Example Generation

Edwina L. Rissland

COINS Technical Report 80-14

June 1980

Example Generation

Edwina L. Rissland

Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

Abstract

This paper addresses the problem of generating examples that meet specified properties which are used to direct and constrain the generation process, which we call CONSTRAINED EXAMPLE GENERATION. We begin by presenting a few examples of CEG taken from protocols. Based upon such examples, we present a model of the CEG process. We describe the architecture of a system that generates examples from specifications and present examples of problems that it has solved.

1. INTRODUCTION

The ability to generate examples that have specified properties is important in many intellectual areas, such as mathematics, linguistics and computer science [Collins 1979]. It is important from the standpoints of learning and teaching as well as performing research. For instance, examples are needed for inductive reasoning, sharpening of conjectures, and concept formation and refinement [Polya 1968, 1973; Lakatos 1963; Winston 1975; Lenat 1976; Soloway 1978]. Having a rich stock of examples is intimately related to understanding [Rissland 1978a, b]. Thus, examples lie at the heart of efforts to learn and reason in a subject.

When an example is called for, one can search through one's storehouse of known examples for an example that matches the properties of the desired example. If a satisfactory match is found, then the problem has been solved through retrieval.

However, when a match is not found, how does one proceed? In many cases, one modifies an existing example that is judged to be close to the desired example, or to have the potential for being modified to meet the constraints.

In some cases, generation through modification fails. Experienced researchers, teachers and learners do not give up however. Rather they switch to another mode of example generation which involves building up an example from very elementary consituents through careful attention to the desiderata and "unpacking" of the concepts involved. This phase of CEG is usually more difficult than either retrieval or modification.

This paper presents a model of CEG that incorporates three phases: RETRIEVAL, MODIFICATION, and CONSTRUCTION. This model is based upon analyses of protocols of example generation tasks taken from mathematics and computer science [Rissland 1979, Woolf and Soloway 1980].

2. PROTOCOLS OF CEG

In this section, we describe some protocols for CEG tasks taken from the domain of elementary function theory in mathematics (which deals with concepts such as continuity) and from elementary LISP programming (especially with regard to concepts concerning list structure).

2.1 Examples of Retrieval

The type of questions that most people answered through retrieval is:

Give an example of a function that is continuous
but not differentiable (at a point).

Give an example of a list
with three elements.

Most people handled these problems by offering their favorite standard "reference" examples [Rissland 1978a, b]: for the first problem, the absolute value function (at the origin) and for the second, a list like "(A B C)". Responses were usually immediate indicating that the retrieval was very readily made.

2.2 Examples of Modification

An example of a problem solved through modification of a known example is:

Give an example of a list
with three elements
where the depth of the first atom is 3.

Subjects frequently modified an example, such as "(A B C)" by adding two more parentheses around the first element, to produce the list

(((A)) B C)

Other subjects truncated a longer list such as the list of digits or added to a shorter list such as (0 1), as well as adding parentheses. The example chosen for modification depends on the context of the problem (e.g., the sequence of recently solved problems) and the subject's data base of examples and its epistemology (e.g., his favorite references).

An example of a mathematics problem which every subject solved by modification is the following:

Give an example of a non-negative,
continuous function
defined on the entire real line
with the value 1000 at 1, and
with area under its curve less than 1/1000.

Most protocols for this question began with the subject selecting a function (usually, a familiar reference example function) and then modifying it to bring in into agreement with the specifications of the problem.

There were several clusters of responses according to the initial function selected and the stream of the modifications pursued. A typical protocol went as follows:

"Start with the function for a "normal distribution". Move it to the right so that it is centered over x=1. Now make it "skinny" by squeezing in the sides and stretching the top so that it hits the point (1, 1000)."

"I can make the area as small as I please by squeezing in the sides and feathering off the sides. But to demonstrate that the area is indeed less than 1/1000, I'll have to do an integration, which is going to be a bother."

"Hmmm. My candidate function is smoother than it need be: the problem asked only for continuity and not differentiability. So let me relax my example to be a "hat" function because I know how to find the areas of triangles. That is, make my function be a function with apex at (1, 1000) and with steeply sloping sides down to the x-axis a little bit on either side of of x=1, and 0 outside to the right and left. (This is OK, because you only asked for non-negative.) Again by squeezing, I can make the area under the function (i.e., the triangle's area) be as small as I please, and I'm done."

