4-22-1986

Design of an expert system to perform cladistic analysis

Walter Wolf

Design of an Expert System to Perform Cladistic Analysis

Walter A. Wolf

Rochester Institute of Technology
School of Computer Science and Technology

Design of an Expert System to Perform Cladistic Analysis

Walter A. Wolf

A thesis, submitted to
The Faculty of the school of Computer Science and Technology,
in partial fulfillment of the requirement for the degree of
Master of Science in Computer Science

Approved by:

John A. Biles 4/22/86
John A. Biles

Lawrence A. Coon 4/22/86
Lawrence Coon

James E. Heliotis 5/4/86
James Heliotis

Title of Thesis Design of an Expert System to Perform Cladistic Analysis

I Walter Wolf hereby (grant, deny) permission to the Wallace Memorial Library, of R.I.T., to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

CONTENTS

1. Project Background

Before I commenced the study of computer science I was a biochemist, most recently involved in the study of the biosynthesis of sex pheromones in certain moths. Sex pheromones are compounds that females emit to attract mates. It seemed natural to look for a thesis project that would utilize the knowledge and training that I already possessed.

It turned out to be difficult to design a meaningful project in my field, but some interesting projects in associated areas were considered. Two of these were particularly appealing.

The first involved modeling the effects of certain drugs on the male insect's reactions to the sex pheromone. Normally the males undergo a set pattern of behaviors which can be predicted statistically. However, various drugs influence and interrupt this behavior pattern in different ways, and this allows them to be used as nervous system probes. Although intriging, this project involves too much experimentation and speculation and not enough use of the computer for a valid computer science thesis. It remains as a potential research project.

The other problem involved the classification of insects into families, etc., an activity that traditionally has depended upon physical characters, but more recently has also used chemical information, such as the identity of the sex pheromone. A particularly interesting, but relatively young, branch of the discipline is called cladistics, which involves consideration of evolutionary relationships as well as physical and biochemical ones. It was decided that this would be a suitable field in which to write a simple expert system , as it was data intensive, thus making the use of a computer attractive, if not necessary, and yet not well developed, thus limiting the number of rules the system would have to consider.

In this thesis I will first try to acquaint the reader with some basic concepts of artificial intelligence, expert systems and cladistics. Then the process we went through designing our system will be outlined and the actual system presented.

One point was brought home to me early in my reading -- creating an actual, functioning, useful system was highly unlikely given the limits of a master's thesis project. The major

systems presently written started with doctoral thesis and developed from there, and it has been estimated that even a "bare-bones" system would take at least a man-year to develop [A. Poole, personal communication], not counting testing and refinement.

However, many of the major decisions (data representation, rule format, etc.) have to be made early in the project, and the inference engine code should be written first, so that even a system too simple to be of much use will serve the educational purposes of a complete development. Of course, the system could then be further fleshed out, or the lessons learned could be applied to other fields, perhaps some of interest to a wider audience.

2. Cladistics.

2.1. Background.

One of the oldest fields in biology is taxonomy, or systematics, which involves the classification of living organisms into species and defining the species relationships. More recently this has been expanded to include the evolutionary aspects of speciation -- that is, how species developed over time.

Historically, three major methods of classification have been used:

(1) Numerical (phenetic)[Colgan; Romesburg]. This method is based on a numerical analysis of traits, either the few exhibiting the widest differences or the maximum number that are shared. This method is quite popular due to its essentially numeric nature which can easily be programmed, thus lending the mystique of the computer to it and its practitioners. However, it has a major problem -- it can lead to grouping of species that do not have a common ancestor (polyphyly) or result in decendants of a single ancestor being placed in different groups (papaphyly).

(2) Evolutionary taxonomy[Rose]. In this method, the rate of evolutionary change (or degree of difference) is reflected in the classification. Organisms that have undergone considerable change are placed in a separate group from close relatives that have remained more "conservative," or similar to the ancestor. A problem with this method is that it gives rise to paraphyly.

(3) Phylogenetic (cladistic) [Cracraft; Hennig]. This method emphasizes recency of common ancestry. Two operational taxonomic units (otu's -- groups on the level you are currently operating, eg, species, families, etc.) that share a common ancestor are considered more closely related than is a third otu that lacks this common ancestor, even if it shares many of the characters of one of the others. In other words, degree of change is subservient to recency of common ancestry. All decendants of a common ancestor must be grouped together.

Two or more otu's that share derived (unique) characters are considered to have an ancestor in common. One of the main duties of the cladist is to determine which characters are derived and which are ancestral. Classification must be done using derived characters only to be valid.

This decision is best based on fossil evidence. Lacking this, "out-group comparisons" are used -- if a character appears in other closely related groups, it is probably ancestral. This requires extensive knowledge of groups close to those being studied, a knowledge that tends to vary amongst practitioners of the subject.

2.2. Cladistic Analysis.

Before one can start a cladistic analysis, which explores the evolutionary relationships between otu's, you must have the species under consideration fully defined and possess some information about their evolutionary history. Then the following decisions must be made:

(1) Which characters define the classification and are they in any way related or grouped (ie, if red hair and freckles go together, are these one character or two?);

(2) What are the sister (or ancestral) groups;

(3) Which characters evolved once and which may be the result of convergent evolution;

(4) Which characters are ancestral and which are derived; and

(5) In cases where independent characters are incongruent, which has more weight (ie, if A and B have trait Q and B and C share trait Z, is B closer to A or C?).

2.3. Biochemical Implications.

Traditionally, classification has been based on morphology -- descriptions of physical parts of the organism under study (shape of wings, size of ova, etc.) Recently, biochemical characteristics have also been considered [Brown; Rose]. As this is an area in which I have done some research, these developments are of interest to me.

Female moths alert potential mates to their position by emitting small amounts of a chemical, called a sex pheromone, that specificly attract males of their own species. As characters related to reproduction are particularly important in speciation, information about the identity and biosynthesis of these compounds should be of great use to the cladist. This is just the kind of information that I have available.

Thus it is of special interest to me to construct a system that would aid the cladist in incorporating biochemical data into his analysis. Adding new information to the system should be easy and it should be simple to modify, so that various ways of interpreting the effects of the added information can be seen. Finally, it would be desirable for the system to have the potential to possess enough expertise to suggest proper weighting and grouping schemes, or at least to assist the cladist in these decisions.

3. Artificial Intelligence.

Artificial Intelligence (AI) has been defined as the branch of computer science dealing with symbolic, non-algorithmic methods of problem solving[Buchanan]. Items are related and conclusions reached through judgmental rules, or heuristics, as well as through theoretical laws and definitions. Programs contain inference mechanisms and/or knowledge bases and posses mechanisms to search for facts and associate items when appropriate.

Historically, this field arose from the study of (human) intelligence (Winston). Computers were employed because they aid in thinking about thinking, force precision in implementation, quantify information processing and, perhaps of most practical importance, are easier to work with than animals. (My wife, the sheep farmer, does not agree with the last reason). Major fields of investigation included matching, goal reduction, constraint exploration, search, problem-solving and logic methods. Additional topics include "common sense," understanding natural language, and learning.

Looking more closely at AI reveals it to be somewhat less than indicated above. What is particularly noticeable is that the field is not non-algorithmic -- rather the types and purposes of the algorithms have been changed. At this early stage in its development, AI is still closely linked to traditional computer science, but is expanding its scope and the types of problems it attempts to cope with.

As in all computer programming, one of the biggest advantages of doing AI is that it forces the practitioner to think about the details of the over-all process he wishes to implement. Such problems as knowledge acquisition and representation, data base manipulation and search strategies are interesting and are not always encountered in more traditional areas. Thus working in this field forces the programmer to consider new representations and implementations and, hopefully, new ideas and strategies.

4. Expert Systems.

4.1. Definition.

The term 'expert system' has proven to be difficult to define exactly. Commonly, such a system is said to achieve expertise in a field, although there is general disagreement as to what is "expertise" and how extensive the scope of a field should be.

The definition I found most useful is an operational one proposed by Buchanan and Shortliffe. They say an expert system is ".. an AI program designed (a) to provide expert-level solutions to complex problems, (b) to be understandable and (c) to be flexible enough to accomodate new knowledge easily." In addition, I feel it should be usable by a variety of users, from non-experts needing direction to experts who themselves need advice and easy access to large data-handling capabilities.

Let us briefly examine these points. We all know that there are experts in certain fields -- practitioners who, with access to the same data as non-experts, arrive at "better" conclusions, usually in less time. How they do this is often unclear, especially to the experts themselves.

It is generally felt that experts employ heuristics, or rules of thumb, that allow them to quickly focus on the key elements of the problem and assemble these into a reasonable conclusion. A key to building an expert system is to find an expert who can (and will) articulate the heuristics in a form that can be coded for a computer. Thus we must have an expert's heuristics, a way to represent them, a data base to which the knowledge can be applied, and an effective language or set of procedures to allow these elements to function together.

The system must be understandable. This implies that an non-expert finds it reasonable to use and understands the logic that supports the presented conclusions. Reasonable to use means it must quickly focus on the problem at hand and explore it in an expected fashion. For example, the builders of the MYCIN system (see below) found it necessary to include questions to which the system already knew the answer because users so expected these

questions that when they were not asked the users would not believe the conclusions.

In addition, the system must be able to 'back up' its answers through some kind of explanation facility. This is not a large problem in a system that closely models human thinking, but can be a real problem in those that use other types of approach to reasoning.

A system should easily accomodate new knowledge. This could be immediate -- information about the problem at hand that guides decision making -- or long term -- changes to the knowledge base of the system. These may be done by different methods, but both should be simple to use and have the bulk of the work done by the system.

Finally, the system should operate in several modes, depending on the expertise of the user. It should make decisions on behalf of the non-expert, make data manipulation and attempting alternates easier for the expert and be capable of learning from those willing to share their expertise. In this way it should reflect a human expert, whose exact behavior depends not only upon the problem but also upon the degree of sophistication of the people he is working with.

4.2. Knowledge and its uses.

For one such as myself, whose experience with computerized problem solving involves mostly interactive, computation intensive programs, it is tempting to conclude that the key to expert performance lies in the schemes (procedures) one designs for manipulating data. However, this is not the case. "The power of an expert system derives from the knowledge it possesses, not from the particular formalisms and inference schemes it employs" [Hayes-Roth]. Knowledge is key -- knowledge representation and inference schemes provide the mechanism.

The knowledge referred to really consists of two components: data (facts in the field) and heuristics (special rules unique to the field.) Thus most successful systems are not general -- their content and implementation are dictated by the content and formalisms of the field under consideration (they are not domain independent).

It has been a "long term" goal (if so young a field can have such a thing) of persons developing expert systems to create a more general inference mechanism. This will not be an "expert in everything," but rather will be a general knowledge manipulator which can be supplied with (or learn) various data bases to become an instant expert in the field of choice. In fact, one can obtain such systems(eg, IRIS, TEIRESIAS). However, at present these are rather restrictive and only serve well for simple problems, even if you have an expert willing to use them.

