STANFORD ARTIFICIAL INTELLIGENCE PROJECT
MEMO AI - 80

HEURISTIC DENDRAL: A FAMILY OF LISP PROGRAMS

by Georgia Sutherland

ABSTRACT: The Heuristic Dendral program for generating explanatory hypotheses in organic chemistry is described as an application of the programming language LISP. The description emphasizes the non-chemical aspects of the program, particularly the "topologist" which generates all tree graphs of a collection of nodes.

The research reported here was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense (SD-183).

HEURISTIC DENDRAL: A FAMILY OF LISP PROGRAMS

by Georgia Sutherland

1. Introduction - The Dendral Algorithm

The motivation for this paper is to describe a successful application of the programming language LISP. The area of application combines both computer science (artificial intelligence in particular) and organic chemistry (mass spectrometry in particular). The collection of computer programs which has resulted has come to be known as "Heuristic Dendral". Although these programs were developed for an application in the field of chemistry, they have remainded fairly general. The discussions which follow ignore the chemistry aspects as much as possible.

The central component of the program is the "Dendritic Algorithm" (ref. 1), usually referred to as "Dendral". This algorithm specifies a canonical notation for tree-like graphs. The notation implies a hierarchy of structures; and consequently provides a generating sequence for all tree-graphs composed of different numbers of various types of nodes.

The Dendral tree-graphs are composed of two elements; nodes (operands) and links (operators). Node types are represented by literal names (usually letters: A, B, C, . . .) and link types are represented by numbers (1, 2, 3, . . .).* The value of a link is proportional to the number used to represent it.

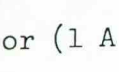
A structure in Dendral notation is a sequence of these link-operators and node-operands. The links are binary operators, each

---

*In true Dendral notation, dots (i.e., ".", ":", and ":") are used instead of numbers to represent links.

connecting two nodes. But nodes in general may be attached to other nodes by any number of links. Usually different node names represent node types which differ with respect to the maximum number of links which can be attached to them.

A tree-graph is represented by a left-to-right sequence of numbers and letters (names). The simplest tree-graph is a single node, the next simplest is a graph with two nodes and one link. A rooted tree-graph is one in which one of the links has a free end; thus the simplest rooted tree-graph has one link and one node.

Structures in Dendral notation* are built of rooted tree-graphs enclosed in parentheses. The simplest is one of the form (1 A). A slightly less trivial graph is (1 A (2 B)), which is drawn as -A=B . Another simple graph is (1 A (2 B (3 C))) which is drawn as -A=B≡C ; or (1 A (2 B) (3 C)) which is drawn as  . Thus, the definition for a tree-graph is a recursive list:

		first		second		n-th
(link,	node,	attached	,	attached	, . . . ,	attached)
		tree-graph		tree-graph		tree-graph

These structures can be written without parentheses by moving the roots of each attached tree-graph, thus forming non-rooted subgraphs:

					first		n-th
root,	node,	first,	. . . ,	n-th,	non-rooted,	. . . ,	non-rooted
		link		link	attached		attached
					tree-graph		tree-graph

*This type notation is also called "list notation".

In this form our preceeding examples would be: 1 A, 1 A 2 B, 1 A 2 B 3 C, and 1 A 2 3 B C .

The Dendral algorithm goes even further in specifying the notational description of a tree-graph by defining the relative order (or "weight") of two graphs and by requiring that at every branch point in the tree the attached subgraphs must be listed in increasing order (when read from left to right). For example, both (1 A (2 B) (3 C)) and (1 A (3 C) (2 B)) represent the same geometrical configuration of nodes and links, but only one of the representations, the first one, is the "canonical form".

The rules for comparing canonical graphs are given in terms of three criteria: the weight of a node, the number of attachments of a node, and the value of the link leading into the node. The weight of a node ia arbitrary, but relative order of node weights remains constant for any canonical system. A frequent rule for assigning weights to nodes is by alphabetical order.

The following seven rules establish the relative weight of any two tree-graphs which are written in canonical form.

1. Compare the number of nodes in each graph. The graph with the most nodes is the higher order (heavier) graph.

2. Compare the composition of the graphs. The graph with the heavier nodes is the heavier graph. For example, if the weight of a node is related to its alphabetical order, then a graph with two C nodes is heavier than a graph with one B node and one C node.

3. Count the unsaturations in the graph. Unsaturations are present in any link which is greater than 1. (A link with value 3 contains 2 unsaturations.) The graph with the most unsaturations is the heavier graph.

4. Look at the first node (called the "apical" node) of the two graphs. Count the number of subgraphs connected to each apical node. The apical node with the largest number of connections determines which graph is heavier.

5. Compare the weight of the two apical nodes. The graph with the heavier apical node is the heavier graph.

6. Compare the links leading to the apical nodes (called "afferent" links). The graph with the larger afferent link is the heavier graph.

7. Arrange the subgraphs which are connected to each apical node in order of increasing weight. Then, compare the subgraphs in pairwise fashion, one from each of the main graphs. The heavier graph is the one which has the first heavier subgraph attached to the apical node.

The rules above apply to rooted graphs. A non-rooted graph may be made to appear rooted by giving it an afferent link of NIL or zero. An additional rule for obtaining the canonical form of non-rooted graphs is that the center of the graph must be that node or link which most nearly divides the node count of the graph into equal portions. More specifically, no rooted subgraph attached to the center can have more than half of the total nodes of the graph. (It has been proven that such a center is unique (ref. 1).) In a two-node, non-rooted graph, for example, the center is the single link connecting the two nodes. It can be imagined, however, that this is a three-node graph of the form A-NIL-B . Then the rooted tree description for this graph would be (NIL NIL (1 A) (1 B)). The first NIL specifies the (non-existent) root-link, the second NIL specifies the central node.

The importance of the Dendral algorithm for assigning unique notation to any tree-graph is not so much in the actual rules listed above, but in the fact that these rules can be systematically applied

to any tree structure, and in the fact that the rules are completely unambiguous, both in their application and in their result. It is not necessary to understand or memorize the rules and notation in order to appreciate the rest of this discussion; it is only necessary to appreciate their effect.

The Dendral notation serves two purposes: first, to specify a canonical form for any tree-graph; and, second, to provide a way to compare two tree-graphs, thus implying that any list of tree-graphs can be placed in an ordered sequence.

