Edwina Rissland and Elliot Soloway

TR 80-06

GENERATING EXAMPLES IN LISP: DATA AND PROGRAMS

Extended Abstract

This paper addresses the process of generating examples which meet a set of specified properties or contraints; we call this activity CONSTRAINED EXAMPLE GENERATION or CEG. Herein, we outline a model for this process and describe an implementation of this model which generates examples of data in LISP and simple recursive programs in LISP.

The CEG process can be deomposed into three phases:

(1) SEARCH for and (possibly) RETRIEVE 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) MODIFY an existing example judged to be close to fulfilling the desiderata;

(3) CONSTRUCT an example from elementary or general knowledge, such as definitions, more elementary examples, and general model examples.

The computer implementation of this model consists of: (1) an EXECUTIVE, which directs the overall flow of control; (2) a RETRIEVER, which retrieves examples from the knowledge base; (3) a MODIFIER which modifies existing examples; (4) a CONSTRUCTOR which instantiates general model examples and templates; (5) a JUDGE which evaluates a candidate example's satisfaction of the desiderata; (6) an AGENDA-KEEPER, which sets up an ordered agenda of examples to be worked on.

Briefly, if the simple RETRIEVAL process finds no example in the current knowledge base which satisfies the constraints, the system enters the MODIFICATION phase. Here, an example is massaged to make it satisfy the constraints. The set of examples which are candidates for modification are ordered by their closeness to the goal example. If no example can be modified, the CONSTRUCTION phase is entered, which attempts to instantiate a model example to fit the constraints.

We present several example problems run on our system. The same system architecture is used to generate both data examples and recursive program examples; only the specific example massaging techniques are different.

We conclude with a discussion of the uses of this system, e.g., in an intelligent computer-assisted instruction system for LISP tutoring; as a vehicle to study the theoretical and computational issues (search, constraint interactions, etc.) in example generation. We also suggest ways to incorporate adaptive hill-climbing techniques into the CEG process; in this way the system could learn from experience and improve its performance over time.

GENERATING EXAMPLES IN LISP: DATA AND PROGRAMS

In this paper we describe an architecture for a system that solves problems of generating examples -- data and simple programs -- that have specified constraints, which we call the process of CONSTRAINED EXAMPLE GENERATION ("CEG"). The CEG system consists of three major parts: retrieval, modification, and construction. This computational model for the CEG process has been implemented on a VAX and currently solves example generation problems in LISP of two types: (1) generation of data (i.e., atoms and lists) having certain attributes concerning depth, length, ordering, grouping; and (2) generation of simple recursive programs. The data examples are generated through all three phases of the CEG model; the programs are generated through instantiation of "templates".

SECTION 1: Introduction

Examples -- specific test cases and code -- are important in computer science not only because they play a central role in the writing and debugging of successful programs but also because they are critical to reasoning and understanding in general.

Having a rich well-organized stock of examples is intimately related to understanding [Polya 1973; Michener 1978a]. Examples are important to developing ideas [Lakatos 1963; Lenat 1976], learning concepts [Winston 1975; Soloway 1978] and directing the reasoning process [Polya 1968; Bledsoe 1977]. The ability to concoct examples often distinguishes the expert from the novice. Thus examples are important to both computers and humans.

Automating example generation has immediate applications to both ICAI and automatic theorem proving. ICAI systems need examples to illustrate points to its students. Automatic theorem provers need examples to prune away bad paths of reasoning.

These needs cannot be totally met by a static storehouse of canned examples: no system (or system-designer) can foresee all the examples it will potentially need; further, it would not be efficient to store every conceivable variation of an example. Thus, we have chosen to study the <u>generation</u> of examples.

Having a generation system also provides us with a "laboratory" to study the generation process itself, with the goal of explicating the process in both humans and machines. Such a system will be instrumental in studying the interactions of constraints and design-processes.

For ICAI work, our CEG system has the added bonus of being able to be used to evaluate (and correct) student-generated examples. The same mechanisms that the system uses to judge (and modify) its own examples can be applied to student examples. This can lead to the trapping of students bugs, modelling of the student, and suggesting remedial tutoring episodes.

SECTION 2: The CEG Architecture

A Model for CEG

We are using a model of CEG that is based upon analyses of protocols [Rissland 1979; Woolf and Soloway 1980]. 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 to fulfilling the desiderata;

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

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.

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 4. This model has also been used to simulate the solution of CEG problems in mathematics.

