Generalized translation of programming languages

by RONALD W. JONAS
Linguistics Research Center
The University of Texas
Austin, Texas

INTRODUCTION

A generalized model for the computer translation of both programming and natural languages has been developed at the Linguistics Research Center of The University of Texas. The design of the model is discussed here and sample grammatical descriptions are given to illustrate how the generalized translation of programming languages may be accomplished.

Discussion of the model

The linguistic theory of transformational grammar provides convenient terminology for explaining the LRC model. This theory maintains that there is not merely one structure underlying a language, but two: *surface* structure and *basic* structure. Viewed as a generative system, a transformational model is composed of a base component, which generates *basic* tree structures, and a transformational component, which maps these basic trees onto *surface* tree structures. The terminals of the resultant surface trees define strings of some particular language, such as FORTRAN or ALGOL.

If we think of the transformational model as primarily generative, the base may be regarded as the first phase of the process, generating structural trees explicitly stating linguistic relationships underlying language strings. These trees have terminal nodes, but they do not necessarily form a string which is meaningful in any particular language. The representation used by most linguists for the base is an ordered context sensitive grammar. Context sensitivity and ordering together control the type of structures output from the base. Figure 1 shows one such structure.

The transformational component, the last phase of the generative process, is expressed as a grammar having ordered rules in a highly specialized format. These rules specify how to take particular basic trees and reshape them so the terminals of the resultant trees are in the desired order. The resultant terminals must form a string, reading left to right, which belongs to the language for which the transformations are written (e.g. FORTRAN or ALGOL). For example, if the string *abcedf* in Figure 1 is not a string of language X, then a transformation may be specified to yield some surface tree with string *abcdef* belonging to the language X. One result of such a transformation is shown in Figure 2. In this case the transformation specifies that terminal branch *d* is to be moved from node *C* to node *B* of the tree.

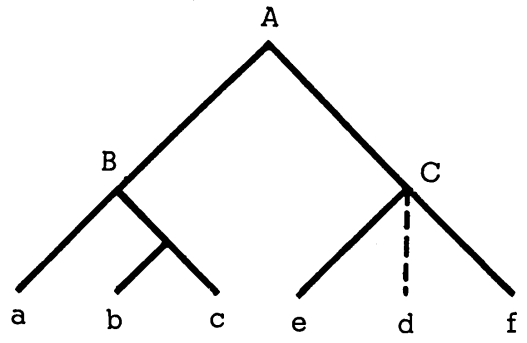


Figure 1—Basic tree

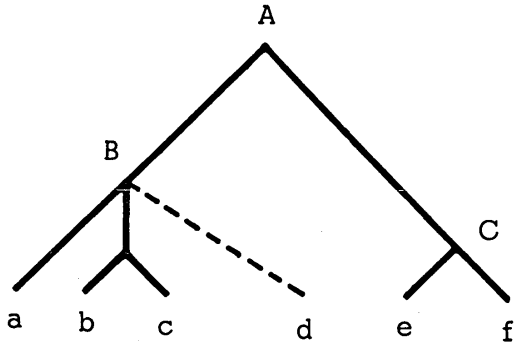


Figure 2—Surface tree

The transformational model details a mapping from base trees onto surface trees for only those parts of the two trees which are not identical, creating a rather economical statement of structure. A minimal statement is achieved for the generative process with base trees (formed by the base component) and a set of transformations (applied by the transformational component), which together yield a surface tree. Since transformations serve simply to reshape the basic tree into a surface tree, the mapping of base structure into identical surface structure is not explicitly stated.

Some linguists maintain that the base component characterizes language-independent organization while the transformational component characterizes language-specific organization. It is tempting to say that the base states semantic relationships and that the transformational component states syntactic relationships, but such a dichotomy is artificial at best. In such a model, syntax, as it is traditionally defined, refers to the structure explicitly stated by both the base grammar and the transformations. Semantics would then be defined as a combination of what is implicitly stated by the syntax and of the theory which led to the particular base and transformational rules in use. The semantics of language is explicit to the extent that the theory specifies algorithmically how to code the base and transformational component grammars. Perhaps it is better to say, for a particular language, that the surface structure *highlights* the syntax while the basic structure *highlights* the semantics.

By virtue of the power available in its pair of hierarchically-arranged grammars, transformational theory provides a system of importance to the mechanical translation of languages. To be sure, transformational theory itself does not solve the problem of translation, for it is concerned only with representing basic trees and mapping them onto the strings of some language(s). Translation must involve a reverse process as well, as suggested by Figure 3. For the compilation of programming languages, this diagram has a special interpretation. The input processes may be viewed as FORTRAN, ALGOL, etc. The output process may be using grammars of programming languages such as viewed as using grammars of particular machine languages such as CODAP, MAP, binary codes, etc.

Although the input and output processes would seem to be inverses of each other, there are some very definite reasons why they are not. The input process is attempting to guess what basic trees underlie the input strings. Box 1 guesses what surface trees are likely to be involved. Box 2 uses this best-guess information to control its guesses as to which basic trees might be involved. Given certain basic trees, the output process merely applies the transformations specified for the target language to yield surface trees (Box 4) and finally synthesizes the surface trees into output strings (Box 5).