Comments

Notice the important use of such operations as "squeezing", "stretching" and "feathering", which are usually not included in the mathematical kit-bag since they lack formality, and descriptors such as "hat" and "apex". All subjects made heavy use of curve sketches and diagrams, and some used their hands to "kinesthetically" describe their functions. Thus the representations and techniques used are very rich.

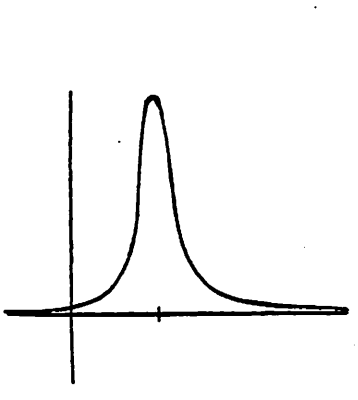

FIG 1A

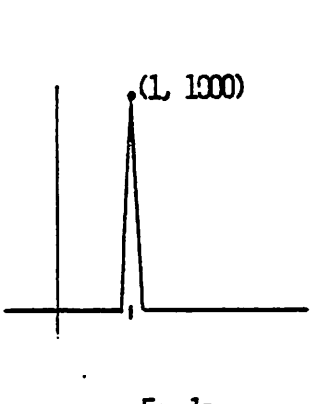

FIG 1B

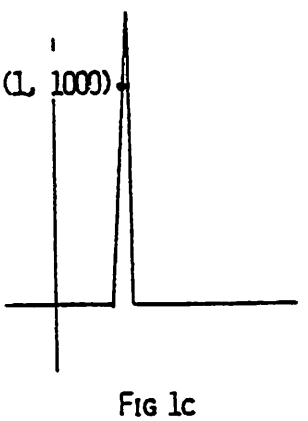

FIG 1C

Another thing observed in all the protocols (of which there were about two dozen for this problem) is that subjects make implicit assumptions about the symmetry of the function (i.e., about the line x=1) and its maximum (i.e., occuring at x=1 and being equal to 1000). There are no specifications about either of these properties in the problem statement; however, they are mathematically simplifying and cognitively natural.

These are the sort of tacit assumptions that Lakatos talks about [Lakatos 1963]; teasing them out is important to study both mathematics and cognition.

Example functions for protocols are shown in Figures 1a and 1b; another mathematically permissible example is shown in 1c.

2.3 An example of Construction

In this subsection, we present a protocol of example generation in which the example is built largely "from scratch" by working with the concepts involved in the specifications of the desiderata, instantiating them, and combining exemplars to produce a new example. The problem is:

> Give an example of a list of lists
> each of which has two elements
> the first of which is a literal atom.

A typical protocol began with the subject sketching out the overall structure of the desired list as:

```
( (A1 L1)
  (A2 L2)
  (A3 L3)
  ...
        )
```

where in each sublist, Ai stands for a literal atom, and Xi the second element.

The subject next focussed his attention on instantiating the Xi's. Since he wanted to emphasize the fact that the elements of the sublists could themselves be lists -- "there's a lot of embeddedness possible here" -- he made each of the Xi's a list of atoms (a "LAT").

The subject began to write each Xi as (Ai1 Ai2 ... Ain) and then remarked that this level of generality was more than the problem called for. In particular, nothing was said about keeping the Xi's different: "So, why not make them all the same, like (00 01)".

The candidate example now looks like:

```
( (A1 (00 01))
  (A2 (00 01))
  (A3 ...
                )
```

The subject next decided to pin down the length of the "big" list by making it be "not too short, like 2, and not too long either; why not 7". He tended to the Ai's by noting that A1, A2, A3, ... A7 are perfectly fine literal atoms.

The list thus offered is:

```
( (A1 (00 01))
  (A2 (00 01))
  (A3 ...
  ...
  (A7 (00 01)) )
```

Even though the subject was satisfied with this answer, he noted that it really didn't have to be so complex or long; the following list would do:

```
( (A1 1) (A2 2) (A3 3) )
```

He said he made his list have a length longer than 2 because he didn't want it to be confused with the length of the sublists (i.e., 2). However, he said that a list of length two would be acceptable, but a list of length one would not since "after all the problem called for a list of lists".

