Missouri S&T Library and Learning Resources

Scholars' Mine

Masters Theses

Student Theses and Dissertations

1984

CIEGEN: A system for testing knowledge base compilation heuristics on a microcomputer

Jayne D. Ward

Follow this and additional works at: https://scholarsmine.mst.edu/masters_theses

Part of the Computer Sciences Commons

Department:

Recommended Citation

Ward, Jayne D., "CIEGEN: A system for testing knowledge base compilation heuristics on a microcomputer" (1984). *Masters Theses*. 4544.
https://scholarsmine.mst.edu/masters_theses/4544

This thesis is brought to you by Scholars' Mine, a service of the Missouri S&T Library and Learning Resources. This work is protected by U. S. Copyright Law. Unauthorized use including reproduction for redistribution requires the permission of the copyright holder. For more information, please contact scholarsmine@mst.edu.

المنارة للاستشارات

www.manaraa.com

CIEGEN: A SYSTEM FOR TESTING KNOWLEDGE BASE
COMPILATION HEURISTICS ON A MICROCOMPUTER

BY

JAYNE D. WARD, 1960-

A THESIS

Presented to the Faculty of the Graduate School of the

UNIVERSITY OF MISSOURI-ROLLA

In Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE IN COMPUTER SCIENCE

1984

Approved by

Billy E. Gillett (Advisor) ____________

____

ABSTRACT

The expert system has proven itself to be a valuable aid in diagnosing and treating problems in domains requiring expertise. The commercial world has been alerted to this fact and the thrust is to make the expert system portable and available on small computers.

The goal of this research has been to lay the groundwork for a domain independant expert system builder on a microcomputer. The result of this effort was CIEGEN, a system consisting of a rule compiler, inference engine, and rule generator developed on the IBM PC. It is domain independant, responsible for transforming a knowledge base of rules into heuristic based decision trees, and capable of performing backward chaining consultations.

The system is also heuristic independant, allowing a knowledge base to be compiled by different heuristics and compared using the log created by the inference engine. A subgoal of the development of CIEGEN has been to study the heuristics used to compile a knowledge base because the efficiency of the expert system is based on the intelligence of the heuristic. The heuristic used by EMYCIN was implemented and compared with a heuristic developed by the author. For the six types of knowledge based generated by CIEGEN's rule generator, EMYCIN's heuristic, on the average, executed more quickly.

ACKNOWLEDGEMENT

The author would like to recognize and express her appreciation to all of the people who contributed to the development of this thesis. Dr. Bill E. Gillett and Dr. Arlan R. DeKock have maintained enthusiasm and been very supportive throughout all phases of this research. They both have contributed a considerable amount of time and effort discussing and developing ideas for this thesis. Dr. Ray B. Kluczny reviewed this work in a timely manner; his comments and criticisms were helpful. Alan D. Christiansen was very helpful in the intial stages of CIEGEN. Finally, a special thanks is due to my family, whose fervent and continuous support enabled the completion of this research.

TABLE OF CONTENTS

LIST OF TABLES

I. INTRODUCTION

The field of Artificial Intelligence (AI) is a subfield of Computer Science and was founded for the purpose of creating systems that could acquire and apply knowledge to problem solving. Researchers discovered, upon attempting to write a program that incorporated intelligence, that very little was understood about the acquisition and storage of knowledge in the human brain. Therefore, one goal of the field is to uncover underlying mechanisms of human intelligence through the developments in AI. The diversity of what constitues intelligence has led to the creation of areas of research in, for example, natural language understanding [1, 2, 3], robotics [4], vision [5], and knowledge engineering [6].

Knowledge engineering, specifically the building of expert systems, is the area of AI that directly relates to this research. The difference between an expert system and a traditional computer program is demonstrated by the types of problems it solves. An expert system typically solves ill structured problems, or problems with incomplete data. It does not solve problems requiring "number crunching" or problems that can be solved by plugging values into a formula as some of the traditional data processing programs.

Expert systems solve problems that require expertise in the areas of, for example, diagnosis, interpretation of data, monitoring, repair, and design. Expertise implies a combination of heuristics or rules of thumb, textbook knowledge, and reasoning. Expert systems use self-knowledge or meta-knowledge enabling them to reason about their solutions and they typically have explanation facilities to justify their solutions. The expert system asks the user for information it needs to solve the problem and allows the user to ask it questions, just as the human expert operates.

Expert systems have been built in many diverse domains. Three examples of expert systems and their domains are: DENDRAL determines molecular structures of unknown compounds [7, 8, 9], MYCIN diagnoses infectious diseases [9, 10, 11], and PROSPECTOR gives advice on finding ore deposits from geological data [12].

The most desirable way to build an expert system is through a domain independant tool insuring the separation of the knowledge and the control structures. A desirable feature of this tool is the ability to build an expert system on a microcomputer making it convenient for most people in the commercial field.

This paper describes a system, CIEGEN, which was designed with an emphasis on the features of domain independance, portability, and efficiency. CIEGEN, consisting of a rule compiler, inference engine, and rule generator, aids the user in building an efficient knowledge base on the IBM PC.

The knowledge base contains the expertise used to solve problems in a particular domain such as medicine or geology. A popular representation for the expertise is rules of the form: if (condition) then (action), because of their modularity, representing single "chunks" of knowledge. Other representations of knowledge include frames and semantic nets, but these will not be addressed in this paper.

In CIEGEN, these rules are transformed into a representation that will execute more efficiently by a process called rule compilation. Rules are compiled into decision trees which effectively allow parallel execution of several rules at once. The antecedent (condition) chosen as a branch of the decision tree is selected by a heuristic. Since the heuristic is responsible for the efficiency of the knowledge base it is important to be able to compare the results of different heuristics. This need is provided by CIEGEN.

Literature related to the compilation process and expert systems are reviewed in Chapter II. Chapter III describes CIEGEN's rule generator, compiler, and inference engine. Demonstrating the usefulness of CIEGEN, two heuristics for rule compilation are compared in this paper. One heuristics is currently in use by an expert system builder, EMYCIN, and the other heuristic was developed by the author. Both of these are described in Chapter III. An example of the compilation and consultations are given in Chapter IV. The results of this research are summarized in Chapter V and suggestions for further research are given in Chapter VI.

II. RELATED LITERATURE

A. RULE COMPILATION

Many researchers agree, that as larger knowledge bases are required for expert systems, techniques such as rule compilation will become mandatory [9]. Heuristic compilation is equivalent to establishing the search strategy. The exception to the equivalency is the meta-knowledge that may be applied at execution time to alter the search path. As knowledge bases become large it is necessary to perform intelligent searches to keep costs from exponentially increasing. It is also important to eliminate redundancy in testing of similar patterns in rules which constitutes the fundamental compilation algorithm.

Researchers working with the EMYCIN system conducted a study comparing consultation times of expert systems using compiled knowledge bases with consultation times of intepreted knowledge bases. Results showed that the inter-question or "think time" was cut close to half for the systems PUFF, SACON, and MYCIN [13].

These systems were backward chaining or goal directed systems. This means that a goal is established, rules concluding about this goal are gathered, and the conditions in these rules become the new subgoals. This process continues until all conditions in a rule are known, the needed actions are executed. To compile a backward chaining knowledge base means that all rules concluding about a single parameter are located in one decision tree. Rather than searching the knowledge base for all rules concluding about a particular parameter, the inference engine simply travels down the branches of the decision tree.

Compilation has also proven effective in the data driven or forward chaining systems. A data driven system begins with known values in what is called working memory, matches the left hand side of the rules with the known values and draws conclusions from those rules. The conclusions enter working memory and cause other rules to be candidates for execution. The problem with these systems is that with large knowledge bases, the matching process is very slow because it has to repeatedly check elements in working memory. Data driven systems have been reported to spend over nine-tenths of their run time performing the matches [9].

The most common attempt at improving the efficiency of these systems, known as production systems, has been by combining indexing with interpretations of the left hand sides. A successful implementation of the compilation process has been developed by Forgy [14] which is the Rete Match Algorithm. The compiler exploits the properties of similar conditions and the fact that individual productions only change a few facts in memory. Forgy showed, through his studies, that by compiling the productions, the execution time was cut by several orders of magnitude.

B. EXPERT SYSTEM BUILDING TOOLS

An expert system building tool is a domain independant system allowing the development of expert systems in several domains. They prompt the knowledge engineer or expert for rules (knowledge), parameters (goals), and definitions of the parameters. They provide the control structure for the expert system, which includes the inference engine and compiler (if this technique is used).

The level of interaction between the user and the system varies among different systems. For example, EMYCIN prompts its user with a terse Abbreviated Rule Language which is a cross between LISP and English. Teiresias [15, 16] interacts with its user in reasonable English. One limitation of this system, due to the difficulty of parsing English, is the assumption that a dictionary and knowledge base have already been established and the user is merely editing the knowledge base. Other systems such as KAS [9] are not as versatile and were developed specifically for use with an expert system (PROSPECTOR).

III. SYSTEM DESIGN

CIEGEN is a system consisting of a compiler, inference engine, and automatic rule generator. The knowledge base used for this research was generic in the sense that the rules consist of arbitrary alphabetic letters with no particular meaning assigned to them. The reason for using a generic knowledge base as opposed to a particular domain was to permit clearer recognition of the results from the compiler.

CIEGEN was developed on an IBM PC and written in IQLISP [17]. Since the IQLISP environment occupies approximately 106k bytes of RAM, the machine should be equipped with at least 256k of RAM.

In order to conduct this research a total of six packages were needed. They include the Rule Builder, the Rule Compiler, Inference Engine, one of the heuristics for compilation: Heuris1 (Most_Often_Occurring), Heuris2 (Minimum Average Antecedent), Rules which is a general utility package used by all other packages, and FLOAT which is a package to enable real arithmetic.

A. RULE BUILDER

The decision to mechanically generate rules was made to allow control over certain parameters describing the knowledge base. Some of those parameters are: the number of rules, the number of unique consequents, the number of antecedents per rule and the number of knowns per rule. Another factor leading to the decision to automatically generate rules was to be able to maintain randomness and to avoid creating rules that would favor one heuristic over another.

1. <u>Rules</u>

The rule (A B C ==> D) is read as: [If A and If B and If C are true then conclude D is true], so that if A or B or C is false then no conclusion is made. Similarly, a rule involving a "not" (where # = "not") such as (#A B C ==> D) reads "not" A and B and C in order to conclude D is true.

In CIEGEN, the structure of a rule is a LISP list ((A B C) D) where the CAR of the rule is the list of antecedents and the CADR of the rule is the conclusion.

Each rule will have only one consequent and its certainty factor will be one (assuming no probability is involved). This first constraint could be changed by allowing the conclusion to be a list rather than an atom and the latter by making the certainty factor a property of the rule. However, for the purpose of studying different heuristics for compiling rules, the former structure proved sufficient.

As rules are generated, certain information about them is stored. For example, the total number of antecedents and the number of knowns are stored in an array called ANT_INFO to be used for examination and for the Minimum Average Antecedent heuristic. Also stored is each upper case letter used in a rule. An upper case letter represents something that is unbound, it will have a set of rules concluding about it. For example, suppose the first two rules in the knowledge base are [((P G L i) A) ((F e d M) A)]. The first and second rows of ANT_INFO would be as follows:

A 1 4 P G L

A 2 4 F M , where the ASCII values of the the letters are used because the array is all integer (A = 65, P = 80, G = 71, L = 76, F = 70, M = 77). The lower case letters are used to represent known values. They are initially given to be true or false corresponding to information asked of a user in a typical expert system consultation. Therefore, they do not have rules concluding about them.

2. Cycling

a. First Order Cycling is something that is expected to happen if nothing is done to prevent it. First order cycling is demonstrated as follows:

B C D ==> A

A E F ==> B .

So in order to conclude A the value of B is needed, but in order to conclude B the value of A is required. This problem was initially eliminated by allowing only letters that appear later in the alphabet be candidates for antecedents.

For example, to conclude A, B - Z are candidates, and to conclude B, C - Z are candidates.

Lower case letters a - e are also candidates to be chosen as antecedents. One reason for the lower case letters is to increase the size of the antecedent bucket, providing something to choose from when antecedents are needed for conclusions later in the alphabet. For example, rules to conclude about X have only Y and Z, Y has only Z, and Z has no letters to choose from if the lower case letters are not included.

After analyzing the resulting rules generated by this method, it was felt that part of the randomness was lost by restricting the bucket of candidates. Letters later in the alphabet appeared often in the earlier rules which favored one heuristic over another.

Thus, a decision was made to allow cycling to occur by keeping the antecedent bucket the same throughout the rule generation. After all rules had been generated, a check for first order cycling was made and corresponding rules were eliminated from the knowledge base. To demonstrate this, suppose we have the rule (Z C P G ==> H). This method says to check the rules concluding about C to see if H was used as an antecedent. If it was, as in the rule (X <u>H</u> S D ==> C), then a cycle exists and the rule (Z C P G ==> H) is removed from the knowledge base. If H did not appear in the rules concluding about C, rules about G are similarly examined. Rules about Z and P are not important at this point since they appear later in the alphabet than H and will be tested later. If later tests find that one of them uses H in their antecedent list then that rule will be eliminated. The decision to eliminate the current rule (Z C P G ==> H) rather than the previous rule (X H S D ==> C) was made arbitrarily, recognizing the fact that the previous rules had already been check for cycling and were all right.

