[*Mechanical Translation*, Vol.6, November 1961]

A Program for the Machine Translation of Natural Languages

by W. Smoke and E. Dubinsky*, University of Michigan, Ann Arbor, Michigan

In the following we give an account of a computer program for the translation of natural languages. The program has the following features: (1) it is adaptable to the translation of any two natural languages, not just to some particular pair; (2) it is a self-modifying program—that is, given the information that it has produced an incorrect translation, together with the translation which it should have produced according to the linguistic judgment of an operator, it will modify itself so as to eliminate the cause of the incorrect translation.

Before the account of the program itself we give a short sketch of the considerations which led to the program, together with a statement of the reasons why we feel a program of the type presented will be adequate for machine translation.

The naive way to do research in machine translation would be to pick a pair of languages, say Russian and English, and to try to discover some sort of transformational rules connecting them, in terms of which a computer program might be written. The transformation rules might be derived from a comparison of the two languages on the basis of old-fashioned grammar, or from the more recent theories developed by structural linguists, or by other means. Most of the effort in machine translation research so far has gone into deriving such transformation rules by one method or another, and making them more explicit; that is to say, putting them into a form in which they can be programmed, and patching up the holes which are apt to appear in such rules when they are applied to an actual text. Assuming that this kind of effort were successful, its result would be a computer program, probably haywired together, which would—given a certain restricted kind of input material—produce a more-or-less accurate, more-or-less readable translation. One would never know exactly when the machine was going to bog down on some particularly difficult Russian passage, and when the program did bog down, no one would know exactly where to put the next piece of haywire to make it run again.

Sapir said, "All grammars leak." The same is going to be true of any computer program for the translation of languages: the time will come when it is inadequate —there will always be exceptions. If for no other reason, this will be true because languages are always changing. For this reason, we feel that any computer program which deserves the name of a language translation program has to be a program which is capable of expansion, in a regular manner, to keep up with the demands that are made on it. Essentially, what one must have is a machine which *learns* to translate, which is automatically modified as it translates more and more. Now how would one program a machine so that it would translate and in addition be able to modify its process of translating?

Let us try to reach a more precise idea of what a self-modifying translation program would look like. The complete program P would consist of two parts, a translation program T and a master program M. The program T would be responsible for the actual translation from one language to another, while M would take care of making the changes in T. Thus suppose that P, or the part T of P, is capable of translating the Russian sentences $S_1, \ldots, S_n$ correctly into English, but that it translates the sentence S_{n+1} incorrectly. Then the modification in P would take place as follows. Given S_{n+1} and a correct English translation of S_{n+1} as input, the master program M would modify T to obtain a translation program T'. The new complete program P' would consist of M and T', and would translate S_{n+1} correctly. Furthermore, while we need not require that P' be capable of translating all of $S_1, \ldots, S_{n+1}$ correctly, it is necessary that after some limited series $P, P', P'' \ldots P^{(m)}$ of modifications to P, a program $P^{(m)}$ be obtained which is capable of translating all of $S_1, \ldots, S_{n+1}$ correctly. That is, while the modifications can introduce errors, we cannot have a strictly recurring series of errors introduced.

Finally, the programs $P^{(m)}$ which are obtained as modifications of P should be subject to some kind of regularity. We do not want a program which becomes complicated and uneconomical too fast; that is, the series of modified programs should converge in some reasonable sense, not diverge.

This process suggests to us the familiar kind of behavior which we call learning behavior. We like to think of a machine which is programmed in the manner outlined as a machine which learns to translate.

How does one go about constructing a translation

* The authors would like to thank A. Koutsoudas, without whose stimulus and support this paper would not have been written.

program of the type we have described? It should be fairly clear by now that this problem is more a computer problem than a linguistic problem. But it is not a problem in programming techniques.

When we set out to attack the problem, we felt that what we needed was a way of discussing languages, translations, computers, etc., from an abstract point of view. That is, the problem in its main features is clearly independent of whether we are translating from Russian into English, or Chinese into Sanskrit. Furthermore, it will be unimportant whether we think of using a Univac or an IBM 709 as a vehicle for the translation program.

We can observe at this point that a solution to the problem as stated would of necessity have certain bonus features: it would not just be a solution to the problem of translating, by machine, Russian into English, but would, in all likelihood, be a solution to the problem of machine translation for any given pair of languages.

But if we do not restrict our use of the term 'language' to Russian or to English, or to any other particular, concrete language, then what do we have in mind? And what do we have in mind when we discuss a translation, a translation program, or a translation program embodied in a machine?

Perhaps we should first examine the question of what we mean by a translation program. The idea of a computer program abstracted from any particular computer is not new; it is usually depicted by a flow-diagram. When the same thing is studied by those with a more abstract turn of mind, it is sometimes called an abstract automaton. Abstract automata, at least the kind we are interested in, can be thought of as a collection or matrix of information-retaining cells. The information retained by any particular group of cells at any one time may be called the state of this part of the automaton. The state of the entire automaton changes discretely through time, its state at one instant completely determining its state at the following instant. In an input state the cells of the automaton are readied with information from the "outside"—the input information. Corresponding to each input state will be an output state, signaled by a "stop" or some such indicator. When the information from the cells is read off to the "outside", it becomes the output information. The output state is a function of the input state, and correspondingly, the output information is a function of the input information.

