ON THE GENERATION OF PROBLEMS

S.Ramani* and A.Newell

November, 1973

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-73-C-0074) and is monitored by the Air Force Office of Scientific Research. This document has been approved for public release and sale; its distribution is unlimited.

*On leave of absence from:
Computer Group,
Tata Institute of Fundamental Research,
Homi Bhabha Road, Bombay 400 005, India.

ABSTRACT

The design of programs that generate questions and exercises for students is discussed. The task of direct interest is the generation of programming exercises. This is dealt with in some detail and sets of such exercises produced by programmed generators are presented.

Several design strategies are investigated, starting with the use of grammar-like generative mechanisms. The design of exercises is then viewed as an assembly task, putting together compatible elements to create an acceptable problem-structure, accessing knowledge of the task area encoded as a semantic net. A sentence synthesizer which generates coherent English sentences to describe the assembled problems is then described. Attention is also paid to the mechanisms necessary for controlling such an assembly process. Directing generators towards problems involving specified concepts and making use of information available about the student's familiarity with a hierarchy of concepts are two issues discussed in this area.

Other strategies discussed are: generalizing 'good' problems to produce useful variants; use of reasoning about actions in a task area to recognize good problem situations; and the design of surface detail to create problems having a given abstract structure.

1. INTRODUCTION

The notion of a generative CAI system has attracted some attention(1). An important issue in this area is the structure of generators for problems and exercises, and it is this issue that we concern ourselves with in the following pages. Design of generators for interesting classes of problems appears within reach and is, of course, advantageous. Among the frequently cited advantages of generation, as against selection from a long list, are the controllability and adaptability that generation provides. These features provide for the creation of exercises having specified content and posing a specified level of difficulty to suit the needs of an individual student at a specific stage in learning. And, of course, generative schemes provide for potentially larger sets of examples to choose from, compared to any finite list.

Developing a generative scheme forces analysis of problems and the task area to a level of detail that is usually avoided in making a collection of exercises. In fact, such analysis might possibly lead to the the definition of knowledge domains operationally. For instance, in classifying exercises on the basis of complexity, one could use a response measure such as 'average time taken for solving', but a generative scheme motivates attempts to understand the difficulty posed by an exercise in terms of its structure.

Another aspect of generative schemes is the level at which information is available to the system that handles the exercises. While a collection of exercises could be handled by a more or less sophisticated page-turner, generative schemes require a fine-grain

---

(1) Uhr, 1969; Uttal et al, 1970; Wexler, 1970 and Koffman, 1972.

representation of some knowledge of the task area in order to work. We use a semantic net in which is encoded a variety of information on the task area. In the case of programming exercises, the semantic net contains information on types of manipulable objects such as numbers, symbols, sets and sequences, their attributes, sets of examples for each type of object or generative procedures for creating examples, data structures in terms of which each type of object may be realized, manipulations that may be performed on each type of object, etc.. Dealing with information in such detail is essential if we are ever to develop systems that understand the task area - to generate answers to exercises or to generate code to recognize a variety of acceptable answers along with the generation of the exercise, to answer questions about a specific exercise or about the task area, and to attempt to carry out generated exercises to find out their characteristics.

The recognition that generative techniques can make good use of a fine-grain representation of semantic information relevant to a task area hardly concludes the analysis of problem generation. In fact, this recognition is more appropriate as the first step than as the conclusion in any treatment of the subject. Development of a problem generator in a given area should start with the identification and analysis of viable sets of problems in that area, and proceed to specify suitable structures for programs that will generate these sets. Specific strategies for accessing representations of knowledge in the process of problem generation should be spelt out.

Undoubtedly, an understanding system with a well integrated store of knowledge of a subject area is the proper ultimate form for a problem generator. Such a system would be able to answer

questions about the area, design good problems in the area, discuss them and even solve them. The work reported here does not yet aim at such a system. The approach taken here is exploratory and quite empirical. The presentation is in the form of a discussion of a sequence of simple strategies for using different aspects of information about a subject area for generating non-trivial problems. Most of the strategies discussed are illustrated with examples of problems generated by programs based on them. All the examples presented are in the area of programming exercises, though the discussions are hopefully relevant to other kinds of problems as well.

2. ENUMERATIVE SCHEMES PATTERNED AFTER CONTEXT-FREE GRAMMARS

The simplest generative scheme is an extension of the slot-and-filler scheme, as it is called in linguistics. In this scheme, one has a set of problem frames incorporating variables. There is a specification of the admissible values that these variables may assume, either in the form of numerical limits or in the form of a set of admissible values for each variable.

There are two directions of development that enrich this scheme. Uhr[1] reports one of them, which is to implement computationally defined relations between the variables. By choosing certain independent variables randomly within the permitted ranges or from sets of admissible values, the others could be computed from them, thus ensuring compatibility. The original scheme, of course, does not provide mechanisms for meeting such compatibility requirements and forces the variables to be chosen independently of each other.

The second direction of development involves the use of a grammar-like scheme. Unlike slot-and-filler schemes, the grammar-like schemes are not limited to finite-state languages. They have been used in the context of problem generation for synthesizing sentences having phrase structure. Koffman (1972) has reported the use of probabilistic grammars for the generation of word problems. Simple programs that use grammar-like schemes to generate questions in artificial intelligence (developed by Newell and Robertson) and in cognitive psychology (developed by Waterman) have been in use at the Carnegie-Mellon University since 1971.

Fig.1 shows the enumerative scheme underlying the program which deals with artificial intelligence. The questions produced by this program are always grammatically and semantically correct, and most of the questions are meaningful. Not shown in the scheme is a probability assignment convention that allows certain choices to be more or less frequent than other choices in the generative procedure, increasing the percentage of desirable questions in the output. Figures 2 and 3 show some of the questions produced by the artificial intelligence program and the cognitive psychology program respectively.

Enumerative schemes such as this derive their power from a basic device - classification of a set of phrases into different subsets of syntactically and semantically similar items. Question frames are written incorporating variables which are substitutable by members of appropriate subsets of phrases. Further, the phrases are themselves generatable by this process by expansions of embedded variables into suitable sub-phrases, the variables in figure 1 are indicated by the terminal angle brackets '<' and '>'. sets of phrases that are substitutable for the variables are defined in the lower half of the figure.

```
<QUESTIONS>:(  ("WHAT IS " <ENTITY> "?")
   ("WHAT METHOD DOES " <PROGRAM> " USE?")
   ("WHAT TASKS DOES " <PROGRAM> " WORK ON?")
   ("HAS " <TASK> " BEEN ACCOMPLISHED BY A PROGRAM?")
   ("WHERE IS AI RESEARCH ON " <TASK> " GOING ON?")
   ("WHAT IS THE CURRENT ACHIEVEMENT IN " <TASK> "?")
   ("WHAT HAVE BEEN THE CONTRIBUTIONS OF AI TO " <AREA> "?")
   ("WHAT HAVE BEEN THE CONTRIBUTIONS OF " <AREA> " TO AI?")
   ("NAME A TASK THAT " <PROGRAM> " DOES NOT DO, BUT MIGHT DO WITH SOME
    MODIFICATION.")
   ("WHY IS " <ENTITY> " SO CALLED?")
   ("WHAT PROGRAMMING LANGUAGE WAS USED FOR " <PROGRAM> "?")
   ("HAS " <LANGUAGE> " BEEN USED TO ACCOMPLISH " <TASK> "?")  )
<ENTITY>: ((<PROGRAM>) (<HEURISTIC>) (<PHENOMENA>) (<CONCEPT>) ...)
<CONCEPT>:(("A METHOD") ("A PROBLEM") ("A HEURISTIC") ...)
<PROGRAM>: ((<RECOGNITION-SYSTEMS>) (<GAME-PLAYERS>) (<THEOREM-PROVERS>)
   (<QUESTION-ANSWERERS>) (<GENERAL-PROBLEM-SOLVERS>) ...
   (<UNDERSTANDING-SYSTEMS>) (<DESIGN-SYSTEMS>)
  (<CONSTRAINT-SATISFIERS>))
<CHESS-PROGRAMS>: (("NEWELL-SHAW-SIMON CHESS PROGRAM")
   ("BERNSTEIN'S CHESS PROGRAM") ("CMU TECHNOLOGY CHESS PROGRAM")  ...)
<TASK>: ((<GAMES>) (<MANAGEMENT-SCIENCE>) (<CONCEPT-FORMATION>) ...)
<PROGRAM-FEATURE>: ((<HEURISTIC>) ("ALPHA-BETA") ("RECURSION")  ...)
```

Fig.1 Part of an Enumerative Scheme Patterned after Context Free Grammars

Q: WHAT AI PROGRAMS HAVE BEEN WRITTEN IN MLISP?
Q: WHY IS FEATURE EXTRACTION SO CALLED?
Q: HAS THERE BEEN ANY CONTRIBUTION OF AUTOMATA THEORY TO AI?
Q: WHAT WAS THE FIRST AI EFFORT ON CHESS?
Q: HAS SAIL BEEN USED TO ACCOMPLISH CHECKERS?
Q: WHAT HAVE BEEN THE CONTRIBUTIONS OF AI TO PHYSICS?
Q: HAS APL BEEN USED TO ACCOMPLISH ASSEMBLY LINE BALANCING?
Q: WHAT IS THE CURRENT ACHIEVEMENT IN CHESS?
Q: WHAT IS THE CURRENT ACHIEVEMENT IN PLANT LOCATION SELECTION?
Q: WHAT HAVE BEEN THE CONTRIBUTIONS OF INDUSTRIAL ADMINISTRATION TO AI?
Q: WHAT IS A HEURISTIC?
Q: HAS SAIL BEEN USED TO ACCOMPLISH RELATIONAL CONCEPT FORMATION?
Q: HAS ASSEMBLY LINE BALANCING BEEN ACCOMPLISHED BY A PROGRAM?
Q: HAS COBOL BEEN USED TO ACCOMPLISH CHESS?
Q: HAS THERE BEEN ANY CONTRIBUTION OF PHYSICS TO AI?
Q: WHAT IS HORIZON PHENOMENA?
Q: WHAT PROGRAMMING LANGUAGE WAS USED FOR REF-ARF?
Q: WHAT WAS THE FIRST AI EFFORT ON RELATIONAL CONCEPT FORMATION?
Q: WHAT HAVE BEEN THE CONTRIBUTIONS OF AI TO YOUR EDUCATION?
Q: WHAT METHOD DOES GPS USE?
Q: WHAT TASKS DOES FREEMAN'S OSD WORK ON?
Q: NAME SOME HEURISTICS USED IN SRI ROBOT.
Q: WHO DEVELOPED EXPONENTIAL GROWTH?