Control over the number of rules in the knowledge base was lost by this method because many rules were being eliminated. In some cases, an entire ruleset (for example, all rules concluding about H) was removed. This means that either H would have to be tagged undeterminable, or would have to be given a value of true or false since there were no rules to conclude about it.

The method chosen for rule generation allowed second and higher order cycling to occur by keeping the antecedent bucket the same throughout the rule generation. This method differs from the previous one in that the check for first order cycling was done as the antecedents were generated.

For example, suppose the antecedent candidate C is generated for the current rule concluding about H. To test for first order cycling, the antecedents used in the rules concluding about C are scanned for an H. If an H is used as an antecedent in a rule concluding about C, then C is discarded as a candidate for an antecedent in the current rule being generated. If the antecedent candidate generated had been an I, or any letter later in the alphabet than H, it would have been accepted as an antecedent. The reason for this is because the rules concluding about I - Z have not been generated yet and do not pose a threat to the cycling problem at this point.

b. <u>Second Order Cycling</u> is also expected. An example of such is:

B C D ==> A

E F G ==> B

A H J ==> E.

Since E does not appear in the antecedent list for A, this set of rules pass the first order cycling test. The first two rules pass the second order test, but the third rule causes a cycle. For example, B is needed to conclude A, E is needed for B, but A is needed for E. It is possible that a cycle would not occur until the fourth, fifth, sixth or higher rule was generated.

It was decided that the amount of time spent checking for cycles higher than first order would be greater than the benefits gained from such a check. Instead, the inference engine was given the responsibility to check for higher order cycling and to take note when this occurred.

3. <u>Random Number Generator</u>

Unfortunately, IQLISP does not have a built in random number generator. However, it does have a function called DTIME that generates hundredths of seconds since midnight, which would be something like 58245 at 9:00 a.m.. The function DTIME produced rules like (E F G H ==> A) which does not give the appearance of being randomly generated. The reason for this is due to the fact that the function DTIME is linear. Adding a delay between generation times eliminated adjacent antecedents but was not effective in altering the linearity.

The next random number generator examined was one that is currently called RANDU and is as follows:

```
      SUBROUTINE RANDU (IX,IY,YFL)
            IY = IX * 65539
            IF (IY) 5,6,6
    5       IY = IY + 2147483647 + 1
    6       YFL = IY
            YFL = YFL * .4656613E-9
            RETURN
            END
```

It involves very large numbers and relies on the fixed overflow mechanisms of the system. This just means that it is possible to get negative numbers which is the reason for the check for a negative number in line 3. However, IQLISP will allow a number to have 77000 digits and will most likely run out of working memory before a number is generated. An error such as "stack exceeds 64 k " will appear on the screen when this happens.

Rather than trying to modify this algorithm, another was chosen and is as follows:

```
SUBROUTINE RANDOM (A)
MULT = 25211
BASE = 32768
A = MOD [ (A * MULT) , BASE ]
RETURN
END
```

The % function is the IQLISP equivalent to the MOD function which takes the remainder of a division.

This method generates a number between 0 and 1, therefore requiring the FLOAT.LSP package to be loaded into memory before running. The random number is then multiplied by one plus the interval of numbers desired, and the integer portion is added lower bound giving an antecedent candidate, for example, between A and Z.

4. Parameters

As mentioned in the section on rule building, a user has a certain amount of control over the number of conclusions and the number of rules per conclusion wanted. For example, suppose rules to conclude about (A - J) and four rules for each conclusion are desired. This way, each conclusion (A - J) will have four rules concluding about it.

The number of antecedents per rule was arbitrarily chosen randomly to be between two and four. The random number generator RANDOM, mentioned previously, is used to produce this number.

After the antecedent has passed all of the tests, another number between one and ten is randomly generated. The purpose of this number is to decide whether to add a "not" symbol to the antecedent just generated. If the number generated is one, then the antecedent is negated. Therefore, on the average, every tenth antecedent generated will be transformed into its negation. This number may be increased or decreased if other distributions are desired.

The symbol chosen to represent a "not" is the pound sign (#). An ideal symbol, the tilde ~, was initially chosen, but IQLISP has its own system meanings for the tilde. It uses it to continue a line and is recognized as a comment delimiter.

B. RULE COMPILER

As mentioned in the literature review, it is possible to incorporate much of the search strategy at compile time. The goal is to organize the rules in a way that decreases the amount of work the inference engine has to do at consultation time.

The amount of knowledge incorporated into the compilation depends on the heuristic being used. The heuristic's responsibility is to determine the order in which the antecedents will be tested. The heuristics will be discussed in further detail in the next section. However, in this section, the selection of an antecedent is assumed to be arbitrary.

1. Algorithm

The basic rule compiler algorithm is :

Compile (ruleset)

For all R in ruleset for which the list of antecedents is empty, do

(1) output conclusion of R;

(2) remove R from ruleset.

While R is non-empty do

(3) select an antecedent, say C, from the list of antecedents of some rule;

(4) output a branch, using antecedent C as the conditional;

(5) on the true side of the branch; compile all rules that contain an antecedent C after deleting C from each;

(6) on the negated side of the branch; compile all rules that contain an antecedent #C after deleting #C from each;

(7) remove from ruleset those rules compiled in (5) and (6).

A ruleset is a set of rules all concluding about the same thing. For example, a ruleset concluding about A might be as follows:

[((H V I d)A) ((E H)A) ((C D #H)A) ((B C E F)A)]

The steps to compile this ruleset are given below

The ruleset is not empty, so steps 1 and 2 are skipped.

3. Suppose H is chosen.

4. H is the branch output (it is actually the top of the tree).

5. On the true side, compile the new ruleset which is:

[((V I d) A) ((E) A)].

This ruleset is not empty, skip steps 1 and 2.

3. Suppose E is chosen.

4. A branch for E is output.

5. On the true side of E, compile new ruleset:

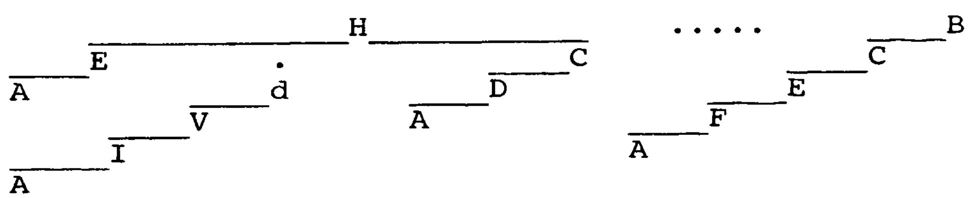
[(( ) A)].

1. The new ruleset is empty, so the conclusion A is output.

6. There is no negated side (rules involving #E).

7. New ruleset : [((V I d) A)].

The process continues, putting out branches for the rest of the antecedents in the ruleset, until the tree looks like:



where the ..... indicates a sequential list of trees.

2. Data Structures

a. Decision Trees are represened as follows : (antecedent (if ant. is true) (if ant. is false)). A conclusion may have more than one decision tree associated with it. In the previous example, there were two associated with A - one tree with H at the top, and one with B at the top. Therefore, each conclusion is associated with a decision tree list.

b. <u>Decision Tree Lists</u> are bound to the name of the conclusion which is represented by an asterisk with the consequent. For example, *A = [(H (if H is true) (if H is false)) (B (if B is true) (if B is false))]. To find out if H is true or false, its decision tree has to be examined which may itself be a decision tree list.

This structure eliminates all negation symbols from the rulesets by having a true decision tree list and a false decision tree list. The need for sequential trees was discovered as more complicated rulesets were experimented with, and this structure will handle any number of sequential trees.

C. <u>HEURISTICS FOR ANTECEDENT SELECTION</u>

The most important part of the compile algorithm is the selection of the antecedent to put out as a branch. In the section on compiling, the selection of an antecedent was assumed to be arbitrary. This section presents two heuristics that were investigated, each of which approach the selection process in very different ways.

The first heuristic was used by van Melle in EMYCIN. It is straightforward and uses no knowledge about the relationship among the rules. The second heuristic incorporates knowledge about the rules generated, specifically, the average number of antecedents required to conclude about a parameter.

1. Most Often Occurring

The reasoning behind this heuristic is that an antecedent appearing in the most rules must be important and therefore should be placed at the top of the tree. To exhibit this heuristic, assume the same ruleset as before:

[((H V I d)A) ((E H)A) ((C D #H)A) ((B C E F)A)].

A new list is made from the ruleset list containing each antecedent and the number of times it appears in the ruleset. For example,

[(H 3) (V 1) (I 1) (d 1) (E 2) (C 2) (D 1) (F 1)]

Note that the interest is in how many times an antecedent is used, disregarding how it was used (negation). Therefore, H would be chosen as the first antecedent (branch).

2. Minimum Average Antecedent

This heuristic examines the relationship between the antecedent in the current ruleset with those in the rest of the knowledge base.

The reasoning is to choose the antecedent that, on the average, requires the least amount of work to determine its value. For example, if the average number of antecedents needed to conclude G is two and the average number of antecedents needed to conclude H is four, then G will be chosen.

The number of antecedents required for a particular rule is readily available from the array ANT_INFO as discussed in the section on rules. The total number of antecedents minus the number of knowns is the number of antecedents that will be inferred to conclude a particular rule. For example, in the rule ((H V I d) X) the total number of antecedents is four, the number of knowns is one. Therefore, the number of antecedents needed to conclude X is three.

An extremely misleading assumption to make is that the number of antecedents required to conclude X is only three. This implies that the number of antecedents required to conclude H is one, the number to conclude V is one and the number to conclude I is one which is possible, but unlikely.

To rectify this problem, the rules with H, V, and I as consequents must be examined. Then the rules whose consequents are the antecedents in H, V, and I must be examined and new averages calculated. As can be seen, this is an n-order problem. For this research, averages were calculated three times in addition to the averages calculated during generation of the rules. The decision to calculate averages an additional three times was chosen arbitrarily. However, consultation results (number of IFs) were very close, in some cases identical, between the knowledge bases compiled with second and third averages indicating a point of diminishing return.

D. INFERENCE ENGINE

The inference engine's job, if the compiler performed correctly, is trivial. It retrieves the decision trees built by the compiler, examining either the true side or false side of the tree depending on the value of the current antecedent. Recall that a decision tree takes the form:
*A = ((a (true side) (false side)) (sequential trees)), where a = antecedent, (true side) or (false side) can be a list of decision trees. So if the antecedent is true, the inference engine recurs with the CAR of the true side. If that returns false or undeterminable, the next tree (still on the true side of the antecedent) is examined. If there are no more trees (possible ways to conclude the antecedent) then "undeterminable" is returned from the true side and also to *A if there are no more sequential trees.
Therefore, using the example in section B.1., the compiled decision trees are structured as follows:

```
 *A = [ (decision tree #1)  (decision tree #2) ]
    = [ (H (true side)(false side) ) (B (true side) (false
                                              side) ) ]
    = [ (H ((decision tree 1) (decision tree 2)) (false
        side))
        (B (true side) (false side = nil) ) ]
    = [ (H ( (E (A) nil)(d ( (V ( (I (A) nil) ) nil)) nil) )
           (C ( (D (A) nil) ) nil) <= false side of H
         )   <== end of decision tree # 1
         (B ((C ((E ((F (A) nil)) nil)) nil)) nil) ]
```

IV. EXAMPLE CONSULTATION

Suppose the following set of rules are under consideration :

```
[ (((b c d)F) ((I H)F)) (((F #H) G) ((a J)G))
 (((e J)H) ((g I b)H)) (((#G a)I) ((#f e b)I))
 (((F e)J) ((#F I)J)) ].
```

Note that each ruleset has two rules concluding about the conclusions F, G, H, I, and J. They will be compiled under the assumption that antecedents will be chosen in the order of (F #F G #G H #H I #I J #J). However, if any rule contains a known (lower case letter) it will be chosen first.

The consultation begins with the user specifying the values of the lower case antecedents. For this research, all lower case letters were set to be true. This was to enable consistency since different combinations of "trues" and "falses" will produce different results. For this example, it is assumed that the goals for the consultation are G and H.

The results of the compilation and consultation follow.

```
*F = (((b ((c ((d (F) NIL)) NIL)) NIL) (H ((I (F) NIL))
                                             NIL)))

*G = (((a ((J (G) NIL)) NIL) (F ((H NIL (G))) NIL)))

*H = (((b ((g ((I (H) NIL)) nil)) nil) (e ((J (H) NIL))
                                             NIL)))

*I = (((a ((G NIL (I))) NIL) (b ((f NIL ((e (I) NIL))))
                                          NIL)))

*J = (((e ((F (J) NIL)) NIL) (F NIL ((I (J) NIL)))))
```

INFERENCE ENGINE LOG:

```
(SET_TREES '(a b c d e f g) '**TRUE**)

(INFER (QUOTE G))

a is known to be true
Attempting to satisfy J
  e is known to be true
  Attempting to satisfy F
    b is known to be true
    c is known to be true
    d is known to be true
  F is deduced to be true
J is deduced to be true

G has been deduced to be true after 7 IFs


(INFER (QUOTE H))

b is known to be true
g is known to be true
Attempting to satisfy I
  a is known to be true
  Attempting to satisfy G
    a is known to be true
    Attempting to satisfy J
      e is known to be true
      Attempting to satisfy F
        b is known to be true
        c is known to be true
        d is known to be true
      F is deduced to be true
    J is deduced to be true
  G is deduced to be true
  I cannot be determined
  b is known to be true
  e is known to be true
  f is known to be true
  I cannot be determined
  I cannot be determined
  I cannot be determined
  H cannot be determined
  H cannot be determined
  e is known to be true
  J is known to be true

  H has been deduced to be true after 16 IFs
```

V. SUMMARY OF RESULTS AND CONCLUSIONS

This paper presented a system, CIEGEN, developed as a tool for building expert systems on a microcomputer and as a valuable aid for research on rule compilation. It was written in IQLISP and implemented on the IBM PC.