An automaton, in its capacity as a means for passing from input to output, is simply a certain kind of realization of a function. In our case, the function which is to be realized is what we have been calling a translation. The domain of this translation function is a certain class of texts in some language, and its range is a class of texts in another language. A text might be anything from a sentence to a paragraph or an article. Whatever it is, however, it is clear that it must be something which can be represented as a part of one of the input states (in the case of the source language), or as a part of the output states (in the case of the target language). That is, however we represent a text in a language, this representation must be essentially equivalent to representation by a state, or a partial state, of an automaton. If we restrict our thinking to reasonably realistic automata, we may suppose that an automaton has only a countable number of cells, each cell having only finitely many states. If we represent the cell states by a countable alphabet—in fact we will consider only finite alphabets—then a state of an automaton, and hence a text in a language, can and must be represented by a sequence from this alphabet.

Thus we are led to the following provisional definition of a language: a language is, for our purposes, nothing more than a collection of sequences of symbols from some finite alphabet. It has turned out to be convenient to study systems with a bit more structure than this definition would imply. In fact, we have been primarily interested in studying systems of finite sequences with some kind of binary composition. In the case of an associative binary composition, the systems are equivalent to a special kind of semigroup.* Lately, we have become interested in systems with non-associative binary composition. The reason for this shift of interest will become clear as we go on.

But before we go on to describe our latest efforts, let us spend a few moments reviewing the earlier work. First, what is the problem? We can formulate it as follows. We are given two collections of corresponding texts, that is, two collections of finite sequences of symbols from two alphabets. The symbols may be thought of as letters, words, or any other convenient linguistic unit (which particular unit we use is of little importance at this stage). The correspondence is, more exactly, a function, the translation function, from the one collection (source language) to the other (target language). But what kind of function? We must require that the function be such as is realizable by an automaton. But this requirement by itself is not sufficiently restrictive. In fact, as long as we are dealing with only a finite number of pairs of corresponding texts, it would always be possible, given sufficiently large storage capacity, simply to program a computer to translate each of the source language texts by looking it up in a text "dictionary", where the complete text together with its translation is stored, and feeding out the translation.

This means that a translation function, defined only on a finite domain, is always realizable in a trivial fashion. Therefore, it is reasonable to consider functions defined on infinite domains. In fact, since it seems to be impossible to give any explicit method for singling out sequences of symbols which we want to translate from those that we will not be called upon to translate (i.e., for separating "meaningful" from "non-meaningful" sequences of symbols) it is reasonable to consider functions which are defined on all sequences of symbols from a given alphabet. But now, we clearly

* See appendix.

can have functions which are not realizable by automata.

What sorts of functions are realizable by automata? A very simple example of such a function is provided by a homomorphism defined on a free and finitely generated semigroup. In fact, a homomorphism is defined by exploiting the sequential character of the objects in its domain. Each element in its domain is a unique sequence of a finite number of symbols, and the definition of the homomorphism on the sequence is accomplished by letting the sequence translate as the sequence (in the same order) of the translations of the symbols. The fact that there are only finitely many symbols, together with the uniqueness of the representation by sequences of these symbols, guarantees the realization of the homomorphism by an automaton. An example of a homomorphism is given by a simple substitution cipher, e.g.

THE BOY WENT HOME

translates as

UIF CPZ XFOU IPNF

using the device of translating each letter of the alphabet by the following letter, translating space as space, and extending the function thus defined to a homomorphism.

What is wrong with using this kind of translation function for Russian to English translation? The difficulty lies partially in the size of the unit that would be necessary. One would probably need to use a unit of clause size, because of the ambiguity which would arise in dealing with units of lesser length. But this is not the only difficulty which might arise.

Suppose that we have a collection of units U and a homomorphism T defined on sequences of elements of U. In other words, U is the set of generators of the free semigroup that is the domain of T. Suppose that a and b are two of the units of U, and that $T(a) = \overline{a}$, $T(b) = \overline{b}$. If then, we encounter the sequence ab, its translation will be $T(ab) = T(a)T(b) = \overline{a}\,\overline{b}$. Suppose this is incorrect, that is, we wish to assign another translation to the sequence ab. Recall that in this case, we wish to modify the translation function T to obtain a new translation function T' with the property that T' translates ab correctly, and also translates those sequences of elements of U which do not contain ab as did T. But now, T' cannot be a homomorphism. For any homomorphism which agrees with T on U will be identical with T. In particular, then, such a homomorphism cannot translate ab correctly, if T does not. Thus we see that we cannot restrict our choices of translation functions to homomorphisms, if we wish to be able to modify these functions as we indicated earlier.