Implicit in the rules for Dendral notation is a procedure for generating the "lowest", or first, member of a sequence of tree-graphs and, consequently, obtaining the "successor" or next-higher graph for any particular tree-graph. The first procedure would be carried out by following these steps (assuming the number and type of nodes is specified):

1. select the lowest node in the group,
2. select the lowest afferent link,
3. make one rooted subgraph from the remaining nodes and links,
4. attach the subgraph to the first node.

This indicates that the "lowest" rooted tree-graph is a single chain of nodes and links, with the nodes being arranged in ascending order.

The process for finding the successor to a tree-graph is similar, but slightly more complicated.

1. If the graph is a single node and link, then nothing can be done to increase it.

2. Attempt to increase one of the subgraphs attached to the top node.

3. Attempt to increase the afferent link.

4. Attempt to find a heavier apical node.

5. Attempt to increase the number of attached subgraphs.

These procedures form the Dendral Algorithm for generating all tree-graphs which can be made from a set of nodes.

Once the algorithm had been invented, it was natural to inquire about automating it with a computer. The main point of this paper is to indicate how the automation was accomplished, and to indicate its usefulness and applicability to a certain area of experimental science.

2. A LISP Program for the Dendral Algorithm

The first problem encountered in mechanizing Dendral* was to find a suitable computer language in which to write the algorithm. The first version was written in Algol, but awkwardness of recursion and the non-numeric flavor of the data soon made the choice of a list processing language obvious. The main list processing language at Stanford is LISP, and it did not take long to determine that LISP was appropriate for representing Dendral tree structures and mechanizing the generating algorithm.

The first version of the program had but one job; that was to list all the tree-graphs which could be made from a given collection of nodes; and to do so in an exhaustive, non-redundant way. The LISP code which produced this output is quite straightforward. All types of nodes which the program recognizes are put on a global list called "Orderlist" in which the nodes are listed in order of increasing weight. Thus the order of the nodes on the list determines the Dendral order of the nodes for purposes of building canonical tree-graphs. Each type of node is made into a LISP atom, with a property specifying the number of connections a node of this type can have in the completed structure.**

The main LISP function is a highly recursive function called "Genrad", whose objective is to generate a single rooted tree from a specified collection of nodes. The allowable ranges for the afferent

---

*We use the term "Dendral" to include both the notation and also the generating algorithm itself.

**The terms "structure", "graph", "tree-graph", "tree", etc., are used interchangeably in this paper.

link (root) are specified as input arguments to Genrad. Genrad determines the apical node, the afferent link, and the number of branches (subgraphs) which are to emanate from the apical node. Then Genrad calls itself to make a structure for each of these branches; and a completed structure is returned as the value of Genrad.

Another LISP function, Genmol, is in charge of selecting the "center" of an unrooted tree and obtaining the branch structures from Genrad. Since the number of nodes in any branch emanating from the top node must not be greater than half of the total number of nodes in the whole graph, this implies that there must be at least two branches emanating from the apical node of an unrooted tree.

The first structure generated by the program is the lowest structure which can be made from the given nodes. This will be the first of the output list. All successive structures are obtained by manipulating the preceeding output structure to obtain the "next higher" canonical structure. The main LISP function which performs this task is Uprad, and it also is a recursive function, similar to Genrad. Uprad is given a rooted tree as input. First, Uprad sees if one of the branches emanating from the apical node can be made higher. If so, it increments this subgraph and returns the new tree as its value. Otherwise, Uprad considers the apical node of the graph and tries to make the leading link greater. If this fails, Uprad tries to make the apical node greater (while resetting the link to its lowest value). If this fails, Uprad tries to increase the number of branches emanating from the apical node (while resetting

both link and node to lowest possible values.). If all these options fail, Uprad returns NIL, indicating that no higher structure is possible. Otherwise Uprad calls Genrad to obtain the lowest substructures to attach to the (possibly revised) top node; and the new composite structure is returned as the value of Uprad.

The success of the Dendral-LISP implementation is due to several factors:

1. the systematic generating sequence implied by the Dendral canons of order,
2. the recursive definition of a graph in the Dendral system,
3. the ease of data representation in LISP lists,
4. subdividing the program into functions which employ both recursion (to trace the structure) and the PROG feature (to consider the individual Dendral canons in order),
5. the ease of specifying different supervisory functions to control the generation of the list of structures,
6. the ease of specifying different output functions (for various types of printing, processing, and filtering).

3. Additions to the Dendral-LISP Program

The first addition to the program was called the "dictionary". The program had been spending much of its time generating the same little graphs over and over again, which it used as building blocks to make the larger structures. The users of the program noticed this* and decided that some structure memory could and should be introduced. The function Genrad should somehow remember that it had seen given input parameters before, and should be able to look up the previous answer in a dictionary. So, this was implemented. The necessary program changes were:

1. every RETURN call in Genrad was changed to a return call through another function called Setdict, i.e., instead of doing (RETURN STR) in Genrad, the call was changed to (RETURN (SETDICT STR));

2. the function Setdict was written. It returned STR as its value; but before doing that it made sure that STR and all other structures which could be made from the same nodes were placed on a list and stored as the value of a certain LISP global variable. The particular variable name was made by taking the list of all nodes and compressing the characters into a LISP atom name. This provided a unique storage place for the particular dictionary entry;

3. in order to make use of the dictionary built by Setdict, every function whose job was to return a structure as its value had to consult the dictionary to see if such structures had been generated previously. If so, a dictionary lookup provided the answer.

The dictionary was not difficult to implement and has proved to be a great time-saver in serious use of the program as a graph

---

*Wouldn't it have been nice if the program could have noticed this itself? For further comments on the "store versus recompute" problem see section 8.

generator. It is even possible to store large dictionaries in auxiliary storage (magnetic tape, disk, etc.) and retrieve the particular entries needed in the course of generating very large structures. The dictionary has an on-off switch which will prevent additional entries from being made. This saves both space and time in one-of-a-kind program runs. Another switch can erase an existing dictionary. These switches are of the nature of global variables. One is referenced by Setdict which does nothing but return its input argument in case the dictionary-generating switch is OFF. The other switch is a list of all names of dictionary entries, which is saved as a global variable and which allows all its entries' lists to be set to NIL if desired.