There is, nevertheless, much similarity between the input and output processes. Boxes 1 and 5 are each applying surface grammars of their particular languages to produce surface trees and output strings, respectively. The two operations differ in that one is using its grammar to find all possible surface trees that may

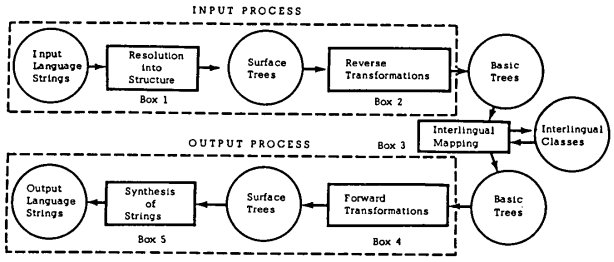


Figure 3—Generalized translation process

lead to the given input string while the other is given a particular surface tree and knows (from its grammar) exactly what string to produce from that tree. Analogously, Boxes 2 and 4 have their grammars of transformations. They differ in that one is trying to find all basic trees underlying the hypothesized surface trees while the other is given a particular basic tree from which to produce related surface trees.

Box 3 defines an interlingual mapping of input language basic trees onto output language basic trees. This process is best regarded as a two-stage operation: a subprocess which maps input basic structure into a set of interlingual classes, and a subprocess which maps these interlingual classes into output basic structure. These subprocesses are driven by grammars which associate basic substructures with certain interlingual classes. Those interlingual classes containing a single basic substructure for each language would provide *exact* translations, while classes containing multiple substructures per language would provide *loose* translations. For programming languages this would reflect the difference, for example, between FORTRAN statements which can be compiled into code only one way (e.g., CALL) and those which may be compiled any number of ways (e.g., A=−(B+C)).

A mechanical translation model much like the one discussed here is operational at the Linguistics Research Center. The specification languages of all boxes are roughly context free grammars, but ones which have operators available for each term of a rule. With these grammars it is possible to define and manipulate a terminal vocabulary in Boxes 1 and 5, to perform a

monolingual mapping between surface and basic trees in Boxes 2 and 4, and to scan trees and associate interlingual classes in Box 3.

Descriptive procedures

To accomplish generalized translation within the framework of this model, it is necessary to define a semantics of computing and to establish grammars suitable for intertranslating procedure-oriented languages, machine-symbolic languages, and machine codes. The present objective is to formulate descriptions which will be suitable for translating procedure-oriented languages and machine-smbolic languages into binary code. If the descriptions prove to be truly generalized, then any-direction translation will be possible; but the current objective is generalized **compilation.**

The usual procedure in generalized compiling is to apply a grammar to the input strings and then to engage semantic routines to interpret the results. The structural trees assigned to the input by the grammar contain nonterminal classifications which control the operation of semantic routines. Careful formulation of the grammars permits the proper semantic effects. The semantic routines build tables of semantic information which eventually yield the necessary machine code.

While certain current compilers have been designed to handle programming language syntax in a general way by accepting syntactic grammars, little has been done to generalize the information contained in the semantic routines of these compilers. The value of the model discussed here is that it permits complete grammatical description of both syntax and semantics. This allows an explicit statement of semantic classes in the form of a grammar rather than the implicit statement afforded by semantic routines. By providing the capability for coding a semantic grammar which captures the semantic information, this model eliminates the necessity for programming in a compiler and puts compilers, instead, completely in the realm of grammatical description of languages.

These are many problem areas requiring investigation before it is possible to write semantic grammars. The principal ones are as follows:

1. Data Representation
   a. Types of data; e.g., floating-point numbers, signed integers, alphabetic strings.
   b. Structural organization; e.g., matrices, lists, files.
   c. Forms of storage; e.g., bit strings, character strings, words, disk files, tape records.
2. Data Representation Operations. These would establish, change, and abolish data representations; exemplary operations: declarations, assignment statements for converting data types.
3. Data Manipulation Operations. This would include all operations upon any of the data defined in 2.
   a. General: add, test, truncate.
   b. Input-output: read, write, print.
   c. Transfers: move block storage.
4. Sequencing Operations. Any operations controlling the flow and synchronization of data manipulation operations would be in this group.
   a. Central processor: conditional and unconditional transfers, index jumps, central processor interrupts.
   b. Input-output: synchronization of independent data units, input-output interrupts.
   c. Console: communication interrupts.

All of these areas have been investigated preliminarily. The remainder of this paper will be devoted to a discussion of the data manipulation and sequencing operations.

As suggested above, discovering and grammatically describing basic trees is of great importance for this translation model. Of equal importance, however, is the necessity for defining the interlingual classes naming equivalent substructures of basic trees (Figure 3, Box 3). It does little good to define the semantics of each language to be translated unless a mapping can be guaranteed between the basic trees. Interlingual classes are used in defining this mapping. When a basic tree is partitioned into meaningful syntactic-semantic units, each block of the partition should be assigned to an interlingual class. If the corresponding basic trees entering and leaving Box 3 (Figure 3) are partitioned into equal numbers of blocks, then interlingual classes may be assigned to the blocks of the partitions in such a way as to define a 1:1 mapping between the input and output basic trees.

These interlingual classes are best defined before semantic grammars of particular languages are coded. Yet these classes can only be defined in a meaningful way by inspecting the types of basic trees they are intended to classify. Both will be discussed together in the following presentation.

Illustrative grammatical descriptions

The basic trees in Figures 4, 5, and 6 illustrate interlingual mappings for ALGOL, MAP, and some imaginary Machine X. Each figure contains basic trees which are equivalent and presumably translatable for the indicated languages. Solid lines define basic trees (such as the ones resulting from the application of grammars in both Boxes 1 and 2 of Figure 3). Broken-line enclosures are interlingual classes assigned to the substructure of the basic trees by the grammar of Box

3; the names of these interlingual classes are indicated as some T_x. The names themselves define the interlingual mapping of basic trees within each figure. See the Appendix for an explanation of MAP mnemonics.