So far we have treated knowledge as an absolute. In fact, much useful knowledge is somewhat uncertain -- this may be true, or usually (but not always) that happens. To be able to deal with complex problems, an expert system must be able to deal with uncertain (fuzzy) data and probabilistic rules. This can be handled, to some degree, by the use of weighting factors, but is an area of expert system research that can use further development [Negoita].

4.3. Kinds of Systems.

There are many possible ways to organize the knowledge that the system contains and manipulates. Some of these are discussed below.

4.3.1. State Space Search.

The problem is structured in terms of alternatives available at each state. The next possible states are determined by a set of rules and transitive operators. In general this leads to a combinatorial explosion which requires discarding certain states (not examining them or states derived from them.) One type of space state search is called generate and test, where all possible next states are generated and some testing procedure discards poor (unlikely) ones.

4.3.2. Logical Systems.

These follow the rules of formal logic and are most suited for small, well defined systems such as theorem provers. The explanation facilities of these systems can easily be constructed if they are limited to the rules used, rather than attempting to explain the process of

selection.

4.3.3. Procedural Representations.

This involves a series of small steps, each the result of a well-defined situation. Often these give operationally good results, but ones that are difficult to explain.

4.3.4. Semantic Nets.

These consist of nodes (objects, concepts, events) and links (interactions). Related facts can be closely linked to reduce the effective search space. The net is not directed, so the entire search space is always available, making this a very versatile and possibly complicated method.

4.3.5. Frames.

These are data structures that include declarative and procedural information as well as relationships. One of the interesting attributes you can program into a frame is the ability to decide if it is useful in a given situation and, if not, to call another in its place.

4.3.6. Production sytems.

The basis of these systems is a set of rules in the form if (condition) then (action). Each rule is explicit and independent (doesn't call another rule.) This organization is good for large systems that include self-modification as a feature, as the rules can refer to the system itself as well as to the area of knowledge the system is about.

A "pure" production system would consist of a rule interpreter (inference engine) and a two-part data base: a set of rules and a collection of facts. How closely the rules are coupled to the rule interpreter and how general their form is will determine how domain specific the system is. As mentioned above, most systems are, in fact, quite domain specific, due to the form of the rules and the tailoring of the inference engine.

The rules themselves would have the conditions evaluated (or matched) with reference to the data base. If this was successful, the action would be performed. If unsuccessful,

another appropriate rule would be applied or a failure indicated. More than one rule may apply to a given situation, so some sort of conflict resolution is necessary.

The data base is the sole storage for all the state variables of the system. It is totally and equally accessible by all rules. If desired, it can be modified either by the user or by the program itself.

The interpreter assists in selecting and executing the relevant rule and in interacting with the user. It may also contain some sort of explanation facility, although this is not really a part of a "pure" system.

4.4. Examples.

4.4.1. DENDRAL.

One of the first successful expert systems, DENDRAL uses a generate and test organization to deduce organic structures from mass spectrograms. Its strength lies in an exhaustive algorithm to generate all possible structures and an effective test series to prune the search tree (ie, an extensive set of rules related to mass spectrography.)

4.4.2. meta-DENDRAL

A system that learns how to interpret mass spectrograms by being taught by an expert. It is supplied spectra and seeks interpretations from the expert, from which it derives its own rules. It then helps the expert (or others) to interpret other spectra.

4.4.3. CASNET.

This is a network system designed to aid in the diagnosis of glaucoma. Nodes are confirmed or denied by questionning the doctor and then the treatment is found in a table. It is easily expandable, has a limited explanation facility and is well-rated by users (opthalomoligists.)

4.4.4. MYCIN.

MYCIN is a production rule system designed to diagnose infectious diseases. It interviews doctors and records facts along with confidence factors. It possess more than 500 rules, and new ones can easily be added. Its explanation system is extensive and is in plain English.

5. Implementations.

5.1. General.

In *Building Expert Systems*, Hayes-Roth et. al. outline an approach to the actual implementation of an expert system. They suggest the following steps:

Identification.

Identifying the problem and its characteristics.

Conceptualization.

Deciding on the concepts to represent factual knowledge and the knowledge contained in the heuristics.

Formalization.

Designing structures to organize the knowledge.

Implementation.

Designing interpretation methods for the data and heuristic, and

Testing.

Validating the system.

The actual design process, of course, consists of multiple iterations through the steps.

Note that the expert plays a key role in the identification and implementation steps, as well as lesser roles in the others. This reemphasizes a point made many times in my reading: the expert must be consulted early and often. Knowledge is the key, not the programmer's cleverness, although this always helps.

The system is more flexible and easier to update if the knowledge base is separate from the inference engine. This does not mean you must have just two large files, but rather that the logical control functions are separated from the data. In general, one can not totally accomplish this, but it should be attempted. The intertwining of control and interpretation and the unique format of the interpretation method is what makes most expert systems more domain specific than general.

5.2. Choice of Language.

Most work in Artificial Intelligence up to this date has been done in some dialogue of LISP, even though this is not always apparent. For example, one of the standard texts in the area [Winston] is more or less language independent, but comes with a supplementary volume in which all of the examples in the book are implemented in LISP. Indeed, this companion volume is designed as a LISP teaching tool, so one can learn Artificial Intelligence and "its language" at the same time.

Some reasons for this are offered by Hayes-Roth. Both authors emphasize the ability of the language to handle symbols and perform 'computations' with them. It is important to realize that a symbol or word means nothing to the computer or to the LISP compiler, but it can be manipulated as though it did (by suitable algorithms.) LISP is designed to handle large numbers of symbols (lists) and their relationships in a recursive fashion, and is quite effective at doing this.

Another advantage of this language is that it is normally used in a rather interactive environment, allowing the program to be developed in small bits, function by function, which saves a great deal of time. Also, most elements of memory management are taken care of automatically, freeing the programmer from some worries.

Finally, there is the least "solid" but perhaps most important reason -- habit [Foster]. LISP is so firmly entrenched in the field that many practitioners simply do not want to invest the effort in learning an alternative. This position is bolstered by the fact that many development tools are available in one dialogue or another of LISP, and these can be massive time-savers.

Recently, a challenger to LISP has emerged. Prolog, a logic programming language, was developed in Europe, and is popular there and in Japan, where it was chosen as the language of the fifth-generation computer project.

One of the main advantages of Prolog is that programming in it is easier. As is LISP, it is generally interpreted, which saves vast amounts of compilation time while in the

development stage, and can be implemented and tested in small stages. The code itself is often easier to understand than is that of LISP, though some recursive procedures can be equally confusing in either language. It is also more portable than LISP, currently, at least, lacking dialects (as we shall see later, this is not strictly true.)

Prolog has some drawbacks, also. It saves the programmer the need to program a search of a large data base by doing it for him, but in doing so limits the available options. The search strategy is depth-first, which can be quite inefficient when dealing with a large data-base, such as those used in expert systems.

It is difficult to change this strategy in Prolog. One can "get around" it by employing rather baroque data structures [Clocksin], but these add little to program clarity or speed. Prolog does employ a hash technique in data searches, but once again the programmer has little control in cases where this is not the best approach.

Prolog is also younger than LISP, and thus lacks the well-thought-out set of development tools of the older language. In addition, it is closely tied to mathematical logic, which sometimes makes "common-sense" logic difficult to implement.

Another difference is that LISP is "lower-level" than Prolog, allowing more versatility at the price of having to handle more details. Prolog does much of the data-base management "under the hood," but certain approaches are harder to employ in it. In particular, Prolog is ideal for rule-based systems, where a rule is of the form 'if x (and/or y and/or ...) then a (and/or b and/or...). Other types of organization (e.g., blackboards) can be implemented in Prolog, but not as easily.

Some choose to continue this conflict between languages while others are trying to resolve it. One resolution approach involves the development of languages that borrow strong points from both LISP and Prolog[Foster]. One such system, Super-LOGLISP, is currently under development at Syracuse University, and is expected to include parallel processing capabilities. A second approach involves writing interpreters that can handle both LISP and Prolog (as well as other languages) so the programmer can choose the most appropriate

language for the use immediate problem. Both LISP and Prolog are quite limited in their ability to do calculations, so it would be helpful to include some sort of upgrade in this area as well.

6. Initial Decisions.

6.1. Scope

Before actually implementing my system, I had to make several decisions. These were 'large' in nature and determined the nature and scope of the actual expert system.

The first, of course, was choosing the problem to be worked on. As already mentioned, we decided to write a system that dealt with the field of cladistics. This is of special interest both in itself and in reference to some other (biochemical) work that I have done.

This led easily to the second decision, one of considerably less import. It seemed natural that an expert system dealing with cladistics be named Claud, and so it was.

A much more important question involved the breadth of the system -- who was to use it and what capabilities it was to have. The standard concept of an expert system involves creation experts (in computer science and the field of interest) but use by non-experts. A non-expert could be a "man off the street," but more usually was envisioned as a person at least somewhat knowledgeable in the field under consideration.

The way such a system functions can vary, but certain basic elements are usually present. These include a data base, a set of manipulations (heuristics) supplied by the expert and a way to apply these manipulations to the data for the case under consideration. In addition there may be some ability to add to or change the data base and to explain the way the system's conclusions were or were not reached.

Thus the system can contain the ability to manipulate and change a large data base specific to a field as well as the ability to apply certain heuristics to it. The manipulative abilities would be extremely useful to another expert, who probably doesn't need, at least constantly, the heuristics in the program -- he brings his own to the problem.

Thus, with just a small (sic) amount of additional effort the system can be made to serve other experts as well as non-experts. It was decided to include these capabilities in Claud, both because they made him more useful and because this would force me to consider how to

implement such facilities.

The next problem to deal with was the role of the expert. I had been promised cooperation by Dr. Richard Brown, an entomologist who did traditional cladistics and was also interested in expanding its scope to include biochemical data. Although I was convinced of the necessity of keeping in close touch with him, the fact that he was (and is) in Mississippi raised some practical difficulties.

Thus we compromised on the following scheme. I discussed the general approach with Dr. Brown several times via telephone. He then supplied me with some of his "rules-of-thumb" as well as some real data and his conclusions from them. I incorporated the heuristics into Claud, performed the type of analysis he does (see below) and checked back with Dr. Brown to be sure that I understood him and that Claud reached reasonable conclusions. The fact that it did was gratifying but expected on the rather simple level that Claud functions.

This process gave me a very clear illustration of the need to be close to the expert. The kind of heuristics Dr. Brown was able to come up with by himself were quite simple and unsophisticated, leading to rather simple (uninteresting) rules -- fine for illustration but not adequate for real analysis. I feel that if we had been able to interact more directly he would have been forced to delve more deeply into the actual mental processes he uses and the rules obtained would have been much more interesting.

We also had to decide on the type of analysis that Claud would actually do. Cladistics normally is done in three stages: gathering data, interpreting it and doing statistical analysis. The first stage, gathering data, is essentially a literature/laboratory project, and is not really what we expect of an expert system. The statistical process is either cluster analysis or a variation of it, and can be done by standard packages (e.g., Genstat). Although this could be included in Claud, it would be awkward and redundant.