Fig.2 Output of the Question Generator for Artificial Intelligence

WHAT CONTRIBUTION HAS LOEHLIN MADE TO THE STUDY OF COGNITIVE PSYCHOLOGY?
WHAT IS CYBERNETICS?
WHAT ARE THE SIMILARITIES BETWEEN TREE STRUCTURES AND NETWORKS?
WHAT ARE THE SIMILARITIES BETWEEN ALGORITHM AND HEURISTIC?
WHAT CONTRIBUTION HAS ABELSON MADE TO THE STUDY OF COGNITIVE PSYCHOLOGY?
WHAT IS CONTENT ADDRESSABLE MEMORY?
WHAT IS AN EXAMPLE OF A GPS GOAL?
WHAT IS A MAGIC SQUARE?
WHAT ARE THE DIFFERENCES BETWEEN SYNTAX AND SEMANTICS?
DEFINE MARKOV PROCESS.
DESCRIBE A FEW METHODS THAT HAVE BEEN USED IN ARTIFICIAL INTELLIGENCE.
WHAT IS A BIT?
WHAT ARE THE DIFFERENCES BETWEEN TEMPLATE MATCHING AND FEATURE
EXTRACTION?
WHAT IS ALDOUS?
WHAT CONTRIBUTION HAS CHOMSKY MADE TO THE STUDY OF COGNITIVE PSYCHOLOGY?
WHAT IS PANDEMONIUM?
WHAT CONTRIBUTION HAS WIENER MADE TO THE STUDY OF COGNITIVE PSYCHOLOGY?
WHAT ARE THE SIMILARITIES BETWEEN COMPUTER AND HUMANS - BY ANALOGY?
DEFINE DISCRIMINATION NET.
WHAT ARE THE SIMILARITIES BETWEEN COMPILER AND INTERPRETER?
WHAT ARE THE SIMILARITIES BETWEEN TEMPLATE MATCHING AND FEATURE
EXTRACTION?

Fig.3 Output of the Question Generator for Cognitive Psychology

Word categorization on a semantic as well as syntactic basis enables these programs to produce a surprisingly high percentage of semantically meaningful sentences compared to programs that are based on generative syntax alone. Restricting oneself to the generative power of context-free rewrite rules has its own advantages. The control structure of the problem generator becomes simple and straightforward, making it easy to construct trouble-free programs. The essence of these schemes is the simple hierarchical top-down control structure that relates a set of choices. For each choice that is made, the grammar simply forces the other (lower level) choices that one is committed to make as a result. Each act of choice transfers control to that part of the grammar where the available alternatives for lower level choices are contained. On the other hand, restriction to such a simple program structure does result in severe limitations.

The basic limitation is the absence of a structured representation of the task environment of the kind found in practically every system that performs a cognitively significant task (Carbonell, 1970, and Wexler, 1970, describe instructional systems incorporating structured representations of knowledge). Generally in such systems associations are used to impose an object-attribute-value scheme on the task environment and to implement functional and relational mappings. Computational mappings of a functional or relational nature are performed by programmed routines. This is in contrast to the implicit representation of knowledge in grammar-like systems which provide only two devices to ensure coherence in the output: co-occurrence of symbols in rewrite rules which ensures co-occurrence of complementary parts of the problem in the output; and set membership which enables semantically and syntactically equivalent items to be distributed in the output in an identical manner.

Even the use of variables for holding data and control information is not possible. This means for instance, that one cannot use a variable to hold on to the name of the data-structure that one has chosen to refer to in a problem being generated, so that it can be used repeatedly. Instead one requires one set of rules for generating, say, sorting exercises on lists and a similar but different set of rules for sorting exercises on arrays, unless of course there is a single reference to the nature of the data-structure.

Finally, syntactical and semantic similarities of phrases are clumped together while classifying these phrases into subsets. The fact that a particular set of phrases refers to investigators in a particular field has to be remembered along with the syntactic information that all entries in this subset are singular in number, and that in addition they are all only last names. Achievements of investigators expressed in the past tense have to be segregated from their activities expressed in other tenses. Flexibility in reference can be obtained only by entering grammatical variants of the same phrase in different subsets.

Summing up, it appears that the initial advantages of a grammar-like scheme are outweighed by the restrictions it imposes on the development of a problem generator. It becomes necessary to recognize and separate two concepts implicit in such schemes, one being more general than the other. The first is the concept of generative syntax with its power to generate well-formed sentences in natural language or in mathematical notation. The second is the concept of an elegant and economical principle of program organization - that of a top-down enumerative scheme for generating a recursively defined set of structures. It is obviously desirable to employ the latter concept whenever

possible, without committing oneself to the formalization of all the machinery required for problem generation in the form of generative syntax. Steps in this direction are particularly important if we visualize CAI systems having access to detailed representations of knowledge of task areas. Whatever form such knowledge is going to be in - semantic nets, simulation models or understanding systems - it certainly is not going to be all syntax.

3. GENERATION OF A PROBLEM AS THE ASSEMBLY OF A STRUCTURE

3.1 Assembly of Problems

We can view the creation of exercises as a task in design. Given the function of a desired object, how does one decide its structure? How does one recognize the system of design decisions necessary in a given case? How does one encode decisions made in the earlier stages of the process so that their implications for later decisions are readily computable? In this section we will deal with the generation of a basic kind of programming problem in these terms. These problems exercise the student's skills in handling arrays and in organizing data in the form of sets and sequences of numbers and symbols. A typical hand generated problem of this type (1) is:

YOU ARE GIVEN THE ARRAYS IARRAY[1:100] AND MARRAY[1:100]. FIND ALL SETS OF NUMBERS IN IARRAY, NO MATTER WHAT THEIR ORIGINAL POSITION, WHICH CAN BE PUT IN COUNTING ORDER (I.E. EVERY ELEMENT IN A SET, EXCEPTING THE FIRST AND THE LAST, SHOULD HAVE ITS SUCCESSOR AND PREDECESSOR IN THE SAME SET). MARK THE FIRST SET BY PUTTING 1'S IN THE CORRESPONDING POSITIONS IN MARRAY, THE SECOND SET BY PUTTING 2'S IN THE CORRESPONDING POSITIONS, AND SO ON. PUT ZEROES IN THE POSITIONS OF ELEMENTS WHICH DO NOT BELONG TO ANY SETS.

Consider the task of a program that has to generate problems in this form. A series of decisions have to be made, some of them independent and some contingent on the others. Analysis of a number of such problems shows that it is possible to represent in a compact manner the basis for making these decisions for

---

(1) Courtesy of Ruven Brooks

producing reasonably large subsets of the studied problems. It is possible in these cases to find a common form in the structures of the individual members of the problem set. For example, the problems(1) of Fig.4 have a common structure outlined in Fig.5. The problems in Fig.4 as well as those in Fig.6 were generated by programs having the simple control structure discussed in the last paragraph of Section 2. Use of this top-down generative structure is based on the recognition that the problem to be constructed has a basically tree-like structure. The highest level of the program therefore deals with the linking up of the major sub-structures of the tree which are assumed to exist. Lower level routines are then defined for creating these smaller structures. The parcelling out of the assembly task can be spread over several levels, the routines at the terminal nodes selecting and creating basic elements of the problem, while the higher level routines assemble parts created at lower levels into bigger structures.

---

(1)Generated by a program to be described shortly.

DEFINE AN ARRAY L(200) AND READ NUMBERS INTO THIS ARRAY FROM THE FILE FOR01.GEN. CHOOSE A SUITABLE FORMAT FOR THE READ STATEMENT BY FIRST TYPING OUT FOR01.GEN AND EXAMINING IT. EXAMINE THE ELEMENTS OF L AND IDENTIFY THE MAXIMAL SET EACH ELEMENT OF WHICH IS A PRIME. SORT THE ELEMENTS OF THE SELECTED SET IN NUMERICAL ORDER. PLACE THIS ORDERED SEQUENCE OF NUMBERS AT THE BEGINNING OF L, IN DECREASING NUMERICAL ORDER. PLACE A ZERO IN ALL THE LOCATIONS OF L WHICH DO NOT CONTAIN ONE OF THE FINALLY SELECTED NUMBERS. PRINT OUT THE FINALLY SELECTED NUMBERS.