The main purpose of CIEGEN's rule generator is for use in research on rule compilation because it allows the generation of parameterized knowlege bases.

In CIEGEN, the knowledge base can be described by the number of parameters to be concluded about, the number of knowns per knowledge base, the number of rules per conclusion, and the number of antecedents per rule. These characteristics are input to the Rule Builder which keeps track of the number of knowns per rule, the total number of antecedents per rule, the unknowns or antecedents to be inferred per rule, and the average number of antecedents used per rule as rules are generated. In order for a generated antecedent to be accepted, it must guarantee not to cause first order cycling. If first order cycling is detected, it is discarded and another antecedent is generated.

After the rules have been generated, they are compiled into a form that will hopefully execute more efficiently. Rule compilation is the process of transforming a knowledge base consisting of rules to a knowledge base of decision trees. The transformation effectively takes place by choosing an antecedent to be placed at the top of the tree, gathering all of the rules using this antecedent, choosing another antecedent as a branch, gathering those rules using this antecedent, continuing the process until a rule has had all of its antecedents chosen as branches, then printing the conclusion as a leaf node. At this point, those rules containing the negation of the antecedents output as branches are gathered and the process is repeated, forming the right side of the tree. As mentioned previously, this technique has been implemented in EMYCIN, tested against the technique of interpreting rules, and shown to cut inter-question time close to half for the expert systems MYCIN, PUFF, and SACON.

The efficiency of the resulting decision trees depends on the method of selecting the antecedents. The author developed a heuristic (Minimum Average Antecedent) for selecting antecedents based on knowledge gathered at generation time. The heuristic forces those antecedents requiring the least amount of work, on the average, to be chosen first. The amount of work is determined by the number of antecedents needed to be inferred in order to conclude the antecedent in the current rule.

This heuristic was compared with a heuristic (Most Often Occurring) used by EMYCIN. The Most Often Occurring heuristic chooses the antecedent that appears in the most rules. The heuristic is based upon the idea that if it is used in the most rules, it must be important and should be placed nearest the top of the tree.

The efficiency of the two heuristics was measured by the number of IFs executed during a consultation with each knowledge base. The performance of a consultation is the responsibility of CIEGEN's inference engine. The inference engine consists of a series of procedures which examine the decision trees built by the rule compiler and counts the number of antecedents it has to infer before a conclusion can be made. The inference engine retrieves the decision tree list of the parameter typed in by the user (what is to be inferred, which can be the negation of a parameter), then recursively examines the decision trees of the antecedents in the original tree until a conclusion is made.

An antecedent's decision tree will only be examined once and at that time the value is bound to the antecedent. It is possible to need the value of an antecedent that is being inferred, thus creating a cycle. When this happens, the inference engine notes that the current path cannot be continued and retrieves the next decision tree in the list if one exists.

The inference engine traces the paths led by the decision trees and outputs the total number of nodes it visited with the value of the parameter being inferred. The value of the parameter will either be "true" or "undeterminable". The parameter will be undeterminable if its decision trees were involved in cycles preempting it from making a conclusion, or if the only path available required the parameter to be false.

The fact that the parameters are never concluded to be false (unless the negation of the parameter is being inferred) is not obvious with the emphasis on a generic knowledge base. The parameters could represent parts of more complex rules. For example, concluding "A" to be true could represent the conclusion "it is true that the bacteria is not present." Similarly, concluding "B" to be true could represent that the bacteria is present. A rule requiring that the bacteria is not present would use "A" instead of "#B" and vice versa. Therefore, it is possible to generate a realistic knowledge base with all non-negated antecedents and conclusions that are true.

Some parameters describing the knowledge base were held constant and some were allowed to vary. For each knowledge base, four rules were generated for each letter in the alphabet (A-Z) and two to four antecedents were created for each rule. These numbers were arbitrarily chosen, as a knowledge base consisting of 104 rules was considered to be a reasonable size. Knowledge bases permitting negations were generated with the number of knowns varying among 5, 7, and 14. Another set of knowledge bases were generated witholding negations and varying the number of knowns among 5, 7, 14, and 21.

For each number of knowns, there were five knowledge bases generated and tested. The intent for varying the number of knowns was to test whether either heuristic would be affected by giving it more information. The goals of each consultation were assumed to be A-E, so the averages, shown in Table I, are based on the total number of IFs required to infer these five parameters. The right side of Table I shows the results of consultations performed on knowledge bases without negations in the rules. The reason for restricting negations from the knowledge base was because the second heuristic Minimum Average Antecedent did not take into account the fact that parameters are never deduced to be false.

Overall, the Most Often Occurring heuristic performed consultations in the least number of IFs. As the number of knowns increased, Most Often Occurring's average increased. The author suggests that this is because there are fewer common antecedents as the number of knowns are increased. In other words, the number of antecedents to choose from ranged from 26 to 28 to 35 which means that there will be less repetitive antecedents in the rules. However, in the case where the number of knowns and the number of parameters to be inferred were equal, Most Often Occurring's average dropped rather than increased. This may be due to the fact that because there are so many knowns in the knowledge base the actual cause of the decrease is obscured.

The Minimum Average Antecedent heuristic explicitly put knowns closest to the top of the decision tree. This means that if there are knowns in a rule, they will be tested first. Therefore, it is expected that as the number of knowns increases, the overall average will decrease. The

TABLE I

CONSULTATION RESULTS

AVERAGE NUMBER OF IFs

	Antecedent selection interval (negations permitted)			Antecedent selection interval (negations withheld)			
HEURISTIC	26	28	35	26	28	35	42
Most Often Occurring	75	81	84	76	118	123	65
Minimum Average Antecedent	152	132	98	123	126	91	74

Often Occurring heuristic did not explicitly place knowns first, it merely placed rules containing knowns closest to the top.

As mentioned previously, the Minimum Average Antecedent algorithm did not take into account the possibility of failures. It makes the assumption that the conclusions made will be true. Since parameters are never concluded to be false, if a rule requires the negation of an antecedent to be true, the rule will fail. In this case the average number of antecedents to conclude a parameter no longer makes sense. Possibly incorporated into this heuristic should be a penalty for those parameters that have "nots" present with their antecedents. The other alternative is the one previously mentioned, letting the parameters or antecedents represent things more complicated than true or false.

Data concerning times of the system were also collected. The average amount of time to generate a knowledge base was twenty minutes. The average amount of time to compile a knowledge base was 31 minutes for the Most Often Occurring heuristic and 29 minutes for the Minimum Average Antecedent. The slight difference may be attributed to the calculations that are done by the Most Often Occurring heuristic tallying the number of times an antecedent appears in the rules. An additional thirteen minutes were required to calculate averages for the Minimum Average Antecedent heuristic. An interesting note to make is that a slightly more complicated heuristic took almost 40% more time to compile.

As demonstrated by the comparison of these two heuristics, CIEGEN is a convenient tool for research and an in depth analysis of heuristics. Information about the rules such as the number of antecedents, values of antecedents, number of rules, and number of knowns is readily available and can easily be varied or held constant. The system is not only domain independant, but is heuristic independant so that heuristics can be easily inserted into the system for testing and comparison with other compilation heuristics.

VI. FURTHER RESEARCH SUGGESTIONS

CIEGEN provides a base for interesting work in at least three areas. A natural extension to the work presented in this thesis are the following areas:

As a research tool -

- a statistical analysis of the interactions between parameters
- a study of additional heuristics
- an addition to the Minimum Average Antecedent heuristic to incorporate negations
- a study of the complexity of heuristics vs. compile time

As groundwork for an expert system -

- explanation capabilities for the inference engine
- the incorporation of uncertainty in the rules
- the allowance of more complex rules with multiple conclusions

In the generation of rules -

- the incorporation of more knowledge about cycling, rather than just checking for first order
- the incorporation of a learning mechanism

CIEGEN provides the groundwork for each of these areas.

BIBLIOGRAPHY

1. Barr, Avron and Edward Feigenbaum. The Handbook of of Artificial Intelligence, vol. 1. California: William Kaufmann, Inc, 1981.

2. Hendrix, Gary G. and Earl D. Sacerdoti, "Natural Language Processing: The Field in Perspective," Byte 6, 9 (1981), 304-352.

3. Winograd, Terry. Language as a Cognitive Process. Massachusetts : Addison-Wesley Publishing Co., 1983.

4. Prendergast, Dan. "A General Purpose Robot Control Language," Byte, 9, 1 (1984), 122-133.

5. Cohen, Paul R. and Edward Feigenbaum. The Handbook of Artificial Intelligence, vol. 3. California: William Kaufman, Inc., 1982.

6. Feigenbaum, E. A. "The Art of Artificial Intelligence : Themes adn Case Studies of Knowledge Engineering," Report : STAN-CS-77-621, Computer Science Department, Stanford University, Stanford, California.

7. Barr, Avron and Edward Feigenbaum. The Handbook of Artificial Intelligence, vol.2. California: William Kaufman, Inc., 1982.

8. Buchanan, Bruce and Edward Feigenbaum. "DENDRAL and META-DENDRAL: Their Application Dimension," Artificial Intelligence, 11 (1978), 5-24.

9. Buchanan, Bruce and Tom M. Mitchell. "Model Directed Learning of Production Rules." Report: HPP 77-6. Heuristic Programming Project, Computer Science Department, Stanford University, Stanford, California.

10. Hayes-Roth, Frederick and others. Building Expert Systems. Massachusetts: Addison-Wesley Publishing Co., 1983.

11. Farley, A. M. "Issues in Knowledge Based Problem Solving," IEEE Transactions System Man and Cybernetics SMC-10, 8 (1980), 446-459.

12. Van Melle, W. "MYCIN : A Knowledge Base Consultation Program for Infectious Disease Diagnosis," International Journal of Man-Machine Studies, 10 (1978), 313-322.

BIBLIOGRAPHY (continued)

13. Van Melle, W. "A Domain Independant Production Rule System for Consultation Programs," STAN-CS-80-820, Stanford, California, June, 1980.

14. Forgy, Charles, L. "Rete: A Fast Algorithm for the Many Pattern / Many Object Pattern Match Problem," Artificial Intelligence, 19 (1982), 17-37.

15. Davis, Randall. "Interactive Transfer of Expertise: Acquisition of New Inference Rules," Artificial Intelligence, 12, 2 (1979), 121-157.

16. Davis, Randall and Douglas Lenat. Knowledge Based Systems in Artificial Intelligence. New York: McGraw-Hill International Book Co., 1982.

17. IQLISP Reference Manual. Washington: Integral Quality, 1983.

VITA

Jayne Denise Ward was born on February 16, 1960 in Springfield, Missouri where she also received her primary and secondary education. She graduated cum laude from Southwest Missouri State University in Springfield, Missouri with a Bachelor of Science in Computer Science and Mathematics in May, 1982.

She has been enrolled in the Graduate School of the University of Missouri-Rolla since August, 1982. Since June, 1983 she has been employed by the United States Geological Survey in Rolla, Missouri and has held a Graduate Teaching Assistantship in the Department of Computer Science from August, 1983 to May, 1984. During May and June, 1984, she has held a Research Assistantship in the Department of Computer Science. She is a member of the honor societies Upsilon Pi Epsilon, Kappa Mu Epsilon, and Sigma Pi Sigma as well as a student member of the American Association for Artificial Intelligene and the Association for Computing Machinery.

