by the very macro-definition facility which made the base language so flexible and 'universal'.

In any event, we are now more and more concerned with growing languages, extended machines, growing machines, self-expandable languages, user-expandable languages and the like. (See Cheatham,[3] Carr,[2] Otrand[15] and Galler & Perlis.[7])

How are we to control the growth?

If it is completely uncontrolled and is allowed to 'just grow', such a man-machine, multi-lingual and multi-levelled processor and language system can die either by choking on the sheer bulk of what it swallows without rejecting or reforming, or it can die by committing mitosis, giving birth to a number of independent such subsystems with practically no intercommunication among them.

However, a natural control of the growth of such a system will have to be the result of continuing and never-ending work by hundreds of people. Each one will have to be able to refocus his attention from the grandiose to the picayune when necessary, without losing sight of the big picture. It is impossible to say whether it is the big wheel or the tiny wheel in such a system which is really connected to the ground. The same remark applies to the people in it.

Therefore, this paper will attempt only to suggest how such growth may be understood and controlled, while restricting itself to a very special category of mechanical languages with a very strong structure, and which grows by a very limited means called 'explicit definition'.

Categories and forms

Let me illustrate the category of languages to which we are limiting our discussion by tracing one simple algebraic expression through a number of its languages. We do this not merely to show how general our considerations are, by illustrating how big the category might be; we do it because the category is a fusion of many languages, rather than only a set, and those concepts and devices are to receive special attention which have a correspondent in each language of the category.

Consider, then, how the simple algebraic expression 'ab + b(c+d)', belonging to a one-dimensional language with which most of us are very familiar, transforms into seven other expressions. (See fig. 1).

It is the comparison among corresponding expressions in different languages of the category which prompts us to make the following decisions:

1. there is an alphabet of 'objects' which includes symbols like a,b,c,d,+, and something called 'times', even though examples 1 and 3 have no separate symbol for it, and even though examples 2, 4 and 8 represent it by ×, where 5, 6 and 7 use *.
2. there is an alphabet of 'context signals', much more concerned with 'how' the expressions are to be read than with 'what' they contain, and consequently more variable among the languages of the category. Examples are parentheses, punctuation marks, connecting lines, and tabulation lines. These are 'control characters'.
3. there is an 'addressing system'; only the object characters have addresses; control characters only help us to find these 'addresses', but are not themselves assigned addresses. The corresponding objects in the corresponding expressions have the 'same' addresses.

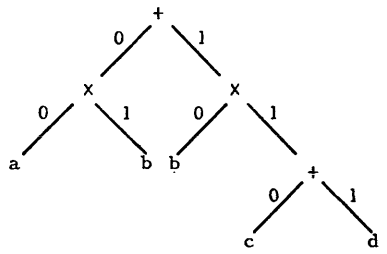
We therefore say that we really have a category of 'language functions', each applicable to many '(object) alphabets' to form the languages in the category. The control characters are really the marks belonging to the language functions, and often are replaced by, or replace, the scanning process.

Thus we might call the 'common' alphabet used here $\mathscr{A}$ = {a,b,c,d,+....}, the language functions $\mathscr{F}T_{\mathscr{D}_d}$, $\mathscr{F}$, $\mathscr{P}$, and $\mathscr{M}$ to correspond to expressions 2,4,5,6,7, and 8. Thus the tree in 2 belongs to the language $\mathscr{T}_{\mathscr{A}}$. If symbols from logic form an object alphabet $\mathscr{B}$ = {∨, ∧, ~, ⊃, ≡, p, q, r....}, then ⊃p⊃qp might be an expression from the language $\mathscr{P}_{\mathscr{B}}$

1.- ab + b(c + d)

left-to-right;precedence to multiplication

2.-



Tree form

3.- (d + cb) + (ba)

right-to-left; precedence to +

4.-

address	character
*	+
0	x
1	x
00	a
01	b
10	b
11	+
110	c
111	d

Function tabulation

5.- + * * a b b + c d (Depth)

6.- +(*(a, b), *(b, +(c, d))) (Functional)

7.- +*ab*b+cd (Prefix)

8.- *, +; 0, x; 1, x; 00, a; 01, b; 10, b; 11, +; 110, c; 111, d. (Address mapping)

Figure 1—Corresponding expressions in eight languages of a category

The two-dimensional language functions seem to have some advantages in human communication (e.g. 𝒹 and T in 2 and 4); those, like 4 and 8, which are explicit about the mapping from addresses to characters, seem to have some advantages when we seek to establish a mathematical theory; those like $\mathscr{P}_d$, $\mathscr{P}$, and $\mathscr{P}$ in 5, 6, and 7 seem to have some advantages when it comes to designing efficient processors. For example $\mathscr{P}$ has an especially efficient 'scope analyzer' (see Gorn[11]), where $\mathscr{P}_d$ is more efficient for 'depth analysis'.

The first restriction we make is therefore to such categories that corresponding expressions in the different languages all have the same 'parsing tree', and are therefore uniquely determined by systems of 'addresses' called 'tree-domains' and the corresponding objects to be found there. Thus, whatever else we may find in the category, we can always depend on the representations in languages like those illustrated in 2, 4 and 7 of fig. 1.

The second restriction that we make is that the languages in the category be 'saturated' in the sense that 'any validly parsed expression' actually belongs to the system. We express this by demanding, first, that the system of language functions be applicable to any and all 'simply-stratified alphabets'. This means that each character have a kind of 'order', called its 'stratification number', much like the number of independent variables possessed by a numerical function. The second demand made by the restriction is that any expression obtained from a tree by putting characters of stratification n at nodes of ramification n (i.e. of 'exiting' order n) is a valid expression in the system.

This second restriction means that the prefix language form (type 7) is to be 'complete'. It is this language function which we have called the 'complete prefix language function'. (In Gorn[12] we show that it is also saturated in that the addition of one more word to its extent would cause it to cease being 'uniquely deconcatenable'. See also[9].) Naturally all the other languages of the category have the corresponding kind of completeness.