If homomorphisms do not lend themselves to modification, what kinds of functions, realizable by automata, do have this property? Perhaps the first such function to consider is what we call a sequential function. A sequential function is a function defined on the free, finitely generated semigroup of all sequences of symbols of some finite alphabet. It is a kind of semi-homomorphism. The defining property of a sequential function f is that if a and b are two elements of the domain semigroup, then $f(ab) = f(a)b'$, where b' *is* some element of the semigroup which contains the range of f. A homomorphism h is a special case of a sequential function, since $h(ab) = h(a)h(b)$, that is, $b' = h(b)$ in this case. In general, b' will depend on a. That is, because of the fact that the range semigroup as well as the domain semigroup is free on its generators, the correspondence which assigns to the elements b, c, d, etc., of the domain, the elements b', c', d', etc., which occur as well-defined parts of the sequences $f(ab) = f(a)b'$, $f(ac) = f(a)c'$, $f(ad) = f(a)d'$, etc., is a function which has the same domain and range semigroups as f. We can denote this function by f_a, so that we have, for any element b of the domain, $f(ab) = f(a)f_a(b)$. Then in order that the sequential function f not be a homomorphism, it is sufficient that there be two elements a and b, such that for some element c we have $fa(c) \neq f_b(c)$. That is, the translation $f_a(c)$ of c in the sequence ac is different from the translation $f_b(c)$ of c in the sequence bc. Furthermore, it turns out that this new function f_a is again a sequential function. For we can calculate $f_a(bc)$ as follows. By definition $f(abc) = f(a)f_a(bc)$. But also $f(abc) = f(ab)f_{ab}(c) = f(a)f_a(b)f_{ab}(c)$. Thus we have $f(a)f_a(bc) = f(a)f_a(b)f_{ab}(c)$ so that $f_a(bc) = f_a(b)f_{ab}(c)$, which shows that f_a is a sequential function. We call f_a a derived function of f. Carrying the above computation a little farther, we have $f_a(bc) = fa(b)(f_a)_b(c)$; hence $f_a(b)(f_a)_b(c) = f_a(b)f_{ab}(c)$, and therefore $(f_a)_b(c) = f_{ab}(c)$. That is, the function derived from fa using b is the same as the function derived from f using ab. Thus the correspondence ψ which associates to an element a of the semigroup and a sequential function f the sequential function $\psi\ (f, a) = f_a$, has the associativity property $\psi\ (\psi\ (f, a), b) = \psi\ (f, ab)$. What this means is that a sequential function f can be defined on a free semigroup by defining the sequential functions derived from f on each of the generators of the semigroup. In particular, then, a sequential function certainly becomes realizable by an automaton if it has only finitely many derived functions, and is defined on a finitely generated free semigroup. In fact, the realization of a sequential function of this kind is accomplished in a very natural way by the type of automaton known as a sequential automaton, or a finite state machine. These automata have been extensively studied by several authors [3,4,5,6]. To obtain the sequential automaton A corresponding to a sequential function f, we need merely take, as a set of states F of A, the set of derived functions f_a of f, letting f itself be the initial state. The input I of A is the semigroup on which f is defined, and the output O is the range of f. The next-state function of A is the function f defined previously, and the output function of A is the cor-

respondence ϕ which associates to an element b of I and to a state f_a of A the element $\phi\ (f_a,\ b) = f_a(b)$ of O. We thus obtain the sextuple $A = (I, O, F, f, \psi, \phi)$ with the requirement $\psi\ (\psi\ (g,\ a),b) = \psi\ (g,ab)$ on ψ and a corresponding requirement $\phi\ (g,ab)$ — ϕ (g,a) $\phi\ (\psi\ (g,\ a),b)$ on ϕ where g is in F, a and b are in I. Except for the designation of f as initial state, the restriction of F to be finite, and the restriction of I and O to be free and finitely generated, this is exactly the definition of a sequential machine as given by Ginsberg.[3]

Equivalently, one may begin with a sequential machine with a designated initial state, and define a sequential function. It is clear intuitively that an automaton will realize a sequential function just in case the output sequence corresponding to an initial segment of some input sequence is an initial segment of the output sequence corresponding to the complete input sequence.

A simple example of a sequential function is given by the translation of

THE BOY WENT HOME

as

TBG IXW TYMG ODQV

accomplished by using the correspondence between the letters and the numbers from 1 to 26, and assigning to each letter in the first row the letter which corresponds to the sum of the numeral values, modulo 26, of the letters up to and including the one to be translated (except that space always translates as space). The sequential function thus defined has 26 derived functions, f_A through $f_Z = f$. Every derived function is equal to one of these; e.g., $f_{AB} = f_C$.

Let us now return to a consideration of the problem of modifying a given translation function T, where we now may let the modified function T' be a sequential function. Suppose, for simplicity that T is the function considered before, defined as an extension to a homomorphism of some function (we can still call it T) defined on the set U of free generators of a free finitely generated semigroup. Suppose also that we wish to have T' agree with T except on sequences containing ab, and that the proposed modification on ab is that b should translate as $\overline{\overline{b}}$ after a, and otherwise as $\overline{b} = T(b)$. Then we can define T' by letting $T'_m = T$ if m is a sequence not ending in a, $T'_a(c) = T(c)$ if $c \neq b$, $T'_a(b) = \overline{\overline{b}}$, and then let T' be the extension which results by enforcing the associativity condition. This kind of modification also succeeds in case T is already a sequential function which is not a homomorphism.