"The list:

```
((A B) (A B))
```

would also do just fine. In fact, the possibilities are endless."

Comments

There are several observations to be made on this protocol. First, the subject had a general model of a list and procedures to instantiate it (e.g., generate literal atoms and lists) and he had procedures to modify lists and properties of lists. Second, the subject made several implicit assumptions on the example to be generated, such as (1) its length, (2) the non-repeatedness of some elements, (3) its complexity (e.g., depth), and (4) uniformity (e.g., of list-structure).

3. A CEG MODEL

From analyses of protocols such as presented in Section 2, we developed the following general model of the CEG process. Presented with a task of generating an example that meets specified constraints, one:

(1) SEARCHES for and (possibly) RETRIEVES examples satisfying the constraints. This is done by searching through the knowledge base and judging examples for their match (or partial match) to the desiderata;

(2) MODIFIES an existing example judged to be close or having the potential for fulfilling the desiderata;

(3) CONSTRUCTS an example from elementary knowledge, such as definitions, principles and more elementary examples from the knowledge base.

Thus, there is a spectrum of responses to a CEG task ranging from having a ready answer as in (1) to having no especially close fitting candidate as in (3). In general, Task N depends on and follows Task N-1.

This information processing model of CEG is useful not only in describing human protocols, but also precisely specifying a computational model.

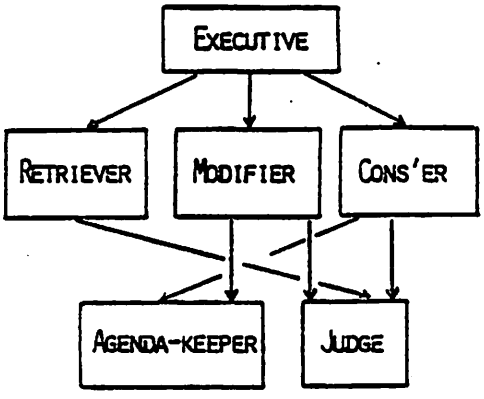

FIG 2

4. ARCHITECTURE OF A CEG SYSTEM

From the model of the last section, we have developed a system that solves CEG problems in the LISP domain. It has also been used to hand-simulate CEG problems in the mathematics of linear and piece-wise linear functions.

We have implemented this CEG model in the LISP domain. Written in LISP, it currently runs interpretively on a VAX 11/780 running under VMS. Examples of problems and solutions are given in Section 6.

The knowledge in our CEG system resides in two major sources: the knowledge base upon which the system runs, and the knowledge embedded in the processes operating on that base. The knowledge consists of general epistemological knowledge (e.g., the structure and types of examples) and domain-specific knowledge (e.g., particular example modification techniques).

The system consists of several components -- roughly one for each of the three phases of the model -- which handle different aspects of CEG. The flow of control between the components is directed by an EXECUTIVE procedure. Figure 2 shows the general architecture of our system.

The components use a common knowledge base which consists of two parts: (1) a "permanent" knowledge base of "Representation-spaces" [Rissland 1978]; and (2) "temporary" knowledge generated during the solution of a CEG problem.

There are four representation spaces, each of which is a set of items, represented as frame-like data structures, and organized according to predecessor-successor relationships. Examples-space, which is by far the most heavily used in our current system, consists of known examples organized according to the relation of constructional derivation reflecting which examples are constructed from which others. The other spaces and their relations are: Concepts-space: definitional dependency; Results-space: logical dependency; and Procedures-space: procedural dependency.

Before the system is given any CEG problems to work on, we create an initial set of representation spaces. The initial state of the Examples-space for the set of problems described in this paper is shown in Figure 3. The spaces are modified -- mostly through the

addition of examples to Examples-space -- as the system works through CEG problems.

The temporary knowledge held by the system during a CEG problem run includes a list of the constraints of the problem, an agenda of candidate examples, and various bookkeeping parameters such as "boxscores", "constraint satisfaction counts" and "recency counts".