The second addition to the Dendral-LISP program is called "graph matching". Users of the program wanted to be able to place restrictions on subgraphs in order to eliminate structures they did not like from the output lists.* Implementing this required adding a global list called Badlist which contains a representation of every subgraph which a user does not like. Because this is a global list, and not part of a LISP function, Badlist can be tailored to suit the fancy of every individual user. A function was added to the LISP program which takes a structure as input and returns T if that structure contains any subgraph listed on Badlist. If the structure is free of all "bad" subgraphs, then this function returns NIL. Note that Badlist has the nature of a switch. If no items are on

*In the task area of chemistry, this reflects the fact that not all topologically possible graphs represent chemically stable substances.

Badlist, then no "bad" subgraphs can be found, and every structure will be permitted in the output list.

Inserting this graph matching process into the program could have been accomplished in any one of several ways.

1. Check at output time. If the structure is bad, do not print it.

2. Check each output of Genrad. If the graph contains a bad subgraph, call Uprad and check the new structure. Repeat the process until Uprad returns a good structure. Then return that as the value of Genrad.

3. Filter a dictionary, remove all bad structures, and use that dictionary to generate further structures.

4. Check the partially built graph at every level of recursion. Every new node and link can be checked as they are about to be added to a partial structure. If a bad structure would result, this node-like combination is not used, but the next is chosen, checked, and so forth.

All four of these methods have been used at one time or another. The first is obviously easiest to implement, but is also (obviously) the most inefficient. The second method is pretty good, but unnecessary work is done in generating the branches to attach to an already-bad structure. The third method still requires some check to be made when good subgraphs are combined to make a structure bigger than one in the dictionary.

The fourth method is relatively difficult to implement, but is conceptually the nicest and most efficient. Implementing this method requires that every LISP function which returns a structure as its answer has to first make a check against Badlist. This requires

modifying nearly a dozen functions,* but this would probably be an even worse task in a computer language which does not segment itself into such manageable portions. The effort was felt to have been quite worthwhile, in that it provided maximum reduction of the total structure generating effort.

It is also interesting to note the evolution of the graph matching algorithm used by the program. Since the algorithm is a separate LISP function and only its name appears anywhere in the structure generating program, any graph matching algorithm (even a completely separate program in another language) could be used. The first was an algorithm based on connection tables. The second was one based on lists of properties of the structure. And the third, and current, algorithm is based on comparing structures emanating from the apical node of the tree. It assumes that subgraphs not including the top node have already been checked (this is true because of the nature of the recursion in the program). The first two methods were hopelessly slow. The third method seems to use the maximum amount of information available, and appears to be as efficient as possible. One helpful assumption speeds the program a little in this regard; that is the assumption that any structure obtained from dictionary lookup is free of bad subgraphs.

---

*One of the input arguments of each of these functions is the "previously constructed portion of the graph". One wonders whether it wouldn't be better to save the "current structure" as a LISP global variable, accessible by all functions.

The third addition to the Dendral-LISP program was named "Goodlist". It is the antithesis of Badlist and resulted from a desire to keep "good" subgraphs together, or to make a preliminary specification about part of a graph and force the structure generation process to ignore all structures which do not contain this graph. The form of this feature is that of a pre-processor operating on the information contained in the global variable Goodlist. (If Goodlist is NIL then this option has no effect.) Goodlist contains a list of names of subgraphs. The names themselves are LISP atoms, with property lists containing such pieces of information as structure (corresponding to node type) and a corresponding property telling how many attachments this subgraph can have and where they are located. Normal nodes were usually referred to as "atoms", so these larger nodes are presently referred to as "superatoms". Introducing superatoms to the program was conceptually not too hard because a simple pre-processor was practically the only programming necessary. The pre-processor takes the input (a list of numbers of nodes of different types, all of which are to be used in every output structure) and inspects it to see if any superatom on Goodlist can be made from the nodes. If so, then the nodes used to make the superatom are removed from the input, and a new superatom node is introduced. It is possible to specify the maximum and minimum number of superatoms that can appear in any output structure, and it is also possible to specify the maximum and minimum of each individual type of superatom, if there is more than one entry on Goodlist. Since the superatoms on Goodlist may sometimes use overlapping atoms, there may be several

different combinations of different superatoms which are compatible with the input nodes. All these combinations are employed in generating structures.

The difficulty in implementing the notion of superatoms was in finding and changing all the utility functions in the program that operate on atoms and structures. Many short functions do such operations as counting the number of atoms in a structure, and these all had to be changed to accommodate the new notion of superatoms. Also, the graph matching algorithm mentioned above had to expand superatom names into their internal graphs in order to check whether bad subgraphs had been formed in the process of putting superatoms and regular nodes together.

The Badlist and Goodlist additions to the Dendral-LISP program did a great deal to help the program users direct the result of structure generation. Occasionally this was not enough, however, and some users of the program felt the need to observe the actual structure generation process and to have periodic opportunities to specify the direction of work.

So "dialog" was added, with the aid of time-sharing and interactive LISP. Whenever Genrad is about to proceed to a deeper level of recursion it pauses, prints the current structure, and asks permission to continue. The user types "Y" or "N" in response. If N is the response, Genrad skips that phase of structure generation and proceeds as if NIL were returned from the recursive call that was not actually executed. (It would have been nice for the program user

to be able to indicate his reasons for changing the course of action of the program, but this was never actually programmed.)

Another addition to the Dendral-LISP program is called "spectrum". This provides another way of referring to the outside world for indications of the plausibility of a structure that is being generated. The spectrum itself consists of a list of numbers and a few additional rules about subgraphs which are not formulatable in the Badlist style described above. A formula for generating a single number (measurement) for a structure is specified and embodied in a single LISP function. Every structure then has to have this measurement compatible with the plausibility rules and the list of numbers in the spectrum in order to have it be used in generating any bigger structure.

Implementing this was quite simple. It took advantage of the fact that all structures come out of the function Genrad which always exits through the function Setdict. Setdict now checks each structure for compatibility with the spectrum rules, and it does this by calling a separate checking function which returns T or NIL depending on whether the structure passed or failed the tests. If failure is the result, Uprad is called until a structure does pass the tests.

There are faults with this implementation, however. One is that much information is present and usable when Genrad is entered. For example, the number calculated for the structure can usually be calculated from the number and kinds of nodes which Genrad is told to use in generating the structure. If this number turns out to be implausible, then Genrad should not be allowed to do any work

at all. Fixing this is not difficult, but it interferes with the dictionary generation process described earlier, since the dictionary mode of program operation says to generate structures independent of outside influence such as that implicit in the spectrum.