Thus it was decided to limit Claud's abilities to those involved in the analysis of data. This would include rules to interpret the data for non-experts (i.e., telling them how to correctly set it up for the statistical analysis) and data-base manipulation capabilities for the

expert, so she could see how various assumptions would change the relationships among characters, otus, etc.

Finally, we had to choose the language in which to implement the system. At RIT we have both LISP and Prolog availible, but do not have any of the development tools that have been written to support expert system writing in LISP. We also have more or less a wide open field -- not many expert systems have been written here.

We decided to use Prolog for several reasons. Without the LISP development tools, the simplicity of Prolog programming was attractive. Also, relatively few expert systems have been written in Prolog, so we would not just be reinventing the wheel. That is, there are many basic procedures that must be used in any system, and these have been well studied in LISP. However, since we did not have access to them we would have to start from scratch. Writing these in Prolog would be more in the nature of breaking new ground, and would also be useful to any who followed us and needed such procedures for use in more fully developed systems.

Another strong argument for using Prolog was that the form of cladistic analysis naturally suggests the use of a rule-based system. These systems contain rules, data and a way to manipulate them (inference engine), all of which are relatively easy to program in Prolog. The rules may have similar or varying formats, and the inference engine can be general or quite specific, depending upon how general (adaptable to other uses) you wish to make the system. As cladistics is a rather specific (esoteric?) branch of biology, the rules would probably be in formats specific to the field, but it would be worthwhile to try and make the manipulative procedures as general as possible.

After having made this decision, I discovered an interesting situation. I have access to a Prime computer at the New York State Agricultural Experiment Station in Geneva, a branch of Cornell University. This computer has only three languages availible on it: APL, FORTRAN and Salford LISP/Prolog. The latter is an interpreter/compiler written at Salford University, England, that allows one to program in LISP and Prolog interchangably -- in

fact, one can even insert compiled FORTRAN code in the program. This meant that I could code in Prolog and still use LISP commands whenever they were more convenient (i.e., input/output.) The Prolog differs slightly from the C-Prolog available at RIT (see below), but has the same basic capabilities.

In addition to the language for an expert system, there was also someone at the station who had actually written part of such a system. Although the system was not fully operative (the expert died as they were writing it), it was extremely helpful to be able to discuss the problems involved in creating such a system with someone who had actually done it.

6.2. Details.

Having decided on the scope of the system, I still had some details to work out. For example, how would the user interact with the system? Although I don't like using menu-driven systems, it seemed that this form was most appropriate for the non-expert part of the system. Basically, there is a lot of information that the system can handle, but the number of choices to be made as to the manner of handling is quite limited and perscribed. Thus there is no need to offer different formats and extensive options -- a choice from a list will do fine.

When being used by an expert, the system could "discuss" his options by asking a few simple yes-no or fill-in-the-blank type of questions. It could then offer options, advice or conclusions, depending upon which was required.

It was decided to find a general way to deal with multiple choice type questions (and appropriate actions.) This one form could service multiple choice, menu and yes-no type questions. Fill-in-the-blanks (e.g., "What otu do you wish to change?") would be handled on an individual basis. This results in all questions of the first three types having the same format and processing, while those of the fourth type could differ in form and treatment.

The next decision was prompted by the awkwardness of the Prime editor. In fact it is virtually unusable -- I used the word processor (Muse) to create my files. The system would be created as a series of small files, with the first file consulting the others to form the com-

plete system. The system presently consists of six files:

(1) The main file, called Claud, which consulted the other files and asked if the user wished to employ the expert or non-expert aspect of the system;

(2) A file containing the multiple choice question processing rules;

(3) A file, called oddsandends, containing other useful 'tools', such as a general exit routine, printing predicates, etc.;

(4) A file that handles input. In general, the system will accept input from files that contain only data, but stores the information as relationships which are predicates in Prolog. Thus some processing of input data is required;

(5) A file that contains the heuristics to be used by the non-expert; and

(6) A file that contains the data base manipulation procedures for use by the expert. This file references the previous one, so the expert can also access the built-in rules.

Although not all expert systems contain one, I always planned to have Claud include an explanation facility. How to do this was worked out rather late in the process (see below), but when to explain actions was decided early on. It seemed that explanations were most useful when something didn't work as expected; when it did work only the results were required (have I been using UNIX too long?) So it was decided to include, with each rule, the ability to explain why it did not function if, indeed, it did not. The explanations were to be as specific as possible -- the "segmentation fault -- core dumped" syndrome was to be avoided as much as possible.

Finally, the decision as to how much of the other languages availible to me -- LISP and FORTRAN -- were to be used had to be made. The fact that no statistical calculations were to be included meant that FORTRAN was not really needed. Although this was not decided entirely until Claud was done, there turned out to be only one occasion where LISP was preferred to Prolog -- input from the keyboard. Salford Prolog requires a period after all input (or a space and a period after numeric input), a rather pointless limitation. Thus the LISP

procedure 'READ', which does not have this requirement, was used. No special coding (other than capitals and single quotes) was required to intermix the languages.

7. Elements of the System.

7.1. Tools.

The first priority in writing Claud was to create a set of generally useful tools that would simplify writing the rest of the system. As mentioned above, I had the opportunity of speaking to a person who had already written an expert system in Prolog, so I had a fair idea of what was needed. Another reason for doing this first was that code for these procedures tends to be fairly simple, so that this offered a chance to perfect (or at least improve) my own abilities in writing Prolog code.

In Prolog, the order in which the various constructs are executed depends upon the success or failure of surrounding constructs. Success allows you to proceed forward while failure causes backtracking -- how much and how far can be controlled, to some extent, by the use of the cut (!). This variability in execution order is one of the special features that makes Prolog so suitable for Artificial Intelligence applications. Program execution sequence is not always under the control of the programmer, so that the variations possible are much more complex than those usually achievable with a simple if-then-else structure.

However, sometimes it is necessary to be able to predict whether a rule or predicate will succeed or not. If failure is desired, a fail predicate can be included in the rule. However, it is not so simple to guarantee success. Thus I wrote my first rule, which assures that a given task succeeds:

```
accomplish(Task) :-
        Task ; true.
```

The semi-colon is an "or", so either the task or true, which is always true. As an aside, we can notice Prolog's rather split personality, where fail is 'fail', but succeed is 'true.' Note that this rule certainly meets the goal of starting with simple code.

Next we needed the ability to easily output phrases. Prolog has the standard predicate 'write', but it is rather limited. In its more general form it accepts two arguments: something to output and the stream to output it on. If the second argument is omitted it is assumed to

be 0, the standard output stream. Thus the following predicates were written, roughly correspond to the Pascal "write" and "writeln" commands:

```
prln(List) :-
        prln(0,List),!.

prln(Unit,List) :-
        pr(Unit,List), nl(Unit),!.

pr(List) :-
         pr(0,List).

pr(Unit,[]).

pr(Unit,[First|Rest]) :-
        write(First,Unit),
                pr(Unit,Rest),
                !.
```

To output a phrase, invoke "pr(['phrase'])." To output phrases and variables, use, for example, "prln([X,' is before ',Y])."

I felt it would be useful to be able to end the program at any time, saving the database. Thus I wrote the rule 'byebye' which dumps the data into four files (char, otu, ances and haschar) and returns the user to the Prolog system ('abort' in Salford Prolog.) Of course, these files contain predicates, not just raw data, and so may simply be consulted to be read in to a new run of the program.

All input by the user is done through the procedure 'readin.' This is so that, at any time, the user can terminate the current program run by typing the letter e.

```
readin :-
        X is 'READ'(),
        (X = e,
        byebye;
        true).
```

READ is a LISP procedure directly usable in Salford Prolog.

Finally, a way to count things such as predicates (i.e., how many instances of type "name" currently exist in the database) was needed. It turns out that this is not so simple, as there are no true variables in Prolog -- statements such as X = X + 1 are not legal. In fact, counting declared predicates can be done recursively, but general counting can not be done

this way. I was stumped by this for a while until Prof. Biles suggested the approach of using the data base as a place to keep track of things. Thus we can declare the predicate count(X), where X is the magnitude of the count.

For counting in general we need two abilities: zero the count and increase the count. Thus we define two new rules to allow us to do this.

```
clearcount :-
        abolish(count,1),
        assert(count(0)).

upcount :-
        count(X),
        abolish(count,1),
        Y is X + 1,
        asserta(count(Y)).
```

The count can be determined at any time by calling count(Value).

We particularly want the ability to count how many instances of a certain predicate occur in the data base. To do this we use the built-in predicate 'functor.' Functor(A,B,C) declares A to be a structure with functor B and number of arguments C. Thus, supplying B and C defines A.

```
countpred(Pred,Arity,Count) :-
        functor(Name,Pred,Arity),
        clearcount,
        accomplish(countpred(Name)),
        count(Count).

countpred(Name) :-
        (Name,upcount),
        fail.
```

Countpred/3 works by declaring the passed predicate, using 'functor,' zeroing the count, counting by means of countpred/1 and returning the value in Count. Note how countpred/1 works. If a predicate of the given name is found, we execute upcount and then fail. This failure causes, by backtracking, the next predicate of the same name to be searched for. If not found, the whole rule returns a failure. But countpred/1 is 'accomplished' in countpred/3, so it is seen as a success and countpred/3 continues.

7.2. Multiple Choice Question Handling.

Next, it was decided to implement the clauses that handled multiple choice types of questions. Discussions with my "Expert Systems Expert" led to the idea of having a multiple choice operator, so that each question could have an associated question identifier. The question itself would consist of a list of prompts and resultant actions. The first would be the main prompt that headed the question and the rest would represent the choices.

Prof. Biles recommended that the questions consist of a list of items, where each item consisted of a string (to be displayed) and a resulting action. This made the form of the question clearer and the numbering of the choices easier. Thus a typical question looks like:

```
mainmenu multiplechoice
        ([item(['Do you wish to:'],
                dummy),
        item('have Claud give you advice',
                advisor),
        item('discuss with Claud your problem',
                discussion),
        item('end this session',
                byebye)]).
```

This will be displayed as :

```
Do you wish to:
1) have Claud give you advice;
2) discuss with Claud your problem;
3) end this session.

Enter choice:
```

Notice that the first item, the header, is different from the latter items, the choices.

The clauses used to display the question and act on the response are shown in Fig. 1. The rule 'multchoice' displays the general prompt, takes any initial action if one is desired, displays the choices, prompts for a response and takes the action the response requests. Note that the form of the actual rule is different from that just described. There are two reasons for this.