Thus we are able to introduce modifications into translation functions which are sequential functions, if these modifications are suitably restricted. Essentially, we can let preceding context modify the translation of a particular unit, thereby modifying the translation function itself. By running the text into the machine from right-to-left instead of from left-to-right, we could equally well modify the translation of a unit on the basis of following context. In fact it would seem that, by proceeding from left-to-right and "holding-up" the translation of a given unit until the machine senses what follows it, it would be possible to take into account both preceding and following context. That is, we could attempt to construct a sequential machine that would translate b as $\overline{b}$ in the context abc and as $\overline{\overline{b}}$ otherwise. This attempt would run into the difficulty that b would go untranslated in the context ab occurring at the end of input sequences, since the machine "waits" to see what comes next before translating b after a, and in case ab is a terminal segment nothing comes next. This difficulty could be avoided by the addition of a special symbol [] to the input alphabet, having the function of "closing off" input sequences, so that the terminal segment ab would become ab[]. This device, however, is awkward.

A more serious problem is encountered when we examine sequential functions from the point of view of their flexibility with regard to alterations of order between input and output. For example, it is impossible to construct a finite-state sequential automaton which will realize the very simple function which translates

THE BOY WENT HOME

as

EMOH TNEW YOB EHT

i.e., the function which simply reverses the order of the letters in an input sequence.

Another difficulty that we run into using sequential functions as translation functions is illustrated by an attempt to construct a sequential function, defined on the alphabet ~, ∨, (,), p_1, p_2, p_3, . . . etc., which will correctly translate well-formed expressions of the propositional calculus, in the primitives ~ and ∨, into the equivalent expressions in the primitives ∧ and ⊃. Consider expressions of the form

$$\sim(\ldots\ (\sim((\sim p_1) \vee p_2) \vee p_3)\ldots) \vee p_n$$

which translate correctly as

$$(\ldots((p_1 \supset p_2) \supset p_3)\ldots) \supset p_n.$$

It is intuitively clear that, reading from left-to-right, a sequential machine would translate ∨ as ⊃ if it "remembers" that a ~ preceded the opening parenthesis paired with the closing parenthesis preceding the ∨ in question. But it is clear that to overtax the "memory" of a given sequential machine, it is enough to try using it to translate correctly a proposition of the above form with sufficiently many "levels".

This difficulty is related to the objection, voiced by Chomsky,[2] that arises when one attempts to employ a "finite-state grammar," which is essentially a sequential automaton without input, as a "sentence generator" for languages which have sentences of the form "if . . . then . . .", or "either . . . or . . .". Again, these sentences

may be "nested" to a level which overtaxes the capacity of the machine.

Thus, sequential functions would seem to be not only awkward, but perhaps even basically inadequate for use as translation functions. This is in accord with our intuitive feeling about language. It is not that we feel that a language has a God-given structure of some kind, which it is our task to discover, adopting then a type of translation function which fits this structure. However, we do feel that a given type of translation function will necessarily impose a corresponding structure on the language on which it is defined; and we can then appraise our choice on the grounds of economy, our intuitive feelings of neatness and elegance, etc. By these standards, it appears that sequential functions do not offer a good choice as translation functions.

We have now reached the point where we shall begin to describe our recent work. We intend now to discuss a type of translation function which does not have the inadequacies of those that we have described. In fact, the type of translation function which we now wish to consider, will lead, at the end of this discussion, to what we believe to be a computer program which is adequate for machine translation.

The origin of the program is a system of notation, proposed by Bar-Hillel[1] which is designed to denote the syntactic categories of linguistic expressions. Bar-Hillel's notation can be built up out of the symbols *n, s, /,* \, (,). Used in conjunction with a natural language, expressions which are commonly called nominals—nouns, pronouns, adjective-noun combinations, noun phrases, etc.—are assigned the category *n*. Sentences are assigned the category *s*. An expression which produces an expression of category β when prefixed to an expression of category *a* is assigned the category (β/a). Thus the adjective *the* prefixed to the noun *boy* produces the nominal *the boy;* hence *the* has the category *(n/n)* since *boy* and *the boy* both have category *n*. Similarly, an expression which produces an expression of category β when affixed to an expression of category *a* is assigned the category $(a\backslash\beta)$. Thus *went* in *the boy went* is assigned the category *(n\s),* and *home* is assigned the category *((n\s) \ (n\s))*. The parts of the sentence are assigned categories as follows:

The	*boy*	*went*	*home*
(n/n)	*n*	*(n\s)*	*((n\s) \ (n\s))*
	n	*(n\s)*	
		s	

Perhaps we can notice now that this process of category assignment is in some sense non-associative. That is, the assignment indicated induces an association of the sentence as follows:

((The boy) (went home))

Associated another way, e.g.:

(((The boy) went) home)

the result is not a sentence. This is reflected in the fact that the category of the juxtaposition of *((the boy) went),* an expression of category *s*, and *home,* an expression of category *((n\s) \ (n\s),* is undefined.

