# WHAT THE NATURE OF NATURAL LANGUAGE TELLS US ABOUT HOW TO MAKE NATURAL-LANGUAGE-LIKE PROGRAMMING LANGUAGES MORE NATURAL

Jerry R. Hobbs
Department of Computer Sciences
City College, CUNY

## 1. Introduction

When a student is learning an algorithm from a textbook, his first approach is frequently through an English description. This is normally easier to understand than raw code, and sometimes easier than a flow chart, in spite of the fact that programming languages are designed for algorithm specification while English is only pressed into its service. If the English is easier to understand, it is likely that it has many features that would ease programming itself. This paper investigates some of these features.

I am not suggesting that we program in pure English, even if it were possible--it is too verbose. Those who believe it would make for succinct programs are comparing very high level English specifications with relatively lower level programming languages whereas English descriptions can run the whole range of levels--down to

> Load register 5 with contents
> of memory location 190.

At every level, there is symbolic specification more economical than any English specification.

Most efforts in the past to make programming languages "like" English have involved spreading a thin veneer of English vocabulary and perhaps some English syntax over a very ordinary programming language. This paper is an attempt to push toward a deeper understanding of some of the properties of natural language which make it easy for people to use, with special emphasis on algorithm descriptions, and a consideration of how these might best be carried over to very high level programming languages. In what follows, four aspects that characterize coherent English texts are considered in turn:

1. redundancy, ellipsis, and contextual interpretation of words;
2. the prevalence of spatial metaphors;
3. various anaphoric devices; and
4. implicit and explicit intersentence relations.

For each we give illustrations from the world of algorithms and describe how a natural language processing system can handle them. We then consider which features would be desirable to have in programming languages and whether they could be incorporated to gain the flexibility and comfort of natural language without opening the door to its variability, imprecision, and ambiguity.

The time is perhaps ripe to ask these questions. For much progress has been made in the past few years in the field of natural language processing, and we understand more now about some of the mechanisms that enable one to extract meaning from an English text. The observations presented here grow out of work that has been done on the semantic analysis of well-written, carefully honed algorithm descriptions, such as one finds in Knuth's Art of Computer Programming (1973). In addition, we have analyzed sets of directions of how to get from one place to another (Hobbs 1975) as well as complex expository texts (Hobbs 1976). The work has been aimed toward characterizing coherence in English texts in general, so that the properties of language under study are not peculiar to one use, but are an important part of what makes natural language natural.

## 2. Related Work

Our work on algorithm descriptions is very

similar to Balzer's work on message distribution instructions (1975) in that both seek to squeeze a precise meaning out of an inherently imprecise English text. Some of the conclusions reached in both lines of research are also similar. However, where in Balzer's work there is emphasis on satisfying the requirements of the program ultimately to be produced, we have concentrated on methods of discourse analysis which are independent of domain of discourse. For example, where he uses a heuristic which says that a variable whose value is changed must subsequently be used, we see this heuristic as a specialization of a more general pattern common to all English discourse (cf. Section 7).

Our work is also related to that of Miller (1976) and that of Scragg (1974) on English descriptions of processes. In one sense, it is complementary to Miller's work. Where he has examined texts in which intersentence relations are generally simple temporal succession and the control information is embedded within complex noun phrases, we have chosen instead to examine the broad range of intersentence relations possible in process specifications in a domain of discourse whose noun phrases are rather spare.

Another effort to incorporate desirable natural language features into programming languages is represented by the work of Nylin & Harvill (1976). They have proposed a set of operators which act something like English tenses and time words and allow the programmer to access states of the machine other than the current one. While there is a rich assortment of time constructions in the algorithm descriptions we have been examining, we have not yet studied them deeply enough for a useful discussion. Moreover, I would like to concentrate in this paper less on the concepts expressed in natural language and more on the way natural language works.

More distantly related is work on automatic programming, exemplified by Green & Barstow (1975), involving user dialogs and English-like, problem-oriented descriptions of programs. Our problem has been easier, since the texts we are studying are highly specified and, from a human standpoint, require no exceptional problem-solving capabilities.