APPENDIX A

CIEGEN PACKAGES

PACKAGE "COMPILER.LSP"

RC

Arguments : PARTITION_LIST -- a list of partitions, FILENAME -- filename (in double quotes) specifying a disk file to receive the listing produced by the compiler.
Called by : the user
Calls : DO
RETURNS : the compilation. RC is the function that causes rules to be compiled. The user should type, for example, (RC PL "B:RC3") which will cause the partition list PL to be compiled and output to be sent to the screen and to the file B:RC3

```
(DEF 'RC
'[LAMBDA (PARTITION_LIST FILENAME)
  [PROG ()
       (SETQ *PARMS* NIL)
       (SETQ SESSIONFILE
           (OUTPUT FILENAME))
       (PRINTC '"RULES SUBMITTED:" SESSIONFILE)
       (PRINTC '"RULES SUBMITTED:")
       (TERPRI NIL SESSIONFILE)
       (TERPRI NIL SESSIONFILE)
       (TERPRI)
       (TERPRI)
       (PP PARTITION_LIST SESSIONFILE)
       (PP PARTITION_LIST)
       (TERPRI NIL SESSIONFILE)
       (TERPRI)
       (PRINTC '"RESULTS OF COMPILATION:" SESSIONFILE)
       (PRINTC '"RESULTS OF COMPILATION:")
       (TERPRI NIL SESSIONFILE)
       (TERPRI)
       (DO PARTITION_LIST)
       (CLOSE SESSIONFILE)
       (TERPRI)
       (RETURN 'END_OF_COMPILATION)]])
```

DO

Arguments : PARTITION_LIST -- a list of rule partitions (a complete set of input rules grouped by conclusion parameters)
Called by : RC, itself
Calls : COMPILE, itself
RETURNS : passes each partition to COMPILE. DO puts an extra set of parentheses around the output of COMPILE making it a decision tree list

```
(DEF 'DO
 '[LAMBDA (PARTITION_LIST)
   [PROG (P)
        (COND
             [(NULL PARTITION_LIST)
               (RETURN 'DONE)]
             [T
               (SETQ P
                   (CAR PARTITION_LIST))
               (SETQ OUTFILE
                   (OUTPUT "B:OUTFILE"))
               (PRINC '"(" OUTFILE)
               (COMPILE P)
               (PRINC '")" OUTFILE)
               (CLOSE OUTFILE)
               (SETQ INP
                   (INPUT "B:OUTFILE"))
               (SETQ TREENAME
                   (READLIST (CONS '"*"
                                  (EXPLODE (CADAR P)))))
               (SET TREENAME
                   (READ INP))
               (CLOSE INP)
               (PRINT TREENAME SESSIONFILE)
               (PP (EVAL TREENAME) SESSIONFILE)
               (PRINT TREENAME)
               (PP (EVAL TREENAME))
               (SET (READLIST (CONS '"!"
                                   (EXPLODE (CADAR P))))
                   (EVAL TREENAME))
               (SETQ *PARMS*
                   (CONS (CADAR P) *PARMS*))
               (DO (CDR PARTITION_LIST))
               (RETURN 'DONE)])]])
```

COMPILE

Arguments : RULESET -- A rule partition (a list of rules concluding about a particular ant)
Called by : DO, itself
Calls : CONCLUDE_ALL_W_SATISFIED_PREMISES, ALL_NOT_SATISFIED, SELECT_A_CLAUSE, UNNEGATED, DEL_RULE_SET, FIND_ALL, REMOVE, NEGATION
RETURNS : a decision tree list

```
(DEF 'COMPILE
'[LAMBDA (RULESET)
  [PROG (C K)
      (CONCLUDE_ALL_W_SATISFIED_PREMISES RULESET)
      (SETQ K
          (ALL_NOT_SATISFIED RULESET))
   CONTINUE
      (COND
```

```
        [(NULL K)
          (RETURN 'DONE)]
        [T
          (SETQ C
              (SELECT_A_CLAUSE K))
          (PRINC '"[" OUTFILE)
          (COND
              [(UNNEGATED C)
                (PRIN C OUTFILE)
                (PRINC '"(" OUTFILE)
                (COMPILE (DEL_RULESET C
                                (FIND_ALL C K)))
                (PRINC '")" OUTFILE)
                (SETQ K
                    (REMOVE C K))
                (PRINC '"(" OUTFILE)
                (COMPILE (DEL_RULESET (NEGATION C)
                                (FIND_ALL (NEGATION C) K)))
                (PRINC '")" OUTFILE)
                (SETQ K
                    (REMOVE (NEGATION C) K))]
              [T
                (PRIN (NEGATION C) OUTFILE)
                (PRINC '"(" OUTFILE)
                (COMPILE (DEL_RULESET (NEGATION C)
                                (FIND_ALL (NEGATION C) K)))
                (PRINC '")" OUTFILE)
                (SETQ K
                    (REMOVE (NEGATION C) K))
                (PRINC '"(" OUTFILE)
                (COMPILE (DEL_RULESET C
                                (FIND_ALL C K)))
                (PRINC '")" OUTFILE)
                (SETQ K
                    (REMOVE C K))])
          (PRINC '"]" OUTFILE)
          (GO CONTINUE)])]])
```

NEGATION

```
Arguments :  ANT -- an antecedent
Called by :  COMPILE
Calls     :  system functions only
RETURNS   :  the negation of the antecedent.  This function
             is used by the compiler to specify rules which
             contain the negation of a particular ant.
```

```
(DEF 'NEGATION
'[LAMBDA (ANT)
  (COND
      [(NULL ANT)
        NIL]
      [(EQUAL (CAR (EXPLODE ANT))
```

```
          '"#")
        (READLIST (CDR (EXPLODE ANT)))]
      [T
        (READLIST (CONS '"#"
                        (EXPLODE ANT)))])])
```

ALL_W_SATISFIED_PREMISES

```
Arguments  :  RULESET -- a rule partition
Called by  :  CONCLUDE_ALL_W_SATISFIED_PREMISES
Calls      :  itself
RETURNS    :  a list of all conclusions which have satis-
              fied premises (a nil antecedent list).
```

```
(DEF 'ALL_W_SATISFIED_PREMISES
 '[LAMBDA (RULESET)
   (COND
       [(NULL RULESET)
         NIL]
       [(NULL (CAAR RULESET))
         (CONS (CADAR RULESET)
             (ALL_W_SATISFIED_PREMISES (CDR RULESET)))]
       [T
         (ALL_W_SATISFIED_PREMISES (CDR RULESET))])])
```

CONCLUDE_ALL_W_SATISFIED_PREMISES

```
Arguments  :  RULESET -- a rule partition
Called by  :  COMPILE
Calls      :  ALL_W_SATISFIED_PREMISES, PRINT_EACH
RETURNS    :  construction of leaf nodes.  Prints each
              element returned by ALL_W_SATISFIED_PREMISES
```

```
(DEF 'CONCLUDE_ALL_W_SATISFIED_PREMISES
 '[LAMBDA (RULESET)
   (COND
       [(NULL (ALL_W_SATISFIED_PREMISES RULESET))
         NIL]
       [T
         (PRINT_EACH (ALL_W_SATISFIED_PREMISES RULESET)
               OUTFILE)] )])
```

ALL_NOT_SATISFIED

```
Arguments  :  RULESET -- a rule partition
Called by  :  COMPILE, itself
Calls      :  itself
RETURNS    :  everything in the rule partition that does not
              have a nil antecedent list. (another partition
              list)
```

```
(DEF 'ALL_NOT_SATISFIED
'[LAMBDA (RULESET)
  (COND
      [(NULL RULESET)
        NIL]
      [(NULL (CAAR RULESET))
        (ALL_NOT_SATISFIED (CDR RULESET))]
      [T
        (CONS (CAR RULESET)
            (ALL_NOT_SATISFIED (CDR RULESET)))])])
```

REMOVE

Arguments : CLAUSE -- an antecedent, RULESET -- a rule partition

Called by : COMPILE, itself

Calls : ANT_IS_IN_RULE, REMOVE

RETURNS : a partition list which is RULESET without the rules that reference CLAUSE

```
(DEF 'REMOVE
'[LAMBDA (CLAUSE RULESET)
  (COND
      [(NULL RULESET)
        NIL]
      [(ANT_IS_IN_RULE CLAUSE
           (CAR RULESET))
        (REMOVE CLAUSE
            (CDR RULESET))]
      [T
        (CONS (CAR RULESET)
            (REMOVE CLAUSE
                (CDR RULESET)))])])
```

PRINT_EACH

Arguments : LST -- a list of parameters

Called by : CONCLUDE_ALL_W_SATISFIED_PREMISES, itself

Calls : itself

RETURNS : prints each of the elements in LST to OUTFILE

```
(DEF 'PRINT_EACH
'[LAMBDA (LST)
  (COND
      [(NULL LST)
        'DONE]
      [T
        (PRIN (CAR LST) OUTFILE)
        (PRINT_EACH (CDR LST))])])
```

UNNEGATED

Arguments : ANT -- an antecedent

```
Called by :  COMPILE
Calls     :  LISP builtin functions only
RETURNS   :  T if the antecedent is unnegated, NIL
             otherwise

(DEF 'UNNEGATED
'[LAMBDA (ANT)
  (COND
      [(EQUAL '"#"
            (CAR (EXPLODE ANT)))
        NIL]
      [T
        T])])
```

PACKAGE "IE.LSP"

INFER

Arguments : PARM -- a parameter
Called by : The user.
Calls : UNNEGATED, IS_KNOWN_TO_BE_TRUE, IS_KNOWN_TO_BE_FALSE, IS_KNOWN_TO_BE_UNDET, INFER_LIST, END_CONSULTATION, OPPOSITE, NEGATION
RETURNS : INFER is the top level function for the inference engine. The user calls the function with the parameter whose truth value is desired. The inference engine then displays a trace of the traversal of the knowledge base as it infers the parameter.

```
(DEF 'INFER
'[LAMBDA (PARM)
  [PROG (TEMP)
      (TERPRI NIL IELOG)
      (SETQ IM_WORKING_ON NIL)
      (COND
          [(UNNEGATED PARM)
            (COND
                [(IS_KNOWN_TO_BE_TRUE PARM)
                  (RETURN (END_CONSULTATION PARM
                                      '(T 1)))]
                [(IS_KNOWN_TO_BE_FALSE PARM)
                  (RETURN (END_CONSULTATION PARM
                                      '(F 1)))]
                [(IS_KNOWN_TO_BE_UNDET PARM)
                  (RETURN (END_CONSULTATION PARM
                                      '(U 1)))]
                [T
                  (SETQ TEMP
                       (INFER_LIST PARM
                           (EVAL (READLIST (CONS '"*"
                                               (EXPLODE PARM))))
                       0 0 NIL))
                  (RETURN (END_CONSULTATION PARM TEMP))])]
          [T
            (COND
                [(IS_KNOWN_TO_BE_TRUE (NEGATION PARM))
                  (RETURN (END_CONSULTATION PARM
                                      '(F 1)))]
                [(IS_KNOWN_TO_BE_FALSE (NEGATION PARM))
                  (RETURN (END_CONSULTATION PARM
                                      '(T 1)))]
                [(IS_KNOWN_TO_BE_UNDET (NEGATION PARM))
                  (RETURN (END_CONSULTATION PARM
                                      '(U 1)))]
                [T
                 (SETQ TEMP
```

```
                    (OPPOSITE (INFER_LIST (NEGATION PARM)
                                (EVAL (READLIST (CONS '"*"
                                    (EXPLODE (NEGATION
                                                  PARM))))
                                  ) 0 0 NIL)))
              (RETURN (END_CONSULTATION PARM TEMP))])])]])
```

INFER_LIST

Argruments	:	PARM -- a parameter, DT_LIST -- a decision tree list which is to be used to determine the truth of the parameter, COUNT -- the count of IFs processed so far, SPACE_COUNT -- controls the indentation of messages printed as the inference process progresses, GIVE_UP-controls whether a message should be printed indicating that the parameter cannot be determined. It is possible to arrive at a place in the knowledge base where the DT_LIST NIL, but there are still decision trees to be searched. (This is due to having sequences decision trees.) GIVE_UP will be T if INFER_LIST has just called itself directly and will be NIL otherwise (INFER2 made the call).
Called by	:	INFER, INFER2, itself
Calls	:	PRINT_LINE, INFER2, itself
RETURNS	:	if the parameter is determined to be true or false, then the search stops and INFER_LIST returns the truth and count value fount. If the parameter cannot be determined using one decision tree, the remaining decision trees used until the parameter is determined or there are no more decision trees in the list.

```
(DEF 'INFER_LIST
'[LAMBDA (PARM DT_LIST COUNT SPACE_COUNT GIVE_UP)
  [PROG (TEMP)
      (RETURN (COND
                  [(NULL DT_LIST)
                    (COND
                        [GIVE_UP
                          (PRINT_LINE SPACE_COUNT PARM
                              '"cannot be determined")
                          (SET (READLIST (CONS '"*"
                                          (EXPLODE PARM)))
                              '**UNDET**)
                          (SETQ IM_WORKING_ON
                        (DEL_CURR_GOAL IM_WORKING_ON PARM))
                          (CONS 'U
                              (LIST (ADD1 COUNT)))]
                        [T
                          (CONS 'U
                              (LIST COUNT))])]
```

```
                [(EQUAL 'U
                     (CAR (SETQ TEMP
                              (INFER2 PARM
                                  (CAR DT_LIST) COUNT
                              SPACE_COUNT))))
                   (INFER_LIST PARM
                       (CDR DT_LIST)
                       (CADR TEMP) SPACE_COUNT T)]
                [T
                  TEMP]))]])
```

INFER2

Arguments : PARM -- the parameter to be determined, DT -- a decision tree to be used in determining PARM COUNT -- the count of IFs seen so far, SPACE_ COUNT -- indentation value for messages