First, in general there will be no initial action desired (in the example question given above the action was "dummy" -- i. e., do nothing.) Thus this action is identified by multchdecompose and then 'accomplished,' so that even if the predicate/rule invoked doesn't

```
 :- mode('prv2').

/*   Clauses to take care of multiple choice questions.  These should
/*   have the form:
/*   questionid multiplechoice
/*       [item(['initial prompt'], initial action(set vars, etc)),
/*       item('choice 1', action resulting from first choice),
/*       ...].
/*   or initial item may be replaced by the single word "noprompt".
/*   Multchoice does the initial steps (if they are there) and
/*   calls other clauses to process the question.  READ is a Lisp
/*   function that reads characters without the necessity of
/*   typing a period after them.

multchoice([noheader¶Question]) :-
     repeat,
         multchdisplay(Question,1),
         pr(['enter choice: ']),
         readin(X),
         multchaction(Question,X),     /*fails on invalid response
         nl.

multchoice(Question) :-
     multchdecompose(Question,Prompt,Task,Rest),
     accomplish(Task),
     repeat,
         prln(Prompt),
         nl,
         multchdisplay(Rest,1),
         pr(['Enter choice: ']),
         readin(X),
         multchaction(Rest,X),          /*fails on invalid response
         nl.


/*
/*   Takes a list (eg, of questions and actions) and decomposes
/*   it to the first item, second item and the rest of the list.
/*

multchdecompose([First¶Rest],Prompt,Task,Rest) :-
     multchdecompose(First,Prompt,Task).

multchdecompose(item(Prompt,Task),Prompt,Task).

/*
/*   Displays the various choices, numbering them.
/*
```

Fig 1-1

```
multchdisplay(Questions,Number) :-
     multchdecompose(Questions,Lastques,Lasttask,[]),
     prln([Number,') ',Lastques,'.']),
     nl,
     !.

multchdisplay(Questions,Number) :-
     multchdecompose(Questions,Nextques,Nexttask,Rest),
     prln([Number,') ',Nextques,';']),
     Numplus is Number + 1,
     multchdisplay(Rest,Numplus),
     !.

/*
/*   These clauses perform an approprate action after
/*   the response is read by the READ function.
/*   The first clause deals with an invalid response, causing a
/*   prompt to be printed and the question to be redisplayed.
/*   Length is a built in function that determines the length
/*   of a list.  The 'get' clauses get the appropriate
/*   command from the question list and return it to be executed.
/*   Note that the command is not "accomplished", thus it can
/*   fail, which causes an error message to be printed
/*   and an alternate selection requested.
/*

multchaction(Queslist,Selection) :-
     length(Queslist,Length),
     Numques is Length,
     (Selection < 1 ; Selection > Numques),
     nl,
     prln(['Invalid Selection -- try again']),
     nl, nl,
     fail.

multchaction(Queslist,Selection) :-
     length(Queslist,Length),
     Numques is Length,
     Selection >= 1,
     Selection =< Numques,
     multchget(Queslist,Selection,Command),
     (Command;nl,prln(['Selection inoperative -- choose again']),nl,fail),
     !.

multchget(Queslist,Place,Command) :-
     multchgetcom(Queslist,Place,[item(Ques,Command)¶Rest]).

multchgetcom(List,1,List) :- !.
```

Fig 1-2

```
multchgetcom([First¶Rest],Number,Dummy) :-
     Num2 is Number - 1,
     multchgetcom(Rest,Num2,Dummy).
```

Fig 1-3

exist the rule continues. Second, one of the choices may have an action associated with it that fails -- e.g., in the design phase one option is not yet implemented. Failure here should result in an error message requesting a different choice and redisplay of the question. Thus the actions in the 'items' are not 'accomplished,' but can lead to failure. An error message is then printed by 'multchaction' and the form of 'multchoice' causes the question to be redisplayed.

The clause 'multchdisplay' prints out the string portion of each item, suitably numbered. It also illustrates Prolog's selection process. Any Artificial Intelligence language has to deal with the situation when the programmer desires different actions depending on the form or content of the parameters passed to a procedure, rather than the chioce of the procedure. Prolog selects the correct rule positionally -- it tries the first correctly named rule in the data base for parameter match, then the second, etc. In our case, when the list representing further choices is empty we are on the last item and want special handling -- e. g., no more recursion. The final cut is necessary to prevent reentry on backtracking in case of failure in 'multchoice.'

The list of items and the users choice are passed to 'multchaction' which then executes the appropriate action. The first clause named multchaction determines the length of the list (the number of possible choices) and generates an error message if the choice is impossible (too large or too small.). It then fails, so that the second clause of this name is always entered.

This clause fails immediately if the choice is impossible. If it is a legal choice, 'multchget' goes down the list to the appropriate item and returns the desired action in the variable Command. Then either this command is successfully executed or a "selection inoperative" error message is generated and the rule fails.

7.3. File Handling.

In general, the input to Claud will come from four previously created files. These files contain only data -- strings or numbers -- and must be read into the system, and the data

```
filehandler :- charlist,otulist,ancestorlist,hascharacter,
     multiplechoice(seehaschar,Question),
     multchoice(Question).

charlist :-
     nl,
     pr(['Enter the name of the character file:  ']),
     readin(Name),
     fileopen(Name,4,read),
     for(Num,1,1000,1),
         read(Char,'LINE',4),
         assertit(Num,Char,char),
     closeunit(4),
     nl,prln(['Characters are:']),nl,
     printit(char).

otulist :-
     nl,
     pr(['Enter the name of the otu file:  ']),
     readin(Name),
     fileopen(Name,4,read),
     for(Num,1,200,1),
         read(Otu,'LINE',4),
         assertit(Num,Otu,otu),
     closeunit(4),
     nl,prln(['Otus are:']),nl,
     printit(otu).

ancestorlist :-
     nl,
     pr(['Enter the name of the ancestral character file:  ']),
     readin(Name),
     fileopen(Name,4,read),
     for(Num,1,1000,1),
         read(Char,'LINE',4),
         assertit(Num,Char,ances),
     closeunit(4),
     nl,prln(['Ancestral characters are:']),nl,
     printances.

printances :-
     ances(Z,X),
     char(X,Y),
     prln([X,') ',Y]),
     fail.

printances :- nl.
```

Fig 2-1

```
hascharacter :-
     nl,
     prln(['Enter the name of the file that ']),
     pr(['relates otus and characters: ']),
     readin(Name),
     fileopen(Name,4,read),
     for(Num,1,1000,1),
        Otu is 'READ'(4),
        asserthasc(Otu),
     closeunit(4).

asserthasc(Otu) :-
     Otu ®= 'END_OF_FILE$',
     Char is 'READ'(4),
     isitances(Otu,Char),
     !,fail.

asserthasc(_).

isitances(Otu,Char) :-
     ances(_,Char),!,
     assertz(haschar(Otu,Char,0)).

isitances(Otu,Char) :-
     assertz(haschar(Otu,Char,10)).

seehaschar multiplechoice
     ([item(['Do you wish to see the otu/character entries?'],
         dummy),
     item('yes',
         display),
     item('no',
         prln([]))]).

display :-
     nl,
     prln(['OTU              Character - (A) means ancestral']),
     nl,
     haschar(Otu,Charnum,Code),
     char(Charnum,Char),
     otu(Otu,Name),
     pr([Name]),tabulate(17),pr([Char]),
     ((Code = 0,pr([' (A)']));prln([])),
     fail.

display :- nl.

/*
```

Fig 2-2

```
/*   Asserts strings(or any other terms) read from a file and fails.
/*   Suceeds when end of file is reached.
/*
assertit(Num,String,Name) :-
     String ®= 'END_OF_FILE$',
     assertz(Name(Num,String)),
     !,fail.

assertit(_,_,_).

/*
/*   Prints out all occurances of a predicate / 2
/*   passed to it in Name(eg, character list, etc).
/*

printit(Name) :-
     call(Name(X,Y)),
     prln([X,') ',Y]),
     fail.

printit(_) :- nl.
```

Fig 2-3

entered into the database as predicates. The clauses to do this are given in Fig 2. The careful reader will notice that in these rules similar functions are sometimes coded differently. This is partly because I wanted to try different things in Prolog and partly because I didn't always recognize that the code could be the same, and so rewrote it.

The files containing the character list and the otu list are similar, each containing one string per line. They should be entered in the data base as predicates containing a number and the string, e.g., char(3, 'big nose') or otu(7,'mother-in-law'). We can see how this is done by looking at the clause 'charlist.'

First, a prompt requests the file name. Then an input stream from that file is opened, enabling us to read from it. The next two actions depend upon special features of Salford Prolog.

Remember that this version of Prolog is closely related to other languages, including FORTRAN. In fact, it contains a structure similar to a FORTRAN 'do' loop. 'For(A,b,c,d) sets A equal to b and continues execution. If a failure is encountered, you return to the 'for' statement (like a 'repeat'), increment A by d and compare it to c. If it is less than or equal to c then you resume execution; if it is greater than c then you continue backtracking. Thus by controlling failures you can loop a certain number of times, and even use A as a counter for the number of times through the loop. We could achieve the same effect in C-Prolog using 'repeat' and 'upcount', but it would take considerably more effort.

A second feature of Salford Prolog is that all files in the system end with a special marker called 'END-OF-FILE$', which may be read. Thus testing if your input variable has this value allows you to identify when you come to the end of a file.

Returning to 'charlist', we can now see that the next part is a loop that reads the next line of the file and calls 'assertit,' passing the line and the count. 'Assertit' has two parts. The first part fails immediately if the line passed consists of the end of file marker. This causes Prolog to attempt the second clause named 'assertit' which always succeeds, breaking out of the loop.

If the line does not represent the end of the file, the appropriate predicate is asserted into the data base. The clause then fails, but the failure is after a cut so that this time the second clause is not attempted and a failure is returned, causing the loop to increment.

Finally, 'printit(char)' is invoked to list the characters on the screen as an error check. As 'char' is passed to the clause in the variable Name, it cannot be directly invoked, but must be 'called.' This is because on the Prolog parsing pass Name is not instantiated, and so is not a legal predicate name.

The clause 'ancestorlist' accepts a file that is a list of the character numbers of ancestral characters, one on a line, and declares predicates of the form ances(5,7), where the first number is a counter and the second is the number of the ancestral character. It is similar to charlist and otulist with the exception that when you print the list of ancestral characters it references the 'char' predicates to get the appropriate strings.

Finally, we have the file that tells us which otu has which character. The file consists of two numbers per line, that of the otu and that of the character. They are to be added to the database as the predicate 'haschar.' This predicate has three arguments: otu number, character number and weight. The weight, presently, is 0 for ancestral characters and 10 for non-ancestral characters. Future development could allow the expert user to vary this number as certainty of data or interpretation warented.

Because the weight factor is included, the program must determine if the character is ancestral or not. Thus after reading the input line and checking for end of file we must also check if the character is ancestral. The clause 'isitances' does this and asserts 'haschar' with the proper weight. Also note that the printout of this file, handled by 'display,' is very long, so that the user is queried as to whether they wish to see it.

It should also be mentioned that when exiting the program the user can dump the data into four files, correspond to the four mentioned above. These files contain the actual predicates, and thus can be reread in by using the '(re)consult' standard predicate available in both Prologs.

