NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

EMYCIN-PROLOG EXPERT SYSTEM SHELL

by

Fikret Ulug

December 1986

Thesis Advisor: Neil C. Rowe

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
2b DECLASSIFICATION / DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Naval Postgraduate School; 6b OFFICE SYMBOL (If applicable)	7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000	7b ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000
8a NAME OF FUNDING / SPONSORING ORGANIZATION; 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO; PROJECT NO; TASK NO; WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification)

EMYCIN-PROLOG EXPERT SYSTEM SHELL

12 PERSONAL AUTHOR(S) Fikret Ulug

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Master's Thesis	FROM ______ TO ______	1986 December	189

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Expert Systems, Expert System Shell, Prolog

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

Building an expert system from scratch requires a long and tedious programming process. To make this easier, expert system shells are devised. We have implemented a shell in the language PROLOG. Our shell is modelled on a famous one, EMYCIN. We built two small-sized expert systems using our shell. The first one (CAR diagnosis system) diagnoses engine problems in a car, and the second one (FINANCE analysis system) gives financial advice. We also designed some explanation facilities for our shell. The choice of PROLOG facilitated our study considerably. PROLOG's built-in pattern-matching and backtracking facilities were two powerful features for the deduction process and EMYCIN's backward-chaining control structure. With our shell we were able to build an expert system quickly. Although they were left as a

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL Neil C. Rowe	22b TELEPHONE (Include Area Code) (408) 646 2462; 22c OFFICE SYMBOL 52Rp

19. ABSTRACT (continued)

future study, implementation of the user interaction and explanation system modules can make our shell a usable product.

EMYCIN-PROLOG Expert System Shell

by

Fikret Ülug
Lieutenant Junior Grade, Turkish Navy
B.S., Turkish Naval Academy, 1980

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1986

ABSTRACT

Building an expert system from scratch requires a long and tedious programming process. To make this easier, expert system shells are devised. We have implemented a shell in the language PROLOG. Our shell is modelled on a famous one, EMYCIN. We built two small-sized expert systems using our shell. The first one (CAR diagnosis system) diagnoses engine problems in a car, and the second one (FINANCE analysis system) gives financial advice. We also designed some explanation facilities for our shell. The choice of PROLOG facilitated our study considerably. PROLOG's built-in pattern-matching and backtracking facilities were two powerful features for the deduction process and EMYCIN's backward-chaining control structure. With our shell we were able to build an expert system quickly. Although they were left as a future study, implementation of the user interaction and explanation system modules can make our shell a usable product.

TABLE OF CONTENTS

I. INTRODUCTION 8

A. EXPERT SYSTEMS 8

B. THE EXPERT SYSTEM SHELL AND EMYCIN . . 10

C. WHY EMYCIN? 12

D. WHY PROLOG? 14

E. THE WORK DONE 14

II. THE EMYCIN-PROLOG CONSULTATION SYSTEM 18

A. INTRODUCTION 18

B. DATA STRUCTURES 18

1. The Context Tree 19

a. Introduction 19

b. Uses Of The Context Tree . . . 20

c. Internal Structure Of Contexts 22

2. Parameters 24

a. Introduction 24

b. Types Of Parameters 25

c. Internal Structure Of Parameters26

3. Rules 28

a. Introduction 28

b. Internal Structure And Definition Of Rules 30

c. Creation Of Context Instances And Rule Evaluation 32

C. INFERENCE MECHANISM 34

D. FUNCTIONS 36

E. CONSULTATION CYCLE 38

1. Detailed Analysis Of The Control Structure 38

2. Departures From The Main Control Structure 44

III. INEXACT REASONING 45

A. CERTAINTY FACTORS 46

B. COMBINING FUNCTIONS 50

IV. KNOWLEDGE ACQUISITION 52

V. EXPLANATION SYSTEM 54

A. INTRODUCTION 54

B. ACQUIRING AND STRUCTURING THE SUPPORT KNOWLEDGE 58

C. NATURAL EXPLANATION AND EXPLANATION TREE 59

D. SEMANTIC EXPLANATION NETWORK 61

E. INFERENCE NETWORK 62

VI. CONCLUSION 66

A. THE LESSONS LEARNED 66

B. REQUIRED HARDWARE AND SOFTWARE 67

C. EVALUATION OF EMYCIN 68

1. Generality Of EMYCIN 68

2. Some Particular Problems 69

D. PROLOG AND EMYCIN-PROLOG 70

E. EFFICIENCY OF EMYCIN-PROLOG 71

F. THE BENEFITS OF OUR WORK 72

APPENDIX A: SOURCE CODE 74

APPENDIX B: LIST OF FUNCTIONS 124

APPENDIX C: KNOWLEDGE BASES 127

1. CAR DIAGNOSIS SYSTEM KNOWLEDGE BASE . . 128

2. FINANCE ANALYSIS SYSTEM KNOWLEDGE BASE . 142

APPENDIX D: SAMPLE CONSULTATIONS 156

1. CAR DIAGNOSIS CONSULTATIONS. 157

2. FINANCE ANALYSIS CONSULTATIONS 163

APPENDIX E: FIGURES 169

LIST OF REFERENCES 184

BIBLIOGRAPHY 186

INITIAL DISTRIBUTION LIST 187

I. INTRODUCTION

A. EXPERT SYSTEMS

Our main goal is to translate the inference engine of EMYCIN into the PROLOG language, and run this inference engine with two different knowledge bases. The expert system shell EMYCIN and its inference engine are explained in Section I.B and I.E respectively. This section provides background information about expert systems.

One of the main interests in the area of artificial intelligence is the development of "expert systems" (ES). An ES is a large computer program which captures professional expertise in a field such as fault diagnosis, chemical analysis, or equipment design, and is capable of providing recommendations as valid as those of human experts. Some well-known expert systems are: the Heuristic DENDRAL program which finds the relatively small set of possible molecular structures of known constituent atoms that could account for the given spectroscopic analysis of an unknown molecule [1]; MACYSMA, which assists mathematicians, scientists, and engineers in solving mathematical problems; and MYCIN, which provides consultative recommendations for diagnosis and treatment of infectious disease. The MYCIN example is especially interesting in its ability to reason with "inexact" data.

Years of experience have yielded a list of prerequisities for the worth of expert systems [2,3]. Some of these prerequisities merit description. First, the program should be useful. It should respond to the actual needs of a domain. Second, the program should be able to explain its advice. It should provide the user with enough information about its reasoning to allow a decision as to whether to follow the recommendation. Finally, the program should be able to communicate naturally with the user. It should avoid confronting the user with computer jargon. It should use a language as close as possible to the natural language to permit understanding of data requests, explanations and recommendations. This would facilitate the transfer of knowledge by the knowledge engineer to the program during the knowledge-base design phase. The knowledge engineer is one of the users of EMYCIN (see following section for a discussion about the different users of an expert system).

B. THE EXPERT SYSTEM SHELL AND EMYCIN

Before the concept of the expert system shell is introduced, the principle builder of the expert system, the knowledge engineer, and his/her relationship to the expert system shell should be described. The knowledge engineer works together with the domain expert during building process. The knowledge engineer is the AI specialist while the domain expert is the specialized senior professional with respect to the domain. The relationship of the knowledge engineer and the expert system shell has been expressed as follows: "The need for a knowledge engineer is inversely proportional to the quality of the tools provided by the expert system environment" [4].

Over the years, methodologies used to build expert systems have developed similarities, and they can be categorized according to the representation of knowledge (first-order predicate calculus, semantic networks, production systems, frames [5]), and inference methods that perform reasoning on the knowledge base (generate-and-test, backward-chaining, forward-chaining). While the first generation of expert system builders used enhanced AI languages like Interlisp and PROLOG, second generation efforts concentrated on building and using languages that embody one or more of the above knowledge representation schemes and inference methods. Such languages reduce the expert system building time considerably, and they are called expert system shells. "Without such an environment, the development process would focus on programming. This burdens and lengthens the task of the knowledge engineers and decreases the quality of communication with the experts; they do not work on the same thing" [6]. EMYCIN is one such second generation expert system building language (expert system shell). Some other expert system shells are presented in detail elsewhere [7].

An expert system shell should facilitate the expression, display, organization, and interaction of thoughts. EMYCIN presents a conceptual model consisting of triples (attribute, object, value) and a context tree, designed to satisfy the above requirements. Here the conceptual model should not be confused with the language used, since EMYCIN has already been implemented with different languages such as Interlisp, and in our work, with PROLOG.

EMYCIN's task is explained by its author as follows: "EMYCIN is used to construct and run a consultation program, a program that offers advice on problems within its domain of expertise. The consultation program elicits information about a particular problem (a "case") by asking questions of a user. It then applies its knowledge to the specific facts of the case and informs the user of its conclusions. The user is free to ask the program questions about its reasoning in order to better understand or validate the advice given" [8]. Once EMYCIN is built by a <u>shell designer</u> there are two other users of it. First is the <u>knowledge engineer</u> who uses EMYCIN to produce a knowledge base for the domain. The knowledge engineer most of the time works with the domain expert (see Figure-1). The knowledge-base is composed of factual knowledge about the domain and production rules [9] showing how to go through the consultation. The third user of EMYCIN is what we call the <u>consultor</u> to whom the advice is given. Thus EMYCIN, together with the knowledge base, constructs a new consultation system. Throughout our study we will refer to the shell designer as "we" or "us".

Figure-1 shows the overall organization of the EMYCIN and interactions with different users.

C. WHY EMYCIN?

In our search for an expert system shell EMYCIN has been chosen for several different reasons. First, as a university research project compared to a commercial one, EMYCIN has increased credibility. EMYCIN in fact originated from the expert system MYCIN which diagnoses infectious diseases. MYCIN's succesful diagnostic results encouraged us to look at its structure. The builders of EMYCIN (Essential MYCIN) stripped off the domain specific knowledge of MYCIN and proposed the remaining structure as an expert system shell and also claimed its applicability for domains other than medicine.

Our primary need was for a higher-level conceptual structure which would embrace the domain knowledge in a structured way. We also needed an inference engine to operate on that knowledge as well as the implementation of these conceptual and structural requirements in reasonable hardware and software resources.

EMYCIN provides a highly organized conceptual structure into which the domain knowledge is to be mapped. It is a tree whose nodes correspond to the hierarchically organized domain-knowledge chunks. These nodes are designated contexts. Our attempt is to have a balance between the complexity of the context tree requirements and their implementation difficulties. EMYCIN is expected to provide this balance.

D. WHY PROLOG?

Programs in PROLOG consist of rules and facts, where each rule is equivalent to a Horn clause [10,11]. The entire set of facts and rules comprises the knowledge base. When this knowledge base is queried, the information which is a logical consequence of facts in the knowledge base can be retrieved. Inference is done in a top down fashion using the resolution principle [12]. PROLOG has a built-in-pattern-matching facility which is based on the unification principle [10]. Since the EMYCIN inference engine works on production rules [9], PROLOG's basic statements, which are rules, facilitate implementation of the rule based structure of EMYCIN.

Emergence of different PROLOG implementations on different machines encouraged us to work with PROLOG [13]. Also this increasing availibility and its ease of use increases the portability of our work.

E. THE WORK DONE

Our work can be seen in four different parts. The first part involves the writing of a program using PROLOG which imitates the inference engine of EMYCIN expert system shell. This phase of our work is called the shell building process. During the shell building process EMYCIN's data structures, inference mechanism, and the way of reasoning were analyzed. The second part was the building of two different knowledge-bases. The knowledge-bases are composed of production rules and all structural information that EMYCIN requires (i.e., context and parameter definitions). The third part involved running these knowledge-bases and obtaining consultations. The final part was the analyzing of the explanation system of EMYCIN.

EMYCIN is composed of three main parts: the knowledge-base construction system, the consultation-driver system (inference engine), and the explanation system (see Figure-1). The inference engine operates on the knowledge-base using EMYCIN's high level conceptual structure (context tree), the data triples [attribute, object, value (see Section II.A.1. for details)], and production rules [9]. Reasoning is done by backwards chaining, which is the main reason for choosing PROLOG as the implementation language, since it already posseses this built-in control structure.

The inference engine builds the context tree dynamically and, according to the definition of parameters (one of the elements of data triples), reasons on the production rules to find the value of the goal parameter defined by the knowledge engineer.

Two different knowledge-bases were built, namely the CAR diagnosis system and the FINANCE analysis system. Their context and parameter definitions and production rules were defined. The inference engine was run on these knowledge bases and sample consultations were recorded (see Appendix D for sample consultations).

The FINANCE analysis system originated elsewhere [14]. This sample knowledge base was chosen specifically to test our inference engine.

Following the implementation of the consultation driver system (inference engine), the EMYCIN explanation system was analyzed, its deficiencies identified, and a new system proposed. Even though the explanation system was not implemented, its basic structural elements were presented using PROLOG definitions, and a small sample of an explanation session was built for the CAR diagnosis system, again using PROLOG (see Section V).

The knowledge-base construction system provides for the acquisition of an expert's domain knowledge and storing of this knowledge, which is then ready to be processed by the consultation driver system. While this system was not implemented, requirements for the knowledge acquisition system are presented in Section IV.

While EMYCIN-PROLOG did not need some of the elements of the control structure of the EMYCIN, (e.g., the UPDATE-BY list is not used to keep track of the list of related rules for every parameter [2]), some new properties were added into the static definition of parameters (e.g., the "is_t" (is traced) property of a parameter is to keep track of whether the parameter's value is traced or not).

II. THE EMYCIN-PROLOG CONSULTATION SYSTEM

A. INTRODUCTION

In this chapter EMYCIN's data structures and inference mechanism are analyzed. Throughout our study, the PROLOG implementation of the EMYCIN inference engine is referred to as EMYCIN-PROLOG. Section II.D explains the functions used in rules and Section II.E gives a step-by-step analysis of the whole consultation cycle.

Throughout this thesis context and parameter names are printed in smaller fonts for clarity purposes (i.e., context, parameter).

B. DATA STRUCTURES

The structural aspect of the expert's problem solving strategy is reflected in the context types and their parameters. These two main elements of the system provide the language to express the expert's problem-solving methods for the domain. Besides contexts and parameters, another main component is the rules which embody domain specific knowledge. The following three sections describe the internal structure of contexts, parameters and rules and introduce the idea of a context tree.

1. The Context Tree

a. Introduction

In this section MYCIN's context tree structure is used as an example (see Figure-3/4).

The context tree forms the backbone of the consultation system by organizing the conceptual structure of the knowledge base and providing a framework for the flow of the consultation system. The tree also includes the goal for which the consultation system will try to determine a value. In our example the goal is therapy (i.e., determine the best therapy recommandation). Therapy is a parameter of the patient context.

The context tree is composed of at least one context type which corresponds to the conceptual entity in the domain. One conceptual entity from our example is the patient context. As its name implies, the context tree is structured in a tree hierarchy. Each context type in the tree resembles a record declaration in a traditional programming language. Since a context type can have more than one instantiation, the context tree has two distinct appearances. The first one corresponds to the declaration phase of a record and is called the <u>static context tree</u>. The static context tree includes every context type in it and shows their hierarchical relationship: their root context (patient), all parent/son connections (patient context is parent of the current culture context), etc. Once the consultation starts, depending upon the specific consultation, not necessarily all context types are included (e.g., therapy context is not included in the dynamic tree of the MYCIN). A given context types might have more than one instances (current culture context has two instances, culture-1 and culture-2). The resulting tree structure therefore would be quite different from the static context tree structure. This structure variation corresponds to the second context appearance and is called the <u>dynamic context tree</u> (see Figure-4). The above distinction of static and dynamic context tree is illustrated in Figure-3 and Figure-4. These samples were taken from MYCIN [2].

Hereafter, we will call the static and dynamic context trees the static tree and dynamic tree respectively.

b. Uses Of The Context Tree

It is very important to understand the purpose of the context tree. Defining contexts of a problem is not simply naming isolated physical entities. The context tree provides a way to represent multiple instances of these entities. One of the main mistakes in defining the context tree is to define contexts which have only one instance and no more. This makes the tree cumbersome and does not bring any advantage since this type of context can simply be viewed as an attribute of the root context. For example, one might want to write rules that use various attributes of a car's carburator, but since there is always exactly one carburator for a car there is no need to have a carburator context; any attribute of the carburator can be attributed to the car context.

There are three main uses of the context tree. The first use is to structure the data or evidence which is required to advise the user about the root context. In our sample system, "subsystem" contexts describe the different tests performed to locate the problem in the CAR. Also additional information about car's prior repairs are also represented in the tree. The context organization is shown on Figure-5 and Figure-6.

The second use is to specify components of some object. An example of this use can be taken from a system called LITHO, which interprets data from oil wells. In this system, each well is decomposed into a number of zones that the petrologist can distinguish by depth. Context organization of this system is shown in Figure-7.

The third use is to distinguish events or situations that an object can have. An example of this use can be shown in CAR example, where different repairs in the past represent different situations that a repair process can have.

c. Internal Structure Of Contexts

One of the important properties associated with a context type is the definition of parameter group. A given parameter group defines a list of parameters which belongs to a context type. While every context generally has its own parameter group, one parameter group can be shared by more than one context.