## 3. The Inferencing System

In all I have to say about processing English texts, I am assuming that the text has been syntactically preprocessed into some fairly simple representation. In our system, this is a collection of logical propositions encoding the information contained in the text. Moreover, I assume there is available a large collection of world knowledge facts or axioms expressed in some symbolic representation which the processor can operate on. For us, they are represented in the form of predicate calculus axioms. There are then mechanisms in the system for building chains of inference out of these axioms. The mechanisms do two sorts of inferencing:

1. forward inferencing: the mechanism is given a proposition in the text as a starting point and a pattern representing the inference sought, and it tries to find a chain of inference linking them;
2. backward inferencing: the mechanism seeks a chain of inference culminating in a given proposition, which begins at some proposition in the previous text.

In addition, there are means of deciding between chains of inference when more than one satisfies particular requirements.

The two inferencing mechanisms are used by a set of "semantic operations" which draw inferences selectively to interpret and structure the text. Among the operations are one for interpreting general words in context and recovering omitted material (called "predicate interpretation"), one for resolving anaphoric expressions, and one for detecting the relations between sentences and hence the overall structure of the text.

## 4. Redundancy, Ellipsis, and Predicate Interpretation

An important characteristic of natural language texts is their very great redundancy. Indeed, we might say it is this that allows us to understand texts at all. Consider the perfectly normal sentence

> Let link variable T point to the root node of a binary tree. (1)

We call T a variable, yet it is implicit in the fact that T is a capital letter in an algorithm description that T is a variable. Moreover, the subject of "point" is necessarily a variable. A link variable is a variable that points. We call the root a node but the root is necessarily a node, as are the object of "point to" and an element of a binary tree. "Tree" is more or less implicit in "root", "node", and "binary".

It is this redundancy that allows ellipsis to occur. Material can be omitted because the information is implicit in what remains. For example, (1) can be paraphrased

Let T point to a binary tree. (2)

The full sentence can be recovered because of the interaction between the requirements of "point" and the nature of binary trees. In English texts in general, the part is frequently recoverable from the whole because of the whole's environment. Quantity words can often be omitted too. For example, if we are linking through two linked lists ordered by the value fields of their nodes, and we encounter

Print out the greater of node P and node Q (3)

we know it is the value fields of the nodes that is referred to, because of the requirements of "greater".

In our natural language processing system, the recovery of omitted material is accomplished by means of an operation called predicate interpretation. This seeks to discover the meaning a word or predicate acquires by virtue of its presence in a particular context. When the word is encountered in a text, the world knowledge associated with the syntactically related words in the sentence is probed in order to satisfy demands imposed by the word.

Among other things, this acts as a kind of type-forcing. Stored with various operators are the types the operands must be, and predicate interpretation forces arguments of a predicate into the correct form. In (2), the predicate "point" requires its second argument to be a node. The knowledge about binary trees is searched for a dominant node, the root node is found and the sentence becomes

Let T point to the root of a binary tree.

In algorithm descriptions it is reasonably safe to assume that the predicate "greater" requires its arguments to be numbers. Thus, to interpret (3) we search our knowledge about nodes to find the most prominent associated number. We find that a node typically has a value field whose value is frequently a number, so (3) is fleshed out to

Print out the greater value of the value fields of node P and node Q.

Note moreover that the phrase "node P" must be expanded into "the node which P points to".

To a limited extent, this feature already exists in programming languages, e.g., the automatic type conversions of FORTRAN and the fact that "+" can be either integer or real addition, depending on context. The difficulty with bringing this facility over wholesale into programming languages is that the operation sometimes requires deep searches through a large data base of inferences, and the results are chancy. For example, if (3) were modified to

Print out the greater of P and Q

it could be interpreted as referring to the values of P and Q or the value fields of the nodes P and Q point to. Which is chosen depends on the search order, which is somewhat accidental.

Nevertheless, it ought to be possible for a programmer to specify in an expanded declaration portion of a program the structure and purpose of and relationships between data objects, and to specify with a procedure the nature of its parameters. No increase in total programming effort would be involved, for this information already goes into the comments. Then when a procedure is called with the wrong type of object, that object is used as the starting point in a search for the right argument.

Schwartz (1975) has put forward very similar ideas in connection with a proposed very high level language to be built on top of the set-theoretic language SETL. It includes a rich collection of possible type sepcifications and a type-coercion operation which uses these to expand

elliptical dictions. Among the recommended declaration forms are the operators has which allows the user to specify the structure and attributes of a data object, is which permits complex type descriptions to be specified, and either which allows several types to be subsumed under one supertype.

