ENGLISH FOR THE COMPUTER

Frederick B. Thompson

California Institute of Technology
Pasadena, California

What about English as a programming language? Few would question that this is a desirable goal. On the other hand, I dare say every one of us has rather deep reservations both about its feasibility and about a number of problems that it entails.[1] This paper presents a point of view which gives some clarity to the relationship between English and programming languages. This point of view has found substance in an experimental system called DEACON. The second paper in this session will describe the specific DEACON system and its capabilities.

There is one source of these reservations that we should recognize, and that is the fact that we have no adequate notion of the nature of natural language and no precise description of its vagaries. It is for this reason that most of those working on language problems have concentrated on programming languages or confined themselves to syntax. However, the semantics of natural language pose important problems. These remarks are related to those problems.

The excellent work that has been done on programming languages, in particular on syntax-directed compiling and its associated semantics, and work in the area of symbolic logic have cast much light on the natural language problem as well. It has illuminated some very real difficulties. It has also illuminated some aspects which can be exploited to good ends. And more important, it has allowed the separation of the deep difficulties of dealing with natural language from some of these exploitative opportunities. We shall build upon this work as well as on recent work in linguistics.

What can be said about English as a computer language? There are certain aspects of a very difficult nature that are involved in a full-blown natural language, namely the fact that it is self-referent. In English, we can speak of English; we are doing so at this very minute. We can say such things as: "John believes Mary lies." Worse, we can say: "This sentence lies." And we are all aware of the implications of this fact as discussed by Tarski,[2] Gödel, and Turing.[3] When we think of the possible uses of English as a computer language, we realize that little will be lost if we abandon those parts of English which are self-referent or involve indirect discourse. Let us do so.

Much of the world's knowledge is in written form. Computers are being applied to the processing of such documentary information and are being programmed to do some of the more routine functions of the research librarian. To this end they must indeed have a certain understanding of English. Good work is being done in this direction, for example, the work of Robert Simmons.[4, 5] However, the use of computers to intelligently service documentary material is quite distinct from the use of English as a computer language.

Thus I would like to focus our attention on the use of English to:

1. input information into a computer,
2. instruct the computer to process the information that it has stored away, and
3. query the computer concerning the information it has stored away, and which results from processing.

These are the functions that programming languages perform.

Usually when we think of English, we are tempted to include the traditional patterns of syntactic analysis. But the parts of speech—noun, adjective, verb—are not a part of English, but rather a method that grammarians imposed long ago in their attempts to understand the regularities of structure that are apparent in language. Modern linguists, in their study of syntax, have ramified, redefined, and modified these traditional categories, well aware that they are, at best, an imperfect tool for understanding the structure of language. We shall feel free, therefore, to choose our syntactic categories in whatever way is useful in our analysis, and shall feel no compunction to stick to the traditional parts of speech.

In programming languages, we also find syntactic categories: operator, label, subscripted integer variable, etc. In the formal expression of the syntax of a programming language these categories are used in a fashion parallel to the use of parts of speech in a phrase structure grammar for English.

Typical phrase structure rule for English:

<Verb phrase>::=<Verb><Noun phrase>

Typical phrase structure rule for a programming language:

<real expression>::=<add op><real factor>

However, there is a striking difference. The parts of speech of traditional linguistics do not have semantic implications beyond that made explicit by the rules of grammar. They can be fully characterized as nonterminal symbols which are used in expressing the recursive relationships of English structure. In times past, loose attempts to define these parts of speech in terms of meaning have been made. However, since Bloomfield such attempts have fallen into disrepute.

In sharp contrast, the syntactic categories used in the description of programming languages have clear semantic implications. An integer variable and a real variable designate two quite distinct entities, independent of how these variables are used syntactically in program statements. In FORTRAN, for example, to say that an expression is a doubly subscripted variable implies a good deal about the associated material in memory, namely that it is a two-dimensional array stored column after column contiguously. The part of speech of a word used in a programming language carries clear structural implications for the way the corresponding material is stored in memory.