5. CEG SYSTEM COMPONENTS

(1) The EXECUTIVE is responsible for initializing the system for a CEG problem, directing the flow of control among the components, and cleaning up afterwards. It accepts a CEG problem in prescribed format from the user and sets up the problem specifications in the temporary knowledge base.

The problem desiderata are kept on the CONSTRAINT-LIST, which has as many entries as there are constraints. Each constraint is recorded as a pair of properties DESIRED-PROPERTY and DESIRED-VALUE. For instance, the specification of the three constraint problem of "a list, of length 3, where the depth of the first atom is 1" is recorded by the following properties (PLIST's) for the constraints:

```
CONSTRAINT-1    DESIRED-PROP: (TYPE X)
                DESIRED-VALUE: LIST

CONSTRAINT-2    DESIRED-PROP: (LENGTH X)
                DESIRED-VALUE: 3

CONSTRAINT-3    DESIRED-PROP: (DEPTH
                    (FIRST-ATOM X) X)
                DESIRED-VALUE: 1

                Problem 1
```

The EXECUTIVE dictates the behavior of the system as a whole by specifying the orderings used by the other processes, such as the order of retrieval of candidate examples used by the RETRIEVER and the order of application for modification techniques used by the MODIFIER.

(2) The RETRIEVER searches the knowledge base for examples on request from the EXECUTIVE. It searches through Examples-space by examining examples in an order specified in terms of attributes such as "epistemological class" [Rissland 1978], position in the Examples-graph, and recency of creation.

In the problems described in Section 6, the "retrieval order" used was:

```
reference examples before
counter-examples before
start-up examples before
examples without epistemological
          class attribute
```

and in the case of ties

```
predecessors before
successors.
```

This retrieval order biases the system to examine ubiquitous and earlier-contructed examples before others. The order of CANDIDATES retrieved from the initial Examples-space of Figure 3 is thus:

(A B C)

(0 1 2 3 4 5 6 7 8 9)

(0 1)

( )

(A)

(A B C D E)

With each new example selected, the RETRIEVER calls the JUDGE to evaluate the example to ascertain how well it satisfies the desiderata.

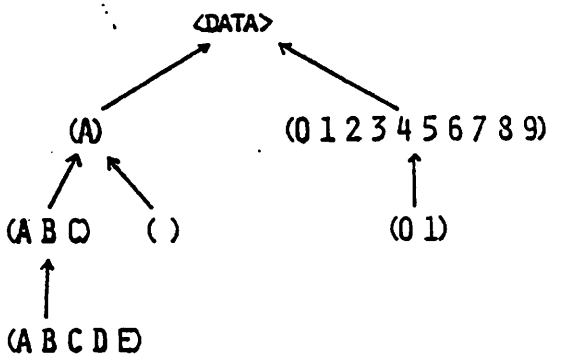

FIG 3

(3) The JUDGE evaluates a CANDIDATE example by cycling through all of the DESIRED-PROPERTY/DESIRED-VALUE pairs on the CONSTRAINT-LIST, comparing them with the actual properties of the CANDIDATE, and recording the results of the comparison. Thus, the JUDGE's basic cycle is evaluation, comparison and record.

The JUDGE records the results of the comparison by FILLING-IN the BOX-SCORE and the CONSTRAINT-SATISFACTION-COUNT ("CSC") slots in the representation frame of the CANDIDATE. The CSC is simply the number of desiderata met by the CANDIDATE.

The BOX-SCORE is a list of 2-tuples, one for each constraint, of the form (ACTUAL-VALUE, T or NIL). The ACTUAL-VALUE is the CANDIDATE's value for the DESIRED-PROPERTY; T is entered if the ACTUAL-VALUE equals the DESIRED-VALUE, and NIL if not.

The BOX-SCORE for the example "(A B C)" in Problem 1 would be:

((LAT T) (3 T) (1 T))

The CSC for this example would be 3, that is, all the constraints are met; the success of this example would be recorded as a T in its "SF" (SUCCESS/FAILURE) slot. With the above retrieval order on the Examples-space of Figure 3, Problem 1 would be solved with the first example retrieved.

If the example "(A)" were judged, its BOX-SCORE would be:

((LAT T) (1 NIL) (1 T))

The CSC for this example would be 2.