DEFINE AN ARRAY L(30,30) AND READ NUMBERS INTO THIS ARRAY FROM THE FILE FOR01.GEN. CHOOSE A SUITABLE FORMAT FOR THE READ STATEMENT BY FIRST TYPING OUT FOR01.GEN AND EXAMINING IT. EXAMINE ALL THE ROWS OF L AND IN EACH ROW, IDENTIFY ALL SEQUENCES OF 7 OR MORE NUMBERS HAVING ELEMENTS EACH ONE OF WHICH IS A NUMBER GREATER THAN 4. POOL THE ELEMENTS OF ALL THESE SEQUENCES AND SORT THEM IN NUMERICAL ORDER. PLACE THE 30 LARGEST ELEMENTS ALONG THE LOWEST ROW IN NUMERICALLY INCREASING ORDER FROM LEFT TO RIGHT. PLACE A ZERO IN ALL THE LOCATIONS OF L WHICH DO NOT CONTAIN ONE OF THE FINALLY SELECTED NUMBERS. PRINT OUT THE FINALLY SELECTED NUMBERS.

DEFINE AN ARRAY L(300) AND ASSUME THAT THIS ARRAY IS FILLED WITH NUMBERS IN THE RANGE FROM -1000 TO 1000. EXAMINE THE ELEMENTS OF L AND IDENTIFY ALL SEQUENCES OF 3 OR MORE NUMBERS HAVING ELEMENTS THAT ARE ALTERNATELY A CUBE AND A NON-CUBE. CONSIDER THE SEQUENCES IN THE ORDER OF INCREASING FIRST TERMS. CHOOSE THE FIRST 6 SEQUENCES IN THIS ORDER, IF THERE ARE MORE THAN 6 SEQUENCES. OTHERWISE, CHOOSE ALL THE SEQUENCES. POOL THE ELEMENTS OF ALL THE CHOSEN SEQUENCES AND SORT THEM IN NUMERICAL

ORDER. PLACE THIS ORDERED SEQUENCE OF NUMBERS AT THE BEGINNING OF L, IN INCREASING NUMERICAL ORDER. PRINT OUT L.

DEFINE AN ARRAY L(10,10) AND READ NUMBERS INTO THIS ARRAY FROM THE FILE FOR01.GEN. CHOOSE A SUITABLE FORMAT FOR THE READ STATEMENT BY FIRST TYPING OUT FOR01.GEN AND EXAMINING IT. EXAMINE ALL THE ROWS OF L AND IN EACH ROW, IDENTIFY ALL SEQUENCES OF 3 OR MORE NUMBERS HAVING ELEMENTS IN THE FOLLOWING ORDER - A NUMBER DIVISIBLE BY 7, AN ODD NUMBER, A NEGATIVE NUMBER. SUCH A SEQUENCE MAY START WITH ANY ONE OF THE SPECIFIED TYPES OF ELEMENTS. POOL THE ELEMENTS OF ALL THESE SEQUENCES AND SORT THEM IN NUMERICAL ORDER. PLACE THE 10 LARGEST ELEMENTS ALONG THE FIRST COLUMN IN NUMERICALLY DECREASING ORDER FROM TOP TO BOTTOM. PRINT OUT THIS COLUMN.

DEFINE AN ARRAY L(600) AND FILL IT UP WITH RANDOM NUMBERS IN THE RANGE FROM -1000000 TO 1000000. EXAMINE THE ELEMENTS OF L AND IDENTIFY ALL SEQUENCES OF 7 OR MORE NUMBERS HAVING ELEMENTS EACH ONE OF WHICH IS A MULTIPLE OF ITS PREDECESSOR. POOL THE ELEMENTS OF ALL THESE SEQUENCES AND SORT THEM IN NUMERICAL ORDER. PLACE THIS ORDERED SEQUENCE OF NUMBERS AT THE BEGINNING OF L, IN DECREASING NUMERICAL ORDER. PLACE A ZERO IN ALL THE LOCATIONS OF L WHICH DO NOT CONTAIN ONE OF THE FINALLY SELECTED NUMBERS. PRINT OUT L.

Fig.4 Examples of 'Array Ordering Problems'

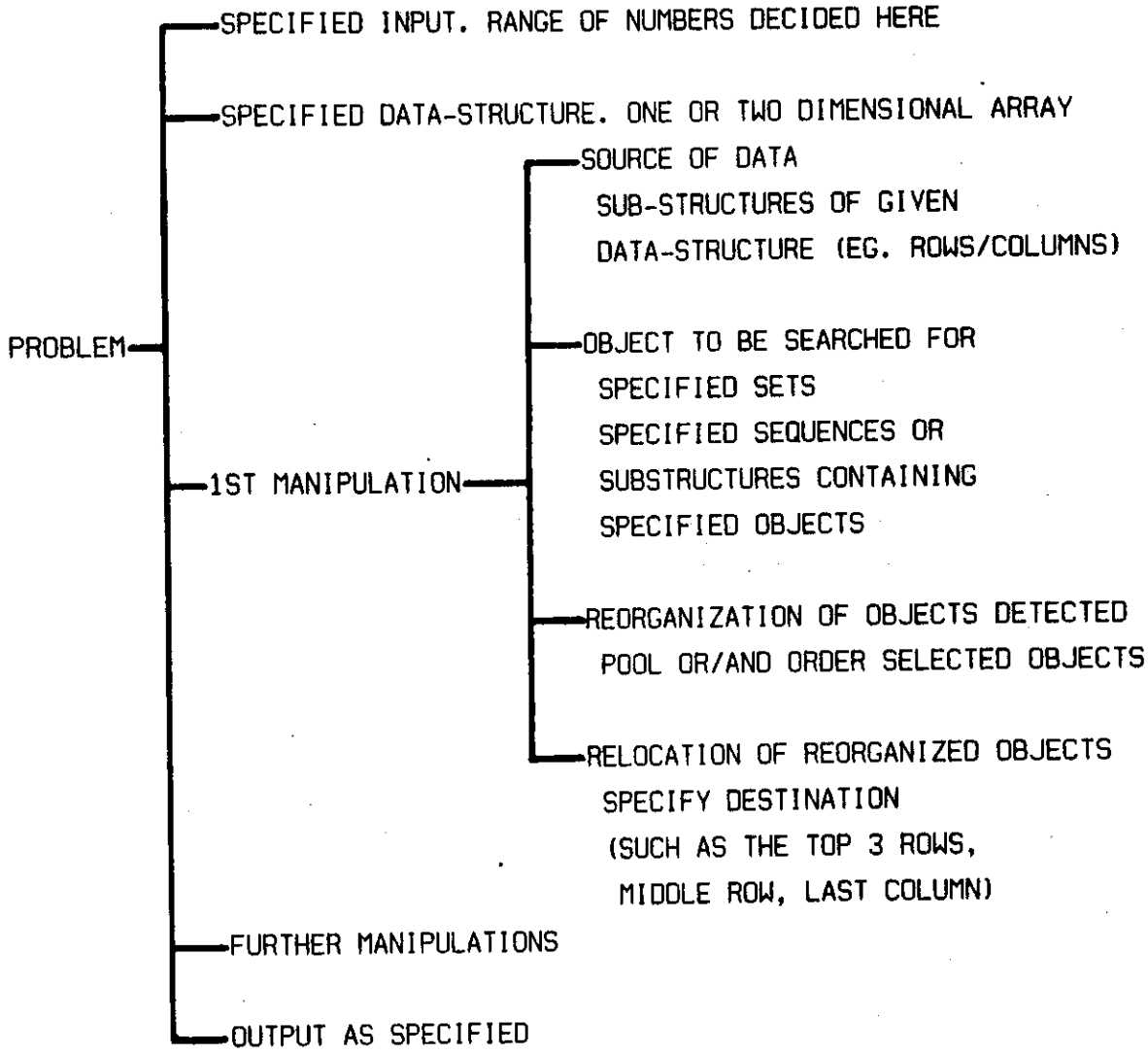

Fig.5 The Structure of a Class of Array Manipulation Problems

CONSIDER THE LIST L19 AND THE UNNAMED LIST ON THE STACK. SELECT THE SET OF ALL EVEN NUMBERS WHICH ARE PRESENT IN L19 BUT NOT IN THE UNNAMED LIST. FIND THE SUM OF ALL THE SELECTED NUMBERS AFTER ELIMINATING MULTIPLE OCCURRENCES.

YOU ARE GIVEN THE LIST-STRUCTURES L13 L14 L15 AND L16 AND 4 UNNAMED LIST-STRUCTURES ON THE STACK. THEY MAY CONTAIN NUMBERS AT ANY AND ALL LEVELS. SELECT THE SET OF ALL PRIME NUMBERS WHICH ARE PRESENT IN L16 , IN L15 , IN L14 , AS WELL AS IN L13 BUT NOT IN EACH UNNAMED LIST-STRUCTURE. FIND THE SUM OF ALL THE SELECTED NUMBERS AFTER ELIMINATING MULTIPLE OCCURRENCES.