The work of Irons,[6] and of those that have followed him in the development of syntax-directed compiling, has exploited this relationship and indeed has gone much further. With each rule of grammar there is associated a corresponding segment of code which expresses the operations on memory structures implied by a grammatical phrase to which the rule applies. The syntactic analysis of a program statement in terms of these rules of grammar provides the necessary directions for compiling these segments of code into a computer program which expresses the semantic context of the statement. One of the most elegant formulations of this point of view has been given by Wirth and Weber in their paper: "EULER: A Generalization of ALGOL, and its Formal Definition."[7]

The following definitions of a formal language are a straightforward generalization of these developments, for example, of the definitions given in the Wirth-Weber paper.

> A *syntax* is an ordered quadruple (V, Φ, B, s) where V is a vocabulary; Φ is a finite set of syntactic rules ϕ_i (these rules may be assumed to be of the form $x \rightarrow y$, where x and y are strings from V); B designates the terminal symbols, a subset of V; and s is an element of $V - B$ (which can be thought of as the part of speech "sentence").

The rule $x \rightarrow y$ is to be read "the substring x may be rewritten as y." Thus it permits a string $w\ x\ z$ to be rewritten as $w\ y\ z$. If a string u can be transformed into a string v by successive rewritings of substrings according to the syntax rules, then u is said to produce v; in symbols, $u \overset{*}{\rightarrow} v$. In a derivation such as $u \overset{*}{\rightarrow} v$, a sequence $(\phi_1, \phi_2, \ldots, \phi_m)$ of syntax rules is applied. The inverted sequence (ϕ_m, ϕ_{m-1}, . . ., ϕ_1) is called a parse of v from u. A string x is a sentence if $s \overset{*}{\rightarrow} x$, and all of its symbols are in B, i.e., are terminal symbols.

If all the rules are of the form $x \rightarrow y$, and x is always a single element of V, the syntax is called a context-free phrase structure syntax. Although context-free phrase structure grammars are convenient to work with, it is known that they are not adequate to describe current programming languages, nor do they appear at all adequate for description of natural language. On the other hand, it is known that any language whose sentences are recursively enumerable has a syntax as defined above, i.e., has a Post production grammar[8]; thus our definition is as general as one would desire. In practice, one may wish a more complex form of syntactic rule—one that specifies more completely the character of the strings x for which a substitution may be made. Such rules will be discussed at length below.

The terminal symbols B can be divided into two parts: $B = F \cup R$. R, the referent symbols, are those which refer to specific values. Typically, variables and constants are referent or English words such as "house" and "red." F, the function symbols, are exemplified by delimiters or by the English words "and" and "all." They play a quite different role from referent words, as will be seen below.

The referent words of a language are differentiated by the type of objects they may denote. In programming languages, referent symbols include integer variable, real two-dimensional array variable, list name, function name, etc. Moreover, phrases (derivations from referent symbols) may also be differentiated by the types of objects they denote. Thus in FORTRAN, not only do J and I denote integers, but so also does $J + I$; in LISP, not only are A and B list names but so is $(A\ B)$. When we examine the rules of syntax for a programming language, we find that the nonterminal symbols appearing in these rules are names for these categories of objects which the corresponding referent symbols or phrases may denote. They may also contain certain syntactic information (for example, the difference between a term and a factor), but there is indeed a relationship which relates each nonterminal symbol to a category or group of categories of objects which the referent symbols and phrases may denote. These categories are the environment for the language.

An *environment* E is a finite set (C_1, C_2, . . . , C_n) of categories of memory structures. The C_i need not be disjoint.

For example, the environment for FORTRAN is the set of integer and floating point scalars, one-, two-, and three-dimensional arrays, and Boolean elements.

An *interpretation rule* ψ defines an action (or sequence of actions) involving the objects of an environment E. This formalizes a semantic counterpart of syntax.

A *formal language* L is a septuple (V, Φ, F, R, s, Ψ, E), where

a. (V, Φ, $F \cup R$, s) is a syntax, and
b. E is an environment.
c. There is a correspondence (possibly many-many) between the symbols of V—($F \cup R$) and the categories of E.
d. There is a correspondence between R and objects of the categories of E, thus establishing the initial values of referent symbols.
e. Ψ is a set of interpretation rules such that a one-one mapping exists between elements of Ψ and Φ, and E is the environment for elements of Ψ.

To be complete, somewhat more than this must be said about the relationship between syntax rules and interpretation rules. We illustrate this with an example:

Rule: ϕ: $a_1a_2a_3 \rightarrow b_1b_2b_3b_4$
Suppose: b_i corresponds to $C_i \in E$
a_i corresponds to $C'_i \in E$
ϕ corresponds to the interpretation rule ψ.

