Title
Recursive Functional Programming for Students in the Humanities and Social Sciences

Authors
Brown, John Seely
Rubenstein, Richard

Publication Date
1973

Peer reviewed

RECURSIVE FUNCTIONAL PROGRAMMING FOR STUDENTS IN THE HUMANITIES AND SOCIAL SCIENCES

by

John Seely Brown
Richard Rubinstein

TECHNICAL REPORT #27 - January,1973

TABLE OF CONTENTS

LIST OF FIGURES

Acknowledgments

We would like to thank Robert Bobrow, Wally Feurzig, and Seymour Papert for various discussions about the pedagogical uses of LOGO. We are also indebted to the students who have taken SS-15 for providing us with numerous insights into the vices and virtues of this approach to programming.

This paper is based on a talk by the first author at the National ACM-72 Conference in Boston, August 14-17, 1972.

NOTE

This paper is also being reproduced as a Social Science Working Paper.

Introduction

A computer can be a great device for capturing the imagination of students, yet for various reasons few students in the Arts, Humanities, or Social Sciences seem to be amused by these giant wizards. Being somewhat idealistic, the authors--at the University of California at Irvine--made another stab at the well-honored problem of introducing these students to computing.

We knew that we could construct all kinds of "games" ranging from an enhanced Eliza [1] (a simulated Rogerian therapist) to sophisticated chess programs [2] and that students could be easily persuaded to play these games. Such ploys can often help students to overcome certain anxieties about computers, but this was not our primary interest. Nor were we especially concerned with teaching students how to "program" per se. Instead we wished to present the computer as a medium in which students can formulate ideas and engage in abstract reasoning.

There are numerous students in the Arts and Social Sciences who are interested in and talented at logical and analytical reasoning. Often these students have rejected the physical sciences and mathematics because they dislike the rigidity of mathematical structures. We suggest that the computer can accomodate a host of interesting meta-languages which appear less restrictive and formidable to these students than the language of mathematics. By introducing these languages as a convenient medium for expressing formal theories or models of a logical but non-mathematical nature, we hoped to provide a context in which these students could generate complete and unambiguous descriptions of their ideas.

In addition, some of these meta-languages can in themselves be sources of powerful theoretical ideas. Mastering them permits the student to experience the "Aha!" phenomenon in a formal, non-mathematical domain.

Because of the orientation of these students, we could not count on their being willing to tolerate inconviences inherent in most computing systems. Not wanting to prematurely turn them off, we were adamant about satisfying the following:

> Maxim: The computer language (system) must be friendly.

If this seems too obvious, we should note that what constitutes a "friendly" system is just now becoming a subject of study in computer science. For our purposes, we felt that a "friendly" language (system) would be truly interactive [3], have excellent debugging and editing facilities, render meaningful error statements, and possess a uniform syntax with few idiosyncratic restrictions (e.g. limits on the lengths of variable names). Since few of our students had any interest in numeric problems, we also felt that "friendliness" would imply a language which excelled in symbolic processing.

Fortunately a language exists which embraces many of these requirements. It is no surprise to discover that this language was invented for use by children. How natural! Of course a ten year old child is not going to tolerate all the petty restrictions found in most current systems. The language we chose was LOGO [4]. Any reader who is not familiar with this elegant little language is referred to Appendix 1 for a brief introduction.

In the next section we will discuss our course and some of the techniques we used. We will proceed by example, describing some of the problems we assigned and discussing the motivation behind each. We will consider how some theoretical ideas about computation can be explained intuitively and how, by choosing some metaphors that are particulary meaningful to the non-science student, these abstract concepts can be presented effectively. In the last section we will describe some of the limitations and hinderances we encountered and offer some suggestions for circumventing them in the future.

An Approach

Although our primary goal was not to teach programming per se, we did require our students to write and debug programs. Each week's assignment required about five hours of work. The homework problems were designed to build on each other and often involved extending the language by adding new functions and predicates. Ideally, by the end of the course, each student would have created his own version of LOGO. Since LOGO is a functional language, these extensions are syntactically indistinguishable from the original set of primitives. This helped foster a notion of custom tailoring the language.

In order to encourage a certain style of analyzing problems, we deleted two constructs from LOGO (which were to be reinstated later in the course). First, we eliminated the GOTO statement. This meant that the only way a process could be repeated was through recursion. The second deletion, consistent with the first, was the assignment statement (i.e. MAKE "X" "A B C"). Our purpose in this was not to be pedantic. Rather, we felt that students could grasp subtle, non-trivial aspects of recursion better if they were forced early to write recursive programs. (Students who already know FORTRAN-like languages often take months to gain the same familiarity with recursion that a neophite acquires in a few weeks.)

We do not wish to argue the virtue of recursive versus iterative procedures from a programming point of view. Nevertheless, through recursive programming we quickly immersed the students in:

A) some interesting theoretical problems which are more logical than mathematical, and

B) some of the deeper problems of how names (variables) take on meanings (values).

This latter problem is most apparent in a recursive context where the student is often baffled by what appear to be paradoxes (i.e. variables take on different values without specific reassignment). Once these "paradoxes" are encountered, a full treatment of how names take on meanings is then more interesting and informative.

In conjunction with the removal of the assignment and "GOTO" statements, we imposed three Cardinal Rules:

1. No function (procedure) can be more than seven lines long (±2 for psychologists).

This, our most important rule, encouraged the student to decompose his problems hierarchically and then solve them by stepwise refinement. We hoped that by making this restriction we would get the students to use a top-down approach to problem solving. (See Wirth [5] for an excellent technical discussion of this approach.)

2. The name of every function should be semantically meaningful. (Remember that in LOGO names can be of arbitrary length.)

This rule not only helped us in assisting a student to debug his program, it also helped him to clearly delineate the purpose of each of his functions. In addition, it helped to keep functions short, thus reinforcing Rule 1.

3. Access to data must be done through function calls.

This rule was not introduced until fairly late in the course since "accessing a piece of data" had hardly any meaning to a beginning student. (The reasons for this rule are discussed later on in this paper.)

Environment

Before launching into a description of some typical problem sets, we would like to comment on some environmental factors that proved to be extremely important.

The first year we taught this course, we had four terminals placed on a large square table. These terminals were more or less dedicated to the LOGO students and the precedent was established that the students could help each other as much as they wanted. We placed no time limits on the use of the machine. This was possible only because LOGO is so inexpensive to use.[6] In addition, as we had few available manuals; we encouraged students to try out a command or procedure to see what it did instead of consulting a manual. As a result, the students were always busily showing each other newly discovered "secrets" of LOGO. The side effect of this was that they were learning preliminary skills in debugging -- i.e. given a procedure, discover what it *really* does.

A Sequence of Problems

Since LOGO contains only a few primitive procedures (we use the term "procedure" interchangeably with "function"), it was reasonable to ask students to create some new ones for their first assignment:

> Write a predicate to be called MEMBERP which is to have two arguments and which checks to see if its first argument is contained in its second. If it is, then MEMBERP should output "TRUE". Otherwise it should output "FALSE".

The purpose of this assignment was twofold. First, it exposed the student to the simplest form of recursion. Second, it called to their attention the possibility of adding new predicates, as well as operators, to the language. We also established the naming convention that any procedure which is to behave as a predicate (i.e. outputs "TRUE" or "FALSE") should have a "P" as the last letter in its name. This helped the students to remember which functions could follow a TEST command.

A solution to this problem might be:

CHECKS FOR TERMINATING CONDITION OF THE RECURSION

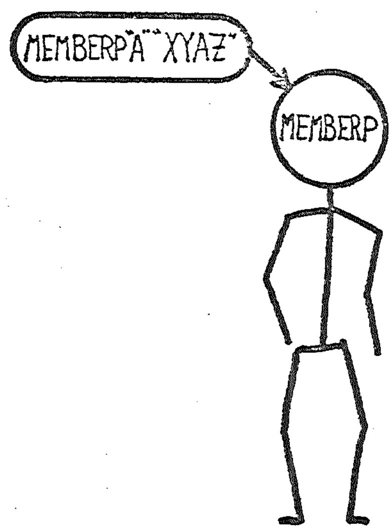
```
TO MEMBERP /ELEMENT/ /SET/
10  TEST EMPTYP /SET/
20  IF TRUE OUTPUT "FALSE"
30  TEST IS /ELEMENT/ FIRST OF /SET/
40  IF TRUE OUTPUT "TRUE"
50  OUTPUT MEMBERP OF /ELEMENT/ AND BUTFIRST OF /SET/
END
```