The only reason we make such a point of this second restriction (It is unduly harsh; in ordinary algebraic notation, for example, we do not demand that the expression '(a=b) + (c=d)' be meaningful.) is that we need it as a guide in our third restriction. We will allow our category to 'grow' by 'explicitly defining'

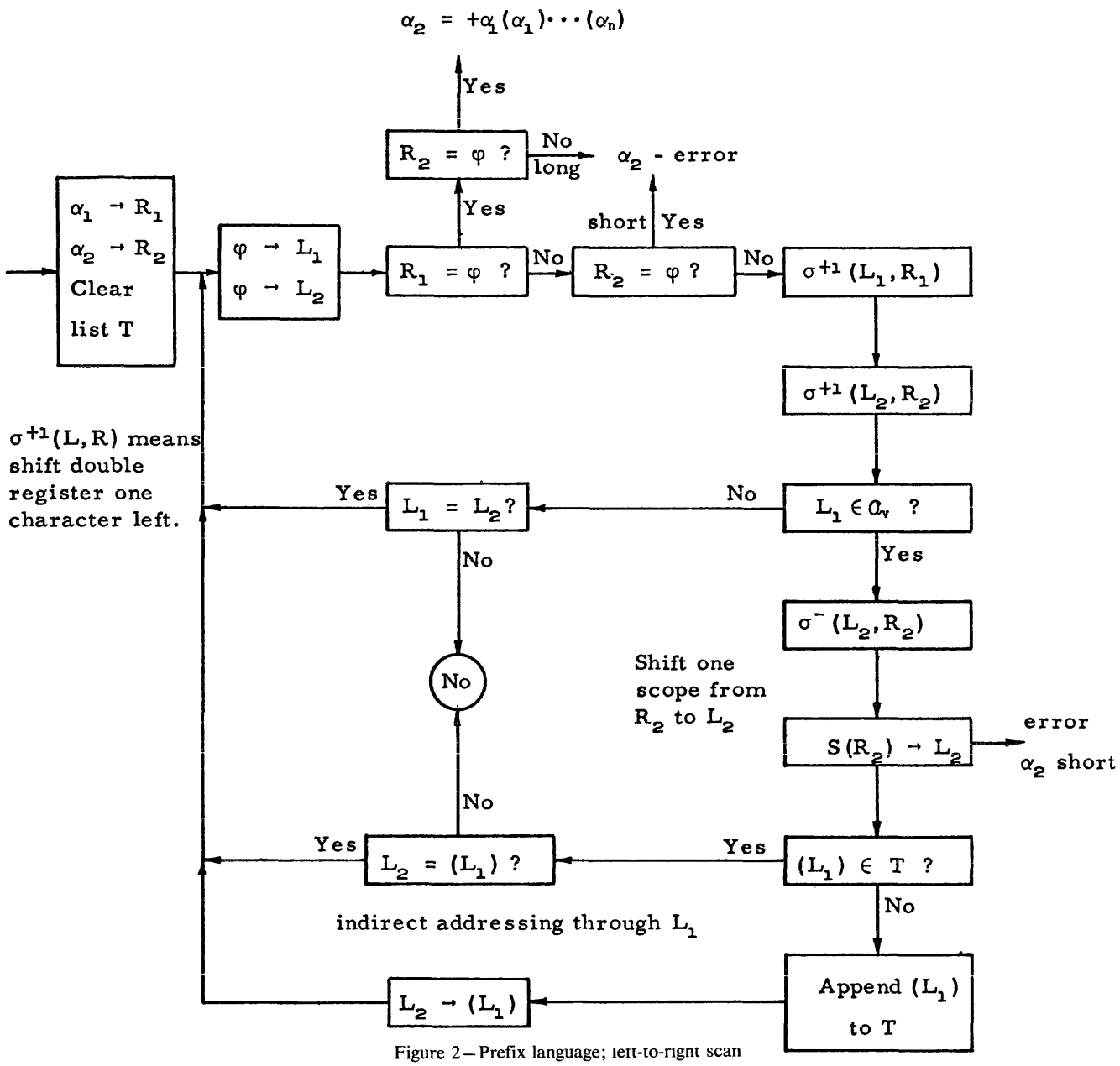


Figure 2 – Prefix language; left-to-right scan

new characters in the object alphabet. Our third restriction limits the form of these explicit definitions by demanding that the extended language category be also complete. Let us see what this requires.

Suppose 'd' is the newly defined object character. It will have to possess, by our first restriction, a fixed stratification number, say n. If, then, α_1, α_2, ..., α_n designate any good expressions of $\mathscr{P}\mathscr{A}$ by our second restriction $\mathscr{P}\mathscr{A}$ must also contain a good expression of the form '$d\alpha_1\alpha_2...\alpha_n$'. Thus, the explicit definition must have the 'form' $\beta_1 \underset{df}{=} \beta_2$, where the 'definiendum' β_1 must have the 'simple form' $d\alpha_1\alpha_2...\alpha_n$, and the 'definiens' β_2 must not contain the character 'd' explicitly (no recursion), and must be an obvious syntactic function of the syntactic variables $\alpha_1,...,\alpha_n$ for all languages of the category. By the 'principle of syntactic invariance' (see Gorn[10]), the α_i must be variables representing any good expressions throughout the category, because they are appropriate 'scopes' and the completeness of the category means that all good 'scopes' are 'good words' and vice versa.

We must therefore be able to define and recognize forms as well as expressions. This is easily done without too much work by introducing an alphabet of 'scope variables': $\mathscr{A}_v = \{\alpha_1\ \alpha_2,...\}$ having no characters in common with α, and each character of which has stratification zero. If $\mathscr{A}' = \mathscr{A} \cup \mathscr{A}_v$, then $\mathscr{P}\mathscr{A}$ is a form language, representing a whole category of form languages in which 'tree forms' can be generated, recognized, and compared.

For example, we have the following definition throughout the category of languages of tree patterns:

Definition: The form β_1 is said to be 'of the form β_0', and β_0 is called 'an initial pattern of β_1' (we write $\beta_0 \leqslant \beta_1$), if there is a complete independent set of scopes $\sigma_1, \ldots, \sigma_n$ of β_1, where n is the number of end-points of β_0, such that, if $\alpha_i = \alpha_j$ in β_0, then $\sigma_i = \sigma_j$, and such that $\beta_1 = \beta_0\ (\sigma \rightarrow \alpha_1, \ldots, \sigma_n \rightarrow \alpha_n)$. We can also write: $\beta_1 = +\beta_0\sigma_1 \ldots \sigma_n$.

If we think of the variables in $\mathscr{A}_V$ as also referring to some unspecified addressing system for the control of storage of expressions, then this definition is effective; figure 2 presents the design of the processor called a 'tree pattern recognizer'.