Then ψ is on $C_1 \times C_2 \times C_3 \times C_4$ to $C'_1 \times C'_2 \times C'_3$. In this example, we have assumed neither side of the rule ϕ contains function words. Function words, being nonreferent, do not enter into the determination of the arguments or values of ψ.

We are now in a position to define the meaning of a sentence of L. The *meaning* $M(x)$ *of a sentence* x *of* L is the effect of the execution of the sequence of intepretation rules ψ_1, ψ_2, . . . , ψ_m on the environment E, where ϕ_1, . . . , ϕ_m is a parse of the sentence x into the symbol s, and ψ_i corresponds to ϕ_i for all i.

I should like to rephrase certain of these notions in diagrammatic form to make clear certain of their interrelationships. Let the environment E consist of the categories $C_1, C_2, \ldots, C_k$. Then the relationship between a referent word or phrase x and its value X can be shown by the following commuting diagram, where $p \varepsilon V - (F \cup R)$ is the part of speech of x and $C \varepsilon E$ is the category associated with p.

$$\begin{tikzcd} x \arrow[r] \arrow[d] & p \arrow[d] \\ X \arrow[r, "\varepsilon"] & C \end{tikzcd}$$

Consider now the case of a context-free phrase structure rule of grammar ϕ: $q \rightarrow p_1 p_2 \ldots p_n$. Suppose we have a string $x_1 \ldots x_n$ where each x_i is a string of terminal symbols and has previously been parsed to p_i, i.e., $p_i \overset{*}{\rightarrow} x_i$. According to the above definitions, there is an interpretation rule ψ corresponding to ϕ such that the following diagram commutes.

$$\begin{tikzcd} x_1 \arrow[d] & x_2 \arrow[d] & \ldots & x_n \arrow[d] \arrow[r] & (\phi\colon q \arrow[d] \arrow[r] & p_1 \arrow[d] & p_2 \arrow[d] & \ldots & p_n) \arrow[d] \\ \psi(X_1, & X_2, & \ldots, & X_n) = Y \arrow[r] & (\psi\colon C & C_1 \arrow[l] \times & C_2 \times & \ldots \times & C_n) \end{tikzcd}$$

where X_i is the value denoted by x_i and X_i, as a memory structure (such as an array or list), is in the category C_i. The top half of the diagram shows that the string $x_1 \ldots x_n$ can be further parsed by ϕ, i.e., $q \overset{*}{\rightarrow} x_1 \ldots x_n$. Correspondingly, the value denoted by $x_1 \ldots x_n$ is $Y = \psi(X_1, \ldots, X_n)$. The interpretation rule ψ is shown as a functor that maps the Cartesian product of the categories $C_1, \ldots, C_n$ into the category C.

A more general diagram for a noncontext-free, Post production rule is shown as follows:

$$\begin{tikzcd} x_1 \arrow[d] & x_2 \arrow[d] & \ldots & x_n \arrow[d] \arrow[r] & (\phi\colon q_1 \arrow[d] & \ldots & q_m \arrow[d] \arrow[r] & p_1 \arrow[d] & \ldots & p_n) \arrow[d] \\ \psi(X_1, & X_2, & \ldots, & X_n) = (Y_1, \ldots, Y_m) \arrow[r] & (\psi\colon C'_1 \times & \ldots \times & C'_m & C_1 \arrow[l] \times & \ldots \times & C_n) \end{tikzcd}$$

What conditions must be placed on the interpretation rule ψ? Considering the matter from the point of view of syntax-directed compiling, it certainly must be the case that the definition of ψ is independent of the particular values X_i and depends solely on the character of the categories C_i of memory structures to which it applies. For example, the code compiled for an arithmetic expression $I + J$, where I and J are integer variables, depends only on this fact that they are integer variables and not upon their particular values. The ψ must be defined in terms of the structural aspects of the categories alone. Further, ψ should be constructive, i.e., there should be an algorithm for computing $\psi(X_1, X_2, \ldots, X_n)$ whenever $X_1, X_2, \ldots$ are in the appropriate categories. A general definition of "interpretation rule" can be given satisfying these *two* requirements, along either the programming line following McCarthy[9] or constructive set theory following Gödel[10]; the details however would take us too far afield here. We shall simply speak of an interpretation rule ψ as being structural and constructive.