CHECKS IF THE CURRENT FIRST ELEMENT OF /SET/ EQUALS THE DESIRED ELEMENT

RECURSES WITH THE CURRENT /SET/ MINUS ITS FIRST ELEMENT

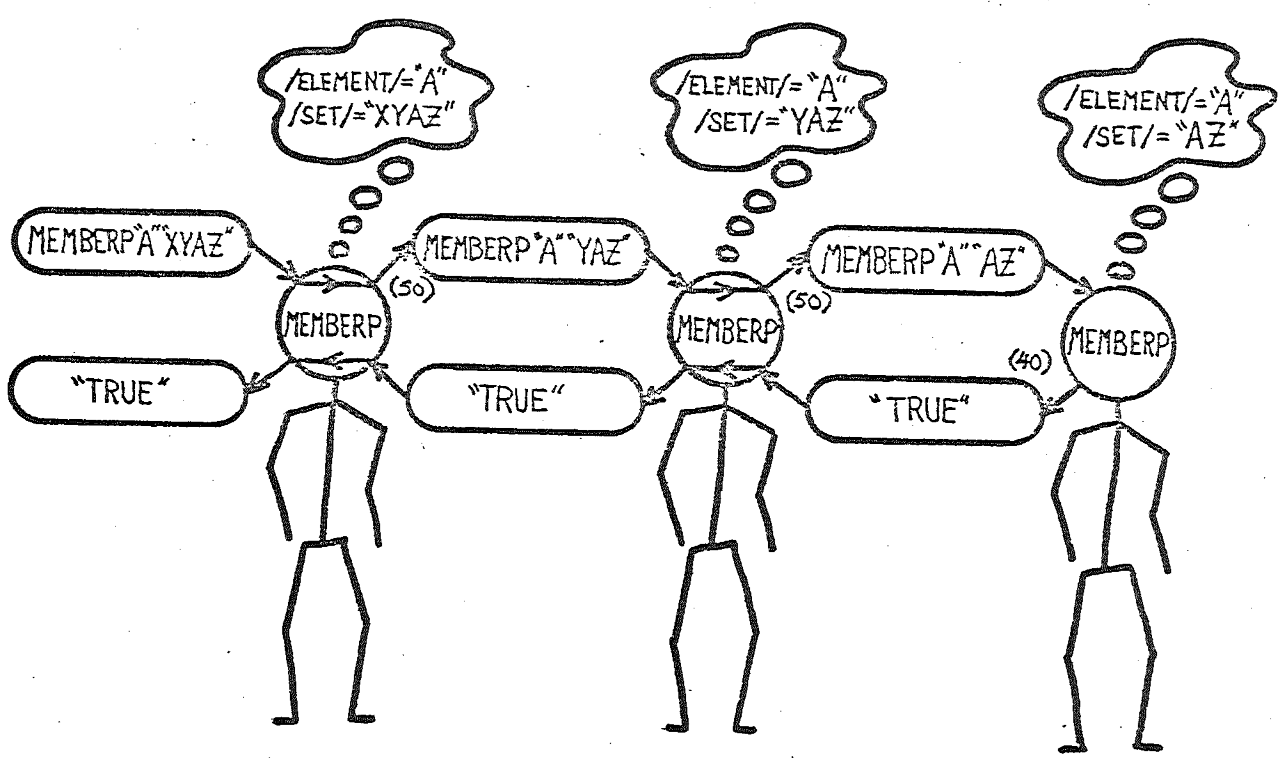
In order to illustrate and explain the underlying structure of recursive functions like the above, a diagramming convention was introduced along with some helpful terminology .[7] We consider the "MEMBERP" predicate to be the name of a "little brother" who has numerous identical twin brothers -- all called by the same name, MEMBERP. This family of MEMBERP brothers works as follows: suppose we make a request of a MEMBERP brother, i.e.

Figure 1 -- A MEMBERP Brother



The first MEMBERP brother executes his definition by first testing if /SET/ has any elements. It is not empty, so he tests if /ELEMENT/ (i.e. "A") is first of "XYAZ". Since "A" is not equal to "X", "IS" outputs "FALSE" to "TEST" (line 30) causing line 40 to fail. We are now at line 50. But in order for this first little brother to complete line 50, he must call for assistance from one of his twins. He requests that his brother tell him the answer to a slightly simpler problem; he asks him to compute: MEMBERP "A" "YAZ". This process continues with each brother calling on another brother to do a slightly simpler task until finally a brother is called who can complete his simpler task (possibly the null task). This last brother then sends his answer back to the brother that called him enabling that brother in turn to finish, (i.e. complete his line 50), and so on:

Figure 2 -- A Chain of MEMBERP Brothers



The explanation omits one very important construct which we dub "conceptual clouds." A conceptual cloud is used to determine the "world-view" of a particular brother. That is, it defines what he knows or what meanings he ascribes to the names in his particular world. Each MEMBERP brother has a conceptual cloud that looks like those above the men in Figure 2. So as far as the first brother is concerned, the meaning of /SET/ (what /SET/ denotes, i.e. the THING OF "SET") is the string "XYAZ". His next brother in line has a different world-view: in his conceptual cloud /SET/ has the meaning "YAZ".

Surely by now the reader must think this description is trivial. We ask indulgence, for without such detailing the next problem would probably stump many of us. Its solution is non-trivial without considering the world-views of each little brother. Pushing toward a deeper understanding of recursion, we formulated the next task:

> Write a procedure (say MAKEPRETTY) which prints a given string and then prints it again chopping off the last letter and so on until there is one letter left. At that point it then starts backing up by printing two letters, then three, and so on. For example:

```
*MAKEPRETTY "ABC"
ABC
AB
A
AB
ABC
```

Writing a procedure to achieve the first part is simple:

```
TO MAKEPRETTY /X/
10 TEST EMPTYP OF /X/ —— CHECKS FOR TERMINATING CONDITION OF THE RECURSION
20 IF TRUE STOP
30 PRINT /X/
40 MAKEPRETTY BUTLAST OF /X/ —— RECURSES ON TRUNCATED STRING OF LETTERS
END
```

Such a procedure given "ABC" as an input would print out:

```
ABC
AB
A
```

The catch is now to unfold this process by somehow recapturing what /X/ used to be. A particularly elegant solution is to add just one line to the above procedure, namely:

```
    TO MAKEPRETTY /X/
    10  TEST EMPTYP OF /X/
    20  IF TRUE STOP
    30  PRINT /X/
    40  MAKEPRETTY BUTLAST OF /X/ |—— RETURNS FROM RECURSION
==> 50  PRINT /X/                 |   AND THEN PROCEEDS TO
    END                               PRINT /X/
```

The reason this modified procedure works is that as each MAKEPRETTY brother returns to his calling brother, that brother still retains in his conceptual cloud exactly the desired information to complete his task. Build the little brother diagram with the appropriate conceptual clouds and see how well it fits into place. Note that the MEMBERP procedure uses a form of recursion so trivial that converting it to an iterative procedure is quite easy. The MAKEPRETTY procedure presents quite a different situation. Converting this procedure into an iteration would require introducing temporary storage locations, indicies, and so on.

At this point in the course, rather than develop any further the structure of conceptual clouds and their relationship to names, we gave a fairly heavy dose of programming assignments. Examples of these assignments are:

A) Using the MEMBERP predicate write a VOWELP predicate which determines if a given letter is a vowel. For example:

```
TO VOWELP /L/
10 OUTPUT MEMBERP /L/ "AEIOU"
END
```

B) Write a set of procedures which remove all vowels from each word in a sentence. Use these procedures to explore how well one can recognize the words of a sentence without vowels printed as contrasted with devoweled words in isolation:

```
TO SCAN /S/
10 TEST EMPTYP /S/
20 IF TRUE OUTPUT /EMPTY/
30 OUTPUT SENTENCE OF (REMOVE-VOWEL FIRST OF /S/)
                AND (SCAN OF BUTFIRST OF /S/)
END
```

```
TO REMOVE-VOWEL /W/
10 TEST EMPTYP /W/
20 IF TRUE OUTPUT /EMPTY/
30 TEST VOWELP FIRST OF /W/
40 IF TRUE OUTPUT REMOVE-VOWEL BUTFIRST OF /W/
50 OUTPUT WORD OF (FIRST OF /W/)
             AND (REMOVE-VOWEL BUTFIRST OF /W/)
END
```

We have included some typical solutions to these problems in order to impart some feeling for the simplicity of LOGO. In fact, most solutions are so simple and the amount of typing so minimal that often a student will explore different strategies for solving the same problem. This in turn often provokes discussions of what makes one solution "better" than another.

The next problem involves a short excursion into number theory. Our purpose was to show the student how one might write some quick and dirty procedures in order to test a hypothesis. Although we were intially hesitant to introduce any numeric or algebraic problems, this problem was well liked and helped tie together many of the points developed during the first few weeks of the course.

> Problem: Explore the following conjecture:
>
> Any number can be made into a symmetrical number by the following operations: First test to see whether the number is already symmetric. If so -- you're done. Otherwise, add to this number its own reverse and repeat. For example, suppose we choose the number 124. Since 124 ≠ 421, it is not already symmetric. So, add 421 to 124 which gives 545. Is 545 symmetric? -- YES! For another example, try 79. 79 ≠ 97, So add 97 to 79 which gives 176. But 176 is not symmetric, so add to it 671 which gives 847. Will this process end by reaching a symmetric number?

Just prior to this assignment the students had written a procedure (called "REVERSE") which forms the reverse of a word:

```
*PRINT REVERSE OF "ABC"
CBA
```

This procedure was typically written:

```
TO REVERSE /W/
10 TEST EMPTYP OF /W/
20 IF TRUE OUTPUT /EMPTY/
30 OUTPUT WORD OF (LAST OF /W/)          — PULL OFF THE LAST LETTER OF THE CURRENT WORD AND PLACE IT FIRST
              AND (REVERSE OF BUTLAST OF /W/) — RECURSE ON THE REMAINDER OF THE WORD.
END
```

Students quickly realized that they could use the REVERSE procedure to test a number for symmetry:

```
TO SYMP /NUMBER/
10 OUTPUT IS /NUMBER/ REVERSE OF /NUMBER/ —— BY DEFINITION
END
```

Hence, to see whether a particular number can be made symmetrical, we can perform the following procedure:

```
TO CHECKSYMP /N/
10 TEST SYMP OF /N/
20 IF TRUE OUTPUT "TRUE"
30 OUTPUT CHECKSYMP OF SUM OF /N/ AND REVERSE OF /N/
END
```

We hoped that by this time most of our students could write such a program in less than half an hour, thus leaving them free to expand the assignment in various directions. For example, most of the students wrote a procedure to generate the integers and applied CHECKSYMP on each successive integer. Many went farther and computed distributions on the depth of the recursion for each number and then looked for patterns on this distribution. At some point, each student inevitably stumbled on the number "196" which leads to a recursion so deep that LOGO runs out of memory. This lead to endless discussions of whether such conjectures can be settled definitely with a computer, and if so, how. It also showed the students how easy it can be to synthesize procedures to probe a conjecture, thereby lessening dependence on "canned" programs.

Before turning the students loose on major projects (which occupied the last several weeks of the course), we introduced some preliminary ideas on representation of information, data structures, and how names take on meaning. For the student of cognitive psychology this was probably the most important aspect of the course, but nearly all the students found that this material contributed greatly toward their understanding of how representations of knowledge can be modeled.

Toward this end, we gave the students the task of creating the simplest form of a Quillian-like semantic net [8] and a fixed format question answerer which would use the net. The behavior of the question answerer is best characterized by example. Assertions are of the form:

Felix is a cat

Cat is an animal

and questions are of the form:

Is Felix a cat?

Is Felix an animal?