A number of other notations might be mentioned as being convenient both in the theory and in the construction of processors:

a. 'Caβ_1' to mean 'the character at (tree) address a in tree (form) β_1'.
b. 'Saβ_1' to mean 'the scope at (tree) address a in tree (form) β_1'.
c. 'Dβ_1' to mean 'the domain of (tree) addresses in tree (form) β_1'.
d. If a_2 and a_3 are tree addresses, then $a_1 = a_2 \cdot a_3$ can be read as 'the tree address of relative address a_3 with respect to address a_2', and a_2 is a 'tree-predecessor' of a_1 : $a_2 \leqslant a_1$. Similarly, if A_1 and A_2 are two sets of tree-addresses, $A_1 \cdot A_2$ will be the set of tree-addresses of relative address some A_2 with respect to some A_1; it is to be interpreted as the null set Λ if either A_1 or A_2 is null.
e. We can also permit the scope-operator 'S' to refer to tree-domains as well as to trees. Thus, if D is a tree domain and $a_1 \in D$, then there is a tree domain D_1 such that $Sa_1 D = a_1 \cdot D_1$; also, $Sa_1 \cdot a_2\, D = a_1 \cdot Sa_2 D_1$. This notation sets up a primitive address computation facility (in fact a semi-group with unit '*' meaning root-address) for tracing and identifying the effects in expressions of growth by definition in the category.
f. 'an occurrence of the character $c \in A$ in β' will mean 'an address a such that $c = C\ a\ \beta'$, and the set of such occurrences can be written '$C^{-1}\ c\ \beta'$. Thus '$a \in C^{-1}\ c\ \beta'$ means 'a is an occurrence of c in β'.

It is now possible to recognize internal patterns as well as initial patterns in trees: β_0 is an internal pattern of β_2 occurring at the tree address a if $\beta_0 \leqslant S\ a\ \beta_2$; in this case we also write $a \in C^{-1}\beta_0\beta_2$, and speak of 'an occurrence of the form β_0 within β_2'. If β_0 is deeper than zero, i.e. is more than just one object character at the root, then we say that the form β_1 'dominates' the form β_2 with the intermediary β_0 whenever β_0 s a scope of β_1 and the initial pattern of β_2, but β_0, β_1 and β_2 are not all the same; if we replace that occurrence of β_0 in β_1 by β_2, the result $\beta_1(\beta_2 \overset{a}{\rightarrow} \beta_0)$ if $a \in C^{-1}\beta_0\beta_1$, is a '$\beta_1\beta_2$-domino'.

We now have the apparatus by which we can specify an 'explicit definition', and trace its effect by means of a primitive kind of computation with tree addresses.

EXPLICIT DEFINITIONS AND TREE MAPS

We have now fixed upon an appropriate definition of 'explicit definition' for our categories of mechanical languages. It is one in which:

a. The definiendum is a simple form, i.e. either of depth zero (hence only the single new character), or of depth one with only distinct variables at the end-points,

b. the definiens is an arbitrary form φ_d, at least as deep as the definiendum (hence $\neq \alpha$), not containing d explicitly, but each variable of which does occur again in the definiendum.

Ordinarily explicit definitions are introduced in order to say in less space something one will want to repeat fairly often. In other words, a very common motive for an explicit definition is the desire to have a large family of abbreviations; each definition provides an infinite set of abbreviations, all of the same form. (Thus each definition is like an 'optional linguistic transformation'.) Because the size of a tree can be measured both in depth and in breadth, two of the simplest types of explicit definition are the pure depth reducer, and the pure breadth reducer:

An explicit definition, $d\alpha_1 \ldots \alpha_n \underset{df}{=} \varphi_d$, is a 'pure depth reducer' if those end-points of φ_d which are variable are, reading from left to right, in $\mathfrak{p}$ or $\mathfrak{z}$ form, exactly $\alpha_1, \alpha_2, \ldots, \alpha_n$, and at least one variable has depth greater than one. If $n = 0$, so that $d \underset{df}{=} \varphi_d$ where no end-point of φ_d is variable, we do not call it a depth reducer, no matter how deep φ_d may be. An example of a pure depth reducer is $d\alpha_1\alpha_2 \underset{df}{=} c\alpha_1 c c_0 \alpha_2$.

An explicit definition is a 'pure breadth reducer' if the depth of every variable in φ_d is one, and those end-points of φ_d which are variable are, reading from left to right, in $\mathfrak{p}$ or $\mathfrak{z}$ form and ignoring repetitions, exactly $\alpha_1, \alpha_1, \ldots, \alpha_n$, and at least one variable occurs more than once. The same interpretation is made if $n = 0$ as before; no matter how broad φ_d may be, it is not a breadth reducer. An example of a pure breadth reducer is $d\alpha_1\alpha_2 = c\alpha_1 c c_0 c_0 c_0 c_0 \alpha_2 \alpha_1$.

More generally, if φ_d is the definiens of an explicit definition, let $A_{di} = C^{-1}\alpha_i\varphi_d$ (the occurrences, possibly

$$d\,\alpha_1\,\alpha_2 \underset{df}{=\!=} c\,\alpha_2\,c\,\alpha_1\,\alpha_2$$

Definiendum

address	character
*	d
0	α_1
1	α_2

Definiens

address	character
*	c
0	α_2
1	c
10	α_1
11	α_2

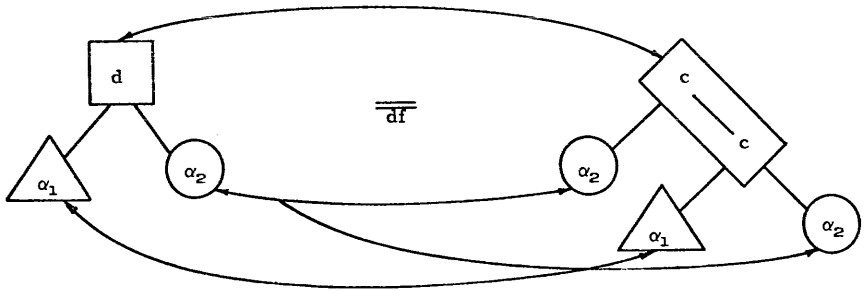


Figure 3 – An explicit definition

null, of α_i in φ_d), and $A_{do}=D\varphi_d - \bigcup_{i=1}^{n} A_{di}$ (the 'interior' of φ_d); let m_i be the number of addresses in A_{di}, which we could call the 'breadth' or 'multiplicity' of α_i, and let d_i be the maximum depth of α_i in A_{di}. Then the definition does some breadth reduction if $m_i>1$ for at least one i, and it does some depth reduction if $d_i > 1$ for at least one i.

If $m_i = 0$ for some $i \leq n$, the stratification of d, then the definition is allowing arbitrary scopes to be introduced, and we say that the definition is a 'scope-expander'. An example of a 'pure scope-expander', i.e. one in which neither depth or breadth reduction occurs, is $d\alpha_1\alpha_2 \underset{df}{=} c\alpha_2$, or $d\alpha_1\alpha_2 \underset{df}{=} cc_0$.