Another property that a context must have is an ASSOCWITH which shows the ancestor context. Also a context typically has MAINPROPS and GOAL properties. The goal property must be defined for the root context. They are explained later in this section. The consultation is started and driven by tracing the parameters defined in the goal or mainprops list.

The example below shows the properties of the context type CAR from the car diagnosis system.

CONTEXT	:	Car
OFFSPRING	:	[subsytem,repairs]
ASSOCWITH	:	nil

```
PARMGROUP    : car_parms
PROMPT3      : 'This is a car diagnosis program'
MAINPROPS    : [year,model,problems]
```

Below is the list of all possible properties of a context type with brief definitions.

<u>offspring</u>

A list of descendant context types. It shows which context types are direct descendants of this context type in the tree.

<u>assocwith</u>

The parent context of this context type in the tree. e.g., CAR context is ASSOCWITH property of the REPAIRS context.

<u>parmgroup</u>

A name which represents group of parameters for this type of context.

<u>prompt1</u>

The prompt asking whether this type of context exists. If the user answer is yes, then an instance of this context type is created and its MAINPROPS parameters will be traced. If there is no PROMPT1 property then it is assumed that there is always at least one instance of the context type and PROMPT3 is displayed.

<u>prompt2</u>

The prompt asking of the user whether additional instances of this context type exists.

<u>prompt3</u>

The prompt that will be displayed when the first instance of this context type is created. This prompt is simply an announcement of the creation of the context instance. Existance of PROMPT3 implies that at least one instance of this context type exists. For example, a PROMPT3 property of the CAR context is: This is a car diagnosis program.

<u>mainprops</u>

List of parameters to be traced once a context of this type has been created. The trace process follows PROMPT3 or PROMPT1 and PROMPT2 if the user's answer to these prompts is 'yes'.

2. <u>Parameters</u>

a. Introduction

Parameters comprise an important class of second level knowledge other than rules; they represent properties of the context or describe facts about the problem space in general. In the structures context, the main use of parameters is to represent the data or evidence. Taking examples from the CAR diagnosis system, parameters are used to describe the status of every subsystem via observations and measurements taken from different parts of the subsystem. BATTERY_VOLT, HYDROMETER, AMMETER and DIMMING_LIGHT are examples of such parameters leading to the description of the status of the subsystem context. A car's status would be completely specified by a context tree if values of all parameters characterizing each node in the tree were known.

Another use of parameters is to represent the goals or advice to be determined. For the CAR problem, the major goal is to determine the defective parts of the car which caused the trouble. One of the goal parameters is STALLED_ENGINE whose value is the information about the defective part of the car causing the engine to be stalled.

Inferences and data are stored using (attribute, object, value) triples. While the object is always some context in the tree, the attribute is a parameter appropriate for that context within the PARMGROUP property of context.

b. Types Of Parameters

Parameters are in three different classes according to the possible values they can take. The simplest are the single-valued parameters. These are the parameters such as model of the car or battery voltage of the electrical system. They can have only one value at a time. Possible values are mutually exclusive for these parameters.

Multivalued parameters can have more than one value at a time. Possible values are not necessarily mutually exclusive. For example the stalled_engine parameter may have more than one value which implies that multiple defects in different parts of the car may cause the engine to stall.

Third parameter type is the yes_no parameter which is a special kind of single valued parameter. It has only two possible values, namely 'yes' and 'no'.

c. Internal Structure Of Parameters

Parameters are categorized according to the context to which they apply. While the PARMGROUP (parameter group) property of a context type defines list of parameters which can be applied to this context, the MEMBEROF property of the parameter defines which one of the above parameter groups the parameter belongs to. Following is a list of properties which define a parameter's internal structure.

memberof

The name of the corresponding category of parameters; any parameter group name.

<u>valutype</u>

The type of parameter (singlevalued, multivalued or yes_no).

<u>expect</u>

Permissible values of a parameter whose value can be asked of the user. Yes_no indicates that a 'yes' or 'no' answer is expected. Number indicates that the expected value is a number. Any indicates that any value can be the answer.

<u>prompt</u>

The question to be asked when the system needs to know the value of any askable parameter.

<u>can ask</u>

Whether the parameter's value can be asked or not.

An example of a parameter and its properties follows:

```
parameter(hydrometer).
hydrometer(memberof,elec_parms).
hydrometer(valutype,singlevalued).
hydrometer(expect,any).
hydrometer(prompt,hydrometer_prompt).
hydrometer(can_ask,1).
```

```
   hydrometer_prompt
        :-
         print('What is the specific gravity measured by
hydrometer ?').
```

The properties of a parameter and context are defined as PROLOG facts. The prompt property of a parameter calls a PROLOG routine which simply prints out the question to be asked of the user. An associated property "is_t" (is traced) of a parameter for a particular context instance is defined dynamically, showing that parameter's value was traced (i.e., an attempt was made to infer its value).

3. Rules

a. Introduction

The largest component of the knowledge base of The EMYCIN-PROLOG consultant is the rule base. The rule base is a collection of production rules which instruct the system how to reason and arrive at conclusions [9].

While the contexts and parameters record the structural information about the domain, the rules describe the action or problem solving component of the expert's knowledge. The content of the rules and their ordering in the database determine the search path taken to conclude a value for goal parameter. The search is depth-first because PROLOG's inherent backtracking mechanism was used. Thus ordering of rules has an important effect on the consultation path. In the EMYCIN-PROLOG consultation system rules which conclude a value with higher certainty were put before the rules with lesser certainty. The heuristic used by EMYCIN named "unity-path" consists of ordering the rules with certainty (CF = 1 or -1) first and executing in that order. Thus if any rule with CFrule = 1 or -1 succeeds, any other rule will not be tried and the search path will be shortened.

Rule execution indirectly causes the context instance to be created, thus providing the mechanism for propagation of the context tree. Creation of a new context occurs when a rule that tried to evaluate a value for a parameter and context tree proves to have no context to which this parameter is applicable. A context is applicable to a parameter if the parameter is a member of the parameter group of this context (MEMBEROF = PARMGROUP). In this case an applicable context is found and its new instance is created (see Section II.E).

b. Internal Structure And Definition Of Rules

Rules have two main parts: action and premise. Below is the general form of a rule in PROLOG form (rule template).

```
PARAM(CNTXT,N,VALUE,CFrule)
   :-
    eval_premise(FUNC1,PARAM1,CNTXT,N,[VAL1],CF1),
    eval_premise(FUNCn,PARAMn,CNTXT,N,[VALN],CFn),
    min([CF1,CFn],CF),
    conclude(CNTXT,N,PARAM,VALUE,CF,CFrule).
```

An example rule and its English translation is:

```
RULE  (PROLOG form)
        battery(CNTXT,N,'weak',Ø.5)
     :-
eval_premise(greateq,hydrometer,CNTXT,N,[125Ø],true),
eval_premise(lessp,battery_volt,CNTXT,N,[12],true),
conclude(CNTXT,N,battery,'weak',Ø.5,1.Ø).
```

RULE (English Translation)

IF hydrometer value of electrical_system is greater or equal to [125Ø] with CF > Ø.2.

AND

battery_volt value of electrical_system is less than [12] with CF > Ø.2.

THEN

battery value of electrical_system is weak with certainity value Ø.5.

(Note that the function min or max is not used since functions greateq and lessp do not return a certainty value).

The PROLOG routines with the eval_premise predicate construct the PREMISE of a rule. After all individual eval_premise routines are executed succesfully, a certainty calculation is made via either the 'max' or 'min' routine, which calculate the maximum or minimum of all certainty numbers that every individual eval_premise routine returns.

The structure of the clause eval_premise is "eval_premise(FUNC, PAR, CNTXT, N, [VAL], CF)" where FUNC is one of the functions defined in Section II.D. PAR is the parameter (attribute) to be evaluated, CNTXT,N is tree pointer showing current context instance at any particular time of the consultation. Its value is left as a variable. The particular context instance to be applied is determined during the consultation by referring to the existing dynamic tree. The determination of the context instance is explained in detail in following section. The [VAL] is one or more parameter values which bind to a given parameter's value. This value or values in the list are of interest. If the parameter has a value in this list with CF value limits defined by the FUNC used, then the eval_premise clause succeeds.

The ACTION part of a rule is simply the last routine in the body of a PROLOG rule. While there may be other action predicates, the only predicate used in EMYCIN-PROLOG is the conclude routine which inserts the (attribute, object, value) triple into the dynamic database with a certainty value. Note that insertion implies updating any other database triple if it is already in the database with a different certainty value. This updating process is explained in Section III.A.

c. Creation Of Context Instances And Rule Evaluation

Creation of new contexts during the consultation process builds the dynamic tree. PREMISE clauses in a rule do not refer to a specific instance of a context, rather context type and instance are determined indirectly depending upon the current dynamic tree.

A consultation begins with the automatic creation of a root context and tracing its

A consultation begins with the automatic creation of a root context and tracing its MAINPROPS parameters with respect to this instance of the root context. The evaluation process executes rules unless the parameter to be evaluated is askable (can_ask = 1). In the ACTION and PREMISE parts of a rule variable pair CNTXT,N is used which is bound to the appropriate value during execution. First the current context is tried; if current context is not applicable then the required context is found on the <u>current branch</u> of the dynamic tree (i.e., the path from the root node to the current context to which the parameter in question can be applied). If no context is found on the current branch, then the applicable context should be a descendant of the current context. All such contexts are found and instantiated and the current rule is applied to each of these contexts.

When each context instance is created its MAINPROPS parameters are traced. After all such contexts have been instantiated and their MAINPROPS parameters traced, the original parameter that triggered this mechanism is traced with respect to all of the newly created instances.

C. INFERENCE MECHANISM

Inference is done in a goal-oriented fashion. The system goal is defined in the MAINPROPS property of the root context. While the system tries to achieve that goal, subgoals are set up and tried in turn. This process is recursive and continues until one of the subgoals is achieved and in turn the top level goal is achieved.

For example, in the CAR diagnosis program one of the top level goals is "stalled_engine". The system calls the rule:

```
        stalled_engine(CNTXT,N,VALUE,1)
          :-
    eval_premise(same,electrical,CNTXT,N,VAL,CF),
    hypothesis(electrical,Cx,Nx,VALUE,CFc),
    conclude(CNTXT,N,stalled_engine,VALUE,1,CFc).
```

The premise of this rule is the subgoal to be pursued which in turn causes other subgoals to be tried until, finally, one of the subgoals succeeds without need to try another subgoal.

At each subgoal-pursuing process in the above reasoning chain, EMYCIN-PROLOG proceeds in two stages. It first attempts to update the dynamic database with the obtained value of the parameter for the related condition.

During the update stage, there are two cases to consider.

In the first case the parameter value can be known by the user (can_ask = 1). In this case the user is directly asked for the value of the parameter. An example from the car diagnosis problem of such a question is: "What is the specific gravity measured by the hydrometer?" EMYCIN-PROLOG uses the prompt property of a parameter to produce this question. The user's response is checked by referring to the expect property of the parameter. If the answer is not an expected value then the user is warned and the same question is repeated until a value in the limits of expected value is obtained. If the answer is "unk" (unknown) then rule base is consulted to evaluate the parameter's value for the context of a particular instance.

In the second case the parameter value cannot be asked of the user, but there are rules which mention the parameter in their action parts. In this case EMYCIN-PROLOG invokes all these rules in order to infer a value for the parameter.

In the second stage (hypothesis retrieving) the dynamic database is consulted for the list of

In the second stage (hypothesis retrieving) the dynamic database is consulted for the list of hypotheses regarding the value of the parameter. The function in the subgoal is applied to this list in an attempt to satisfy the condition, and in turn to achieve the subgoal.

After all rules mentioning the parameter in their action part have been tried, the parameter for the given context is marked as 'traced' i.e., the "is_t" property of the parameter for the given context is set to "1". Further requests for this parameter's value for the given context are met directly from the dynamic database. This process prevents redundant invocation of the rules.

D. FUNCTIONS

There are different types of premise functions that can appear in rules. During the consultation process, the system wants to know for a given parameter one or more of the following:

Whether or not its value is known;

Whether or not its value satisfies the specific value(s) with a specific certainty value limit;

Whether or not its value is known to be true with a certainty value; or

Whether or not its value satisfies a numerical value with CF > Ø.2.

The function names used for each category above are:

(1) KNOWN,NOTKNOWN,DEFINITE,NOTDEFINITE;
(2) SAME,THOUGHNOT;
(3) NOTSAME, MIGHTBE, VNOTKNOWN, DEFIS, NOTDEFIS, DEFNOT, NOTDEFNOT;
(4) GREATERP,LESSP,GREATEQ,LESSEQ.

Functions in the first three groups have certainty factor limits which change according to whether they are applied to a multivalued, singlevalued or yes_no parameter. The first three groups of functions are called nonnumeric predicate functions and the last groups of functions are numeric predicate functions. Another group consists of conclusion functions. Only the conclusion function conclude is used in EMYCIN-PROLOG.

Functions are applied to data triples stored in the dynamic database. All return a truth value except SAME and THOUGHNOT. Functions in the first group are concerned not with the actual value of a parameter but with whether or not it is known. For example, known(condition, electrical_system, 1, true) succeeds if and only if the condition of the electrical system is known with a certainty factor greater than Ø.2.

A list of all functions with their formal definitions are given in Appendix B.

E. CONSULTATION CYCLE

1. Detailed Analysis Of The Control Structure

A consultation starts with the creation of the root node in the context tree, a context of type CAR in our example. Creation of any context involves two basic processes to be done at the outset.

First the root node is added to the context tree.

Second the parameters in its MAINPROPS list are traced.

The MAINPROPS property for context type CAR is the list [year,model,problems]. EMYCIN-PROLOG traces the value of each of these three parameters in turn. Once all of these three parameters have been traced the consultation terminates since finding a value for PROBLEMS parameter is the final goal of the CAR diagnosis system.

While YEAR and MODEL are askable parameters, PROBLEMS is not an askable parameter. Therefore, following the evaluation of the first two parameters, the system proceeds to infer the third parameter's value by consulting the rule base. The rule that mentions this parameter presents a menu to the user asking the kind of the problem occurring in the car. Representing this menu and asking for information are not considered part of the goal-oriented reasoning which the system follows. However, this initial information serves to focus the search and eliminate unnecessary search paths. The user's answers cause another parameter value to be sought. This value is stalled_engine in our example consultation. Again stalled_engine is not an askable parameter. Thus the rule base is consulted and the rules mentioning this parameter are tried in order. Our current context and its instance is CAR,1 (the value of the tree pointer).

Rules mentioning the stalled_engine parameter are:

```
          stalled_engine(CNTXT,N,VALUE,1.Ø)
          :-
eval_premise(same,electrical,CNTXT,N,VAL,CF),
hypothesis(electrical,C,Nx,VALUE,CFc),
conclude(CNTXT,N,stalled_engine,VALUE,1.Ø,CFc).

          stalled_engine(CNTXT,N,VALUE,1.Ø)
        :-
eval_premise(same,fuel,CNTXT,N,VAL,CF),
hypothesis(fuel,C,Nx,VALUE,CFc),
conclude(CNTXT,N,stalled_engine,VALUE,1.Ø,CFc).
```

The premise of the first rule refers to the parameter electrical which is the parameter of the electrical_system context. Since this parameter is not applicable to the current context type car, the applicable context electrical_system has to be found in the tree. It is not in the tree so it will be created. Here the main consultation has to stop temporarily to create this new context.

The electrical_system context is a direct descendant of the car context. The system makes use of the PROMPT1, PROMPT2 and PROMPT3 properties of that context type during the creation process. If there is a PROMPT1 property, the context may not have any instance at all. If there is a PROMPT3 property then there must be at least one instance of the context. The electrical_system context has a PROMPT3 property; hence it is printed out and context instance is created.

When the second context is to be created, the PROMPT2 property is printed out and the user is asked whether another instance of this context type is to be created. The creation process continues until the user replies "no". Then this context is marked as nonaskable by inserting into the database the fact showing that the askable property of this context is zero.

The next step is to trace the parameter(s) in the MAINPROPS list of the context. Since it is an empty list for electrical_system (there is no parameter to be traced immediately), control of the consultation goes back to the evaluation of the parameter electrical. The tree pointer's value is now ELECTRICAL_SYSTEM,1.

The following sequence of events describes the rest of the consultation process.

Electrical is not an askable parameter so the rule base is consulted. The first rule has two parameters in its premise, DIMMING_LIGHT and BATTERY.

DIMMING_LIGHT and BATTERY are applicable to the current context and also DIMMING_LIGHT is an askable parameter (can_ask = 1). The prompt property of this parameter is invoked and the question: 'Turn on your lights and operate the starter; Do the lights go out or become dim ? (yes/no)' is asked. If the answer is "yes" then the condition is satisfied since the specified value [yes] is defined in the [VAL] part of the eval_premise clause (first premise clause). The returned certainty value is "1" unless a number is specifically given with the answered value (e.g., 'yes_9' means that answer is yes with certainty value Ø.9).