At this juncture we had to allow the use of the assignment statement (i.e. MAKE "X" "5"). The first apparent way to model the above assertions is to use the "MAKE" statement as:

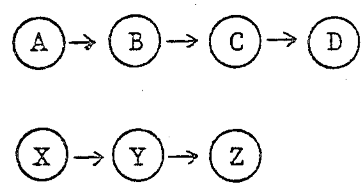
```
MAKE "CAT" "ANIMAL"
MAKE "FELIX" "CAT"
```

which results in

```
/FELIX/ IS "CAT"
/CAT/ IS "ANIMAL"
```

That is to say that the THING (or value) of "FELIX" is "CAT" and the THING of "CAT" is "ANIMAL". (In this same manner n-ary trees can be built, since the thing of a name can also be a sentence consisting of the daughters of the name.) This approach for linking information has some subtle problems that can challenge even the best student. For example, suppose we have the following data:

```
/A/ IS "B"
/B/ IS "C"
/C/ IS "D"
/X/ IS "Y"
/Y/ IS "Z"
```



in which the last node is being located by the recursive procedure:

```
TO FIND-LAST-NODE /Y/
10 TEST EMPTYP THING OF /Y/
20 IF TRUE OUTPUT /Y/
30 OUTPUT FIND-LAST-NODE OF THING OF /Y/
END
```

THE NEXT ELEMENT

Executing this procedure with the input "A" one gets back "D" as the answer. However, asking for FIND-LAST-NODE of "X" causes a baffling problem -- the procedure enters an infinite loop because the second FIND-LAST-NODE brother asks for the THING OF "Y". But we happened to use the symbol "Y" as the name of the input, (i.e. function argument) and this meaning of the variable takes precedence over any meaning assigned external to the function call. Hence when the THING OF /Y/ becomes "Y" we get into an infinite recursion. Most students stumbled across this apparent "bug" in one way or another and were totally at a loss to explain what could be happening.

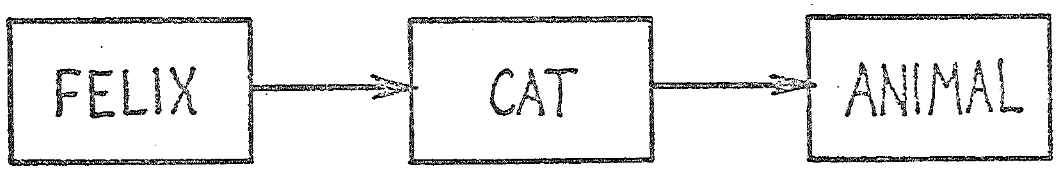
With their suspicions and curiosity aroused, we were in a position to develop the next powerful idea: how functional arguments and local variables are handled with push-down stacks. Once this idea was understood, the students were more willing to consider alternative techniques for linking information to names. We therefore introduced the notion of a property list as a means of storing information which is not local to the given procedure. Although LOGO has no direct mechanism for property lists, it is trivial for students to "provide" LOGO with such capabilities.

The key idea centers around LOGO names (or numbers) being arbitrarily long strings of letters. This allows us to synthesize a unique name from the given name of the property and the name of the variable to which the information is to be attached. For example, to represent the above data chain, let us invent the property "NEXT" and define it by the function "GET-NEXT", i.e.

```
TO GET-NEXT /X/
10 OUTPUT THING OF (WORD OF "$NEXT$" AND /X/)
END
```

SYNTHESIZES APPROPRIATE NEW SYMBOL

The above chain would have the same modeling structure, i.e.,



but its implementation would look like:

```
/$NEXT$FELIX/ IS "CAT"

/$NEXT$CAT/ IS "ANIMAL"
```

(The "$" symbols are used simply to reduce the chance that such a name could crop up in another context.)

To store such information one might write another one line procedure called PUT-NEXT:

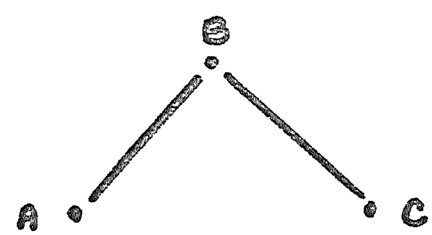
```
TO PUT-NEXT /NAME/ /VALUE/
10 MAKE (WORD OF "$NEXT$" AND /NAME/) /VALUE/
END
```

With these procedures FIND-LAST-NODE could be rewritten:

```
TO FIND-LAST-NODE /Y/
10 TEST EMPTYP OF GET-NEXT /Y/
20 IF TRUE OUTPUT /Y/
30 OUTPUT FIND-LAST-NODE OF GET-NEXT /Y/
END
```

From this example one can guess that efficiency is not our main concern. Instead we are trying to convey a style of problem solving in which minor decisions can be postponed (i.e. How do we implement GET-NEXT?) and global solutions sketched out without concern for smaller details. Cardinal Rule 3, mentioned earlier, encouraged writing programs in this fashion. This problem solving method has the added advantage of allowing one to experiment with different representations of information simply by modifying a few functions.

Before turning the students loose on their final projects, we tried to unify some of the above ideas by giving them the classical task of ordering a list of distinct words. The approach we asked the students to consider was that of growing a tree representation of the list of words and then recursively searching the tree and outputing the ordered list. The tree is constructed so that all nodes in the left sub-tree of a node are lexicographically less than that node, and all nodes in its right sub-tree are greater. Once this is achieved, the student next must discover the simple but elegant way to search the tree, building up a sentence of the words in alphabetical order. Recursively stated, the key concept is to create a list (sentence) of all the nodes in the left subtree, then adjoin to the end of that list the current node and finally adjoin to the end of that list all the nodes in the right subtree. The left and right subtrees are smaller than the original tree and hence by recursing on the subtree we eventually encounter the trivial three-node subtree:



where A or C are possibly null

which, on applying the above steps, forms an ordered list (A B C).

The close correspondence between the way the tree is structured and the way it is searched is not accidental. We hoped that this example would illustrate how careful consideration of the data representation problem can contribute to the efficiency and ease of the total solution.

Postponing any decisions on how the tree is actually stored, we can write a top level ORDER procedure which walks over the tree gathering nodes in their alphabetical order. Note that there is no output until the walk is completed, at which time a sentence is returned which consists of the ordered words.

```
TO ORDER /NODE/
10 TEST TERMINAL-NODEP /NODE/
20 IF TRUE OUTPUT /EMPTY/
30 OUTPUT SENTENCES OF (ORDER GET-LEFT /NODE/)
                    AND (GET-VALUE OF /NODE/)
                    AND (ORDER GET-RIGHT /NODE/)
END
```

Of course, before ORDER could be executed we would have to specify the four data accessing functions TERMINAL-NODEP, GET-LEFT, GET-VALUE, and GET-RIGHT. (Appendix 2 shows one possible implementation of these functions and a tree on which ORDER could be applied.)

One of the more interesting aspects of this problem is that the shape of the tree depends upon the order of the initial list of words. Once students discovered how to write a tree-growing procedure, we posed such puzzles as finding the orderings of the the initial list of words that caused the most lop-sided or well balanced trees to be generated. By using the TRACE feature, the students quickly discovered a relationship between the shape of the tree and the depth of the recursion.

By the time we finished our discussion about sorting and trees, most of the students were ready to proceed on their own projects, but some were discouraged. For the latter students, we provided a two-week excursion into computer art. For the former, we posed a choice of projects. Some of these are summarized in the next paragraphs.

Projects

One of the simplest projects involved the generation of political slogans over a basic sentence structure which has "slots" that are to be filled in. Each kind of slot has an associated list of sub-expressions and the program simply selects at random an element from each list and places the expression in the appropriate slot. As a programming exercise this project is undemanding. Constructing good lists of sub-expressions, however, introduces the student to the problems of semantic anomalies. The immediate result of this project was to impress the student with the tight structure and slight content of slogans. Our primary purpose, however, was to show how easily a computer can be caused to generate something which, superficially at least, resembles an act of intelligence.

A more complex project was to create a procedure which could randomly generate sentences with a non-deterministic, finite-state grammar. If the student completed this task satisfactorily, we suggested that he invert the process and write a procedure which could decide whether or not a sentence was in the given language. The non-deterministic nature of the grammar leads the student to the discovery and comparision of depth-first and breadth-first strategies. What is striking about this project is that while the logic involved is non-trivial, the process is inherently recursive and can be executed with a simple procedure. (See Appendix 3 for a typical solution.)

Another project originated in Rubinstein's experimental LOGO course for the blind. The student is given a dictionary and is asked to write a program which will print out the definition of a word, then expand each non-primitive word in that definition into its definition, and so on. The solution to this problem is, of course, inherently recursive. For the initial dictionary, we chose a non-circular but unusual subset of the Meriam Webster New Collegiate Dictionary:

/$dhow/ is "an Arab lateen-rigged vessel with a long
     overhang forward, a high poop, and an open waist"
/$poop/ is "deck above the upper deck abaft the mizzen"
/$mizzen/ is "mizzenmast"
/$mizzenmast/ is "aftermost mast of a two-masted vessel"
/$abaft/ is "to the rear"
/$lateen/ is "triangular sail, extended by a long yard,
     slung to the mast"
/$yard/ is "long spar"
/$spar/ is "mast"
/$waist/ is "that part of a vessel's deck between the
     quarter-deck and the forecastle"
/$forecastle/ is "forward part of vessel"

(See Appendix 4 for solution.)

Other less formal projects involved modifying the question answerer previously discussed to work with semantic nets containing cycles, and to answer questions like: "Tell me all you know about 'X'."

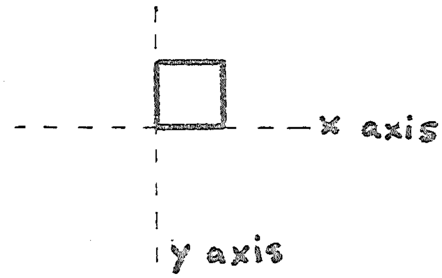
The above is only a small subset of the projects that have been attempted. All of the advanced projects involved symbol manipulation as contrasted with numeric computation. All of them shared the property that once a clean attack on the problem had been achieved, only a small amount of code was necessary to effect a solution.