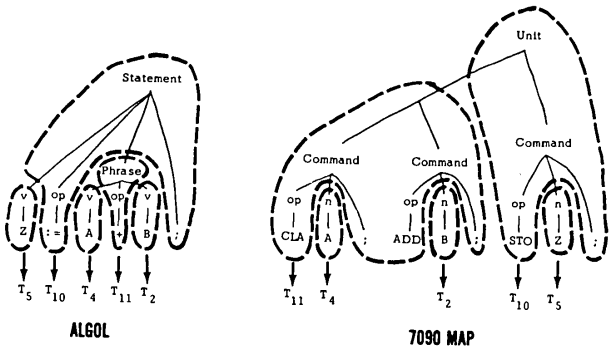


Figure 4—Structural mapping of addition operation

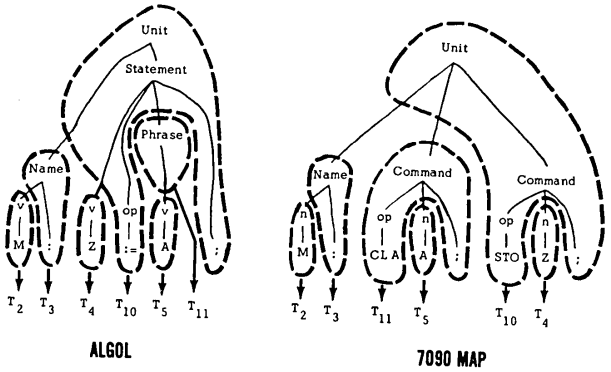


Figure 5—Structural mapping of assignment statement

As Figure 6 shows, the interlingual class T_1 serves to relate four equivalent operations within the three languages represented. Because the IBM 7090 is a single-address machine, more structure is included in the class T_1 for MAP than for either ALGOL or the multiple-address Machine Code X. In the same example, classes T_2,T_4, and T_5 provide an interlingual mapping of the variables involved in programming statements. As suggested in the earlier discussion of the translation model, these classes differ from other interlingual classes in that they may be defined during the translation process by special operators. For example, the intermediate symbols v in the ALGOL structures of the above examples may have associated operators which assign the lexical items *M*, *Z*, and *A* to unique interlingual classes (in this case, T_2, T_4, and T_5, respectively).

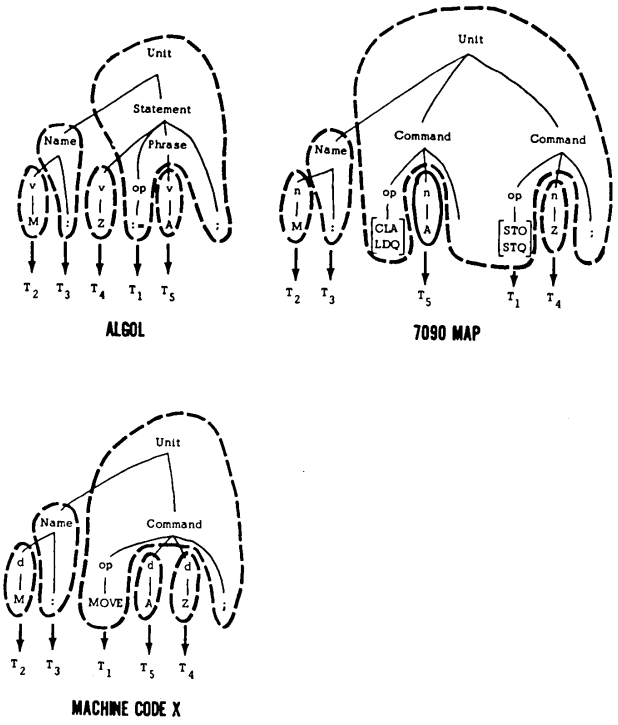


Figure 6—Structural mapping of data transfer operation. Terms within the two pairs of brackets are to be mutually selected. That is, CLA and STO co-occur or LDQ and STQ do. Colons and semicolons do not occur in MAP but are simply added for clarity (with ALGOL interpretations)

The relationships implied by the interlingual class names in these examples might be:

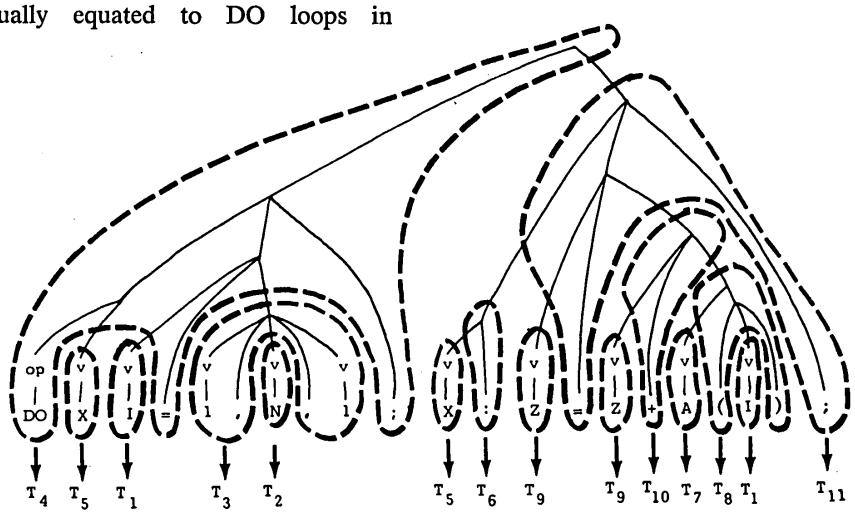
T_1 : Move data "statement"
T_3 : "Statement" name
T_{10} : Store value "statement"
T_{11} : Summation "statement"
$T_{2,4,5}$: map particular data structures from language to language

These classes explicate the semantics implied by the basic trees. For example, anyone who sees the symbol $:=$ in a statement quickly realizes it is an "assignment statement". He has mentally given a gross semantic classification to the part of the structure involving $:=$. One important fact that this overlooks is that the symbol ; was just as important in the definition of *assignment statement* as was $:=$.