It should be noted that such facilities will not necessarily make programs shorter. Rather, it will shift the programming effort, especially the required attention to detail, from the dynamic instructional portion to the static declaration portion, where people are more comfortable with detail.

Section 8 contains an example illustrating many of these points.

5. The Spatial Metaphor

It is very frequent in algorithm descriptions, as in every kind of English text, to use spatial metaphors to describe more abstract concepts. For example, we speak of the processor going from one step to another in an algorithm, of a variable going from 1 to N, and of a pointer moving along a linked list.

In our system for analyzing English, the spatial metaphor is accommodated by the way in which the world knowledge is organized at its deepest levels. Some primitive concepts are a Scale or a "becoming", which is roughly a partial ordering; a point being on a Scale, or being a member of the partially ordered set; one point exceeding another on a Scale; and an entitity being at a point on a Scale, or an entity being at another entity. "At" is in fact a very general predicate capable of a wide variety of specific interpretations, depending on context. In a given text, the predicate interpretation operation seeks an interpretation or binding for "at" by probing the nature of its arguments.

The three uses given above of spatial metaphor can be interpreted via the following models:

1. An algorithm is a Scale. The points on the Scale are instructions. For the processor to be at an instruction on the Scale is for it to execute the instruction.

2. There is a Number Scale, which is a Scale. The points on it are numbers. For a variable to be at a number is for its value to equal the number.

3. A linked list is a Scale. A node in the list is a point on the Scale. For a pointer to be at a node is for the variable to point to the node.

Verbs of motion are then decomposable into expressions involving the primitive "at". For example, we have the axiom

$$(\forall y_1,y_2,y_3)(go(y_1,y_2,y_3) \supset become(at(y_1,y_2),at(y_1,y_3)))$$

That is, we can decompose "$y_1$ goes from $y_2$ to $y_3$" into "$y_1$'s being at $y_2$ becomes $y_1$'s being at $y_3$". Then consider the sentence

Go to step T4.

We know that the subject is the processor. The goal is an instruction. Therefore the underlying "at" is interpreted as "execute". In

N goes down to 0

N is a variable, 0 is a number, and therefore the underlying "N at 0" is interpreted to mean the value of N equals 0. "Move" has the same decomposition as "go". Consider

P1 moves along the list one node behind P.

P1 is a variable and it is located at successive positions on the list. In interpreting a list as a Scale, we discover that these positions are nodes, and that P1 is thus being used as a pointer.

Since we tend to have very strong visual images of the entities our programs deal with and the actions performed on them, it is possible that a healthy collection of motion verbs--such as "move", "go", and the visual analog of "go", the arrow--would make a programming language more convenient to use. Decompositions in terms of "scale" and "at" could be either known by the system or the user could specify how "scale" and "at" were to be interpreted. All of this would require some education of the users, but it would pay off in more natural programming.

Consider another example of a spatial metaphor: It is not one of the mathematical properties of a stack that it has a vertical orientation, but it is the way we visualize a stack, and thus the way we talk about it. For example, in the algorithm description system, we can handle

Remove the top element from the stack

using a mathematical definition of "top". But in

> Remove the top two elements from
> from the stack

the mathematical definition no longer works. We must use the fact about "top" that it refers to a portion of a vertical scale whose high end coincides with the high end of the scale, and the fact about "stack" that it has a (metaphorical) vertical orientation. Another example requiring this knowledge is

> A is above B in the stack.

To understand this we must first interpret the stack as a scale with upward vertical orientation. Then we can infer that A exceeds B on that scale.

Finally consider the word "contain". In the basic meaning of "contain", for A to contain B is for the object B to be physically inside the enclosed region A. We can tap many of the metaphorical uses of "contain" in algorithm descriptions by specifying a set as metaphorically a region and its members as being inside the region. This corresponds to a common visual image of a set, and is required for the following examples:

> The queue contains a node for each
> item with no predecessor.
> Each node contains two fields.
> If the matrix contains any row which
> contains a 0, . . . .
> Ignore any instruction containing an
> undefined operand.

A compiler for a very high-level programming language allowing such uses of "contain" would need the system-provided or user-provided knowledge that a queue is a set of nodes, that a node is a set of fields, and than an instruction is a sequence and hence a set of symbols. It would need to know that a matrix may be thought of as a set of rows, a set of columns, or a set of elements. But the "set" interpretation for "contain" does not seem to work for the sentence

> PSUM contains the partial sum of the
> numbers input so far.