Computer Art

Our motivation for introducing "computer art" was two-fold. First, those students who have trouble catching on to LOGO seldom have any feeling for what we call the structure of a process. For them, a function or a procedure is a black box whose components remain a mystery. Computer drawings often can clarify these issues since they enable the student to "see" inside the procedure by viewing on the plotter the result (or execution) of each step. In a sense, the plotter can act as a very detailed and useful trace feature.

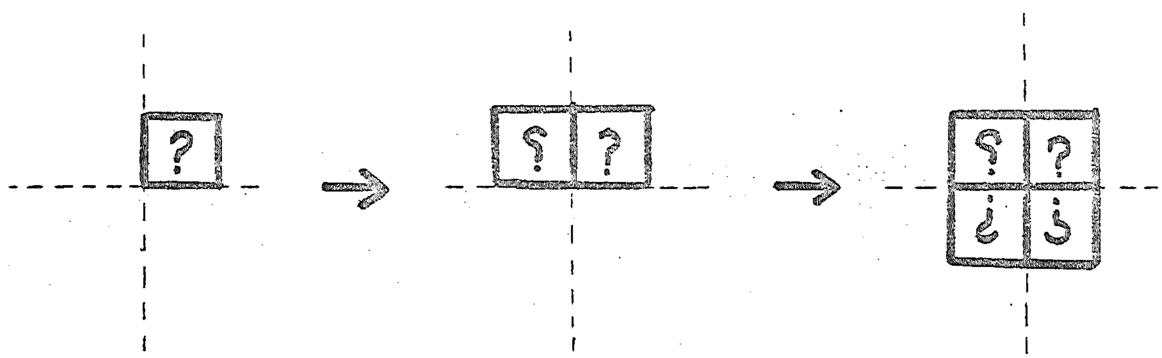
The second reason for introducing computer art applies equally to all of our students. Inevitably, they realize how dumb and mundane the operations of a computer really are. How, then, can a computer generate something new? How can it reveal properties of a theory heretofore unknown? In other words, how can a computer synthesize anything surprising (besides bugs)? Computer art provides a beautiful vehicle for the exploration of such questions. In the picture below, for example, one cannot help but be impressed by the totally unexpected Gestalt effect of a simple operation repeated a large number of times. First we asked our students to visualize the effect of this simple problem.

1. Consider a co-ordinate system with its origin at the center of the paper. Imagine a square in the first quadrant with side of 1 or 1 1/4 inches.



2. Draw the square on the paper and then rotate and shrink it a little. Repeat this operation several times (e.g. six).

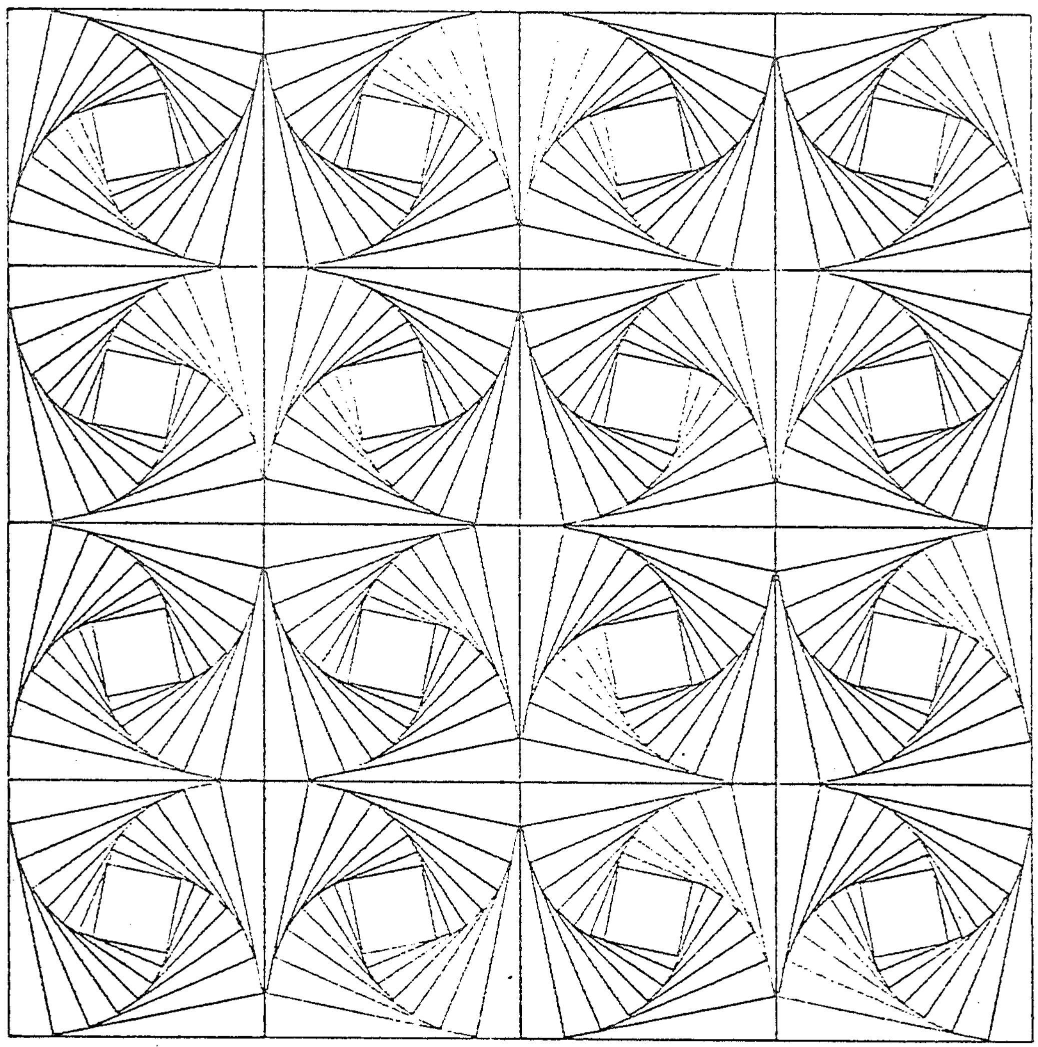
3. Copy the resulting figure into the other quadrants as follows: Reflect the figure in the Y axis, forming a figure with squares in the first and second quadrants. Now reflect this whole picture in the X axis, creating a figure with squares in each quadrant.



4. Move the resulting figure so that it is just in the first quadrant, resting on the axes, as the original square was.

5. Repeat step 3.

Figure 3 -- Resulting Design of Computer Art Description



We then permitted them to compute the effect and our point was made!

Computer drawings provide an interesting metaphor for linguistic processes. Throughout the course we stressed how a process could be used to describe a static situation. Consider the problem of describing the above figure down to the details of all its surface variations and complexities. Then let us note how simple it is to represent the structure of this surface pattern by the structure of the process. It is not too far fetched to think of the procedure generating the figure as the deep structure of this surface pattern. While this is just a metaphor, we feel there is some virtue in it. Visual figures that appear wildly complex often have simple, insightful descriptions when considered from the point of view of their generating procedures. A detailed account of the pedagogical uses of computer art is forthcoming in a doctoral dissertation by Richard Rubinstein.

BASIC

Although we had talked very little about other languages, our students expressed the wish to learn something about BASIC and how it differs from LOGO. Consequently we devoted the last week of class meetings to a survey of BASIC. We discovered that most of the students had little difficulty in understanding and using BASIC. The one concept foreign to them was the "FOR ... NEXT" structure, but they were able to see this as a trivial recursion. There was also some confusion caused by arrays. In some cases this was caused by their lack of understanding of matrix algebra. To our chagrin, however, even those with knowledge of matrix algebra were not able to see how to trivially introduce matrices into LOGO.

From our limited experience we have found that the transition from LOGO to BASIC is fairly easy for most students whereas the transition from BASIC to LOGO is often incredibly difficult. The problem in the latter case is that if a student initially experiences iteration, his understanding of recursion is often limited to the simplest form where the last statement of a function is a call to itself. Even within this context he becomes baffled if extra arguments must be introduced to keep track of the depth of recursion (simple indexing).

A Complication

We did encounter one difficulty with using LOGO that was completely unexpected. One impressive aspect of the language is that interesting, logically complex problems can be coded in surprisingly few lines. This was one of our reasons for choosing LOGO, but it was also a characteristic that caused several problems. In particular we discovered that most functions had only one or two key "schema." This meant that if a student couldn't figure out how to write a function, we could not slowly lead him down the path of discovery. Most hints we could give him would be either too obscure and therefore worthless or else divulge too much and lead immediately to the solution. It also created a problem when students helped each other, since any help at all often meant the two solutions would be isomorphic. To some extent the problem is not inherent in LOGO itself, but is inherent in the level of problems we thought reasonable for the course.

Conclusions

An important aim of the course was to help our students develop a sensitivity to precise problem specification and then to expose them to some problem solving strategies. The processes of decomposing a problem into subproblems, enhanced through the paradigm of functional programming and bottom-up debugging are clearly arts, requiring attention, effort, and experience to develop. Of course, the value of learning such methods rests heavily on their transferability to other areas of concern to the student. By stressing problems involving symbol manipulation instead of numeric computation we hope to increase the chance of such transferability. The notions that computers can be made to respond sensibly to input (such as English) and that precise specification can be made of how "information" is represented opens the door to thinking about the problems of long term memory, the representation of knowledge, and of course, the nebulous domain of natural language comprehension.

Any evaluation as to how well we met these stated aims or goals is incredibly difficult. Appendix 5 gives a summary of the students' responses to standard course evaluation forms provided by the ICS Department. Although these student reponses were surprisingly favorable we cannot be sure of the lasting effect of the course.

Footnotes

1. Joseph Weizenbaum, "Eliza -- A Computer Program for the Study of Natural Language Communication Between Man and Machine", Communications of the ACM, Vol. 9, No. 1, Jan., 1966.

2. R. Greenblatt, D. Eastlake, & S. Crocker, "The Greenblatt Chess Program", Proceedings of the Fall Joint Computer Conference, 1967.

3. Although most languages can be made to be interactive, few have been designed for promoting or facilitating meaningful interaction.