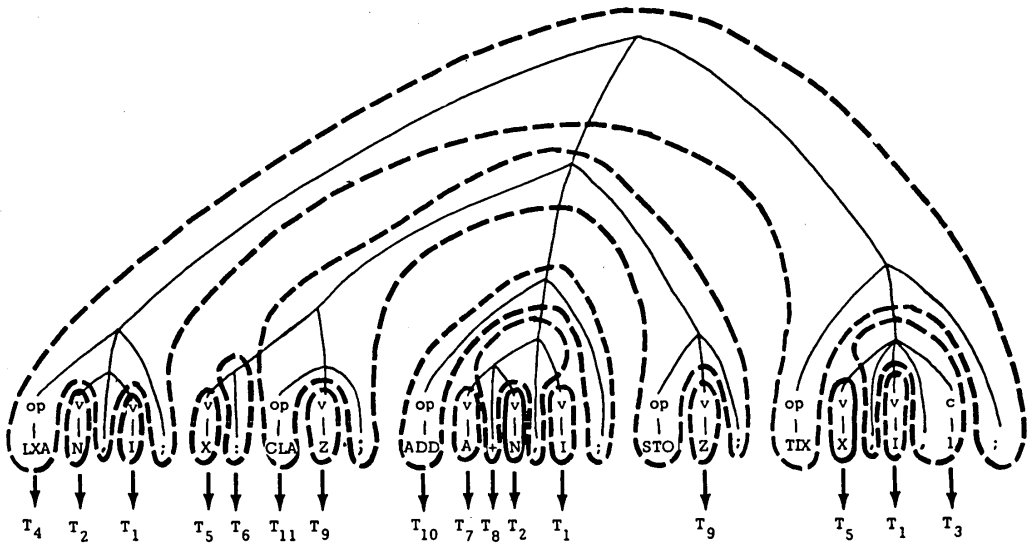
The classes named above are intended to reflect greater refinement of the semantic classifications. "Assignment statement" is too gross in the sense that it unites the semantic distinctions made by interlingual classes T_1 and T_{10}, i.e., the basic difference between moving a block of data into a new location (T_1) and storing the result of a summation (T_{10}). By more exact

isolation of structural information, the grammars which map basic trees into interlingual classes are able to specify explicitly all the semantic distinctions.

As might be guessed, there is no restriction on the size structure which might be included in an interlingual class. The smallest classes will consist of such things as variable names, some punctuation, and rather simple operations. The larger classes may be increasingly inclusive: programming loops in machine language may be interlingually equated to DO loops in FORTRAN or FOR loops in ALGOL; subroutines and even whole programs may become single interlingual classes or a set of such classes. For single interlingual classes to be semantically more inclusive, it is necessary to define larger basic trees. The only limit on such definition is the practical one of how inclusive the grammar is to be. For example, Figure 7 illustrates the basic trees for one possible DO loop to sum the elements of a matrix.



FORTRAN



7090 MAP

Figure 7—Structural mapping of basic trees underlying a DO loop

Clearly the MAP code used in Figure 7 is not the most efficient for representing the DO loop, although it is perfectly valid. This is due to the fact that the basic tree for the MAP statements is isomorphic with its corresponding surface tree. An improved sequence of MAP code is shown in Figure 8. The solid lines represent the surface structure imposed on the code. Dotted lines represent a transformation which maps that surface structure into the basic structure illustrated in Figure 9. Only the interlingual mappings which are different from those of Figure 7 are shown.