Finally, an explicit definition may be introduced neither to expand scopes, nor to reduce depths or breadths, but simply to permute scopes. In a pure permuter' $m_i = 1$ and $d_i = 1$ for $i = 1, \cdots, n$. Thus $d\alpha_1\alpha_2 \underset{df}{=} c\alpha_2 cc_0c_0c_0\alpha_1$ is a pure permuter.

Most definitions we might envision would be mixtures of these types, and for each such mixture we might imagine each type of effect to be introduced by a separate 'ideal definition', thereby 'factoring' the definition into a sequence of pure types. For example, the explicit definition $d\alpha_1\alpha_2 \underset{df}{=} c\alpha_2 c\alpha_1\alpha_2$ might be envisioned as the result of

1. a pure depth reducer $d_1\alpha_1\alpha_2\alpha_3 \underset{df}{=} c\alpha_1 c\alpha_2\alpha_3$,
2. a pure breadth reducer $d_2\alpha_1\alpha_2 \underset{df}{=} d_1\alpha_1\alpha_2\alpha_1$,
3. a pure permuter $d\alpha_1\alpha_2 \underset{df}{=} d_2\alpha_2\alpha_1$.

Figure 3 illustrates this definition.

We can now be precise about the meaning of such expressions as:

a. The elimination of an occurrence of d in a form β_2: if $a \in C^{-1}d\beta_2$, and the scopes $\sigma_i = Sa{\cdot}i\beta_2$, and ψ_1 is the form obtained from φ_d by replacing each α_i by σ_i (in other words, ψ_1 is the application of the definiens form to the scopes of the occurrence of d in β_2), i.e. $\psi_1 = \varphi_d(\sigma_1 \rightarrow \alpha_1, \ldots, \sigma_n \rightarrow \alpha_n)$, then the replacement of the scope at a of β_2 by ψ_1 yields an expression $\beta_1 = \beta_2(\psi_1 \rightarrow Sa)$ which is said to be obtained from β_2 by the elimination of d at a; we also write

$$\beta_2 \overset{E_d}{\underset{a}{\rightarrow}} \beta_1, \text{ and } \beta_1 = E_d\, a\beta_2,$$

and call β_1 an immediate d-descendent of β_2, thereby beginning the definition of a partial ordering relation among expressions: $\beta_2 \underset{d}{\geqslant} \beta_1$.

b. The introduction of d in a form β_1 at an occurrence of φ_d: if $a \in C^{-1}\varphi_d\beta_1$, so that $\varphi_d \leq Sa\beta_1$, i.e. φ_d

is an internal pattern in β_1, then there is a complete independent set of scopes $\sigma_1,\ldots, \sigma_n$, occurring with the repetitions corresponding to the α_i in φ_d, at addresses in β_1 whose relative addresses with respect to a equal the address of the end-points α_i in φ_d; let β_0 will say that β_2 is obtained from β_1 by introduction of d at a; we also write $\beta_1 \xrightarrow[a]{I_d} \beta_2$ and $\beta_2 = I_d a \beta_1$; β_2 is an immediate d-predecessor of β_1 and $\beta_1 \underset{d}{\leqslant} \beta_2$.

Clearly $I_d a$ and $E_d a$ are inverse operators.

c. In general, $\beta_2 \underset{d}{\geqslant} \beta_1$ and $\beta_1 \underset{d}{\leqslant} \beta_2$, if either $\beta_2 = \beta_1$ or there is a chain of d-eliminations $\beta_2 = \beta_{0\,1} \xrightarrow[a_1]{E_d} \beta_{02} \xrightarrow[a_2]{E_d} \cdots \beta_{0n-1} \xrightarrow[a_{n-1}]{E_d} \beta_{0n} = \beta_1$.

In this case we say that β_2 is a d-predecessor of β_1, β_1 is a d-descendent of β_2, and β_1 is obtained from β_2 by d-reduction. Figure 4 presents the design of a recognizer of $\beta_2 \underset{d}{\geqslant} \beta_1$.

d. β_2 is d-equivalent to β_1, and we write $\beta_2 \underset{d}{\equiv} \beta_1$ if the chain of operations, as in c, may contain any combination of d-introductions and d-eliminations.

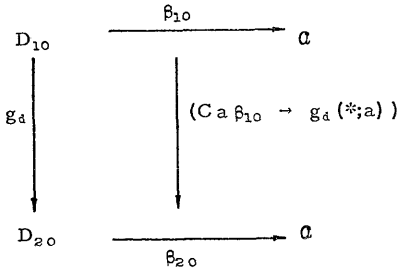
It is easy to see that $\underset{d}{\leqslant}$ is a partial-ordering, and $\underset{d}{\equiv}$ is an equivalence relation. We will designate the set of all d-descendents of β_0 by $L(\beta_0)$, and the d-equivalence class of β_0 by $E(\beta_0)$.

If $\beta_2 \underset{d}{\leqslant} \beta_1$, there might be many elimination chains leading from β_2 to β_1; in spite of this, for each character occurrence in β_1, there is only one occurrence in β_2 'responsible for it'. This is shown by exhibiting a uniquely determined mapping from β_1 *into* β_2.

Figure 3 has already shown this in the critical case where $\beta_{10} = \varphi_d$ and $\beta_{20} = d\alpha_1 \ldots \alpha_n$, here $CA_{do}\,\beta_{10} \to d = C^*\beta_{20}$, and $CA_{di}\beta_{10} \to \alpha_i = Ci{-}1\beta_2$. This is the 'generating d-map', $g_d(*;a)$, which performs a linguistic transformation from β_{10} into β_{20}, where each of these expressions can also be looked upon as mappings, called trees, from tree-domains into the object alphabet; these linguistic transformations are therefore representable as composite mappings fulfilling a commutative condition:

$$(Ca\beta_{10} \to g_d(*;a)) \circ \beta_{10} = \beta_{20} \circ g_d(*;a),$$

a relation we might choose to represent by the symbolism $\beta_{10}\ E\ \beta_{20}\ g_d$, or even by the diagram extending the type used by algebraic topologists:



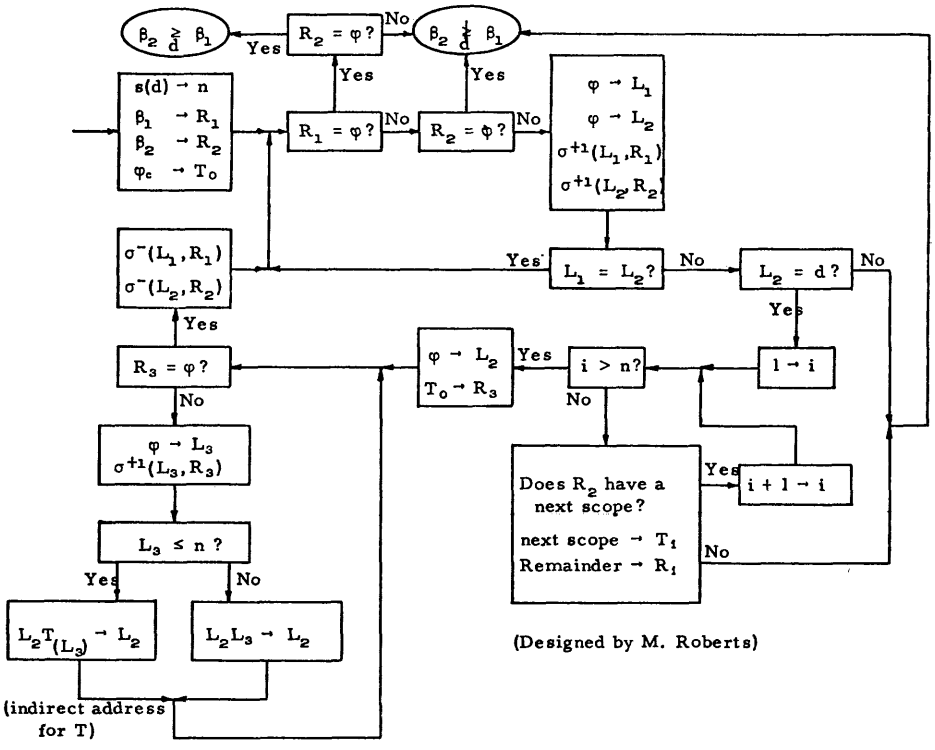


Figure 4 – Recognizer for the relation $\beta_2 \geqslant \beta_1$

We can extend such mappings from occurrences of d and φ_d in β_{20} and β_{10} alone by first defining the address map:

$$g_d(a_1;a) = \begin{cases} a \text{ if } a_1 \leq a \\ a_1 \text{ if } a \in a_1 \cdot A_{do} \\ a_1 \cdot i\text{-}1 \cdot a'' \text{ if } a \in a_1 \cdot A_{di} \cdot a'' \end{cases}$$

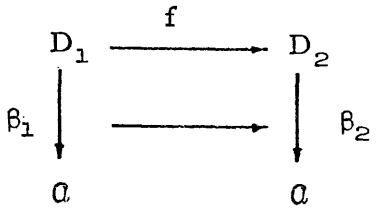
and then considering composites of such address maps, f, which we call d-maps.

The design of the recognizer of the relation $\beta_2 \underset{d}{\geq} \beta_1$, shown in figure 4, depends upon the:

Theorem: The following conditions are equivalent:

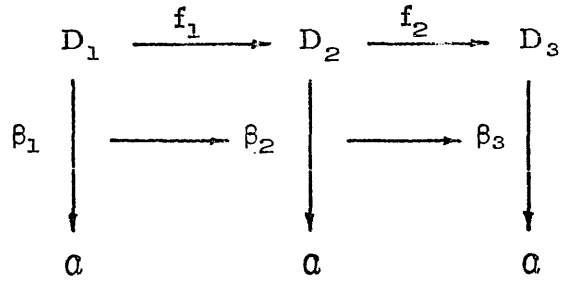
1. $\beta_1 \in L(\beta_2)$,
2. $\beta_2 \underset{d}{\geq} \beta_1$,
3. There is a d-map of β_1 into β_2,
4. There is a unique d-map of β_1 into β_2.

The generalized d-map can therefore be symbolized by $\beta_1 \overset{E}{\sim} \beta_2 \circ f$, or by the diagram:



and the composition of d-maps can be reflected in such notation as:

$\beta_1 \overset{E}{\sim} \beta_3 \circ (f_2 \circ f_1)$, or



If, furthermore, the explicit definition is not a scope expander, and $\beta_2 \underset{d}{\geq} \beta_1$, the uniquely determined mapping from β_1 into β_2 is a mapping 'onto'.

In any case, such mappings f are uniquely determinded by addresses only; i.e. by a computation which given any $a \in D_1 = D\beta_1$, yields $f(a) \in D_2 = D\beta_2$.

Actually, an even stronger statement can be made: these address maps are even independent of the particular trees. $E_d a$ is a functor applicable to any tree β_2 for which $Ca\beta_2 = d$, and it is possible to compute $g_d(a;b)$ for any address without knowing anything else about β_1 beyond the fact that $a \in C^{-1}\varphi_d\beta_1$. We can therefore talk, within the context of a fixed explicit definition, of 'the elimination functor <a>'.

Furthermore, $E_d a_1 E_d a_2$ will be applicable, as a functor, to an infinite number of expressions, if to any at all; first of all, if it is applicable to β, then $Ca_2\beta = d$; secondly we must have $d = Ca_1E_da_2\beta$. If we let:

$$A_{di} = \{a_{i1}, a_{i2}, \ldots, a_{imi}\},$$

where m_i is the multiplicity of α_i, such an occurrence of d at address a_1 is a priori impossible only if $g_d(a_2;a_1) \in A_{do}$; in other words it is possible only if $a_1 \geq a_2 \Rightarrow (\exists\, i,j)(a_1 \geq a_2 \cdot a_{ij})$. Let the relation $_1 \geq a_2$ among addresses mean just this, so that our condition reads: $a_1 \geq a_2 \Rightarrow a_1 \geq a_2$. This, then, is precisely the condition under which the 'elimination functor $<a_1,a_2>$' exists; it is applicable to any one of the infinite number of expressions β with an occurrence of d at a_2 and also at $g_d(a_2;a_1)$. We could similarly validate and identify the 'elimination functor $F = <a_1, \cdots, a_n> = E_d\, a_1 \cdots E_d a_n$' and restrict ourselves to a primitive address computation as in Figure 5.