Here it seems necessary to specify directly that a variable may be considered metaphorically a region. This may correspond to many people's most naive visual image of a variable.

6. Anaphora

The word "anaphora" is a linguistic term for the various devices used in natural language for referring to an entity occurring in or deducible from the previous text. For our purposes we may divide the kinds of anaphora that occur into two categories. In the first, the anaphor--a pronoun or a definite noun phrase--refers to an entity mentioned explicitly in the previous text:

> Suppose we have a binary tree. This
> algorithm traverses the binary tree.
> Suppose we have a binary tree. This
> algorithm traverses it.

We probably do not want to introduce this feature into a programming language. The use of variables is a clear improvement over English, in clarity and brevity.

However, the second kind of anaphora--a definite noun phrase referring to an entity only implicit in the previous text--would be a desirable feature.

> Link through list L, printing out
> the value fields.

The natural language processor recognizes the reference of the definite noun phrase by means of a backward search through the collection of axioms for a chain of inference beginning in the preceding text and implying the existence of the definite entity. In this example, we find first the fact about nodes that a node contains a value field and next the fact about lists that a list consists of nodes, and finally we find the occurrence of "list" in the preteding text. Thus, it is the value fields of the nodes in list L.

If a similar capability were incorporated into a programming language, the compiler could resolve the reference by accessing the structural and relational information discussed in Section 4. In a sense, this is the other side of the coin--the part is specified and not the whole.

7. Intersentence Relations

The implicit and explicit intersentence relations in algorithm descriptions encode much of the flow of control of the algorithm. Looping struc-

tures simply do not occur in ordinary English discourse; in algorithm descriptions they are encoded in verbs like "go", "repeat", and "perform". Other patterns do occur, however, and it is worthwhile to see what they are and how they are recognized.

In the natural language processor, intersentence relations are determined by matching successive sentences against a small number of patterns, stated in terms of inferences to be drawn from the sentences. The most common pattern is Overlapping Temporal Succession. Instructions it relates are translated into successive lines of code. We give one variety of the pattern. (In all the pattern specifications, $S_2$ refers to the current clause or sentence and $S_1$ to the previous.)

> $S_1$ asserts a change whose final state is presupposed by $S_2$.

More precisely, the patterns tells us to seek from the previous sentence an inference of the form "become(A,B)", where "become" is a predicate indicating a change from state A to state B, and to seek to infer from the current sentence some form of state B. This pattern occurs most frequently in algorithm descriptions when $S_1$ describes a change in vlaue for some variable and $S_2$ uses that variable, as in

> Decrease N by J. If it is 0, reset it to MAX.

Using pre-stored knowledge of the word "decrease", we can infer from the first sentence,

> become(equal(N,X),equal(N,X-J)) (4)

for some X. The second sentence decomposes into

> imply(equal(it,0),become(equal(it,0), equal(it,MAX))).

The appearance of "equal(it,0)" as the first argument of "imply" means that "it" is equal to something. If "it" is identified with N, we have a match with the final state of (4) and hence a match with the Overlapping Temporal Succession pattern. Note that if we had assumed that "it" referred to J, we would not have matched the pattern. Recognizing intersentence relations frequently aids in pronoun resolution in precisely this fashion.

Contrast is another particularly important pattern, because as Balzer (1975) has noted, a contrast between implications translates into a "CASE" statement. Letting "element" refer to either the predicate or one of the arguments of a proposition, the Contrast pattern may be stated as follows:

> 1. $S_1$ and $S_2$ have one corresponding pair of elements which are contradictory or lie at opposite ends of some Scale;
> 2. the other corresponding pairs of elements are identical or belong to the same small set (i.e., are "similar").

In the sentences

> If INFO(M) < INFO(N), then set M to LINK(M). If INFO(M) > INFO(N), then set N to LINK(N). If INFO(M) = INFO(N), add one to COUNT and advance on both lists. (5)

the highest level predicate is "imply". The first arguments of "imply"--"INFO(M) < INFO(N)", "INFO(M) > INFO(N)". and "INFO(M) = INFO(N)"--are contradictory conditions. The second arguments are similar assignment statements, although recognizing this in the case of the final sentence of (5) requires accessing knowledge about how one represents and talks about data structures. In particular, we must know that to advance is to move forward and here "forward" is determined by the direction of the pointers in the linked list. One moves along a linked list by following the links, in this case by setting M to LINK(M) and N to LINK(N).