The spectrum is what we call "real data", and allows the program to turn to the outside world for assistance in the direction of structure generation. Another way of consulting outside sources of information was brought to the program by still another addition, called "partitions", This process consults a list of parameters to calculate the plausibility of different groupings or compositions of nodes. Whenever a tree-graph is about to branch, for example, two or more subtrees will be attached. The question is how to divide the remaining nodes among the sub-trees in the most plausible way.

The partitioning sub-program makes a list of all possible ways of grouping the nodes and calculates a plausibility score for each member of the list. A single LISP function calculates this score, depending on relative numbers of different kinds of nodes. The function refers to a dozen global parameters which a user can vary to indicate his preferences about likely node groupings.

Naturally, the partitioning part of the program is turned on or off by a global switch. The existence of this and all the other global program switches reflects one basic philosophy that has existed during the entire life of the program. In particular, it should be possible to eliminate all options from the Structure

Generator's operation, and allow all topologically possible graphs to be generated from a set of input nodes. This goal has been scrupulously followed, with a great deal of assistance from the LISP feature of global variables and lists stored as the value of these global variables. NIL-ing a few lists (Goodlist, Badlist, Spectrum) or turning OFF some switches (Dictswitch, Dialog, Partitions) accomplishes this goal in a very neat and systematic way.

LISP has also made it convenient for the program to expand in the way it has, from the inside out (with the addition of pre-processors and post-processors) rather than serially, taking advantage of previously written parts of the program and the growing collection of utility functions.

A most recent addition to the program has been a true ring-generator, that is, a processor which uses the facilities of the tree-graph generator in order to generate all multiply-connected graphs that can be made from the list of nodes. Previously it was possible to generate ring-containing graphs by specifying the ring part as a superatom. The new "ring-generator" works under the following philosophy: any ring structure has an underlying tree structure which will be generated in the ordinary course of work by the Structure Generator program. A tree underlying a ring graph is discovered by removing one or more of the links. Therefore, a cyclic structure can be made by adding links to previously generated tree graphs. So, the ring-generator of the program is a combination of pre-processor and post-processor added to the tree structure generator.

The pre-processor alters the list of nodes and links so that fewer links are required, and sets a global variable indicating how many links have been removed. The post-processor intercedes after a tree has been generated but before output has been done. It takes the particular structure which is to be output, adds the required number of links in all possible ways, and then prints all the structures that result from these operations.

The problems encountered in implementing this type of generation of cyclic structures are in avoiding symmetric structures. Several tree graphs underlie a cyclic structure, and so the same cyclic structure may be produced by several different tree outputs. Similarly, certain different combinations of added links may give similar cyclic structures, even though the underlying tree structure (or structures) may be different. This symmetry problem was solved in an interesting way, which seems somewhat alien to LISP but which does take advantage of a LISP feature, namely the ARRAY feature. The arrays that are used are connection tables in which each entry represents the value of the link connecting two nodes.* An arbitrary canonical form is imposed on each array made from a structure, and the arrays of different cyclic structures are compared to insure that a symmetric structure has not previously been generated.

---

*Interestingly enough, to artificial intelligence researchers at least, this way of solving the symmetry problem requires a shift of representation of the data structures; that is, a shift from list notation of graphs to connection tables representing the same graphs.

Adding the ring-generator to the program posed no problems as far as the Structure Generator and its main options were concerned. Superatoms can still be introduced on Goodlist before the Structure Generator is called; Badlist can still be set up to force the elimination of structures containing certain undesired substructures. This latter option currently just checks the tree structures, no check is made after the rings are added. But this is not a serious problem, we will just require another graph match to take place after the rings have been added. This check can concern itself only with substructures containing the added ring-links, so that implementing it in a simple and obvious way is no problem. We are currently investigating how to implement it in the best way, because of the following additional pieces of information which now are present and relative:

1. Would it be better to use a connection-table type of graph matching, since the connection table is already present and since more general types of graphs can be represented this way (the current Badlist can only represent tree-graphs),

2. Some things appear to be bad in rings which are not bad in trees, so an expanded Badlist, if it were able to represent cyclic structures, would have two or three types of entries, one for "bad things in trees", one for "bad things in rings", and one for "bad things in combined structures".

A few of the other options do not pass from trees to rings quite as easily as do the Badlist and Goodlist options. Partitioning, for example, loses its effect when previously separate branches of a tree are connected to form a ring. The spectrum rules, too, becomes invalid when sub-structures are allowed to become multiply-connected. But, historically at least, these are the less frequently used options,

and Goodlist and Badlist are the more general and useful ways of controlling the nature of the generated structures.

The information about whether to generate trees alone or rings alone or both trees and rings (or neither!) is contained in a pair of global switches. All the global switches for the options in structure generation reflect the fact that the users of the early versions of the Structure Generator program were, to some extent, unhappy about the long lists of output they saw. The desire for the program to consult the "real world" for advise on which structures to generate was the impetus for a large expansion of the Dendral-LISP effort. Instead of remaining a solitary LISP program, the Structure Generator was joined by three other LISP programs which together form a family of programs called "Heuristic Dendral". The three other programs are, first, a data pre-processor called the "Preliminary Inference Maker"; second, an output processor called the "Predictor"; and, third, an output evaluator called the "Scoring Function". These programs are described in detail in later sections of this paper.

4. Chemical Mass Spectrometry - An Application for Dendral

The discussions, up to this point, have avoided much mention of chemistry. Actually, the subject of chemistry was always present during the entire development of the Structure Generator program, but the nature of the Dendral Algorithm itself is so general that non-chemical applications can easily be imagined. The parts of the program other than the Structure Generator do not separate themselves from chemistry very easily. But the discussion which follows does not demand that the reader have any chemical background at all.

The particular application for Dendral is an area of organic chemistry known as "mass spectrometry".

Imagine the following situation: an unknown substance is placed in an instrument which causes it to break apart. Some of the fragments are collected and weighed, and the resulting data is a list of weights and a table of the relative frequency of occurrence of each different fragment weight. Such data is called a "mass spectrum". Chemists must decide whether the mass spectrum provides any clues to the structure of the original substance.

This is an application for Dendral, because the structure of the substance must be a graph of the atoms and bonds which form the molecule. The initial attempt at solving the chemists' problem was to use the Structure Generator to compute all possible graphs which could be constructed from a list of atoms whose weights sum to the weight of the unfragmented molecule. (Usually the largest weight associated with a positive amplitude in the

data is the weight of the unfragmented molecule.) The problems with this approach were that too many hypotheses were being generated and that not very much of the spectral data was being used. Each of these observations led to a computer program designed to aid the Structure Generator in its work. One is a program which inspects the list of structures and makes judgments about which is the best hypothesis. The other program inspects the data to uncover patterns and pass suggestions along to the Structure Generator about the nature of the graphs which should be generated.