4. There are several centers developing LOGO and each has various documents describing their version of LOGO and, of course, their research. The following three reports provide a flavor of two of these centers:

   a. Wallace Feurzeig et.al., Programming-Languages as a Conceptual Framework for Teaching Mathematics, Report No. 2165, Vol. 1, Bolt Beranek and Newman Inc., June 30, 1971.

   b. Seymour Papert and Cynthia Solomon, "Twenty Things to Do With a Computer", Artificial Intelligence Memo No. 248, LOGO Memo No. 3, Mass. Institute of Technology, A. I. Laboratory, June, 1971.

   c. Seymour Papert, "Teaching Children To Be Mathematicians vs. Teaching About Mathematics", Aritificial Intelligence Memo No. 249, LOGO Memo No. 4, Mass. Institute of Technology, A.I. Laboratory, July 1971.

5. Niklaus Wirth, "Program Development by Stepwise Refinement", Communications of the ACM, Vol. 14, No. 4, April 1971.

6. The LOGO interpreter consumes 5K of shareable core on the PDP-10 and each student requires around 2K additional core for his programs and work space.

7. Wallace Feurzeig, Seymour Papert, et.al., Programming-Languages as a Conceptual Framework for Teaching Mathematics, Report No. 1889, Bolt Beranek and Newman Inc., Nov. 30, 1969.

APPENDIX 1

A Brief Sketch of LOGO

LOGO has two basic data types -- words and sentences. A word consists of an arbitrary sequence of letters, numbers, or other symbols, and a sentence consists of an arbitrary sequence of words.

Examples:

a) "ONE", "5", and "ABCDEFGHI" are all words where the quotes mean take the included sequence as a literal.

B) "THIS IS A SENTENCE" and "5 32 ABCDEF" are sentences.

Since LOGO specializes in non-numeric computations, it contains a number of procedures for tearing apart and concatenating data items. There are four basic functions for tearing data items apart: FIRST, BUTFIRST, LAST and BUTLAST. These act in the following manner:

```
FIRST OF "ABCD" --> "A"
FIRST OF "HI OUT THERE" --> "HI"
BUTFIRST OF "ABCD" --> "BCD"
BUTFIRST OF "HI OUT THERE" --> "OUT THERE"
LAST OF "ABCD" --> "D"
LAST OF "HI OUT THERE" --> "THERE"
BUTLAST OF "ABCD" --> "ABC"
BUTLAST OF "HI OUT THERE" --> "HI OUT"
```

(Note: the words "OF" and "AND" are noise words which are ignored by LOGO but increase the readability of the code.)

Each of these functions expects exactly one input and outputs the resulting answer.

Concatenation of objects is achieved in LOGO through two functions-- WORD and SENTENCE:

```
WORD OF "AB" AND "CD"      -->     "ABCD"
SENTENCE "AB" AND "CD"     -->     "AB CD"
```

Both the WORD and SENTENCE functions require exactly two inputs. Two closely related functions, "WORDS" and "SENTENCES", take an arbitrary number of inputs.

Inputs to a function need not be literal strings but may be the outputs of other functions. An example of such a composition of functions is:

```
PRINT BUTFIRST OF WORD OF "AB" AND "CD"
```

where WORD outputs "ABCD" which is the input to BUTFIRST which outputs "BCD" which is the input to PRINT. A more profound example of functional composition is found in the extension of LOGO's random number generator (which outputs one digit numbers) into a procedure which produces two digit random numbers. Since numbers are treated as strings of characters,

```
WORD OF RANDOM AND RANDOM
```

will output a two digit number which is the concatenation of the outputs of the two invocations of RANDOM. Balanced parentheses may be used for clarity.

PROCEDURES

In order to add the above construct to the language we need to define a one line procedure (called, perhaps, BIGRANDOM). This is accomplished by the "TO" statement:

```
TO BIGRANDOM
10 OUTPUT WORD OF RANDOM AND RANDOM
END
```

"OUTPUT" expects one input (WORD OF RANDOM AND RANDOM) and when executed terminates that level of that procedure, returning its input as the output of the procedure. If the procedure title contains variable names, these names become local to the procedure and allow referencing of inputs. Additional local variables may be declared with a LOCAL statement.

NAMES

In the case of LOGO, one speaks of names and the things that names name rather than of variables and their values.

Assigning things to names (i.e., values to variables) is performed with a MAKE statement. For example, the expression:

```
MAKE "SEX" "MALE"
```

assigns to the name "SEX" the value "MALE".

There are two ways of accessing the thing (value) of a name. If we want to print the thing of SEX, we could do either:

```
PRINT THING OF "SEX"
```

or:

```
PRINT /SEX/
```

That is to say that to fetch the value of X we can ask for either THING OF "X" or simply /X/. Since the thing of a name can be a name, we can have the following situation:

```
MAKE "ANIMAL" "DOG"
MAKE "DOG" "FIDC"
MAKE "FIDO" "MAN'S BEST FRIEND".
```

The function THING can be composed with itself an arbitrary number of times, e.g.:

```
PRINT "ANIMAL" --> ANIMAL
PRINT /ANIMAL/ --> DOG
PRINT THING OF /ANIMAL/ --> FIDO
PRINT THING OF THING OF /ANIMAL/ -->
    MAN'S BEST FRIEND
```

In using variables, one must always specify whether it is the name (variable) or the thing (value) that you are talking about. This is in contradistinction to most languages where an expression such as:

```
LET X = Y
```

means that the variable X is to be assigned the value of Y. Requiring students to always make this distinction is pedagogically nice when dealing with a complex symbolic structure in which the name of one object may be the value of another.

TESTS

LOGO also has a collection of predefined predicates which can be used to test for certain properties. Each predicate outputs either "TRUE" of "FALSE" and is usually used in conjunction with a TEST statement. Some basic predicates are:

ZEROP NUMBERP EMPTYP MINUSP WORDP SENTENCEP IS

All of these predicates expect one input except "IS" which requires two since it is checking for identity:

```
*PRINT IS "4" "A"
FALSE
```

or:

```
*PRINT IS FIRST OF "HI OUT THERE" WORD OF "H" AND "I"
TRUE
```

The TEST statement precedes a predicate (i.e. "TEST" has either "TRUE or "FALSE" as its input) and sets a truth flag which can later be read by the "IF TRUE" or "IF FALSE" statements. For example, suppose we want a procedure to form the absolute value of a number. This can be done as follows:

```
TO ABSOLUTE /N/
10 TEST IS FIRST OF /N/ "-"
20 IF TRUE OUTPUT BUTFIRST OF /N/
30 OUTPUT /N/
END
```

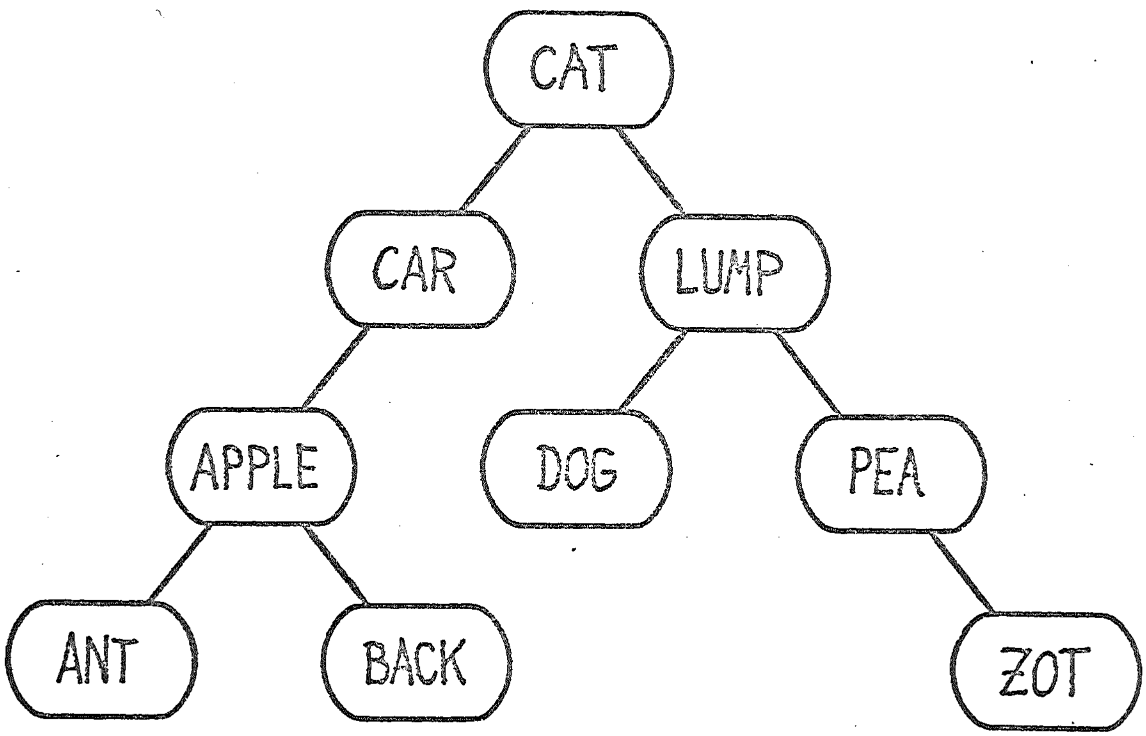
In other words, if /N/ is a negative number (i.e. -543) then line 10 sets the truth flag to TRUE causing line 20 to chop off the first character and then output its result (i.e. 543). Otherwise, line 30 is executed which simply outputs the number /N/.