Thus are the sentences recognized as fitting the Contrast pattern, not as Temporal Succession, and hence are interpreted as a branching condition rather than as successive instructions.

The next two patterns reflect a common phenomenon--a stretch of text acts as an attempt at the successive approximation of a meaning, or an attempt to avoid misunderstanding. The first pattern is Paraphrase

> $S_1$ and $S_2$ are (inferrably) the same except that either
> 1. an argument of $S_2$ is more fully specified than the corresponding argument of $S_1$; or
> 2. $S_2$ has adverbial modification $S_1$ lacks.

An example is

> Initialize. Set stack A to empty and set link variable P to ROOT.

Note that it is necessary to recognize this relation if we are to realize "Initialize" does not refer to some kind of initialization other than what is in the second sentence. Recognizing the pattern in this example is quite complex. We must know that by convention the implied subject in each of the clauses is the "processor". To initialize is to cause to be in an initial state, and the only thing the processor can cause, beyond a change in the order in which it executes instructions, is a change in the value of a data structure. We then recognize that stack A and link variable P are data structures, and that "empty" and "ROOT" are plausible initial states.

In algorithm descriptions, it is common for one of the sentences in a Paraphrase to relate the action to the overall course of the algorithm and the other to relate it more directly to code. In a sense, the one is for the benefit of the human reader, the other for the benefit of the machine.

Next is the Example pattern:

> The elements of $S_2$ are subsets or members of the corresponding elements of $S_1$.

An instance is

> Reverse list L. If L is "A B C", then set L to "C B A".

If we failed to recognize the Example pattern and assumed they were successive instructions, the two instructions would cancel each other whenever L began as "C B A". The Example pattern is recognized here by decomposing "reverse" into a description of the change it effects on an ordered set, and recognizing "A B C" as a specific member of the class of lists.

To what extent can a very high level programming language profit from these relations? The Contrast pattern is simply "CASE" and suggests nothing new. Temporal Succession is just successive instructions, but the "Overlapping" imposes a coherence on texts that programs could profit from. If the compiler for our very high level language does not find sufficiently proximate pairings between assignments to and uses of variables, a warning is issued. This could catch such insidious errors as

```
N1 = N1 - 1;
IF (N = 0) THEN N1 = MAX;
```

where N1 is meant instead of N. This is just the sort of error that can escape detection for months in a large program.

The feature of successive approximation of meaning, or clarification, or simply redundant specification becomes more important as our programming language becomes more English-like and thus more open to ambiguity. To an extent, it exists already, in that a comment next to a line of code may be considered a paraphrase. In a sense we want to break down the sharp distinction between comment and code. There is a difficulty in that in English texts, deep inferencing is frequently required to recognize the patterns. But this can be overcome by introducing the operators "IE" and "EG" which would signal Paraphrase and Example respectively. The compiler could then use the line so tagged to check its interpretation of the previous line, or to try again for an interpretation if it failed on the previous.

## 8. An Example

In this section we will look at a "program" written in an imaginary programming language incorporating some of the ideas discussed above. We will then examine the work a compiler would have to do in order to turn it into correct "lower-level" code, say PL/1.

```
REVERSE(LIST);
    LIST points to head of linked list L;
    L contains nodes NODE;
    NODE contains 2 fields: INFO, LINK;
    P pointer, moves along L;
    P1 moves along L one node behind P;
    P2 moves along L one node ahead of P;
    REVERSE reverses L; EG REVERSE
       ( <A,B,C> ) = <C,B,A>;
    REVERSE returns pointer to head of
       reversed list;
FOR EACH NODE P
    RESET LINK FROM P2 TO P1;
END REVERSE;
```

The following points may be noted about this "program":

1. Assignments to temporary variables are allowed in the declaration segment via appositives--"linked list L", "nodes NODE". This allows us to avoid using anaphora referring to explicitly mentioned entities.

2. "Contain" occurs twice, but it must be interpreted differently in each case. In line 4, it leads to the declaration of a structure array or of two parallel arrays. In line 3, it is not reflected directly in the code but aids the compiler in interpreting the phrases "moves along L" and "FOR EACH NODE P".