7.4. Advising the Non-expert.

The clauses that allow Claud to advise the non-expert are contained in the file advise (Fig.3.) 'Advisor' presents a menu allowing data entry, proceeding or ending the session. If the user wishes to proceed he is asked to enter two character numbers. These characters are then compared, and if they are all the same in all otus the user is advised to group them. If they differ in one otu the advice is that they probably should not be grouped, while if they differ in two or more otus they definitely should not be grouped.

This advice represents the result of applying the rules (heuristics) given me by the expert in the field. Although the results, and the code thar produced them, are specific to this field, I am presenting the entire file to illustrate how I choose to implement the rules given me.

One decision I felt important was that the nature of the field suggested sequential application of a number of rules rather than a more random "use whichever rule fits best at this time" approach. Thus, as one can see in the clause 'group', the rules are all called each time rather than being allowed to "fire" when the system reaches a certain state. A special approach (explained below) allows us to bypass rules when so desired (e.g., when there is an error in a previous rule.)

Another decision, mentioned previously, was that the system should explain what went wrong in case it can't answer the question posed. Once again this was done by using the data base as a memory. Whenever an error occurred a predicate called 'explain' was added to the data base. These predicates have the form 'explain(string)', where the string contains the name of the character(s) causing the problem and an explanation of what the problem is. The last thing done by 'advise' is to print out any 'explain' predicates and then remove them from the data base so as to be ready for the next run.

One more point should be considered. Lest we completely identify rules with expert-derived heuristics, let me point out that all programmers are familiar with one important rule -- bad input should not be processed. Basic error-checking should be a part of the program

```
/*
/*
/*   advisor is a clause that makes CLAUD act as an expert --
/*   ie, CLAUD advises the non-expert user.
/*

advisor() :-
     multiplechoice(advmenu1,Menu1),
     nl,
     repeat,
     advisor1(Menu1),
     fail.

advisor1(Menu1) :-
     multchoice(Menu1),
     cleanup,
     advisor2,!.

advisor2 :-
     prln(['Input the numbers of the groups you wish']),
     prln(['to compare -- eg, 1 2']),
     group,
     explaination,
     !.


group :-
         readin(X),
         readin(Y),
         checkchar(X),
         checkchar(Y),!,
         firstrule(X,Y),
         secondrule(X,Y),
         thirdrule(X,Y).

/*
/*   Checkchar makes sure the characters are legal.
/*

checkchar(X) :- char(X,_);fail.

checkchar(X) :-
     assertz(explain([X,' is an illegal character number.'])),
     assert(code(ng)).

/*
/*   Firstrule checks if the characters are ancestral.
/*
```

Fig 3-1

```
firstrule(X,Y) :- code(ng).

firstrule(X,Y) :-
     accomplish(firstrule1(X)),
     accomplish(firstrule1(Y)).

firstrule1(X) :-
     ances(_,X),!,
     char(X,Z),
     assertz(explain(['Char ',Z])),
     assertz(explain([' is ancestral -- can not be grouped'])),
     assert(code(ng)).

/*
/*   Secondrule checks that the 2 characters have the
/*   same ancestral character.
/*

secondrule(X,Y) :-
     code(ng).

secondrule(X,Y) :-
     findances(X,Xances),
     findances(Y,Yances),
     (Xances = Yances;
         diffances(X,Xances,Y,Yances)).

findances(Num,Ances) :-      /* Finds ancestral char for given char
     ances(_,Num),!,
     Ances = Num.

findances(Num,Ances) :-
     Num1 is Num - 1,
     findances(Num1,Ances).

diffances(X,Xances,Y,Yances) :-        /* If chars have different ancestors.
     char(X,X1),
     char(Xances,Xan),
     char(Y,Y1),
     char(Yances,Yan),
     assertz(explain(['Character ',X,', ',X1])),
     assertz(explain(['has ancestor ',Xan,','])),
     assertz(explain(['while character ',Y,', ',Y1])),
     assertz(explain(['has ancestor ',Yan,'.'])),
  assertz(explain(['Characters with different ancestors may not be grouped.'])),
     assert(code(ng)).

/*
```

Fig 3-2

```
/*   Third rule checks if all otus share the characters, if one otu
/*   has only one of the characters or if more than one otu has
/*   only one of the characters and gives appropriate advice.
/*

thirdrule(X,Y) :- code(ng).

thirdrule(X,Y) :-
     haschar(Otu,X,_),
     thirdrule1(Otu,Y).

thirdrule(X,Y) :-
     haschar(Otu,Y,_),
     thirdrule1(Otu,X).

thirdrule1(Otu,Y) :-
     not(haschar(Otu,Y,_)),
     assertz(bad(Otu)),!,
     fail.

thirdrule(X,Y) :-
     countpred(bad,1,Count),
     intro(X,Y),
     thirdrule(Count).

intro(X,Y) :-
     char(X,X1),char(Y,Y1),
     assertz(explain(['Characters ',X1])),
     assertz(explain(['and ',Y1])),
     assertz(explain(['differ in the following otus:'])).

thirdrule(0) :-
     assertz(explain(['none.'])),
     assertz(explain(['Thus they may be grouped.'])).

thirdrule(1) :-
     bad(W),
     otu(W,Z),
     assertz(explain([Z])),
     assertz(explain(['They probably should not be grouped.'])).

thirdrule(_) :-
     listbad,
     assertz(explain(['They should not be grouped.'])).

listbad :-
     bad(X),
     otu(X,Y),
```

Fig 3-3

```
     assertz(explain([Y])),
     fail.

listbad.


advmenu1 multiplechoice
     ([item(['Do you wish to enter data from:'],
         dummy),
     item('a file',
         filehandler),
     item('files previously dumped from this program',
         getem),
     item('the terminal',
         dummy),
     item('or do you want to not enter more data',
         prln([])),
     item('or do you wish to end the session',
         byebye)]).

/*
/*   Explaination facility.  As rules fire, they also may, if needed,
/*   assert explainations in the predicate "explain".  This causes any
/*   such assertations to be printed.
/*
explaination :-
     nl,
     explain(X),
     prln(X),
     fail.

explaination :- nl.

/*
/*   Removes asserted predicates.
/*
cleanup :-
     abolish(explain,1),
     abolish(code,1),
     abolish(bad,1).
```

Fig 3-4

as well as sophisticated rule application. In our case this means, for example, the input character number should correspond to the number of a character in the data base. The clause 'checkchar' does this for us.

Notice how 'checkchar' works. If the character exists it succeeds and the program continues. If it does not, the first instance of 'charcheck' fails (belt and suspenders) and the second 'charcheck' is invoked. This asserts two predicates into the data base; an explanation of the failure and the predicate 'code(ng).' All following rules check for the presence of 'code(ng)' in the data base before they do anything else. If it is there, it indicates an error, so they (the rules) do not continue processing.

The goal of 'advisor' is to determine if two characters can be grouped or treated as a single character. The decision is based on three rules supplied by my cladistics expert: 1)ancestral characters may not be grouped with any other character, ancestral or not; 2) characters with different ancestors may not be grouped; and 3) characters that differ in more than one otu should not be grouped. Characters that satisfy points 1 and 2 and differ in one otu probably should not be grouped, while characters that differ in no otus may be grouped. "Differ in an otu" means that one of the otus in the data base has the character while another does not.

These rules are embodied in the clauses named 'firstrule', 'secondrule' and 'thirdrule.' 'Firstrule' is quite straightforward. Note that if 'code(ng)' exists in the database the rule will essentially not function.

'Secondrule' depends upon the organization of the character file from which the data base was built. This file must have the following form: ancestral character, characters derived from that character, next ancestral character, etc. Thus you can find the ancestor of any given character by looking at sucessively lower numbered characters until you find one that is ancestral (as indicated by the 'ances' predicate.) In other words, given character number 10 you would look at character number 9, number 8 ... until you found the first character that was ancestral, which would then be the ancestor of 10. To be absolutely safe you

might want to start with character 10 instead of 9, although rule one should make this unnecessary.

'Secondrule' determines the ancestor of both characters and sees if they are the same. If not, 'code(ng)' and an explaination are asserted into the data base.

'Thirdrule' compares the otus in regards to the characters under question. If each otu either has or does not have both characters it is a match; if an otu has one character but not the other there is a mismatch. It is complicated a bit by the fact that not possessing a character is not directly indicated in the data base. Rather it is indicated by the absence of the corresponding 'haschar' predicate.

Thus I had to adopt the following, rather convoluted, logic: 1) check if the first character is possessed by an otu; 2) if it is check if the otu also possesses the second character; 3) check if an otu possesses the second character but not the first. In both cases, if the otu has only one of the characters, assert it in the predicate 'bad.' When done, count the number of 'bad' predicates and offer advice and explanation based on this number (0, 1 or more than 1.)

Finally, after all is done as far as interaction with the user is concerned, you should remove all the declared predicates from the data base. The clause 'cleanup' takes care of this housekeeping duty.

7.5. Interaction with the Expert.

Fig 4 shows the file discuss, which is designed to be used by the more expert user. This file allows manipulation of the data base by the user using the rule 'changeancs' and calculation of a similarity coefficient between otus, which is the sum of the weights of characters possessed in common by both otus ('dosim'.)

Discuss gives the expert user the power to change which character is considered ancestral, an important tool in the operation of cladistic analysis. To do this we must do several things:

```
/*
/*   Discussion allows Claud to "discuss" things with
/*   a more expert user.  It allows the user to manipulate
/*   the data base in certain ways, and then see how that
/*   effects the conclusions.
/*

discussion :-
     multiplechoice(dismenu1,Dismenu),
     repeat,
         discussion1(Dismenu),
         fail.

discussion1(Dismenu) :-
     multchoice(Dismenu),
     !.

dismenu1 multiplechoice
     ([item(['Do you wish to:'],
         dummy),
     item('Enter data from dumped files',
         getem),
     item('Enter data from other files',
         filehandler),
     item('Enter data from the terminal',
         dummy),
     item('Change the characters that are ancestral',
         changeances),
     item('Check groupings',
         advisor2),
     item('Determine the Similarity Coef. of 2 otus',
         dosim),
     item('Leave the program',
         byebye)]).

changeances :-
     repeat,
         pr(['Input the current ancestral character: ']),
         readin(X),
         nl,
         checkances(X),
     repeat,
         pr(['Input the current derrived character: ']),
         readin(Y),
         nl,
         checkderr(X,Y),
     switchthem(X,Y).
```

Fig 4-1

```
checkances(X) :-
     (ances(_,X);
     (prln([X,' is not currently an ancestral character']),
         nl,fail)),!.

checkderr(X,Y) :-
     findances(Y,Yances),
     (X = Yances;
     (prln([X,' is not the ancestor of ',Y,'.']),
     fail)).

switchthem(X,Y) :-
     accomplish(changechar(X,Y)),
     accomplish(changehaschar(X,10)),
     accomplish(changehaschar(Y,0)).

changechar(X,Y) :-
     char(X,Z),
     char(Y,A),
     retract(char(X,Z)),
     asserta(char(Y,Z)),
     retract(char(Y,A)),
     asserta(char(X,A)).

changehaschar(X,Num) :-
     retract(haschar(Otu,X,_)),
     asserta(haschar(Otu,X,Num)),
     fail.

dosim :-
     repeat,
         pr(['Input the first otu: ']),
         readin(Otu1),nl,
         checkotu(Otu1),
     repeat,
         pr(['Input the second otu: ']),
         readin(Otu2),nl,
         checkotu(Otu2),
     sumthem(Otu1,Otu2).

checkotu(Otu) :-
     otu(Otu,_);
     (prln(['Not a legal otu.']),fail).

sumthem(Otu1,Otu2) :-
     otu(Otu1,X),otu(Otu2,Y),
     prln(['Otus ',X,' and ',Y,' share']),
     prln(['the following characters (weights):']),
```

Fig 4-2

```
     asserta(sum(0)),
     haschar(Otu1,Char,Wt),
     shared(Char,Otu2).

sumthem(Otu1,Otu2) :-
     retract(sum(X)),nl,
     prln(['Similarity Coef is ',X,'.']).

shared(Char,Otu2) :-
     haschar(Otu2,Char,Wt),
     retract(sum(X)),
     Y is X + Wt,
     asserta(sum(Y)),
     char(Char,C),
     (Wt = 0, prln([C,' (A)']);
      prln([C, ' (',Wt,')'])),
     !,fail.
```

Fig 4-3

(1) Make sure the characters are entered correctly -- one is ancestral and is the ancestor of the other ('checkances' and 'checkderr');

(2) due to the special ordering of the character file, we must interchange their numbers in this file ('changechar');

(3) We must change the weights in the 'haschar' predicates, interchanging 0 and 10 ('changehaschar', called twice with the appropriate weight passed to it.)