Called by : INFER_LIST

Calls : IS_KNOWN_TO_BE_TRUE, IS_KNOWN_TO_BE_FALSE, IS_KNOWN_TO_BE_UNDET, PRINT_LINE, INFER_LIST

RETURNS : the truth and count pair indicating the result of its search. (see INFER_LIST) It searches a single decision tree.

```
(DEF 'INFER2
'[LAMBDA (PARM DT COUNT SPACE_COUNT)
  [PROG (TEMP)
      (RETURN (COND
                [(ATOM DT)
                  (COND
                      [(EQUAL PARM DT)
                        (SET (READLIST (CONS '"*"
                                          (EXPLODE PARM)))
                            '**TRUE**)
                        (CONS 'T
                            (LIST COUNT))]
                      [T
                        (TERPRI)
                        (SPACES SPACE_COUNT)
                        (PRINC '"malformed decision
                                                tree")
                        (COND
                            [ECHO_ON?
                              (TERPRI NIL IELOG)
                              (SPACES SPACE_COUNT IELOG)
                              (PRINC '
                                     "malformed decision
                                                tree")
                              IELOG)]
                            [T
                              NIL])
                        (LIST 'U COUNT)]))]
                [(IS_KNOWN_TO_BE_TRUE (CAR DT))
```

```
  (PRINT_LINE SPACE_COUNT
      (CAR DT)
      '"is known to be true")
  (INFER_LIST PARM
      (CADR DT)
      (ADD1 COUNT) SPACE_COUNT NIL)]
[(IS_KNOWN_TO_BE_FALSE (CAR DT))
  (PRINT_LINE SPACE_COUNT
      (CAR DT)
      '"is known to be false")
  (INFER_LIST PARM
      (CADDR DT)
      (ADD1 COUNT) SPACE_COUNT NIL)]
[(IS_KNOWN_TO_BE_UNDET (CAR DT))
  (PRINT_LINE SPACE_COUNT
      (CAR DT)
      '"is known to be undeterminable")
  (CONS 'U
      (LIST (ADD1 COUNT)))]
[T
  (COND
      [(NULL (MEMBERS (CAR DT)
                      IM_WORKING_ON))
        (SETQ IM_WORKING_ON
           (CONS (CAR DT) IM_WORKING_ON))
        (TERPRI)
        (SPACES SPACE_COUNT)
        (PRINC '"Attempting to satisfy ")
        (PRIN (CAR DT))
        (COND
            [ECHO_ON?
              (TERPRI NIL IELOG)
              (SPACES SPACE_COUNT IELOG)
              (PRINC '"Attempting to
                 satisfy " IELOG)
              (PRIN (CAR DT) IELOG)]
            [T
              NIL])
        (COND
            [(EQUAL 'T
                 (CAR (SETQ TEMP
                    (INFER_LIST (CAR DT)
                          (EVAL (READLIST
                             (CONS '"*"
              (EXPLODE (CAR DT))))) COUNT
                (+ 2 SPACE_COUNT) NIL))))
              (PRINT_LINE SPACE_COUNT
                  (CAR DT)
               '"is deduced to be true")
              (SETQ IM_WORKING_ON
             (DEL_CURR_GOAL IM_WORKING_ON
                     (CAR DT)))
              (INFER_LIST PARM
                  (CADR DT)
```

```
                                        (ADD1 (CADR TEMP))
                                   SPACE_COUNT NIL)]
                                 [(EQUAL 'F
                                        (CAR TEMP))
                                   (PRINT_LINE SPACE_COUNT
                                        (CAR DT)
                                    '"is deduced to be false")
                                   (SETQ IM_WORKING_ON
                                  (DEL_CURR_GOAL IM_WORKING_ON
                                           (CAR DT)))
                                   (INFER_LIST PARM
                                        (CADDR DT)
                                        (ADD1 (CADR TEMP))
                                   SPACE_COUNT NIL)]
                                 [T
                                   (LIST 'U
                                        (CADR TEMP))])]
                           [T
                             (PRINT_LINE SPACE_COUNT
                                  (CAR DT)
                                  '"is involved in a cycle")
                             (CONS 'U
                                  (LIST (ADD1 COUNT)))])]))]])
                            OPPOSITE
```

```
Arguments :  TRUTH_AND_COUNT -- a pair, or list of two
             elements, containing the "truth" of a
             parameter and the count of the number of IFs
             processed in extablishing the truth.
Called by :  INFER
Calls     :  builtin functions only
RETURNS   :  the negation of the parameter and the same
             count
```

```
(DEF 'OPPOSITE
'[LAMBDA (TRUTH_AND_COUNT)
  (COND
      [(ATOM TRUTH_AND_COUNT)
        (PRINT '"malformed truth and count value")
        (COND
            [ECHO_ON?
        (PRINT '"malformed truth and count value" IELOG)]
            [T
              NIL])
        (NIL)]
      [(EQUAL 'T
           (CAR TRUTH_AND_COUNT))
        (CONS 'F
            (CDR TRUTH_AND_COUNT))]
      [(EQUAL 'F
           (CAR TRUTH_AND_COUNT))
        (CONS 'T
            (CDR TRUTH_AND_COUNT))]
      [(EQUAL 'U
```

```
            (CAR TRUTH_AND_COUNT))
         TRUTH_AND_COUNT]
      [T
         (PRINT '"malformed truth and count value")
         (COND
             [ECHO_ON?
          (PRINT '"malformed truth and count value" IELOG)]
             [T
               NIL])
         (NIL)])])
```

IS_KNOWN_TO_BE_TRUE

```
Arguments :  PARM -- a parameter
Called by :  INFER, INFER2
Calls     :  builtin functions only
RETURNS   :  T if the decision tree list corresponding to
             the parameter PARM is bound to the value
             **TRUE**, corresponding to PARM being known
             to be true.  Returns NIL otherwise.
```

```
(DEF 'IS_KNOWN_TO_BE_TRUE
'[LAMBDA (PARM)
  (EQUAL '**TRUE**
      (EVAL (READLIST (CONS '"*"
                          (EXPLODE PARM)))))])
```

IS_KNOWN_TO_BE_FALSE

(Similar to IS_KNOWN_TO_BE_TRUE)

```
(DEF 'IS_KNOWN_TO_BE_FALSE
'[LAMBDA (PARM)
  (EQUAL '**FALSE**
      (EVAL (READLIST (CONS '"*"
                          (EXPLODE PARM)))))])
```

IS_KNOWN_TO_BE_UNDET

(similar to IS_KNOWN_TO_BE_TRUE)

```
(DEF 'IS_KNOWN_TO_BE_UNDET
'[LAMBDA (PARM)
  (EQUAL '**UNDET**
      (EVAL (READLIST (CONS '"*"
                          (EXPLODE PARM)))))])
```

SET_TREES

Arguments : PARMLIST -- a list of parameters, VAL -- a value
Called by : the user, itself
Calls : itself
RETURNS : the decision tree list set equal to the value VAL.

```
(DEF 'SET_TREES
'[LAMBDA (PARMLIST VAL)
  (COND
      [(NULL PARMLIST)
        'FINE]
      [T
        (SET (READLIST (CONS '"*"
                             (EXPLODE (CAR PARMLIST)))) VAL)
        (SET_TREES (CDR PARMLIST) VAL)
        'FINE])])
```

IT

Arguments : none
Called by : the user.
Calls : TREE_INIT
RETURNS : the reinitialization of decision tree lists without recompilation of rules.

```
(DEF 'IT
'[LAMBDA ()
  (TREE_INIT *PARMS*)])
```

TREE_INIT

Arguments : PARMLIST -- a list of parameters
Called by : IT, itself
Calls : itself

```
(DEF 'TREE_INIT
'[LAMBDA (PARMLIST)
  (COND
      [(NULL PARMLIST)
        'SURE]
      [T
        (SET (READLIST (CONS '"*"
                             (EXPLODE (CAR PARMLIST))))
             (EVAL (READLIST (CONS '"!"
                                   (EXPLODE (CAR PARMLIST))))))
        (TREE_INIT (CDR PARMLIST))
        'SURE])])
```

PRINT_LINE

Arguments : SPACE_COUNT -- how many spaces to indent the

```
                printed line, THING -- the thing to be printed
                MSG -- a message to be printed with the value
                of THING
Called by :     INFER_LIST, INFER2, END_CONSULTATION
Calls     :     builtin functions
RETURNS   :     a line of text to be printed as part of the
                trace of execution of the inference engine.

(DEF 'PRINT_LINE
'[LAMBDA (SPACE_COUNT THING MSG)
  [PROG ()
       (TERPRI)
       (SPACES SPACE_COUNT)
       (PRIN THING)
       (SPACES 1)
       (PRINC MSG)
       (COND
           [ECHO_ON?
             (TERPRI NIL IELOG)
             (SPACES SPACE_COUNT IELOG)
             (PRIN THING IELOG)
             (SPACES 1 IELOG)
             (PRINC MSG IELOG)]
           [T
             NIL])]])
```

END_CONSULTATION

```
Arguments :     PARM -- a parameter, PAIR -- a truth and count
                pair
Called by :     INFER
Calls     :     PRINT_LINE
RETURNS   :     prints some final information that appears
                after the inference engine has accomplished
                it can.

(DEF 'END_CONSULTATION
'[LAMBDA (PARM PAIR)
  [PROG ()
       (TERPRI)
       (COND
           [ECHO_ON?
             (TERPRI NIL IELOG)]
           [T
             NIL])
       (PRINT_LINE 0 PARM
           (COND
               [(EQUAL 'T
                     (CAR PAIR))
                  '"has been deduced to be true"]
               [(EQUAL 'F
                     (CAR PAIR))
                  '"has been deduced to be false"]
```

```
                [T
                  '"is undeterminable"]))
        (PRINC '" after ")
        (PRIN (CADR PAIR))
        (PRINC '" IFs")
        (TERPRI)
        (TERPRI)
        (COND
            [ECHO_ON?
              (PRINC '" after " IELOG)
              (PRIN (CADR PAIR) IELOG)
              (PRINC '" IFs" IELOG)
              (TERPRI NIL IELOG)]
            [T
              NIL])
        (RETURN 'CONSULTATION_ENDED)]])
```

NEGATION

Arguments	:	ANT -- an antecedent
Called by	:	INFER
Calls	:	builtin functions only
RETURNS	:	the negation of the parameter (antecedent). This function allows a user to infer the value of a negated parameter. The unnegated form is inferred and the answer is returned negated. (The decision trees built by the rule compiler never contain negated parameters.)

```
(DEF 'NEGATION
'[LAMBDA (ANT)
  (COND
      [(NULL ANT)
        NIL]
      [(EQUAL (CAR (EXPLODE ANT))
            '"#")
        (READLIST (CDR (EXPLODE ANT)))]
      [T
        (READLIST (CONS '"#"
                        (EXPLODE ANT)))])])
```

ECHO

Arguments	:	FILENAME -- the name of a file (in double quotes) which is to receive the text generated by INFER, IT or SET_TREES
Called by	:	the user.
Calls	:	builtin functions only
RETURNS	:	ECHO captures keyboard input and passes that input to EVAL. However, any text that is generated is also placed in FILENAME. This continues until the user requests that the

echo be turned off, or the user invokes a function which is not one of INFER, IT or SET_TREES

```
(DEF 'ECHO
'[LAMBDA (FILENAME)
  [PROG (TEMP)
       (SETQ ECHO_ON? T)
       (SETQ IELOG
           (OUTPUT FILENAME))
       (PRINTC '"INFERENCE ENGINE LOG:" IELOG)
       (TERPRI NIL IELOG)
   LOOP
       (PRINTC '"+")
       (PRINTC '"+" IELOG)
       (SETQ TEMP
           (READ))
       (COND
           [(EQUAL TEMP
                'ECHO_OFF)
             (SETQ ECHO_ON? NIL)
             (PRINC '"END OF LOG." IELOG)
             (CLOSE IELOG)
             (TERPRI)
             (RETURN 'ECHO_OFF)]
           [(OR (ATOM TEMP)
                (EQUAL (CAR TEMP)
                    'INFER)
                (EQUAL (CAR TEMP)
                    'SET_TREES)
                (EQUAL (CAR TEMP)
                    'IT))
             (PRIN TEMP IELOG)
             (PRINT (PRIN (EVAL TEMP)) IELOG)
             (TERPRI)
             (TERPRI NIL IELOG)
             (GO LOOP)]
           [T
             (SETQ ECHO_ON? NIL)
             (PRINC '"END OF LOG." IELOG)
             (CLOSE IELOG)
             (PRIN (EVAL TEMP))
             (TERPRI)
             (TERPRI)
             (RETURN 'ECHO_OFF)])]])
```

DEL_CURR_GOAL

Arguments : LIST -- the list of parameter currently being inferred which is IM_WORKING_ON, ELE -- the parameter that has just been inferred
Called by : INFER2
Calls : builtin functions only
RETURNS : an updated IM_WORKING_ON list

```
(DEF 'DEL_CURR_GOAL
'[LAMBDA (LIST ELE)
  (COND
      [(NULL LIST)
        NIL]
      [(EQUAL (CAR LIST) ELE)
        (CDR LIST)]
      [T
        (CONS (CAR LIST)
            (DEL_CURR_GOAL (CDR LIST) ELE))])])
```

PACKAGE "RULE_BUILDER.LSP"