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 list 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 1 shows the general architecture of our system.

The Knowledge

The components use a common knowledge base which consists of two parts: (i) a "permanent" knowledge base of "Representation-spaces" [Michener 1978a, 1978b], and (ii) "temporary" knowledge generated during the solution of a CEG problem.

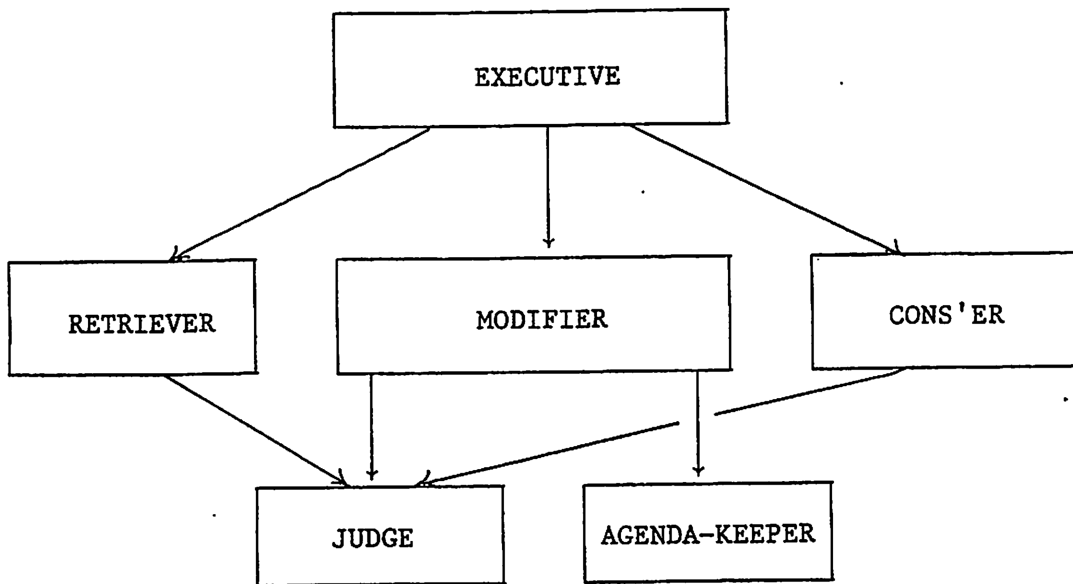

Figure 1

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 2. 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".

The Component Processes

The system consists of several interacting components. Briefly, the components and their roles are:

(1) EXECUTIVE - directs the overall flow of control;

(2) RETRIEVER - retrieves examples from the knowledge base;

(3) MODIFIER - modifies examples;

(4) CONS'ER - constructs examples by instantiation of model examples;

(5) JUDGE - evaluates an example's satisfaction of the constraints;

(6) AGENDA-KEEPER - sets up agenda of examples to be worked on;

Examples-space

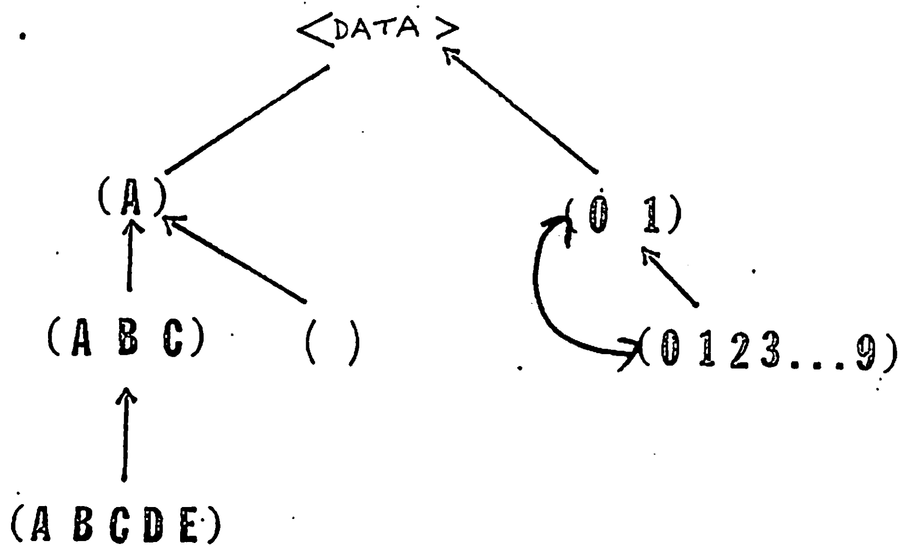

Figure 2.