CONSIDER THE 3 UNNAMED LIST-STRUCTURES ON THE STACK AND THE LIST-STRUCTURES L18 L19 AND L20. THEY MAY CONTAIN NUMBERS AT ANY AND ALL LEVELS. SELECT THE SET OF ALL NEGATIVE NUMBERS WHICH ARE PRESENT IN L20 BUT NOT IN L19. MERGE THESE WITH THE NUMBERS WHICH ARE GREATER THAN 100 FROM THE TOPMOST LIST-STRUCTURE ON THE STACK. PLACE ALL THE SELECTED NUMBERS ON L18 AFTER EMPTYING IT FIRST. SORT THE CONTENTS OF L18 IN DESCENDING NUMERICAL ORDER. FIND OUT ALL SEQUENCES OF PRIME NUMBERS IN IT AND PLACE THEM ON THE 2 OTHER UNNAMED LIST-STRUCTURES.

Fig.6 A Set of Simple List-Processing Problems

Some of the terminal routines make random choices, while others access the structures already assembled and choose compatible elements to be used in the sub-structures being constructed. Access to a representation of knowledge of the task area having the form described on page 4 also occurs at this level as a part of the relevance and compatibility computation.

It is useful at this stage to evaluate the utility of the selective mechanisms discussed above. Very rough estimates can be made of the total number of structures that are in some sense possible, and the fraction of these structures that are meaningful. The number of branchings that occur in the course of generating a problem of the type shown in Fig.4 is of the order of 15, corresponding to the branchings of Figure 5.

While the choices that create structural differences between problems usually involve only two or three alternatives, e.g. the choice between one dimensional arrays and two dimensional arrays, numerical choices involve large numbers of alternatives. Most of the numerical choices are equivalent, but selecting a number in the wrong range could create absurd problems, e.g., comparing numbers in the range from 0 to 100 with the threshold of 2500. Estimating that there are 2 or 3 significant alternatives at most branches, the total number of choice sequences that are available over 15 steps is in the range from 10↑4 to 10↑7 . Typically, five or six of these decisions are contingent upon others, so that the amount of selection exercised by the program in producing a coherent problem (out of what would be mostly nonsense problems if free choices were made at each point) is roughly 1 in 500.

3.2 Assembly of Sentences

What is the nature of the structure assembled by a problem generator in relation to a comprehendable external description of the problem? One possibility is that the structures assembled by the generator be directly the deep-structures of sentences which would constitute a readable description of the problem. In this case all the knowledge required to translate these structures into readable descriptions can be localized in a 'sentence synthesizer'. Such a synthesizer would remove the burden of syntactic considerations from the problem-generator. The input requirements of this synthesizer would define a range of possible formats for the data-structures to be created by the generator. We have implemented such a sentence synthesizer (see Simmons and Slocum, 1972 for a detailed treatment of sentence synthesis). The problems in Figures 6, 8, 12 and 13 were all generated by programs which first created well-defined deep-structures for their sentences. These were then rendered into readable form by the sentence synthesizer. A brief discussion of the nature of the sentence structures used is given here. Some information on programming considerations and on the sentence synthesizer may be found in Appendices A and B.

The deep-structures are in the form of labelled directed graph-structures. The nodes are unnamed lists representing specific occurences (or 'tokens') of concepts symbolized by word-roots, numbers or symbols. The edges are associations implemented by using association lists and a hash-coding scheme. While the list itself represents the token, its content, generally a word-root (referring to a specific word-sense, in the case of ambiguous words), represents the 'type'. There may be several lists with the same content to represent different tokens of the

same type. Structures are formed by creating labelled (and directed) associations between nodes to represent deep-case relations between the entities referred to in a sentence. Some of the functions performed by these relations are illustrated by the examples in Fig.7. the two appendices describe briefly the construction and interpretation of such sentence-structures.

While token-to-token associations create sentence structures, other associations involving types embody the sentence synthesizer's knowledge of the language. For example, such associations link words to their syntactic categories and to their exceptional plural forms when these exist. They also link syntactic categories with appropriate synthesis routines which recognize basic sub-structures of sentences and have the capabilities to generate appropriate word-groups for them. While token-to-token associations are dynamic in the sense of being created and destroyed as sentences are generated and printed out, the syntactic and semantic associations are permanent. However, the same association mechanism in the programming facility provides the three types of associations discussed so far: associations embodying information on the task area as described on page 4, the token-to-token links which are used to create sentence structures and the syntactic and semantic associations embodying the sentence synthesizer's knowledge of English. As may be expected, there is some overlap between what should belong to the first type and what should belong to the third type.

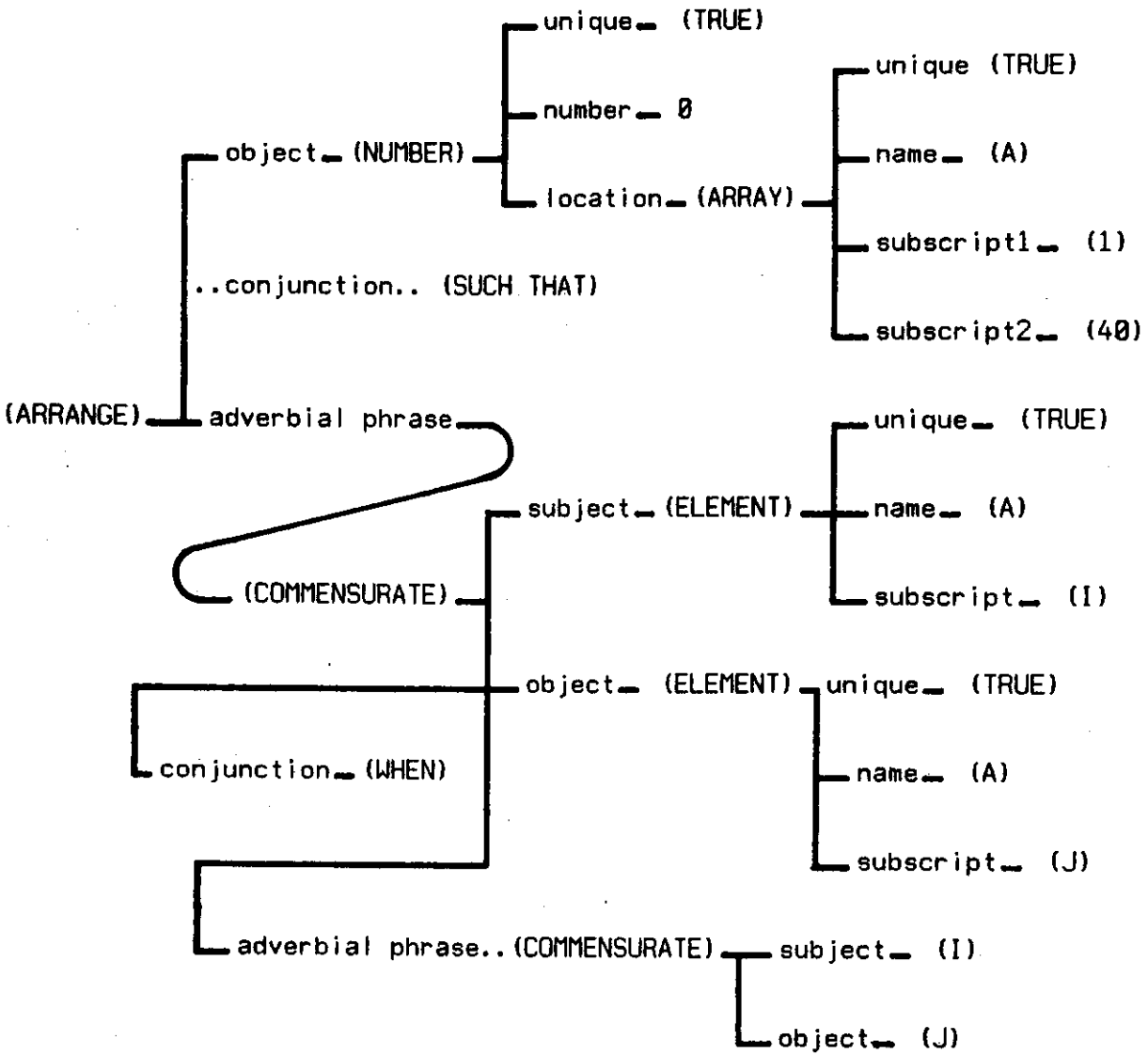

Fig.7 Deep Structure of a One-Sentence Problem

3.3 Control of the Generative Process

In discussing programs having the control structure described in Section 3.1 it is useful to talk in terms of a search of the 'design space' which is a special case of the 'action space', using the search representation for problem-solving (Newell and Simon, 1972). The series of design decisions that leads to the assembly of a suitable object is a special case of the 'solution-action-sequence'.

Complementing this description is the set representation for problem-generation which deals with the target space of all problems that can be produced by the generator. Starting at the highest level, the program selects progressively smaller subspaces, till it identifies a single point in it which represents the problem generated at this attempt. The design of the program has to start with the visualization of a rich enough space. The control structure has to ensure that all the visualized space is accessible by a suitable sequence of choices.

An important question is concerned with the control of this access path. How can one influence a series of choices selecting one possible access path out of a large number all of which lead to meaningful problems?