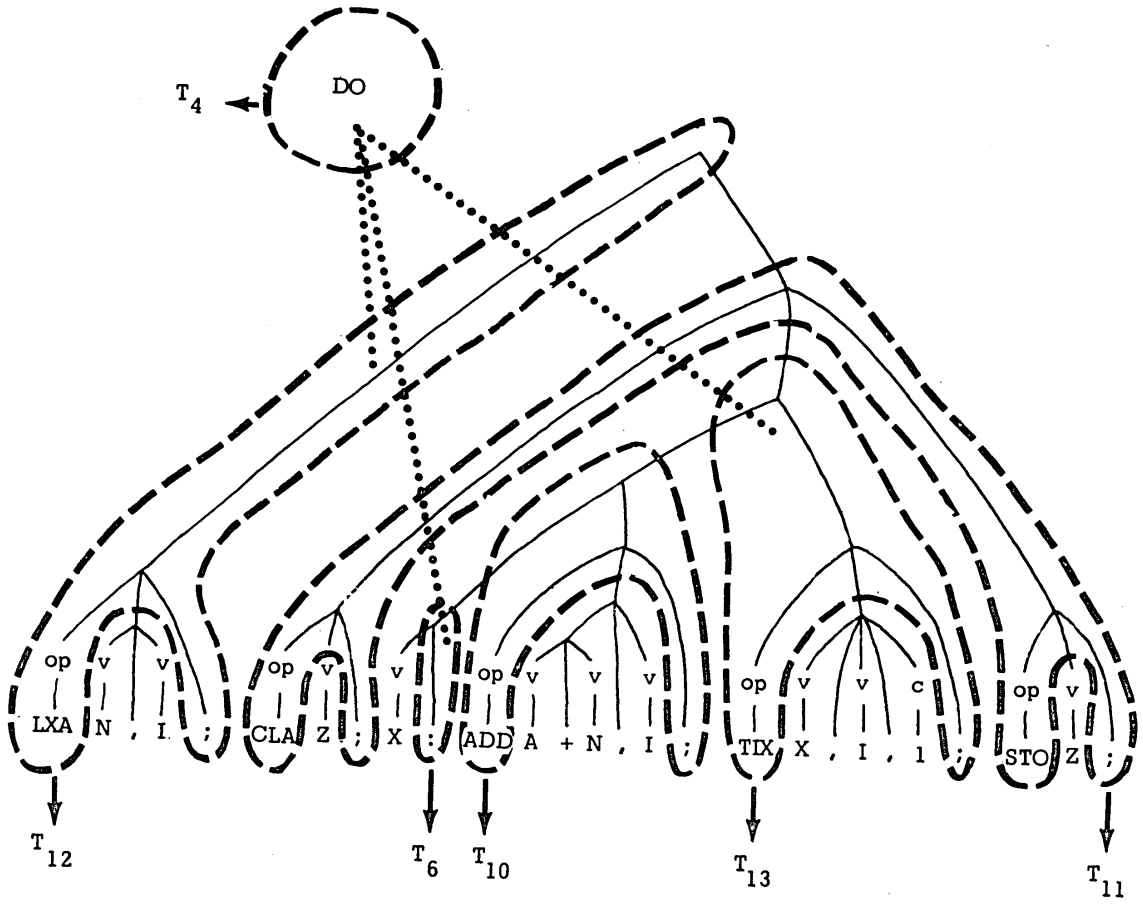


Figure 8—Structural mapping of surface tree for 7090 MAP code underlying DO loop

The surface tree (Figure 8) illustrates the improved MAP code in normal order. The basic tree (Figure 9) demonstrates the semantic functions isolated by the interlingual classes, as follows:

Section α : Opening and closing of DO loop
β : Name of loop
γ : DO loop proper

Clearly the basic tree explicates semantic information which is only implied in the surface tree. This is the motivation for investigating basic trees and interlingual classes rather than surface trees in defining the semantics of computing.

Data manipulation operations

With the advantages of basic trees in focus, it is timely to ask what the semantic classes should be and from what primitive units they should be built. The matter of semantic primitives is of most importance, for these will be the terminals of basic trees. For ALGOL the terminals might be operations such as :=+—/*. For MAP it might be ADD CLASTO, etc. If one were a computer engineer, the primitives of interest might be the operations of switching theory: *and, or, complement* and perhaps *nand, nor, exclusive-or.* Semantic primitives defined in this way appear not to