The second condition is the evaluation of the BATTERY parameter. It is not askable so again the rule base is consulted and the first related rule has two parameters to be evaluated in order to succeed: HYDROMETER and BATTERY_VOLT.

The eval_premise conditions which mention these parameters have the numerical predicate function LESSP. It is evaluated the same way as DIMMING_LIGHT. The question is asked and the answer is compared with the specified value, which is [1250] for HYDROMETER and [12] for BATTERY_VOLT. If in our case the answers are less than these two values, the conditions succeed. The next condition is the conclusion function which inserts the hypothesis about the BATTERY parameter into the database:

hypothesis(battery,electrical_system,1,weak, 1)

Following this first BATTERY rule all other rules about BATTERY are also tried and, if applicable, other hypotheses about this parameter are inserted into database and the is_t property of this parameter is set to "1" for this current context, indicating that parameter's value was traced. If its value is needed in any subsequent rule, the value is retrieved from database directly. At this point control goes back to the first electrical rule. Since the first two conditions have succeeded, the next premise clause returns a minimum of concluded certainty values as a certainty value for the premise of the rule. The next clause before the rule succeeds is the conclusion function; another hypothesis now is entered into database:

hypothesis(electrical,electrical_system,1,battery,Ø.8)
assuming CF1 and CF2 are both 1, min([CF1,CF2],CF) returns CF = 1, and the concluded hypothesis' CF value is the multiplication of the rule's certainty value (here Ø.8) and the premise's certainty value 1.

Following this first electrical rule all other rules about electrical also are tried. At the end the is_t flag is set to "1" and control is sent back to the first stalled_engine rule. The next clause retrieves the concluded value of electrical by calling the clause hypothesis(electrical,C,N,VAL,CF), whose variables in the argument list binds the previously concluded value. The last condition is the conclusion function, which concludes a value (i.e., inserts a hypothesis into the database). The inserted hypothesis is:

hypothesis(stalled_engine,car,1,battery,Ø.8).

Following this rule's execution, other rules about stalled_engine are also tried, and the is_t property is set to "1" again. An exhaustive execution of all stalled_engine rules concludes the consultation. Before the consultation ends all concluded values of stalled_engine parameter are printed out. The rest of the concluded hypotheses obtained during the consultation are also printed out for debugging purposes.

2. Departures From The Main Control Structure

At any particular time, evaluation of a parameter is exploited via rules in the evaluation of goal parameter values. So the backwards chaining mechanism is not strictly followed throughout the consultation.

Evaluation of any parameter's value may require creation of a context instance as explained in a previous section. Each time a context is created its MAINPROPS parameters are traced whether they are needed or not. Following this, the trace process brings control back to the point from which it departed. A typical example of this departure is seen when an attempt is made to evaluate the THROTTLE_TEST parameter of a fuel_system context following the creation of this context. The MAINPROP parameter will be traced whether this parameter is needed at once or not.

III. INEXACT REASONING

Since the knowledge base of an expert system is basically a collection of facts and rules obtained from the user and domain expert and since most of the data/knowledge obtained are imprecise in nature, it is common that both the fact and inference rules are not completely certain.

Uncertainty is introduced into the EMYCIN-PROLOG expert system in two ways. First, factual knowledge provided by the user represents observable evidence or symptoms. This evidence might be difficult to observe or might have to be measured with inaccurate or unreliable equipment. A number as a measurement of this type of uncertainty can be associated with the observed value.

The second type of uncertainty exists in the inference rules. The inference rules are intended to capture the expert's experience, heuristics, judgement, and intuition, which is inherently vague and nondeterministic. While the rules are being written, the expert's reluctance to give a strong relationship between the premise and conclusion of a rule would force the rule author to introduce a number accounting for such uncertainty (CFrule).

Since the decisions are made by human experts without perfect information - this is what makes experts experts - our concern in this section is to explain the calculus used in EMYCIN-PROLOG to combine different kinds of uncertainty into a final uncertainty measure associated with the final conclusion.

A. CERTAINTY FACTORS

Factual information is stored in the database as (object, attribute, value) triples as mentioned before. A number in the range of -1 to 1 is associated with these triples assigning a measure of belief or disbelief to the statement:

The <attribute> of <object> is <value>

where object (CNTXT,N) is a context instance as previously defined. An object may have several attributes (PAR). For example, electrical_system-1 in the context tree in Figure-2 has attributes of HYDROMETER and BATTERY_VOLT (See knowledge-base of CAR diagnosis system in Appendix C.1.). Each attribute is called a parameter. The third field is simply the value of that attribute of the object.

A hypothesis is a (object, attribute, value) triple and a certainty value associated with it. The object is represented as a tuple: context name and instance number (e.g., electrical_system,1). For example, hypothesis (battery, electrical_system, N, weak, Ø.8) denotes that the condition of the battery is believed to be weak with the belief value Ø.8.

Whenever a hypothesis is constructed, either with the help of the rule or with information from the user, the associated certainty value is also calculated. If a rule with rule certainty value "CFrule" is used, then the calculation proceeds as follows.

The certainty of premise is calculated by taking the minimum or maximum of the certainty values (CF) of the premise. For example, in the rule from CAR diagnose system:

```
    electrical (CNTXT,N, 'starter_circuit', Ø.6)
            :-
eval_premise (same,dimming_light,CNTXT,N,[no],CF1),
eval_premise (same,fuel_sys,CNTXT,N,[ok],CF2),
min([CF1,CF2],CF),
conclude(CNTXT,N,electrical,'starter_circuit',Ø.6,CF).
```

There are two CF values, namely, CF1 and CF2. The certainty of the premise calculated by taking the minimum of these two values since the premises are ANDed. We would be taking the maximum of those values if they had been ORed. Following this calculation, CFnew=CF*CFrule is formed (CFrule is Ø.6 in above example rule). The final result taken from the multiplication process becomes the certainty value for the concluded hypothesis. If there is another hypothesis in the database with the same triple, then its certainty (CFold) is combined with the new certainty value (CFnew).

Combining uncertainty values into a final value proceeds by updating existing hypotheses until all applicable rules have been executed. The following small sample sessions show the use of combining functions and obtaining a final conclusion based on the criteria of certainty values. In the EMYCIN-PROLOG, we preferred to list all concluded values of the goal parameters so that user will have a chance to see all possible conclusions with their certainty values.

Assume the goal parameter of the consultation is battery and the database has the following hypotheses:

hyp #1

```
hypothesis(battery,electrical_system,1, bad_connectio-
ns, Ø.5).
```

hyp #2

```
hypothesis(hydrometer,electrical_system,1,12ØØ,1.Ø).
```

hyp #3

```
hypothesis(battery_volt,electrical_system,1,10,1.0).
```

hyp #4

```
hypothesis(battery,electrical_system,1,weak,0.7).
```

The following rule concludes a hypothesis for the attribute battery:

```
     battery(CNTXT,N,'weak',0.8)
        :-
 eval_premise(lessp,hydrometer,CNTXT,N,[1250],true),
 eval_premise(lessp,battery_volt,CNTXT,N,[12],true),
 conclude(CNTXT,N,'weak',1.0,0.8).
```

If the first two clauses in the premise succeed then the following hypothesis is concluded:

hyp #5

```
hypothesis(battery,electrical_system,1,weak,0.8).
```

[Note that the tree pointer points to (electrical_system,1.)]. Now hypothesis #4 and #5 should be combined into a new hypothesis since they conclude for the same (attribute,object,value) triple.

Using the first combination function (see next section for the explanation of combination functions) leads to the calculation:

```
CFcomb = CFold + CFnew * (1 - CFold)
       = Ø.7   + Ø.8   * (1 - Ø.7)
CFcomb = Ø.94
```

Following the update process hyp #4 becomes:

hypothesis(battery,electrical_system,1,weak, Ø.94).

Comparing the final certainty values and taking the maximum one, the final conclusion of the consultation is:

concluded(battery,electrical_system,1,weak,Ø.94).

which translates as:

The concluded (value) of battery (attribute) of electrical_system,1 (object) is "weak" with certainty value Ø.94.

As we mentioned earlier in our implementation a list of all hypotheses which conclude a value of the goal parameter are presented.

B. COMBINING FUNCTIONS

There are two possible cases during the combination process which are determined by the sign of the old and new certainty values (CFold,CFnew). These different cases and corresponding combining functions are:

```
(1)   CFold > Ø and CFnew > Ø
      CFcomb = CFold + CFnew * (1- CFold )
(2)   (CFold * CFnew) < Ø
      CFcomb = (CFold + CFnew )/(1- min(CFold,CFnew))
(3)   CFold < Ø and CFnew < Ø
      CFcomb = -(-CFold - CFnew * (1 + CFold))
```

The update process or combining certainty values is used when the same value for the same object of an attribute (PAR) is evaluated with different certainty values.

IV. KNOWLEDGE ACQUISITION

The knowledge engineer's main task is to enter and debug the production rules and the facts about static knowledge other than rules. Acquisition of this static knowledge requires two levels of control, namely catching common syntax input errors such as misspellings and catching inconsistencies which are likely to occur between rules. In the EMYCIN-PROLOG consultation system, rules are typed from the terminal by the knowledge engineer and no automatic consistency or error checking are performed. In this section possible mechanisms for such controls are discussed.

Rules are in PROLOG rule format as explained in Section II.A.3.

Acceptance of a rule into the rule base requires the following consistency checks:

All parameters used in the rule should be defined.

The sum of certainty values of rules whose PREMISEs can be true at the same time but conclude different values should not exceed 1. For example the following two rules cannot exist in the same rule base:

```
    battery(CNTXT,N,'bad_connections',Ø.7)
      :-
 eval_premise(greateq,battery_volt,CNTXT,N,[12],true),
 conclude(CNTXT,N,battery,'bad_connections',Ø.7,1).

    battery(CNTXT,N,'weak',Ø.5)
      :-
 eval_premise(greateq,battery_volt,CNTXT,N,[12],true),
 conclude(CNTXT,N,battery,'weak',Ø.5,1).
```

since Ø.5 + Ø.7 = 1.2 and 1.2 > 1.

At least one rule should exist in the rule base for every non-askable (can_ask = Ø) parameter.

V. EXPLANATION SYSTEM

A. INTRODUCTION

One of the main design considerations in building an expert system is the ability to explain its advice (i.e., provide the user with enough information about its reasoning so that the user can decide whether to follow the recommendation).

In this section we will introduce the requirements for a complete explanation module. One of the main issues involves answering the question 'WHY' asked by the user when the system requests data to continue the consultation. (It is numbered as WHY1 to distinguish it from other WHY questions.) In this case the WHY1 question can be interpreted as: "How is the request for this data related to a goal?" Other WHY questions can be defined; one of them would be: Why did you request this data to reach this goal? - WHY2- (i.e., give the strategy behind the inferencing process).

Besides these two main WHY questions, some other questions about the system's reasoning process include:

How does one goal lead to another?

How is a goal achieved?

Why is one hypothesis considered before another?

Why is one question asked before another?

In the current explanation scheme of EMYCIN, the question WHY is handled by giving to the user the rule which evaluates the parameter under consideration. Successive WHY questions invoke antecedent rules [8].

This explanation scheme uses just the rules. Since rules do not have all the necessary knowledge elements, as discussed below, this scheme has some deficiencies.

First the ordering of hypotheses in a rule's premise will affect the order in which goals are pursued. There is no explicit knowledge showing reasons for this ordering.

Second the ordering of rules affects the order in which hypotheses and hence subgoals are pursued. There is no explicit knowledge about why a particular ordering is preferred (i.e., why is one hypothesis considered before another).

Third the inference steps taken by the author which connect the premise of a rule to the action part are omitted. The intermediate reasoning steps provide justification for a particular rule used. The argument could arise as to whether there is intermediate reasoning connecting a premise to an action. Our empirical study in some existing rule-based systems showed us that most of the rules used in those systems have the knowledge in compiled form (i.e., parts of the expert's reasoning is left out of a rule). Specifically in our CAR diagnosis system design we did not need intermediate reasoning steps to be defined explicitly for a working consultation program with respect to the running of the consultation session.

The above three types of knowledge, implicit in rule design, should be defined explicitly to satisfy one of the main design considerations of an expert system, namely the explanation of its reasoning. Our special interest has been focused on the type of the knowledge explained in the third item above.

An example rule from the CAR diagnose system is:

```
     electrical(CNTXT,N,'low_starter_resistance',Ø.7)
              :-
 eval_premise(same,ammeter,CNTXT,N,[yes],CF1),
 eval_premise(same,starter_motor,CNTXT,N,[ok],CF2),
 min([CF1,CF2],CF),
 conclude(CNTXT,N,electrical,'low_starter_resistance',
         Ø.7,CF).
```

The above rule concludes a value about the parameter electrical. Two premise clauses require values of the parameters ammeter and starter_motor in order. In either of these premise clauses, any information which connects them to the parameter electrical is not known. The answer to the question: "WHY do we need to know about ammeter and starter_motor to be able to obtain a value for the electrical parameter?" is implicit in the rule. We need additional knowledge (support knowledge) to answer the above question, which corresponds to the answer of WHY1. This question is asked of the system by the user when the system requests a value of ammeter or starter_motor. One of the possible explanations for such a WHY question is as follows: The reason for looking for a value of ammeter is that ammeter measures electric current and electric current is produced by the electrical system, so any change of the ammeter value gives a clue about the condition of the electrical system. Similarly, an explanation for the search a value for the starter_motor parameter would be that the starter motor requires battery voltage to operate. If the starter motor resistance is short circuited, the battery voltage is used up and very little voltage is left to crank the engine. The battery voltage measures the battery performance and battery performance is the quality of the battery. Since the battery is part of the electrical system, any problem in the starting system is likely to have an affect on the condition of the electrical system. So we need to know about the starter_motor parameter.

The above two paragraphs provide the <u>support knowledge</u> required to bring a sound explanation to the user's queries about system's reasoning. The rest of this chapter illustrates ways of structuring and representing this support knowledge and giving explanations by using this knowledge when it is requested.

Once the representation scheme for the support knowledge is defined, this knowledge is acquired from the expert, we then proceed with structuring and representing this knowledge. In the following three sections these issues will be presented using CAR diagnosis system.

B. ACQUIRING AND STRUCTURING THE SUPPORT KNOWLEDGE

In the explanation phase, we focus on a particular WHY question, namely: How is a request for this data related to a goal? This question focuses on a rule. Each individual premise and action part of a rule requires the support knowledge. After all this knowledge is obtained, it is first converted into an explanation tree from the expert's natural language form and then into a semantic network, and finally all of such semantic networks for individual rules are combined into one semantic explanation network which corresponds to the support knowledge for the whole rule base. Support knowledge is represented in a semantic network. To explain the above process we use the CAR diagnosis rule-base. Some of the system's requests for data from the user can be seen in Appendix D. The explanation system is invoked if the user answers WHY to any of these questions.

C. NATURAL EXPLANATION AND EXPLANATION TREE

The explanation process involves three main activities: giving examples, eliminating alternatives and giving reasons [15]. The expert tries to reach commonly known concepts using the above three building blocks of natural explanation. Given an explanation from the expert, our first goal is to structure and represent this discoursive form of explanation in the tree form.

The explanation tree is composed of nodes and statements connected to them. A statement may be another node, thereby providing an embedded structure. Nodes are nonterminals of the grammar and statements are terminals [15]. In our case study only one main building block, "giving a reason" is used. The corresponding grammar is :

```
start ==> 1CLUE/RSN  e  e
```

```
e     ==> STMT/RSN    e   e(e^n)
e     ==>  RSN/STMT    e(e^n)  e
e     ==>  AND - OR    e   e(e^n)
e     ==>  IF/THEN     e   e
e     ==>  statement
```

where n>=1.

1CLUE/RSN, STMT/RSN, RSN/STMT, AND, OR and IF/THEN are possible statement connectors. Their brief descriptions are:

1CLUE/RSN e1 e2 : one of the clues to get a value of e2 is e1.

STMT/RSN e1 e2	: the reason for e2 is e1.
RSN/STMT e1 e2	: the reason for e1 is e2.
AND e1 e2	: e1 and e2.
OR e1 e2	: e1 or e2.
IF/THEN e1 e2	: if e1 then e2.

The explanation tree provides a more powerful method of acquiring an explanation from the expert. Once the expert's explanation is structured into the tree, it is easier to proceed since the explanation is divided into smaller parts and each part corresponds to one element of the grammar. For this reason construction of a individual explanation tree (acquisition of explanation knowledge) becomes the crucial step. An example of an explanation tree is given in Figure-8.

D. SEMANTIC EXPLANATION NETWORK

After the explanation tree is formed, the next step is to construct a corresponding semantic network. First the terminal nodes in the tree are structured into their semantic network equivalents. Each node in the semantic network has a STATUS and PATH link. The path link provides information about relationship between nodes. The status link provides information about possible conditions of the node (parameter). Rules in the inference network connect these conditions to each other (see following section for inference network).