K_B

Arguments : NC -- number of complex rules, NS -- number of simple rules, NP -- number of parameters to be concluded about, FILENAME -- file in double quotes to send rules to when generated
Called by : the user
Calls : KNOWLEDGE_BASE
RETURNS : the knowledge base

```
(DEF 'K_B
[LAMBDA (NC NS NP FILENAME HEURISTIC_NUMB)
  [PROG (RESULTS)
       (SETQ ANT_INFO
           (ARRAY 2 105 8))
       (SETQ LIST_OF_AVER_ANT NIL)
       (SETQ TL_RULES 0)
       (SETQ LIST_OF_ASKFIRST
        '((a)(b)(c)(d)(e)(#a)(#b)(#c)(#d)(#e)))
       (SETQ RESULTS
           (APPEND (KNOWLEDGE_BASE NC NS NP 1)
               (KNOWLEDGE_BASE 4 0 26 6)))
       (SETQ OP
           (OUTPUT FILENAME))
       (PP RESULTS OP)
       (CLOSE OP)
       (RETURN RESULTS)]])
```

KNOWLEDGE_BASE

Arguments : N_COMPLEX -- number of complex rules, N_SIMPLE -- number of simple rules, N_CONCL -- number of conclusions, BEG -- place in alphabet to begin
Called by : K_B
Calls : K_B_CREATOR
RETURNS : the knowledge base

```
(DEF 'KNOWLEDGE_BASE
'[LAMBDA (N_COMPLEX N_SIMPLE N_CONCL BEG)
  (K_B_CREATOR BEG N_CONCL N_COMPLEX N_SIMPLE)])
```

K_B_CREATOR

Arguments : L -- parameter will be generating rules about, NUMBER_OF_PAR -- total number of conclusions, N_COMPLEX -- number of complex rules, N_SIMPLE -- number of simple rules
Called by : KNOWLEDGE_BASE

```
Calls       :  GENERATE_RULES, itself
RETURNS     :  knowledge base

(DEF 'K_B_CREATOR
'[LAMBDA (L NUMBER_OF_PAR N_COMPLEX N_SIMPLE)
  (COND
      [(LE L NUMBER_OF_PAR)
        (APPEND (LIST (GENERATE_RULES 2 L N_COMPLEX NIL))
            (K_B_CREATOR (+ 1 L) NUMBER_OF_PAR N_COMPLEX
            N_SIMPLE))]
      [T
        NIL])])
```

MEMBERS

```
Arguments  :  ELEMENT -- an antecedent to be tested, LST --
              the list being tested for ELEMENT
Called by  :  CONSISTENCY
CALLS      :  MEMBER
RETURNS    :  T if ELEMENT is in LST, NIL otherwise

(DEF 'MEMBERS
'[LAMBDA (ELEMENT LST)
  (COND
      [(NULL (MEMBER ELEMENT LST))
        NIL]
      [T
        T])])
```

GENERATE_RULES

```
Arguments  :  N_ANT -- lower bound of interval of antecedent
              L -- current conclusion, N_RULES -- number of
              rules to generate for this conclusion
Called by  :  K_B_CREATOR
Calls      :  RANDOM_NUMBER, CREATE_LIST, COUNT_ANT
RETURNS    :  the list of rules concluding about L

(DEF 'GENERATE_RULES
'[LAMBDA (N_ANT L N_RULES FLAG)
  [PROG (I J K M N)
      (SETQ I 1)
   LOOP
      (SETQ J
          (FIX (+ N_ANT
                  (* (RANDOM_NUMBER) 3))))
      (SETQ K 1)
      (SETQ N NIL)
      (COND
          [(LE I N_RULES)
            (SETQ M
                (CONS (CONS (SETQ LIST_OF_ANT
```

```
                    (CREATE_LIST J K N L))
                  (LIST (READLIST (LIST (ASCII (+ L 64))
                                          )))) M))
          (SETQ TL_RULES
              (ADD1 TL_RULES))
          (COUNT_ANT TL_RULES
              (+ L 64) J LIST_OF_ANT)
          (SETQ I
              (+ I 1))
          (GO LOOP)]
        [T
          (SETQ LIST_OF_AVER_ANT
              (APPEND (LIST (CONS (READLIST (LIST
               (ASCII (+ L 64))))
                       (LIST (FIND_AVER_ANT 1 N_RULES
                               )))) LIST_OF_AVER_ANT))
          (NOTE_LIST_OF_ANT_USED M)
          (SETQ LIST_SO_FAR NIL)
          (RETURN (REVERSE M))]))]])
```

COUNT_ANT

```
Arguments :  ARRAY_NUMBER -- array number in ANT_INFO,
             CONCLUS -- current conclusion, MAX_NO_ANT --
             number of upper case letters in rule,
             LIST_OF_ANT --  list of antecedents in rule
Called by :  GENERATE_RULES
Calls     :  NUMBER_OF_LOWER_CASE
RETURNS   :  NIL, the purpose is to store information in
             ANT_INFO

(DEF 'COUNT_ANT
'[LAMBDA (ARRAY_NUMBER CONCLUS MAX_NO_ANT LIST_OF_ANT)
  [PROG ()
      (STORE (ANT_INFO ARRAY_NUMBER
                1) CONCLUS)
      (STORE (ANT_INFO ARRAY_NUMBER
                2)
          (NUMBER_OF_LOWER_CASE LIST_OF_ANT 0))
      (STORE (ANT_INFO ARRAY_NUMBER
                3) MAX_NO_ANT)
      (COND
          [(GT (- MAX_NO_ANT
                  (ANT_INFO ARRAY_NUMBER 2)) 0)
            (STORE_REQ_ANTS 1
                (- MAX_NO_ANT
                  (ANT_INFO ARRAY_NUMBER 2)) ARRAY_NUMBER)])
      (RETURN)]])
```

NUMBER_OF_LOWER_CASE

```
Arguments :  LIST_OF_ANT -- list of antecedents in rule,
             NUMBER -- counter for the number of lower case
```

```
                antecedents in rule
Called by :     COUNT_ANT
Calls     :     itself
RETURNS   :     the number of lower case antecedents in rule

(DEF 'NUMBER_OF_LOWER_CASE
'[LAMBDA (LIST_OF_ANT NUMBER)
  (COND
      [(NULL LIST_OF_ANT)
        NUMBER]
      [(GE (CHRVAL (CAR LIST_OF_ANT)) 97)
        (NUMBER_OF_LOWER_CASE (CDR LIST_OF_ANT)
            (ADD1 NUMBER))]
      [(EQUAL '"#"
           (CAR (EXPLODE (CAR LIST_OF_ANT))))
        (COND
            [(GE (CHRVAL (READLIST (CDR (EXPLODE (CAR
                                        LIST_OF_ANT))))) 97)
              (NUMBER_OF_LOWER_CASE (CDR LIST_OF_ANT)
                  (ADD1 NUMBER))]
            [T
              (NUMBER_OF_LOWER_CASE (CDR LIST_OF_ANT)
                                                  NUMBER)])]
      [T
        (NUMBER_OF_LOWER_CASE (CDR LIST_OF_ANT) NUMBER)])])
```

CREATE_LIST

```
Arguments :     J -- number of antecedents for this rule,
                K -- counter of number of antecedents gener-
                ated so far, N -- NIL, L -- current conclu-
                sion
Called by :     GENERATE_RULES
Calls     :     itself, CONSISTENCY, NUMBER_REFORM
RETURNS   :     the list of antecedents for a rule

(DEF 'CREATE_LIST
'[LAMBDA (J K N L)
  (COND
      [(LE K J)
        (CONS (SETQ TEMP
                   (CONSISTENCY L N))
            (CREATE_LIST J
                (+ 1 K)
                (CONS (CHRVAL (NUMBER_REFORM TEMP)) N) L))]
      [T
        NIL])])
```

NUMBER_REFORM

```
Arguments :     TEMP -- an antecedent
Called by :     CREATE_LIST
```

```
Calls       :  LISP buildin functions
RETURNS     :  its purpose is to strip the "not" symbol from
               an antecedent to be used in the list of ants
               generated for the rule so far

(DEF 'NUMBER_REFORM
'[LAMBDA (TEMP)
  (COND
      [(EQUAL '"#"
           (CAR (EXPLODE TEMP)))
        (READLIST (CDR (EXPLODE TEMP)))]
      [T
        TEMP])])
```

CONSISTENCY

```
Arguments  :  L -- the current conclusion, N -- the list of
              antecedents generated so far
Called by  :  CREATE_LIST
Calls      :  MEMBERS, TEST_FOR_NOT, LOOK_AT_ANTS_USED_BY
RETURNS    :  an antecedent

(DEF 'CONSISTENCY
'[LAMBDA (L N)
  [PROG (CONCL)
      (SETQ CONCL
          (+ 64 L))
   LOOP
      (SETQ PRELIM
          (FIX (+ 56
                   (* (RANDOM_NUMBER) 35))))
      (COND
          [(EQUAL PRELIM CONCL)
            (GO LOOP)]
          [(MEMBERS PRELIM N)
            (GO LOOP)]
          [(LT PRELIM 70)
            (SETQ PRELIM
                (+ PRELIM 41))
            (COND
                [(MEMBERS PRELIM N)
                  (GO LOOP)]
                [T
                      (RETURN (TEST_FOR_NOT PRELIM))])]
          [(LE PRELIM CONCL)
            (COND
                [(EQUAL (LOOK_AT_ANTS_USED_BY (READLIST
                   (LIST (ASCII PRELIM)))
                         (READLIST (LIST (ASCII CONCL))))
                     'T)
                  (GO LOOP)]
                [T
                  (RETURN (TEST_FOR_NOT PRELIM))])]
```

```
          [T
            (RETURN (TEST_FOR_NOT PRELIM))])]])
```

TEST_FOR_NOT

```
Arguments :  PRELIM -- the antecedent just generated
Called by :  CONSISTENCY
Calls     :  LISP builtin functions
RETURNS   :  a negated antecedent averaging 1 in 10
```

```
DEF 'TEST_FOR_NOT
[LAMBDA (PRELIM)
  (COND
      [(EQUAL (FIX (+ 1
                         (* (RANDOM_NUMBER) 10))) 1)
        (SETQ PRELIM
            (READLIST (CONS '"#"
             (EXPLODE (READLIST (LIST (ASCII PRELIM
                                                      )))))))]
      [T
        (SETQ PRELIM
            (READLIST (LIST (ASCII PRELIM))))])])
```

PRINT_STATS

```
Arguments :  FILENAME -- filename, in double quotes, of file
             that will contain the information in ANT_INFO
Called by :  the user
Calls     :  PRIN_ARR
RETURNS   :  outputs the contents of ANT_INFO
```

```
(DEF 'PRINT_STATS
'[LAMBDA (FILENAME)
  [PROG (K)
      (SETQ OF
          (OUTPUT FILENAME))
      (SETQ K 1)
   LOOP
      (COND
          [(LT K 105)
            (PRINT_ARR 1 7 K)
            (SETQ K
                (ADD1 K))
            (GO LOOP)]
          [T
            (CLOSE OF)])
      (RETURN)]])
```

PRIN_ARR

```
Arguments :   INDEX -- counter, MAX -- number of columns,
              K -- current row in array
Called by :   PRINT_STATS
Calls     :   itself, builtin functions
RETURNS   :   prints elements of array ANT_INFO to a file
```

```
(DEF 'PRIN_ARR
'[LAMBDA (INDEX MAX K)
  (COND
      [(LE INDEX MAX)
        (SETQ ANS
            (ANT_INFO K INDEX))
        (PRIN ANS OF)
        (SPACES 3 OF)
        (PRINT_ARR (ADD1 INDEX) MAX K)]
      [T
        (TERPRI NIL OF)])])
```

LOOK_AT_ANTS_USED_BY

```
Arguments :   ANT_CANDIDATE -- antecedent candidate,
              CURR_CONCL -- conclusion of the rule the
              antecedent is being tested for
Called by :   CONSISTENCY
Calls     :   FIND_CORRECT_ANT_LIST
RETURNS   :   T if the current conclusion appears as an
              antecedent in a rule concluding about the
              ANT_CANDIDATE, NIL otherwise
```

```
(DEF 'LOOK_AT_ANTS_USED_BY
'[LAMBDA (ANT_CANDIDATE CUR_CONCL)
  (FIND_CORRECT_ANT_LIST ANT_CANDIDATE CUR_CONCL
  LIST_OF_ANT_&_THEIR_CONCL)])
```

FIND_CORRECT_ANT_LIST

```
Arguments :   ANT_CANDIDATE -- antecedent candidate,
              CUR_CONCL -- conclusion of current rule,
              LIST --list of conclusions and the antecedents
              used by them
Called by :   LOOK_AT_ANTS_USED_BY
Calls     :   itself, builtin functions
RETURNS   :   T if CUR_CONCL appears as an antecedent in a
              rule concluding about ANT_CANDIDATE, NIL
              otherwise
```

```
(DEF 'FIND_CORRECT_ANT_LIST
'[LAMBDA (ANT_CANDIDATE CUR_CONCL LIST)
  (COND
      [(NULL LIST)
```

```
  NIL]
[(EQUAL (CAAR LIST) ANT_CANDIDATE)
  (COND
      [(NULL (MEMBER CUR_CONCL
                     (CDAR LIST)))
        NIL]
      [T
        T]))]
[T
  (FIND_CORRECT_ANT_LIST ANT_CANDIDATE CUR_CONCL
      (CDR LIST))])])
```