$$d\alpha_1\alpha_2 \overset{df}{=} c\alpha_2 c\alpha_1\alpha_2$$

$$g_d(*;a) \begin{cases} <*>\{*,0\} = \{10\} \\ <*>\{*,1\} = \{0,11\} \end{cases}$$

$$\begin{aligned} <111,0,*>\{*,1,11\} &= <111,0>\{<*>\{*,1\} \cup <*>\{*,1\cdot 1\}\} \\ &= <111,0>\{\{0,11\} \cup \{0,11\}\cdot 1\} \\ &= <111,0>\{0,11,01,111\} \\ &= <111>\{<0>\{0,01\} \cup <0>\{0,11,111\}\} \\ &= <111>\{0\cdot <*>\{*,1\} \cup \{11,111\}\} \\ &= <111>\{0\cdot\{0,11\} \cup \{11,111\}\} \\ &= <111>\{00,011,11,111\} \\ &= <111>\{111,00,011,11\} \\ &= \{00,011,11\} \end{aligned}$$

Figure 5 – An address computation for d-elimination

The problem solved by the computation in figure 5 could be stated as follows: If we explicitly define d by means of $d\alpha_1\alpha_2 = c\alpha_2 c\alpha_1\alpha_2$, then we would like to know what 'traces' are left from original occurrences at {*,1,11} in any word to which the following elimination functor is applicable; d is eliminated at *, then at 0, and then at 111. According to the computation, the only traces will be at addresses 00, 011, and 11. In particular, if d's occurred originally only at *, 1, and 11, they would not be completely eliminated; there would still be occurrences of d at 00, 011, and 11.

A study of Figure 5 reveals that the address computation is driven to a conclusion by an algorithm which applies the following rules:

Rule 1: $<*>\{*\} =$

Rule 2: If $i < s(d)$ (the stratification of d), then for every address $a <> \{*,1 \cdot a\} = A_{di} \cdot a = \{a_{i1} \cdot a, \ldots, a_{imi} a\}$.

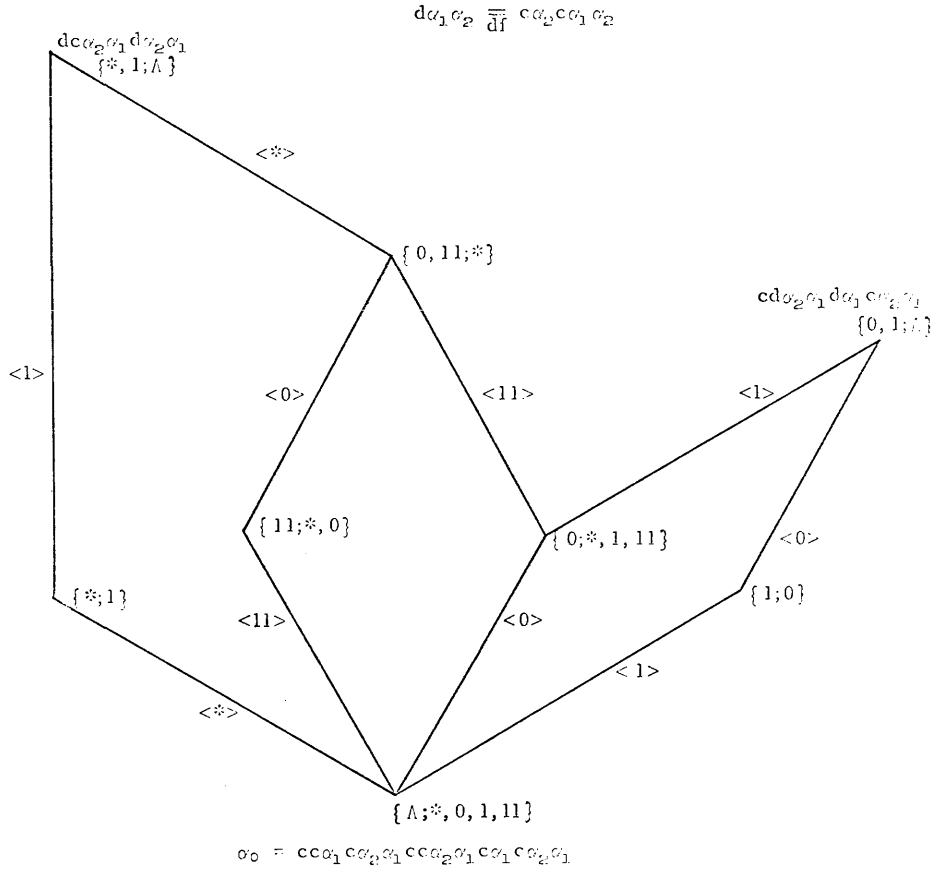


Figure 6—The Hasse diagram of E (α_0)

The address set is interpreted to be null if the multiplicity, m_i, is zero, i.e. if the definition is scope expanding at i.

Rule 3: If $*\in A_1$, then $<a>\{a \cdot A_1\} = \{a \cdot <*> A_1\} \cdot$

Rule 4: If $a \leqslant b$, then $<a>\{a,b\}=\{b\}$.

Rule 5: (the first commutation rule) If a and b are independent addresses (i.e. neither preceding the other), then $<a,b> = <b,a>$.

Rule 6: (the second commutation rule) If $m_i \neq 0$ (i.e. if the definition is not scope expanding at i), then $<a,a{\cdot}i{\cdot}a'> = <a{\cdot}a_{i1}{\cdot}a', \ldots, a{\cdot}_{mi}{\cdot}a{\cdot},a>$. If the multiplicity is zero, $m_i=0$, then we can only say $<a,a{\cdot}i{\cdot}a'> \subseteq <a>$ (the functor $<a>$ has in its domain any expression with an occurrence of d at a; the functor $<a,a{\cdot}i{\cdot}a'>$ is more restricted because it also requires an occurrence at $a{\cdot}i{\cdot}a'$).

Rule 7: (the distribution rule) If the d-elimination functor F is applicable to the address sets A_1 and A_2, then it is applicable to $A_1 \cup A_2$ and

$$F(A_1 \cup A_2) = FA_1 \cup FA_2.$$

Suppose we now represent the distribution of occurrences of d and φ_d in an expression β by another expression of the form $\{C^{-1}d\beta;C^{-1}\varphi_d\beta\}$. Then, for the definition of figure 3, the expressions in $\rho\alpha$: $dc\alpha_2\ \alpha_1\ d\ \alpha_2\ \alpha_1$, $c\ d\ \alpha_2\ \alpha_1\ d\ \alpha_1\ c\ \alpha_2\ \alpha_1$, and $\alpha_0 = c\ c\ \alpha_1\ c\ \alpha_2\ \alpha_1$ $c\ c\ \alpha_2\ \alpha_1\ c\ \alpha_1\ c\ \alpha_2\ \alpha_1$, will be represented by $\{*,1;\Lambda\}$, $\{o,1;\Lambda\}$, and $\{\Lambda;*,0,1,11\}$ respectively. Moreover these three expssions all belong to $E(\alpha_0)$, consisting of eight expressions in a partial ordering whose Hasse diagram appears in Figure 6.