The following example explains the construction of the semantic network, starting from the natural language form of explanation. An explanation (answer) to the question, "How is the data about starter_motor related to the electrical parameter?" would be: "The starter motor works with battery voltage; the battery voltage is the quality of the battery; and the battery is part of the electrical system. If there is any problem in the starter motor, then the electrical system is likely to exhibit of this problem. For example: if a starter motor has a low resistance, then the battery voltage is consumed which in turn causes the battery to be in bad condition."

The explanation tree corresponding to the above explanation is depicted in Figure-8 and the semantic network in Figure-9. The IF/THEN conditions of the tree represented in the status links and other terminal nodes provide the relationship links between nodes (parameters).

E. INFERENCE NETWORK

The inference network is the representation of the semantic network in PROLOG rules and facts. It is composed of three main parts: inference rules, relationship facts and path facts.

Inference rules provide all hypothetical conditions of parameters and their connections to each other. They correspond to the IF/THEN nodes of the explanation tree.

Relationship facts simply represent relationships between parameters. For example the fact starter_motor(works_with,battery_voltage) shows that the relationship between the parameters starter_motor and battery_voltage is one in which the starter motor requires the battery voltage to operate properly.

A path fact is used to facilitate the implementation of our explanation system. It directs the inference process in the inference rules. There may be more than one path between two parameters in the semantic network. Only one path is traced at a time and the inference to be considered next is determined by the path list obtained from path facts.

The inference network is depicted in Figure-11 which corresponds to the semantic network in Figure-9 and also corresponds to the natural explanation given in Section V.D. Two parameters are the key values of the tracing process: Starter_motor (one premise condition parameter of the rule) and electrical (action parameter of the rule). The tracing of the inference network and the providing of an explanation can be summarized in following sequence of events.

The path list(s) are obtained from path facts using key parameters:
path(starter_motor,electrical,[battery_voltage, battery]). Complete path list for this example is:
[starter_motor,battery_voltage,battery,electrical]

Every consecutive parameter in the list should have a corresponding relationship fact in the inference network. After tracing, the following list of facts are obtained:

```
starter_motor(works_with,battery_voltage).
battery_voltage(quality_of,battery).
battery(part_of,electrical).
```

Inference rules mentioning every parameter in the list are extracted. Starting from the last element of the list (electrical in this case):

```
electrical(status,bad) :-  battery(status,bad).
battery(status,bad) :-
   battery_voltage(status,used_up).
battery_voltage(status,used_up) :-
   starter_motor(status,low_resistance)
```

The rule extracting process ends when the first element of the list is encountered (starter_motor here).

Finally obtained facts and rules are put in explanation form as follows:

```
     Starter motor gives a clue about electrical
  SINCE
     Starter motor requires battery voltage  AND
     Battery voltage is quality of battery   AND
     Battery is part of electrical
  AND
     IF starter motor has low resistance
     THEN battery voltage is used up
          AND
     IF battery voltage is used up
     THEN battery condition is bad
```

AND

IF battery condition is bad

THEN electrical system condition is bad.

The first statement mentions the two key parameters. The subsequent list of ANDed sentences are facts obtained from the inference network and connected to the first sentence with "since". The rest of the explanation is ANDed rules, again obtained from the inference network. Figure-10 through Figure-16 shows all elements of the explanation system (explanation trees, semantic networks, inference network) for a CAR diagnosis system. An explanation may not have the second part of above example; in this case only first part of ANDed sentences are given as explanation.

VI. CONCLUSION

A. THE LESSONS LEARNED

First we studied EMYCIN in detail. EMYCIN has some weaknesses and problems. Some of them are explained in Section VI.C. We also discovered that the existing explanation system was insufficient and proposed a new explanation system in Section V. In building the EMYCIN-PROLOG inference engine, and the knowledge-base (CAR diagnosis system), and running the consultation system, we experienced the building process of a complete expert consultation system. We can divide this building process into two main parts. The first part is building the shell (e.g., EMYCIN-PROLOG) and second part is constructing a knowledge-base (e.g., CAR diagnosis knowledge base). During the first part a high-level conceptual structure should be defined. This structure should be independent of the knowledge domain and should be able to work with different domains. In our study the two different domains are the CAR diagnosis system and the FINANCE analysis system. The high-level conceptual structure of EMYCIN is the context tree and the parameter definitions. The construction of the knowledge-base consists of the definition of contexts, parameters,

and rules. We worked by starting from a small model of the domain and expanded the model gradually. After the complete knowledge base was built we ran a consultation and, according to the results obtained, we made changes to the knowledge-base (i.e., adding the new parameters, the new contexts, changing
or adding new rules, etc). This process continued iteratively until satisfactory recommendations were obtained from the consultation system.

Implementing the above two parts showed us the complete cycle of the expert system building process. We experienced the role of the shell designer and the knowledge engineer. Finally we had a clear understanding of the expert system design process.

B. REQUIRED HARDWARE AND SOFTWARE

During our study we translated the EMYCIN inference engine into the PROLOG system (EMYCIN-PROLOG) and succesfully combined two different knowledge-bases with this PROLOG system. The EMYCIN-PROLOG was first implemented on the PROLOG-86 interpreter [16] with 16-bit IBM-PC machine working under MS-DOS or PC-DOS. Later the program was transferred, with minor syntactical changes, onto C-PROLOG on the 32-bit VAX machine under the UNIX operating system In fact, PROLOG-86 allowed us to use variable predicate names which facilitated our implementation. On C-PROLOG we wrote additional routines (see "variable_predicate" routine in the UTILITIES file in Appendix A).

The total space requirement for the EMYCIN-PROLOG codes was 40288 bytes, and the knowledge-base of the CAR diagnosis and the FINANCE analysis systems required 16244 bytes. During the consultation the FINANCE analysis system space requirements in bytes were: atom space (38584), aux stack (612), trail (1200), heap (87728), global stack (8808), and local stack (10240). Runtime was 22.33 sec. The above space and time requirements of the EMYCIN-PROLOG consultation system provide a highly portable system since it is possible to run the system on microcomputers with minor syntactical changes and 288 Kbytes of memory.

C. EVALUATION OF EMYCIN

1. Generality Of EMYCIN

EMYCIN imposes a data structure of (attribute, object, value) triples and these triples must be used in a backward-chaining control structure applied to production rules [9]. Even though EMYCIN can be applied to different domains of diagnosis problems, another domain of design problem may not work properly because of above constraints.

2. Some Particular Problems

The context tree structure imposes the main restriction. Every node in the context tree leads to the root node by a single pathway. In real applications contexts in any domain are not partitioned so artificially. Any improper building of the static tree causes big troubles later in consultations, and it is a very costly process to go back to the start and rearrange the static tree.

Contexts are instantiated only when needed. This brings considerable complexity of implementation. This property helps avoid acquiring information which is not needed for a particular consultation, but there may be domains where a set of contexts will always be needed at the beginning of a consultation, which makes the whole propagation method for the tree obsolete.

Another restriction imposed is the requirement to include the parameter as the goal of consultation in the MAINPROPS list of the root node, since instantiating the root node initiates the reasoning chain for the consultation.

Multivalued parameters cannot be used successfully in the function KNOWN since the function would succeed immediately after any one value were known. On the contrary, multivalued parameters have more than one value, and the function KNOWN does not have a control to check all those other possible values of parameters before success.

MAINPROPS parameters should be either singlevalued or yes_no parameter, since there is no specific value of maltivalued parameter defining whether parameter's evaluation process is done or not, as in the case of known function explained in previous paragraph.

D. PROLOG AND EMYCIN-PROLOG

EMYCIN-PROLOG possesses most of the properties of EMYCIN since the main conceptual and control structure is preserved, as explained in previous section. Contrary to above problems of EMYCIN in EMYCIN-PROLOG, PROLOG's unification pattern-matching made deduction possible without any additional programming. This in turn increased the expressive power in the representation of factual knowledge and its manipulation. For example the hypothesis-retrieving process was easily performed using the unification property.

PROLOG also succesfully facilitated implementation of EMYCIN, especially in the data structures, rules (even though rules are not part of

EMYCIN, they are required for a working consultation system and thus was mentioned here), and hypothesis inference.

Compared to Interlisp (the language in which EMYCIN was first implemented) PROLOG seems to be a better language for implementing EMYCIN. Especially during the rule execution phase, we did not need any aditional programming because of PROLOG's built-in pattern-matching facility (see "try_all_rules_for_PAR(...)" routine in the source codes in the Appendix A).

E. EFFICIENCY OF EMYCIN-PROLOG

The user interaction module of an expert system shell typically covers 30% of the whole programming effort. EMYCIN-PROLOG didn't have a user interaction module, and in a usable product it should be implemented. Suggestions for this module are given in chapter IV. In addition to the user interaction module, the suggested explanation system (see chapter V) also should be implemented. Rather than having usable end product we were mostly concerned about making the inside of a shell visible. The benefits of this work are explained in the following section.

Another alternative approach to the EMYCIN-PROLOG shell would be the decision-lattice shell [18].

A decision-lattice shell is highly domain-dependent and it does not bring the advantages of EMYCIN-PROLOG as explained in the following section (i.e., EMYCIN-PROLOG prevents the same codes from being repeated and shortens the programming work considerably when building several expert systems).

F. THE BENEFITS OF OUR WORK

Once a shell is provided, building a complete expert consultation system is much easier than starting from scratch and programming the whole expert system. During the building of the CAR diagnosis and FINANCE analysis systems the work mostly focused on the mapping of the domain knowledge into the rules rather than programming.

EMYCIN-PROLOG performs better on diagnostic problems than nondiagnostic problems. Different domains can use EMYCIN-PROLOG for building a complete expert consultation system as long as they are diagnostic-type domains. The CAR diagnosis and FINANCE analysis systems were two such domains. Two different consultation systems were built for them using EMYCIN-PROLOG during our work. Without EMYCIN-PROLOG we wouldn't have been able to build them within our time constraints.

Besides the above advantage of using EMYCIN-PROLOG, we have demonstrated the phases of expert system programming. Once the structural requirements of EMYCIN are understood, the different phases of the building process can be seen easily (e.g., defining structural requirements, building a shell, building the knowledge-base, etc).

Another advantage is that a reader can experiment with the code, since a complete list of the program code with the sample consultations is provided.

APPENDIX A
SOURCE CODE

This appendix contains a listing of the main program (held in the files ENGINE, FUNC, and UTILITIES).

EMYCIN-PROLOG is written in the version of the PROLOG language known as C-PROLOG and runs under the UNIX operating system on VAX Machine. This version of PROLOG is closely based on standards as described in Clocksin and Mellish [10].

The knowledge engineer and consultor has no responsibility or relation to the writing of the codes which presented in this appendix.

Having entered the PROLOG, program prints a short message about EMYCIN-PROLOG and then user starts the consultation with the query of "begin".

The lines that limited with "*" are comment lines. They should not be confused with actual PROLOG codes.

% FOLLOWING LIST OF CODES ARE CONTENTS OF ENGINE FILE.

```
/*********************** MAIN PROGRAM ***************/
```

All asserted facts are cleaned from database (cleandatabase), nextnum and pasked properties of contexts are initialized (initialize nextnum pasked), and user is asked of name of the root context. Since root context is askable at start its askable property is set to "1" (initialize askable). Then root context is created and its MAINPROPS parameters are traced (create root and start consultation), once this routine succeeds then consultation ends. Following the consultation results are printed (print result) and also all concluded hypotheses in the dynamic database are printed (print dbase).

```
/*****************************************************/
:-
 nl,nl,nl,
 write(' WELCOME  TO EMYCIN-PROLOG  CONSULTATION
PROGRAM'),nl,
 write(' Please enter "begin" to start the
consultation '),
 nl,nl,nl.
```

```
begin
  :-
   cleandatabase,
   assert(fact(not_first_run)),
   not(initialize_nextnum_pasked),
   nl,nl,nl,
   write('Enter the name of the root context
   (CAR,LEASE) '),    write(' ==>'),read(DOMAIN),
   not(initialize_askable(nnil,1)),
   create_root_and_start_consultation(DOMAIN,N),
   print_result,
   print_dbase,!.

create_root_and_start_consultation(C,N)
  :-
   v_func_2(C,assocwith,Cp),
   v_func_2(Cp,nextnum,N2), Np is N2+1,
   create_and_trace_mainprops(Cp,Np,C,N).

/***********  EVALUATE  PARAMETER VALUE  ************/
```

This routine evaluates the value of a parameter. As explained in section II.B. there are two possible cases ; parameter's value can be known by the user (can_ask = 1) or parameter's value cannot be asked of user, in first case user is directly asked of the value of parameter, in latter case all rules about the parameter are tried (try all rules for PAR). User can answer as "unk" if the data is not available at all. User's answer is checked against the expected value of the parameter and if the value is unexpected one, user is warned and question is repeated. Evaluation of a parameter is done for a all instances of a context. Evaluation ends when all instances of the context is tried (nextnum = Ø).

```
/*****************************************************/
eval2(C,Ø,PAR,VAL,CF) :- !.
eval2(C,N,PAR,VAL,CF)
  :-
   v_func_2(PAR,can_ask,1),
   message_askable(C,N,PAR),
   get_the_answer(VAL,CF),nl,
   v_func_2(PAR,expect,EXPECT),
   check_the_answer(C,N,PAR,CF,VAL,EXPECT),
   VAL \== 'unk',
   assert(hypothesis(PAR,C,N,VAL,CF)),
   assert(is_t(PAR,C,N,1)),
   Nn is N-1,
   eval2(C,Nn,PAR,VALn,CFn).

eval2(C,N,PAR,VAL,CF)
  :-
   v_func_2(PAR,can_ask,1),
   write('Unexpected answer  !!! Please try
   again.'),nl,nl,
   eval2(C,N,PAR,VAL,CF).

message_askable(C,N,PAR)
  :-
   v_func_2(PAR,prompt,PROMPT),
   write(C),write('-'),write(N),nl,
   PROMPT,!.

eval2(C,N,PAR,VAL,CF)
  :-
   not(try_all_rules_for_PAR(PAR,C,N,VAL,CFrule)),
   assert(is_t(PAR,C,N,1)),
   Nn is N-1,
   eval2(C,Nn,PAR,VALn,CFn).

/************  TRY_ALL_RULES_FOR_PAR  ***************/
     All rules  which mentions particular parameter in
their head part are tried. The parameter is  passed in
last eval2  routine. If  the parameter is singlevalued
or yes_no parameter and  there  is  a  hypothesis with
certainity  (CF  =  1)  then  execution  of  rules  is
stopped. (!,fail) combination stops the execution.
/***************************************************/
```

```
try_all_rules_for_PAR(PAR,C,N,VAL,CFrule)
   :-
    (v_func_2(PAR,valutype,singlevalued);
    v_func_2(PAR,valutype,yes_no)),
    hypothesis(PAR,C,N,VAL,1),!,fail.

try_all_rules_for_PAR(PAR,C,N,VAL,CFrule)
  :-
   v_func_4(PAR,C,N,VAL,CFrule),fail.
```

```
/************  FIND APPLICABLE CONTEXT  *************/
```

At any time of the consultation if the current context is not applicable then this routine finds the applicable one. First parent context is checked then descendant contexts and finally brother contexts are tried. If there is not any applicable context in the dynamic tree then it is created (create by traversing). Last argument in "create_by_traversing" routine is used to keep track of the context which traversing has been started. After creation process is done then "descendant_or_brother" routine finds this applicable context.

```
/***************************************************/
```

```
find_applicable_context(C,N,PAR,Cap,Nap)
  :-
   parent_test(C,N,PAR,Cap,Nap).

find_applicable_context(C,N,PAR,Cap,Nap)
  :-
   descendant_test(C,N,PAR,Cap,Nap).

find_applicable_context(C,N,PAR,Cap,Nap)
  :-
   brother_test(C,N,PAR,Cap,Nap).

find_applicable_context(C,N,PAR,Cap,Nap)
  :-
   create_by_traversing(C,N,PAR,C),
   descendant_or_brother(C,N,PAR,Cap,Nap).
/**********************   PARENT_TEST   ***************/
parent_test(C,N,PAR,Cp,Np)
  :-
   v_func_5(C,Cp,Np,C,N,tree),
   cntxt_applicable(Cp,PAR).

parent_test(C,N,PAR,Cp,Np)
  :-
   v_func_5(C,Cp,Np,C,N,tree),
   Cp == nnil,!,fail.
```

```
parent_test(C,N,PAR,Cp,Np)
  :-
   v_func_5(C,Cp,Np,C,N,tree),
   not(cntxt_applicable(Cp,PAR)),
   parent_test(Cp,Np,PAR,Cpp,Npp).

/******************* DESCENDANT_TEST ****************/
descendant_test(C,N,PAR,Cd,Nd)
  :-
   v_func_2(C,offspring,Cd),
   v_func_5(Cd,C,Ng,Cd,Nd,tree),
   cntxt_applicable(Cd,PAR).

descendant_test(C,N,PAR,Cd,Nd)
  :-
   v_func_2(C,offspring,Cd),
   v_func_5(Cd,C,Ng,Cd,Nd,tree),
   not(cntxt_applicable(Cd,PAR)),
   descendant_test(Cd,Nd,PAR,Cdd,Ndd).

descendant_test(C,N,PAR,Cd,Nd)
  :-
   v_func_2(C,offspring,Cd),
   not(v_func_5(Cd,C,N,Cd,Nd,tree)),!,fail.
```

```
/*********************   BROTHER_TEST   *************/
brother_test(C,N,PAR,Cb,Nb)
  :-
   v_func_5(C,Cp,Np,C,N,tree),
   v_func_2(Cp,offspring,Cb),
   Cb \== C,
   cntxt_applicable(Cb,PAR),
   v_func_5(Cb,Cp,Na,Cb,Nb,tree).
/*********************  DESCENDANT OR BROTHER  *****/
 descendant_or_brother(C,N,PAR,Cdb,Ndb)
  :-
   descendant_test(C,N,PAR,Cdb,Ndb);
   brother_test(C,N,PAR,Cdb,Ndb).


/*************  CONTEXT CREATION ROUTINES  **********/
```