A powerful and yet simple control mechanism that provides a certain degree of direction in the generative process depends upon the use of a quasi-formal language to specify desired sets of problems. For example, problems of the type in Fig.8 are easily produced by appropriate interpretation of brief high-level descriptions, e.g.:

(CREATE A LIST)
(SCAN A CIRCULAR LIST-STRUCTURE)
(SUBSTITUTE COMPONENTS IN A SYMMETRIC LIST-STRUCTURE)
(FIND PROPERTY OF GIVEN LIST-STRUCTURE)
(TEST IF A DATA-STRUCTURE IS RECURSIVE)
(DELETE COMPONENTS FROM A SYMMETRIC LIST-STRUCTURE)

A sentence in such a language predetermines a specific set of choices to be made by the generator, leaving other choices to be made freely. Naturally, the structure of the sentence interpreting mechanism will determine the effectiveness of any such scheme. The interpreting mechanism has to analyse the sentences and determine which steps of the generative process are being forced, and in which direction.

An appealing form for such an interpreter is a semantic parser which finds out the deep-case relationships between the entities referred to by a sentence, sharing syntactic and semantic information used by the sentence synthesizer.

```
CREATE A LIST CONTAINING THE FIRST 42  PRIME NUMBERS
 >
COUNT OCCURRENCES OF SEQUENCES OF NEGATIVE NUMBERS IN THE CIRCULAR
LIST-STRUCTURE GIVEN
 >
SUBSTITUTE ALL OCCURRENCES OF THE COMPONENT-LIST C2 IN THE SYMMETRIC
LIST-STRUCTURE GIVEN BY THE TERMINAL-NODE GIVEN
 >
WRITE A PROGRAM TO CONCATENATE THE LISTS LIST-L1 AND LIST-L2
 >
SUBSTITUTE THE LAST-BUT-ONE OCCURRENCE OF THE TERMINAL-NODE T3 IN
THE SYMMETRIC LIST-STRUCTURE L3 BY THE COMPONENT-LIST GIVEN
 >
FIND THE LENGTH OF THE LIST-STRUCTURE GIVEN
 >
FIND IF THE CIRCULAR LIST-STRUCTURE L1 IS RECURSIVE
 >
WRITE A PROGRAM TO CONCATENATE THE LISTS LIST-L1 AND LIST-L2
 >
DELETE ALL OCCURRENCES OF TERMINAL-NODES OF THE TYPE I IN THE
SYMMETRIC LIST-STRUCTURE GIVEN
```

Fig.8 A Set of Problems Generated by Interpreting Brief Problem Specifications

The high-level descriptions of problems indicate to objects and processes which are to be referred to in the problem to be generated. Reference is made to them directly or, sometimes, indirectly by mentioning sets of entities from which the generator may choose suitable members. Fig.9 shows some of the objects and processes referred to by the problems being discussed and their organization into progressively larger sets. The interpreter has to recognize case relationships to find the function each entity is to perform in the problem and use this information to control the generative process.

The simple interpreter which produced the problems of Fig.8 from high-level descriptions was implemented by defining a 'search routine' for each word category, the categorization being basically syntactic. Interpretation is performed by executing the search routine associated with each word, allowing it to find other words in the sentence which are related to it. This process results in the construction of parts of deep-structures of potential sentences to describe entities being referred to and their relationships to each other. The associative schemes referred to on page 24 are useful here too. After this parsing, the interpreter triggers off the problem generator giving it the references to the entities it must use in its operation. The generator processes any indirect references to obtain suitable entities which are to be used in problem generation and completes its task.

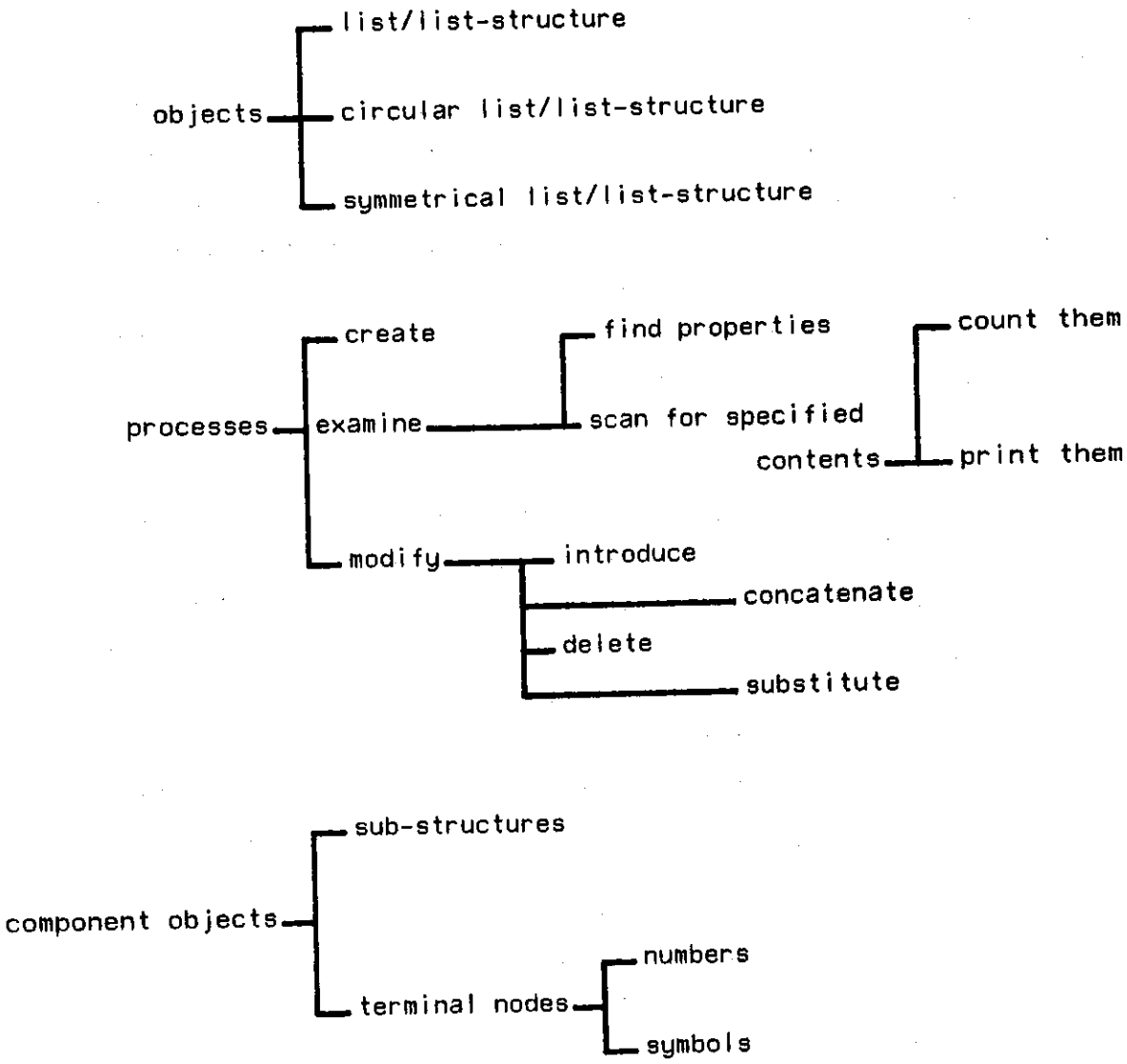

Fig.9 Alternatives for the Choice of Components for the Problems in Fig.8

While explicitly specified choices are easily forced upon the problem generator, there are several possibilities for handling the choices that are not forced. One such possibility is that information about the generation of sub-problems can be accumulated in the course of generating a series of problems. This information can then be used to ensure that sub-problems are well distributed in a generating run. For instance, from the knowledge that a particular series of tasks have been successfully carried out by a student, a set of sub-problems that he has carried out can often be identified. It can then be ensured that later problems do not incorporate these sub-problems already familiar to the student, unless there are special reasons. There is also the possibility that hierarchic relationships between tasks can be exploited to produce sequences of problems of graded difficulty. Fig.10 shows recognizable sub-tasks which are incorporated in the problems of Fig.4, organized to exhibit hierarchic dependencies. The sub-tasks appearing in underlined script are complete in the sense of being usable as self-contained exercises. Appropriate organization of the generator should allow such exercises to be produced when needed, without use of any additional machinery.