SECTION 3: 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 CONSTRAINTS-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: (TYPEP 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" [Michener 1978b], position in the Examples-graph, and recency of creation.

In the problems described in Section 4, 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

and

left-positioned before
right-positioned examples

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 2 is thus:

(A B C)

(0 1)

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

( )

(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.

(3) The JUDGE evaluates a CANDIDATE example by cycling through all of the DESIRED-PROPERTY/DESIRED-VALUE pairs on the CONSTRAINTS-LIST and comparing them with the actual properties of the CANDIDATE and recording the results of the comparison in the CANDIDATE's representation frame. 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 a DESIRED-PROPERTY; T is entered if the ACTUAL-VALUE equals the DESIRED-VALUE, and NIL if not.

For instance, the BOX-SCORE for the example "(A)" as a candidate solution to Problem 1 would be:

((LIST T) (1 NIL) (1 T))

The CSC for this example would equal 2.

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

((LIST 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.

(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. In a GPS fashion [Newell, Shaw and Simon 1959], 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 in a difference-reducing table, it 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 thus +2. The difference-reducer finds the modification technique MAKE-LONGER by looking for modification techniques affecting the LENGTH attribute of a list and reducing the DIFFERENCE 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). The MODIFIER thus would make the current candidate longer by adding two elements to the list. (The MAKE-LONGER routine pads a list by adding literal atoms to the end of the list.)

Currently, the system continues to modify the candidate example in its attempt to satisfy the specific constraints. However, the CSC could stay the same or even go done after any specific modification. That is, the constraints can interact and one modification routine could destroy the work of another modification routine, or destroy a property satisfied by the original data. For example, 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, the LENGTH is 2. If the DESIRED-VALUE for LENGTH was 3, we have created a problem!

Thus, in the next version of this system, we shall re-judge an example after <u>each</u> modification. If the CSC is improved, the MODIFIER should continue modifying this example further by going through another difference-analysis, difference-reduction, judgement cycle. If the CSC goes down, the MODIFIER might abandons its attempt to bring the example into line. That is, the MODIFIER should return to the AGENDA and compare the current candidate's CSC with the top candidate in the AGENDA; the one with the highest CSC should be chosen for subsequent modification. In this way, the MODIFIER would be engaging in a form of hill-climbing.

When there is more than one unsatisfied constraint, the MODIFIER orders its modifications according to the order specified by the

EXECUTIVE. In the sample problems in this paper, the modification order (w) is to apply modification techniques that affect the CANDIDATE's attributes of:

TYPE before
LENGTH before
DEPTH before
GROUPING

(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. This latter case is particularly important to the generation of programs from templates.

SECTION 4: Generation of Data Examples

Problem 2

[NOTE: the text in this section (and in Section 5, Sample Output) is actually computer output generated by our CEG system; however, explanatory text has been added, and some output modified to improve readability.]

The second CEG problem asks for a list of length 3 whose first atom has a depth of 2. The constraint list is:

```
(x1 (desired-value list desired-prop (typep candidate)))
(x2 (desired-value 3 desired-prop (length candidate)))
(x3 (desired-value 2 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 examples in the data base, and thus the modification phase is entered.

The AGENDA of candidates for modification is as follows (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):