(4) The MODIFIER is invoked by the EXECUTIVE when the RETRIEVER has been unable to find an example meeting the constraints from its search through Examples-space.

The MODIFIER calls the AGENDA-KEEPER to set up an agenda of examples to be modified. The MODIFIER then works down the AGENDA trying to modify each entry in turn until success is achieved or the agenda exhausted.

To modify an example, the MODIFIER examines its BOX-SCORE for the constraints that were unsatisfied. It calculates the DIFFERENCE between the DESIRED-VALUE and the ACTUAL-VALUE for each DESIRED-PROPERTY not satisfied. Using the DESIRED-PROPERTY and the DIFFERENCE as an index to a difference-reducing table, the MODIFIER's DIFFERENCE-REDUCER finds and then applies modification techniques to the example.

For instance, for the example "(A)" with a CSC of 2 for Problem 1, the property not met is that of having a length equal to 3. The DIFFERENCE between the DESIRED- and ACTUAL-VALUE is +2. The difference-reducing technique MAKE-LONGER is found by looking for modification techniques affecting the LENGTH attribute of a list and reducing the DIFFERENCE, i.e., by making it longer by 2. (If the difference were -2, as would be the case for the example "(A B C D E)", the appropriate technique would be MAKE-SHORTER).

When there is more than one unsatisfied constraint, the MODIFIER orders its modification attempts according to the order specified by the EXECUTIVE. For the sample problems of this paper, the modification order is to apply techniques that affect:

TYPE before
LENGTH before
DEPTH before
GROUPING

The modified example is then re-judged and a new BOX-SCORE and CSC calculated. If the CSC is improved, the MODIFIER prints a message to the user of "success" or "failed" and asks whether it should continue modifying this example by going through another difference analysis, difference reduction, judgement cycle. If the CSC goes down, the MODIFIER abandons its attempt to bring the example into line, goes on to the next example on the AGENDA, and does not re-queue the example. Thus the MODIFIER engages in a form of hill-climbing.

The modified example must be re-judged for two reasons: (1) some techniques are heuristic and do not guarantee successful modification; and (2) there can be interaction between the constraints, that is, a successful modification for one constraint may undo satisfaction of another.

For instance, the system can make a NESTED-LIST from the LAT "(A B C)" by GROUPing "A" and "B", i.e., "((A B) C)". However, before the modification technique was applied the LENGTH was 3, but now, after modification, it is 2. Satisfaction of the NESTED-LIST constraint has undone the LENGTH 3 constraint.

In the next version of our system, we shall re-judge an example after each modification, and also protect some contraints.

(5) The AGENDA-KEEPER is called by the MODIFIER and CONS'ER to set up the AGENDA of examples to be modified or instantiated.

When called by the MODIFIER, the AGENDA-KEEPER compiles an agenda of items to be modified based upon the CSC's calculated and recorded during the retrieval phase: the examples are ranked in order of their CSC's. Thus, the CSC is used as a measurement of the closeness of the example to meeting the constraints. In the case of a tie, the retrieval ordering is used.

(6) The CONS'ER is called by the EXECUTIVE when the MODIFIER is unsuccessful in its attempts to produce a solution or a model needs to be instantiated. The CONS'ER uses the procedural formulations of concepts stored in Concepts-space.

6. SAMPLE PROBLEMS

[NOTE: Text in this section is actual computer output generated by our CEG system; however explanatory text has been added (indicated by a "$") and some output modified to improve readability.]

Problem 2

$Problem 2 asks for a list of length 3 whose first atom has a depth of 3:

```
(x1 (desired-value list desired-prop
 (typep candidate)))
(x2 (desired-value 3 desired-prop (length
 candidate)))
(x3 (desired-value 3 desired-prop (depth
 (first-atom candidate) candidate)))
```

$The retrieval phase is entered with the Examples-space of Figure 2. The retrieval order of candidates is:

```
<abc>
<**digits>
<**bits>
<empty>
<a>
<abcde>
```