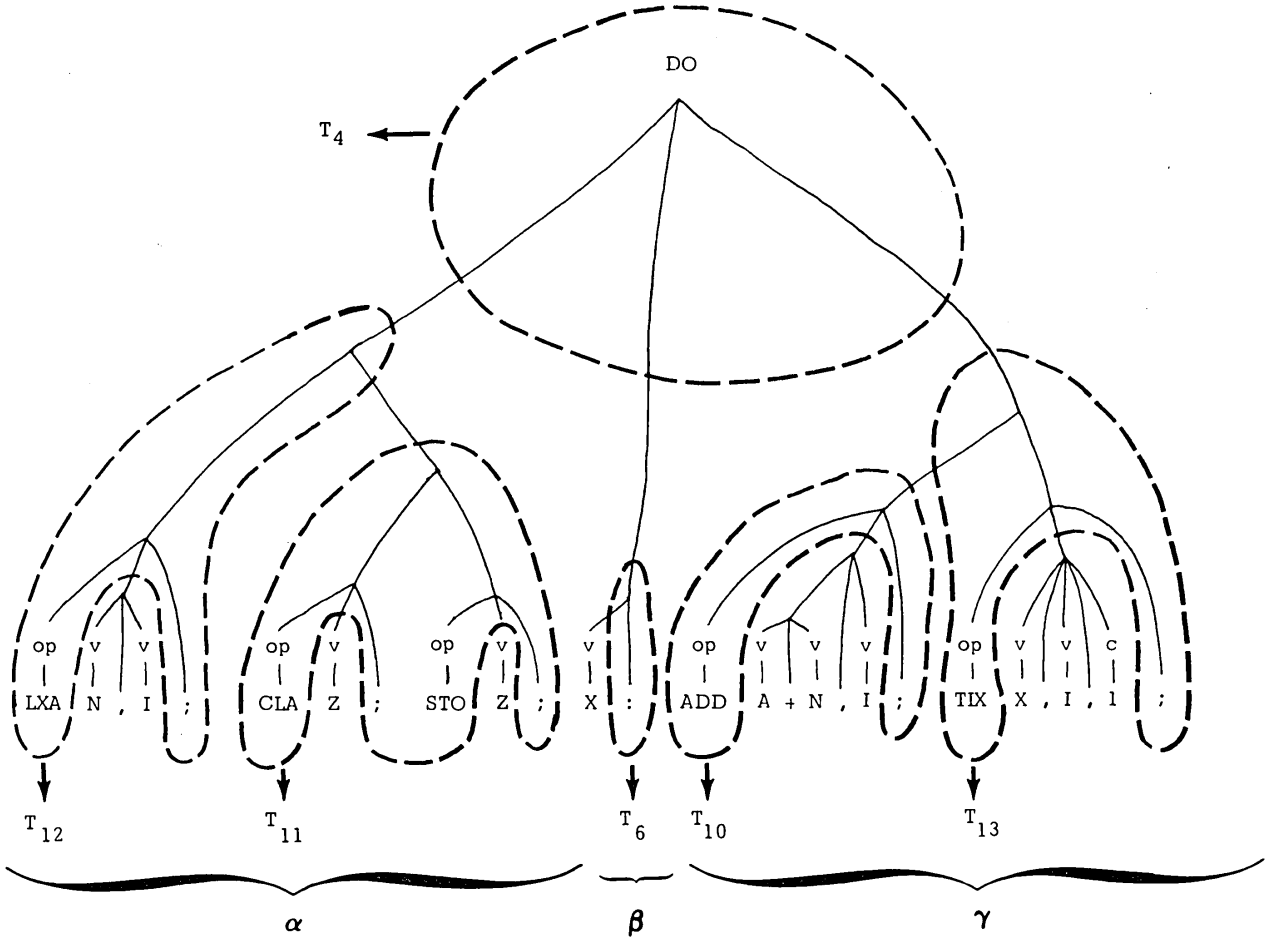


Figure 9—Structural mapping of basic tree for 7090 MAP code underlying DO loop

be generally applicable. One would not, for example, be happy with switching theory primitives as a basis for grammatical descriptions of ALGOL; and the ALGOL primitives would be entirely too gross for a grammatical description of sub-machine operations. In short, a separate set of semantic primitives must be defined for each language to be grammatically described.

Semantic primitives of this type, while useful for describing each language for which they are chosen, appear to have little interlingual value. The problem of mapping the primitives of one language onto those of another remains. This mapping is part of the more general problem of forming interlingual classes. Given that our initial task is the generalized compilation of programming languages, the mapping of primitives is an exercise in discovering the interlingual intersection of primitives for binary code and procedure-oriented or machine-symbolic languages. If we look at interlingual class T_{11} in Figure 4, we see that it contains the primitive operation $+$ for ALGOL. The corresponding MAP code for $\overline{T_{11}}$ is not at all primitive, however. Can the class T_{11} be called primitive? As suggested above, a given interlingual class is likely to be primitive only with respect to one of the two languages being inter-translated. This will be the language whose primitives are larger. Only when the two languages being translated are highly alike will there be any classes which are *primitive* with respect to both languages.

It appears, then, that the more useful process is that of finding, for two languages, which primitives must be defined in terms of sets of other primitives. For example, in Figure 4 the ALGOL structure mapped into interlingual Class T_{11} involves a single primitive. The MAP structure mapped into T_{11} involves the sequence of primitives *CLA ; ADD ;*. We can say

that the ALGOL primitive + is equivalent to the sequence of MAP primitives. This suggests an algorithm for creating interlingual classes; it would be applied to all languages in pairwise fashion: In general, the primitives of the more high-level language would define interlingual classes. The structure of the low-level language would be mapped into these classes as sequences of low-level primitives.

As an illustration of this process, assume we are translating MAP to 7090 binary code. The smallest interlingual classes would correspond to the primitive operation codes of MAP, such as *CLA, STO, ADD,* etc. While there is one complete 7090 binary code for each MAP operation, complete operations would not necessarily be the best primitives used in the grammar of the 7090. The 7090 operation code values reflect rather clearly the suboperations of the computer, and it is these suboperations which best serve as monolingual primitives of the binary code grammar. Table 1 is a partial list of MAP operations and the corresponding 7090 binary codes. The primitive suboperations indicated in Table 1 are explained in Table 2.

Description	Mnemonic	Octual Code	Binary Code
Multiply	MPY	0200	0 0 0 | 0 1 0 0 0 | 0 | 0 | 0 0
Variable-length multiply	VLM	0204	0 0 0 | 0 1 0 0 0 | 0 | 1 | 0 0
Divide	DVH	0220	0 0 0 | 0 1 0 0 1 | 0 | 0 | 0 0
Variable-length divide	VDH	0224	0 0 0 | 0 1 0 0 1 | 0 | 1 | 0 0
			D
Floating add	FAD	0300	0 | 0 0 | 0 1 1 0 0 | 0 | 0 | 0 0
Unnormalized floating add	UFA	4300	1 | 0 0 | 0 1 1 0 0 | 0 | 0 | 0 0
Floating add magnitude	FAM	0304	0 | 0 0 | 0 1 1 0 0 | 0 | 1 | 0 0
Floating subtract	FSB	0302	0 | 0 0 | 0 1 1 0 0 | 0 | 0 | 1 0
Unnormalized floating subtract	UFS	4302	1 | 0 0 | 0 1 1 0 0 | 0 | 0 | 1 0
Floating subtract magnitude	FSM	0306	0 | 0 0 | 0 1 1 0 0 | 0 | 1 | 1 0
Add	ADD	0400	0 | 0 0 | 1 0 0 0 0 | 0 | 0 | 0 0
Subtract	SUB	0402	0 | 0 0 | 1 0 0 0 0 | 0 | 0 | 1 0
			F *E* *B*
Place accumulator address in index	PAX	0734	0 | 0 0 | 1 1 1 0 1 | 1 | 1 | 0 0
Place accumulator decrement in index	PDX	4734	1 | 0 0 | 1 1 1 0 1 | 1 | 1 | 0 0
Place accumulator address in index comp.	PAC	0737	0 | 0 0 | 1 1 1 0 1 | 1 | 1 | 1 1
Place accumulator decrement in index comp.	PDC	4737	1 | 0 0 | 1 1 1 0 1 | 1 | 1 | 1 1
Place index in accumulator address	PXA	0754	0 | 0 0 | 1 1 1 1 0 | 1 | 1 | 0 0
Place index in accumulator decrement	PXD	4754	1 | 0 0 | 1 1 1 1 0 | 1 | 1 | 0 0
			C *A* *G*

Table 1. Partial list of operation codes for IBM 7090 computer. Certain suboperations are enclosed in boxes.