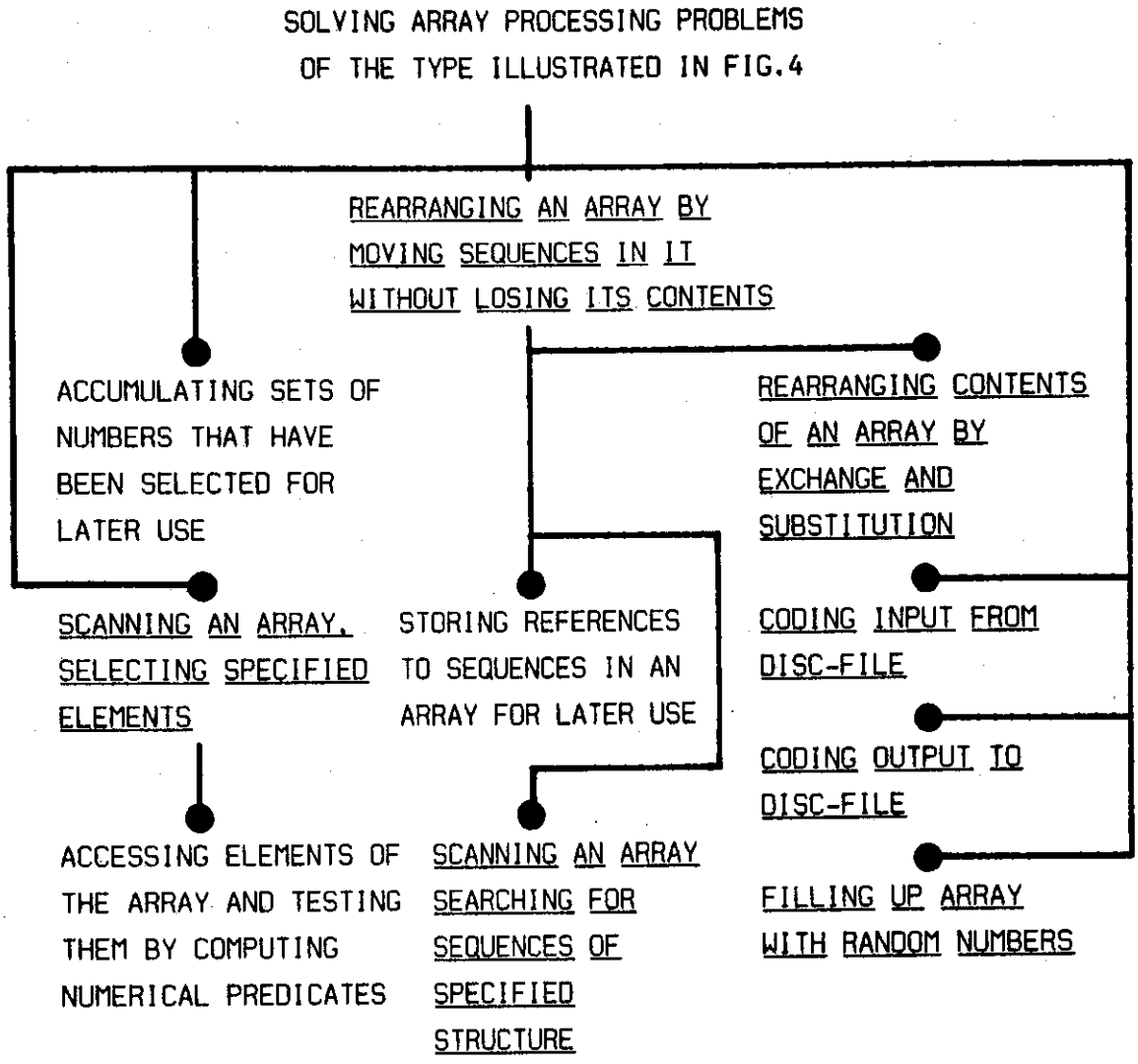

Fig.10 Sub-Tasks Encountered in the Array Processing Problems of Fig.4

4. GENERALIZATION OF A PROTOTYPICAL PROBLEM

Is it possible to start from a well-known problem and write a generator for producing useful generalizations of the original problem? This section describes experiments with this technique of generalization, applying it to a problem first discussed by Hoare(1964) and then by Djikstra (1971). The problem is:

> Rearrange the elements of the array A[1:N] such that
> for a given value of f (1≤f≤N)
> A[k]≤A[f] if 1≤k<f and
> A[f]≤A[k] if f<k≤N .

The required arrangement is said to 'split' the array around f.

As a programming exercise this task has manifold appeal. It confronts the student with the problem of planning manipulations which interact heavily. If we eliminate the brute force solution of sorting the array, perhaps by adding an economy clause to the problem statement, the solution is no longer obvious. Djikstra describes an elegant solution based on recursion.

We look at it from another point of view, attempting to generalize it into a class of array-ordering problems which test the student's understanding of the concepts relevant to sorting. We do not specifically concern ourselves with the existence or non-existence of an elegant recursive solution to the problems generated.

Fig.11 lists some of the skills needed to program basic sorting, in an order intended to indicate the dependence of some of these skills on others, the more basic ones being generally lower in the list.

REARRANGING ARRAY AS SPECIFIED

MANIPULATING ELEMENTS OF AN ARRAY LOCALLY
(SUBSTITUTIONS AND INTERCHANGES)

PERFORMING ARITHMETIC ON ELEMENTS OF ARRAY
(COMPUTING FUNCTIONS AND PREDICATES
DEFINED OVER ELEMENTS OF THE ARRAY)

SCANNING ARRAY FOR SPECIFIED ELEMENTS

ACCESSING SPECIFIED ELEMENTS OF ARRAY
BY GENERATING DESIRED SEQUENCE OF SUBSCRIPTS

Fig.11 Elements of the Sorting Exercise

In terms of these operations, sorting turns out to be a rearrangement of a given array A[1:N] such that A[j] ≥ A[i] if j ≥ i .

A simple generalization of the Hoare problem which still has the listed concepts as essentials is:

> Rearrange elements of the array A[1:N] such that
> if R1(<i>,<j>) then R2(A<i>,A<j>)
> where R1 and R2 stand for relations and <i> and <j> are generator variables which are to be substituted by a variable such as i,j,k or, when suitable, by a numerical constant.

A related set of problems is of the form:

Rearrange elements of array A[1:N] such that
if P1(<i>) then P2(A[<i>])
where P1 and P2 are predicates.

Fig.12 shows some of the problems generated by a program based on the two generalizations indicated above. Comparison of these problems with those presented in Section 3 shows that these problems have an element of programming difficulty not generally present in the earlier problems. The programmer now has to tackle the question of space needed by the contents of the array. Moving the content of a cell requires that space be found for it elsewhere. Creating space could involve disturbing elements which have been properly placed already and considerable interaction between desirable changes is caused thereby. This is a significant aspect of the original problem that is preserved.

```
>
 ARRANGE THE NUMBERS IN THE ARRAY A[1,40] SUCH-THAT THE ELEMENT IN
CELL A[I] IS A SQUARE IF I IS A CUBE.
>
 ARRANGE THE NUMBERS IN THE ARRAY A[1,25] SUCH-THAT THE ELEMENT A[I]
IS COMMENSURATE WITH THE ELEMENT A[11] WHEN I IS LESS THAN 11.
>
 ARRANGE THE NUMBERS IN THE ARRAY A[1,30] SUCH-THAT THE ELEMENT A[I]
IS LESS THAN THE ELEMENT A[13] WHEN I IS DIVISIBLE BY 13.
>
 ARRANGE THE NUMBERS IN THE ARRAY A[1,20] SUCH-THAT THE ELEMENT IN
ANY AN EVEN-NUMBERED CELL IS A SQUARE.
>
 ARRANGE THE NUMBERS IN THE ARRAY A[1,30] SUCH-THAT THE ELEMENT A[I]
IS LESS THAN THE ELEMENT A[7] WHEN I IS COMMENSURATE WITH 7.
>
 ARRANGE THE NUMBERS IN THE ARRAY A[1,40] SUCH-THAT THE ELEMENT A[I]
IS COMMENSURATE WITH THE ELEMENT A[J] WHEN I IS COMMENSURATE WITH J.
```

Fig.12 Array Ordering Problems

A sequence of several problems derived from the same prototype would be unsuitable for presentation to a student, if they show no more variation than the set of problems in Fig.12. A library of prototypes could provide some coverage and variety. On the other hand, focus on selected concepts would be useful, even at the expense of variety, when producing problems for certain purposes: for introducing important concepts, for testing, for remediation or for review. If the student has a convenient addressing facility, he would often find it useful to ask for a few similar problems involving a specific set of concepts to confirm his own understanding of these, before he moves on to other concepts.

5. REASONING ABOUT ACTIONS

The basic characteristic of a problem situation is that the scheme of actions leading to a solution is not obvious, while the elements out of which such a scheme could be constructed are more or less available. It is generally important in problem generation to estimate and control the complexity of the action-schemes which might lead to solutions, ensuring that the problem presents a desired level of difficulty to the student. To some extent, the generator may also predetermine the content and structure of fruitful action-schemes.

One device for the estimation of complexity of a problem involves the visualization of possible problem spaces that might be searched for a solution. In the case of programming exercises, the cost of searching the most easily intuited space of solutions can often be estimated easily by the generator. Ensuring this cost is too high forces a step of refinement on the student, making him look for clues for augmenting the problem-space or, in some cases, compelling him to restructure the program, for example by providing for storage of computed values to reduce computational cost or by substituting iteration for recursion. When dealing with well-structured problems, a lower bound for the complexity of the program needed to solve a given problem may be estimated by identifying those features of the program which have been rendered essential. A generator could use such estimates to arrive at problems presenting desired levels of difficulty.

A second device that could be employed is the programming of limited deduction about the problem situation. The problems appearing in underlined script in Fig.13 provide good examples of situations where the scheme of actions needed to perform the given

task is of controlled complexity (1). Each of these problems involves an action or a set of actions that is to be performed in order to achieve a desired change or to achieve a desired state in the given situation. The task is meaningful only if the action specified is relevant to the state change desired and if the performance of this action presents a desired level of difficulty. In order to take these requirements into account the problem generator has to have information relating possible actions to their effects. It also needs estimates of difficulty in carrying out possible actions (i.e. 'applying operators', to use the wording in Newell and Simon, 1972). The use of a table of connections (ibid) is a possibility. Deducing the information by following a suitable line of reasoning may be necessary in those cases where a table of connections becomes unwieldy.

The problems under discussion deal only with a few possible types of simple manipulations that can be performed on graphs, i.e. addition or deletion of vertices and edges and the repositioning of edges. Each type of manipulation is generally capable of multiple application to a graph.

---

(1) The other problems shown in this figure are not discussed here. They were produced by generators having the structure discussed in Section 3 and share association networks with the problems being discussed in this section.