1. If β_1 and β_2 are forms over the extended alphabet and $\beta_1 \overset{d}{\equiv} \beta$, then there exists another form β_o such that $\beta \geqslant_{d\ u}^{d} \beta_0$ and $\beta_2\ d\ \beta_o$.

This is a very special case of the Church-Rosser theorem, which deals with the very general extension of formal systems by definitions. See, for example, Curry & Feys.[6]

2. Every expression β_1 over the extended alphabet determines a unique expression β_0 over the original alphabet such that $\beta_1 \geqslant_d \beta_0$.

In other words, every equivalence class of expressions contains one and only one expression over the original alphabet, its unique 'normal form'.

3. For every expression β, $L(\beta)$ is a lattice. Figure 6 shows that this need not be true of $E(\beta)$.

4. $E(\beta)$ can only be infinite if the definition is scope expanding.

5. $E(\beta)$ can only fail to be a lattice if φ_d is self-dominating, and β contains a $\varphi_d\varphi_d$-domino.

For example, this is the case in figure 6 because the occurrence of φ_d at 1 in α_0 dominates the occurrence

$d\alpha_1\alpha_2$	$\overline{\overline{df}}$	$c\alpha_2 c\alpha_1\alpha_2$
$d_1\alpha_1\alpha_2\alpha_3$	$\overline{\overline{df}}$	$c\alpha_1 c\alpha_2\alpha_3$
$d_2\alpha_1\alpha_2$	$\overline{\overline{df}}$	$d_1\alpha_1\alpha_2\alpha_1$
$d\alpha_1\alpha_2$	$\overline{\overline{df}}$	$d_2\alpha_2\alpha_1$

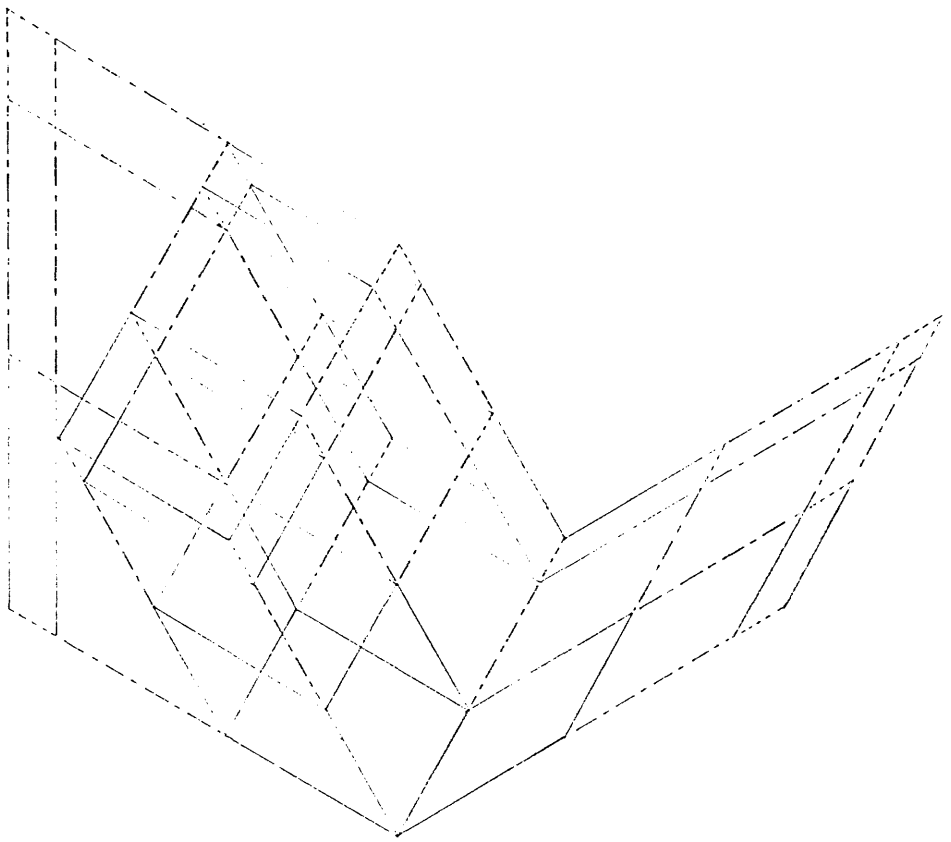


Figure 7—The Hasse diagram of E (α_0) when d is factored

at 11 and is dominated by the occurrence at *.

6. A partial ordering is called modular if, whenever $\beta_2 \geqslant_d \beta_1$, any two complete chains between β_2 and β_1 have the same length. Figure 6 is not modular because one complete chain between {*,1;Λ} and {Λ;*,0,1,11} is of length two and the other two are of length three. This happens because the definition is a breadth reducer, $C^{-1}\alpha_2\varphi_d = \{0,11\}$, and $C^{-1}\varphi_d\alpha_0 \supseteq \{*,0,11\}$.

7. If we factored this definition into a pure depth reducer, a pure breadth reducer, and a pure permuter, as mentioned before, we separate their several effects into three phases of extension, as one can observe by studying Figure 7.

CONCLUSION

One could extend the very special theory of this paper in a number of directions. For example, one could go on to study the effects of sets of definitions; the factorization into 'pure' types illustrated this. One could relax the 'completeness' condition we were using in our categories of languages, and consider various kinds of incomplete categories. One could relax the restrictions implied in our use of the word 'explicit' to permit definitions by cases, or suppression of variables, or even recursion. The fact that there is such a theorem as the Church-Rosser theorem means that some kind of light should emerge.

However, it seems to me that even our simple example has taught us some unexpected things.

First of all, the processing we have been discussing can occur at meta-language level as well as at object-language level.

For example, in Church's axiomatic system for the propositional calculus the first axiom is

$$\text{A1} \quad :\vdash \alpha_1 \supset (\alpha_2 \supset \alpha_1).$$

The Name A_1 belongs to the meta-language just as the punctuation, parentheses, and '| —: do. We can give A_1 several important types of interpretation in the meta-language, all of which we would like to use in the same system (see Gorn.[9] For example:

1. A_1 is an address in an addressing system for the storage of axioms and theorems.

2. A_1 is an instruction representing an operation on two expressions; $A_1\alpha_1\alpha_2$ will produce $\supset\alpha_2\supset\alpha_1\ \alpha_2$ in the language $\mathscr{L}\mathscr{S}$ of a category. This semantic effect carries with it all the syntactic effects of the explicit definition of figure 6 with A_1 for d and $\supset$ for c. In particular, the syntactic relation

$$A_1\supset\alpha_2\alpha_1A_1\alpha_2\alpha_1\equiv\supset A_1\alpha_2\alpha_1A_1\alpha_1\supset\alpha_2\alpha_1$$

would also have the semantic consequence that both expressions specify methods of deriving $\alpha_0 = \supset\supset\alpha_1\supset\alpha_2\alpha_1\supset\supset\alpha_2\alpha_1\supset\alpha_1\supset\alpha_2\alpha_1$. We also know, because of the A_1A_1 -domino in α_0, that there can be no expression in the $\mathscr{L}\mathscr{S}$ from which both can be derived.