A Basic operation: multiply, floating point add-subtract, fixed point add-subtract, load into index, load from index

B Suboperation: add, subtract

C Word segment: address, decrement

D Length: word length, variable length

E Signing: signed value, magnitude

F Normalization

G Complementation

Table 2.'—Primitive suboperations chosen for IBM 7090 computer.

Clearly, the binary code is readily partitioned and yields a better set of primitives for the grammar of the machine than the full operation code. If each MAP operation defines an interlingual class, then the chart reveals the set of 7090 binary primitives which maps into each interlingual class. For example, the MAP operation *FAD* would be interlingually mapped together with the binary primitives *A. B. E, F* to cause translation into the complete binary operation 0300. An ALGOL operation would be mapped quite differently into the primitives, however. For example, the ALGOL addition operation makes no explicit statement about the variety of addition chosen. In the statement X : = Y + Z, the operation + would map into only the binary primitive *B* (Table 2). It would not involve the choice offered by primitives *A, E, F;* i.e., floating point vs. fixed, signed vs. unsigned, normalized vs. unnormalized. These parts of the binary operation could only be translated from information interlingually mapped with the variables X, Y, Z.

Sequencing operations

The grammatical description of sequencing operations is equally as important as the description of data manipulation operations. Such a description must account for the order in which data manipulation operations are executed and in which registers of the computer are addressed by these operations. The formulation of the description depends, once again, upon the definition of semantic primitives for sequencing.

In a computer like the IBM 650, every command explicitly states the location of the next command by an overt address. But in contemporary binary computers, sequencing is determined by a meta-process. That is, except for certain branching commands in the computer's repertoire, next addresses are determined by an addition process. There is an Address Register, much like the accumulator, which is stepped as each command is operated in order to cause sequencing to proceed linearly through memory. This process is depicted in Figure 10. It illustrates three alternate types of sequence control, which are characterized by the basic structures in Figure 11.

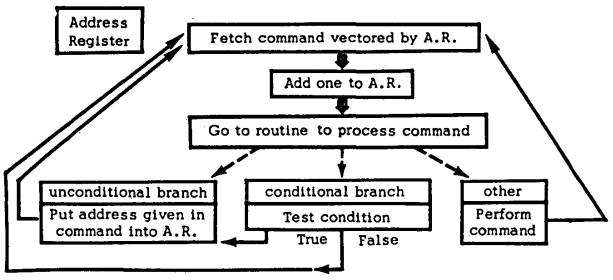


Figure 10—Conceptualization of meta-addressing in IBM 7090 computer

Ordinarily when we talk of computer operations, we refer only to what is going on in Boxes C (Figure 11). All of the hidden performance which goes into the processing of a sequenct of operation, however, is represented by the rest of the basic trees. These trees suggest a level of control in the computer which is superordinate to that of the operations proper, namely the control of sequencing.

While the above trees account for only the simplest variety of sequence control, related ones are capable of explaining a more difficult sequence phenomenon: interrupts. Each interrupt available on a computer is linked with some device which operates independently of the central processor. The key to the independent operation of each device and the behavior of its interrupt lies in Address Registers like the ones referenced in Figure 10 and 11. Each independent device may

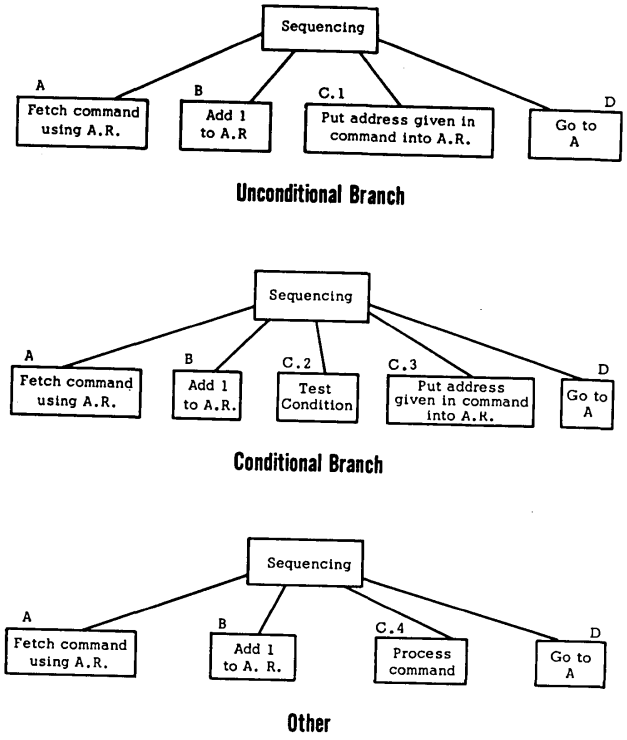


Figure 11—Basic structure for three types of sequence control in the central processor of the IBM 7090 computer

have an Address Register of its own to control its own sequencing. It may likewise have Address Register controls much like those in Figure 11, Boxes A and B. The device will only need such a control if it has a sequence of suboperations to perform. In any event, these devices are all coordinated with the central devise via its (central) Address Register.

As indicated in Figure 11, after each operation of the central processor is performed (Boxes C), control is returned to the central Address Register. Any time some independent device needs the services of the central processor, it may *interrupt* by altering the contents of the central Address Register while Box C is in progress for the central processor. Upon arriving at Box D (and subsequently A), the central processor will automatically be redirected by the contents of its altered Address Register. Therefore, we may account for even the interrupt behavior of computers by positing a basic structure for each interrupt. For example, the interrupt behavior of a typical independent device A might be characterized as in Figure 12.