In following routines, first applicable context is found then it is created and its MAINPROPS parameters are traced (<u>create and trace</u>). "Cx" in the "create_by_traversing" routine is needed to keep track of the context which traverse began. Traversing will stop when Cx is reached on the way back. If create_applicable_cntxt did not create any context (PROMPT2=NO) at any point then trace_back continues back from the current context (C,N) which doesn't have any other instance i.e., prompt2 for "C" is no.

```
/*****************************************************/
```

```
create_by_traversing(C,N,PAR,Cx)
  :-
   create_applicable_cntxt(C,N,Cc,Nc,PAR),
   trace_back(Cc,Nc,Cx,PAR).

create_applicable_cntxt(C,N,C,N,PAR)
  :-
   fact(context_is_not_created),
   retractall(fact(context_is_not_created)).

create_applicable_cntxt(C,N,Cb,Nb,PAR)
  :-
   v_func_2(C,assocwith,Cp),
   v_func_2(Cp,offspring,Cb),
   Cb \== C,
   cntxt_applicable(Cb,PAR),
   not(v_func_5(Cb,Ca,Na,Cb,Nb,tree)),
   v_func_2(Cp,nextnum,Np),
   create_and_trace_mainprops(Cp,Np,Cb,Nb).

/*****************************************************/
     Go down by creating intermediate contexes until
the applicable context is hit then create all other
occurrences of applicable context with
"create_and_trace_mainprops" routine. If context is
not applicable to PAR, another context "Cc" is tried
```

```
by backtracking to C(offspring,Cc).
/*****************************************************/
create_applicable_cntxt(C,N,Cc,Nc,PAR)
  :-
   v_func_2(C,offspring,Cc),
   not(v_func_5(Cc,C,N,Cc,Nc,tree)),
   cntxt_applicable(Cc,PAR),
   create_and_trace_mainprops(C,N,Cc,Nc).

/****************************************************/
     Context is  not applicable  then create  it as an
intermediate context  and continue recursively.
/****************************************************/
create_applicable_cntxt(C,N,Cc,Nc,PAR)
  :-
   v_func_2(C,offspring,Cs),
   not(v_func_5(Cs,C,N,Cs,Ns,tree)),
   not(cntxt_applicable(Cs,PAR)),
   create_cntxt(C,N,Cs,Ns),
   create_applicable_cntxt(Cs,Ns,Cc,Nc,PAR).

create_applicable_cntxt(C,N,Cc,Nc,PAR)
  :-
   v_func_2(C,offspring,Cs),
   v_func_5(Cs,C,N,Cs,Ns,tree),
   create_applicable_cntxt(Cs,Ns,Cc,Nc,PAR).
```

```
create_and_trace_mainprops(C,N,Cc,Nc)
  :-
   v_func_2(Cc,pasked,Ø),
   create_cntxt(C,N,Cc,Nc),
   create_cntxt2(C,N,Cc,Nc).

/*********************************************************/
     IMPORTANT NOTE  !!! create_and_trace_mainproprops
is called  when applicable context is found. If
applicable context was not  created yet  and if answer
to the  prompt to create context is NO then PAR cannot
be   evaluated  without  creating  the applicable
context.In this  case either  user asked for PAR value
or ERROR message is  sent.  "Cc"  which  is applicable
context  has  its  instance  in  context  tree,  which
Cc(pasked,1). "Create_cntxt2" routine asks for another
instances with PROMPT2.
/*********************************************************/
create_and_trace_mainprops(C,N,Cc,Nc)
  :-
   create_cntxt2(C,N,Cc,Nc).

create_cntxt(C,N,Cc,Nc)
  :-
   v_func_2(Cc,pasked,Ø),
   create_prompt3(C,N,Cc,Nc).
```

```
create_cntxt(C,N,Cc,Nc)
  :-
   v_func_2(Cc,pasked,Ø),
   create_prompt1(C,N,Cc,Nc).

/*****************************************************/
     Answer to  PROMPT1 is "no". "Askable" property is
used in "trace_back" routine.
/****************************************************/
create_cntxt(C,N,Cc,Nc)
  :-
   v_func_2(Cc,pasked,Ø),
   assert(fact(context_is_not_created)),
   update_askable(Cc,C,N,askable,Ø).

create_cntxt(C,N,Cc,Nc)
  :-
   v_func_2(Cc,pasked,1),
   v_func_2(Cc,C,N,askable,1),
   prompt_2(Cc,Nc).

/****************************************************/
     Answer to PROMPT2 is "no".
/****************************************************/
```

```
create_cntxt(C,N,Cc,Nc)
  :-
   assert(fact(context_is_not_created)),
   update_askable(Cc,C,N,askable,Ø).

create_prompt3(C,N,Cc,Nc)
  :-
   v_func_2(Cc,prompt3,PROMPT3),
   write(PROMPT3),
   create_and_trace(C,N,Cc,Nc),
   update_num(Cc,pasked,1).

create_prompt1(C,N,Cc,Nc)
  :-
   v_func_2(Cc,prompt1,PROMPT1),
   write(PROMPT1),write('   ==>'), read(Ans), !,
   affirmative(Ans),
   create_and_trace(C,N,Cc,Nc),
   update_num(Cc,pasked,1).

prompt_2(C,N)
  :-
   v_func_2(C,prompt2,PROMPT2),
   write(PROMPT2),write('    ==>'), read(Ans), !,
   affirmative(Ans),nl,
   v_func_2(C,assocwith,Cp),
   v_func_2(Cp,nextnum,Np),
   create_and_trace(Cp,Np,C,N).

create_prompt2(C,N)
  :-
   v_func_5(C,Cp,Np,C,N,tree),N1 is N+1,
   v_func_2(C,prompt2,PROMPT2),
   write(PROMPT2),write('   ==>'),
read(Ans),!,affirmative(Ans),nl,
   create_and_trace(Cp,Np,C,N1),!,
   create_prompt2(C,N1).

/*****************************************************/
     "Cc,Nc" is  the context to be created.
/*****************************************************/
 create_cntxt2(C,N,Cc,Nc) :- create_prompt2(Cc,Nc).

/*****************************************************/
     Answer to PROMPT2=NO.
/*****************************************************/
create_cntxt2(C,N,Cc,Nc) :-
   update_askable(Cc,C,N,askable,Ø).

create_and_trace(C,N,Cc,Nc)
  :-
   v_func_2(Cc,nextnum,Nn),Nc is Nn + 1,
   update_num(Cc,nextnum,Nc),
   add_5(Cc,C,N,Cc,Nc,tree),nl,nl,
   write('----'),
write(Cc),write('-'),write(Nc),write('----'),nl,nl,
   not(initialize_askable(Cc,Nc)),
   lookmainprops(Cc,Nc).

lookmainprops(C,N)
  :-
   v_func_2(C,mainprops,MAINPROPS),
   eval_par(C,MAINPROPS,N).

eval_par(C,[],N).

eval_par(C,[PAR|REST],N)
  :-
   cntxt_applicable(C,PAR),
   eval2(C,N,PAR,VAL,CF),
   eval_par(C,REST,N).

eval_par(C,[PAR|REST],N)
  :-
   not(cntxt_applicable(C,PAR)),
   find_applicable_context(C,N,PAR,Cap,Nap),
   eval2(Cap,Nap,PAR,VAL,CF),
   eval_par(C,REST,N).
```

```
/*****************  TRACE_BACK  *******************/
```

Once the applicable context is found then all intermediate contexts between this context and the context which traversing started ("Cx") are tried whether any of them has any other descendant context to be created.

"C" and "Cx" are brother contexts.There is no need for trace back.

```
/**************************************************/
trace_back(C,N,Cx,PAR)
  :-
   v_func_2(C,assocwith,Cp),
   v_func_2(Cp,offspring,Cx).


/**************************************************/
```

"Cp" is parent context of "C" and "Cpp" of "Cp". "Cpp" is needed to find askable property of "Cp". If "create_cntxt routine did not creat context (PROMPT2=NO), then "create_applicable_cntxt" returns (Ck,Nk=Cp,Nc) and trace_back continues back from Cp,Nc.

```
/**************************************************/
trace_back(C,N,Cx,PAR)
  :-
   v_func_5(C,Cp,Np,C,N,tree),
   Cp \== Cx,
   v_func_5(Cp,Cpp,Npp,Cp,Np,tree),
   v_func_4(Cp,Cpp,Npp,askable,1),
   create_cntxt(Cpp,Npp,Cp,Nc),
   create_applicable_cntxt(Cp,Nc,Ck,Nk,PAR),
   trace_back(Ck,Nk,Cx,PAR).

trace_back(C,N,Cx,PAR)
  :-
   v_func_5(C,Cp,Np,C,N,tree),
   Cp \== Cx,
   v_func_5(Cp,Cpp,Npp,Cp,Np,tree),
   v_func_4(Cp,Cpp,Npp,askable,Ø),
   trace_back(Cp,Np,Cx,PAR).

trace_back(C,N,Cx,PAR)
  :-
   v_func_2(C,assocwith,Cp),
   Cp == Cx.

/*******  COMBINE CERTAINTY AND CONCLUDE  ***********/
```

A hypothesis is asserted into dynamic database. During the assertion process database is checked if there is any aother hypothesis which concludes the same value for "PAR", if there is then two hypothesis's certainty values are conbined using "combine_func" routine and new hypothesis with new CF

```
value is asserted into database.
/*****************************************************/
conclude(C,N,PAR,VALUE,CFrule,CFmm)
  :-
   CF is  CFrule * CFmm,
   certainity_combine(C,N,PAR,VALUE,CF),!.

certainity_combine(C,N,PAR,VALUE,CFnew)
  :-
   hypothesis(PAR,C,N,VALUE,CFold),
   combine_func(CFold,CFnew,CF),
   retract(hypothesis(PAR,C,N,VALUE,CFold)),
   assert(hypothesis(PAR,C,N,VALUE,CF)).

/*****************************************************/
     If PAR value  is  concluded  for  the  first time
then there  would not  be any   concluded value in the
database.
/*****************************************************/
certainity_combine(C,N,PAR,VALUE,CFnew)
  :-
   not(hypothesis(PAR,C,N,VALUE,CFold)),
   assert(hypothesis(PAR,C,N,VALUE,CFnew)).
```

```
/*****************************************************/
     There  are  three  functions to combine certainty
values:
     CFcomb = CFold + CFnew * (1 - CFold)
     CFcomb = (CFold + CFnew)/(1 - min(CFold,CFnew))
     CFcomb = -(- CFold - CFnew * (1 + CFold))
 /*****************************************************/
combine_func(CFold,CFnew,CF)
  :-
   CFold > Ø,
   CFnew > Ø,
   CF is CFold + CFnew*(1 - CFold).

combine_func(CFold,CFnew,CF)
  :-
   CFmult is CFold*CFnew,
   CFmult < Ø,
   min([CFold,CFnew],CFmin),
   CF is (CFold + CFnew)/(1 - CFmin).

combine_func(CFold,CFnew,CF)
  :-
   CFold < Ø,
   CFnew < Ø,
   CF is -1*(-CFold - CFnew*(1 + CFold)).
```

/******************   CLEANDATABASE   ****************/

The following facts are asserted into the database during the consultation

<u>hypothesis(PAR,C,N,VAL,CF)</u>

The evaluated value of parameter "PAR".

<u>fact(context is not created)</u>

The Warning flag showing that after a call to the "create_cntxt" routine no context is created.Answer to PROMPT1/PROMPT2 is NO.

<u>fact(not first run)</u>

A flag to cleandatabase routine. If this fact is in the database then database is cleaned.

<u>Cc(C,N,Cc,Nc,tree)</u>

A new context is added into context tree.

<u>Cc(C,N,askable,Num)</u>

An askable property ; context "C,N" has no other context Cc descendant to it.

<u>C(nextnum,N)</u>

The context "C" has "N" instances created so far.

<u>C(pasked,Num)</u>

The number (Num) for the context "C" is "1", if context is created via PROMPT1 or PROMPT3, otherwise "Ø". Before PROMPT2 is asked this flag is checked first.

Above facts are retracted from database before consultation starts by using "cleandatabase" routine.

```
/*****************************************************/
cleandatabase
  :-
   fact(not_first_run),
   abolish(hypothesis,5),
   abolish(concluded_PAR_for_C_N,5),
   abolish(applicable_descendant,5),
   abolish(fact,1),
   abolish(descendant,1),
   abolish(is_t,4),
   not(clean1),
   not(clean2),
   not(clean3),
   not(clean4).

cleandatabase.

clean1
  :-
   context(Cc),
   delete_5(Cc,C,N,Cc,Nc,tree),fail.
```

```
clean2
  :-
   context(C),
   delete_2(C,nextnum,N),fail.

clean3
  :-
   context(Cc),
   delete_4(Cc,C,N,askable,K),fail.

clean4
  :-
   context(C),
   delete_2(C,pasked,N),fail.
/*****************  OUTPUT ROUTINES   ***************/
Goal  parameter  is  found  and  all  hypotheses which
concludes about this parameter are printed.  After the
goal parameter,  all other  hypotheses in the database
are printed.
/****************************************************/
print_result
  :-
   goal(PROBLEM),
   v_func_2(PROBLEM,trans,TRANS),nl,nl,nl,
   write(TRANS),nl,
   not(print_conclusion(PROBLEM)).
```

```
print_conclusion(PROBLEM)
  :-
   hypothesis(PROBLEM,C,N,VALUE,CF),
   write(VALUE),nl,
   write('with the certainity :   '),
   write(CF),nl,nl,fail.

print_dbase
  :-
   nl,nl,nl,
   write(' -------  CONCLUSIONS MADE DURING '),
   write('THE CONSULTATION  -----------'),nl,
   nl,nl,not(write_all_concluded_values).

write_all_concluded_values
  :-
   write('parameter / value / '),
   write('certainity / context instance'),nl,
   write('------------------------------'),
   write('------------------------------'),nl,

   write_all_concluded_values2.

write_all_concluded_values2
  :-
  hypothesis(PAR,C,N,VALUE,CF),
  write(PAR), write('  --- '), write(VALUE),
  write('---'),write(CF),
  write('--- '),write(C),write('--'),write(N),nl,fail.

 /*************  PROCESSING THE USER INPUT  *********/
     User's answer  for any data request by the system
is checked against expected value of the parameter. If
the answer  is unexpected  then user is warned and the
question is repeated.
/****************************************************/
 get_the_answer(VAL,CF)
  :-
   read(STRING),
   name(STRING,LIST),
   parse(LIST,VALUE,CERTAINITY,LIST),
   name(VAL,VALUE),
   name(CF,CERTAINITY).

/****************************************************/
     95 is ascii code for underscore "_"
/****************************************************/
parse([X|REST],VALUE,CERTAINITY,LIST)
  :-
   X \== 95,
   parse(REST,VALUE,CERTAINITY,LIST).
parse([X|REST],VALUE,REST,LIST)
  :-
   X == 95,
   seperate_val(LIST,VALUE).

/*****************************************************/
     49 is ascii code for "1" which corresponds to the
   default value  for CF. Default value 1 is used when
user did not specified any certainty of his/her answer
explicitly.
/*****************************************************/
parse([],LIST,[49],LIST).
seperate_val([X|L1],[])
  :-
   X == 95.

seperate_val([X|L1],[X|L3])
  :-
   X \== 95,
   seperate_val(L1,L3).
```