An expression may belong to several categories. Thus *home* could also be in category *n;* or in category *(n/n),* as in *home run*. Sometimes the context will determine that a given expression must be functioning in a certain capacity within that context, as *flying* in *they are flying*. That is, if it is known that the entire expression has only the category *s,* then an analysis of the assignments resulting from

They	*are*	*flying*
n	*((n\s)/n)*	*(n/n)*
		(((n\s)/n)\((n\s)/n))
		n

shows that of the three choices of category for *flying* only *n* can be correct. However, consider the sentence

They	*are*	*flying*	*planes*
n	*((n\s)/n))*	*(n/n)*	*n*
		(((n\s)/n)\((n\s)/n))	

Depending on whether we read the sentence as

(They ((are flying) planes))

(They (are (flying planes)))

or as

we choose *((n\s)/n) \ ((n\s)/n))* or *(n/n)* as a category for *flying*. This ambiguity occurs not only in sentences, of course, but also in such an expression as the nominal *purple people eater*. Is it *((purple people) eater)* or is it *(purple (people eater))?*

We have observed that the way we associate the words in a sentence or a phrase can alter the meaning of the expression. It is reasonable to suppose then, that the association of the units in an expression can influence its translation. But this means that we should be studying translation functions defined, not on associative systems such as semigroups, but on non-associative systems. We will not be satisfied, of course, with a computer program which requires that a pre-editor insert parentheses into a Russian sentence before it is given to the machine to be translated. This is not what we have in mind, but rather we think it might prove convenient to break our problem into two parts—to supply parentheses, and to translate. In fact, one way of correctly supplying parentheses will be to try translating all possible associations of a given input sequence, and then to consider that association the correct one which has a translation. If there are two associations with differing translations, this means, of course, that we are dealing with an ambiguous sequence, just as in the case of a sentence with two meanings corresponding to two different associations.

Let us now turn to the program. It will be evident how the construction of the program was influenced by Bar-Hillel's notation.

Recall that we have said that a self-modifying program P for machine translation would consist of a translating part T and a modifying part M. It will be convenient to describe our program in these terms. Let us first describe T, that is, we will describe $T^{(n)}$, the translation program at the nth stage of modification.

The information which is stored in the machine and forms the reference material for T consists of a dictionary and a category multiplication table. The input to T is a source language text. The action of T on this input text is as follows.

1. The units of the input text are referred to the dictionary, and for each unit for which an entry is present in the dictionary, the entry is extracted and brought to the working space of the machine. For each unit for which a dictionary entry is not present, a special entry, indicating dictionary blank, substitutes as a dictionary entry for the unit. A dictionary entry consists of a list of pairs of output units and symbols designating categories.

2. We now have stored in the working space of the machine a list for each input unit. Together these lists comprise a sequence of lists in the same order as the corresponding sequence of input units in the text. This sequence of lists is now processed by a multiplication operation on all possible associations.

For each ordered pair of associated lists, i.e., *(A,B)* in *((AB)(CD))*, and each ordered pair *(a,b)* of entries in *(A,B)*, i.e., a in A and b in B, the machine refers to the category multiplication table. The category multiplication table is a square array of the following type:

	λ	α	β	γ
λ	λ,λ	λ,λ	λ,λ	λ,λ
α	λ,λ	λ,λ	γ,α	$\alpha,$-
β	λ,λ	$\beta,$-	$\lambda,$-	-,-
γ	λ,λ	-,α	α,β	-,β

where the row refers to the first, the column to the second element of the ordered pair. The two elements of *(a,b)* each consist of a pair, the first element an output unit, the second a category. Let us suppose that the category of a is a and that of *b is* β. The machine then locates the entry corresponding to a and β, which in the example is (γ,α), and places two entries in the derived list AB. One entry consists of the pair $(\overrightarrow{\overleftarrow{a}\overleftarrow{b}},\gamma)$ where $\overleftarrow{a}$ and $\overleftarrow{b}$ are the output units of a and b respectively, and the other is the pair $(\overrightarrow{\overleftarrow{b}\overleftarrow{a}},\alpha)$. The derived list AB consists of all such pairs for all choices of *(a,b) in (A,B)* except for the pairs $(\overrightarrow{\overleftarrow{a}\overleftarrow{b}},$-$)$. That is, if in the example the category of α were γ and that of b were α, then the multiplication table entry corresponding to this pair would be $($-$,\alpha)$, which indicates that the first element of the product is "undefined".

In this way, building up derived lists from the basic dictionary entry lists by means of the category multiplication table, a given association of the text is successively reduced. Either the process ends with at least one category assignment to this association, or some derived list is empty because products are undefined. In the latter case the association is considered to have no translation. In the former case the list corresponding to the association is considered to be a possible translation of the original input text and is printed out. The output consists of the complete list of all possible translations corresponding to all associations. If the complete list is empty an indication of this fact replaces the translation.