Now we come to the point of the matter. The above two diagrams show that the domain of definition of ψ is the whole of the Cartesian product $C_1 \times C_2 \times \ldots \times C_n$. There is no particular need for this stringent a requirement. Its domain of definition may be some appropriate subset $K \subseteq C_1 \times C_2 \times \ldots \times C_n$. However, just as ψ itself must be defined in terms of the structural aspects of the C_i alone, so also must this subset be identified by restrictions of a similar character. A particular important class of such restrictions are those which refer not only to the parts of speech p_i and their associated categories C_i but also to the existence of certain parsings of the strings x_i and the categories associated therewith. Such rules are of particular importance because the restriction on their domain of application can be stated in terms of parsings of the constituent x_i strings and thus stated in purely syntactic terms. Such rules of grammar for natural languages have been identified by Chomsky and Harris who have correctly stressed their importance.[11–13] These are the transformation rules. The importance Chomsky gives to the concomitant transformation of the phrase marker (roughly: parsing tree), as well as his condition of the substitutability of strings in elementary transformations, can be seen in the above terminology to insure that the restriction on the domain of ψ is indeed dependent only on questions concerning categories and not on particular values involved (see in particular pp. 300–3 of Ref. 12). Such a condition, we have seen, is exactly the one necessary to insure compilable code in a syntax directed compiler.

An example at this point may be in order. Consider the situation where one wishes to analyze the sentence "John saw Mary and Joan" into the two sentences: "John saw Mary. John saw Joan." Notice that the rule: *NVN.NVN.→NVN and N.* is not adequate, for it does not signal the condition that the first and fifth constituents (namely "John," and "John" in the example sentence) must be identical. This extra condition cannot be simply stated in phrase structure form but is easily and correctly stated as a transformational rule. The passive transformation, $N_1VN_2 \to N_2$ aux V by N_1, is another example where an extra condition is necessary to correctly identify the switched positions of subject and object. Indeed, in the formation of many transformation rules, it is desirable that the rule be applied to the entire sentence where the restriction of the domain of the transformation is stated in terms of an analysis of the structure of the sentence. In this case, the phrase structure aspect of the rule takes on the trivial form $s \to s$ (Ref. 12, pp. 300–303). It is interesting to conjecture that the use of such rules in defining programming languages might well permit the statement of rules covering parentheses conventions in arithmetic expressions without the introduction of superfluous parts of speech as is now done. The compiling time such complex rules entail would, of course, not warrant the change.

The final diagram, encompassing transformational rules, can thus be shown:

$$\begin{tikzcd}
x_1 \arrow[d] & x_2 \arrow[d] & \ldots & x_n \arrow[d] \arrow[rr] & & (\phi\colon q_1 \ldots q_m \to p_1 \ldots p_n) \\
\psi(X_1, & X_2, & \ldots, & X_n) & = (Y_1, \ldots, Y_m) \to & \text{with side condition} \\
 & & & & & (\psi\colon C'_1 \times \ldots \times C'_m \leftarrow K \subseteq C_1 \times \ldots \times C_n)
\end{tikzcd}$$

The explanatory power of the approach presented here can be seen to greater advantage by starting with the semantic aspects rather than the syntax. Let us focus our attention for a moment on the memory of the computer. Considering it independently from any particular program or programming language, it is difficult to say whether it contains any fixed point variables, arrays or list structures. But it unmistakably has a complex, interknotted web of structure. Now consider a structural, constructive interpretation rule ψ, say taking n arguments (where each of its arguments may be considered as an address in memory). We note that the value $\psi\ (X_1, X_2, \ldots, X_n)$ obtained by application of the rule ψ depends on the structure of memory "local" to $X_1, X_2, \ldots, X_n$. This statement follows from the structural and constructive nature of ψ. In fact, using the particular definition of ψ, we can characterize certain structural categories $C_1, C_2, \ldots C_n$. If $X_i \in C_i$, $i = 1, \ldots, n$, that is if the structures in memory which can locally be reached from the X_i have the determined characteristics, then we can determine from the definition of ψ precisely what the value $\psi\ (X_1, \ldots, X_n)$ will be, independent of the rest of memory. (On arguments which do not have these structuarl characteristics, i.e., $X_i \in C_i$ is not true, we can not predict what ψ will do; thus if a program applies a list processing operation to a "non-list" address, the resulting indeterminacy is characterized as a bug).