```
current candidate = <abc>   "value = "  (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 1)
  "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:

```
   candidate value =  ((a) b c)
     csc =  3   sf =  t
(entry-x1 (nlist t))
(entry-x2 (3 t))
(entry-x3 (2 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> *does not reflect structure of E-sp*
<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 " mar11-010 ((a) ((b)) c d e)) "success!!"

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

Examples-space

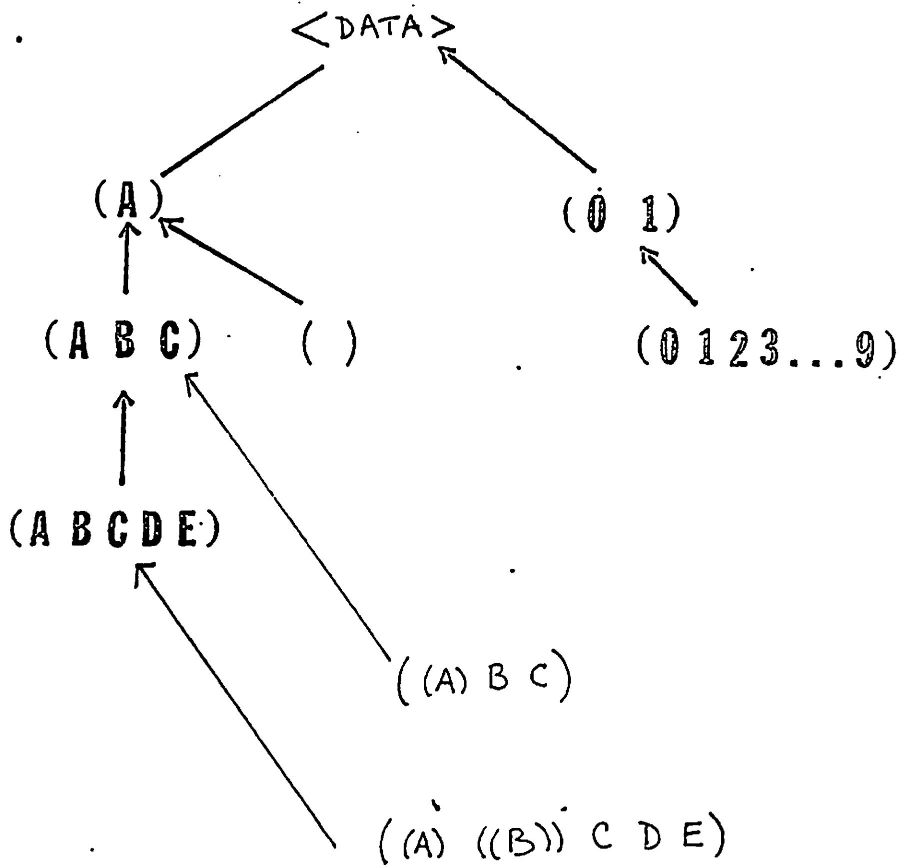

Figure 3.

.nhy

SECTION 5: Generation of Program Examples

In order to apply the CEG model to a particular task area, one needs to identify a set of descriptors, or constraints, for examples in that task area. Above, we have identified several which describe simple data examples: TYPE, LENGTH, DEPTH, etc. The set of descriptors we use in our initial implementation of the CEG process for programs, are based on a model of the underlying structural and functional similarities of simple LISP programs [Soloway and Woolf, 1980].

In particular, we can divide such problems and programs into three classes: CLASSES, BUILDERS, and SELECTORS. A PREDICATE returns True or False (NIL); a BUILDER returns the elements of a list which did not meet the test criteria plus some operation on those that did; and a SELECTOR returns only those elements which did satisfy the test criteria.

In addition, within each class, one can identify "OR" problems and "AND" problems. For example, MEMBER is an OR-PREDICATE, while IS-LIST-OF-ATOMS is an AND-PREDICATE; REMOVE-MEMBER could be construed as an OR-BUILDER, since it deletes only the first occurrence of a element from the list, while REMOVE-ALL-MEMBERS might be called an AND-BUILDER since it removes all occurrences of an element from a list.

The above functional similarity among the problems is nicely mirrored by the structural similarity in the programs which solve such problems. In the predicate case, for example, one can abstract a template from the code for an OR-PREDICATE and an AND-PREDICATE, i.e.,

```
( IS-LIST-OF-ATOMS (LST)
     (COND ((NULL LST) T)
           ((LISTP (CAR LST)) NIL)
           (T  (IS-LIST-OF-ATOMS
                  (CDR LST)]
```

```
( MEMBER (LST ARG)
     (COND ((NULL LST) NIL)
           ((EQ (CAR LST) ARG) T)
           (T  (MEMBER
                  (CDR LST) ARG]
```

```
( PREDICATE-TEMPLATE (CDRING-VARIABLE ARG1 ARG2)
     (COND ((NULL CDRING-VARIABLE)  <T, NIL>)
           ((TEST (CAR CDRING-VARIABLE) [ARG1] [ARG2])  <NIL, T>)
           (T  (PREDICATE-TEMPLATE
                  (CDRING-VARIABLE) ARG1 ARG2]
```

A similar analysis can be given for BUILDERS and SELECTORS; also, extensions can be made to handle nested lists rather than single lists-of-atoms [Soloway and Woolf, 1980]. For this paper, we shall concentrate on programs in the PREDICATE class.

Based on the above discussion, the descriptors we have used to specify the constraints in this part of the system are:

descriptor	possible-values
FUNCTION-TYPE	PREDICATE, BUILDER, SELECTOR
QUANTIFIER	AND, OR
TEST-TYPE	legal LISP predicate
TEST-ELEMENTS	legal arguments to the test-type

For example, the specification of MEMBER would be:

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

CONSTRAINT-2    DESIRED-PROP: (QUANT  X)
                DESIRED-VALUE: OR

CONSTRAINT-3    DESIRED-PROP: (TEST-TYPE X)
                DESIRED-VALUE: EQ

CONSTRAINT-4    DESIRED-PROP: (TEST-ELEMENTS X)
                DESIRED-VALUE: ( (CAR LST) ARG )
```