This completes the description of T. We now describe M, the modifier program. The program M is called into action only when T makes an error, that is, only when it is decided, by a comparison of the input and output texts, that the translation is unsatisfactory. There are two ways in which the translation can be unsatisfactory. On the one hand the list of translations may not contain any translation which is correct. On the other hand the list of translations may contain some translations which are incorrect. In the first case the necessary modification involves supplying a correct translation, in the second case it involves eliminating the incorrect translations.

We must organize the modification process in such a way that these two kinds of modification do not interfere with one another. What we shall do is to perform the modifications of the second type, i.e., eliminating incorrect translations, in such a way that correct translations are never eliminated. Then an unsatisfactory translation of the first kind can occur only if the dictionary is inadequate. That is to say, when there is no correct translation present in the output list, the modification amounts to augmenting the dictionary.

Thus the first part of M is a program which makes up new dictionary entry lists and adds to lists already present in the dictionary. When no correct translation is present in the output list, one must be supplied by the operator. Corresponding to this translation the operator will also indicate, for each input unit, which sequence of units in the translation it corresponds to. This material then becomes the input of M, which locates the unit in the dictionary corresponding to each input unit, or enters it into the dictionary if it does not already appear there, and adds to the dictionary entry list thus obtained the corresponding sequence of output units, assigning them to a special "universal" category. The universal category is defined as that unique category, such that its product with any category is a pair of universal categories.

This completes the first stage of the correction process. If T was the original translation program, the new translation program T' which results from T by the modifications described above will yield a translation of the text which is satisfactory on at least the first count—the list of translations will contain at least one which is correct.

The next problem is to eliminate from the list the incorrect translations. As a first step the operator must

inform the machine exactly in what respect an incorrect translation is incorrect. For example, a translation of a sentence might be incorrect if it contains an incorrectly translated phrase; or each phrase within a sentence may be correct if considered without reference to context, but incorrect when considered in context; or finally, the translation of each phrase may bo correct even when considered in context, but the arrangement of the translation may be incorrect.

The task of the operator is thus as follows: for each association of the text which leads to an incorrect translation, he must decide, for every indicated juxtaposition of two associated elements—assuming it has already been decided that each of the two elements is correctly translated—whether the indicated juxtaposition of the elements (in either order) is a correct translation of the corresponding part of the input. That is, he must think of the corresponding part of the input as entirely divorced from its context, and decide whether in fact it is correctly translated by the juxtaposition (in either order) of the two output units in question. Essentially then he must decide this on the same basis on which he decides on the translations of complete texts: for the purposes of this decision the part of the input in question is treated as a complete text. In particular, if the translation is considered incorrect in one association, it must also be considered incorrect in any other association which contains the two elements associated in the same order, as a translation of the same part of the input.

If it is decided that the translation is correct, the two elements are combined to produce a new element which is also considered correct. Proceeding in this way the operator must eventually encounter a pair of elements which are correct, but whose juxtaposition is incorrect (he cannot encounter a *unit* which is incorrect since we may suppose the dictionary not to contain incorrect entries).

Suppose then that $\overline{a}$ and $\overline{b}$ are two elements, each correct, but $\overline{ab}$ is incorrect. The operator then gives this information to the machine. That is, he supplies the machine with the part of the input which led to the translation $\overline{ab}$, together with the association of the units in $\overline{ab}$, and indicates for each unit of the input text to which units of $\overline{ab}$ it corresponds. Since $\overline{ab}$ is a permissible combination according to the present category multiplication table, this means that the first element of the product $\alpha\beta$ is defined. In the example $\alpha\beta = (\gamma,\alpha)$. The action of M will be to change the categories of $\overline{a}$ and $\overline{b}$ to categories α' and β' such that the first element of $\alpha'\beta'$ is not defined, while at the same time keeping $\alpha'\delta = \alpha\delta$ for every category $\delta \neq \beta'$, keeping $\delta\beta' = \delta\beta$ for every category $\delta \neq \alpha'$, and keeping $\delta\alpha' = \delta\alpha$ and $\beta'\delta = \beta\delta$ for every category δ. In other words M will change the categories of $\overline{a}$ and $\overline{b}$ to α' and β' and respectively, and will add two rows and two columns to the category multiplication table (unless these rows and columns are already present). In the example, the new multiplication table will be as follows.

	λ	α	β	γ	α'	β'
λ	λ,λ	λ,λ	λ,λ	λ,λ	λ,λ	λ,λ
α	λ,λ	λ,λ	γ,α	$\alpha,-$	λ,λ	γ,α
β	λ,λ	$\beta,-$	$\lambda,-$	$-,-$	$\beta,-$	$\lambda,-$
γ	λ,λ	$-,\alpha$	α,β	$-,\beta$	$-,\alpha$	α,β
α'	λ,λ	λ,λ	γ,α	$\alpha,-$	λ,λ	$-,\alpha$
β'	λ,λ	$\beta,-$	$\lambda,-$	$-,-$	$\beta,-$	$\lambda,-$

If now $\overline{a}$ and $\overline{b}$ are not translations of units, but are elements built up out of combinations of units, not only must the categories of $\overline{a}$ and $\overline{b}$ be changed from α and β to α' and β' with the first element of $\alpha'\beta'$ undefined, but also the categories of the successive segments of which $\overline{a}$ and $\overline{b}$ are resulting combinations must be correspondingly changed. For example, if $\overline{a} = \overline{cd}$ and $\overline{c}$ has category γ, $\overline{d}$ has category δ, then the categories of $\overline{c}$ and $\overline{d}$ must be changed to γ' and δ', where γ' and δ' have all the properties of γ and δ except that the first element of $\gamma'\ \delta'$ is α'. This procedure will finally result in changes in the categories of the units of which $\overline{a}$ and $\overline{b}$ are composed. When the category of a unit is changed the corresponding dictionary entry is also changed.

