College students who were not familiar with computers were asked to produce written natural language procedural instructions as directions for others to follow. These directions were solutions for six file-manipulation problems that also could reasonably be solved by writing computer programs. The written texts were examined from five points of view: solution correctness, preferences of expression, contextual referencing, word usage, and formal programming languages. The results provide insight both on the manner in which people express computer-like procedures "naturally" and on what features programming languages should include if they are to be made more "natural-like."

Natural language programming: Styles, strategies, and contrasts

by L. A. Miller

Computer programming is perhaps the best example of a class of problem-solving tasks that can be called "procedure specification." In these tasks a sequence of actions is specified in some language such that, when these tasks are executed by a designated agent, a particular goal can be accomplished. In procedure specification tasks other than computer programming, like the writing of trouble-shooting manuals or kitchen recipes, the language of specification is the writer's natural language. Computer programming, however, is accomplished in unnatural (some would say "unholy"), formally defined, and self-contained languages. Thus, to specify a procedure for a computer, it is *not* sufficient to have the process in mind or to be able to describe it in

one's own tongue; one must express this procedure in the computer's tongue. This latter requirement greatly restricts the usability of computers.

But what are the alternatives? Some have argued that it is feasible to build a computer interface that permits the user to interact in more or less "natural" language, at least under restricted conditions.[1,2] Others have been near-derisive in their dismissal of such a possibility.[3-5] Still others have argued that "naturalness" is perhaps best attained by working the other way around—by modifying programming languages so that they have more natural language features.[6-8] And finally there are a few unpublished but outspoken queries of computer sophisticates who wonder what the fuss is all about: What's so difficult anyway about learning a programming language or three? (For discussions of these various positions, see References 1 and 9 through 11. For reviews of natural language processing projects, see References 12 through 14. For an example of a natural language processing project, see References 15 and 16.)

The motivation for the present research was that, regardless of our intuitions, we still do not have enough empirical data to go on to make an informed choice among the foregoing alternatives. To be sure, there is evidence in abundance about the difficulties people have with current programming languages[17-19] (also see the extensive annotated bibliography of Atwood *et al.*[20]). Nevertheless, we know very little about how people would perform with *constrained* natural-like features built into programming languages, on the one hand, or how they would express computer-like solutions in *unconstrained* natural language, on the other. This study focused on the latter aspect, by studying "programming"—procedure specification—independently of formal programming languages, computer systems, and prior programming experience. Our purpose was to contrast this type of *au naturel* programming to that which is accomplished with computer languages such as PL/I and FORTRAN.

The question to which we are really seeking a partial answer is: In what ways would it be difficult, given today's state-of-the-art capabilities in artificial intelligence and computational linguistics, to actually *implement* a system that could take the texts of our unsuspecting subjects and truly and completely *understand* them? The reader is asked at this point to imagine what indeed these difficulties might be, probably guessing—as we did—that there might be many difficulties, but being uncertain as to which of the envisioned problems would be found, and which encountered ones would be unanticipated. Various results of the experiment may indeed confirm some expectations. Our agnostic objective, however, was to provide a body of empirical data that could help shift the basis of the discussions from conjecture towards fact.

Method

overview

We here first provide a perspective for our general research methodology and then discuss in detail its various aspects. The method chosen to achieve our exploration objectives was (1) to design an information-request task that was highly representative of computer applications, (2) to generate specific problems for testing which suggest—at least to programmers—the writing of computer programs for their solution, and (3) to ask our computer-naive subjects to create natural language texts that provided a procedure for someone else to follow to solve the problems. We take these texts to be the natural-language equivalents of what would be produced by a programmer writing a short *ad hoc* program to satisfy information requests in a situation involving actual computerized files.

Our methodology is somewhat similar to the problem-solving protocol analyses of Newell and Simon[21] and the communication mode analyses of Chapanis.[22] In the former, subjects are asked to verbalize out loud their thinking process as they attempt to solve some problem; these verbalizations are subsequently transcribed and analyzed in relation to "micro-process" models of problem solving. An essential difference between our situation and this one is that our subjects were not only to provide a procedural solution to some problem but also to adopt the role of describing this procedure as if it were to be instructions for other people to follow. In this respect our task is much more like that of Chapanis, in which two people communicated to achieve some goal (e.g., assembling a mechanical device). Nevertheless, our subjects were writing procedures in the abstract, not in the context of a real problem situation with a real cohort. This abstract impersonal aspect of our task renders it, we argue, more typical of program writing than *in vivo* communication.

In our analyses, we examined and re-examined the subjects' texts from a variety of perspectives, and each usually involved making some theoretical assumptions about the psychological processes of language use and problem-solving capability. Although it would have been nice to have drawn upon strong psychological theories relevant to the behavior we are studying, the state of the art of psychology (and psycholinguistics) is such that it afforded us little in this regard. Rather we have derived some new empirical *descriptions* of "natural language programming" and have tried to show how our results are relevant to the design of programming languages.[23-25]

subjects and experimental design

The subjects were 14 undergraduate students from local colleges who were paid for their participation. None had any prior experience with either programming or computer systems.

Figure 1 Description of information data structures used in the experimental problems

File 1: Salary File
Records organized alphabetically by employee's name, which is the first item on the record. Second item is hourly wage; third is hours worked in last pay period; fourth is amount deducted each period for savings bonds.

1. Name
2. Wage
3. Hired
4. Deduction

File 2: Personal File
Records organized alphabetically by employee's name, which is the first item on the record. Second item is employee number; third item is age at last pay period; fourth is marital status.

1. Name
2. Number
3. Age
4. Marital Status

File 3: Job File
Records organized in terms of increasing employee number, which is the first item on the record. Second item is job title; third is year in which employee was hired; and fourth is supervisor's rating of the employee's performance.

1. Number
2. Title
3. Hours
4. Rating

The single independent variable of the study was the factor of problems, and, in what is called a "repeated-measures" design, each subject specified natural English procedural solutions for the same (randomly ordered) six problems.

apparatus

A terminal-based (IBM 2741 SELECTRIC® typewriter terminal) interactive computer system (System/360 Model 91) was used for controlling (in APL) all aspects of the experiment, including presentation of the problems, entry of procedures, data measurements, and data analysis. Instructions were provided by means of a tape recorder and headphones.

task scenario and files

The subjects were asked to imagine themselves as file clerks in the personnel office of a hypothetical company with the responsibility of maintaining and searching the company's files in response to management requests. A set of three hypothetical files was described, each file containing records concerning individual employees, with four pieces of information on each record (see Figure 1). This information was given to each subject for reference throughout the experiment.

Subjects were asked to note that Files 1 and 2 are organized alphabetically by name, while File 3 is organized by increasing employee number. Thus, given only an employee's name, and needing to find information from File 3 (i.e., title, date hired, rating), it is first necessary to obtain the person's employee number (Item 2 in File 2) and then use this number to find the corresponding record in File 3. The files were deliberately constructed in this way to permit some problems to involve more complicated file accessing than others. The subjects were also told that they could not modify these file structures in any way.

Table 1 Statement of the six procedure-specification problems given subjects in the experiment; Problems 1-4 are called the attribute-testing problems and 5-6 are called the noncontingent problems

1. *Attribute-testing, conjunctive, File 3*
   "Make a list of employees who have a job title of photographer and who are rated superior. List should be organized by employee number."
2. *Attribute-testing, conjunctive, Files 1 and 2*
   "Make a list of those employees who make more than 8 dollars/hr. and also are over 50 years old. List should be organized by employee name."
3. *Attribute-testing, conjunctive, Files 2 and 3*
   "Make a list of all those employees who are 64 or more years old and who also have 20 or more years of experience. List should be organized by employee name."
4. *Attribute-testing, disjunctive, Files 1-3*
   "Make one list of employees who meet either of the following criteria:
   (1) They have a job title of technician and they make 6 dollars/hr. or more.
   (2) They are unmarried and make less than 6 dollars/hr. List should be organized by employee name."
5. *Noncontingent, wage-computation, File 1*
   "Make a list of employees along with the wages they should receive for the last pay period. List should be organized by employee name."
6. *Noncontingent, new-entry, Files 1-3*
   "A new person has been hired. Enter the following information about him in the appropriate files: Name—Xavier Tungsten; Employee number—4444; Married; Wage—5 dollars/hr.; Title—technician; 21 years old; Hired in 1973; Deductions—10 dollars/wk; no rating as yet.