≥ DELETE LESS THAN 3 EDGES FROM THE GRAPH G TO INCREASE ITS RADIUS TO 3.

> ASSIGN NUMBERS IN THE RANGE FROM 1 TO 10 TO THE EDGES OF THE GIVEN GRAPH SUCH THAT THE NUMBER ASSIGNED TO AN EDGE IS COMMENSURATE WITH THE TOTAL NUMBER OF THE EDGES ADJACENT TO IT. ENSURE THAT THE SUM OF THE ASSIGNED NUMBERS IS A MAXIMUM.

> FIND BRIDGES WITH A LENGTH OF 4 IN THE GIVEN GRAPH.

≥ ADD LESS THAN 5 EDGES TO THE GRAPH G SUCH THAT IT IS NO LONGER A PLANAR GRAPH.

> FIND IF THERE ARE CYCLES IN THE GRAPH G.

≥ ADD LESS THAN 3 NEW VERTICES WITH NECESSARY INCIDENT EDGES TO THE GRAPH G SUCH THAT THE VERTICES V7 V2 V10 V6 AND V1 ARE NO LONGER THE CENTRAL VERTICES.

> FIND CYCLES IN THE GRAPH G.

> ASSIGN A COLOR NAME TO EACH EDGE SUCH THAT THE COLOR OF ANY EDGE IS DIFFERENT FROM THE COLOR OF ALL ITS ADJACENT EDGES.

> ASSIGN NUMBERS, NOT NECESSARILY UNIQUE, IN THE RANGE FROM 1 TO 15 TO THE EDGES OF THE GRAPH G SUCH THAT THE NUMBER ASSIGNED TO AN EDGE IS MULTIPLE THE NUMBER OF THE CYCLES IT IS A MEMBER OF.

> GENERATE BRIDGES WITH A LENGTH OF 4 FROM THE GRAPH G.

> FIND IF THE GIVEN GRAPH IS A BLOCK.

≥ REPOSITION 3 OR MORE EDGES IN THE GRAPH G SUCH THAT THE ECCENTRICITY OF V9 REMAINS UNCHANGED.

Fig.13 Graph Manipulation Problems

Some of the problems require that the graph be modified to attain a given property by performing only a limited number of a specified type of manipulation. Other problems require that a given number of manipulations of a specified type be performed without altering the graph in a specified way. Since these are programming problems, the real task is to write a program which performs the required manipulations on any graph given as its input.

The problems are interesting because performance of the task under the given constraints becomes a critical matter, requiring considerable analysis of the problem situation and careful planning of the solution. A special difficulty is that the solution should be applicable to an indefinitely large set of graphs any of which may be given as input to the program that is to be written. These features, and the presence of a body of theory on graphs, make these problems good exercises in non-numerical programming.

The generator which produced the examples in Fig.13 performs limited deduction to arrive at the relation between the five allowed manipulations on graphs to a dozen or so state changes that could occur in the graphs as a result. In the case of the problems in Fig.13, the deduction is based on the heuristic use of knowledge about gross changes that occur in the graph as a result of simple manipulations. Two gross indicators are the change in the number of paths between an arbitrary pair of points as a result of a manipulation and the change in distance between an arbitrary pair of points. Table 1 relates manipulations to their effects in terms of these gross indicators. Table 2 relates these indicators to several state changes that can be produced in graphs. In these tables, an entry of a '1' shows a tendency for

the cause to increase the quantity it is related to; and an entry of a '-1' shows the opposite tendency; and an entry of '0' indicates that a predictable link is not present.

Operation	Change in Path Length	Change in Number of Paths
Delete Vertices with edges	+1	-1
Delete Edges	+1	-1
Add Edges	-1	+1
Add Vertices and edges	0	+1
Reposition Edges	0	0

Table 1 Operators and Their Effect on Graph Parameters

State Descriptors	Increase in Path Length	Increase in Number Of Paths
Radius	+1	-1
Diameter	+1	-1
Eccentricity	+1	-1
Connectivity	0	+1
Number of Components	0	-1
Planarity	0	-1
Possibility of Hamiltonian Paths	0	+1
Identity of Center	0	0
Distance Between Two Nodes	+1	-1
Possibility of Eulerian Paths	0	0

Table 2 State Changes and Associated Variations in Graph Parameters

To generate the examples referred to, the generator chooses a permitted manipulation and identifies a relevant state change by using the tables. Further, it computes from these tables the direction of state change and introduces a suitable constraint which ensures that the task being constructed is not too easy.

Extensions can easily be made to the generator employing the line of reasoning described above. Consider, for instance, the problems which require that a particular state change be realized without altering a specified feature of the given graph. A simple computation involving Table 2 enables the generator to produce pairs of state changes requiring opposite directions of change in the two parameters involved. A reference to Table 1 identifies a relevant manipulation on the graph. These two steps enable the generator to create a problem by requiring that the specified manipulation be repeatedly performed to obtain a given state change ensuring that the other state change does not occur.

6. COVER STORIES

Large classes of problems can be viewed as having an abstract structure. In these problems, the specific objects functions or relations that appear are not of much significance, and are generally substituted by formal variables early in the problem-solving process. The same abstract-structure obtained from many problems which differ only trivially describe the essence of the problem situation. School text-books on mathematical subjects employ the well-known technique of taking interesting problem-structures and fleshing them out with sufficient surface detail to generate exercises. It is conceivable that problems generated by this technique, apart from providing variety , also focus attention on the skills necessary to abstract the basic structure of given problems (1). The process of creating a coherent surface structure is worth a brief discussion, even though we have not yet implemented a program to illustrate it.

At the lowest level of sophistication would be schemes which operate with a fixed number of abstract problem-structures and a fixed number of surface structures. Using some indexing scheme, a compatible pair of structures would be selected and the formal varibles in the abstract structure would be replaced by the corresponding terms of the surface structure. At the other end of the spectrum would be an understanding system which has strategies for finding or creating real-world situations corresponding to any given abstract problem-structure.

---

(1) Hayes and Simon (1973) are studying subjects' behavior when presented with differently structured external cover stories for the same abstract problem structure (tower of hanoi).

Possible compromises between what is desirable and what is implementable include systems which exploit strong links between a large body of theory of a class of structures and a related area of 'real-world' problems with adequate diversity. Consider, for instance, problems involving transport and communication networks. It is obvious that many graph manipulation problems can be paraphrased as problems in this area. Implementing a system which can perform such a paraphrasing intelligently involves several problems. Such a system could use a set of associations linking entities in the abstract theory with corresponding entities in the application area. A partial mapping may be possible, and it will indicate, to some extent, those aspects of the theory that are relevant to the application area. For example, the concept of a weighted graph is specially relevant to the application area being considered here because it provides for the possibility of taking into account physical lengths of the edges.

Unless such information on relevance is used to shape the abstract problem being created, only a very small fraction of generated structures can be paraphrased as natural-looking application area problems. However, the complexity of detail that has to be attended to appears formidable. This is best illustrated with examples. In the area of transport and communication networks, some of the information that should be available to the generator, directly or indirectly, is:

Road networks which are non-planar have to be constructed with the use of underpasses and overpasses which are expensive in relation to the cost of roads. This is not the case with air-routes and telecommunication circuits.

The presence of cut-vertices and small cut-sets is related to the problems of reliability of the network.

Paths passing through every edge (Eulerian paths) are relevant to bus routes (edges being streets), while they are not of significance in the case of telephone networks.

It is possible to think of railroad stations as fuelling or non-fuelling stations (say for generating problems involving dominant sets of vertices in a graph). This is generally not true in the case of airports, all of which can provide fuel.

These examples illustrate the wealth of knowledge that a human designer brings to bear on the task of creating cover stories. Programming a generator to design cover stories therefore has to face the problems of representing the basis for such knowledge and of utilizing it.

7. CONCLUSIONS

While simple generative schemes do, in some cases, produce surprisingly coherent questions, it is clear that the creation of a good problem is in general a design task which involves considerable knowledge of the task area. Detailed representations of relevant knowledge of the task area are as important in problem generators as they are in the case of any program which performs a cognitively significant task. These representations structure the task area in terms of objects and processes, assign attributes to these entities and provide static links as well as computational routines to implement functional and relational mappings.

Given such a representation of knowledge, the questions that arise about a problem generator concern its structure and the strategies that it incorporates. Are generators arbitrary programs which have to be conceived ad-hoc, or is there a common set of principles they can be based upon?

We have developed the view that generators perform a design task, assembling a complex object - the problem - while operating within a variety of constraints. A top-down generative control-structure appears attractive. The second suggestion was that this generative mechanism be freed from the burden of syntactic considerations it carries in the case of grammar-like mechanisms. The data-structures it operates with could be deep-structures of sentences, providing a rich representational scheme easily extended by borrowing from natural language. A sentence synthesizer which creates readable descriptions of generated problems was described.

Problem generators for specific areas have to be developed by inventing appropriate representations for knowledge about problems in that area. Some sets of problems have a prototype and a process of what might be called semantic generalization results in a set of problems which are similar to the original and yet provide significant diversity. Some sets of problems can be created by performing limited deduction about operators and state changes they cause in the relevant problem space. These strategies have been discussed and illustrated with examples produced by programs based on them.

Another strategy, which is based on the design of cover stories for good abstract problems has been introduced. Problems of controlling the generative process by using explicit specifications and the problems of using information available regarding the student's familiarity with a set of concepts have also been briefly dealt with.