This means that much of the processing needed in theorem proving is not essentially different, in spite of the shift in level of interpretation, from the processing at object level.

Moreover the introduction of explicitly defined symbols, as in $\supset\alpha_1\alpha_2 \overset{\text{df}}{=} V\sim\alpha_1\alpha_2$, and the introduction of systems of names of axioms and theorems are not only similar to each other but also play the same role in the meta-language as the introduction of new names of instructions, or of macro-instructions in a programming language. All these new symbols can also be interpreted as names of linguistic transformations on the object language, and therefore as operators in a meta-language over the object language.

This means that there is a second effect from our study. A number of seemingly different problem areas in information science, logic, and linguistics become identified:

a. The growth of a mechanical language by the explicit defining of new characters: it can be controlled if we have the definitional forms recorded, and use such processors as form recognizers, recognizers of the relation $\geqslant$ no matter what the 'd', reducers to normal form independent of d and φ_d, equivalence recognizers, etc. Each of these processors will permit one to vary the alphabets, the forms, and the definitions, of which there will be very many. But of the basic processors there will be only a handful, and they will be applicable at many levels as well as to many alphabets and many definitions. The system should also contain a number of the translators among the language functions of the category.

b. A number of problems in artificial intelligence: imagine that a language is specified for which a recognizer is either unknown or impossible; we want a 'heuristic' recognizer which will 'often' determine whether an expression belongs to the language by attempting to solve the problem of setting up a derivation for it. If the processor is halted before this happens, the recognition is left undecided. The investigations of Newell, Shaw and Simon, such as the logic theorist,[13] and the general problem solver[14] can be formulated this way, because the set of all 'true' expressions can be considered a language just as easily as the set of all expressions. This is because the axioms can be considered as ad hoc syntactic generators of good expressions, and such 'transformation rules' as modus ponens, substitution, etc. can be considered as derivation rules in a generative grammar.

The heuristic program recognizes patterns of applicability of these transformation rules in order to set up the 'subgoals'. The availability of our standard form recognizers, as efficient processors rather than heuristic ones, makes the main heuristics more efficient. See Boyer,[1] and Chroust.[4]

c. The extension of formal systems by definition: one considers the 'derivation rules' of the formal system to be the 'transformation rules of the language', in the sense of Carnap, just as described in b. This is why the material in the first four chapters in Curry & Feys,[6] leading up to the Church-Rosser theorem for general formal systems had its counterpart in our simplified and much more highly structured languages.

d. We handled an explicit definition as though it were an addition to the grammar of the language which permitted certain simple linguistic transformations to be performed. Our classification into depth reducers breadth reducers, etc. is a very primitive classification very much in the spirit of the much more complicated 'linguistic transformations' in the sense of Chomsky and Harris. Can we not consider that anyone who defines a new expression in a language is causing the language itself to change; is he not really changing the grammar of the language, and not merely adding new expressions?

e. Suppose we have two programming languages, such as, for example, the assembly languages of two different binary general purpose machines of the von Neumann type. We can consider that each instruction of either language is defined in terms of a common 'micro-language' of bit-manipulations. Some of these definitions are recursive, but many can be made explicit. Among the difficulties of machine-to-machine translation is the fact that many instructions, like 'ADD', do not really have the same definition in the different machines.

The same problem arises if a community of users, beginning with a common language, have the freedom

of introducing macro-definitions independently of one another. Sub-communities with common problems will develop different sub-languages by different systems of macro-definitions, and if these different sub-communitities do not remain in constant communication, their distinct 'dialects' will, in effect, grow into distinct languages, and the translations, even with only explicitly defined macro-instructions, will be come difficult because of what, in this paper, is called the domino effect among definitions.

The non-lattice effect we remarked on for definitions related by dominance we can now interpret as a danger of loss of communication; when the processors we have discussed are not available, the different sub-communities will not even know when they are saying the same things.

REFERENCES

1 M CHRISTINE BOYER
A tree structure machine for proving theorems
Moore School Master's Thesis August 1964

2 JOHN W CARR III
The growing machine
to be submitted to a computer publication

3 T E CHEATHAM
The introduction of definitional facilities into higher level programming languages
AFIPS Conference Proceedings Vol 29
Fall Joint Computer Conference 1966

4 GERHARD CHROUST
A heuristic derivation seeker for uniform prefix languages
Moore School Master's Thesis August 1965

5 *A CHURCH*
Introduction to mathematical logic
Vol I Princeton 1956

6 H B CURRY R FEYS
Combinatory logic
Vol I North Holland 1958

7 B A GALLER A J PERLIS
A proposal for definitions in ALGOL
to appear in Communications of ACM

8 SAUL GORN
Common Programming Language Task
Rep AD 236-997 July 1959 and
Rep AD 248-110 June 30 1960 U S Army
Signal Res and Develop Labs Fort Monmouth
N J Contract No DA-36-039-SC-75047

9 IBID
The treatment of ambiguity and paradox in mechanical languages
Am Math Soc Proc Symposia in Pure Mathematics
Vol V (1962) *Recursive function theory*
Apr 1961

10 IBID
Mechanical pragmatics: a time motion study of a miniature mechanical linguistic system
No 12 Vol 5 December 1962 pp 576 589

11 IBID
Language naming languages in prefix form
Formal language description languages for computer programming
North Holland 1966 pp 249-265

12 IBID
An axiomatic approach to prefix languages
Proceedings of the symposium at the international computation centre
Symbolic languages in data processing
Gordon and Breach 1962

13 A NEWELL J C SHAW H A SIMON
Emperical explorations of the Logic theory machine: a case study in heuristics
Proceedings of the western joint computer conference 1957 pp 218-230

14 IBID
Report on a general problem-solving program
Information processing
UNESCO Paris 1959 pp 256-264

15 T J OSTRAND
An expanding computer operating system
Moore School Master's Thesis December 1966