% <u>FOLLOWING LIST OF CODES ARE CONTENTS OF FUNC FILE</u>

```
/************ PREMISE EVALUATION ROUTINES ****************/
  First all  asserted facts during the execution of previous
"eval_premise" routine are retracted.  Evaluation process is
done  in  two  stages  ;  first  database  is  updated i.e.,
parameters value is  evaluated  using  "eval2"  routine then
related hypothesis  is retrieved using "retrieve_hypothesis"
routine.  The  variables  used  in  the  argument  lists  of
routines and their explanations are :
  L=[VAL1,VAL2,....,VALn] list  of values  determined by the
rule writer
       Lcommon=[[VAL1,CF1],[VAL2,CF2],....,[VALn,CFn]] :
  intersection of  evaluated values and values specified
  in the rule. Lcommon = intersection[V,LST]
           V :  set of all hypothesis about PAR.
         LST : the possible values of PAR given  by rule
  author. "L"  usually contains only a single element,if
  L=[] then Lcommon also equal to [].
         When "eval_premise"  fails, then  the rule also
  fails and control goes back to "try_all_rules_for_PAR"
  routine.  Note  that  "PAR,C,N"  is  the key, same PAR
  might  have  different  "concluded_PAR_for_C_N" values
  for different (C,N) pairs.
  /*****************************************************/
  eval_premise(FUNC,PAR,C,N,L,CF)
  :-
 retractall(concluded_PAR_for_C_N(PAR,C,N,VAL,CFm)),!,
 retractall(applicable_descendant(C,N,PAR,Ca,Na)).
 eval_premise2(FUNC,PAR,C,N,L,CF),!.

/*****************************************************/
     IMPORTANT !
THE CUT  (!) OPERATOR  PREVENTS BACKTRACKING AND GIVES
THE CONTROL  TO THE  RULES. IF  EVAL_PREMISE3 FAILS WE
WANT EVAL_PREMISE  TO BE  FAILED AND GIVE CONTROL BACK
TO THE RULES SO THAT SOME OTHER RULE WILL BE TRIED
/*****************************************************/
eval_premise2(FUNC,PAR,C,N,L,CF)
  :-
   cntxt_applicable(C,PAR),!,
   eval_premise3(FUNC,PAR,C,N,L,CF).

/*****************************************************/
     The tree  pointer is  bound to  its correct value
before "eval_premise3"  ROUTINE is called.
/*****************************************************/
eval_premise2(FUNC,PAR,C,N,L,CF)
  :-
   not(cntxt_applicable(C,PAR)),
   find_applicable_context(C,N,PAR,Cap,Nap),!,
   eval_premise3(FUNC,PAR,Cap,Nap,L,CF).


/*****************************************************/
     CFe is different than CF since commonlist chooses
desired  ones  from  all  evaluated values of PAR. PAR
value is evaluated first by "eval2"  routine if either
"is traced"     flag is "0" or PAR is multivalued. "is
traced" flag is ignored  if PAR  is multivalued. "cut"
(!)  operator  is  used to prevent backtracking inside
the  eval2.  If  eval2  could  not  conclude  a  value
("unknown" answer  from user),  then eval2 will return
reasonable value, in this case we  want eval_premise3,
eval_premise2   and   eventually  eval_premise  to  be
failed.
/*****************************************************/
eval_premise3(FUNC,PAR,C,N,L,CForTRUE)
  :-
   (not(is_t(PAR,C,N,1));
   v_func_2(PAR,valutype,multivalued)),
   eval2(C,N,PAR,VAL,CFe),!,
   retrieve_hypothesis(PAR,C,N,L,FUNC,CForTRUE).

eval_premise3(FUNC,PAR,C,N,L,CForTRUE)
  :-
   is_t(PAR,C,N,1),!,
   retrieve_hypothesis(PAR,C,N,L,FUNC,CForTRUE).
retrieve_hypothesis(PAR,C,N,L,FUNC,true)
  :-
   member(FUNC,[greaterp,greateq,lessp,lesseq]),
   v_func_5(FUNC,PAR,C,N,L,true).

retrieve_hypothesis(PAR,C,N,L,FUNC,CF)
  :-
   not(member(FUNC,[greaterp,greateq,lessp,lesseq])),
   commonlist(PAR,C,N,L,Lcommon),!,
   v_func_5(FUNC,PAR,C,N,Lcommon,CF).

/*************   FUNCTIONS IN RULE PREMISE **********/
```

Two main types of functions can be named as "func1" and "func2" where :

<func1> : Does not form conditionals on specific values of a parameter.

<func2> : Controls conditional statements regarding specific values of the parameter in question.

As defined above unlike the <func1> predicates, <func2> predicates control conditional statements regarding specific values of the parameter in the question.These specific values are passed by the argument "L" in eval_premise routine."L" is the list of values to be compared with to evaluate the function "FUNC". Evaluation of premise includes some simple functions.

Functions KNOWN,NOTKNOWN,DEFINITE and NOTDEFINITE are concerned not with the actual value of a parameter,but with whether or not it is known.

Functions SAME,THOUGHNOT both either fail or return a numerical value signifying "true".

Functions NOTSAME, MIGHTBE, VNOTKNOWN, DEFIS, NOTDEFIS, DEFNOT and NOTDEFNOT are all concerned with the certainity factor with which the value of a parameter is known to be true and all return truth values.The empty list "[]" passed by commonlist routine in the first clause "FUNC(PAR,C,N,[],FALSE)" implies that PAR does not have any value which included in the value(s) list, defined by the rule author in the rule premise, then the premise clause which mentions this FUNC fails by returning value "false".

Functions GREATERP,LESSP,GREATEQ and LESSEQ are applied to those parameters which have a numerical value and which return a truth value.These are called numerical functions. Functions $AND and $OR of EMYCIN are changed to MIN and MAX functions. Either of them is added after premises in each rule if the premises are to be ANDed or ORed.

/***/

```
same(PAR,C,N,L,CF)
  :-
   get_most_strongly_confirmed_hyp(L,VAL,CF),
   CF > Ø.2.

notsame(PAR,C,N,[],false).
notsame(PAR,C,N,L,true)
  :-
   get_most_strongly_confirmed_hyp(L,VAL,CF),
   CF =< Ø.2.

notsame(PAR,C,N,L,false).

mightbe(PAR,C,N,[],false).
mightbe(PAR,C,N,L,true)
  :-
   get_most_strongly_confirmed_hyp(L,VAL,CF),
   CF > - Ø.2.

mightbe(PAR,C,N,L,false).

thoughnot(PAR,C,N,L,CF)
  :-
   CF < - Ø.2.

vnotknown(PAR,C,N,[],false).
vnotknown(PAR,C,N,L,true)
  :-
   get_most_strongly_confirmed_hyp(L,VALs,CFs),
   absolute_value(CF,CFabs),
   CFabs =< Ø.2.

vnotknown(PAR,C,N,L,false).

defis(PAR,C,N,[],false).
defis(PAR,C,N,L,true)
  :-
   get_most_strongly_confirmed_hyp(L,VAL,CF),
   CF = 1.

defis(PAR,C,N,L,false).

notdefis(PAR,C,N,[],false).
notdefis(PAR,C,N,L,true)
  :-
   get_most_strongly_confirmed_hyp(L,VAL1,CF),
   CF > Ø.2, CF < 1.

notdefis(PAR,C,N,L,false).

defnot(PAR,C,N,[],false).
defnot(PAR,C,N,L,true)
  :-
   get_most_strongly_confirmed_hyp(L,VAL,CF),
   CF = -1.

defnot(PAR,C,N,L,false).

notdefnot(PAR,C,N,[],false).
notdefnot(PAR,C,N,L,true)
  :-
   get_most_strongly_confirmed_hyp(L,VALs,CF),
   CF < - Ø.2, CF > -1.

notdefnot(PAR,C,N,L,false).

known(PAR,C,N,L,true)
  :-
   v_func_2(PAR,valutype,yes_no),
   get_most_strongly_confirmed_hyp(L,VAL,CF),
   absolute_value(CF,CFabs),
   CFabs > Ø.2.

known(PAR,C,N,L,true)
  :-
   get_most_strongly_confirmed_hyp(L,VAL,CF),
   CF > Ø.2.
```

```
known(PAR,C,N,L,false).

notknown(PAR,C,N,L,true)
  :-
   v_func_2(PAR,valutype,yes_no),
   get_most_strongly_confirmed_hyp(L,VAL,CF),
   absolute_value(CF,CFabs),
   CFabs =< Ø.2.

notknown(PAR,C,N,L,true)
  :-
   get_most_strongly_confirmed_hyp(L,VAL,CF),
   CF =< Ø.2.

notknown(PAR,C,N,L,false).
definite(PAR,C,N,L,true)
  :-
   v_func_2(PAR,valutype,yes_no),
   get_most_strongly_confirmed_hyp(L,VAL,CF),
   absolute_value(CF,CFabs),
   CFabs = 1.

definite(PAR,C,N,L,true)
  :-
   get_most_strongly_confirmed_hyp(L,VAL,CF),
   CF = 1.
```

```
definite(PAR,C,N,L,false).


notdefinite(PAR,C,N,L,true)
  :-
   get_most_strongly_confirmed_hyp(L,VAL1,CF),
   CF < 1, CF > -1.


notdefinite(PAR,C,N,L,true)
  :-
   get_most_strongly_confirmed_hyp(L,VAL1,CF1),
   CF1 < 1.


notdefinite(PAR,C,N,L,false).


/************  NUMERICAL FUNCTIONS  ****************/
     Numerical functions return "true" if the value of
"VALx" is  known with  a CF  >=   Ø.2 and is  greater/
greater or equal/ less/ less or equal than the [Value]
specified.
/***********************************************************/
greaterp(PAR,C,N,[Value],true) :-
               eval_num_val(PAR,C,N,Value,greaterp).
greateq(PAR,C,N,[Value],true)  :-
               eval_num_val(PAR,C,N,Value,greateq).
lessp(PAR,C,N,[Value],true) :-
               eval_num_val(PAR,C,N,Value,lessp).
```

```
lesseq(PAR,C,N,[Value],true) :-
               eval_num_val(PAR,C,N,Value,lesseq).

eval_num_val(PAR,C,N,Value,FUNC)
  :-
   find_appl_cntxt_for_C_N(C,N,PAR),

bagof(CF,concluded_PAR_for_C_N(PAR,Cx,Nx,VAL,CF),L),
   min(L,X),X > Ø.2,
   concluded_PAR_for_C_N(PAR,C,N,VALx,X),
   satisfied(FUNC,Value,VALx).

satisfied(greaterp,Value,VALx) :- VALx > Value.
satisfied(greateq,Value,VALx) :- VALx >= Value.
satisfied(lessp,Value,VALx) :- VALx < Value.
satisfied(lesseq,Value,VALx) :- VALx =< Value.

/**********  GET_MOST_STRONGLY_CONFIRMED_HYP  ******/
     L  is  list  of VAL,CF pairs. "get_most_strongly_
confirmed_hyp" routine returns to VAL,CF pair which CF
is largest value of list L.
/****************************************************/
get_most_strongly_confirmed_hyp([[VAL,CF]],VAL,CF).
get_most_strongly_confirmed_hyp(L,VAL,CF)
  :-
   member([X,CF1],L),
   member2([X,CF1],L,[Y,CF2]),
   ((CF1 =< CF2,delete([X,CF1],L,L1));
   (CF2 =< CF1,delete([Y,CF2],L,L1))),
   get_most_strongly_confirmed_hyp(L1,VAL,CF).
```

% FOLLOWING LIST OF CODES ARE CONTENTS OF UTILITIES FILE

```
/*************** INITIALIZATION ROUTINES ***********/
     Three properties of a context are dynamically
stored in database. These properties are : askable,
nextnum, and pasked. They are initialized to "Ø" at
the beginning of a consultation.
     Context might have more than one spring.
/****************************************************/
initialize_askable(C,N)
  :-
   v_func_2(C,offspring,Cc),
   add_4(Cc,C,N,askable,1),fail.

initialize_nextnum_pasked
  :-
   context(C),
   add_2(C,nextnum,Ø),
   add_2(C,pasked,Ø),fail.

update_askable(C,Cp,Np,askable,Num)
  :-
   delete_4(C,Cp,Np,askable,N),
   add_4(C,Cp,Np,askable,Num).
```

```
update_num(C,XX,N)
  :-
   delete_2(C,XX,N1),
   add_2(C,XX,N).

cntxt_applicable(CNTXT,PAR)
  :-
   v_func_2(CNTXT,parmgroup,PT),
   v_func_2(PAR,memberof,P_categ),
   PT == P_categ.

/**********   VARIABLE PREDICATE ROUTINES  **********/
     Some of  the predicate  names are  bound to their
values dynamically  during the cansultation, following
routines make their use possible with  PROLOG's built-
in "=.." and "call" functions.
/*****************************************************/
add_2(A,B,C)
  :-
   Z=..[A,B,C],
   assert(Z).

add_4(A,B,C,D,E)
  :-
   Z=..[A,B,C,D,E],
   assert(Z).
```

```
add_5(A,B,C,D,E,F)
  :-
   Z=..[A,B,C,D,E,F],
   assert(Z).

delete_2(A,B,C)
  :-
   Z=..[A,B,C],call(Z),
   retract(Z).

delete_4(A,B,C,D,E)
  :-
   Z=..[A,B,C,D,E],call(Z),
   retract(Z).

delete_5(A,B,C,D,E,F)
  :-
   Z=..[A,B,C,D,E,F],call(Z),
   retract(Z).

v_func_1(PRED,VAR1)
  :-
   Z=..[PRED,VAR1],call(Z).

v_func_2(PRED,VAR1,VAR2)
  :-
```

```
   Z=..[PRED,VAR1,VAR2],call(Z).


v_func_4(PRED,VAR1,VAR2,VAR3,VAR4)
  :-
   Z=..[PRED,VAR1,VAR2,VAR3,VAR4],call(Z).


v_func_5(PRED,VAR1,VAR2,VAR3,VAR4,VAR5)
  :-
   Z=..[PRED,VAR1,VAR2,VAR3,VAR4,VAR5],call(Z).


retractall(CLAUSE)
  :-
   (CLAUSE =..  [PRED,A,B,C,D,E]  ;  CLAUSE =..
   [PRED,A]),
   not(retractall2(PRED)).


retractall2(PRED)
  :-
   (Z =.. [PRED,A,B,C,D,E]  ;  Z =.. [PRED,A]),
   retract(Z),fail.


ancestor_descendant(C,N,C,N).
ancestor_descendant(C,N,Cx,Nx)
  :-
   v_func_5(Cx,C,N,Cx,Nx,tree).
```

```
ancestor_descendant(C,N,Cx,Nx)
  :-
   v_func_5(Cs,Cs,Ns,C,N,tree),
   ancestor_descendant(Cs,Ns,Cx,Nx).


/**************  CHECKING USER'S RESPONSE  *********/
     User's response for  a  data  request  is checked
against expected value of a parameter. There are three
possible expected values : a number, yes or no and any
value.
/****************************************************/
check_the_answer(C,N,PAR,CF,VAL,EXPECT)
  :-
   EXPECT == 'number',
   number(VAL).

check_the_answer(C,N,PAR,CF,VAL,EXPECT)
  :-
   EXPECT == 'yes_no',
   member(VAL,[yes,no]).

check_the_answer(C,N,PAR,CF,VAL,EXPECT)
  :-
   EXPECT == 'any'.
```

```
/*********   LOCATING HYPOTHESES IN THE DATABASE  ***/
```