APPENDIX 2

The ORDER Program

The purpose of the ORDER program is to walk a binary tree as structured below and return a list (sentence) of the nodes in alphabetical order. For this example, the following tree is used:

Figure 4 -- The Binary Tree



```
*LIST ALL
TO ORDER /NODE/
10 TEST TERMINAL-NODEP /NODE/
20 IF TRUE OUTPUT /EMPTY/
30 OUTPUT SENTENCES ( ORDER GET-LEFT /NODE/ ) AND ( GET-VALUE OF
    /NODE/ ) AND ( ORDER GET-RIGHT /NODE/ )
END

TO TERMINAL-NODEP /NODE/
10  UTPUT IS /NODE/ "*"
END

TO GET-LEFT /NODE/
10 OUTPUT FIRST OF THING OF /NODE/
END

TO GET-VALUE /NODE-NAME/
10 OUTPUT BUTFIRST /NODE-NAME/
END

TO GET-RIGHT /NODE/
10 OUTPUT LAST OF THING OF /NODE/
END


/$CAT/ IS "$CAR $LUMP"
/$CAR/ IS "$APPLE *"
/$LUMP/ IS "$DOG $PEA"
/$PEA/ IS "* $ZOT"
/$ZOT/ IS "* *"
/$APPLE/ IS "$ANT $BARK"
/$ANT/ IS "* *"
/$BARK/ IS "* *"
/$DOG/ IS "* *"


*TRACE ORDER
*PRINT ORDER OF "$CAT"
ORDER OF "$CAT"
  ORDER OF "$CAR"
    ORDER OF "$APPLE"
      ORDER OF "$ANT"
        ORDER OF "*"
        ORDER OUTPUTS ""
        ORDER OF "*"
        ORDER OUTPUTS ""
      ORDER OUTPUTS "ANT"
      ORDER OF "$BARK"
```

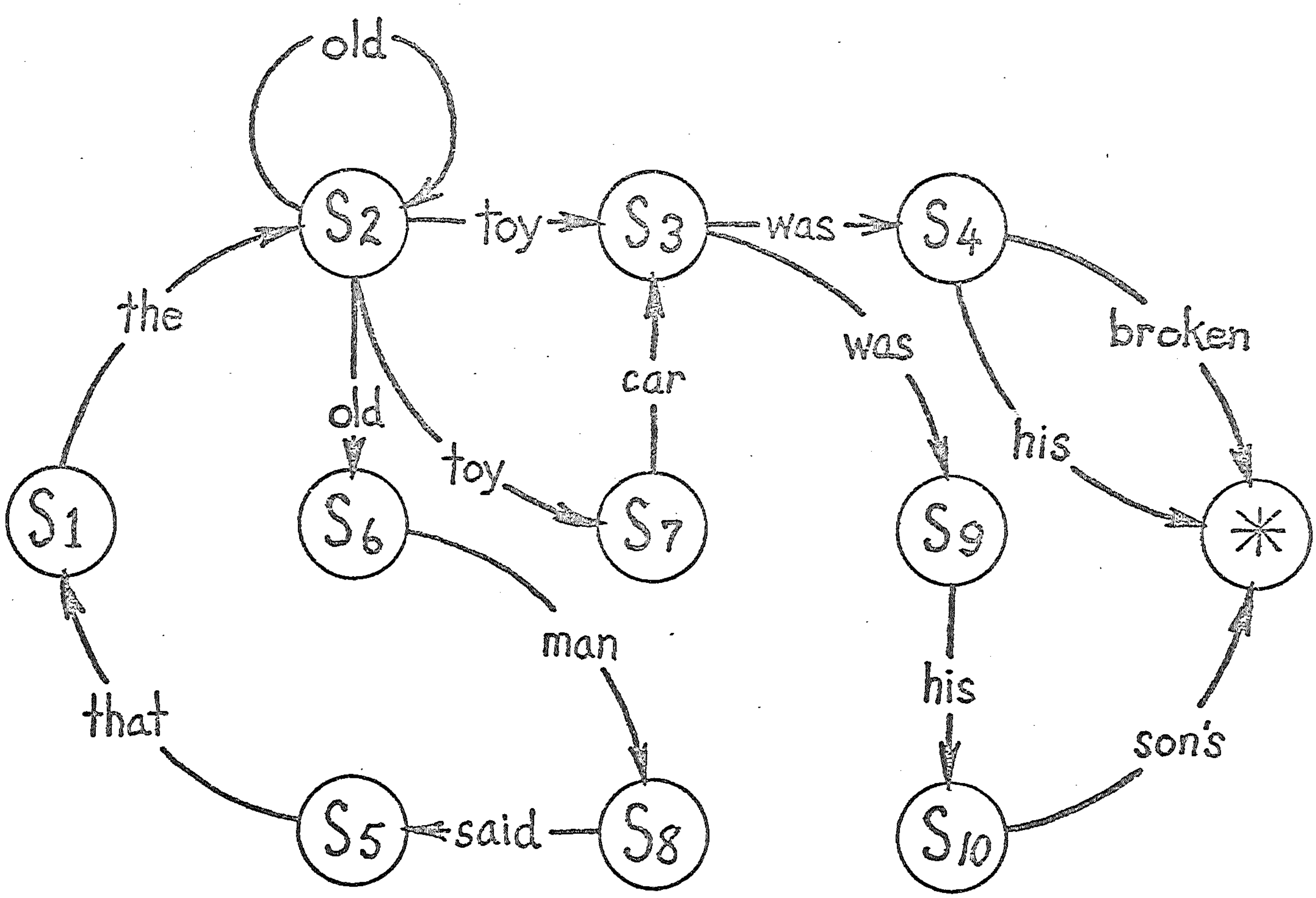
```
        ORDER OF "*"
        ORDER OUTPUTS ""
        ORDER OF "*"
        ORDER OUTPUTS ""
      ORDER OUTPUTS "BARK"
    ORDER OUTPUTS "ANT APPLE BARK"
    ORDER OF "*"
    ORDER OUTPUTS ""
  ORDER OUTPUTS "ANT APPLE BARK CAR"
  ORDER OF "$LUMP"
    ORDER OF "$DOG"
      ORDER OF "*"
      ORDER OUTPUTS ""
      ORDER OF "*"
      ORDER OUTPUTS ""
    ORDER OUTPUTS "DOG"
    ORDER OF "$PEA"
      ORDER OF "*"
      ORDER OUTPUTS ""
      ORDER OF "$ZOT"
        ORDER OF "*"
        ORDER OUTPUTS ""
        ORDER OF "*"
        ORDER OUTPUTS ""
      ORDER OUTPUTS "ZOT"
    ORDER OUTPUTS "PEA ZOT"
  ORDER OUTPUTS "DOG LUMP PEA ZOT"
ORDER OUTPUTS "ANT APPLE BARK CAR CAT DOG LUMP PEA ZOT"
ANT APPLE BARK CAR CAT DOG LUMP PEA ZOT
```

APPENDIX 3

The PARSE Program

The purpose of the PARSE program is to determine whether a given sentence is in the grammar determined by a finite-state transition network.

Figure 5 -- The Finite-State Transition Network



```
*LIST ALL

TO PARSEP /STR/
10 OUTPUT WALKP GET-OUTS "S1" AND /STR/
END

TO WALKP /OUTLIST/ /STR/
10 TEST BOTH ( TERMINALP /OUTLIST/ ) AND ( EMPTYP /STR/ )
20 IF TRUE OUTPUT "TRUE"
30 TEST EITHER ( EMPTYP /OUTLIST/ ) AND ( TERMINALP /OUTLIST/ )
40 IF TRUE OUTPUT "FALSE"
50 TEST IS ( FIRST /OUTLIST/ ) ( FIRST /STR/ )
60 IF TRUE TEST WALKP ( GET-OUTS OF FIRST OF BUTFIRST OF /OUTLIST/ )
   (BUTFIRST OF /STR/ )
70 IF TRUE OUTPUT "TRUE"
80 OUTPUT WALKP ( BUTFIRST OF BUTFIRST OF /OUTLIST/ ) AND /STR/
END

TO GET-OUTS /NAME/
10 OUTPUT THING OF ( WORD OF "$" AND /NAME/ )
END

TO TERMINALP /LIST/
10 OUTPUT IS /LIST/ "*"
END


/$S1/ IS "THE S2"
/$S2/ IS "OLD S2 TOY S3 TOY S7 OLD S6"
/$S3/ IS "WAS S4 WAS S9"
/$S4/ IS "BROKEN * HIS *"
/$S5/ IS "THAT S1"
/$S6/ IS "MAN S8"
/$S7/ IS "CAR S3"
/$S8/ IS "SAID S5"
/$S9/ IS "HIS S10"
/$S10/ IS "SON'S *"


*PRINT PARSEP OF "THE TOY WAS BROKEN"
TRUE
*PRINT PARSEP OF "THE OLD MAN SAID THAT THE OLD MAN SAID THAT THE TOY"
FALSE
*PRINT PARSEP OF "A B C"
FALSE
*PRINT PARSEP OF "THE OLD OLD TOY WAS BROKEN"
TRUE
```

```
*TRACE WALKP
*PRINT PARSEP OF "THE OLD MAN SAID THAT THE TOY CAR WAS HIS SON'S"
WALKP OF "THE S2" AND "THE OLD MAN SAID THAT THE TOY CAR WAS HIS SON'S"
  WALKP OF "OLD S2 TOY S3 TOY S7 OLD S6" AND "OLD MAN SAID THAT THE TOY
  CAR WAS HIS SON'S"
    WALKP OF "OLD S2 TOY S3 TOY S7 OLD S6" AND "MAN SAID THAT THE TOY
    CAR WAS HIS SON'S"
      WALKP OF "TOY S3 TOY S7 OLD S6" AND "MAN SAID THAT THE TOY CAR WAS
      HIS SON'S"
        WALKP OF "TOY S7 OLD S6" AND "MAN SAID THAT THE TOY CAR WAS HIS
        SON'S"
          WALKP OF "OLD S6" AND "MAN SAID THAT THE TOY CAR WAS HIS
          SON'S"
            WALKP OF "" AND "MAN SAID THAT THE TOY CAR WAS HIS SON'S"
            WALKP OUTPUTS "FALSE"
          WALKP OUTPUTS "FALSE"
        WALKP OUTPUTS "FALSE"
      WALKP OUTPUTS "FALSE"
    WALKP OUTPUTS "FALSE"
    WALKP OF "TOY S3 TOY S7 OLD S6" AND "OLD MAN SAID THAT THE TOY CAR
    WAS HIS SON'S"
      WALKP OF "TOY S7 OLD S6" AND "OLD MAN SAID THAT THE TOY CAR WAS
      HIS SON'S"
        WALKP OF "OLD S6" AND "OLD MAN SAID THAT THE TOY CAR WAS HIS
        SON'S"
          WALKP OF "MAN S8" AND "MAN SAID THAT THE TOY CAR WAS HIS
          SON'S"
            WALKP OF "SAID S5" AND "SAID THAT THE TOY CAR WAS HIS SON'S"
              WALKP OF "THAT S1" AND "THAT THE TOY CAR WAS HIS SON'S"
WALKP OF "THE S2" AND "THE TOY CAR WAS HIS SON'S"
  WALKP OF "OLD S2 TOY S3 TOY S7 OLD S6" AND "TOY CAR WAS HIS SON'S"
    WALKP OF "TOY S3 TOY S7 OLD S6" AND "TOY CAR WAS HIS SON'S"
      WALKP OF "WAS S4 WAS S9" AND "CAR WAS HIS SON'S"
        WALKP OF "WAS S9" AND "CAR WAS HIS SON'S"
          WALKP OF "" AND "CAR WAS HIS SON'S"
          WALKP OUTPUTS "FALSE"
        WALKP OUTPUTS "FALSE"
      WALKP OUTPUTS "FALSE"
      WALKP OF "TOY S7 OLD S6" AND "TOY CAR WAS HIS SON'S"
        WALKP OF "CAR S3" AND "CAR WAS HIS SON'S"
          WALKP OF "WAS S4 WAS S9" AND "WAS HIS SON'S"
            WALKP OF "BROKEN * HIS *" AND "HIS SON'S"
              WALKP OF "HIS *" AND "HIS SON'S"
WALKP OF "$*" AND "SON'S"
WALKP OUTPUTS "FALSE"
WALKP OF "" AND "HIS SON'S"
WALKP OUTPUTS "FALSE"
              WALKP OUTPUTS "FALSE"
```

```
            WALKP OUTPUTS "FALSE"
            WALKP OF "WAS S9" AND "WAS HIS SON'S"
             WALKP OF "HIS S10" AND "HIS SON'S"
WALKP OF "SON'S *" AND "SON'S"
  WALKP OF "$*" AND ""
  WALKP OUTPUTS "TRUE"
WALKP OUTPUTS "TRUE"
             WALKP OUTPUTS "TRUE"
            WALKP OUTPUTS "TRUE"
          WALKP OUTPUTS "TRUE"
        WALKP OUTPUTS "TRUE"
      WALKP OUTPUTS "TRUE"
    WALKP OUTPUTS "TRUE"
  WALKP OUTPUTS "TRUE"
WALKP OUTPUTS "TRUE"
             WALKP OUTPUTS "TRUE"
            WALKP OUTPUTS "TRUE"
          WALKP OUTPUTS "TRUE"
        WALKP OUTPUTS "TRUE"
      WALKP OUTPUTS "TRUE"
    WALKP OUTPUTS "TRUE"
  WALKP OUTPUTS "TRUE"
WALKP OUTPUTS "TRUE"
TRUE
```