Both of these programs need to incorporate a great deal of chemists' knowledge about structures and their fragmentation processes. The knowledge base is common to both the programs, and a future program will be written which contains all this knowledge in a general form and provides the two other programs with the facts they need in order to operate. Section 7 discusses this concept of the central store of chemical information. The next two sections discuss the current state of the existing programs.

5. Other Programs Join Dendral - The Preliminary Inference Maker

The buffer between the real world of chemistry and the Structure Generator program is another LISP program called the "Preliminary Inference Maker". In general outline, this program is conceptually simple and efficient. It incorporates a great deal of specialized chemical information, which is gathered together in lists and properties of LISP atoms, easily accessed, changed, or augmented.

In studying the fragmentation of known substances, chemists have accumulated knowledge about which links in a graph are most or least likely to break apart in a mass spectrometer. Certain subgraphs are known to stay together; and these have been given specific names. These subgraphs have characteristic effects on the resulting mass spectra. The absence of a certain characteristic peak in a spectrum may indicate the absence of the associated subgraph in the original structure, while the presence of a set of characteristic peaks in the spectrum is good evidence for the presence of the subgraph in the original structure.

The characteristic subgraphs are called "functional groups", being groups of atoms which are connected to each other in a fixed way and which stay together, acting functionally as a single subgraph or "superatom". The operation of the Preliminary Inference Maker is centered on a list of functional groups which are presently of interest to chemists. Each functional group has the form of a LISP global atom with properties describing both the characteristics it will impart to a spectrum, and also its relation to other functional groups.

The list of names of functional groups is called "Grouplist", and the outline of operation of the program is the following series of steps:

1. Two inputs are given: one is the spectrum, which resembles any statistical histogram, being a bar graph, or list of pairs of numbers; the other is a list of atoms which are assumed to comprise the total molecule.*

2. Consider, one at a time, each name on Grouplist.

3. Determine whether the group's composition is contained within the composition of the molecule. If it is not, then this group is ignored.

4. Determine whether all conditions necessary for the appearance of this functional group are represented in the data. If any necessary condition is not satisfied, then it will be impossible for this subgraph to appear in any final graph. So the name of this functional group is added to the Badlist of the Structure Generator program. When a functional group is rejected in this way, all names of other groups whose structure contains the rejected group as a subgraph are removed from Grouplist.

5. Determine whether all conditions sufficient for the appearance of this functional group are represented in the data. If all sufficient conditions are present, this group will become part of the Goodlist of the Structure Generator, insuring that every graph which is output as a hypothesis to explain the structure of the unknown substance will contain this particular subgraph. If all sufficient conditions are not satisfied, then this functional group is ignored.

A few examples of functional groups and their necessary and/or sufficient conditions are listed below. (In this description of conditions, "M" is the weight of the unfragmented molecule, and explicit numbers and variables such as x1 refer to mass points in the spectrum.)

---

* We assume this "composition" is given. It could be calculated by the program from the molecular weight and atomic weights of common atoms, but it was felt that the combinatorial problem was not of much interest.

1. Ketone, Identifying conditions:

   i. There are two peaks at mass units x1 and x2 such that:

      a. x1+x2 = M+28
      b. x1-28 is a high peak
      c. x2-28 is a high peak
      d. at least one of x1 or x2 is a high peak

2. Methyl-Ketone3, Identifying conditions:

   i. Ketone conditions must be satisfied
   ii. 43 is a high peak
   iii. 58 is a high peak
   iv. M-43 is a low peak
   v. M-15 is low or possibly zero

3. Ether2, Identifying conditions:

   i. Ether conditions are satisfied
   ii. There are two peaks at x1 and x2 such that

      a. x1+x2 = M+44
      b. at least one of x1 or x2 is a high peak

The supervisory function for the Preliminary Inference Maker is quite simple and general. It calls other functions such as "Contained", "Necessary", and "Sufficient", which are predicate functions, returning T or NIL. Only if all three functions return T is there the proper evidence for the appearance of a particular functional group. Once all the groups have been considered, the supervisory function then inspects the proposed Goodlist entries and eliminates redundant information. For example, some of the functional groups are subgraphs of other functional groups. Goodlist retains the larger, more specific, groups. This makes the graph generation process more specific, and reduces the number of output structures.

Because of the interdependence of functional groups, the ordering of Grouplist is of great interest. The order can help the program avoid

unnecessary work by placing the more general groups earlier on the list. When a condition such as "necessary conditions for group X must be satisfied" is encountered as one of the conditions for group Y, if X has already been considered, then the program just needs to check the temporary Goodlist and temporary Badlist to determine the status of group X and determine whether its conditions were satisfied or not.

Throughout this whole process, the spectral data is kept as a global variable. Its form is a list of LISP dotted pairs of number and amplitude, taken directly from published tables.* The main testing function of the program is called "Within". It determines if a given condition of a given functional group is satisfied by the spectral data. The usual question is: given a number and a range, does the amplitude associated with the given number in the spectrum fall within the required range? This is a simple test of numerical value. The interesting part of the process is in the range specifications that are permitted.

The program has a set of global variables which define concepts such as "high", "low", "medium", "possibly zero", "zero or one", or "any". A function called "Range" takes one of these global variables as input and returns a number pair stored under the range property of the global variable. An alternate type of input to the function Range is a single number. In this case, the function calculates an interval about the number

---

* Sometimes the spectral data is pre-processed to remove "noise".

and returns as its value the pair of numbers defining the endpoints of this interval. The global range parameters are assigned such values as:

HIGH = (11-100)

LOW = (1-5)

MEDIUM = (1-10)

The function Within is quite specialized and knows how to handle all the forms of necessary and sufficient conditions that may be stored as properties of the functional group names. Within can have any of several different types of arguments.

1. A group name will be checked against the list of previously checked and passed functional group names. If it does not appear, then Within returns NIL.

2. An atomic name which is not the name of a functional group will be evaluated as a function.

3 A dotted pair of (X . Y) will result in a call on the function Range. Y is always one of the global variables with a range property, namely: HIGH, LOW, MED, ANY, ZERONE, POSSO, etc.) X itself may have any of several types of forms, all of which specify a single number or a calculation to perform in order to obtain a single number. The spectrum is then consulted to see whether the amplitude associated with the number X lies in the range specified by property Y. T or NIL will be returned as a result of this test.