It is asserted that this procedure will lead to the elimination of all incorrect translations and retain all correct translations. It should be clear, in the first place, that an incorrect translation is eliminated if and only if it is eliminated as a result of every association, and that a correct translation is retained if and only if it is retained as a result of some association. Thus, in order to convince ourselves that the procedure actually does lead to the desired result, it will be sufficient to consider a fixed association, and show that any correct translation which results from this association before the modification will continue to do so after the modification, and that no incorrect translation will result after the modification. But it is clear than any pair of output units which enter into at least one correct translation, e.g., $\overline{a}$ and $\overline{b}$ in $\overline{(ab)c}$, are such that there is a choice for the other units, $\overline{c}$ in the example, such that the resulting juxtaposition is a correct translation. Therefore the juxtaposition of these two units is correct, and their categories are not changed as a result of the modification.

On the other hand, given an incorrect translation it must result either from the incorrect juxtaposition of its two highest order segments, in which case it is eliminated at this stage, or from one of these two segments being incorrect, etc. Again, inductively one sees that there must be two segments of some order whose juxtaposition is incorrect, causing their categories to be altered and the translation eliminated.

This completes the description of the modification program M. It will probably be helpful at this point to consider an example of the use of T and M.

Let us suppose we are translating from English into German. We will take as our input unit the word, and

consider the input text *the boy left.* Let us suppose also that, corresponding to the three input units, the dictionary contains the three entries

THE: DER α BOY: KNABE δ LEFT: LINKS ε
DAS β
DIE γ

and that the portion of the category multiplication table in which we are interested is as follows (only the required products are indicated):

	λ	α	β	γ	δ	ε	μ
λ							
α	λ,λ				$\mu,-$		
β	λ,λ				$-,-$		
γ	λ,λ				$-,-$		
δ	λ,λ					$-,\delta$	
ε							
μ						$-,-$	

The first act of *T* is to place the dictionary entries in sequence in the work space:

DER α KNABE δ LINKS ε
DAS β
DIE γ

There are two possible associations from which a translation might be obtained:

(1) $\left[\begin{array}{l}\text{DER } \alpha \quad \text{KNABE } \delta \\ \text{DAS } \beta \\ \text{DIE } \gamma\end{array}\right]$ LINKS ε

(2) DER α (KNABE δ LINKS ε)
DAS β
DIE γ

Since of the products $\alpha\delta$, $\beta\delta$, and $\gamma\delta$, only the first element of $\alpha\delta$ is defined, the first association reduces to

DER KNABE μ LINKS ε

but, as $\mu\varepsilon$ is undefined, no translation results from this association.

From the second association we obtain first the derived list

DER α LINKS KNABE δ
DAS β
DIE γ

since the first element of $\delta\varepsilon$ is undefined, and the second is δ. This list then reduces to

DER LINKS KNABE μ

so that the entire output consists of this one translation.

Suppose now that it is decided that the correct translation of *The boy left* is not *Der links Knabe* but *Der Knabe verliess.* Assuming that the correspondence between input units and output units is indicated as

THE—DER
BOY—KNABE
LEFT—VERLIESS

the modification program *M* will locate the dictionary entries corresponding to the input units, and will enter *verliess* in the list for *left,* assigning to it the universal category λ.

Again using *The boy left* as input, the new translation program will cause the sequence

DER α KNABE δ LINKS ε
DAS β VERLIESS λ
DIE γ

to appear in the work space. From the association

$\left[\begin{array}{l}\text{DER } \alpha \quad \text{KNABE } \delta \\ \text{DAS } \beta \\ \text{DIE } \gamma\end{array}\right]$ LINKS ε VERLIESS λ

we obtain

DER KNABE μ LINKS ε
VERLIESS λ

and from this list, the two translations

DER KNABE VERLIESS λ
VERLIESS DER KNABE γ.

From the second association

DER α $\left[\begin{array}{l}\text{KNABE } \delta \quad \text{LINKS } \varepsilon \\ \text{VERLIESS } \lambda\end{array}\right]$
DAS β
DIE γ

we get

DER *a* LINKS KNABE δ
DAS β KNABE VERLIESS λ
DIE γ VERLIESS KNABE λ

which leads to the translations

DER LINKS KNABE μ
DER KNABE VERLIESS λ
KNABE VERLIESS DER λ
DER VERLIESS KNABE λ
VERLIESS KNABE DER λ
DAS KNABE VERLIESS λ
KNABE VERLIESS DAS λ
DAS VERLIESS KNABE λ
VERLIESS KNABE DAS λ
DIE KNABE VERLIESS λ
KNABE VERLIESS DIE λ
DIE VERLIESS KNABE λ
VERLIESS KNABE DIE λ

so that the complete list of translations, from both associations, has fourteen members. *Der Knabe verliess* resulting from both associations.