We told subjects that their specific task would be to respond to six requests for information. However, they were not actually to obtain the information themselves; rather, they were to write down a detailed instruction procedure that would be followed by someone else, e.g., a new clerk they were breaking in.

problems

Four of the problems (see Table 1 for full text) required the evaluation of two pieces of information about the same individual. If a person's records met the problem criteria, then certain information about him (e.g., name) was to be entered into a final list. Three of these four *attribute-testing* problems involved a conjunctive relation between the two pieces of information (both criteria had to be satisfied), but differed in terms of the files that had to be accessed, increasing in complexity from Problem 1 to Problem 3. The last of the four attribute-testing problems was the most complex, in terms of data accessing and the testing criteria.

The remaining two problems, called *noncontingent* problems, did not involve testing of attribute-value information, but one required a computation of salary earnings, and the other the entry of information about a new employee.

procedure

The subjects were given a 30-minute training session, conducted using tape-recorded instructions accompanied by practice at the

computer terminal. These instructions explained the scenario, the nature of the task, and the nature of the data structure involved in all problems. Following presentation of each experimental problem, subjects were instructed to type in a sequence of steps that was to represent a procedural solution for accompiishing the objective of the problem. Each step was to contain a more or less independent action, and we emphasized that the procedures should be written so as to be easily understood and executed by persons similar to the originator.

The subjects were given no suggestions as to the form or language to be used, but the instructions emphasized the requirement for detail, particularly concerning the basis for making decisions. A minimum of five steps was required for each problem to ensure that some level of specificity in the procedural description was obtained. Using the terminal, subjects typed their solution for each problem, limiting each line of input (a step) to 80 characters (if more than 80 were typed, they were asked to re-enter the line). They were permitted to modify their procedure at any point (change, insert, or delete steps). Completion of the task for a particular problem was signaled by the subject typing the word "END." The next problem could then be self-initiated after a short delay during which coded data were printed out. The total time to complete both the training and the six problems ranged from about three to seven hours, on from one to three days, with an average total time of about four hours.

Results and discussion

Presentation of specific results is organized under the following six headings: (1) general overview, (2) nature of problem solutions, (3) preferences of expression, (4) contextual referencing, (5) word usage, and (6) comparison to programming languages. A summary included under each heading evaluates those results with respect to the main thesis being investigated in this study: that the way to vastly extend the usability of computer systems for computer-naive people is to provide a full natural language interface for them to specify computer procedures.

General overview

Almost all of the subjects expressed some reluctance about having to specify a detailed sequence of steps to solve the problems. They were willing to do so for the experimenter's sake but indicated they were used to following, not specifying, procedures; besides, they commented, the problems were straightforward and required little explanation (!). In producing the protocols, subjects typically began typing within a very short time of being presented with the problem. There was no evidence of their having thought through a complete problem solution beforehand. There

Figure 2 Example of a solution for attribute-testing Problem 2, with content-category codes inserted following the text to which they have been assigned

1. Go to the personal file (1a).
2. Make a list of all employees (2b) over 50 (3g).
3. Take this list (2c) to the salary file (1a).
4. Make a list of all employees on the list (2b) who make more than 8 dollars an hour (3d).
5. Arrange the employees on the final list into alphabetical order (5a).

Table 2 Content measures of the protocols for the six problems

Content measures	*Problems*					
	1	*2*	*3*	*4*	*5*	*6*
Number of steps	6.00	8.07	8.50	12.86	8.93	14.29
Number of words	68.10	83.00	95.20	144.50	93.50	136.70
Number of unique words	13.80	15.10	14.90	20.20	16.50	15.60
Number of codes	10.34	13.36	13.64	21.87	11.65	24.13
Words/step	11.35	10.29	11.20	11.24	10.47	9.57
Codes/step	1.72	1.66	1.60	1.70	1.30	1.69
Words/code	6.59	6.21	6.98	6.61	8.30	5.67

was almost no editing of their work except for correcting overruns of the 80-character limit on a step. Thus, protocols appear to have been produced in a linear incremental fashion, with no evidence of other than localized planning (an example of a somewhat shorter solution to Problem 2 is given in Figure 2; the codes given in parentheses are explained under the third heading).

The main experimental hypothesis of this study really concerns the effect of problems: that performance would vary significantly from problem to problem, particularly increasing with the complexity of the attribute-testing problems 1–4. The first four measures shown in Table 2 assess the amount of "content" in the individual solutions, and these were our primary performance measures. (The number of "codes" refers to the mapping of text into the classifications given in Table 3 and discussed under the next heading.) For each of these measures we performed two separate analyses of variance: one for all six problems, and one just for the four attribute-testing problems. In all cases we determined that the effect of problems was significant (the probability, p, of this occurring by chance alone was less than 0.05). Such an unremarkable gross effect *had* to occur and be detected, else there would be no statistical justification for the detailed analyses which follow.

The last three measures in Table 2 essentially measure the "density" of information per unit of expression. Together they pro-

vide a sensitive assessment of the extent to which subjects solved the problems more or less in the same way, using the same mapping of concepts to words: If there were significant differences among problems on any of these measures, this could imply that subjects varied the way they expressed concepts as a function of the particular problem.

Such a result would make detailed comparisons among the problems much more tenuous. However, none of the analyses of variance tests showed significant effects of problems for these measures, for either the first four problems or all six ($p > 0.20$). Although these findings are not conclusive, they do suggest that subjects indeed approached all problems with the same conceptual problem-solving framework—analogous to using the same "programming language" for all problems, instead of switching between languages.

Nature of problem solutions

We now describe the strategies the subjects used in solving the problems, then discuss the completeness of the solutions, and finally examine the relation between the strategy chosen and its completeness.

subjects' strategies

In examining the subjects' solutions from the point of view of the underlying algorithms they chose to solve the problems, we first tried to think of the different conceptual ways each problem could be solved, and we classified these into a small number of prototype methods. We then matched (rather easily) the subjects' solutions to these prototypes.

Our main concern in developing the algorithm prototypes was for the attribute-testing problems 2-4. There are really two different approaches one can take for these problems: *a person-by-person search* or a *file-by-file search*. In the former, you pick a file containing information on one attribute value and check through the records until you find a person meeting that criterion. You then interrupt the search in that file, keeping your place, and find that person's record in the second file to check out the second criterial attribute value; if the second is as it should be, you jot down the person's name for the final output list. Because of the interrupted back-and-forth nature of this approach, it requires some complex transfer-of-control specification, although it can actually involve lesser amounts of data accessing and temporary data creation than the second approach (especially for Problem 2). For the file-by-file search, however, you pick the file containing information on the first criterion and check *every* record in that file, noting down information (e.g., on a scratch list) about those people who meet the criterion. Only after the first file has been completely examined do you then go to the second file to check the other criterion. Furthermore, for this particular type of search, there

are two different ways you can use your scratchpad notes: in *Type 1* you can use the notes from the first file to direct your search in the second to a few specific records; else, in *Type 2*, you can exhaustively search both files making independent notes on each and subsequently compare (intersect) the lists of names to find those common to both. There are, therefore, two basic types of file-by-file algorithms, and a single person-by-person algorithm, with a number of variations on these themes possible, especially for the fourth problem. We note in passing, however, that not even the identified person-by-person algorithm is as sophisticated or efficient as those which a moderately competent programmer would likely produce (based on asking FORTRAN and APL programmers to produce reasonable programs for Problems 2-4).

Having developed the three prototypes, we then examined the subjects' solutions to determine how they fit these. We found a very clear-cut and consistent preference for the file-by-file strategy, with 12 of the 14 subjects choosing this approach for all three of Problems 2-4. The remaining two subjects picked the person-by-person strategy for the three problems. For the 12 file-by-file subjects, the number picking the Type 2 variation of intersecting separately compiled lists was 4, 6, and 10 for Problems 2-4, respectively.