Suppose we start with a finite number of interpretation rules $\psi_1, \psi_2, \ldots, \psi_m$. From there we can determine as above a finite number of structural categories $C_1, C_2, \ldots C_k$ such that the domains of the ψ_i are subdirect products of the C_j's, that is, the domain of ψ_i is not only a subset of $C_{i1} \times C_{i2} \times \ldots \times C_{in_i}$, but can be characterized structurally in terms of relationships among elements identified in the definitions of the C_{ij}'s.

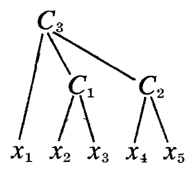
Suppose further that certain arguments $X_1, X_2, \ldots, X_f$ are initially given. We now ask what arguments can be reached from the X_i by application and composition of the functions ψ_j? What is computable? We shall answer this question by relating interpretation rules, categories and initial arguments to our previous discussion.

Correspond to each of the X_i a referent word, or formative; this will be our referent vocabulary R. The categories C_i will constitute the vocabulary $V - (R \cup F)$. If the domain of a rule ψ_i is a subdirect product of $C_{i1} \times \ldots \times C_{in_i}$ we will adopt a transformational rule of grammar establishing "$C_{i1}C_{i2} \ldots C_{in_i}$" as a grammatical phrase subject to the structural side condition.

Composition of interpretation rules applied to appropriate arguments can now be seen to have as an exact counterpart the parsing of the corresponding string of formatives. For example (in the case of context-free rules where it is easiest to see):

$$\psi_1(X_1, \psi_2(X_2, X_3), \psi_3(X_4, X_5))$$

corresponds to the parsing tree



where x_i is the referent word corresponding to X_i. Thus those arguments in memory which can be reached starting with the X_i by using functional composition of the interpretation rules are exactly those which can be defined in the corresponding formal language.

It is the underlying structural, constructive interpretation rules on memory which are at the heart of language. From these, the rest including syntax can all be reconstructed. The expressiveness of a formal language reduces to what can be reached from the references of its words by functional composition of its interpretation rules.

Before going on, let us pause to consider the role of function words. According to the definition of meaning given above, a sentence may have multiple meanings, i.e., be semantically ambiguous. This may arise when a sentence has two parsings (though this by itself does not necessarily imply semantic ambiguity). A typical case of ambiguity would occur if parentheses were dropped from all arithmetical expressions. Consider, for example, the expression $I + J \times K$. By convention we assume the multiplication is to precede the addition. If the addition were to be done first, delimiters would be inserted: $(I + J) \times K$. It has been shown by David Benson that any syntax can be made unambiguous through appropriate augmentation by function words, and this in such a way that no possible meaning (in the above sense) will be lost. Thus function words are seen as a device for reducing or eliminating syntactic ambiguity. English sentences are replete with function words, including all sorts of suffixes, prefixes and auxiliary words. Many words play dual roles in this regard, both as pointers which help to establish meaning, and as delimiters; for example, prepositions and determiners.

The above definition of a formal language has been developed in such a way as to show its clear relationship to the notions of syntax directed compilers and programming languages, and to current investigations of the syntax of natural languages. An equally close relationship exists between this definition and the formal languages of symbolic logic. Rather than formally show this correspondence here, let us see whether we can use the above mechanism to identify the "logic" of a programming language.

The semantic studies which lie at the root of modern logic and metamathematics are based upon an adequate definition of the notion of truth. The fundamental problem can be stated as the problem of determining for a sentence those environments where it is satisfied. To this end, let us choose s, the preferred symbol of our formal language, to be a Boolean variable. In this case, we see that for any sentence x, the meaning $M(x)$ of x will be either "true" or "false." The interpretation rules become the counterpart of Tarski's definition of satisfaction for languages of symbolic logic.[14] A sentence x is logically true, or a tautology, if $M(x)$ is "true" for every initial assignment of values to the referent symbols in R. By this simple means, the notions and results of mathematical semantics can be extended to the generalized notion of formal language given by the above definition.