NOTE_LIST_OF_ANT_USED

```
Arguments :  LIST_OF_RULES --  the list of rules concluding
             about the same parameter
Called by :  GENERATE_RULES
Calls     :  itself
RETURNS   :  a list of the antecedents used in the rules
             concluding about one parameter
```

```
(DEF 'NOTE_LIST_OF_ANT_USED
'[LAMBDA (LIST_OF_RULES)
  (SETQ LIST_OF_ANT_&_THEIR_CONCL
      (APPEND (LIST (APPEND (LIST (READLIST (LIST (ASCII
       (+ L 64)))))
                 (MAKE_LIST_FROM_RULES LIST_OF_RULES)))
      LIST_OF_ANT_&_THEIR_CONCL))])
```

MAKE_LIST_FROM_RULES

```
Arguments :  RULELIST
Called by :  NOTE_LIST_OF_ANT_USED
Calls     :  PASS_ANT_LIST, itself
RETURNS   :  the list of antecedents used in the rules
```

```
(DEF 'MAKE_LIST_FROM_RULES
'[LAMBDA (RULELIST)
  (COND
      [(NULL RULELIST)
        LIST_SO_FAR]
      [T
        (PASS_ANT_LIST (CAAR RULELIST))
        (MAKE_LIST_FROM_RULES (CDR RULELIST))])])
```

PASS_ANT_LIST

```
Arguments :  ANTLIST -- list of antecedents of one rule
Called by :  MAKE_LIST_FROM_RULES
Calls     :  ANT_TO_CHECK, itself
RETURNS   :  an updated list of antecedents
```

```
(DEF 'PASS_ANT_LIST
'[LAMBDA (ANTLIST)
  (COND
      [(NULL ANTLIST)
        LIST_SO_FAR]
      [(ANT_TO_CHECK (CAR ANTLIST))
        (SETQ LIST_SO_FAR
            (CONS ANT_TO_ADD LIST_SO_FAR))
        (PASS_ANT_LIST (CDR ANTLIST))]
      [T
        (PASS_ANT_LIST (CDR ANTLIST))])])
```

ANT_TO_CHECK

Arguments : ANT -- antecedent to check
Called by : PASS_ANT_LIST
Calls : MEMBER
RETURNS : a list of antecedents with ANT added to it if it is not already present. The list contains antecedents with no negation symbols

```
(DEF 'ANT_TO_CHECK
'[LAMBDA (ANT)
  (COND
      [(EQUAL '"#"
            (CAR (EXPLODE ANT)))
        (COND
            [(NULL (MEMBER (READLIST (CDR (EXPLODE ANT)))
                    LIST_SO_FAR))
              (SETQ ANT_TO_ADD
                  (READLIST (CDR (EXPLODE ANT))))
              T]
            [T
              NIL])]
      [T
        (COND
            [(NULL (MEMBER ANT LIST_SO_FAR))
              (SETQ ANT_TO_ADD ANT)
              T]
            [T
              NIL])])])
```

RANDOM_NUMBER

Arguments : none
Called by : CONSISTENCY
Calls : builtin functions
RETURNS : a random number between 0 and 1

```
(DEF 'RANDOM_NUMBER
'[LAMBDA ()
  [PROG ()
```

```
(SETQ MULT 25211)
(SETQ BASE 32768)
(SETQ SEED
    (% (* MULT SEED) BASE))
(RETURN (/ (FNORM SEED 0)
              (FNORM BASE 0)))]])
```

PACKAGE "RULES.LSP"

FIND_ALL

```
Arguments :  ANT -- an antecedent, RULESET -- a rule
                                               partition
Called by :  COMPILE, itself
Calls     :  ANT_IS_IN_RULE, itself
RETURNS   :  the list of all rules which have ANT as an
           antecedent.

(DEF 'FIND_ALL
'[LAMBDA (ANT RULESET)
  (COND
      [(NULL RULESET)
        NIL]
      [(ANT_IS_IN_RULE ANT
           (CAR RULESET))
        (CONS (CAR RULESET)
            (FIND_ALL ANT
                (CDR RULESET)))]
      [T
        (FIND_ALL ANT
            (CDR RULESET))])])
```

ANT_IS_IN_RULE

```
Arguments :  ANT -- an antecedent, RULE -- a rule
Called by :  FIND_ALL
Calls     :  ANT_IS_IN_LIST
RETURNS   :  Predicate - T if the antecedent is present in
                       in the rule, NIL otherwise.

(DEF 'ANT_IS_IN_RULE
'[LAMBDA (ANT RULE)
  (ANT_IS_IN_LIST ANT
      (CAR RULE))])
```

ANT_IS_IN_LIST

```
Arguments :  ANT -- an antecedent,  LIST_ANT -- a list of
                                   antecedents
Called by : ANT_IS_IN_RULE, itself
Calls     : itself
RETURNS   : Predicate.  T if the antecedent is in the list
                      of antecedents, NIL otherwise.

(DEF 'ANT_IS_IN_LIST
'[LAMBDA (ANT LIST_ANT)
  (COND
      [(NULL LIST_ANT)
```

```
    NIL]

  [(EQUAL (CAR LIST_ANT) ANT)
    T]
  [T
    (ANT_IS_IN_LIST ANT
        (CDR LIST_ANT))])])
```

DEL_RULE

```
Arguments  :  ANT -- an antecedent, RULE -- a rule
Called by  :  DEL_RULESET
Calls      :  DEL_ANT_LIST
RETURNS    :  The rule which is RULE without antecedent ANT
```

```
(DEF 'DEL_RULE
'[LAMBDA (ANT RULE)
  (COND
      [(NULL RULE)
        NIL]
      [T
        (CONS (DEL_ANT_LIST ANT
                  (CAR RULE))
            (CDR RULE))])])
```

DEL_ANT_LIST

```
Arguments  :  ANT -- an antecedent,  ALIST -- an antecedent
              list
Called by  :  DEL_RULE, itself
Calls      :  itself
RETURNS    :  the antecedent list which is ALIST with all of
              the occurrences of ANT removed.
```

```
(DEF 'DEL_ANT_LIST
'[LAMBDA (ANT ALIST)
  (COND
      [(NULL ALIST)
        NIL]
      [(EQUAL (CAR ALIST) ANT)
        (DEL_ANT_LIST ANT
            (CDR ALIST))]
      [T
        (CONS (CAR ALIST)
            (DEL_ANT_LIST ANT
                (CDR ALIST)))])])
```

DEL_RULESET

Arguments : ANT -- an antecedent, RULESET -- a rule partition (list of rules)
Called by : COMPILE, itself
Calls : DEL_RULE, itself
RETURNS : the rule set which is RULESET with ANT removed from the antecedent list of each rule.

```
(DEF 'DEL_RULESET
'[LAMBDA (ANT RULESET)
  (COND
      [(NULL RULESET)
        NIL]
      [T
        (CONS (DEL_RULE ANT
                  (CAR RULESET))
            (DEL_RULESET ANT
                (CDR RULESET)))])])
```

PACKAGE "HEURS1.LSP"

SELECT_A_CLAUSE

Arguments : RULESET -- a rule partition, or list of rules concluding about the same parameter
Called by : COMPILE
Calls : MOST_OFTEN_OCCURRING, RULE_GROUP_TALLY
RETURNS : an antecedent which will be placed nearest the top of a decision tree. In this package the heuristics used is the antecedent that appears in the most rules will be placed nearest the root of the tree.

```
(DEF 'SELECT_A_CLAUSE
'[LAMBDA (RULESET)
  [PROG ()
      (SETQ SET
     (CHECK_RULESET_FOR_ASKFIRST LIST_OF_ASKFIRST RULESET))
      (COND
          [(NULL SET)
            (RETURN (MOST_OFTEN_OCCURRING (RULE_GROUP_TALLY
                                                    RULESET)))]
          [T
       (RETURN (MOST_OFTEN_OCCURRING (RULE_GROUP_TALLY SET)
                           ))])]]
```

MOST_OFTEN_OCCURRING

Arguments : TALLYLIST -- a list of antecedent/count pairs
Called by : SELECT_A_CLAUSE
Calls : MOST
RETURNS : the antecedent whose count in tallylist is the greatest. If two antecedents tie, the "leftmost" one in TALLYLIST is returned

```
(DEF 'MOST_OFTEN_OCCURRING
'[LAMBDA (TALLYLIST)
  (MOST TALLYLIST NIL 0)])
```

MOST

Arguments : TLIST -- a tally list, CLAUSE -- the clause which has been determined to be the most often occurring (so far), COUNT -- the count of this parameter
Called by : MOST_OFTEN_OCCURRING, itself
Calls : itself
RETURNS : the antecedent with the highest count. MOST does the work described under MOST_OFTEN_OCCUR.

```
(DEF 'MOST
'[LAMBDA (TLIST CLAUSE COUNT)
  (COND
      [(NULL TLIST)
        CLAUSE]
      [(GT (CADAR TLIST) COUNT)
        (MOST (CDR TLIST)
            (CAAR TLIST)
            (CADAR TLIST))]
      [T
        (MOST (CDR TLIST) CLAUSE COUNT)])])
```

TALLY

Arguments : ANTECEDENT -- an antecedent which we want to update tally for, TALLYLIST -- a list of antecedent/count pairs
Called by : LIST_TALLY, itself
Calls : NEGATION, itself
RETURNS : a tally list which contains an updated entry for ANTECEDENT. If there was no entry for ANTECEDENT, then one is built with a count of 1. If a negated antecedent is found, the entry is updated for the unnegated antecedent

```
(DEF 'TALLY
'[LAMBDA (ANTECEDENT TALLYLIST)
  (COND
      [(NULL TALLYLIST)
        (LIST (LIST ANTECEDENT 1))]
      [(EQUAL (CAAR TALLYLIST) ANTECEDENT)
        (CONS (LIST ANTECEDENT
                (ADD1 (CADAR TALLYLIST)))
            (CDR TALLYLIST))]
      [(EQUAL (CAAR TALLYLIST)
           (NEGATION ANTECEDENT))
        (CONS (LIST ANTECEDENT
                (ADD1 (CADAR TALLYLIST))))]
      [T
        (CONS (CAR TALLYLIST)
            (TALLY ANTECEDENT
                (CDR TALLYLIST)))])])
```

LIST_TALLY

Arguments : ANT_LIST -- a list of antecedents, TLIST -- a tally list
Called by : RULE_TALLY, itself
Calls : TALLY, itself
RETURNS : performs the TALLY function for each antecedent in ANT_LIST

```
(DEF 'LIST_TALLY
'[LAMBDA (ANT_LIST TLIST)
  (COND
      [(NULL ANT_LIST)
        TLIST]
      [T
        (LIST_TALLY (CDR ANT_LIST)
            (TALLY (CAR ANT_LIST) TLIST))])])
```

RULE_TALLY

```
Arguments :  RULE -- a rule of the form (antecedent-list
             consequent), TLIST -- a tally list
Called by :  RULE_GROUP_TALLY
Calls     :  LIST_TALLY
RETURNS   :  calls LIST_TALLY to process its antecedent
             list
```

```
(DEF 'RULE_TALLY
'[LAMBDA (RULE TLIST)
  (COND
      [(NULL RULE)
        RULE]
      [T
        (LIST_TALLY (CAR RULE) TLIST)])])
```

RULE_GROUP_TALLY

```
Arguments :  R_GROUP -- a rule group (a list of rules)
Called by :  SELECT_A_CLAUSE, itself
Calls     :  RULE)TALLY, itself
RETURNS   :  a tally list of all the antecedents used in
             R_GROUP.  This function processes each rule
             by successively calling RULE_TALLY for each
             rule in R_GROUP
```

```
(DEF 'RULE_GROUP_TALLY
'[LAMBDA (R_GROUP)
  (COND
      [(NULL R_GROUP)
        NIL]
      [T
        (RULE_TALLY (CAR R_GROUP)
            (RULE_GROUP_TALLY (CDR R_GROUP)))])])
```

CHECK_ANT_LIST

```
Arguments : AF -- a list of known parameters (askfirst),
            ANT_LIST -- a list of antecedents of a rule
Called by :
Calls     :  ANT_IS_IN_LIST
```

```
RETURNS     :  T if there is an askfirst parameter, NIL
               if there is not

(DEF 'CHECK_ANT_LIST
'[LAMBDA (AF ANT_LIST)
  (COND
      [(NULL ANT_LIST)
        NIL]
      [(ANT_IS_IN_LIST (CAR ANT_LIST) AF)
        T]
      [T
        (CHECK_ANT_LIST AF
            (CDR ANT_LIST))])])
```

LOOK_AT_RULE

```
Arguments  :  AF -- list of known parameters, RULE -- rule
              being examined
Called by  :
Calls      :  CHECK_ANT_LIST
RETURNS    :

(DEF 'LOOK_AT_RULE
'[LAMBDA (AF RULE)
  (CHECK_ANT_LIST AF
     (CAR RULE))])
```