Acknowledgement: One of the authors (S.R.) wishes to acknowledge the award of a Homi Bhabha Fellowship which provided partial support for this research. he also wishes to thank Dr. Mathai Joseph for helpful comments and suggestions on problem generation in the early stages of the work reported.

APPENDIX A: PROGRAMMING DETAILS

1. Languages Used

All the generators mentioned in the preceding pages were programmed in the list processing implementation language L✲(F) (Newell, et al, Jan. 1971). The main implication of the use of L✲(F) is that one adopts a specific approach to system-building (Newell, et al, Sept. 71). This approach leads to the construction of large systems in a layered fashion, adding facility after facility (e.g.,: an association mechanism based on the use of hash-coding techniques for accessing association lists; the semantic parser discussed in Section 3.3) in a number of steps, later layers using the tools provided by the earlier ones and building on them. The other implication of the use of (L✲(F) is that the total access provided to the language processor's machinery encourages the user to adopt this machinnery when needed rather than building a language processor on his own.

2. Primitive operations

The basic operations of random choice, weighted choice, value assignment, creation of associations and accessing of associations are specified by the use of the operators ↑ , # , ← , <= , ? and $, the last one being the statement terminator used in conjunction with the operators ← and <= . These six operators as well as the comma are assigned necessary 'character actions' and 'syntax actions' (Newell, et al, Jan. 1971) as per the conventions of L✲(F), ensuring the proper parsing and interpretation of expressions of the following form embedded within regular L✲(F) programs.

```
                        ; Comment fields follow the
                        ; semi-colons.
[A1 A2 A3 A4]           ; Create the specified data-list
                        ; (of type list on L*(F)).
[A1 A2 A3 A4] ↑         ; Choose an item randomly from
                        ; the given list.
[A1  3 A2  5 A3] ↑      ; Choose by a weighted random
                        ; process.  Weights are prefixed
                        ; with the number sign.
[R1 R2 R3] ↑ X          ; Choose one of the routines in
                        ; the list and execute it, obtaining
                        ; a value.  Expressions of the above
                        ; forms can be used in the place of
                        ; the variable B in the following
                        ; statements.
A ← B $ $               ; Replace the contents of list A
                        ; by the contents of list B.  used
                        ; mainly to assign values to variables.
ATT, OBJ <= B $         ; Associates a value, B, with the
                        ; given attribute of the given object.
ATT, OBJ ?              ; Obtains the value of the given
                        ; attribute of the given object.
ATT, -- ?               ; Obtains the value of the given
                        ; attribute of the object on top
                        ; of the L*(F) working stack.
```

The brief description above does not cover all primitives used in the system. However, a brief mention should be made of variants of the choice operator ↑. One has a memory cell for its most recent choice and avoids repeating it. Another variant refers to information associatively linked to its data-list by other parts of the program before making a selection. This latter

variant is useful in programming generators which are controllable.

3. Creation of Data-Structures

A structure synthesizing routine .C is used extensively by the problem generators. The execution of a statement of the form

([TYP, ASUB1 VSUB1, ASUB2, VSUB2, ASUB3 VSUB3] .C)

creates a list representing a token, or instance, of an entity of the type TYP. Associatively linked to this list are the attributes ASUB1, ASUB2 and ASUB3 which are assigned to values VSUB1, VSUB2 and VSUB3 respectively. If any attribute value pair ASUBI VSUBI is followed by the separator .e instead of the comma, VSUBI is executed (it is presumed to be a routine) and the value obtained is assigned to the attribute ASUBI. SIMILARLY IF TYP is followed by .E, the type is obtained by executing the routine TYP, the type is obtained by executing the routine TYP. Examples of the usage follow:

([LIST, LINK-TYPE SYMMETRICAL, UNIQUE TRUE] .C)

creates a structure to represent 'THE SYMMETRICAL LIST'.

([ARRAY, DIMENSION 2, NUMBER 3] .C)

creates a structure to represent 'THREE TWO DIMENSIONAL ARRAYS'.

There are special separators which are used in the place of the comma following the type to specify in a compact manner the values of the attributes UNIQUE and NUMBER. These separators are not used in the illustrations in this Appendix in order to keep the illustrations simple.

The main reason for the utility of the routine .C in the programming of top-down generators is that it can be used resursively. A specification list on which .C operates may

indicate that the value of ASUBI should be computed by the routine VSUBI. Frequently, VSUBI itself is defined in terms of a construction to be performed by .C, creating a sub-structure which is linked to the main structure by the ASUBI -- VSUBI bond. The following example illustrates such usage. The routine TP specifies the construction of members of a set of simple sentence-structures.

```
TP:   ((TP1 .E OBJ. TP2 .E LOCATIVE TP3 .E) .C)
TP1:  ((FIND DELETE PRINT) ↑
TP2:  ((NUMBER, QUANT ALL, UNIQUE TRUE, NUMBER 0,
         ADJC TP21 .E) .C)
TP21: ((ODD EVEN POSITIVE NEGATIVE PRIME ↑)
```

Assignment of '0' to NUMBER is interpreted to mean that NUMBER>1. The routine TP3, not defined here, accesses a fragment of sentence-structure assumed to have been created before the execution of the statements in the example. The accessed fragment represents an object such as 'THE LIST L1', 'THE GIVEN LIST', 'THE ARRAY L(20)' or THE ARRAY L(30, 30)'. TP3 synthesizes a sentence-fragment to refer to the object or its sub-objects. The new fragments are translatable as 'THE FIRST ROW OF THE ARRAY', 'THE ARRAY L', 'ALL THE SUBLISTS OF L' or 'THE LAST COLUMN OF L'.

The sentence synthesizer, briefly described in Appendix B, synthesizes readable sentences from the sentence-structures of the kind produced by the routine TP, to generate sentences such as the following:

```
FIND ALL THE PRIME NUMBERS IN THE LAST COLUMN OF THE ARRAY L.
DELETE ALL THE EVEN NUMBERS IN THE FIRST SUBLIST OF L.
PRINT ALL THE POSITIVE NUMBERS IN THE ARRAY L.
```

APPENDIX B: THE SENTENCE SYNTHESIZER

The sentence synthesizer deals with associative structures representing the content of potential sentences or parts thereof. Fig.7 showed an example of such a structure. By convention, the sentence structure as given to the synthesizer starts with the list-structure representing an occurrence of a predicate term. Linked to this list-structure by labelled, directed associations are sub-structures representing entities that stand in various deep-case relations to the predicate. Each sub-structure similarly consists of a list-structure representing the occurrence of a word-sense with associations leading off to related entities. It is also the convention to bind each occurrence of a word-sense with relevant markers such as those for tense, number and gender using attribute-value associations.

The implementation of the synthesizer is based on the principle that for each category of semantic entity in the sentence structure there should be a synthesis routine which has operational knowledge of that category of entity. At the highest level, the synthesizer finds out the category of entity it has to deal with and triggers off the associated synthesis routine. Recursive calls by one synthesis routine to others are necessary because parts of the sentence structure are composed of smaller sub-structures.

The routine dealing with predicates looks for associations to sub-structures of the following types: subject; voice marker; tense marker; object, in case of transitive verbs; indirect objects; locatives; adverbial of purpose; and other adverbial phrases. The routine dealing with objects attends to any need for

:terminers, quantifiers, adjectives, number markers, and several
ypes of adjectives and adjectival phrases.

REFERENCES

1. Carbonell, J. R., "AI in CAI: An Artificial-Intelligence Approach to Computer-Assisted Instruction," IEEE Transactions on Man-Machine Systems, MMS-11 (4), 190-202, (Dec., 1970)

2. Hayes, J. R. and H. A. Simon, "Understanding Written Problem Instructions," Complex Information Processing Working Paper 236, Department of Psychology, Carnegie-Mellon University, (May, 1973)

3. Koffman, E.B., "A Generative CAI Tutor for Computer Science Concepts," Proceedings of the Spring Joint Computer Conference, 1972.

4. Koffman, E.B., "Individualizing Instruction in a Generative CAI Tutor Communications of the ACM, 15 (6), pp. 472-473, (1972).

5. Newell, A., D. McCracken, G. Robertson, and L. DeBenedetti, "L☆(F)," Department of Computer Science, Carnegie-mellon University, (Jan., 1971)

6. Newell, A., P. Freeman, D. McCracken, and G. Robertson, "The Kernel Approach to Building Software Systems," Computer Science Research Review, 1970-71, Department of Computer Science, Carnegie-Mellon University, (Sept., 1971)

7. Newell, A. and H.A.Simon, Human Problem Solving, Prentice Hall, Englewood Cliffs, N.J., (1972)

8. Simmons, R. and J. Slocum, Generating English Discourse from Semantic Networks, Communications of the ACM, 15 (10), pp. 891-905 (1972).

9. Uhr, L., "Teaching Machine Programs that Generate Problems as a Function of Interaction with Students," Proceedings of the 24th ACM National Conference, (New York, N.Y.), pp. 125-134, 1969.

10. Uttal, W. R., T.Pasich, M.Rogers and R.Hieronymus, Generative Computer-Assisted Instruction in Analytic Geometry, Entelek, Newburyport, Mass. (1970) 11. Wexler, J. D., "Information Networks in Generative Computer-Assisted Instruction," IEEE Transactions on Man-Machine Systems, MMS-11 (4), 181-190, (Dec., 1970)