APPENDIX 4

The Dictionary Program

This programming example demonstrates the use of recursion to build a "complete" definition based on dictionary entries. By complete we mean that every word in a definition for which we have a dictionary is defined when it is used. Note that the program does not check for loops in the dictionary.

```
*LIST ALL

TO DEFINE /S/
10 TEST EMPTYP /S/
20 IF TRUE OUTPUT ""
30 TEST WORDP /S/
40 IF TRUE OUTPUT DEFINE SENTENCE /S/ ""
50 TEST EMPTYP GET-DEF OF FIRST OF /S/
60 IF TRUE OUTPUT SENTENCE FIRST /S/ AND DEFINE BUTFIRST /S/
70 OUTPUT SENTENCES FIRST /S/ AND "(" DEFINE
   GET-DEF FIRST /S/ ")" AND DEFINE OF BUTFIRST /S/
END

TO GET-DEF /NAME/
10 OUTPUT THING OF WORD OF "$" AND /NAME/
END

TO MAKE-DEF /NAME/ /DEF/
10 MAKE WORD "$" /NAME/ /DEF/
END

TO ADDWORD
10 TYPE "TYPE WORD (CR), DEF (CR): "
20 MAKE-DEF REQUEST REQUEST
END
```

```
/$DHOW/ IS "AN ARAB LATEEN-RIGGED VESSEL WITH A LONG
     OVERHANG FORWARD, A HIGH POOP, AND AN OPEN WAIST"
/$POOP/ IS "DECK ABOVE THE UPPER DECK ABAFT THE MIZZEN"
/$MIZZEN/ IS "MIZZENMAST"
/$MIZZENMAST/ IS "AFTERMOST MAST OF A TWO-MASTED VESSEL"
/$ABAFT/ IS "TO THE REAR"
/$LATEEN/ IS "TRIANGULAR SAIL, EXTENDED BY A LONG YARD,
     SLUNG TO THE MAST"
/$YARD/ IS "LONG SPAR"
/$SPAR/ IS "MAST"
/$WAIST/ IS "THAT PART OF A VESSEL'S DECK BETWEEN THE
     QUARTER-DECK AND THE FORECASTLE"
/$FORECASTLE/ IS "FORWARD PART OF VESSEL"
```

```
PRINT DEFINE OF "DHOW"
```

```
DHOW ( AN ARAB LATEEN ( TRIANGULAR SAIL , EXTENDED BY A LONG YARD ( LONG
SPAR ( MAST ) ) , SLUNG TO THE MAST ) -RIGGED VESSEL WITH A LONG
OVERHANG FORWARD , A HIGH POOP ( DECK ABOVE THE UPPER DECK ABAFT ( TO
THE REAR ) THE MIZZEN ( MIZZENMAST ) ) , AND AN OPEN WAIST ( THAT PART
OF A VESSEL'S DECK BETWEEN THE QUARTER-DECK AND THE FORECASTLE ( FORWARD
PART OF VESSEL ) ) )
```

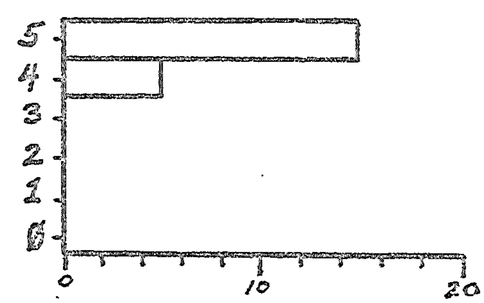
APPENDIX 5

Student Evaluations

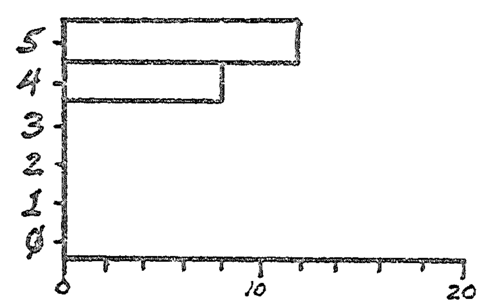
The course evaluation form consists of 25 questions with six possible responses (1 - 5 and "no reply") for each question. On the scale from 1 to 5, strongly no or strongly unfavorable corresponds to 1 and strongly yes or strongly favorable corresponds to 5. In our bar graph summary, 0 corresponds to "no reply".

In addition to these fixed reponse questions, there are six other questions requiring free formed responses. We have chosen the first of these questions and listed all the responses to it.

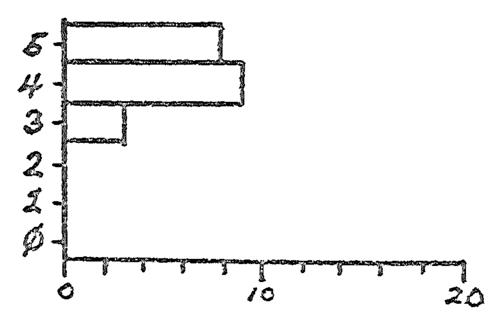
1. Is he willing to answer questions in and out of class?



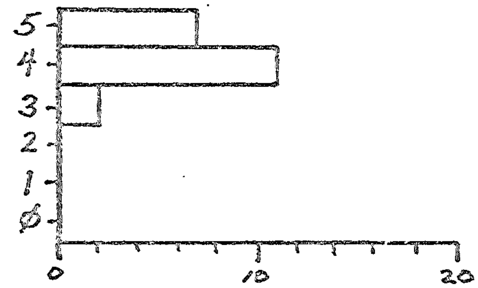
2. Is he helpful and useful when you have difficulties?



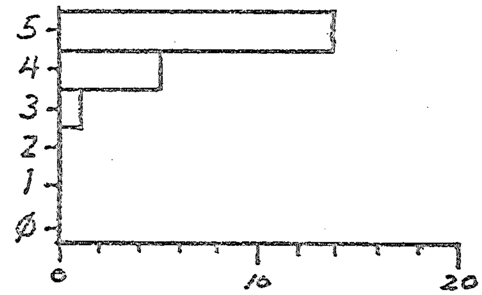
3. Does he present the material in a well-prepared and organized fas. on?



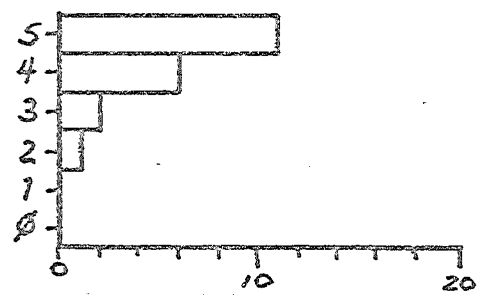
4. Does he provide enough examples and illustrations?



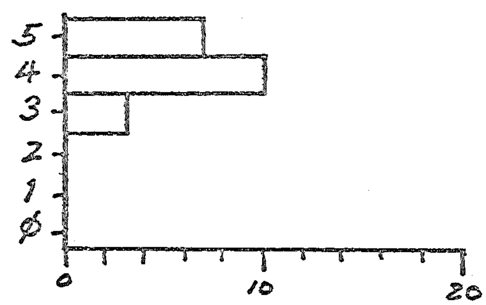
5. Is the course taught in an interesting and stimulating manner?



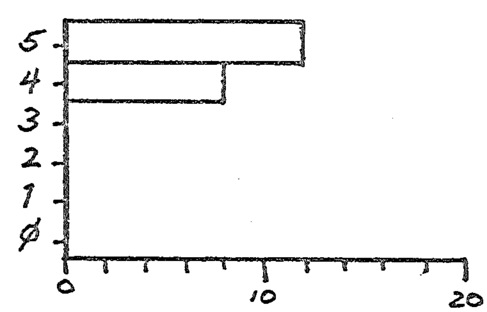
6. Did you feel you were encouraged to think independently?