CHECK_RULESET_FOR_ASKFIRST

```
Arguments  :  AF -- list of known parameters, RULESET -- a
              list of rules
Called by  :
Calls      : LOOK_AT_RULE, itself
RETURNS    :

(DEF 'CHECK_RULESET_FOR_ASKFIRST
'[LAMBDA (AF RULESET)
  (COND
      [(NULL RULESET)
        NIL]
      [(LOOK_AT_RULE AF
           (CAR RULESET))
        (CONS (CAR RULESET)
            (CHECK_RULESET_FOR_ASKFIRST AF
                (CDR RULESET)))]
      [T
        (CHECK_RULESET_FOR_ASKFIRST AF
            (CDR RULESET))])])
```

PACKAGE "HEURIS2.LSP"

SELECT_A_CLAUSE

```
Arguments :  RULESET -- a partition list
Called by :  COMPILE
Calls     :  TRAVEL_LIST_MIN_ANT
RETURNS   :  an antecedent to be put out as a branch
```

```
(DEF 'SELECT_A_CLAUSE
'[LAMBDA (RULESET)
  (COND
      [(NULL (SETQ CLAUSE
        (TRAVEL_LIST_MIN_ANT RULESET LIST_OF_ASKFIRST)))
        (TRAVEL_LIST_MIN_ANT RULESET LIST_OF_MIN_ANT)]
      [T
        CLAUSE])])
```

CREATE_AVERAGES

```
Arguments :  none
Called by :  K_B
Calls     :  REORDER, NEW_AVERAGES, ADD_NOTS
RETURNS   :  an updated LIST_OF_AVER_ANT after calculating
             averages three times
```

```
(DEF 'CREATE_AVERAGES
'[LAMBDA ()
  [PROG ()
       (SETQ LIST_OF_ANTS NIL)
       (SETQ AVER_ONE
           (REORDER 0 4
               (LIST_OF_AVER_ANT)))
       (SETQ LIST_OF_ANTS NIL)
       (SETQ AVER_TWO
           (REORDER 0 16
               (NEW_AVERAGES 6 26 AVER_ONE)))
       (SETQ LIST_OF_ANTS NIL)
       (SETQ AVER_THREE
           (REORDER 0 64
               (NEW_AVERAGES 6 26 AVER_TWO)))
       (RETURN (ADD_NOTS AVER_THREE))]])
```

NEW_AVERAGES

```
Arguments :  ST -- letter to begin calculating averages,
             FIN -- letter to stop calculating averages (Z)
             AVER_LIST -- current list of average
             antecedents
Called by :  CREATE_AVERAGES
Calls     :  EVAL_FOR_ALL_RULES, itself
```

RETURNS : an updated list of averages

```
(DEF 'NEW_AVERAGES
'[LAMBDA (ST FIN AVER_LIST)
  (COND
      [(GT ST FIN)
        NIL]
      [T
        (SETQ NEW_SUM_&_AVER
            (EVAL_FOR_ALL_RULES 1 N_RULES ST 0 AVER_LIST))
        (COND
            [(GE (% NEW_SUM_&_AVER N_RULES) 2)
              (SETQ NEW_SUM_&_AVER
                  (+ (FIX (/ NEW_SUM_&_AVER N_RULES)) 1))]
            [T
              (SETQ NEW_SUM_&_AVER
                  (FIX (/ NEW_SUM_&_AVER N_RULES)))])
        (SETQ NEW_AVER
          (CONS (CONS (READLIST (LIST (ASCII (+ ST 64))))
                      (LIST NEW_SUM_&_AVER))
             (NEW_AVERAGES (ADD1 ST) FIN AVER_LIST)))])])
```

EVAL_FOR_ALL_RULES

Arguments : IND -- index, NUMB_RULES -- number of rules to evaluate, BLK -- previous block of rules in ANT_INFO, R_SUM -- sum of averages for rules, AVER_LIST -- current list of averages
Called by : NEW_AVERAGES
Calls : CALC_NEW_NUMB, itself
RETURNS : the new sum of antecedents

```
(DEF 'EVAL_FOR_ALL_RULES
'[LAMBDA (IND NUMB_RULES BLK R_SUM AVER_LIST)
  (COND
      [(GT IND NUMB_RULES)
        R_SUM]
      [T
        (COND
            [(EQUAL (SETQ N_ANTS
                          (- (ANT_INFO (+ IND BLK) 3)
                             (ANT_INFO (+ IND BLK) 2))) 0)
              (EVAL_FOR_ALL_RULES (ADD1 IND) NUMB_RULES BLK
              R_SUM AVER_LIST)]
            [T
              (SETQ R_SUM
                  (+ R_SUM
                     (CALC_NEW_NUMB (+ IND BLK) N_ANTS 0
                     AVER_LIST)))
             (EVAL_FOR_ALL_RULES (ADD1 IND) NUMB_RULES BLK
              R_SUM AVER_LIST)])])])
```

CALC_NEW_NUMB

```
Arguments :  ARR_NO -- array number, N_ANTS -- number of
             antecedents that have to be inferred, ANT_SUM
             -- sum of antecedents, AVER_LIST -- current
             average of antecedents
Called by :  EVAL_FOR_ALL_RULES
Calls     :  GET_NUMB_OF_NEEDED_ANTS, itself
RETURNS   :  new sum of antecedents
```

```
(DEF 'CALC_NEW_NUMB
'[LAMBDA (ARR_NO N_ANTS ANT_SUM AVER_LIST)
  (COND
      [(GT N_ANTS 0)
        (SETQ ANT_SUM
            (+ (GET_NUMB_OF_NEEDED_ANT (ANT_INFO ARR_NO
                                              (+ 3 N_ANTS))
               AVER_LIST) ANT_SUM))
        (CALC_NEW_NUMB ARR_NO
            (SUB1 N_ANTS) ANT_SUM AVER_LIST)]
      [T
        ANT_SUM])])
```

REORDER

```
Arguments :  ST -- lowest average in list of antecedents,
             FIN -- highest average in list of antecedents,
             LIST -- current list of averages
Called by :  CREATE_AVERAGES
Calls     :  PICK_OUT
RETURNS   :  a list of averages in ascending order
```

```
(DEF 'REORDER
'[LAMBDA (ST FIN LIST)
  (COND
      [(LE ST FIN)
        (PICK_OUT ST LIST)
        (REORDER (ADD1 ST) FIN LIST)]
      [T
        LIST_OF_ANTS])])
```

PICK_OUT

```
Arguments :  THESE_CONCL -- current average, LIST -- list
             of averages
Called by :  REORDER
Calls     :  itself
RETURNS   :  list of conclusions with the current average
```

```
(DEF 'PICK_OUT
'[LAMBDA (THESE_CONCL LIST)
  (COND
```

```
[(NULL LIST)
  LIST_OF_ANTS]
[(EQUAL (CADAR LIST) THESE_CONCL)
  (SETQ LIST_OF_ANTS
      (APPEND LIST_OF_ANTS
          (LIST (CAR LIST))))
  (PICK_OUT THESE_CONCL
      (CDR LIST))]
[T
  (PICK_OUT THESE_CONCL
      (CDR LIST))])])
```

STORE_REQ_ANTS

Arguments : ST -- counter, N_ANTS -- number of antecedents to be stored, ARR_NUMB -- array number
Called by :
Calls : WHAT_THE_ANT_IS, itself
RETURNS : nothing, the purpose is to store information about the rules in ANT_INFO

```
(DEF 'STORE_REQ_ANTS
'[LAMBDA (ST N_ANTS ARR_NUMB)
  (COND
      [(LE ST N_ANTS)
        (STORE (ANT_INFO ARR_NUMB
                  (+ ST 3))
            (WHAT_THE_ANT_IS LIST_OF_ANT ST 1))
        (STORE_REQ_ANTS (ADD1 ST) N_ANTS ARR_NUMB)])])
```

WHAT_THE_ANT_IS

Arguments : LIST -- list of antecedents, N_TO_FIND-- number of antecedents to store, COUNT -- index
Called by : STORE_REQ_ANTS
Calls : itself
RETURNS : an antecedent

```
(DEF 'WHAT_THE_ANT_IS
'[LAMBDA (LIST N_TO_FIND COUNT)
  (COND
      [(NULL LIST)
        NIL]
      [(EQUAL '"#"
            (CAR (EXPLODE (CAR LIST))))
        (COND
            [(LT (CHRVAL (READLIST (CDR (EXPLODE
                  (CAR LIST))))) 97)
              (COND
                  [(EQUAL COUNT N_TO_FIND)
                    (CHRVAL (READLIST (CDR (EXPLODE
                                    (CAR LIST)))))]
```

```
                  [T
                    (WHAT_THE_ANT_IS (CDR LIST) N_TO_FIND
                        (ADD1 COUNT))]))]
           [T
           (WHAT_THE_ANT_IS (CDR LIST N_TO_FIND COUNT))]))]
     [T
       (COND
           [(LT (CHRVAL (CAR LIST)) 97)
             (COND
                 [(EQUAL COUNT N_TO_FIND)
                   (CHRVAL (CAR LIST))]
                 [T
                   (WHAT_THE_ANT_IS (CDR LIST) N_TO_FIND
                       (ADD1 COUNT))]))]
           [T
             (WHAT_THE_ANT_IS (CDR LIST) N_TO_FIND
                                             COUNT)])])])
```

FIND_AVER_ANT

Arguments : INDEX -- index to array number, N_RULES-- number of rules for a particular conclusion
Called by :
Calls : builtin functions
RETURNS : new average

```
(DEF 'FIND_AVER_ANT
'[LAMBDA (INDEX N_RULES)
  [PROG ()
       (SETQ SUM 0)
   LOOP
       (COND
           [(LE INDEX N_RULES)
             (SETQ SUM
                 (+ SUM
                   (- (ANT_INFO (+ (- TL_RULES INDEX) 1) 3)
                   (ANT_INFO (+ (- TL_RULES INDEX) 1) 2))))
             (SETQ INDEX
                 (ADD1 INDEX))
             (GO LOOP)]
           [T
             (COND
                 [(GE (% SUM N_RULES) 2)
                   (RETURN (+ (FIX (/ SUM N_RULES)) 1))]
                 [T
                   (RETURN (FIX (/ SUM N_RULES)))])])]])
```

ADD_NOTS

Arguments : ORDERED_LIST -- list of conclusions and averages
Called by : CREATE_AVERAGES

```
Calls       :  itself
RETURNS     :  the ordered list with negations

(DEF 'ADD_NOTS
 '[LAMBDA (ORDERED_LIST)
   (COND
       [(NULL (CAR ORDERED_LIST))
         NIL]
       [T
         (APPEND (APPEND (LIST (CAR ORDERED_LIST))
                      (LIST (CONS (READLIST (CONS '"#"
                            (EXPLODE (CAAR ORDERED_LIST))))
                             (LIST (CADAR ORDERED_LIST)))))
             (ADD_NOTS (CDR ORDERED_LIST)))])])
```

GET_NUMB_OF_NEEDED_ANT

```
Arguments :  ANT -- current antecedent, LIST -- list of
             antecedents and their averages
Called by :  CALC_NEW_NUMB
Calls     :  itself
RETURNS   :  the average for ANT

(DEF 'GET_NUMB_OF_NEEDED_ANT
'[LAMBDA (ANT LIST)
  (COND
      [(NULL LIST)
        NIL]
      [(GE ANT 97)
        0]
      [(EQUAL ANT
           (CHRVAL (CAAR LIST)))
        (CADAR LIST)]
      [T
        (GET_NUMB_OF_NEEDED_ANT ANT
            (CDR LIST))])])
```

TRAVEL_LIST_MIN_ANT

```
Arguments :  R_GROUP -- a partition list, LIST_ANTS -- list
             of antecedents and their averages
Called by :  SELECT_A_CLAUSE
Calls     :  LOOK_FOR_ANT_IN_RULESET, itself
RETURNS   :  nothing

(DEF 'TRAVEL_LIST_MIN_ANT
'[LAMBDA (R_GROUP LIST_ANTS)
  (COND
      [(NULL LIST_ANTS)
        NIL]
      [(LOOK_FOR_ANT_IN_RULESET (CAAR LIST_ANTS) R_GROUP)
         (CAAR LIST_ANTS)]
```

```
        [T
          (TRAVEL_LIST_MIN_ANT R_GROUP
              (CDR LIST_ANTS))])])
```

LOOK_FOR_ANT_IN_RULESET

Arguments : ANT -- an antecedent, R_GROUP -- a partition list
Called by : TRAVEL_LIST_MIN_ANT
Calls : ONE_RULE, itself
RETURNS : T if antecedent is found, NIL otherwise

```
(DEF 'LOOK_FOR_ANT_IN_RULESET
'[LAMBDA (ANT R_GROUP)
  (COND
        [(NULL R_GROUP)
          NIL]
        [(ONE_RULE ANT
              (CAR R_GROUP))
          T]
        [T
          (LOOK_FOR_ANT_IN_RULESET ANT
              (CDR R_GROUP))])])
```

ONE_RULE

Arguments : ANT -- antecedent, RULE -- a rule
Called by : LOOK_FOR_ANT_IN_RULESET
Calls : itself
RETURNS : T if a match is found between an antecedent in a rule and in the list of antecedents

```
(DEF 'ONE_RULE
'[LAMBDA (ANT RULE)
  (COND
        [(MEMBER ANT
              (CAR RULE))
          T]
        [T
          NIL])])
```