Since the file-by-file strategy is viewed to be simpler, as is its Type 2 variation, these results suggest that most subjects had a consistent preference for simpler approaches, and this predilection for simpler strategies increased as solution complexity increased (from Problem 2 to 4).

solution completeness

Having first categorized the solution strategies, we then graded them for completeness. (It was the case that most solutions were almost never really clearly incorrect; either they were obviously right—albeit incomplete—or else they were sufficiently incomplete that one could generously imagine omitted steps which, if supplied, would render them correct.) In this scoring we first established an overall understanding of the subject's solution approach and then asked ourselves whether each logically necessary step in this solution method was explicitly stated or at least strongly implied. Our general bias was to give the subject the benefit of the doubt wherever possible. In terms of the number of subjects, out of 14, who provided 100 percent complete solutions (whatever strategy), performance was high for only the first two problems: 13, 13, 3, 4, 1, and 8 subjects were 100 percent complete for Problems 1-6. When this "perfect" criterion was relaxed to an estimate of what would be at least 80 percent complete solutions, the performance level improved but was still poorer for the last three problems: 13, 13, 12, 9, 7, and 9 subjects were 80 percent complete for Problems 1-6. This partitioning of

Problems 1-3 being relatively complete and 4-6 being relatively incomplete was verified by totaling, over the 14 subjects, the number of omitted actions for each problem: Problems 1-6 were found to have a total of 1, 1, 13, 54, 17, and 16 omitted actions, respectively.

It is clear, then, at least for the attribute-testing problems, that the completeness of solutions decreases markedly as the complexity of the problems increases. We speculate that this finding could imply that the direct translation of natural language programs into formal computer programs may be feasible only for rather simple problems; for more complex ones we could envision as being necessary much more complicated interactive processes intervening between the subjects' initial specifications and their ultimate interpretations (see our "cognitive mismatch" hypothesis below). This point of view assumes that people in general can develop solutions for problems of even high complexity, and it is just the *manner* in which they express the solutions that can cause translation difficulties. Another view—certainly not counter-indicated by our present data—is that the locus of difficulty may well be conceptual, not expressional; that is, maybe subjects' solutions decrease in completeness with complexity because subjects are less and less able to formulate conceptually adequate solutions, regardless of whether they are expressed in "thoughts," natural language, or computer programs. Anecdotal evidence from direct studies of the problems of naive-user programming at least provides external support for this latter view.[2]

Examination of the nature of the omissions for Problems 4-6 suggests that the omissions were *not* due to a single common factor but have at least two different interpretations. Concerning the most complex attribute-testing problem, Problem 4, subjects were clearly not as sensitive as they should have been to the complexities of accessing File 3 from File 1 or vice versa, as these access actions were among the most frequently omitted. Thus, they failed to pay sufficient attention to the data structure and its organization. An implication of this finding for programming is that an important area for assisting naive persons in specifying procedures may be to provide functions that do not force them to take into account the nature of the data structure, particularly as it becomes complex.

As for the wage computation problem, the fifth one, most of the omissions involved leaving out mention of finding an item, like hourly pay, or omitting the hours-times-wage computation. Thus, despite the rather clear problem specification, fairly obvious and necessary aspects were omitted. We interpret this in terms of the subjects being unwilling or careless in providing details of an action that is viewed as being unitary or not easily analyzable. Similarly, for the new entry problem, the omissions mainly concerned

the specifics about writing down the new information on a new record and inserting that record in the file in the appropriate place. Again, as for Problem 5, it appears that these omissions can be attributed to subjects' difficulties in being explicit about the details of a process which for them was so well internalized or practiced that it was a single undifferentiated action.

The obvious but important implication for programming of this latter interpretation, for Problems 5 and 6, is that there may be serious "cognitive mismatches" between the solutions provided by naive programmers *intended for other people* and the solutions required for effective computer programming. It may be expected that, without extensive computer experience, naive users will project onto the computer the same expectations they have of other people, with respect to what things need to be specified in great detail and what things are clearly "obvious," requiring little elaboration. To the extent that this is true, an approach to the problem of making computer use more "natural" would be clearly inadequate if it emphasized only the capability to accept the rich syntactic structures of natural language; such a system would also have to incorporate a good deal of the semantic and pragmatic "understanding" underlying the use of language (e.g., controlling the decomposition of verbs into complex programs as a function of the context, the specification of default operations as a function of the semantic nature of the entities acted upon, etc.).

relation between strategy choice and completeness

On the average, subjects who chose a file-by-file strategy had almost twice as many omissions per problem as did those who chose the person-by-person solution strategy (1.8 versus 1). Thus, although more people adopted the simpler approaches, they produced more incomplete (and therefore more erroneous) solutions. Since subjects stayed with the same strategy across problems, it is likely that this phenomenon can be attributed to differences in subjects' specification skills, with those persons capable of generating more complex (but possibly more efficient) strategies also better able to provide more complete specification of them.

summary

We emerge from this analysis of subjects' solutions with a somewhat tentative yet nagging uncertainty concerning the viability of an *unconstrained* natural language computer interface for programming. Our problems were really not difficult, yet not only did subjects employ rather simple—perhaps even simplistic—solution algorithms, but also the quality of these algorithms (as measured by completeness) decreased markedly with only moderate increases in problem complexity. While we have no doubts at all concerning the capacity of natural English for specifying highly precise and complete procedures, we *are* concerned about people's abilities to use the language in this way, particularly for difficult problems.

Table 3 Classes and subclasses used for the content analysis of subjects' solutions

1. *Actions involving existing data structures*
   (a) Files—go, get, use, look, open, select
   (b) Records—same
   (c) Records—movement to a different location
   (d) Item—go, get, use, look, select
   (e) Problem—problem statement, other given information
2. *Actions involving new data structures*
   (a) Creating new item/record (single)
   (b) Creating new item/record (multiple)
   (c) Manipulation—go, get, use, move, label
3. *Attribute testing*
   (a) Check on single record, attribute, and value mentioned explicitly
   (b) Check on multiple records, attribute, and value mentioned explicitly
   (c) Check on single record, only value explicit
   (d) Check on multiple records, only value explicit
   (e) Check on something other than attribute or value
   (f) Check on single record, attribute/value mentioned only implicitly or in incomplete linguistic structure
   (g) Check on multiple records, attribute/value mentioned only implicitly or in incomplete linguistic structure
4. *Transfer of control*
   (a) Iteration, repetition
   (b) Full conditional (with provision for both outcomes)
   (c) Partial conditional (provision only for successful test outcome; no provision for "ELSE" or no outcome).
   (d) Unconditional transfer or "GOTO"
   (e) Sequencing reference—"after, when, until"
   (f) Reference to terminating procedure ("stop")
5. *Transformations*
   (a) Explicit ordering of new data ("alphabetize")
   (b) Computations involving item information
   (c) Invoke some other general procedure
6. *Miscellaneous*
   (a) Nonprocedural comments

Preferences of expression

We are concerned here with fine details about the elemental conceptual process explicitly expressed in subjects' textual solutions, e.g., what kinds of data-referencing actions were expressed, how attribute checking was accomplished, etc. We were not concerned, however, with the particular *linguistic* structures subjects chose to express the same kind of concept; we felt that such an analysis was probably premature and, in any event, would require skills and programs that we did not at that time possess. We therefore, after a number of iterative re-examinations of subjects' texts, developed a descriptive taxonomy of elemental processes, as shown in Table 3. All of the results described are based on examination of the subjects' texts after they were transformed into sequences of these codes.

content analysis methodology

All 84 texts were rated and transformed into content codes by two judges according to the following procedure. By reading the texts from left to right (and top to bottom), a string of words sufficient to correspond to one of the content categories was identified within a step (ranging from about three to ten words). This segment was assigned the appropriate category code, and the process was repeated for any remaining material. Figure 2 illustrates this coding procedure for a subject's solution to the second attribute-testing problem. Only a few times did the assignment of text to codes cross step boundaries; further, the left-to-right analysis was adequate except for a small number of instances in which one action was embedded within another (the embedded action was scored as occurring second). It is important to note that the present analysis does not evaluate subjects' solution strategies or their correctness; it focuses only on the frequency with which the various conceptual actions were expressed in the subjects' text (for discussion of the many problems of such "content analyses" see References 26 and 27).

overall results and validation

We sought initially to determine whether subjects' solutions reflected the logical structure of the problems. For example, the first three attribute-testing problems differ primarily in the extent to which data accessing in different files is required, and a monotonic increase in Class 1 codes was expected. In the analyses, the code frequencies were summed for each protocol, and these totals were used as the basic measure in a two-factor (problems and subjects), repeated-measure analysis of variance. The *mean* frequencies per protocol are shown in Table 4, and the corresponding percentages are shown in Table 5 for the attribute testing versus the noncontingent problems (the Knuth data are discussed under the last of the six headings—comparison to programming languages).