A solution to the more general problem of addressing memory follows from this. The important consideration is that arithmetic may be performed on the

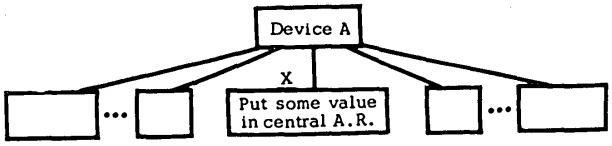


Figure 12—Basic structure underlying the interrupt behavior of Device A. Box X causes central processor control to be diverted

reference addresses themselves, not merely the data they reference. For example, every reference has potentially a base address and an index, which, when added together (or subtracted), yield the *effective* memory address. The execution of every command (Figure 1, Boxes C) having such an indexable address may be regarded as having a subcycle in which this effective address is computed, as shown in Figure 13. The 7090 has another such meta-operation. When more than one index is referenced by the same command, the logical sum of the indexes is taken and the result is then added (or subtracted) to the base address. This would require another subcycle on the far left of Figure 13 which accounted for the summing of indexes.

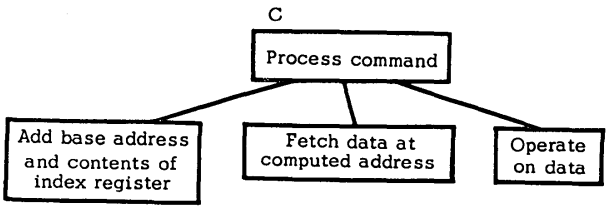


Figure 13—Basic structure underlying address indexing

In general, the grammars accepted by current compilers express relationships only within individual programming statements of the language being described. The relationships implied by the order in which these individual statements are executed is not explicitly coded. Compilers have not required or used this variety of information in the grammars. Data manipulation operations, on the other hand, involve well-defined functions for which convention has come to dictate fairly final grammars. Until grammars are written which state explicitly *all* the features of programming languages, *including* sequencing, translation of programming languages will not be general enough to be altogether semantically satisfying.

Optimization

The foregoing discussion has revealed that it is not enough merely to look at procedure-oriented languages as a means to formulating a theory of computing. The features of semantic grammars for procedure-oriented languages and for binary codes must be highly correlated if translation is to be realized. Such correlation is complicated by the fact that there is not always a single *best* way to map a procedure-oriented language such as ALGOL into the binary code for a particular computer. For example, one can define multiplication or division by powers of 2 in terms of either arithmetic or bit-shifting operations. A generalized compiler should be able to select either mapping. Furthermore, one would expect the compiler to select the code which leads to the shortest execution time for the given computer. If bit-shifting operations are faster but arithmetic operations are more general, then clearly the interlingual classes should include *both* to achieve the best translation of all multiplication and division statements into machine code. Inclusion of all such alternatives should be an objective for good semantic grammars.

SUMMARY

A model for language translation has been presented as a means for compiling computer languages. While in the same class as compiler building systems, it offers the advantage that the specification of programming languages and computer characteristics may be accomplished completely grammatically. In addition to providing generalized descriptive methods, the model offers the potential of free inter-translation of languages once grammars are coded.

An approach to the writing of grammars for programming languages is outlined. The suggestion is made that currently available syntactic grammars may be considered adequate and that effort should be put into the coding of semantic grammars. Major problem areas requiring investigation to this end are: ways of representing and storing data, operations for data representation and manipulation, and sequencing control. Exemplary grammatical descriptions are shown.

Particular attention is given to the description of data manipulation and sequencing operations. The discussion includes consideration of both the monolingual and interlingual problems of grammars for procedure-oriented and machine-symbolic languages and binary machine codes. The definition of a semantics of computing involves, consequently, not only determining what machine code is substituted for programming statements but also what the meanings of statements and sequences of statements are in both languages and what their common structural intersection is (interlingual classes). This has led to some tentative conclusions about what units are needed in a semantic theory in order to completely describe programming languages.

APPENDIX: IBM 7090 MAP Operations

Mnemonic	*Explanation*
ADD X	Add contents of X to accumulator
CLA X	Clear accumulator and add contents of X
LDQ X	Load Q register with contents of X
LXA X, Y	Load index register Y with address of word at X
STO X	Store accumulator contents at X
STQ X	Store Q register contents at X
TIX X, Y, Z	Reduce index register Y by Z amount; if result is 0, transfer to location X

BIBLIOGRAPHY

1 E W BACH
An introduction to transformational grammars
Holt Rinehart and Winston Inc New York 1964

2 N CHOMSKY
Aspects of the theory of syntax
The MIT Press Cambridge 1965

3 R S GAINES
On the translation of machine language programs
Communications of the ACM 8:736-741 December 1965

4 *IBM 7090 Principles of operation*
IBM Systems Reference Library File No 7090-01 Form A22-6528-5 August 1963

5 R K LINDSAY
Inferential memory as the basis of machines which understand natural language
Computers and Thought Edited by E. A. Feigenbaum and J Feldman McGraw-Hill Book Co New York 1963 pp 217-233

6 J D McCAWLEY
Concerning the base component of a transformational grammar
Ditto Revised University of Chicago August 16 1966

7 M R QUILLIAN
Semantic memory
Bolt Beranek and Newman Inc Cambridge Scientific Report no 2 October 1966

8 W A SASSAMAN
A computer program to translate machine language into FORTRAN
AFIPS Conference Proceedings 28:235-239 Spring Joint Computer Conference 1966

9 *Thirteenth quarterly progress report*
Linguistics Research Center The University of Texas at Austin pp 22-34 1 May 1962-31 July 1962

10 A M ZWICKY et al
The MITRE syntactic analysis procedure for transformational grammars
AFIPS Conference Proceedings 27:317-326 Fall Joint Computer Conference 1965