But what about English? Recall that our interest in this paper is English as a programming language. If we are to develop a syntax-directed interpreter for English, we must first determine what structural categories are to make up its environment E. This question is in some sense a priori to the question of English, for English presumably does not prejudice the structural relationships that exist among the elements of a universe of discourse. On the other hand, the decision as to the memory structures the computer is to use in storing its data is a crucial one. The efficiencies of a programming language depend strongly on policies concerning memory management and structuring. If the universe of discourse is weakly structured with few cross-relationships one would expect any language, English or not, dealing with such subject matter to be inefficient to use and of very limited expressiveness.

The first major issue, then, in using English as a programming language is the same as that for any other programming language, the policy concerning memory management and structuring. When using English, we take for granted a richly connected web of implicit relationships, which we must now make explicit in computer memory. In the current DEACON work, data is organized into ring structures. These structures are similar in many respects to the plex structures defined by Ross[15] and used by

Sutherland in Sketchpad,[16] and are an extension of the notion of list structure.

Once the structural categories of the environment have been chosen, the central issue can be immediately clarified. Each of the referent words and phrases of the language have, as their denotational values, elements which are members of these categories. These categories correspond therefore to parts of speech. Can a syntax for English be developed, using these new parts of speech, which accounts for all of its richness of grammatical structure? A second way to put the same question is this: if the subject matter of English is limited to material whose interrelationships are specifiable in a limited number of precisely structured categories, does English essentially become a formal language as defined above? I believe that the DEACON work to date constitutes a confirmation of this hypothesis.

DEACON makes use of transformational rules as discussed above. It does this in a rather clear way by dividing the syntactic aspect of the grammar rule into two parts. The first is a straightforward phrase structure rule (not necessarily context-free). The second part can be considered as essentially determining whether the constituents fall within the subspace on which the interpretation rule is defined.

In their discussions of transformational grammars, both Harris and Chomsky have pointed out that it is possible through transformations to reduce a complex sentence to a series of interrelated sentences of simple type. Quite independently, we found it most expeditious to use what we refer to as a Verb Table for analysis of a sentence in the DEACON System. The columns in this table correspond quite directly to kernel sentences. The various columns are cross-linked from right to left showing the role of one kernel sentence in defining a constituent of a prior one in the table. The Verb Table is a rudimentary realization of the notion of the deep structure of a sentence. It is interesting to note that the Baseball English language query system by Green et al [17] produces a spec list as an intervening table between syntactic and semantic analysis, which can also be viewed as a realization of the notion of deep structure when applied to the segment of English used in that system.

It is the central thesis of this paper that, when the subject matter of English is limited to material whose interrelationships are specifiable in a limited number of precisely structured categories, English essentially becomes a formal language as defined above. This hypothesis has far-reaching consequences. It implies that the complexities of natural language arise neither from vagaries of syntax nor from the variety of its subject matter, but rather from the immense complexities of the intervening memory structures which mediate between stimulus and verbal response. The words of the language are keys to the specific structures in memory which carry the referenced information. The relationships established among a particular set of words by a particular sentence are keys to the structural transformations, the interpretation rules, that develop the meaning of the sentence from the structures keyed to its constituent words. If we artificially limit these structural forms, English reduces homomorphically to a formal language.

I should like to make a few remarks on certain issues concerned with the efficacy of English as a programming language. First, it can be said that certain other existent programming languages are English-like in their sentence formats, for example COBOL. What is the essential difference between such languages and extended versions of DEACON? COBOL and other similar languages have chosen a restricted set of formats for their statements which are, to be sure, English-like. However the number of such phrase formats is very limited and any divergence from these formats is excluded. In developing these languages there appears to have been a hesitancy to allow the great plurality of forms which one finds in everyday English, possibly because of a fear that unacceptable levels of ambiguity might arise, possibly because of the acknowledged computing time required by a more elaborate parsing algorithm. In particular little has been done to capitalize on the rich variety of function words which one finds in English. In the DEACON work the bull was taken by the horns, so to speak. It has been found that a wide variety of forms can be accommodated in a reasonable number of rules. Although computing time due to parsing is still a critical problem, even here times are achieved which make the result feasible for a number of applications.

What about ambiguity? It is well known that systems for the syntactic analysis of natural languages produce an unacceptably high number of ambiguous syntactic analyses for a given sentence. This is to some extent true in the DEACON syntactic analysis as well. However, systems built along lines described herein go beyond syntactic analysis. It is found in practice that the semantic analysis aspects of the system resolve many of these syntactic ambiguities. In many cases, several parsings will yield a single meaning. More often, the interpretive rules, when applied to a parsing, will indicate it to be semantically vacuous, thereby reducing the number of meaningful analyses.