The analysis results provided strong evidence for the joint hypothesis that the category codes validly reflected the semantic information in the natural text *and* that the code frequencies reflected the logical requirements of the problems. Statistically reliable effects ($p < 0.05$) were found only when there was a strong plausible basis for predicting them; further, the ordering of the problem means was consistent with predictions in almost all cases. A partial list of such findings follows: (1) the frequency of data-accessing codes increased from Problems 1 to 3; (2) there was a monotonic increase in Class 1a and 1c code frequencies from Problems 1 to 3 (reflecting increasing complexity of data accessing); (3) the first three problems do *not* show significant overall differences of the attribute-testing codes (as they should not, since this aspect was the same for these problems; $p > 0.20$); (4) there *was* a significant effect of the attribute-testing code when Problem 4 was included in the analysis (which problem is much more complex in this respect); (5) the new-entry problem had sig-

Table 4 Mean frequencies for each of the major content classes for each of the six problems

Content class	*Problems*						*Overall means*
	1	*2*	*3*	*4*	*5*	*6*	
1	3.21	5.29	6.71	8.29	4.43	3.00	5.15
2	1.64	2.14	1.36	4.36	2.86	13.29	4.27
3	3.00	3.50	3.79	5.50	0.57	0.71	2.85
4	1.21	1.29	1.00	1.93	0.86	0.35	1.11
5	1.14	0.93	0.57	1.43	2.79	6.64	2.25
6	0.14	0.21	0.21	0.36	0.14	0.14	0.20

Table 5 Percentage of all content codes deriving from each of the six content categories for the four attribute-testing problems (Column 1) and the two noncontingent problems (Column 2); Column 3 shows the results of applying the content analysis to Knuth's (1971) data[32]

Content class	*Attrib. (%)*	*Noncont. (%)*	*Knuth (%)*
1. Existing data	40	21	16
2. New data	16	45	18
3. Attribute test	27	4	7
4. Control	9	3	22
5. Procedures	7	26	27
6. Comments	1	1	10

nificantly greater data-creation code frequencies (Class 2) than the computation problem (as it should).

Had there not been a reasonable relationship between the logical characteristics of the problems and the coded solutions, either the "rationality" of the subjects or the adequacy of the content analyses could have been open to question. As it is, confidence must accrue to both, and the following analyses can be viewed with some credibility.

subjects' preferences of expression

The data given in Tables 4 and 5 suggest that, overall, subjects were most concerned with the access, creation, and manipulation of data; they were *least* concerned with specification of transfer of control. We now examine the five major content categories in turn, focusing on those subcategories that occurred most frequently. This examination, while highly detailed, provides strong evidence of subjects' preferences in the expression of specific procedures; such preferences in turn have implications for the design of programming interfaces.

Examination of the frequencies of *Class 1 subcategories* reveals which aspects of existing data structures were of most concern to subjects—the files themselves, the records in the files, or the items on records (1a, 1b, and 1d, respectively). The subcategory accounting for the largest proportion of all Class 1 codes was that

of general reference to files (1a), with percentages ranging from 70 to 100 percent. Thus, the dominant process associated with existing data structures was to identify the file of interest but *not* to specify actions concerning records within files or items on records (this finding is interpreted in the discussion of Class 3 codes).

The *Class 2 codes* for creation of new data structures indicate that subjects could choose to focus either on individual elements (2a) or on groups of elements (2b), but the latter action accounted for the predominant portion of these actions (ranging from 92 to 95 percent across the attribute-testing problems, and from 53 to 72 percent for the noncontingent problems, 6 and 5, respectively). Subjects clearly opted to specify creation of new data structures *en masse* rather than on an individual entry basis. This finding suggests that a programming language in which operations can be performed on whole structures (as in APL) may be more compatible with natural propensities than a language requiring iterative item-by-item operations (as in FORTRAN).

The *Class 3 subcategories* provide the main basis for detecting preferences of expression for attribute-testing problems (the noncontingent results are not included due to the low frequency of these codes in Problems 5 and 6). The previously mentioned Class 1 predominance of reference to files rather than records or items is explained by an analysis of use of the multiple record-checking tests (Classes 3b, 3d, and 3g) versus tests of single records (Classes 3a, 3c, and 3f). The strongly preferred mode of expression was for multiple record checking, with percentages ranging from 85 to 96 percent. Since subjects expressed an attribute test that was to be performed over all records in a file, it was sufficient for them merely to specify the file of concern and the attribute value of interest; no reference to subcomponents of the file (records or items) is necessary with such an array operator method. This finding of attribute testing at an array level is consistent with the finding of array-type creation of new data structures detected from the Class 2 code frequencies, and it corroborates the inference that subjects prefer to deal with data structures on a mass rather than iterative basis.

A second major attribute-testing preference is indicated by comparing content codes for expressions that clearly identify both the attribute category *and* the value of interest (3a and 3b) to codes that clearly identify only the value (3c and 3d); the combined subcategories for the latter mode of expression accounted for much the higher percentages (70–94 percent). Since, in many cases, the appropriate attribute is implicit in the statement of a specific value, this result could be interpreted by hypothesizing that subjects considered it unnecessary to also specify the attribute name. A tentative generalization of this interpretation is that naive programmers might well want or expect a computer system to figure

out the semantic implications involved in *every* aspect of their input, a capability that would require extraordinarily detailed and complex computer representations of word semantics and knowledge structures—which capability would certainly be well in the future, if at all.

There are two findings concerning *Class 4 codes*. First, the subcategories of the full "if-then-else" conditional (Class 4b) and the unconditional transfer (Class 4d), both included on *a priori* grounds from knowledge of programming languages, *never* occurred. The second finding is that the majority of Class 4 actions were accounted for by the partial "if-then" conditional statement (Class 4c), with a range of values from 82 to 87 percent in the attribute-testing problems. These findings raise the general question of the interpretability of the subjects' protocol solutions, i.e., whether they could be understood and executed by other persons. The attribute-testing problems, if written completely, need explicit transfer-of-control structures. In our data, such control statements were mostly not provided (cf. Table 4), and even when they were present, they were incompletely specified.

In an attempt to provide qualitative information on the question of interpretability of protocols, the following segment from a protocol was given informally to about two dozen persons untrained in programming and not involved in the experiment (after describing the scenario):

"(1) See if the age of the person is greater than 50;
(2) Write his name down on a list."

Those interviewed were asked if they would know what to do if the age was *not* greater than 50 and, further, if they believed this was implied by the protocol. The response was almost unanimous. "Of course," the reply went, "you just check the next person, or if there are no more, you just go on." When asked whether such a course of action was implied by the protocol segment, the response was typically: "Well, this is what one would always do in this kind of situation." Apparently, the respondents were drawing from some base of experience in following iterative procedures. When searching for particular target values, the decision either to repeat an action or continue to the next one (or stop) is apparently derived from this kind of experience. For example, shampoo labels typically state: "Wet hair, apply shampoo, rinse, and repeat." As it stands, this procedure creates an eternal loop, but it is doubtful we would find many bathers forever cycling through such a procedure. This anecdotal information suggests that a large and complex body of experience is important in interpreting natural language procedure specifications.

Finally, for the *Class 5 codes* the subcategory of invoking a general procedure (Class 5c) accounted for most of the Class 5 code

occurrences for the attribute-testing problems and for the new-entry problem (Problem 6), ranging from 75 to 92 percent. Although Class 5c was a catch-all, occurrences of this code do reflect the invoking of general procedures by subjects—as does the organizing category 5a. For the wage-computation problem (Problem 5), the computation subcategory (Class 5b) appropriately accounted for the largest proportion, 46 percent. In one sense, use of these actions may be viewed as instances of invoking "macro" procedures which are viewed by the subjects as unanalyzed (or at least undetailed) "primitive" processes; the complexity built into these primitives allows for a significant reduction in the level of detail needed to be expressed in the problem solutions. While it may be that subjects have these primitives (formatting, alphabetizing, etc.) already well-learned and readily available, it is not clear whether subjects could easily provide an algorithmic description of these procedures.

summary

The overall nature of the results of this section also does not easily provide support for the view that providing a natural language programming interface is the way to radically improve computer usability for naive programmers. Although the strong preference for aggregate data accessing could certainly be implemented to some extent (e.g., by APL), aggregate *data creation* could not be supported so easily, and the other results imply extensive requirements for semantic interpretation of input and use of "world knowledge" well beyond present capabilities.