$The RETRIEVER reports on each candidate tried, by printing out its BOXSCORE, CSC and SF:

```
   candidate name = <abc>
 candidate-value = (a b c)
     csc = 2   sf = nil
(entry-x1 (lat t))
(entry-x2 (3 t))
(entry-x3 (1 nil))
"failed"

   candidate name = <**digits>
 candidate-value = (0 1 2 3 4 5 6 7 8 9)
     csc = 1   sf = nil
(entry-x1 (lat t))
(entry-x2 (10 nil))
(entry-x3 (1 nil))
"failed"

   candidate name = <**bits>
 candidate-value = (0 1)
     csc = 1   sf = nil
(entry-x1 (lat t))
(entry-x2 (2 nil))
(entry-x3 (1 nil))
"failed"

   candidate name = <empty>
 candidate-value = nil
     csc = 0   sf = nil
(entry-x1 (atom nil))
(entry-x2 (0 nil))
(entry-x3 (0 nil))
"failed"

   candidate name = <a>
 candidate-value = (a)
     csc = 1   sf = nil
(entry-x1 (lat t))
(entry-x2 (1 nil))
(entry-x3 (1 nil))
"failed"

   candidate name = <abcde>
 candidate-value = (a b c d e)
     csc = 1   sf = nil
(entry-x1 (lat t))
(entry-x2 (5 nil))
(entry-x3 (1 nil))
"failed"
```

$The problem desiderata are not met by any example in the data base, and thus the modification phase is entered.

$The AGENDA of candidates for modification is (the CSC is given after the candidate's name):

```
(<abc> 2)
(<**bits> 1)
(<a> 1)
(<**digits> 1)
(<abcde> 1)
(<empty> 0)
```

```
$The MODIFIER goes to work on the first
candidate, (A B C):

-----------------------------------------
constraint = ((typep candidate) list)
actual score = (entry-x1 (lat t))

modify-candidate   ok

-----------------------------------------
constraint = ((length candidate) 3)
actual score = (entry-x2 (3 t))

modify-candidate   ok

-----------------------------------------
constraint = ((depth (first-atom
 candidate) candidate) 2)
actual score = (entry-x3 (1 nil))

  "find-diff"   (increase-depth-by 2)
  "apply-diff"
      reducer =  make-deeper-x
 new-candidate =  (((a)) b c)

modify-candidate   modified

-----------------------------------------

$The candidate's depth attribute has been
modified   by the modification routine
MAKE-DEEPER-X to produce a new example,
which is then judged and added to the
Examples-space:

   candidate value =  (((a)) b c)
     csc =  3   sf =  t
(entry-x1 (nlist t))
(entry-x2 (3 t))
(entry-x3 (3 t))
("created new frame for example "
 mar11-009 (((a)) b c))
"success!!"
```

Problem 3

```
$The CONSTRAINT-LIST for the next problem
is:

(x1 (desired-value list
     desired-prop (typep candidate)))
(x2 (desired-value 5
     desired-prop (length candidate)))
(x3 (desired-value 2
     desired-prop (depth
         (first-atom candidate)
 candidate)))
(x4 (desired-value 3
     desired-prop (depth
         (first-atom (cdr candidate))
         candidate)))
```

```
$The order of candidates retrieved and
judged is:

        <abc>
        <**digits>
        <**bits>
        <empty>
        <a>
        <abcde>
        mar11-009

$Since no example meets the constraints,
the modification phase is entered with
the following AGENDA:

        (<abcde> 2)
        (mar11-009 2)
        (<**bits> 1)
        (<a> 1)
        (<**digits> 1)
        (<abc> 1)
        (<empty> 0)

$The MODIFIER sets to work on the first
candidate (A B C D E):

-----------------------------------------
constraint = ((typep candidate) list)
actual score = (entry-x1 (lat t))

modify-candidate   ok

-----------------------------------------
constraint = ((length candidate) 5)
actual score = (entry-x2 (5 t))

modify-candidate   ok

-----------------------------------------
constraint = ((depth (first-atom
 candidate) candidate) 2)
actual score = (entry-x3 (1 nil))

  "find-diff"   (increase-depth-by 1)
  "apply-diff"
      reducer =  make-deeper-x
 new-candidate =  ((a) b c d e)

modify-candidate   modified

-----------------------------------------
constraint = ((depth (first-atom (cdr
 candidate)) candidate) 3)
actual score = (entry-x4 (1 nil))

  "find-diff"   (increase-depth-by 2)
  "apply-diff"
      reducer =  make-deeper-x
 new-candidate =  ((a) ((b)) c d e)

modify-candidate   modified

-----------------------------------------
   candidate value =  ((a) ((b)) c d e)
     csc =  4   sf =  t
(entry-x1 (nlist t))
(entry-x2 (5 t))
(entry-x3 (2 t))
(entry-x4 (3 t))
```

$The modification is successful and the new example is added to the Examples-space.