The value of Within is always T or NIL. If NIL is the result, then the given condition has been found to be absent from the data, and the program takes appropriate action.

The Preliminary Inference Maker communicates with the Structure Generator by giving it copies of the lists Goodlist and Badlist. The Structure Generator then computes all the graphs which can be made compatible with the

two lists of subgraphs, good and bad.

The Inference Maker is a good example of an efficient program for LISP. It does a non-trivial analysis process, using very general functions which access global variables for the information which they need in order to make decisions. Because of the large amount of specific chemical information contained in the Preliminary Inference Maker, quite a bit of effort has been put into making the man-machine interface smooth so that information may be easily transferred. The interactive parts of the program include "dialog" which questions a user about adding information, and an "editor" which saves all lists, atoms, and property lists in an updated memory for use in future program operations. Naturally, the success of the dialog portion of the program is mainly attributable to the interactive nature of the LISP system which we are using. LISP contains convenient mechanisms for storing words and sentences and printing them. We have had a few complaints about the formatting of LISP output, but these are minor compared to the overall successes of the program so far.

The current version of the program can discriminate in its dialog, depending upon what sort of mechanism is being used to communicate with the user. The list of messages to the user is contained on a disk file. If a cathode ray tube is being used for communication with the user, the program will fetch the message and print it on the display. If a teletype is being used for communication, on the other hand, the program will print the message number and inform the user where to find the message if he needs further information.

The effect of using the Preliminary Inference Maker in conjunction with the Structure Generator is to greatly simplify the graph making process and to drastically edit the long output lists which used to result from the operation of the Structure Generator alone. The Preliminary Inference Maker contains the specialized information necessary to interpret the data from the real world. The Structure Generator remains a general graph manipulating tool.

6. The Predictor and Scoring Programs

In the course of employing the Dendral program to formulate hypotheses to explain mass spectral data, it became apparent that the computer needed a detailed theory of mass spectral fragmentation processes. This is because the Preliminary Inference Maker and Structure Generator suggest plausible candidate structures for explaining the data, but have no way of testing these candidates. Having a theory by which the computer can make some verifiable predictions about each candidate helps to reduce the number of likely candidate structures.

The two programs described in this section provide a suggested mass spectrum for each candidate structure, and a comparison between the predicted spectra and original spectrum to determine which structural candidate is the most satisfactory explanation.

A mass spectrometer is, briefly, an instrument into which is put a small sample of some chemical compound and out of which comes data representable as a bar graph. The instrument itself bombards molecules of the compund with electrons, producing ions of different masses in varying amounts. The x-points of the bar graph represent the masses of ions produced, and the y-points represent the relative abundances of ions of these masses. The Predictor program is an attempt to codify the numerous descriptions of what happens inside the instrument, and thus to generate mass spectra in the absence of both the instrument and the actual sample of the substance. Thus, the Predictor is a computer simulation program. The input to the program is the graph structure of a

molecule. The output is a bar graph representing the predicted mass spectrum for this molecule, and a list of those peaks which are thought to be the most significant in the simulated spectrum.

The molecule is represented internally in the list notation for tree structures described in Section 1. This notation omits explicit mention of hydrogen atoms, but shows all the other connections of the chemical graph. The program attaches a unique name to each atom, and keeps track of each atom's place in the graph by putting names of adjacent atoms on the property list of each atom under indicators "FROM" and "TO". Thus, the program contains two different representations of the structure. Each representation has its uses, the list notation for purposes of calculating masses and searching for subgraphs, and the property list (or connection table) for examining details of structure in the vicinity of a particular bond or atom.

In outline, the Predictor uses the following series of steps to calculate a spectrum for a molecule:

1. Consider each subgraph (fragment) of the molecule. The original list of all fragments includes the whole molecule itself, plus the list of all fragments which result from breaking each bond in the molecule one at a time. This list is later augmented by other fragments which are produced by recombinations of fragments and atoms.

2. Calculate the mass of the fragment ion. The mass is the sum of the masses of all atoms plus the masses of all implicit hydrogen atoms.

3. Calculate an intensity (on an absolute scale) for the fragment ion by estimating the probability that the bond between it and the rest of the structure will break and estimating the probability associated with ionization of each of the resulting fragments.

4. Determine the nature and extent of eliminations and rearrangements of each fragment. These will provide new fragment types, which should be added to the list formed in #1 above.

5. When all members of the fragment list have been considered, process the bar graph to account for variations in the exact weights of the atom types (isotopes), and to put the data into the form usually seen as output from a real mass spectrometer (namely, the highest peak has intensity = 100, and all other peaks are scaled relative to this one.)

The process of obtaining the original list of fragments and their masses is straightforward. The process for determining the intensity for each fragment ion considers the likelihood that the molecular ion will break at each of the bonds between adjacent atoms. Its theory says that only single bonds will break apart, thus it skips over double and triple bonds in the molecule. Of the single bonds, it distinguishes bonds between carbon atoms from bonds between a carbon and a non-carbon atom. The probability that the ion breaks at a given bond depends upon the environment of the bond and the functional groups present in the molecule. The probability associated with the ionization of one or the other of the resulting fragments also depends on these features. This part of the program is organized as a set of conditional sentences: if the molecule contains functional group X and this bond environment has feature b, then include f(b,X) in calculating the probability that the molecule breaks apart at this bond.

The program contains three lists which allow it to perform these tasks. One is the list of significant radicals (functional groups, as in the Preliminary Inference Maker program). Another is the list of features of the bond environment. And the third is the associated list of functions to calculate probability factors for each of the features. The result of calculating

the probability of ionization of a fragment is a number pair, (x . y). The first component is the mass of the fragment. The second is the relative abundance of these fragment ions ($y \geq 0$).

The fourth step in the process for obtaining the mass spectrum requires the program to determine the nature and extent of "eliminations" and "rearrangements" for each fragment ion. Since any ion, including the molecular ion, tends to a more stable form if possible, the program must take account of the relative stability of each ion, as compared to the stability of possible products it may form. The program looks for ways that certain neutral molecules may be eliminated from the ion, and also for ways of rearranging the atoms already present in the fragment. The program has a list of very stable ion products, which are preferred structures; and the program must determine whether, and to what extent, the given fragment ion can form one of these products. To do this, it looks for any occurrence of significant radicals as subgraphs of the ion. Associated with each of these significant radicals ia a set of rules for restructuring the ion to make it more stable, and a set of parameters for specifying the extent to which this restructuring should (or does) occur. Heuristic programmers recognize such a plan as a list of situation-action rules of the form: in situation X perform action Y. A very desirable feature of this is that the list of significant radicals can be extended or amended very easily and the associated parameters are all available for changes at the interactive level of the program.