However, ambiguities remain. In some areas of computing, areas indeed which currently account for the great bulk of computations, a single program will be used to make a large number of calculations, and speed of processing and precision of statement are prime requirements. In this case, an algebraic language allowing no ambiguities, with an optimizing compiler to produce an efficient object program, is certainly called for. Even here the question of ambiguities at the problem definition level, before the programmer begins his translation, cannot be wholly overlooked.

When the ultimate user is less clear concerning his problem and the computer enters into the creative feedback loop, there is great advantage in providing the means for communication directly with the computer in a language he finds natural and which has greater flexibility. Further, there is a vast area where the computer can be of great value to ongoing operations, where military and management staffs need effective access to data in forms responsive to their immediate needs. The expression of these data manipulating requirements to the computer differs only by degree from programming as the computer specialist knows it. It is in these latter categories of programming that the programming language should be English. The conversational mode provides the means for immediately resolving ambiguities. The advantages of the interpretive mode for immediate response are not over balanced by the need for optimized code. And the naturalness of the language frees the user for concentration on the problem at hand rather than on its translation.

REFERENCES

1. For recent comment on the problem and prospects of "English as a Programming Language" see the discussion between Jean Sammett, R. W. Floyd et al in *Comm. ACM,* vol. 9, pp. 228–30 (1966).

2. A. Tarski, "Der Wahrheitsbergriff in den formalisierten Sprachen," *Studia Philosophica,* vol. 1, pp. 261–405 (1936); English translation in *Logic, Semantics, Metamathematics,* Oxford University Press, New York, 1956, pp. 152–278.

3. See papers of Turing and Gödel in M. Davis (ed.), *The Undecidable,* Raven, New York, 1956.

4. R. F. Simmons and K. L. McConlogue, "Maximum-Depth Indexing for Computer Retrieval of English Language Data," *Amer. Documentation,* vol. 14, pp. 68–73 (1963).

5. ——, S. Klein, and K. L. McConlogue, "Indexing and Dependency Logic for Answering English Questions," ibid, vol. 15, pp. 196–204 (1964).

6. E. Irons, "A Syntax Directed Compiler for ALGOL 60," *Comm. ACM,* vol. 4, pp. 51–55 (1961).

7. N. Wirth and H. Weber, "EULER: A Generalization of ALGOL, and its Formal Definition," ibid, vol. 9, pp. 13–25, 89–99 (1966).

8. E. L. Post, "Formal Reductions of the General Decision Problem," *Am. J. of Math.,* vol. 65, pp. 197–215 (1943).

9. J. McCarthy, "A Basis for a Mathematical Theory of Computation," in P. Braffort and D. Hirschberg, *Computer Programming and Formal Systems,* North Holland, Amsterdam, 1963, pp. 33–70.

10. K. Gödel, *The Consistency of the Axiom of Choice and of the Generalized Continuum-Hypothesis,* Princeton University Press, 1940.

11. N. Chomsky, "Three Models for the Description of Language," *IRE Transactions on Information Theory,* IT-2(3), pp. 36–45 (1956).

12. ——, and G. A. Miller, "Introduction to the Formal Analysis of Natural Languages," in R. D. Luce, R. Bush, and E. Galanter (eds.), *Handbook of Mathematical Psychology,* vol. II, Wiley, New York, 1963, pp. 269–322.

13. Z. S. Harris, "Transformational Theory," *Language,* vol. 41, pp. 363–401 (1965).

14. A. Tarski, "The Semantic Conception of Truth and the Foundations of Semantics," *Phil. and Phenomenological Research,* vol. 4, pp. 341–76 (1944); reprinted in H. Feigl and W. Sellars (eds.), *Readings in Philosophical Analysis,* New York, 1949.

15. D. T. Ross and J. E. Rodriguez, "Theoretical Foundations for the Computer-Aided Design System," Proc. of Spring Joint Computer Conference, 1963, pp. 305–22.

16. J. E. Sutherland, "Sketchpad: A Man-Machine Graphical Communication System," ibid, pp. 329–346.

17. B. F. Green, Jr., et al, "Baseball: An Automatic Question Answerer," Proc. of Western Joint Computer Conference, 1961, pp. 219–24.