"Commonlist" routine returns list of "VAL,CF" pairs where they satisfy specified values in the "eval_premise" routine. "find_appl_cntxt_for_C_N" routine finds all hypotheses in the database which concludes a value for "PAR" and stores them as facts in the form; "conclude_PAR_for_C_N (PAR,C,N,VAL,CF)". "Commonlist" routine has two choices, either a value list is specified or not. If the value list is not specified then "List" in the argument list is variable. All "conclude_PAR_for_C_N(...)" facts are retrieved and then "VAL,CF" pairs are returned as commonlist "Lcommon" in the argument list of the routine. Second choice is the case where a list of values are specified. In this
case "Hypothesislist" variable corresponds to all "VAL,CF" pairs of "conclude_PAR_for_C_N" facts. This list is intersected with specified list and resulting list is returned as "commonlist".

```
/*****************************************************/
commonlist(PAR,C,N,[List|L],Lcommon)
  :-
   var(List),
   find_appl_cntxt_for_C_N(C,N,PAR),
   bagof([VAL,CF],
concluded_PAR_for_C_N(PAR,Cx,Nx,VAL,CF),Lcommon).
```

```
commonlist(PAR,C,N,[List],Lcommon)
:-
 find_appl_cntxt_for_C_N(C,N,PAR),!,
 bagof([VAL,CF],
 concluded_PAR_for_C_N(PAR,Cx,Nx,VAL,CF),Hypo
 thesislist),
 ((v_func_2(List,list,L),
 intersection(L,Hypothesislist,Lcommon));
 intersection([List],Hypothesislist,Lcommon)).

intersection(L,[],[]).
intersection(L,[[X,Y]|L1],[[X,Y]|L2])
  :-
   member(X,L),
   intersection(L,L1,L2).

intersection(L,[[X,Y]|L1],L2)
  :-
   intersection(L,L1,L2).

/*****************************************************/
     Find applicable contexts for current context
instance (C,N) and parameter (PAR).
/*****************************************************/
```

```
 find_appl_cntxt_for_C_N(C,N,PAR)
  :-
   check_applicable_context(PAR,Ca),
   not(find_all_appl_descendants(C,N,Ca,PAR)),
   not(do_assertion(C,N,PAR)).

check_applicable_context(PAR,C)
  :-
   v_func_2(PAR,memberof,P_categ),
   context(C),
   v_func_2(C,parmgroup,P_categ).

/*****************************************************/
     Current context is already the one which is
applicable to PAR, so there is no need to look for any
descendant.
/*****************************************************/
 find_all_appl_descendants(C,N,Ca,PAR)
  :-
   C == Ca,
   not(applicable_descendant(C,N,PAR,C,N)),
   assert(applicable_descendant(C,N,PAR,C,N)),fail.

find_all_appl_descendants(C,N,Ca,PAR)
  :-
   find_brother(C,Ca,Na),
   not(applicable_descendant(C,N,PAR,Ca,Na)),
   assert(applicable_descendant(C,N,PAR,Ca,Na)),fail.

/*****************************************************/
     If C,N is immediate parent for applicable context
then  the  fact "applicable_descendant(C,N,PAR,Cx,Nx)"
is  asserted  into  dbase for all immediate descendant
contexts of C,N. Note  that  "C,N,PAR"  triple  is our
key.
/*****************************************************/
find_all_appl_descendants(C,N,Ca,PAR)
  :-
   v_func_5(Ca,C,N,Ca,Na,tree),
   not(applicable_descendant(C,N,PAR,Ca,Na)),
   assert(applicable_descendant(C,N,PAR,Ca,Na)),fail.

find_all_appl_descendants(C,N,Ca,PAR)
  :-
   v_func_5(C,Ca,Na,C,N,tree),
   not(applicable_descendant(C,N,PAR,Ca,Na)),
   assert(applicable_descendant(C,N,PAR,Ca,Na)),fail.

/*****************************************************/
     Applicable context is not  reached  yet,  go down
one more level.
/*****************************************************/
```

```
find_all_applicable_descendants(C,N,Ca,PAR)
  :-
   not(applicable_descendant(C,N,PAR,Cx,Nx)),
   v_func_2(C,offspring,Cs),
   v_func_5(Cs,C,N,Cs,Ns,tree),
   find_all_applicable_descendants(Cs,Ns,Ca,PAR).

/*****************************************************/
    HERE by using "fail" we use all applicable
descendant contexts one by one and assert
"concluded_PAR_for_C_N" for each of them.
/*****************************************************/
do_assertion(C,N,PAR)
  :-
   applicable_descendant(C,N,PAR,Ca,Na),
   not(do_assertion2(C,N,PAR,Ca,Na)),fail.

do_assertion2(C,N,PAR,Ca,Na)
  :-
   hypothesis(PAR,Ca,Na,VAL,CF),
   not(concluded_PAR_for_C_N(PAR,Ca,Na,VAL,CF)),

assert(concluded_PAR_for_C_N(PAR,Ca,Na,VAL,CF)),fail.
/*****************************************************/
    DON'T assert  if it is already asserted.
/*****************************************************/
```

```
do_assertion2(C,N,PAR,Ca,Na)
  :-
   hypothesis(PAR,Ca,Na,VAL,CF),fail.


find_brother(C,Cb,Nb)
  :-
   v_func_5(C,Cp,Np,C,N,tree),
   v_func_5(Cb,Cp,Np,Cb,Nb,tree),
   Cb \== C.


absolute_value(CF,CFabs)
  :-
   CF < Ø,
   CFabs is CF * -1.


absolute_value(CF,CF).


min(A,B) :- min2(A,B),!.


min2([X],X).
min2([X¦L],X) :- min2(L,Y),X =< Y.
min2([X¦L],Y) :- min2(L,Y).


max(A,B) :- max2(A,B),!.


max2([X],X).
```

```
max2([X|L],X) :- max2(L,Y),X >= Y.

max2([X|L],Y) :- max2(L,Y).


insert(A,[B],[A|B]).


delete(X,[X|L],L).

delete(X,[Y|L1],[Y|L2]) :- delete(X,L1,L2).


member(X,[]) :- !,fail.

member(X,[X|L]).

member(X,[Y|L]) :- member(X,L).


affirmative(y).

affirmative(yes).

negative(no).

negative(n).


member2([X,CF1],[[X,CF1]|L1],[Y,CF2]) :-

member([Y,CF2],L1).
```

APPENDIX B

LIST OF FUNCTIONS

This appendix contains the list of functions used in EMYCIN-PROLOG. Their original descriptions are given in [2].

NONNUMERIC PREDICATE FUNCTIONS

KNOWN

Returns true if the value of the parameter is known with a CF > Ø.2.

NOTKNOWN

Returns true if the CF of the parameter is less than or equal to Ø.2.

DEFINITE

Returns true if the value of the parameter is known with certainty (CF = 1.Ø).

SAME

Returns the CF associated with the value of interest if it is greater than Ø.2, otherwise returns false.

NOTSAME

Returns true if the CF associated with the value of interest is less than or equal to Ø.2.

MIGHTBE

Returns true if the CF associated with the value of interest is greater than or equal to Ø.2.

THOUGHTNOT

Returns -CF associated with the value of interest if it is less than -Ø.2, otherwise returns false.

VNOTKNOWN

Returns true if the CF associated with the value of interest lies between -Ø.2 and 1.

DEFIS

Returns true if the CF associated with the value of interest is equal to 1.

DEFNOT

Returns true if the CF associated with the value of interest is equal to -1.

NOTDEFIS

Returns true if the CF associated with the value of interest lies between Ø.2 and 1.

NOTDEFNOT

Returns true if the CF associated with the value of interest lies between -1 and -Ø.2.

NUMERIC PREDICATE FUNCTIONS

GREATERP

Returns true if the value of interest is known with a CF> Ø.2 and is greater than or equal to the number specified.

GREATEQ

Returns true if the value of interest is known with a CF> Ø.2 and is greater than or equal to the number specified.

LESSP

Returns true if the value of interest is known with a CF> Ø.2 and is less than the number specified.

LESSEQ

Returns true if the value of interest is known with a CF> Ø.2 and is less than or equal to the number specified.

CONCLUSION FUNCTIONS

CONCLUDE

Updates the value of a parameter in the dynamic database. Update process includes combining certainty values and explained in Section III.A.

APPENDIX C
KNOWLEDGE BASES

This appendix contains listing of the static knowledge and rulebase of the CAR diagnosis system and FINANCE analysis system, which are held in the files CARRULES, and FINANCERULES. Each file contains all static knowledge/information about contexts and parameters too.

Prolog rules and facts which presented in this appendix are written by knowledge engineer. This process corresponds to the knowledge base construction phase of the expert system development process.

Following are some of the the cautions about writing rules for EMYCIN-PROLOG. The knowledge engineer should be careful in these details.

Since all rules are tried, every rule should include all required premises explicitly.

Premises which has VAR as a value list has to be treated differently. After execution of a premise clause PAR value is asserted into data base as "hypothesis(PAR,CNTXT,N,VAL,CF)", this fact should be called explicitly to be able to use VAL in other premises of a rule.

In the "conclude" clause of each rule CF value should be passed by "min" or "max" function. Otherwise

CF value should be defined explicitly.

In "concluded" routine tree pointer should be a variable different than current tree pointer.

The value to be searched for, should be enclosed in brackets in "eval_premise" routine.

All rules are tried unless PAR is singlevalued and its value is concluded with CF=1(-1) in any of previous rules. The point that all rules are tried should be remembered during the rule writing process otherwise surprising answers can be obtained !!!

1. CAR DIAGNOSIS SYSTEM KNOWLEDGE BASE

```
/*************** CONTEXT DEFINITIONS ***************/
context(nnil).
/***************************************************/
```

This fact is required for "initialize_askable" routine

```
/***************************************************/
nnil(offspring,car).
context(car).
car(offspring,electrical_system).
car(offspring,fuel_system).
car(assocwith,nnil).
car(parmgroup,car_parms).
car(prompt3,'This is a car diagnoses program').
car(mainprops,[year,model,problems]).


context(electrical_system).
electrical_system(offspring,nnil).
electrical_system(assocwith,car).
electrical_system(parmgroup,elec_parms).
electrical_system(prompt3,'Electrical system  needs to
be checked !! ?').
electrical_system(mainprops,[]).


context(fuel_system).
fuel_system(offspring,nnil).
fuel_system(assocwith,car).
fuel_system(parmgroup,fuel_parms).
fuel_system(prompt3,'Fuel  system  needs to be checked
!! ?').
fuel_system(mainprops,[throttle_test]).


/********************  PARAMETER DEFINITIONS ********/
parameter(year).
year(memberof,car_parms).
year(valutype,singlevalued).
year(expect,number).
year(prompt,year_prompt).
year(can_ask,1).
```

```
year_prompt
  :-
   write(' What is the year of the car ?'),
   write(' ==> ').

parameter(model).
model(memberof,car_parms).
model(valutype,singlevalued).
model(expect,any).
model(prompt,model_prompt).
model(can_ask,1).

model_prompt
  :-
   write('What is the model of the car ?'),write(' ==>
   ').

parameter(problems).
problems(memberof,car_parms).
problems(valutype,singlevalued).
problems(expect,any).
problems(can_ask,Ø).

parameter(stalled_engine).
stalled_engine(memberof,elec_parms).
stalled_engine(valutype,multivalued).
stalled_engine(expect,any).
stalled_engine(can_ask,Ø).
stalled_engine(trans,'The cause  of the stalled engine
problem is : ').

parameter(electrical).
electrical(memberof,elec_parms).
electrical(valutype,multivalued).
electrical(expect,any).
electrical(can_ask,Ø).

parameter(battery).
battery(memberof,elec_parms).
battery(valutype,multivalued).
battery(expect,number).
battery(can_ask,Ø).

parameter(dimming_light).
dimming_light(memberof,elec_parms).
dimming_light(valutype,yes_no).
dimming_light(expect,yes_no).
dimming_light(prompt,dimming_light_prompt).
dimming_light(can_ask,1).
```

```
dimming_light_prompt
  :-
   print('Turn on your lights and operate  the starter
  '), nl,
   print('Do the  lights go out or become dim ?
   (yes/no) '),
   write(' ==> ').

parameter(hydrometer).
hydrometer(memberof,elec_parms).
hydrometer(valutype,singlevalued).
hydrometer(expect,number).
hydrometer(prompt,hydrometer_prompt).
hydrometer(can_ask,1).

hydrometer_prompt
  :-
   print('What is the specific gravity measured by
   hydrometer ?'),
   write(' ==> ').

parameter(battery_volt).
battery_volt(memberof,elec_parms).
battery_volt(valutype,singlevalued).
battery_volt(expect,number).
battery_volt(prompt,battery_volt_prompt).
battery_volt(can_ask,1).


battery_volt_prompt
  :-
   print('Disconnect the  battery connections and
   measure the voltage'),nl,
   print('What is the voltage measured on battery ?'),
   write(' ==> ').


parameter(ammeter).
ammeter(memberof,elec_parms).
ammeter(valutype,yes_no).
ammeter(expect,yes_no).
ammeter(prompt,ammeter_prompt).
ammeter(can_ask,1).


ammeter_prompt
  :-
   print('Does the  ammeter shows a slight discharge
   (or does the '),nl,
   print('telltale lamp light) when the ignition  is
   turned  on.?   (yes/no)'),
   write(' ==> ').


parameter(starting_motor).
starting_motor(memberof,elec_parms).
starting_motor(valutype,yes_no).
starting_motor(expect,yes_no).
starting_motor(prompt,starting_motor_prompt).
starting_motor(can_ask,1).

starting_motor_prompt
  :-
   print('Does the  electrical system go dead when the
   starter switch '),nl,
   print('is turned on?      (yes/no)'),
   write(' ==> ').

parameter(fuel_sys).
fuel(memberof,fuel_parms).
fuel(valutype,multivalued).
fuel(expect,any).
fuel(can_ask,Ø).

parameter(fuel_to_carb).
fuel_to_carb(memberof,fuel_parms).
fuel_to_carb(valutype,single).
fuel_to_carb(expect,any).
fuel_to_carb(can_ask,Ø).

parameter(throttle_test).
throttle_test(memberof,fuel_parms).
throttle_test(valutype,yes_no).

throttle_test(expect,yes_no).

throttle_test(prompt,throttle_test_prompt).

throttle_test(can_ask,1).


throttle_test_prompt

  :-

   print('Move the throttle manually, do you see a

   spray of fuel'),nl,

   print('mixture in  the carburator throat. ?

   (yes/no)'),

   write(' ==> ').


parameter(fuel_pump).

fuel_pump(memberof,fuel_parms).

fuel_pump(valutype,yes_no).

fuel_pump(expect,yes_no).

fuel_pump(prompt,fuel_pump_prompt).

fuel_pump(can_ask,1).


fuel_pump_prompt

  :-

   print(' Disconnect the fuel line at the

   carburator.'),nl, print(' Crank the engine '),

   print('Do you see fuel pulsating out '),nl,

   print('of the line. ?  (yes/no) '),write(' ==> ').
```

```
parameter(fuel_line).
fuel_line(memberof,fuel_parms).
fuel_line(valutype,yes_no).
fuel_line(expect,yes_no).
fuel_line(prompt,fuel_line_prompt).
fuel_line(can_ask,1).

fuel_line_prompt
  :-
   print(' Disconnect the fuel line at the '),
   print('inlet side of the pump.'),nl,
   print(' Blow into the line, '),
   print('Does your friend hear gurgling sound'),nl,
   print('from the  fuel tank inlet.?  (yes/no)   '),
   write('==> ').

parameter(fuel_filter).
fuel_filter(memberof,fuel_parms).
fuel_filter(valutype,yes_no).
fuel_filter(expect,yes_no).
fuel_filter(prompt,fuel_filter_prompt).
fuel_filter(can_ask,1).

fuel_filter_prompt
  :-
```

```
   print(' Disconnect the inlet side of the line
   filter (usually'),nl,
   print('a small, clear plastic canister spliced into
   the fuel'),nl,
   print('line) Crank the engine. Do you see a  good
   shot of fuel. ?'),
   print(' (yes/no)'),write(' ==> ').

/**********************   RULES   **********************/
     The  "cut"  operator  (!)  is  used   to  prevent
backtracking.    "try_all_rules_for_PAR" routine tries
other parameters  if it   cannot  find then backtracks
so, "!" stops it and ends consultation.
/*****************************************************/
problems(CNTXT,N,PROBLEMS,CFrule)
  :-
   print('What is/are the problem(s) ?'),nl,
   print('1.Stalled engine'),nl,
   print('2.Dieseling'),nl,
   print('3.Engine noise'),nl,
   print('4.Slow cranking'),nl,
   print('5.Hard starting'),nl,
   print('6.Rough idle'),nl,nl,nl,
   print('Enter the  number which corresponds to the
   problem '),
   write(' ==> '),
   read(NUMBER),nl,
   problem_fact(NUMBER,PROBLEM),
   assert(goal(PROBLEM)),
   eval_par(CNTXT,[PROBLEM],N),nl,nl,!.

/*****************************************************/
     If any one of the rules can be satisfied more
than once, then all such choices are tried since
'try_all_rules_for_PAR' routine uses "fail" predicate
and any individual rule will be tried until all
possible choices are satisfied in the rule premise.
For example in first stalled_engine rule VALc,CFc can
bind more than one values, if there are more than one
hypotheses in database which concludes a value about
"electrical".
/*****************************************************/
stalled_engine(CNTXT,N,VALc,1)
  :-
   eval_premise(same,electrical,CNTXT,N,VAL,CF),
   hypothesis(electrical,Cx,Nx,VALc,CFc),
   conclude(CNTXT,N,stalled_engine,VALc,1,CFc).

stalled_engine(CNTXT,N,VALc,1)
   :-
    eval_premise(same,fuel,CNTXT,N,VAL,CF),
    hypothesis(fuel,Cx,Nx,VALc,CFc),
    conclude(CNTXT,N,stalled_engine,VALc,1,CFc).

electrical(CNTXT,N,'battery',Ø.8)
  :-
   eval_premise(same,dimming_light,CNTXT,N,[yes],CF1),
   eval_premise(same,battery,CNTXT,N,[weak],CF2),
   min([CF1,CF2],CF),
   conclude(CNTXT,N,electrical,'battery',Ø.8,CF).

electrical(CNTXT,N,'neutral_safety_switch',Ø.7)
  :-
eval_premise(same,ammeter,CNTXT,N,[yes],CF1),
eval_premise(same,starting_motor,CNTXT,N,[yes],CF2),
min([CF1,CF2],CF),
conclude(CNTXT,N,electrical,
              'neutral_safety_switch',Ø.7,CF).

electrical(CNTXT,N,'starter_circuit',Ø.6)
  :-
eval_premise(same,dimming_light,CNTXT,N,[no],CF1),
eval_premise(same,fuel,CNTXT,N,[ok],CF2),
min([CF1,CF2],CF),
conclude(CNTXT,N,electrical,'starter_circuit',Ø.6,CF).

battery(CNTXT,N,'weak',1)
  :-
```

```
eval_premise(lessp,hydrometer,CNTXT,N,[1250],true),
eval_premise(lessp,battery_volt,CNTXT,N,[12],true),
conclude(CNTXT,N,battery,'weak',1,1).

battery(CNTXT,N,'bad_connections',0.8)
  :-
eval_premise(greateq,hydrometer,CNTXT,N,[1250],true),
eval_premise(greateq,battery_volt,CNTXT,N,[12],true),
conclude(CNTXT,N,battery,'bad_connections',1,0.8).

battery(CNTXT,N,'weak',0.5)
  :-
eval_premise(greateq,hydrometer,CNTXT,N,[1250],true),
eval_premise(lessp,battery_volt,CNTXT,N,[12],true),
conclude(CNTXT,N,battery,'weak',0.5,1).

battery(CNTXT,N,'weak',0.5)
  :-
eval_premise(lessp,hydrometer,CNTXT,N,[1250],true),
eval_premise(greateq,battery_volt,CNTXT,N,[12],true),
conclude(CNTXT,N,battery,'weak',0.5,1).

fuel(CNTXT,N,'fuel_pump',1)
   :-
eval_premise(defis,throttle_test,CNTXT,N,[no],true),
eval_premise(same,fuel_pump,CNTXT,N,[no],CF1),
eval_premise(same,fuel_filter,CNTXT,N,[no],CF2),
eval_premise(same,fuel_line,CNTXT,N,[yes],CF3),
min([CF1,CF2,CF3],CF),
conclude(CNTXT,N,fuel,'fuel_pump',1,CF).

fuel(CNTXT,N,'fuel_line',1)
   :-
eval_premise(defis,throttle_test,CNTXT,N,[no],true),
eval_premise(same,fuel_pump,CNTXT,N,[no],CF1),
eval_premise(same,fuel_filter,CNTXT,N,[no],CF2),
eval_premise(same,fuel_line,CNTXT,N,[no],CF3),
min([CF1,CF2,CF3],CF),
conclude(CNTXT,N,fuel,'fuel_line',1,CF).

fuel(CNTXT,N,'carburator',1)
   :-
eval_premise(defis,throttle_test,CNTXT,N,[no],true),
eval_premise(same,fuel_pump,CNTXT,N,[yes],CF),
conclude(CNTXT,N,fuel,'carburator',1,CF).

fuel(CNTXT,N,'carburator',1)
   :-
eval_premise(defis,throttle_test,CNTXT,N,[yes],true),
eval_premise(same,starting_motor,CNTXT,N,[yes],CF),
conclude(CNTXT,N,fuel,'carburator',1,CF).
```

```
fuel(CNTXT,N,'fuel_filter',1)
   :-
eval_premise(defis,throttle_test,CNTXT,N,[no],true),
eval_premise(same,fuel_pump,CNTXT,N,[no],CF1),
eval_premise(same,fuel_filter,CNTXT,N,[yes],CF2),
min([CF1,CF2],CF),
conclude(CNTXT,N,fuel,'fuel_filter',1,CF).

/***************  FACTS ABOUT MAIN MENU   ***********/
problem_fact(1,stalled_engine).
problem_fact(2,dieseling).
problem_fact(3,engine_noise).
problem_fact(4,slow_cranking).
problem_fact(5,hard_starting).
problem_fact(6,rough_idle).
```