To calculate the similarity coefficient we have to check that the otus are legal, find their shared characters and sum the weights ('shared'.) Once again a summation is done by asserting and retracting a predicate in the data base, this time called 'sum.' 'Sum' is removed from the data base before we are finished.

8. Conclusion

8.1. Evaluation.

In the large sense I accomplished what I set out to do -- implement a (very) simple expert system in the field of cladistics. Let us now look more closely at the specific goals previously established and how well they have (or have not) been satisfied.

One goal that was not met was to function extensively as a "knowledge engineer," interacting with the expert, helping him specify the actual knowledge and thought processes he uses in his profession, and then putting these in computer-usable terms. In fact, this failure was so obvious that it made me reconsider the exact role of the knowledge engineer, a point I will return to in the next part of this section.

Another area that was dealt with in a less than totally satisfactory manner had to do with uncertainty and weighting factors. As in many other expert systems, provision was made for this by the use of a numerical factor, supplied by the user, and propagated, for one step only, by addition. (In other systems multiplication or an arbitrary function may also be used.) There is no real evidence that this is adequate or even usable. Also, the system does nothing with the result, other than report it to the user. This is conceptually adequate, as the user of this section is supposedly also an expert, but practically results in the computer being used as a rather expensive hand calculator, hardly an innovative, efficient,or informative approach.

Most of the other goals of this project were realized to a reasonable extent. These include:

a) *Ease of program modification.* The separate file approach has kept the system fairly modular (as far as this term can be applied to Prolog) and it is easy to add new tools (for data manipulation) and rules to. Note that this would be done in different ways in various parts of the program. In the discussion with the expert part of the program this could be done by adding choices to the "dismenu1," which calls clauses. In the part used by the nonexpert, this involves defining and adding new rules. Notice that the way the program is currently writ-

ten, rules are called sequentially, so a new call would also have to be added. This approach was adopted because the rules were always invoked in order, if they were invoked at all. However, this could easily be changed by having the rules called in a loop which depends on a declared fact. This fact would initially direct the first rule to fire. Each rule would remove the fact and then declare a new one, directing what to do next (or to exit the loop.) This would allow all rules to have the same format while making addition of new ones easy and allowing non-sequential execution as an option.

b) *Ease of adding new data.* Claud functions in a field where there is a large amount of data. By having most input from files, it is easy to add new data (consult) or modify the existing data (reconsult) at any time. In addition, the data base does not have to be recreated from scratch at each session because file dumps are available.

c) *Functioning at two levels.* Although the system does not do anything very complicated, it does allow both non-expert and expert users to gain information from it (ie, it is unsophisticated on two levels.) The menu format can get a bit cumbersome and repetitious, but seems to allow the desired versatility. A plus would be some ability to control when the menus are displayed, to cut down on unnecessary interaction. Also, if one wandered around the system in a rather vague way, you could force several levels of recursion that you could never back out of. This was more a problem of misuse than use, and so was not really addressed.

d) *Encoding heuristics.* Prolog proved an excellent choice for ease of coding heuristics. Both its recursive nature and its limitations in handling numeric information were not problems. (Remember, if number crunching were necessary, Salford Prolog allows FORTRAN subroutines to be included in the code.) Indeed, having to use recursion for certain operations proved a benefit, as it forced me to think them through carefully. Prolog's input/output limitations were more frustrating, but once again Salford came to the rescue with LISP functions.

e) *Explanation facility.* The need for explanation became more obvious as I progressed, and the method choosen proved versatile and able to provide clear, specific messages. It also could be easily extended, if necessary. It seems this ability is required for psychological reasons as well as factual ones. As a user, it made me feel better to gain some insight into how the program dealt with my questions, especially when it was telling me how dumb they were.

In summary, I feel the major goals of this project were realized. Certainly, the one concerned with my learning about expert systems and their implementation in Prolog was. Although Claud is limited it is functional and could be expanded and prettied up. One final point -- I had a lot of fun doing this, along with the normal amount of frustration associated with a major programming effort in a new language.

8.2. Further work. This thesis has suggested to me the following areas for further development:

(1) *The role of the knowledge engineer.* As expert system building tools become more sophisticated, the knowledge engineer becomes less of a programmer and more of an expert in aiding people identifying and formalizing their thought processes. It is possible that, in the near future, you would not want, in places where you can use these tools, a person whose main training was in computer science in this position. Identification of the variations in this role and when the different approachs (program from scratch, use simple tools, use complex tools) should be employed would be highly valuable.

(2) *Uncertainty.* This is a wide open field, both in theory and practice. Investigations in specific cases would be very helpful, as there is no reason to feel a good general method will be discovered soon.

(3) *Explanation.* Extension of Claud in general is not recommended, as it functions in too specialized a field. However, beefing up the explanation facility as a model for other systems (ie, using Claud to develop a generally useful scheme) would be worthwhile. For example, optional explanation of requests that succeed could be added. Also, crea-

tion of an explanation file that keeps track of a session (perhaps with annotations) would also be very helpful. Finally, a multilevel system could be instituted, which allowed for as much (to a limit) explanation as the user needed to be available.

BIBLIOGRAPHY

(1) Bailey, D. *The University of Salford Lisp/Prolog System,* Salford, England, 1984.

(2) Brown, Richard L. and Roelofs, Wendell, "Pheromones and Evolutionary Relationships of Tortricidae," Ann. Rev. Ecol. Syst., *13,* 395-422 (1982).

(3) Buchanan, Bruce G. and Shortliffe, Edward H., *Rule-Based Expert Systems,* Addison-Wesley, Mass., 1984.

(4) Clark, K. L. and McCabe, F. G. "PROLOG: A language for implementing expert systems," Logic Programming, 1984.

(5) Clocksin, W. F. and Mellish, C. S., *Programming in Prolog,* Springer-Verlag, New York, 1984.

(6) Colgan, Patrick W., *Quantitative Ethology,* Wiley, New York, 1978.

(7) Cracraft, Joel, "Phylogenetic Models and Classification," Syst. Zool., *23,* 71-90 (1974).

(8) Foster, Edward, "Artificial Intelligence faces a Crossroads," Mini-Micro Systems, May, 1984.

(9) Hayes-Roth, Frederick, Waterman, Donald A. and Lenat, Douglas, eds, *Building Expert Systems,* Addison-Wesley, Reading, Mass., 1983.

(10) Hennig, Willi, "Phylogenetic Systematisc," Ann. Rev. Entolol., *10,* 97-116 (1965).

(11) Littleford, Alan, *A Mycin-Like Expert System in Prolog,* Intl Logic Prog. Conf., 1984.

(12) Negoita, Constantin V., *Expert Systems and Fuzzy Systems,* Benjamin/Cummings, Menlo Park, Calif., 1985

(13) Romesburg, H. Charles, *Cluster Analysis for Researchers,* Wadsworth, Belmont, Calif., 1984.

(14) Rose, Michael R. and Doolittle, W. Ford, "Molecular Biological Mechanisms of Speciation," Science, *220,* 157-162 (1983).

(15) Warren, David, Bowen, David, Byrd, Lawrence and Pereira, Luis, "C-Prolog User's Manual," 1985.

(16) Winston, Patrick Henry, *Artifical Intelligence, second ed,* Addison-Wesley, Reading, Mass, 1984.

(17) Wipke, W. Todd, Ouchi, Glenn I. and Krishnan, S., "Simulation and Evaluation of Chemical Synthesis -- SECS.", Artifical Intelligence, *11,* 173-193 (1978).

Appendix 1

Main and Oddsandends

```
 :- mode('prv2'),mode(negnum).

 :- consult(multiplechoice).
 :- consult(oddsandends).
 :- consult(fromfiles).
 :- consult(advise).
 :- consult(discuss).

go() :-
     nl,nl,
     multiplechoice(mainmenu,Menu),
     prln(['Session may be ended at any time by typing e.']),nl,
     repeat,
         multchoice(Menu).

mainmenu multiplechoice
     ([item(['Do you wish to:'],
         dummy),
     item('have CLAUD give you advice',
         advisor),
     item('discuss with CLAUD your problem',
         discussion),
     item('end this session',
         byebye)]).


 /*
 /*   Start CLAUD up.
 /*
  ? go.
```

```
 :- mode('prv2').
 :- mode(negnum).
/*
/*   Accomplish does a task and suceeds even if the task fails.
/*

accomplish(Task) :-
     Task;
     true.

/*
/*   Routines for writing to the terminal or a file, with or without
/*   a carrage return.  Modeled after Pascal, but uses 'pr' instead
/*   of 'write', as write is a reserved Prolog predicate.
/*

prln(List) :-
     prln(0,List),!.

prln(Unit,List) :-
     pr(Unit,List),
     nl(Unit),!.

pr(List) :- pr(0,List),!.

pr(Unit,[]).

pr(Unit,[First¶Rest]) :-
     write(First,Unit),
     pr(Unit,Rest),!.

/*
/*   Standard procedure for getting input.
/*

readin(X) :-
     X is 'READ'(),
     (X = e, byebye;
     true).

/*
/*   General logout message.
/*

byebye() :-
     nl,
     filedump(char,char),
```

```
     filedump(otu,otu),
     filedump(ances,ances),
     filedump(haschar,haschar),
     prln(['Data saved in files char, otu, ances and haschar.']),
     prln(['Session may be restarted by typing go.']),nl,abort.

/*
/*   Counting clauses.
/*

countpred(Pred,Arity,Count) :-
     functor(Name,Pred,Arity),
     clearcount,
     accomplish(countpred(Name)),
     count(Count).

countpred(Name) :-
     (Name,
     upcount),
     fail.


clearcount :-
     abolish(count,1),
     assert(count(0)).

upcount :-
     count(X),
     abolish(count,1),
     Y is X + 1,
     asserta(count(Y)).

/*
/*   Get data from dumped files.
/*
getem :- nl,reconsult([char,otu,ances,haschar]),
         prln(['Files char, otu, ances and haschar consulted.']),nl.


/*
/*   compile predicates
/*

 :- compile(pr/1).
 :- compile(pr/2).
 :- compile(prln/1).
 :- compile(prln/2).
 :- compile(accomplish/1).
```

```
/*
/*   Define operators.
/*

 :- op(1200,xfx,multiplechoice).
 :- op(1200,xfx,rule).
```