Before going into a detailed discussion of our particular choice of descriptors, we shall first present two sample runs of the CEG system.

Sample Output

In the first example, we shall assume that our knowledge base has in it only a description of the PREDICATE-TEMPLATE which belongs to the "model" class of examples. Thus, in creating a specific program (e.g., MEMBER), we need to go into the CONSTRUCTION phase of the CEG processes in order to instantiate the template.

Above we listed the constraints which specify MEMBER.

The RETRIEVAL process finds that the only example in the knowledge base is a MODEL example.

```
   candidate name =  PREDICATE-TEMPLATE   candidate-value =
( PREDICATE-TEMPLATE
          (lst arg)
    (cond ((null cdring-variable) <t-or-nil>)
          ((test (car cdring-variable) arg1 arg2) <nil-or-t>)
          (t
           (PREDICATE-TEMPLATE    (cdr cdring-variable)
                                  arg1
                                  arg2))))
     csc =  1   sf =  nil
(entry-i1 (predicate t))
(entry-i2 (template nil))
```

```
(entry-i3 (test nil))
(entry-i4 (((car cdring-variable) arg1 arg2) nil))

"failed"
```

Since this example is by nature general, it must fail the RETRIEVAL process; now, the CONSTRUCTION phase can begin.

```
constraint = ((fun-type candidate) predicate)
actual score = (entry-i1 (predicate t))

modify-candidate   ok
```

**comment: the example already met this constraint; thus no modification is necessary.

```
constraint = ((quant candidate) or)
actual score = (entry-i2 (template nil))

  "find-diff"   (make-or-predicate-from-template-predicate)
  "apply-diff"
      reducer =  make-or-predicate-from-template-predicate
 new-candidate =
( member  (lst arg)
    (cond ((null lst) nil)
          ((test (car cdring-variable) arg1 arg2) t)
          (t ( member  (cdr lst) arg))))

modify-candidate   modified
```

**comment: the 'OR' semantics are instantiated.

```
constraint = ((test-type candidate) eq)
actual score = (entry-i3 (test nil))

  "find-diff"   (make-test-type)
  "apply-diff"
      reducer =  make-test-type   new-candidate =
( member  (lst arg)
    (cond ((null lst) nil)
          ((eq (car cdring-variable) arg1 arg2) t)
          (t ( member  (cdr lst) arg))))

modify-candidate   modified
```

**comment: EQ is substituted for test.

```
constraint = ((test-elements candidate) ((car lst) arg))
actual score = (entry-i4 (((car cdring-variable) arg arg2) nil))

  "find-diff"   (make-test-elements)
  "apply-diff"
      reducer =  make-test-elements   new-candidate =
```

```
( member  (lst arg)
    (cond ((null lst) nil)
          ((eq (car lst) arg) t)
          (t ( member  (cdr lst) arg))))
```

modify-candidate   modified

**comment: the correct arguments to EQ are instantiated.

```
   candidate value =
( member  (lst arg)
    (cond ((null lst) nil)
          ((eq (car lst) arg) t)
          (t ( member  (cdr lst) arg))))
     csc =  4   sf =  t
(entry-i1 (predicate t))
(entry-i2 (or t))
(entry-i3 (eq t))
(entry-i4 (((car lst) arg) t))
("created new frame for example " mar12-009   MEMBER )

"success!!"
```

A new example, named 'mar12-009' is added to Examples-space; it can be used for subsequent example modification.

Now, assume that IS-LIST-OF-ATOMS, an AND-PREDICATE, has been produced through modification of MEMBER, an OR-PREDICATE. Let us follow an example in which the system is asked to produce a function called ALPHA-COMPARE, which returns true if the given character is greater than any other character in the given list.