("created new frame for example " mar11-011 ((a) ((b)) c d e)) "success!!"

The Examples-space after the successful solution of Problems 2 and 3 is shown in Figure 4.

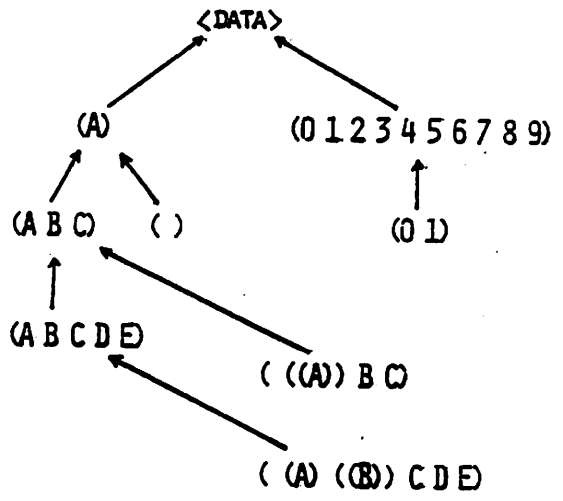

FIG 4

7. CONCLUSIONS

In this paper we have described a computer system that models Constrained Example Generation ("CEG") in domains from computer science and mathematics. We described how the CEG system generates examples of data in LISP.

We are currently using the system to explore issues such as

1. the effect of the initial contents of Example-space and the sequence of solved problems on the evolution of Examples-space;

2. the effect of alternative orderings on the retrieval and modification processes;

3. the effect of interacting constraints, e.g., impossible constraints.

We also plan to use our system to study machine learning by the incorporation of adaptive techniques, e.g., by keeping track of the performance of various orderings and techniques and choosing the ones that perform best. Such extensions of our system will enable it to "learn" from its own experience.

References

Collins, A. and A. Stevens (1979) Goals and Strategies of Effective Teachers Bolt Beranek and Newman, Inc., Cambridge, Mass.

Friedman, D. (1974) The Little LISPer, Science Research Associates, Menlo Park, Calif.

Lakatos, I. (1963) Proofs and Refutations, British Journal for the Philosophy of Science, Vol. 19, May 1963. Also published by Cambridge University Press, London, 1976.

Lenat, D.B. (1976) An Artificial Intelligence Approach to Discovery in Mathematics as Heuristic Search, Stanford Univ. Artificial Intelligence Memo 286.

Polya, G. (1968) Mathematics and Plausible Reasoning, Volumes I and II, Second Edition, Princeton Univ. Press, N.J.

Rissland (Michener), E. (1978a) Understanding Understanding Mathematics, Cognitive Science, Vol. 2, No. 4.

Rissland (Michener), E. (1978b) The Structure of Mathematical Knowledge, Technical Report No. 472, M.I.T Artificial Intelligence Lab, Cambridge.

Rissland, E. (1979) Protocols of Example Generation, internal report, M.I.T., Cambridge.

Soloway, E. (1978) "Learning = Interpretation + Generalization"; A Case Study in Knowledge-Directed Learning. Univ. of Massachusetts, COINS Technical Report 78-13, Amherst.

Winston, P. (1975) Learning Structural Descriptions from Examples, in The Psychology of Computer Vision, P. Winston (Ed.), McGraw-Hill, New York.

Woolf, B., and Soloway, E. (1980) Analysis of Student Protocals: Misconceptions in Understanding Programming in LISP, in preparation.

Acknowledgments

Special thanks to my colleagues Elliot M. Soloway, for his invaluable assistance in bringing up the CEG system, and Oliver G. Selfridge, for many useful discussions and critiques.