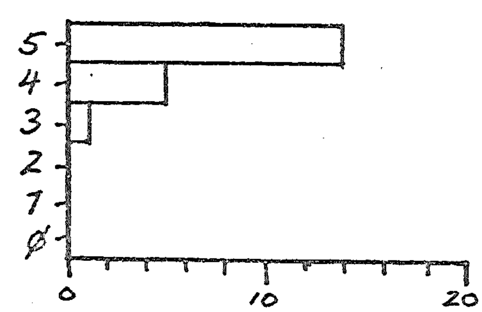
7. Did you feel your instructor's comments and criticisms of you work were valid and helpful?



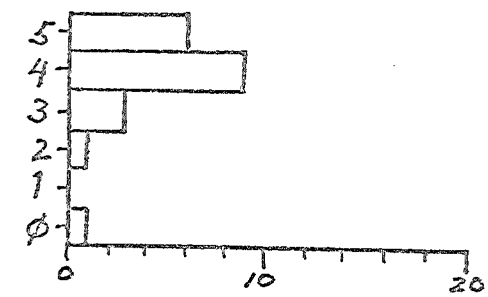
8. How would you rate the instructor, overall?



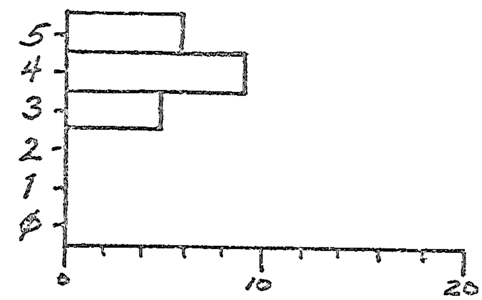
9. Would you like to take another course from this instructor?



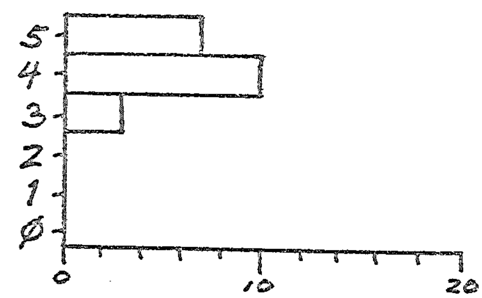
10. Are the objectives of the course clear?



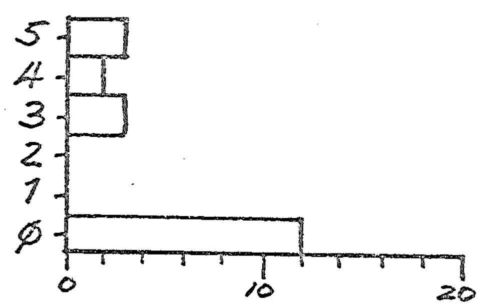
11. Do. you believe that the material presented will be useful to you?



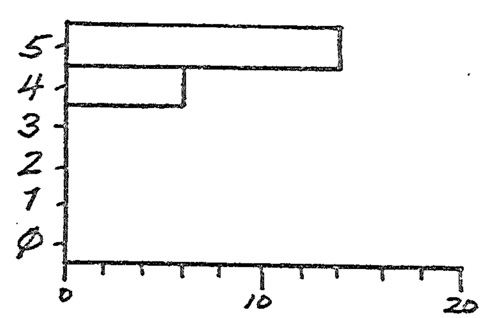
12. Is the amount of work required reasonable?



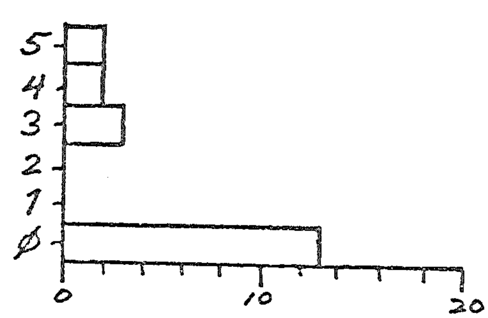
13. Does the textbook contribute to the course?



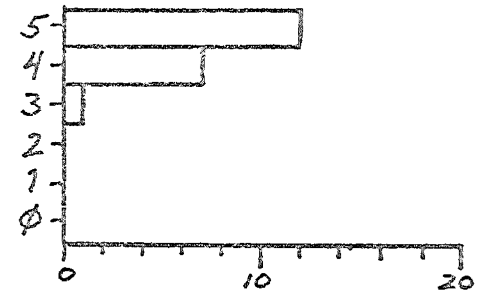
14. Does the homework contribute to the course?



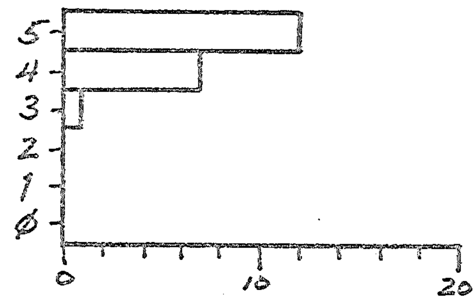
15. Are the exams a fair test of you knowledge of the course?



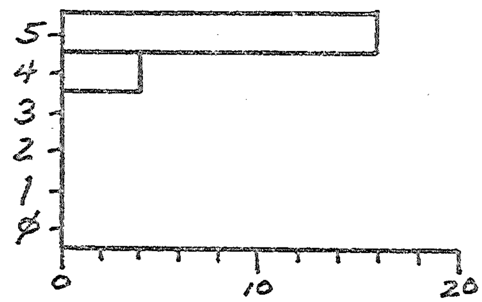
16. Is much of the course material completely new to you?



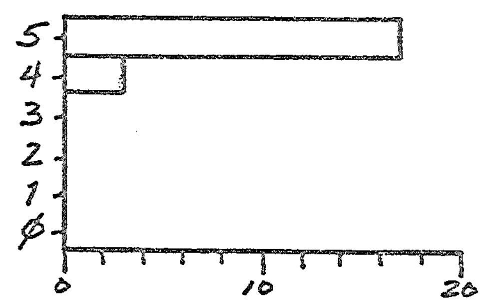
17. Did you put a reasonable amount of time (1/3 - 1/4 of a full-time student load) into this course?



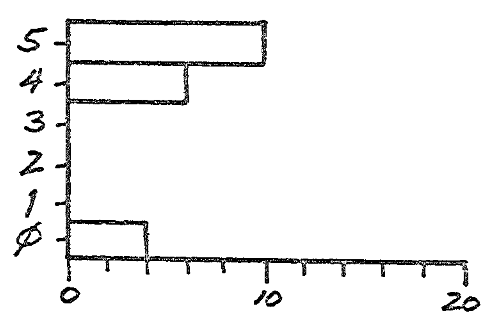
18. How would you rate this course, overall?



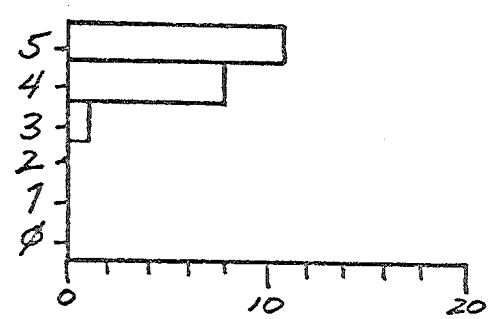
19. Did you enjoy this course?



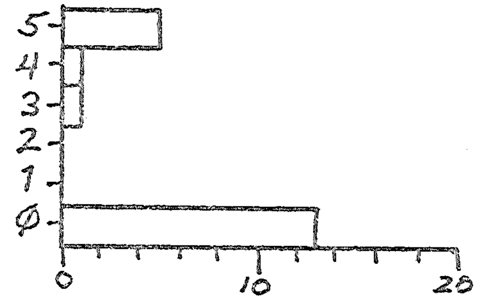
20. How would you rate the discussion leader?



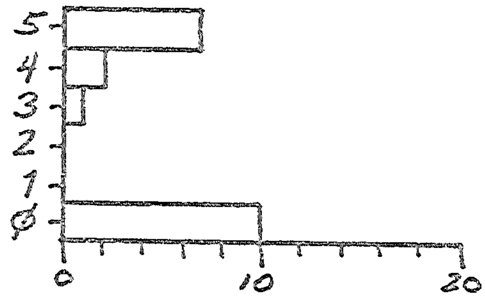
21. Are the discussion sections useful?



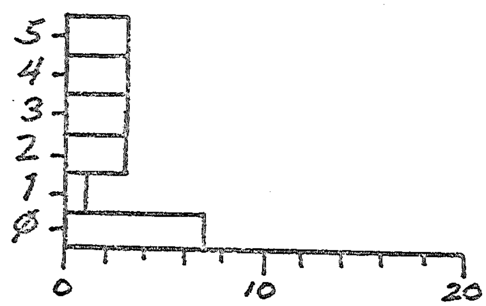
22. How would you rate the laboratory assistant?



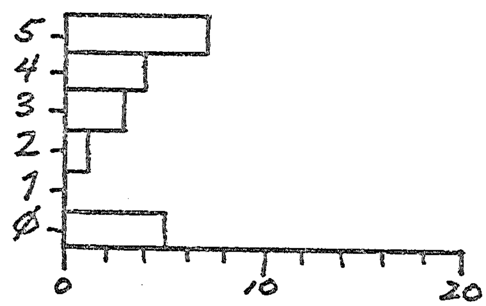
23. Is the laboratory a valuable part of the course?



24. How adequate for this course was you computer science preparation?



25. How adequate for this course was your mathematics preparation?



How does this course compare with other science and math courses you have taken?

Much more imaginative than other classes.

More interesting, more time-consuming, more enjoyable.

It was the most stimulating course I have taken in both math and science.

Extremely interesting and seemingly relevent to my field of interest.

More interesting; involving than most.

It's a fun course and if I had time I would like to pursue it.

I like it much better than any science course and just better than math.

More enjoyable but more time-consuming.

It was more fun using the computer than just texts.

A hell of a lot better.

Far more interesting and stimulating.

Very good.

Better.

Not applicable, since this course is essentially non-numeric.

I haven't taken any before.

More interesting.

Doesn't.

Favorably.

(two no-responses)