Below we list the constraints which specify ALPHA-COMPARE.

```
(i1 (desired-value predicate desired-prop (fun-type candidate)))

(i2 (desired-value and desired-prop (quant candidate)))

(i3 (desired-value lex-gte desired-prop (test-type candidate)))

(i4 (desired-prop (test-elements candidate) desired-value (character
 (car lst))))
```

As usual, the RETRIEVAL process attempts to find an example in the current Examples-space which satisfy the above constraints. The candidates to be examined are:

```
"candidates are:"
mar12-009
mar12-010

   candidate name =  mar12-009   candidate-value =
( member  (lst arg)
    (cond ((null lst) nil)
          ((eq (car lst) arg) t)
          (t ( member  (cdr lst) arg))))
     csc =  1   sf =  nil
(entry-i1 (predicate t))
(entry-i2 (or nil))
(entry-i3 (eq nil))
(entry-i4 (((car lst) arg) nil))
"failed"


   candidate name =  mar12-010   candidate-value =
( is-list-of-atoms  (lst)
              (cond ((null lst) t)
                    ((listp (car lst)) nil)
                    (t ( is-list-of-atoms  (cdr lst)))))
     csc =  2   sf =  nil
(entry-i1 (predicate t))
(entry-i2 (and t))
(entry-i3 (listp nil))
(entry-i4 (((car lst)) nil))
"failed"
```

The MODIFICATION process can now begin. The list of candidate examples below are ordered on the basis of their CSC value; the MODIFICATION process will take mar12-010 (is-list-of-atoms) from the AGENDA to work on first.

```
"candidates are:"
(mar12-010 2)
(mar12-009 1)

current candidate = mar12-010
value =
( is-list-of-atoms  (lst)
              (cond ((null lst) t)
                    ((listp (car lst)) nil)
                    (t ( is-list-of-atoms  (cdr lst)))))
----------------------------------------------
constraint = ((fun-type candidate) predicate)
actual score = (entry-i1 (predicate t))

modify-candiate   ok

-----------------------------------------------
```

```
------------------------------------------
constraint = ((quant candidate) and)
actual score = (entry-i2 (and t))

modify-candiate   ok

------------------------------------------
constraint = ((test-type candidate) lex-gte)
actual score = (entry-i3 (listp nil))

  "find-diff"   (make-test-type)
  "apply-diff"
      reducer =  make-test-type   new-candidate =
( alpha-compare  (character lst)
           (cond ((null lst) t)
                 ((lex-gte (car lst)) nil)
                 (t ( alpha-compare  (cdr lst) lst))))

modify-candidate   modified

**comment:  the new test, lex-gte, replaces listp.
------------------------------------------
constraint = ((test-elements candidate) (character (car lst)))
actual score = (entry-i4 (((car lst)) nil))

  "find-diff"   (make-test-elements)
  "apply-diff"
      reducer =  make-test-elements   new-candidate =
( alpha-compare  (character lst)
           (cond ((null lst) t)
                 ((lex-gte character (car lst)) nil)
                 (t ( alpha-compare  character (cdr lst) ))))

modify-candiate   modified

------------------------------------------
   candidate value =
( alpha-compare  (character lst)
           (cond ((null lst) t)
                 ((lex-gte character (car lst)) nil)
                 (t ( alpha-compare  (cdr lst) lst))))
     csc =  4   sf =  t
(entry-i1 (predicate t))
(entry-i2 (and t))
(entry-i3 (lex-gte t))
(entry-i4 ((character (car lst)) t))
("created new frame for example " mar12-011   alpha-compare )

"success!!"
```

The new program, ALPHA-COMPARE, was produced as a modification of an existing program, IS-LIST-OF-ATOMS. The contents of Examples-space now is:

PREDICATE-TEMPLATE <---constructed-from--- MEMBER

MEMBER <---constructed-from--- IS-LIST-OF-ATOMS

IS-LIST-OF-ATOMS <---constructed-from--- ALPHA-COMPARE

Issues in CEG Program Generation

The CEG system in this domain works exactly like the system in the data domain; the basic GPS architecture is used to massage the candidate example into the desired goal. As is the case in the data domain, the assumption of noninteracting subgoals will be violated when we try and extend the system for the generation of programs of the BUILDER and SELECTOR types.