In order to carry out these processes of elimination and rearrangement, the program has rules for removing atoms, changing the order of

bonds, and moving atoms from place to place, so that the structure of rearrangement products will be available to add to the list of fragments.

By the time the program has finished considering all the fragments, it has calculated a list of mass-intensity pairs which are considered the most significant peaks in the mass spectrum. Isotope peaks are added to reflect the distribution of weights for each type of atom. Some additional minor processing of this list yields the predicted spectrum for the input structure.

The Predictor program reflects the chemists' uncertainties about the actual operation of a mass spectrometer by employing many parameters to represent possible effects of different mechanisms. These parameters are global variables, and only their names are built into the program; their values are accessible to the user at the interactive level. The performance of the program can be changed noticeably by altering one or more of these parameters. One of our coworkers has had some success in optimizing the values of some of these parameters, one at a time, with a hill-climbing procedure using comparisons between predicted and real spectra for known structures. Setting some parameters to zero has the effect of making the program ignore the associated feature or probability factor. Changing the list of significant radicals also has its effect on operation of the program. Whether the effect is good or bad is for the chemists, not the programmers, to determine.

The Predictor program has evolved in an ad hoc fashion over the last three years, under the supervision of several different programmers, and with the advice and guidance

of several chemists with different approaches to the subject of mass spectrometry. The part of the program which is organized as situation-action rules is the most coherent and most easily observed, tested, and changed. It is planned that the whole Predictor program should be re-written along these lines and in such a form that it will be possible to insure that the information used by it and the Preliminary Inference Maker will be consistent. (See Section 7).

The output of the Predictor is input to a fourth LISP program, which compares the predicted spectra with the real data and decides how "good" the match is. Each predicted spectrum has associated with it a list of peaks and their degrees of theoretical significance. The first action of the Scoring program is a consistency check with the real data, based on this list of significant peaks for the hypothesized molecule. If a predicted peak is significant according to the Predictor, but does not appear in the real data, then the hypothesized structure has to be discarded.

The rationale for this is the following: from a given set of spectral data alone there is no idea which are the significant peaks. High amplitude does not necessarily indicate significance. However, the process of analyzing the fragmentation yields information about which fragments are significant, because there is some theory about bonds likely to break and about the relative abundance of the resulting fragments. So the process of forming a spectrum yields information about which are the important peaks it has formed. In addition, the same process indicates that some peaks are more significant than others, which reflects the fact that

chemists are more certain about some parts of the theory than about others.

After the scoring program has removed all hypotheses which are not consistent with the data, it orders the list of remaining structures according to the total number of significant peaks each contains. In the case of ties, where two or more spectra have the same number of significant peaks, the structures are ordered with respect to the sum of the degrees of significance associated with the significant peaks. This final ordering is the final output of the Heuristic Dendral program. It represents the program's judgment about which hypotheses best describe the structure of the original unknown substance.

The program has been run many times, mostly with data from known structures so that the power of the method may thereby be verified. The results have been remarkably good; the real structure almost always appears as the first hypothesis in the final output list. When it is not the case that the right answer comes first, the chemists then must start thinking about the inference and fragmentation rules they have specified, and in such cases they usually find good reasons for changing existing rules or introducing new ones.

7. Program Expansions

It is interesting to notice that both the Preliminary Inference Maker and Predictor programs use the notion of functional groups or subgraphs. The Preliminary Inference Maker associates characteristic spectral peaks with each functional group. A set of spectral data is inspected for the presence or absence of these peaks, and the functional group or groups are selected by a process of elimination. The Predictor associates actions with each functional group. An action tells the program what will happen to a structure containing the group when it is fragmented in a mass spectrometer. The spectral peaks are then calculated from the resulting fragments.

It appears that both these programs are based on the same data source, and there have been quite a number of discussions about how to insure that the two programs are consistent, but not necessarily equivalent, expressions of the same theory. It would be nice to have the two programs be task independent, and refer to some common body of information to obtain the rules needed in a given case. (The Structure Generator is task independent in the sense that its chemical information is all contained on a few global lists.) Currently there is no such computer collection of chemical data, nor any translator program to reference the data if it were available. But it will be an interesting and challenging task to develop such a system.

In the course of developing Heuristic Dendral and showing it to many chemists, the question was asked about why mass spectral data, in particular, was used, in view of the fact that other types of data are available which

provide inferences about the structure of an unknown substance. From the point of view of artificial intelligence, it does not matter which type of data is used; but from the chemists' point of view it is desirable to use as much data as can be made available in deducing an unknown structure. To facilitate this process, other versions of the Preliminary Inference Maker have now been programmed which are of the same form but which contain different sorts of information in the global variables and property lists. These other inference makers now use rules obtained from the study of infra-red spectral data and nuclear magnetic resonance data. The programs can be used either before the mass spectral inference maker is called, to shorten the list of suggested functional groups (Grouplist). Or, alternatively, the auxiliary inference makers can be used between the Preliminary Inference Maker and the Structure Generator to further modify Goodlist and Badlist according to the specialized information which they bring to bear on the problem.

This is a very nice feature of Heuristic Dendral from the point of view of computer scientists, and artificial intelligence researchers in particular. It means that a program has been written which is general enough in its structure to be able to incorporate different types of inference rules, and that several such programs can be made to work in accord to pass a unified set of suggestions to a general program such as the Structure Generator.

8. General Comments on the Heuristic Dendral Family of LISP Programs

The four LISP programs which comprise Heuristic Dendral have been shown to perform remarkably well. The results compare favorably with the work of scientists in the field of mass spectrometry. The four programs are usually combined into two big units, the Preliminary Inference Maker and Structure Generator function together and write the hypothesized structures on disk or tape devices. Then the Predictor and Scorer read these devices to obtain their input, process the structures, and print the resulting ordered list of hypotheses. There is no theoretical reason why all four programs could not be combined into one program. The practical reason for not doing so is that the estimated size of computer core memory to house the entire system is about 90-100 thousand LISP words, each LISP word being one computer word in the case of the PDP-6/10 or being eight bytes on an IBM-360 class computer. With the current trend to time sharing on big computers, we find that most other users are not eager to give us the whole computer for our program.