Suppose now it is decided that only *Der Knabe verliess* is correct, and that in fact we wish to retain it only as a result of the first association. That is, we can decide first that *links Knabe* is incorrect as a translation of *boy left* and that so also are *Knabe verliess* and *verliess Knabe,* and finally, that while *Der Knabe*

and *verliess* are correct as translations of *the boy* and *left,* that *verliess der Knabe* is incorrect as a translation of *The boy left.* In terms of the categories, this means that the dictionary entries are corrected to:

THE: DER α' BOY: KNABE δ' LEFT: LINKS ε'
DAS β VERLIESS λ'
DIE γ

and the multiplication table becomes (part of it):

	λ	α	β	γ	δ	ε	μ	δ	ε	λ
λ										
α	λ,λ				μ,-					
β	λ,λ				-,-					
γ	λ,λ				-,-					
δ	λ,λ					-,δ				
ε										
α'								μ',-		
δ'									-,-	-,-
μ'								-,-	-,-	λ,-

(One notes that it would be possible for a category to become empty, all units belonging to it becoming reassigned. Thus it would be reasonable to periodically examine the multiplication table for unnecessary categories.)

We will conclude by offering a few comments on methods of using the program. In the first place, it should be clear that it would be possible to institute several different kinds of "training programs" for the program. One could begin with a completely blank dictionary and a multiplication table of the form

	λ
λ	λ,λ

and begin translating sentences as texts. It would probably be more reasonable, however, to begin with the above multiplication table and a dictionary already reasonably large, and begin translating short and more or less unambiguous phrases, thus adding gradually to the category system.

It is of course evident that a text need not be any one in particular of the standard linguistic units, but it might be mentioned that the segment which we have been referring to as a unit is similarly unrestricted. The only requirement on the system of segmentation of the input text, leading to these units, is that it be such as to give a free decomposition, that is, that no input text should have two distinct decompositions as a sequence of units. The obvious choice is of course the word, but theoretically one could use letters of the alphabet, syllables, sentences, etc. In fact, if the details of the decomposition could be worked out, some choice of stems, prefixes, and endings might materially reduce the size of the dictionary (at the cost of increasing the size of the multiplication table, of course). There is no restriction at all on the output units. Thus if the input units were words, the output units could be, and frequently would be, sequences of two or more words.

Received July 16, 1959

APPENDIX

Binary Composition and Semigroups

A set S is said to have defined on it a (not necessarily associative) law of binary composition if there exists a map $S \times S \rightarrow S$. The image of a pair (*a, b*) of elements of S under this map is denoted *ab.* The map $S \times S \rightarrow S$ is associative if for every three elements *a, b, c* of S we have

$$(ab)c = a(bc)$$

A system with an associative binary composition is called a semigroup.

A subset *T* of S is a subsemigroup of S if the restriction of $S \times S \rightarrow S$ maps $T \times T$ into *T.* The intersection of any family of subsemigroups of S is again a subsemigroup of S. If G is any set of elements of S, the subsemigroup generated by G is the intersection of all subsemigroups containing G, and G is called a set of generators for this subsemigroup. Every subsemigroup *T* of S has at least one set of generators, namely *T* itself. In particular, S has a set of generators. A semigroup S is finitely generated if it has a finite set of generators.

The product of any sequence $s_1, s_2, \ldots, s_n$ of elements of a semigroup S is an element of S defined inductively in terms of the binary composition, and is shown to be independent of the association of the sequence. A set *F* of elements of S is said to be free in S if every element of S is a product of at most one sequence of elements of *F.* A semigroup S is free if it has a free set G of generators. It is easily shown that this is the ease if and only if every element of S is the product of one and only one sequence of elements of G. It is shown that if a semigroup S is free then its set G of free generators is unique.

Given two semigroups S and *T,* a homomorphism of S into *T* is a map h:S $\rightarrow$ *T* with the property that $h(ab) = h(a)h(b)$ for *a* and *b* in S.

REFERENCES

1. Y. Bar-Hillel, "A Quasi-Arithmetical Notation for Syntactic Description," *Language* 29 (1953) 47-58
2. N. Chomsky, *Syntactic Structures* (The Hague, 1957).
3. S. Ginsburg, "Some Remarks on Abstract Machines," *Transactions of the American Mathematical Society* 96 (1960) 400-444.
4. E. Moore, "Gedanken-Experiments on Sequential Machines," *Automata Studies (*Princeton, 1956).
5. M. Rabin and D. Scott, "Finite Automata and their Decision Problems," *IBM Journal of Research and Development* 3 (1959) 114-125.
6. G. Raney, "Sequential Functions," *Journal of the Association for Computing Machinery* 5 (1958) 177-180.