Also, the set of descriptors will need to be expanded. For example, a BUILDER takes some action when it comes across an element of the desired characteristics, e.g.,

```
( REMOVE-MEMBER  (ARG LST)
      (COND ((NULL LST)  () )
            ((EQ (CAR LST) ARG)  (CDR LST) )
            (T  (REMOVE-MEMBER ARG (CDR LST]
```

More generally, the "level" of the descriptors is uneven. Function-type and Quantifier are more abstract descriptions, while Test-type, Test-elements, etc., are at the LISP level. One might want a higher level specification of these attributes, with the machine making the inferences concerning the specifics of the LISP realization.

How far can the underlying "template model" be pushed? Hardy [1975] used templates to generate a wide class of programs for examples. Burge [1976] uses template-like structures in describing an even wider class of problems. Typically, however, the program synthesis approach has been more akin to "construction from principles" in which truth-preserving transformations successively modify the program specification [Barstow, 1977; Manna and Waldinger, 1977; Ulrich and Moll, 1979]. We have found that the extended taxonomy can account for the majority of programs in the introductory LISP text, The Little LISPer [Friedman, 1974]. Thus, we are optimistic that this model will be sufficient to build a system which can produce a rich set of programs.

SECTION 6: Conclusions and Future Work

In this paper we have described a computer system based on a model of Constrained Example Generation. The system is able to generate examples in LISP of data and programs using the same architecture. We are currently actively using the system to explore issues such as:

1. the effect of the initial contents of Examples-space and the sequence of solved problems on the growth of Examples-space
2. the effect of alternative example ordering functions on the RETRIEVAL and MODIFICATION processes
3. the effect of interacting constraints, e.g., impossible constraints.

We plan to use our CEG system in an intellegent computer-assisted instruction tutoring environment. We are currently building a tutor to teach students about programming languages such as LISP and BASIC. In this context, CEG will serve the tutor in two ways: (1) it will generate examples to the specifications set by the tutor; and (2) it will evaluate student generated examples for the tutor. The same JUDGE used by CEG, can be used to evaluate a student's example and help track down his misconceptions and bugs through analyses of differences between what the tutor requested and he generated. Of course, for such interaction, the tutor must be able to generate specifications for examples.

In the future, we also plan to incorporate adaptation into the system. For example, the system can keep track of the performance of the various example ordering functions and choose the one that has the best performance. Also, we plan to apply hill-climbing techniques to the modifying processes themselves. That is, since there are alternative ways to massage and modify an example, those routines which lead to the most succeses should eventually be selected first. Adaptation on the modification techniques will be particularly important if the system is to be able to improve its performance, and thus "learn" from its own experience. This capability will require the addition of a NOTICER, a STATISTICIAN, and an EVALUATOR.

The current implementation is only a "first-pass" and does not capture the richness of the CEG model. Nonetheless, we feel that it has demonstrated the utility of this model, and, we feel that subsequent implementations and additional task domains will permit us to continue exploring the process of example generation.

Bibliography

Barstow, D. (1977) Automatic Construction of Algorithms and Data Structures Using a Knowledge Based of Programming Rules, Stanford Univ. Artificial Intelligence Memo 308.

Bledsoe, W.W. (1977) A Maximal Method for Set Variables in Automatic Theorem Proving, The Univ. of Texas at Austin Math. Dept. Memo ATP-33.

Burge, W. (1976) Recursive Programming Techniques, Addison Wesley, Reading, Mass.

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

Hardy, S. (1975) "Synthesis of LISP Functions from Examples," Fourth International Joint Conference on Artificial Intelligence, Tbilisi, U.S.S.R.

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.

Manna, Z., and Waldinger, R. (1977) Synthesis: Dreams --> Programs, SRI International, A.I., Tech. Note 156.

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.

Newell, A., Shaw, J., and Simon, H. (1959) Report on a General Problem-Solving Program. Proc. of the International Conference on Information Processing. UNESCO House, Paris.

Polya, G. (1973) How To Solve It, Second Edition, Princeton Univ. Press, N.J.

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

Soloway, E. (1978) "Learning = Interpretation + Generalization"; A Case Study in Knowledge-Directed Learning. Univ. of

Massachusetts, COINS Technical Report 78-13, Amherst.

Soloway, E., and Woolf, B. (1980) Problems, Plans and Programs, Proc. of the ACM Eleventh SIGCSE Technical Symposium, Kansas City.

Ulrich, J., and Moll, R. (1979) Program Synthesis: A Transformational Approach, Proc. of Eighth Texas Conference on Computer Systems, Dallas.

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.