Appendix 2

Sample run

```
OK, prolog claud


Session may be ended at any time by typing e.

Do you wish to:

1) have CLAUD give you advice;
2) discuss with CLAUD your problem;
3) end this session.

Enter choice: 1

Do you wish to enter data from:

1) a file;
2) files previously dumped from this program;
3) the terminal;
4) or do you want to not enter more data;
5) or do you wish to end the session.

Enter choice: 1

Enter the name of the character file:  charfile

Characters are:

1) colliculum short, near ostium
2) colliculum long, curved
3) colliculum without denticles
4) colliculum with denticles
5) socius broad, apically rounded
6) socius sub-quadrate
7) socius apex curved dorsally
8) socius with dorsally curved apical hook
9) socius triangular
10) socius narrowed
11) socius with dorsal margin lobed basally
12) uncus simple, flat
13) uncus with lateral margins rolled ventrally
14) uncus with ventral keel
15) uncus cleft apically
16) uncus apically notched
17) forwing with pre-tronal triangular spot
18) forwing without pre-tornal triangular spot
19) forwing without apical streak
20) forwing with apical streak
21) forwing termen simple, concave
```

22) forwing termen emarginate
23) forwing termen convex
24) pre-papillae plate absent
25) pre-papillae plate present
26) pre-papillae plate reduced
27) anellar ring open
28) anellar ring closed around aedeagus
29) dorsal anellar plate with straight margins
30) dorsal anellar plate constricted laterally
31) saccular spine cluster present
32) saccular spine cluster reduced
33) sterigma ring short
34) sterigma ring long
35) female tergite VIII without scales
36) female tergite VIII with scales
37) female sternite VII smooth
38) female sternite VII with medial posterior depressions
39) female sternite VII with lateral posterior depressions
40) costal fold present
41) costal fold absent
42) costal sex scale line absent
43) costal sex scale line present
44) labial palpus third segment not reduced
45) labial palpus third segment reduced
46) pre-pupal diapause absent
47) pre-pupal diapause present
48) forwing with basal and radial streaks

Enter the name of the otu file: otufile

Otus are:

1) medioviridana
2) emarginana
3) solandriana
4) hopkinsana
5) madderana
6) cruciana
7) bigemina
8) kasloana
9) lindana

Enter the name of the ancestral character file: ancesfile

Ancestral characters are:

1) colliculum short, near ostium
3) colliculum without denticles
5) socius broad, apically rounded
12) uncus simple, flat
17) forwing with pre-tronal triangular spot
19) forwing without apical streak
21) forwing termen simple, concave
24) pre-papillae plate absent
27) anellar ring open
29) dorsal anellar plate with straight margins
31) saccular spine cluster present
33) sterigma ring short
35) female tergite VIII without scales
37) female sternite VII smooth
40) costal fold present
42) costal sex scale line absent
44) labial palpus third segment not reduced
46) pre-pupal diapause absent

Enter the name of the file that
relates otus and characters: hascharfile
Do you wish to see the otu/character entries?

1) yes;
2) no.

Enter choice: 1

OTU	Character - (A) means ancestral
medioviridana	colliculum short, near ostium (A)
medioviridana	colliculum with denticles
medioviridana	socius sub-quadrate
medioviridana	uncus simple, flat (A)
medioviridana	forwing with pre-tronal triangular spot (A)
medioviridana	forwing without apical streak (A)
medioviridana	forwing termen simple, concave (A)
medioviridana	pre-papillae plate present
medioviridana	anellar ring closed around aedeagus
medioviridana	dorsal anellar plate constricted laterally
medioviridana	saccular spine cluster present (A)
medioviridana	saccular spine cluster reduced
medioviridana	sterigma ring short (A)
medioviridana	female tergite VIII without scales (A)
medioviridana	female sternite VII smooth (A)
medioviridana	costal fold present (A)
medioviridana	costal sex scale line absent (A)

medioviridana	labial palpus third segment not reduced (A)
medioviridana	pre-pupal diapause absent (A)
emarginana	colliculum with denticles
emarginana	socius triangular
emarginana	socius narrowed
emarginana	uncus with ventral keel
emarginana	forwing with pre-tronal triangular spot (A)
emarginana	forwing without apical streak (A)
emarginana	forwing termen simple, concave (A)
emarginana	forwing termen emarginate
emarginana	pre-papillae plate present
emarginana	anellar ring closed around aedeagus
emarginana	dorsal anellar plate with straight margins (A)
emarginana	saccular spine cluster present (A)
emarginana	sterigma ring long
emarginana	female tergite VIII with scales
emarginana	female sternite VII smooth (A)
emarginana	costal fold present (A)
emarginana	costal sex scale line absent (A)
emarginana	labial palpus third segment not reduced (A)
emarginana	pre-pupal diapause absent (A)
solandriana	colliculum with denticles
solandriana	socius triangular
solandriana	uncus with ventral keel
solandriana	uncus apically notched
solandriana	forwing without pre-tornal triangular spot
solandriana	forwing without apical streak (A)
solandriana	forwing termen simple, concave (A)
solandriana	pre-papillae plate present
solandriana	anellar ring closed around aedeagus
solandriana	dorsal anellar plate with straight margins (A)
solandriana	saccular spine cluster present (A)
solandriana	sterigma ring long
solandriana	female tergite VIII with scales
solandriana	female sternite VII smooth (A)
solandriana	costal fold present (A)
solandriana	costal sex scale line absent (A)
solandriana	labial palpus third segment not reduced (A)
solandriana	pre-pupal diapause absent (A)
hopkinsana	colliculum with denticles
hopkinsana	socius triangular
hopkinsana	socius narrowed
hopkinsana	uncus with ventral keel
hopkinsana	uncus cleft apically
hopkinsana	uncus apically notched
hopkinsana	forwing without pre-tornal triangular spot
hopkinsana	forwing without apical streak (A)
hopkinsana	forwing termen simple, concave (A)

hopkinsana	pre-papillae plate present
hopkinsana	pre-papillae plate reduced
hopkinsana	anellar ring closed around aedeagus
hopkinsana	dorsal anellar plate with straight margins (A)
hopkinsana	saccular spine cluster reduced
hopkinsana	sterigma ring long
hopkinsana	female tergite VIII with scales
hopkinsana	female sternite VII with medial posterior depressions
hopkinsana	costal fold present (A)
hopkinsana	costal sex scale line absent (A)
hopkinsana	labial palpus third segment not reduced (A)
hopkinsana	pre-pupal diapause absent (A)
madderana	colliculum without denticles (A)
madderana	socius triangular
madderana	socius narrowed
madderana	uncus with ventral keel
madderana	uncus cleft apically
madderana	uncus apically notched
madderana	forwing without pre-tornal triangular spot
madderana	forwing without apical streak (A)
madderana	forwing termen simple, concave (A)
madderana	pre-papillae plate present
madderana	pre-papillae plate reduced
madderana	anellar ring closed around aedeagus
madderana	dorsal anellar plate with straight margins (A)
madderana	saccular spine cluster reduced
madderana	sterigma ring long
madderana	female tergite VIII with scales
madderana	female sternite VII with medial posterior depressions
madderana	costal fold present (A)
madderana	costal sex scale line absent (A)
madderana	labial palpus third segment not reduced (A)
madderana	pre-pupal diapause absent (A)
cruciana	colliculum with denticles
cruciana	socius triangular
cruciana	socius narrowed
cruciana	uncus with ventral keel
cruciana	uncus cleft apically
cruciana	uncus apically notched
cruciana	forwing without pre-tornal triangular spot
cruciana	forwing without apical streak (A)
cruciana	forwing termen simple, concave (A)
cruciana	pre-papillae plate present
cruciana	pre-papillae plate reduced
cruciana	anellar ring closed around aedeagus
cruciana	dorsal anellar plate with straight margins (A)
cruciana	saccular spine cluster reduced
cruciana	sterigma ring long