Contextual referencing

The expression "put them back and check the others . . ." is typical of many of the subjects' statements and illustrates the necessity for interpreting the words as a function of their particular contexts. Specifically, the statement illustrates the two pervasive reference problems found in our subjects' solutions: (1) knowing *what kind* of entity was being referred to by the pronouns and similar terms, and (2) once knowing what entity, determining *which one* of several possibilities was intended.

We were not able to develop a satisfactory scheme to quantify these problems separately, but we were able to assess the extent to which a reference required *some kind* of other information for interpreting either the entity type or the specific reference (or both). We classified data references into four categories: (1) files, (2) records within files, (3) items or other specific value information on records, and (4) new data structures such as lists. We then read each subject's solution to identify each data reference, and we classified it into one of these four types. We next assessed what information was required to resolve the reference—actually *where* in the solutions or elsewhere we found this information. Three such information-location "levels" were used: (1) Level 0—the information needed to determine the "what" and

Table 6 Percent of data references requiring prior context for disambiguation; Level 1 indicates that a minimal, local context was sufficient, whereas Level 2 requires substantial backtracking before the reference is disambiguated

Referent	*Contextual references*		
	Total (%)	*Level 1 (%)*	*Level 2 (%)*
Record	55	21	34
New data	53	24	18
Item	44	18	26
File	14	8	6
Average	42	18	24

"which" of a reference was to be found within the same step as the reference (e.g., "records" in "Take all records from File 1 . . ." needs only the information of that step for interpretation), (2) Level 1—references were resolved by information provided in the immediately preceding step, and (3) Level 2—resolution of references required information from steps earlier than just the last, or else required interpretation of the problem statement or other information. These levels roughly index the amount of and basis for inferencing required for resolution: for Level 0, the references were often quite explicit in themselves or within the noun phrase in which they occurred; resolution of many of the Level 1 pronouns was often achieved by the simple syntactic-based action of finding the noun in the previous sentence which was in the same grammatical case (e.g., subject, object) as the pronoun; but Level 2 references mostly required a good deal of semantic inferences considerably more complex than the simple checks of the first two levels.

Of all the data references identified (see Table 6), 42 percent were found to require the use of information outside the step unit to resolve the referent; 18 percent were rated as Level 1 (requiring only the previous line) and 24 percent were rated as Level 2 (requiring other information). The breakdown by data category shows that this overall finding of higher Level 2 contextuality is also true for the individual data categories except the references to files which, after all, had only three possible alternatives.

These estimates of contextual referencing, particularly those of Level 2, are quite large, somewhat unexpectedly so in view of the well-defined constraints of the problem and scenario, and in view of our scoring bias to give subjects the benefit of the doubt whenever we were uncertain (scoring the referent as Level 0). In addition, a number of the Level 0 referents, although resolvable within the step, involved contextual inferencing similar to the referents of Levels 1 and 2. These facts suggest that the overall 42 percent figure may well be a lower bound on the degree to which

context-dependent referencing occurs, with a greater extent to be expected in less-constrained or less-well-specified situations. Indeed, corroborating evidence for the contextual nature of language use can be found in Kucera and Francis' list[28] of the 100 most frequent words occurring in a million-word sample of texts. Almost every word that is clearly contextually referential occurred in this list, including all personal, demonstrative, and relative pronouns. In addition, the word "the" alone accounted for about seven percent of all the words; most expressions of the form "the X" can legitimately be countered with the question "*which* X?"—a question which is answerable only by the context. Other aspects of these word-frequency data suggest, if not contextuality, at least a level of expression far removed from the "concrete:" among the 100 words, only four nouns were "concrete" (time, man, years, way); also, no adjective expressing an absolute attribute of an object (e.g., green, heavy) was on the list; and, finally, the only verb occurring which was of a "concrete" or explicit action nature was "said."

This evidence for high contextuality in natural language does not necessarily imply that people cannot or will not learn to be specific in their references; indeed, the success in training people in programming languages indicates that they do learn. Nevertheless, the propensity to refer to things contextually appears to be very strong, and designers of new programming languages oriented towards the naive programmer might do well to consider providing *some* capability for interpreting contextual referencing to support this normative characteristic.

summary

The high degree of contextual referencing found in subjects' solutions provides yet another perspective for doubting the feasibility of unconstrained natural language programming, at least by people inexperienced in procedure specification. Natural language supplies a variety of means for providing links between one segment of a text and other portions, often known as "cohesive" mechanisms, and many of these are exceedingly complex in terms of the required syntactic and semantic processing (see Halliday and Hasan,[29] also Miller[15]). However, most people are typically unaware of these complexities and would have difficulty in limiting their use of such devices to just those for which computer implementation could possibly be achieved. Thus, a fully unconstrained natural language programming interface would probably have to support *all* of these mechanisms.

Word usage

We now focus on the most elemental aspect of subjects' solutions: the individual words. Considered first are the general characteristics of word usage and then the possibility of enforcing a restricted vocabulary.

general characteristics of word usage

There were a total of 8708 words (tokens) used in all of the 84 protocols, and these were repetitions of 610 unique words (types). The per-protocol averages were 104 words, 9.8 steps, and 10.7 words per step. Each unique word was used, on the average, 14.3 times, a rather high token-to-type ratio for such a relatively small body of text. Overall, then, it appears that verboseness was not a characteristic of the subjects' productions, and a relatively small vocabulary sufficed for their work.

For the four attribute-testing problems a total of 5485 words were used, whereas 3223 words were used in the noncontingent problems. On the average, however, somewhat fewer words were required for the former (98 words per solution) than for the latter (115 words per solution). The numbers of unique words used for the two kinds of problems were 473 and 360, and the average token-to-type ratios were 11.2 and 9.0, respectively. (The latter finding suggests a greater uniformity of expression in the attribute-testing problems.)

We assessed *commonality* of word usage among subjects by the following procedure. For each of the 14 subjects we determined their 25 most frequently used words, which accounted for from 47 to 69 percent of the total words (words differing only in the ending signifying plurality were grouped together—e.g., "file" and "files"). Then, working with one list at a time, we took each of the 25 words on this list and tallied how many times it appeared anywhere within the other 13 subjects' lists. This resulted in a 14 by 25 matrix of frequencies (subjects by words), with each cell containing a number ranging from the maximum of 13 (the word was found on all other lists) to the minimum of 0 (the word occurred on no one else's list). On the average, each high-frequency word used by one subject was also used by 5.7 other persons; given the maximum of 13, this means that almost 44 percent of the top 25 words were shared in common. The top three words were shared by an average of 9.9 persons (71 percent), whereas the top five were shared by an average of 8.7 (62 percent). While this analysis suggests a substantial degree of commonality among subjects in word usage in the present experiment, the author unfortunately does not know of other similar data which would permit comparative assessment of this finding.

The final general analysis focused on the imperative verbs used by subjects in their solutions (similar to procedure calls in programming languages). We examined the attribute-testing problems separately from the noncontingent ones and computed for each problem set the 10 most frequent words that were judged unambiguously to be "commands"; excluded were words having multiple unrelated meanings like "make" (e.g., "make a list . . ." versus "find those employees who make . . ."). For the

attribute-testing problems these words, in the order of most to least frequent, were "go, look, take, check, find, put, remove, pull, write, see"; for the noncontingent problems these commands were "write, go, enter, put, take, indent, look, multiply, leave, find." These lists appear to be consistent with the nature of the problems, with more "examination"-type words occurring in the attribute-testing list and more "data-manipulation"-type words in the noncontingent problems. We note, in passing, that this type of method for characterizing word usage might be generally useful in providing a basis for selecting commands in restricted-lexicon situations.

vocabulary restriction

Although subjects were in no way restricted in the words they could use, the above results show that they did not use all that many words, and there was a good deal of commonality in their usage. These findings suggested to us the possibility that perhaps a restricted vocabulary *could* have been given to subjects without serious ill effects.