It has been convenient to have LISP as an environment in which to develop this very successful project. The language lends itself to gradual expansion of programs and to the "hanging on" of many small special purpose functions and sub-programs without requiring a great deal of internal reprogramming in the main program. The lambda and prog variable bindings assure us that we can write functions which can be executed and have no unwanted side effects on other parts of the program.

An early section of this paper mentioned that LISP data structures seem ideally suited to the Dendral representation of graph structures in recursive lists. Other types of representation are used in the programs also. The spectrum is usually represented as a list of LISP dotted pairs. Certain parts of the Structure Generator find it useful to represent structures as connection tables in the form of arrays. The Preliminary Inference Maker stores all information on property lists and global lists of names of things, which in turn have property lists. The Structure Generator uses property lists mostly for node-attributes, but this seems quite an efficient device. An early version of the Structure Generator placed all its information on global lists, and searched these lists to find the values it wanted. The property list scheme seems more direct, and makes it far easier for the programmer to keep track of the types of information needed by the various programs.

The Predictor uses a connection table type of representation in addition to the list notation for the structures it is manipulating. These connection tables are not implemented as arrays, but rather as property lists of nodes of the graph. Each node in a structure is numbered, and a LISP atom name is made by compressing the node name and node number. Each node has a property list containing "TO" and "FROM" properties, indicating the names of the other atoms to which it is connected and the types of bonds which connect them. However, the Predictor also uses the list notation for some purposes, in particular for checking for the presence of subgraphs in a larger graph. The method for graph-matching is the same as the one used by the Structure

Generator, but it is currently felt that some sort of graph-matching using the connection tables would be more efficient.

In summary, the things that we like about LISP include the following:

1. The way the program has grown from the inside out (the ease of adding top level functions).
2. It is convenient to store miscellaneous types of information on property lists (convenient indirect referencing).
3. Different types of variables can be used for different purposes: global variables for switches, special variables for efficiency, and prog and lambda bound variables for independence of different parts of the program.
4. Many different types of representation are convenient: lists, arrays, connection tables, property lists.
5. Our version of the LISP system allows a program to be expanded as more core is needed. This is frequently quite useful when large data structures are built and saved for future reference in the dictionary.
6. It is possible to reference (and change) functions like any other type of data.

There have been several characteristics we would like to have found in LISP which are not currently implemented, or perhaps not even currently contemplated:

1. List structures are one-way. It is not as convenient to access a preceeding item on a list as it is to find a successor. Many attempts have been made at two-way list-type languages. None has been successful enough, apparently, to find its way into the Stanford LISP system. Using property lists of atoms has been our solution in the cases in which it has been useful to have the predecessors and successors of a node be equally accessible (particularly in the Predictor program).

2. The store-versus-recompute problem could be (but is not) handled more automatically by LISP. It is of interest to know whether a function is being entered repeatedly with the same arguments. The POP-2 language developed at the University of Edinburgh has something called MEMO Functions, which apparently attack with some success this problem of knowing whether a given function has already been executed with certain arguments. Our program has only one such function, that being the Structure Generator function, Genrad, which generates rooted trees. The function builds a dictionary of trees upon its first call with a given set of arguments, and thereafter has this dictionary to refer to when about to proceed on a given line of work. Naturally, if no dictionary entry appears for a given set of arguments, the whole function is executed. There are several bad points about this type of dictionary-memory function. Much time is spent by the program in generating the dictionary, which may or may not ever be used. The function needs to return only a single rooted tree, but is required to produce all of them in order

to make a complete dictionary. (This particular problem is more a fault of the implementation than of the language. Knowing our desires now, in retrospect, it would be quite reasonable to make the automatic argument-memory more efficient.)

3. Right now we are battling the Standard LISP problem. We have spent several months trying to make our program run on both the PDP-10 and the IBM 360/67 at Stanford. The Structure Generator program was developed on the Q-32 computer, which seems to approach the ideal of Standard LISP. But most of the embellishments to the Structure Generator, and all of the coding of the other three programs were done in the non-standard LISP of the PDP system. Although some of the features of the PDP implementation are quite convenient, in retrospect it was not worth it to take advantage of these features, since we are now having either to change our code, or program around the inconsistencies in an attempt to have programs which will be machine-independent. It is also apparent that some of the inconsistencies between the two LISP implementations were done just for the sake of being different. (Why else should MAPLIST have its arguments reversed in the two implementations of LISP?)

At least we are complaining only about the difference between two implementations of a relatively stable language. It is a definite advantage to be programming in a language which is widely known, respected, and implemented. We expect the Heuristic Dendral program to reach the stage where many scientists will be interested in using it. Since we are really not chemists, but rather researchers in Artificial Intelligence,

it is of great value to us to know that our program can be expected to run on many other computers, and that educated LISP programmers can be taught how to modify the program in order to satisfy their own needs. LISP makes this possible, while still retaining the character of a list processing language. (FORTRAN and ALGOL are universal enough for our purposes, but not versatile enough.) So, it is our goal to make our programs separate from the information which they manipulate, by putting the chemical information on easily changeable lists and variables. Then we can turn the Heuristic Dendral program over to the chemists or topologists so that they may adapt the program to their own particular applications.

References

1. Lederberg, J., DENDRAL-64, A System for Computer Construction Enumeration, and Notation of Organic Molecules as Tree Structures and Cyclic Graphs, Parts I - V, Interim Report to the National Aeronautics and Space Administration, December 1964.

2. Lederberg, J. and Feigenbaum, E.A., Mechanization of Inductive Inference in Organic Chemistry, Stanford Artificial Intelligence Project, Memo No. 54, August 2, 1967.

3. Buchanan, Bruce and Sutherland, G., HEURISTIC DENDRAL: A Program for Generating Explanatory Hypotheses in Organic Chemistry, Machine Intelligence IV, University of Edinburgh Press, 1969, and Stanford Artificial Intelligence Project, Memo No. 62, July 26, 1968.

4. Feigenbaum, E., Buchanan, B. and Sutherland, G., HEURISTIC DENDRAL: A Program for Generating Explanatory Hypotheses in Organic Chemistry, Machine Intelligence IV, University of Edinburgh Press, (in press). This is virtually the same article as reference 3.