3. "Pointer" in line 5 is probably implicit in "moves along L", but its inclusion insures the correct interpretation of "moves along". In interpreting "moves along" the compiler will access the knowledge that a linked list is a scale whose orientation is determined by the direction of the links. That is, it is a partially ordered set whose partial ordering is the transitive closure of the relation between A and B defined by "the LINK field of A points to B". This fact in turn will enable us to interpret "behind" and "ahead of".

4. While lines 5-7 describe the purposes or functions of variables P, P1, and P2, they allow us to reconstruct the actions of the variables in the instructional portion of the program. This is an example of static details about purposes, which people feel comfortable with, replacing dynamic details about successive values, which people have trouble integrating into their overall view of the program.

5. Line 8 may play no role in the final program beyond that of a comment. On the other hand, a sophisticated compiler might use it to check the code it has constructed, or alternatively, to decide among several possible interpretations in the instructional portion. The example tagged by "EG" gives a means of checking the code that is less general than "REVERSE reverses L", but the case to be checked is easier to construct. This assumes that the compiler can make the translation from the triple < A,B,C > into its corresponding representation as a linked list.

6. Since the instructional portion does not specify what value is returned, this must be deduced from line 9. The compiler must know enough about "reverse" to know that the head of the reversed list is the last node in the original list, and it must keep track of which variable points there after the loop.

7. "FOR EACH NODE P" does not specify the range of P nor the order in which P visits the nodes. These must be recovered from line 5 and the information about the ordering of L that was inferred to interpret "moves along".

8. "LINK" is an anaphoric reference to the LINK field implied by "NODE" of the preceding line. The resolution uses the information given in line 4. "LINK" is expanded into "LINK(P)".

9. Most of the instructions in the body of the loop come from the declaration segment. The compiler uses the facts of lines 5 and 7 to move P along L by the assignment "P = P2". To remain one node behind P, P1 must be reset to the old P at the same time, and to keep one node ahead of P, P2 must follow the link--"P2 = LINK(P2)".

10. In a sense "from P2" in line 11 is redundant, since it is implicit in the definition of P2 in line 7. But in addition to serving as a check on the interpretation of line 7, it insures that P2 is set before LINK(P) is changed.

11. The length of this "program" is roughly the same as the length of the corresponding program in "lower-level" code. But the balance between the static, purpose-oriented declaration segment and the dynamic, action-oriented instructional segment has shifted completely. Indeed, the instructional segment is confined to a brief statement of the key trick. As a result, the "program" has a natural quality that obviates the use of comments.

## 9. Conclusion

The observations we have made about natural language come out of the careful investigation of algorithm descriptions and other English texts. The suggestions for a very high level programming language, on the other hand, are still at the stage of speculation. Whether they can be implemented without sacrificing the precision required of a programming language is an open question.

But of the many features that could be built into such a programming language, it seems reasonable to choose those that make natural language easy to use. It seems reasonable to aim for programming languages that have the flexibility and richness of natural language and for programs that have the texture and coherence of a natural language paragraph.

BIBLIOGRAPHY


1. Balzer, R., Imprecise Program Specification, Proc. Meeting on 20 Years of Computer Science, Pisa, 1975.

2. Green, C., and D. Barstow, Some Rules for the Automatic Synthesis of Programs, Advance Papers Fourth IJCAI, Tbilisi, USSR, 1975, pp. 232-239.

3. Hobbs, J., A General System for Semantic Analysis of English and Its Use in Drawing Maps from Directions, American Journal of Computational Linguistics, Microfiche 32, 1975.

4. Hobbs, J., A Computational Approach to Discourse Analysis, Dept. Computer Sciences Research Report 76-2, City College, CUNY, December 1976.

5. Knuth, D., The Art of Computer Programming, Vol. 1, Reading, Mass., 1973.

6. Miller, L., Natural Language Procedures: Guides for Programming Language Design, Sixth Congress, Intl. Ergonomics Association, College Park, Md., July 1976.

7. Nylin, W., and J. Harvill, Multiple Tense Computer Programming, SIGPLAN Notices, December 1976.

8. Schwartz, J.T., Reflections on some very-high level Dictions having an English/'Automatic Programming' Flavor, SETL Newsletter 141, January 1975, Courant Institute, New York.

9. Scragg, G., Answering Questions about Processes, University of California, San Diego, 1974.