One check on this possibility is provided by examination of the very infrequently used words. We found that words that (on an overall basis) occurred less than three times accounted for only 4.3 percent of the total words produced but accounted for 50 percent of the unique words. We examined a sufficient number of these words to give us confidence that the semantic content of such idiosyncratically used words could be conveyed by other, higher-frequency, words; this being true, the size of the lexicon necessary for the present natural language expressions could be cut in half—to about 300 words—leaving the protocols roughly 96 percent intact. We next examined the references to data structures and elements, finding that these accounted for almost 39 percent of the total words and 17 percent of the unique words, substantially larger than any other word class examined (such as articles, pronouns, prepositions, "command" verbs, etc.). The overall 39 percent total word usage is quite appropriate to the nature of the problems; however, the 17 percent proportion of the unique words is not strictly necessary, since many synonyms were used for the same data referent. Training in the use of the data structures, along with instructions to use certain terms, could certainly have permitted this number to be reduced substantially. Thus, with elimination of idiosyncratic low-frequency words and standardization of data referencing, it can be argued that a lexicon of about 200 words (eliminating 100 data structure words) would suffice for the present language productions without seriously affecting the amount of content or the semantic nature of that content. We further estimate that this number could be cut approximately in half if synonyms for words other than data structures were eliminated; the resulting base lexicon would thus be about 100 words. The behavioral feasibility of constraining word usage is given support by the study of Kelly and

Chapanis,[30] who compared performance with a restricted vocabulary versus an unrestricted one and found no behavioral difficulties.

summary

The data on word usage provide for the first time something other than doubts and hesitations for the notion of an unconstrained natural language programming interface. Subjects used a relatively small number of unique words and total words, and they appeared to use these words in the same way. The relatively high frequency of synonym usage would not necessarily require extensive semantics but rather a complete synonym dictionary.

Comparison to programming languages

We now compare and contrast the natural language productions of this study to "typical" features of programming languages. In doing so, many of the issues and problems associated with discriminations among programming languages (e.g., Ledgard[31]) have been ignored so that very general points can be made.

We first make a very gross comparison based on the overall frequencies of the six classes of actions we discussed earlier; then we discuss the details of specific differences.

overall classes of actions

To provide a programming perspective for assessing the relative frequencies of actions in our subjects' solutions we analyzed Knuth's frequency statistics[32] for FORTRAN commands in programs written by Stanford University students. These students, like ours, were relatively new to the task of specifying procedures, and, similarly, the problems they were working on were apparently of relatively low complexity. By making reasonable assignments of the FORTRAN commands into our six major content categories we were able to compute the relative percentage of each category, shown as the third column in Table 5.

Comparison of the programming percentages to those of our subjects shows that the major discrepancies are for categories 4 and 6. Concerning the latter, the 10 percent level of occurrence of comments in the programs probably reflects training in documentation practices rather than a real propensity for the specification of procedures. However, the 22 percent figure in Knuth's data for transfer-of-control actions most probably reflects definite requirements of programming language specifications. Indeed, as Knuth pointed out, the students' programs lacked the large sections of code concerned with checking format, syntax, etc. that characterizes the work of experienced programmers; the 22 percent might therefore be viewed as an *underestimate* of the proportion of transfer-of-control commands in professionals' programs.

The discrepancy between the value of 22 percent from Knuth's data and the much lower values of nine percent and three percent

found in our data suggests an important area of difference between the two specification modes: Formal programming languages seem to require much more extensive and explicit specification of control aspects than natural language. Our earlier analysis of control actions as well as those that follow further suggests that this much lower degree of specificity would probably *not* be corrected by subjects' elaboration of their internal conceptualizations concerning the *semantics* of specific words; rather, it would appear that the control aspects are supplied by *pragmatic* experiences with procedures. In turn, this implies that substantial *psychological* assessments of the subjects' understanding of actions and procedures would have to form the basis for any implementation of an unconstrained natural language interface, and such assessments have hardly yet begun.

detailed comparisons

Table 7 shows the highlights of the detailed comparisons, which are discussed in order below.

Data. With respect to data aggregates, the very high level of contextuality of referents discussed under contextual referencing is perhaps the most outstanding contrast between the natural language and programming language procedures. While it is true that the value of a particular variable in a program, for example, may not be determinable except during execution, the variable (or other data aggregate) must always be defined or declared, else a syntactic language violation will occur. In the present protocols there was a cavalier disregard for such details.

A second important data finding was the tendency to express data-checking operations over a whole data aggregate, as previously noted. This is similar to that capability provided in APL, for example, but is in contrast to the element-by-element mode in a language such as FORTRAN.

There were also some other interesting differences. When indexing occurred, the specification was typically in terms of local adjacency, using words like "next." Such a capability is not provided in most present-day programming languages. In contrast to the penchant of programming languages for various data types (e.g., fixed-point, double-precision, etc.), there were no such distinctions in the protocols. Similarly, data aggregates were not defined, dimensioned, or declared, nor were distinctions made between literal and numeric entities. Explicit assignments, which comprised roughly half of the commands in Knuth's data,[32] almost never occurred; synonyms were introduced and values were assigned implicitly.

These striking results concerning the contrast to programming languages of our untutored subjects' *data manipulations* makes one pause at the sheer magnitude of the difficulty of developing a

Table 7 Comparison of natural language expressions of procedures to characteristics of programming languages

Features	*Programming languages*	*Natural language specifications*
Data		
Declarations, etc.	Always explicit	Never occurred
References	Explicit, well-defined	Implicit, contextual
Examination/ creation	Usually iterative, element by element	On aggregate basis
Indexing	By numerical/variable value, major aspect	Seldom occurred, then contextually defined (e.g. "next," "previous")
Data types	Many, defined	No distinction
Format specs.	Many, explicit	Infrequent, contextual
Transfer of control		
Extent	Major aspect of programs and style	Seldom specified
IF-THEN-ELSE	Most used at present	When occurred, only partial—IF-THEN (no else)
IF (cond.) GOTO	Major feature	Never occurred
Uncond. GOTO	Was major, still common	Never occurred
Exception detec.	Important feature	Never occurred
Structure	Many types: recursion, co-routines, nonlinear	Basically linear block structures
Procedure calls	Frequent, specified completely	Major control mechanism, but context specified
Argument passing	Always explicit	Mostly implicit
General language		
Lexicon	Very limited, except for variable names	Can be rich and large, with many synonyms, may be restricted
Sentence type	Active imperative and conditional	Mainly active imperative, but can be declarative/conditional
Sentence syntax	Quite rigid	Extremely variable, may be very complex

system capable of turning these fuzzy, incomplete, ambiguous, and oh-so-knowledge-dependent specifications into the smooth, precise statements required to guide computer execution.

Transfer of control. There is a sizable discrepancy between the proportion of transfer-of-control commands in computer programs and the virtual absence of these in the present protocols (review the previous discussion of this topic under the content-analysis heading). In addition, even for the few instances of this

kind identified, the typical programming language constructions of the full "IF-THEN-ELSE" and the unconditional "GOTO" never occurred. Certainly, no subject used any of his meager amount of conditional expressions to do any exception handling, syntax checking, end-of-file testing, consistency verification, etc., that is a hallmark of normal good programming practice.

A partial explanation for the paucity of control specifications may be due to the structure of the subjects' procedures. In general, they appeared to be very much like a linear sequence of blocks of procedure calls in a somewhat narrative form and, of course, missing the BEGIN-END delineators, arguments, declarations, etc. Further, in informal testing, asking subjects to describe the details of specific procedures *does not* change the level of control detail supplied: Subjects either provide another linear sequence of more-detailed procedure calls, or else they give what is essentially an example of the main processing path of the procedure, without explicitly stating the general control structures. It therefore looks as though this major difference in the level of control information supplied is not due to the level of detail at which subjects are considering the problem; rather, it appears that specifying control is simply not an emphasized aspect of procedures (or their memory representations) at *any* level of detail.