cruciana	female tergite VIII with scales
cruciana	female sternite VII with medial posterior depressions
cruciana	costal fold absent
cruciana	costal sex scale line present
cruciana	labial palpus third segment reduced
cruciana	pre-pupal diapause absent (A)
bigemina	colliculum with denticles
bigemina	socius apex curved dorsally
bigemina	socius narrowed
bigemina	uncus with ventral keel
bigemina	uncus cleft apically
bigemina	uncus apically notched
bigemina	forwing without pre-tornal triangular spot
bigemina	forwing without apical streak (A)
bigemina	forwing termen simple, concave (A)
bigemina	pre-papillae plate present
bigemina	pre-papillae plate reduced
bigemina	anellar ring closed around aedeagus
bigemina	dorsal anellar plate with straight margins (A)
bigemina	saccular spine cluster reduced
bigemina	sterigma ring long
bigemina	female tergite VIII with scales
bigemina	female sternite VII with medial posterior depressions
bigemina	costal fold absent
bigemina	costal sex scale line present
bigemina	labial palpus third segment reduced
bigemina	pre-pupal diapause present
bigemina	forwing with basal and radial streaks
kasloana	colliculum with denticles
kasloana	socius apex curved dorsally
kasloana	socius narrowed
kasloana	socius with dorsal margin lobed basally
kasloana	uncus with ventral keel
kasloana	uncus cleft apically
kasloana	uncus apically notched
kasloana	forwing without pre-tornal triangular spot
kasloana	forwing with apical streak
kasloana	forwing termen simple, concave (A)
kasloana	pre-papillae plate present
kasloana	pre-papillae plate reduced
kasloana	anellar ring closed around aedeagus
kasloana	dorsal anellar plate with straight margins (A)
kasloana	saccular spine cluster reduced
kasloana	sterigma ring long
kasloana	female tergite VIII with scales
kasloana	female sternite VII with medial posterior depressions
kasloana	costal fold absent
kasloana	costal sex scale line absent (A)

```
kasloana        labial palpus third segment not reduced (A)
kasloana        pre-pupal diapause present
kasloana        forwing with basal and radial streaks
lindana         colliculum long, curved
lindana         colliculum with denticles
lindana         socius apex curved dorsally
lindana         socius with dorsally curved apical hook
lindana         socius narrowed
lindana         uncus with lateral margins rolled ventrally
lindana         uncus with ventral keel
lindana         uncus cleft apically
lindana         uncus apically notched
lindana         forwing without pre-tornal triangular spot
lindana         forwing without apical streak (A)
lindana         forwing termen convex
lindana         pre-papillae plate present
lindana         pre-papillae plate reduced
lindana         anellar ring closed around aedeagus
lindana         dorsal anellar plate with straight margins (A)
lindana         saccular spine cluster reduced
lindana         sterigma ring long
lindana         female tergite VIII with scales
lindana         female sternite VII with lateral posterior depressions
lindana         costal fold absent
lindana         costal sex scale line absent (A)
lindana         labial palpus third segment not reduced (A)
lindana         pre-pupal diapause present
lindana         forwing with basal and radial streaks


Input the numbers of the groups you wish
to compare -- eg, 1 2
1 2

Char colliculum short, near ostium
 is ancestral -- can not be grouped

Do you wish to enter data from:

1) a file;
2) files previously dumped from this program;
3) the terminal;
4) or do you want to not enter more data;
5) or do you wish to end the session.

Enter choice: 4
```

```
Input the numbers of the groups you wish
to compare -- eg, 1 2
1 3

Char colliculum short, near ostium
 is ancestral -- can not be grouped
Char colliculum without denticles
 is ancestral -- can not be grouped

Do you wish to enter data from:

1) a file;
2) files previously dumped from this program;
3) the terminal;
4) or do you want to not enter more data;
5) or do you wish to end the session.

Enter choice: 4


Input the numbers of the groups you wish
to compare -- eg, 1 2
2 4

Character 2, colliculum long, curved
has ancestor colliculum short, near ostium,
while character 4, colliculum with denticles
has ancestor colliculum without denticles.
Characters with different ancestors may not be grouped.

Do you wish to enter data from:

1) a file;
2) files previously dumped from this program;
3) the terminal;
4) or do you want to not enter more data;
5) or do you wish to end the session.

Enter choice: 4


Input the numbers of the groups you wish
to compare -- eg, 1 2
6 7

Characters socius sub-quadrate
and socius apex curved dorsally
differ in the following otus:
```

```
medioviridana
bigemina
kasloana
lindana
They should not be grouped.

Do you wish to enter data from:

1) a file;
2) files previously dumped from this program;
3) the terminal;
4) or do you want to not enter more data;
5) or do you wish to end the session.

Enter choice: 4


Input the numbers of the groups you wish
to compare -- eg, 1 2
15 16

Characters uncus cleft apically
and uncus apically notched
differ in the following otus:
solandriana
They probably should not be grouped.

Do you wish to enter data from:

1) a file;
2) files previously dumped from this program;
3) the terminal;
4) or do you want to not enter more data;
5) or do you wish to end the session.

Enter choice: 4


Input the numbers of the groups you wish
to compare -- eg, 1 2
47 48

Characters pre-pupal diapause present
and forwing with basal and radial streaks
differ in the following otus:
none.
Thus they may be grouped.
```

```
Do you wish to enter data from:

1) a file;
2) files previously dumped from this program;
3) the terminal;
4) or do you want to not enter more data;
5) or do you wish to end the session.

Enter choice: 5

Data saved in files char, otu, ances and haschar.
Session may be restarted by typing go.

] ?go.


Session may be ended at any time by typing e.

Do you wish to:

1) have CLAUD give you advice;
2) discuss with CLAUD your problem;
3) end this session.

Enter choice: 2
Do you wish to:

1) Enter data from dumped files;
2) Enter data from other files;
3) Enter data from the terminal;
4) Change the characters that are ancestral;
5) Check groupings;
6) Determine the Similarity Coef. of 2 otus;
7) Leave the program.

Enter choice: 1

Files char, otu, ances and haschar consulted.


Do you wish to:

1) Enter data from dumped files;
2) Enter data from other files;
3) Enter data from the terminal;
4) Change the characters that are ancestral;
5) Check groupings;
6) Determine the Similarity Coef. of 2 otus;
```

```
7) Leave the program.

Enter choice: 3

Selection inoperative -- choose again

Do you wish to:

1) Enter data from dumped files;
2) Enter data from other files;
3) Enter data from the terminal;
4) Change the characters that are ancestral;
5) Check groupings;
6) Determine the Similarity Coef. of 2 otus;
7) Leave the program.

Enter choice: 6
Input the first otu: 11

Not a legal otu.
Input the first otu: 1

Input the second otu: 5

Otus medioviridana and madderana share
the following characters (weights):
forwing without apical streak (A)
forwing termen simple, concave (A)
pre-papillae plate present (10)
anellar ring closed around aedeagus (10)
saccular spine cluster reduced (10)
costal fold present (A)
costal sex scale line absent (A)
labial palpus third segment not reduced (A)
pre-pupal diapause absent (A)

Similarity Coef is 30.

Do you wish to:

1) Enter data from dumped files;
2) Enter data from other files;
3) Enter data from the terminal;
4) Change the characters that are ancestral;
5) Check groupings;
6) Determine the Similarity Coef. of 2 otus;
7) Leave the program.
```

```
Enter choice: 6
Input the first otu: 7

Input the second otu: 8

Otus bigemina and kasloana share
the following characters (weights):
colliculum with denticles (10)
socius apex curved dorsally (10)
socius narrowed (10)
uncus with ventral keel (10)
uncus cleft apically (10)
uncus apically notched (10)
forwing without pre-tornal triangular spot (10)
forwing termen simple, concave (A)
pre-papillae plate present (10)
pre-papillae plate reduced (10)
anellar ring closed around aedeagus (10)
dorsal anellar plate with straight margins (A)
saccular spine cluster reduced (10)
sterigma ring long (10)
female tergite VIII with scales (10)
female sternite VII with medial posterior depressions (10)
costal fold absent (10)
pre-pupal diapause present (10)
forwing with basal and radial streaks (10)

Similarity Coef is 170.

Do you wish to:

1) Enter data from dumped files;
2) Enter data from other files;
3) Enter data from the terminal;
4) Change the characters that are ancestral;
5) Check groupings;
6) Determine the Similarity Coef. of 2 otus;
7) Leave the program.

Enter choice: e

Data saved in files char, otu, ances and haschar.
Session may be restarted by typing go.

] ?listing(char).
char(1,colliculum short, near ostium).
char(2,colliculum long, curved).
char(3,colliculum without denticles).
```

```
char(4,colliculum with denticles).
char(5,socius broad, apically rounded).
char(6,socius sub-quadrate).
char(7,socius apex curved dorsally).
char(8,socius with dorsally curved apical hook).
char(9,socius triangular).
char(10,socius narrowed).
char(11,socius with dorsal margin lobed basally).
char(12,uncus simple, flat).
char(13,uncus with lateral margins rolled ventrally).
char(14,uncus with ventral keel).
char(15,uncus cleft apically).
char(16,uncus apically notched).
char(17,forwing with pre-tronal triangular spot).
char(18,forwing without pre-tornal triangular spot).
char(19,forwing without apical streak).
char(20,forwing with apical streak).
char(21,forwing termen simple, concave).
char(22,forwing termen emarginate).
char(23,forwing termen convex).
char(24,pre-papillae plate absent).
char(25,pre-papillae plate present).
char(26,pre-papillae plate reduced).
char(27,anellar ring open).
char(28,anellar ring closed around aedeagus).
char(29,dorsal anellar plate with straight margins).
char(30,dorsal anellar plate constricted laterally).
char(31,saccular spine cluster present).
char(32,saccular spine cluster reduced).
char(33,sterigma ring short).
char(34,sterigma ring long).
char(35,female tergite VIII without scales).
char(36,female tergite VIII with scales).
char(37,female sternite VII smooth).
char(38,female sternite VII with medial posterior depressions).
char(39,female sternite VII with lateral posterior depressions).
char(40,costal fold present).
char(41,costal fold absent).
char(42,costal sex scale line absent).
char(43,costal sex scale line present).
char(44,labial palpus third segment not reduced).
char(45,labial palpus third segment reduced).
char(46,pre-pupal diapause absent).
char(47,pre-pupal diapause present).
char(48,forwing with basal and radial streaks).
Yes
] ?go.
```

```
Session may be ended at any time by typing e.

Do you wish to:

1) have CLAUD give you advice;
2) discuss with CLAUD your problem;
3) end this session.

Enter choice: 2
Do you wish to:

1) Enter data from dumped files;
2) Enter data from other files;
3) Enter data from the terminal;
4) Change the characters that are ancestral;
5) Check groupings;
6) Determine the Similarity Coef. of 2 otus;
7) Leave the program.

Enter choice: 4
Input the current ancestral character: 2

2 is not currently an ancestral character

Input the current ancestral character: 5

Input the current derrived character: 22

5 is not the ancestor of 22.
Input the current derrived character: 6


Do you wish to:

1) Enter data from dumped files;
2) Enter data from other files;
3) Enter data from the terminal;
4) Change the characters that are ancestral;
5) Check groupings;
6) Determine the Similarity Coef. of 2 otus;
7) Leave the program.

Enter choice: e

Data saved in files char, otu, ances and haschar.
Session may be restarted by typing go.

] ?listing(char).
```

char(5,socius sub-quadrate).
char(6,socius broad, apically rounded).
char(1,colliculum short, near ostium).
char(2,colliculum long, curved).
char(3,colliculum without denticles).
char(4,colliculum with denticles).
char(7,socius apex curved dorsally).
char(8,socius with dorsally curved apical hook).
char(9,socius triangular).
char(10,socius narrowed).
char(11,socius with dorsal margin lobed basally).
char(12,uncus simple, flat).
char(13,uncus with lateral margins rolled ventrally).
char(14,uncus with ventral keel).
char(15,uncus cleft apically).
char(16,uncus apically notched).
char(17,forwing with pre-tronal triangular spot).
char(18,forwing without pre-tornal triangular spot).
char(19,forwing without apical streak).
char(20,forwing with apical streak).
char(21,forwing termen simple, concave).
char(22,forwing termen emarginate).
char(23,forwing termen convex).
char(24,pre-papillae plate absent).
char(25,pre-papillae plate present).
char(26,pre-papillae plate reduced).
char(27,anellar ring open).
char(28,anellar ring closed around aedeagus).
char(29,dorsal anellar plate with straight margins).
char(30,dorsal anellar plate constricted laterally).
char(31,saccular spine cluster present).
char(32,saccular spine cluster reduced).
char(33,sterigma ring short).
char(34,sterigma ring long).
char(35,female tergite VIII without scales).
char(36,female tergite VIII with scales).
char(37,female sternite VII smooth).
char(38,female sternite VII with medial posterior depressions).
char(39,female sternite VII with lateral posterior depressions).
char(40,costal fold present).
char(41,costal fold absent).
char(42,costal sex scale line absent).
char(43,costal sex scale line present).
char(44,labial palpus third segment not reduced).
char(45,labial palpus third segment reduced).
char(46,pre-pupal diapause absent).
char(47,pre-pupal diapause present).
char(48,forwing with basal and radial streaks).

```
Yes
] ?como -e
]... .
No
] ?q
OK, como -e
```