Given that the major control characteristic of subjects' programs is that of a linear sequence of procedure calls, we can at least examine how arguments are specified for the two methods. In programming languages, of course, the arguments in a procedure call are always completely and explicitly specified; further, the order in which they are given almost always conforms to a single predefined sequence. (This is even true of the majority of most system commands, which can also be considered to be something like procedures.) In natural language procedures, however, the opposite is true: arguments were never completely and explicitly stated, and missing arguments had to be inferred from context; further, argument order could vary extensively.

General language features. It is in the area of vocabulary size and sentence types that the two groups have the greatest similarity. Both have relatively small vocabularies, and we have argued that the subjects' vocabularies could be extensively reduced without significant impact. Also, the instructions in both are expressed typically by active imperative sentences.

From the point of view of developing a parser for the commands, the similarities vanish, however. Programming language commands are restricted to a fixed syntactical arrangement of information, with key words (e.g., "DO") or operators (e.g., assignment symbol) determining a single syntactic construction for each command aspect. The grammar rules in the program language

parsers need only consist of, at most, a relatively small set of context-free rules; any complexity of the parsing algorithm itself (e.g., LL versus LR versus Earley) is introduced from considerations of efficient performance and is not due to any complexities or vagaries in the source-command syntax (cf. Aho and Ullman[33]).

In contrast to the "one syntactic structure maps to one command" feature of programming languages, natural languages are a compiler-writer's nightmare! First, the same "command" can be expressed by a wide variety of different words (the differences in meaning among them requiring wide variations in what other words also occur) and also by a huge variety of syntactic structures (even keeping the words constant); second, in reverse, the same syntactic structure—depending on the different words inserted into the structure—can map onto a wide diversity of "commands."

A second and more profound linguistic difference between natural and programming languages has to do with the notion of "style." Although there are many perspectives as to what linguistic style is (see, for example, Sebeok[34]), we use the term to refer to an author's *communication strategy* as indicated in the author's text by two features: (1) the nature of the conceptual propositions conveyed by the text, and (2) the manner in which these propositions are organized or structured. Concerning (2), there are two primary levels of organization to be examined for stylistic features: (a) the structure within a sentence (or command), *sentential syntax*, and (b) the structure among sentences in an overall cohesive text, *textual syntax*. In these terms, our previous discussion of programming and natural language syntactic differences indicates that, for the latter, there may be widely differing styles at the sentential level of syntax; for the former, however, the programming language formalisms enforce such strong syntactic restrictions that there is little opportunity for programs written in the same language to show stylistic variations at the sentential level.

We therefore focused on stylistic differences from the viewpoint of *textual syntax*, including in our observations a variety of different programming languages and a variety of other types of procedural text than just our present data (e.g., kitchen recipes, assembly instructions, trouble-shooting manuals). Looking first at programs, we see a very strong propensity in their opening "sentences" to define, dimension, declare, and otherwise "size" *data structures*. Such introductory data propositions can often comprise a major component of the overall contents of the programs. In comparison, while we duly note that "natural" programs often begin with (usually much shorter) lists of ingredients (edible or otherwise), equally often these lists are not incorporated into the

Figure 3 Comparison of "normal forms" for programs versus natural language specification of procedures: Task involves packing Christmas decorations into boxes; Figure 3A illustrates typical *conditioned action* style of programming; 3B illustrates natural *action qualification* style (the arrow represents the primary action to be accomplished by the program)

```
DO END UNTIL TIME = 5:00 PM
    I = 0
    DO  END.OUT WHILE I < 200
        I = I + 1
        OPEN BOX(I)
        J = 0
        DO  END.IN WHILE J < 12
            GET NEXT BALL
            IF RED THEN
                IF LARGE THEN
                    IF UNBROKEN THEN
                        J = J + 1
--------------------->  PACK BALL IN BOX(I) CELL(J)
                        RETURN (END.IN)
                    ELSE RETURN (END.IN)
                ELSE RETURN (END.IN)
            ELSE RETURN (END.IN)
        END.IN
        CLOSE BOX(I)
    END.OUT
END
```

(A) *Program Normal Form*

```
---------> PACK LARGE RED DECORATIONS TWELVE TO A BOX.
           MAKE UP A TOTAL OF 200 BOXES.
           STOP AT 5:00 PM IF NOT FINISHED.
           BE SURE TO PACK ONLY THE UNBROKEN ONES.
```

(B) *Natural Normal Form*

main dialogue but are set aside and referred to in accompanying tables. Further, there is not the programs' single-minded concern with "sizing," although this is indeed a strong feature; rather, a great deal of what can be called "preprocessing" is specified in these data propositions (e.g. ". . . 1 dozen medium peppers, *seeded and chopped*" or ". . . 16 four-inch lengths of No. 18 solid wire, *with insulation removed* 1/4 inch on each end . . .").

The *real* stylistic differences appear *after* these initial data propositions, however. One has only to glance over a few well-written programs to see the dominant textual style of programs: great massive control structures of DOs and IFs, with the primary data-manipulation activities embedded deep within these. For this reason, we characterize this style as "*conditionalized action*." However, natural language procedures provide a reverse emphasis: they almost always begin with those primary actions that are so deeply embedded in programs; special conditions or circumstances that control if and how the action is to be applied are expressed rather as "qualifications," usually following the action words. We therefore characterize natural language "program-

ming" style as "*action qualification.*" The contrast between these styles is shown by the contrived pseudo-program of Figure 3A and the corresponding natural language transformation in Figure 3B (see Miller[2,35] for further discussion).

summary

The enormous and profound differences so readily apparent in comparisons of text and program samples signal differences that may not be so obvious—e.g., the way exceptions are handled, how values are assigned, the manner in which parameters are passed, the defaults for unspecified function operands. Almost all of the evidence thus points to fundamental, even incompatible, differences between natural and programming specifications of procedures. We believe that all of the activities associated with generating, comprehending, and using natural language procedures are deeply rooted in long-developed and practiced habits; changing so firmly entrenched a manner of specification is akin to asking people to change the way they walk and talk.

Conclusions

Our objective in this study was to obtain detailed empirical information about the nature of natural language "programming" to bring to bear on the issues of increasing the usability of computer language interfaces. Although we expected numerous difficulties to be detected concerning the potential of actually implementing a system to interpret natural language programs, we were not prepared for the magnitude of what we see as being the three major obstacles: *style*, *semantics*, and *world knowledge*. Concerning the first, there is little way in which the vast differences in styles could be increased: programming-language style is simply alien to natural specification. With respect to semantics, we also were unprepared to find out the extent to which the selection of the appropriate "meaning" (to a word, phrase, or sentence) is dependent upon the immediate and prior context. And as for world knowledge, we suspect that the extent to which shared experiences and knowledge are critical to procedural communication and understanding among people has barely been hinted at by our present data.

These findings would seem to remove from active consideration the notion of radically improving computer usability by a totally unrestricted natural language interface: the technology to accomplish this is simply not there, and probably will not be, even in approximate form, for a number of years. Aside from the technical difficulties, some other aspects of our study make us skeptical that merely (!) providing a natural language interface would permit anyone to become a programmer, capable of specifying the procedures necessary to develop complex computer programs. We suspect that what would happen is that a lot of people would

be able to generate easily lots of programs that did not do what they were supposed to—because the subtle conceptual complexities of the problem were not appropriately understood and dealt with. If there is any single thread running through our other studies of programming and design (e.g., Miller[2]), it is that the way to achieve better-quality output is to provide people with tools that *structure* the problem and the implementation processes. We suspect that this principle would hold equally well for programming and for a natural language interface.

Although our results lead us to be negative towards the idea of an unrestricted natural language interface, the same results (along with other work) *do* give us some hope about the two obvious compromises: (1) implementing a natural language interface subject to several constraints, and (2) modifying programming languages to include more natural language features. Our view is that such compromises have in common two assumptions: (1) they would include provision for interactive clarification of user input and system output (such as proposed by Codd[36]), and (2) they would best be implemented, at least initially, as *add-on features* (or overlaid interfaces) to existing languages rather than requiring the design of completely new and stand-alone languages.

Concerning the first compromise, a *constrained natural interface*, we believe there are selected application areas for which this approach would be quite workable and desirable—e.g., accounting or computation (see Biermann and Ballard[1]) and highly standardized office tasks, including data base maintenance or query.

For this possibility we would recommend three kinds of constraints. First, the activity supported ought to be limited for use by persons familiar with and experienced in the tasks; presumably they would already be quite accustomed to the idea of algorithmic procedures and would also be able to tolerate the other constraints. Second, rather strong restrictions should be placed on the users' vocabularies, limiting each word also to only one meaning. A basic vocabulary of perhaps 500 words might suffice for about 80 percent of the interaction vocabulary required in the envisioned tasks. Third, moderate restrictions should be placed upon the *syntax* of the user's input; constraints would include limitations on sentence length, number of qualifications per noun, sentence type, and prohibition of complex constructions (such as nominalizations, complements, appositives, etc.). An interface developed with these and the more general constraints should not only be sufficiently powerful for an acceptable range of tasks but should also "feel" quite comfortably "natural," at least to the experienced user.

With respect to the second compromise, *making programming languages more natural*, we believe there are several "language

encoding mechanisms" that are sufficiently well-defined and "stand-alone" to be implemented as features in computer languages. In view of the strong degree to which references involved context, the potentially most useful such mechanisms would be those that permitted contextual referencing. Among such mechanisms are those of pronominalization, ordination ("first, second, . . ."), discriminant feature referencing (picking the salient or unique feature; as with two black blocks, one larger, saying "the *large* one"), relative referencing ("the *next* record . . ."), and collective referencing ("for *all of these*"). Conventions for the details of resolution and scope would have to be worked out and communicated, of course, but implementation would appear to be quite feasible.

A second type of implementable mechanism would be the development of high-level programming "macros" which—in appearance, variation, and syntax—would be highly similar to natural language expressions and would accomplish the same extensive operations as their natural counterparts. For example, two such macros that one can imagine being implemented for the problems of our study are a "find" and a "call-it" macro. With the key words given in capitals and the variable fields in lower case, the format for the *find* might be

FIND <what> IN <where> WHICH HAVE
<attribute> <operator> <value>

(e.g., FIND names IN file 3 WHICH HAVE
title = photographer)

And the "call-it" macro might be

CALL RESULT <what> and < do what to> IT
(e.g., CALL RESULT list 1 AND alphabetize IT)

Such macros could at least be defined and implemented for stable application activities and might well be made more general operations of the added-to programming language itself.

Summing up, we believe the present study illustrates the great difficulties opposing implementation of an unconstrained programming-language interface. Nonetheless, important beginning steps have been identified which *can* be taken now to improve the natural ease of use of computer systems.

ACKNOWLEDGMENT

This paper is a highly augmented and rewritten version of an earlier internal report (with Curtis Becker as a joint author; IBM Research Report RC 5137, 1974). The research was supported, in part, by a contract from Engineering Psychology Programs, Office of Naval Research.

CITED REFERENCES

1. A. W. Biermann and B. W. Ballard, "Toward natural language computation," *American Journal of Computational Linguistics* **6,** 71–86 (1980).
2. L. A. Miller, *Behavioral Studies of the Programming Process*, Research Report RC 7637, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598 (1978).
3. E. Dijkstra, "On the design of machine independent programming languages," *Annual Review in Automatic Programming* (R. Goodman, Editor), III, 27–42, Pergamon Press, New York (1963).
4. E. Dijkstra, "Some comments on the aims of MIRFAC," *Communications of the ACM* **7,** 190 (1964).
5. I. D. Hill, "Wouldn't it be nice if we could write programs in ordinary English—or would it?" *Computer Bulletin* **16,** 306–312 (1972).
6. M. Elson, *Concepts of Programming Languages*, Science Research Associates, Inc., Chicago (1973).
7. S. R. Petrick, "On natural language based computer systems," *IBM Journal of Research and Development* **20,** No. 4, 314–325 (July 1976).
8. J. E. Sammet, *Programming Languages: History and Fundamentals*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1969).
9. J. E. Sammet, "Roster of programming languages," *Computers and Automation* **20,** 6–13 (1971).
10. M. Halpern, "Foundations of the case for natural-language programming," *IEEE Spectrum* **4,** 140–149 (1967).
11. W. A. Woods, "A personal view of natural language understanding," in "Natural Language Interfaces," *ACM SIGART Newsletter*, 17–20 (February 1977).
12. G. E. Heidorn, "Automatic programming through natural language dialogue: A survey," *IBM Journal of Research and Development* **20,** No. 4, 302–313 (July 1976).
13. N. Findler (Editor), *Associative Networks: Representation and Use of Knowledge by Computers*, Academic Press, Inc., New York (1979).
14. R. J. Brachman and B. C. Smith (Editors), Special Issue On Knowledge Representation, *ACM SIGART Newsletter*, No. 70 (1980).
15. L. A. Miller, "Project EPISTLE: A system for the automatic analysis of business correspondence," *Proceedings of the First Annual National Conference on Artificial Intelligence* (August 1980), pp. 280–282.
16. L. A. Miller, G. E. Heidorn, and K. Jensen, *Text Critiquing with the EPISTLE System: An Author's Aid to Better Syntax*, Research Report RC 8601, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598 (1980).
17. M. E. Sime, T. R. G. Green, and D. G. Guest, "Psychological evaluation of two conditional constructions used in computer languages," *International Journal of Man-Machine Studies* **5,** 105–113 (1973).
18. T. R. G. Green, "Conditional program statements and their comprehensibility to professional programmers," *Journal of Occupational Psychology* **50,** 93–109 (1977).
19. L. A. Miller, "Programming by non-programmers," *International Journal of Man-Machine Studies* **6,** 237–260 (1974).
20. M. E. Atwood, H. R. Ramsey, J. N. Hooper, and D. A. Kullas, "Annotated bibliography on human factors in software development," *Army Research Institute Technical Report*, P-79-1 (1979).
21. A. Newell and H. A. Simon, *Human Problem Solving*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1972).
22. P. R. Michaelis, A. Chapanis, G. D. Weeks, and M. J. Kelly, "Word usage in interactive dialog with restricted and unrestricted vocabularies," *IEEE Transactions on Professional Communication* **PC-20,** 214–221 (1977).
23. J. E. Sammet, "The use of English as a programming language," *Communications of the ACM* **9,** 228–230 (1966).
24. J. E. Sammet, "Programming languages: History and future," *Communications of the ACM* **15,** 601–610 (1972).

25. B. M. Leavenworth and J. E. Sammet, *An Overview of Nonprocedural Languages*, Research Report RC 4685, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598 (1974).
26. G. Gerbner, O. Holsti, K. Krippendorff, W. J. Paisley, and P. J. Stone, *The Analysis of Communication Content*, John Wiley & Sons, Inc., New York (1969).
27. P. J. Stone, D. C. Dunphy, M. S. Smith, and D. M. Ogilvie, *The General Inquirer: A Computer Approach to Content Analysis*, MIT Press, Cambridge, MA (1966).
28. H. Kucera and W. N. Francis, *Computational Analysis of Present-Day American English*, Brown University Press, Providence, RI (1967).
29. M. A. K. Halliday and R. Hasan, *Cohesion in English*, Longman Group Ltd., London (1976).
30. M. J. Kelly and A. Chapanis, "Limited vocabulary natural language dialogue," *International Journal of Man-Machine Studies* **9**, 479–501 (1977).
31. H. F. Ledgard, "Ten mini-languages: A study of topical issues in programming languages," *Computing Surveys* **3**, 115–146 (1971).
32. D. E. Knuth, *An Empirical Study of FORTRAN Programs*, Research Report RC 3276, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598 (1971).
33. A. V. Aho and J. D. Ullman, *The Theory of Parsing, Translation, and Compiling*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1972).
34. T. A. Sebeok, *Style in Language*, M.I.T. Technology Press, Cambridge, MA (1960).
35. L. A. Miller, *Natural Language Procedures: Guides for Programming Language Design*, Reprint of talk delivered at International Ergonomics Association Meeting, University of Maryland, College Park, MD (July 1976).
36. E. F. Codd, "Seven steps to rendezvous with the casual user," in J. W. Klimbie and K. L. Koffeman (Editors), *Data Base Management: Proceedings of the IFIP TC-2 Working Conference on Data Base Management Systems*, North-Holland Publishing Co., Amsterdam (1974).

The author is located at the IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598.

Reprint Order No. G321-5146.