2.   FINANCE ANALYSIS SYSTEM KNOWLEDGE BASE

```
/*************  CONTEXT DEFINITIONS  ****************/
context(nnil).
nnil(offspring,lease).
goal(payment).
```

```
context(lease).
lease(offspring,finance).
lease(assocwith,nnil).
lease(parmgroup,lease_parms).
lease(prompt3,'The following is a part of a
lease/acquire_by/finance DSS').
lease(mainprops,[payment]).
lease(prompt2,'Is  there  any  other lease problem you
want to solve ?').

context(finance).
finance(offspring,nnil).
finance(assocwith,lease).
finance(parmgroup,finance_parms).
finance(prompt1,'Do you want to  analyze the financing
for asset ?').
finance(mainprops,[]).
finance(prompt2,'Do  you  have  any  other  finance to
analyze ?').

/**************  PARAMETER DEFINITIONS  *************/
parameter(asset_cost).
asset_cost(memberof,finance_parms).
asset_cost(valutype,singlevalued).
asset_cost(expect,number).
asset_cost(prompt,asset_cost_prompt).
asset_cost(can_ask,1).
asset_cost(trans,'the cost of the asset ').

asset_cost_prompt
  :-
   write('What is the asset cost'),
   write('  ==>').

parameter(down_payment).
down_payment(memberof,finance_parms).
down_payment(valutype,singlevalued).
down_payment(expect,number).
down_payment(prompt,down_payment_prompt).
down_payment(can_ask,1).
down_payment(trans,'amount  of  down  payment  for the
asset').

down_payment_prompt
  :-
   write('What is the amount of down payment ?'),
   write('  ==>').

parameter(finance_it).
finance_it(memberof,finance_parms).
finance_it(valutype,singlevalued).
finance_it(expect,number).
finance_it(can_ask,Ø).
finance_it(trans,'The yearly payment on the asset').

parameter(finance_interest).
finance_interest(memberof,finance_parms).
finance_interest(valutype,singlevalued).
finance_interest(expect,number).
finance_interest(prompt,finance_interest_prompt).
finance_interest(can_ask,1).
finance_interest(trans,'The  percentage  yield  to the
firm for the loan').

finance_interest_prompt
  :-
   write('Percent charged by the leasing firm ?'),
   write('  ==>').

parameter(finance_period).
finance_period(memberof,finance_parms).
finance_period(valutype,singlevalued).
finance_period(expect,number).
finance_period(prompt,finance_period_prompt).
finance_period(can_ask,1).
finance_period(trans,'The  length  in  years   of  the
leasing period line for the asset').
```

```
finance_period_prompt
  :-
   write('Lease period ?'),
   write('  ==>').

parameter(option_lease).
option_lease(memberof,finance_parms).
option_lease(valutype,yes_no).
option_lease(expect,yes_no).
option_lease(prompt,option_lease_prompt).
option_lease(can_ask,1).
option_lease(trans,'Lease is to be modified lease').

option_lease_prompt
  :-
   write('Do you  want a lease with the option to
   terminate ?'),
   write('  ==>').

parameter(straight_lease).
straight_lease(memberof,finance_parms).
straight_lease(valutype,yes_no).
straight_lease(expect,yes_no).
straight_lease(prompt,straight_lease_prompt).
straight_lease(can_ask,1).
straight_lease(trans,'Lease  is  to   be   a  straight
lease').

straight_lease_prompt
  :-
   write('Do you want a straight lease ?'),
   write('  ==>').

parameter(asset_name).
asset_name(memberof,lease_parms).
asset_name(valutype,singlevalued).
asset_name(expect,any).
asset_name(prompt,asset_name_prompt).
asset_name(can_ask,1).
asset_name(trans,'The  asset  that tou are considering
for').

asset_name_prompt
  :-
   write('Asset name ?'),
   write('  ==>').

parameter(acquire_by).
acquire_by(memberof,lease_parms).
acquire_by(valutype,singlevalued).
acquire_by(expect,any).
acquire_by(can_ask,Ø).
acquire_by(trans,'Determination to  straight  lease or
acquire_by the asset').

parameter(cannot_borrow).
cannot_borrow(memberof,lease_parms).
cannot_borrow(valutype,yes_no).
cannot_borrow(expect,yes_no).
cannot_borrow(can_ask,Ø).
cannot_borrow(trans,'Your credit  is too  low to get a
loan').
parameter(cash_reserve_needed).
cash_reserve_needed(memberof,lease_parms).
cash_reserve_needed(valutype,yes_no).
cash_reserve_needed(expect,yes_no).
cash_reserve_needed(prompt,
cash_reserve_needed_prompt).
cash_reserve_needed(can_ask,1).
cash_reserve_needed(trans,'You  do  need  to  maintain
large cash reserves').

cash_reserve_needed_prompt
   :-
write('Do you  need to maintain larger cash reserves ?
(yes/no)'),
write('  ==>').
```

```
parameter(how_to_acquire).
how_to_acquire(memberof,lease_parms).
how_to_acquire(valutype,multivalued).
how_to_acquire(expect,any).
how_to_acquire(can_ask,Ø).
how_to_acquire(trans,'My recommendation').

parameter(lender_checks).
lender_checks(memberof,lease_parms).
lender_checks(valutype,yes_no).
lender_checks(expect,yes_no).
lender_checks(prompt,lender_checks_prompt).
lender_checks(can_ask,1).
lender_checks(trans,'Lender does  check on outstanding
leases when making a loan').

lender_checks_prompt
  :-
   write('When you  go to borrow money , Does the
  lender check'),nl,
   write('on any outstanding leases you have ?
  (yes/no)'),
   write('  ==>').

parameter(lessee_cash).
lessee_cash(memberof,lease_parms).
lessee_cash(valutype,singlevalued).
lessee_cash(expect,any).
lessee_cash(prompt,lessee_cash_prompt).
lessee_cash(can_ask,1).
lessee_cash(trans,'The cash reserves').

lessee_cash_prompt
  :-
   write('how would  you describe your cash reserves ?
   (good/fair/poor)'),
   write('  ==>').

parameter(lessee_credit).
lessee_credit(memberof,lease_parms).
lessee_credit(valutype,singlevalued).
lessee_credit(expect,any).
lessee_credit(prompt,lessee_credit_prompt).
lessee_credit(can_ask,1).
lessee_credit(trans,'Your credit rating').

lessee_credit_prompt
  :-
  write('How would you describe your current  credit
  rating ?  (good/fair/poor)'),
  write('  ==>').
parameter(payment).
```

```
payment(memberof,lease_parms).
payment(valutype,singlevalued).
payment(expect,any).
payment(can_ask,Ø).
payment(trans,'Payment on  the asset  for the asset is
($)         :').

parameter(preserves_cash).
preserves_cash(memberof,lease_parms).
preserves_cash(valutype,yes_no).
preserves_cash(expect,yes_no).
preserves_cash(can_ask,Ø).
preserves_cash(trans,'This  lease  does  preserve your
cash reserves').

parameter(preserves_credit).
preserves_credit(memberof,lease_parms).
preserves_credit(valutype,yes_no).
preserves_credit(expect,yes_no).
preserves_credit(can_ask,Ø).
preserves_credit(trans,'This lease  does preserve your
credit rating').
```

```
/********************  RULES  ***********************/

cannot_borrow(CNTXT,N,'yes',1.0)
  :-
   eval_premise(same,lessee_credit,CNTXT,N,[poor],CF),
   conclude(CNTXT,N,cannot_borrow,'yes',1.0,CF),nl,
   write('Your credit  is not adequate.You cannot
   borrow money to acquire_by the asset '),nl,
   write('Therefore LEASE the asset '),nl.

acquire_by(CNTXT,N,'lease',1.0)
  :-
(eval_premise(same,cannot_borrow,CNTXT,N,[yes],CF1);
eval_premise(same,preserves_credit,CNTXT,N,[yes],CF2);
eval_premise(same,preserves_cash,CNTXT,N,[yes],CF3)),
conclude(CNTXT,N,how_to_acquire,'lease',1.0,CF1),
conclude(CNTXT,N,acquire_by,'lease',1.0,CF1),nl,
write('My recommendation is lease the asset'),nl.

acquire_by(CNTXT,N,'purchase',1.0)
  :-
   not(hypothesis(acquire_by,Cx,Nx,VAL,CFx)),
   conclude(CNTXT,N,acquire_by,'purchase',1.0,1.0),nl,
   write('My recommendation is buy the asset'),nl.
```

```
preserves_credit(CNTXT,N,'yes',1.Ø)
  :-
eval_premise(same,lessee_credit,CNTXT,N,[fair],CF1),
eval_premise(same,lender_checks,CNTXT,N,[no],CF2),
min([CF1,CF2],CF),
conclude(CNTXT,N,preserves_credit,'yes',1.Ø,CF),nl,
write('Your credit  rating  will  not  be  affected by
leasing the asset'),nl.

preserves_cash(CNTXT,N,'yes',.9)
  :-
eval_premise(same,lessee_credit,CNTXT,N,[fair],CF1),
eval_premise(same,lender_checks,CNTXT,N,[yes],CF2),
eval_premise(same,lessee_cash,CNTXT,N,[fair],CF3),
eval_premise(same,cash_reserve_needed,CNTXT,N,[yes],
CF4),
min([CF1,CF2,CF3,CF4],CF),
conclude(CNTXT,N,preserves_cash,'yes',.9,CF).

finance_it(CNTXT,N,VAL,1.Ø)
  :-
eval_premise(same,acquire_by,CNTXT,N,[lease],CF1),
eval_premise(same,straight_lease,CNTXT,N,[yes],CF2),
eval_premise(known,asset_cost,CNTXT,N,[VAL1],true),
eval_premise(known,finance_interest,CNTXT,N,[VAL2],
true),
eval_premise(known,finance_period,CNTXT,N,[VAL3],
true),
hypothesis(asset_cost,Cx,Nx,VAL4,CFx),
hypothesis(finance_interest,Cy,Ny,VAL5,CFy),
hypothesis(finance_period,Cz,Nz,VAL6,CFz),
min([CF1,CF2],CF),
VAL is (VAL4*(VAL5/100)+(VAL4/VAL6)+((VAL4*2)/100)),
conclude(CNTXT,N,finance_it,VAL,1.0,CF).

finance_it(CNTXT,N,VAL,1.0)
  :-
eval_premise(same,acquire_by,CNTXT,N,[lease],CF1),
eval_premise(same,option_lease,CNTXT,N,[yes],CF2),
eval_premise(known,asset_cost,CNTXT,N,[VAL1],true),
eval_premise(known,finance_interest,CNTXT,N,[VAL2],
true),
eval_premise(known,finance_period,CNTXT,N,[VAL3],
true),
hypothesis(asset_cost,Cx,Nx,VAL4,CFx),
hypothesis(finance_interest,Cy,Ny,VAL5,CFy),
hypothesis(finance_period,Cz,Nz,VAL6,CFz),
min([CF1,CF2],CF),
VAL is ((VAL4*(VAL5/100))+(VAL4/VAL6)),
conclude(CNTXT,N,finance_it,VAL,1.0,CF).
```

```
finance_it(CNTXT,N,VAL,1.0)
  :-
eval_premise(same,acquire_by,CNTXT,N,[purchase],CF),
eval_premise(known,asset_cost,CNTXT,N,[VAL1],true),
eval_premise(known,finance_interest,CNTXT,N,[VAL2],
true),
eval_premise(known,finance_period,CNTXT,N,[VAL3],
true),
eval_premise(known,down_payment,CNTXT,N,[VAL4],true),
hypothesis(asset_cost,Cx,Nx,VAL5,CFx),
hypothesis(finance_interest,Cy,Ny,VAL6,CFy),
hypothesis(finance_period,Cz,Nz,VAL7,CFz),
hypothesis(down_payment,Cw,Nw,VAL8,CFw),
VAL is (((VAL5-VAL8)/VAL7)*((100+VAL6)/100)),
conclude(CNTXT,N,finance_it,VAL,1.0,CF).

payment(CNTXT,N,VALx,1.0)
  :-
   eval_premise(same,finance_it,CNTXT,N,VAL,CF),
   hypothesis(finance_it,Cx,Nx,VALx,CFx),
   conclude(CNTXT,N,payment,VALx,1.0,1.0).
```

APPENDIX D

SAMPLE CONSULTATIONS

The sample consultations presented in this appendix occur between expert system and consultor. The consultor is in charge of finding required data. The consultor can answer any data request as "unk" which implies that there is no data available.

1. CAR DIAGNOSIS CONSULTATIONS

Depending upon the data provided by the consultor three different consultations are obtained for the CAR diagnosis system.

```
C-Prolog version 1.5
| ?- [engine,func,utilities,carrules].
engine consulted 12380 bytes 3.08333 sec.
func consulted 6572 bytes 1.98333 sec.
utilities consulted 7020 bytes 1.71667 sec.
carrules consulted 10000 bytes 2.9 sec.
 WELCOME TO EMYCIN-PROLOG CONSULTATION PROGRAM
 Please enter "begin" to start the consultation
yes
| ?- begin.
Enter  the  name  of  the  root  context  (CAR,LEASE)
==>car.
This is a car diagnoses program
----car-1----
car-1
What is the year of the car ? ==> 86.
car-1
What is the model of the car ? ==> new.
What is/are the problem(s) ?
1.Stalled engine
2.Dieseling
```

```
3.Engine noise
4.Slow cranking
5.Hard starting
6.Rough idle
Enter the number which corresponds to the problem  ==>
1.
Electrical system needs to be checked !! ?
----electrical_system-1----
electrical_system-1
Turn on your lights and operate the starter
Do the  lights go  out or  become dim ?        (yes/no)
==> si.
Unexpected answer !!! Please try again.
electrical_system-1
Turn on your lights and operate the starter
Do the lights go out or become dim ?          (yes/no)
==> yes.
electrical_system-1
What is  the specific gravity measured by hydrometer ?
==> 1200.
electrical_system-1
Disconnect the  battery  connections  and  measure the
voltage
What is the voltage measured on battery ? ==> 10.
electrical_system-1
Does the ammeter shows a slight discharge (or does the
telltale lamp light) when the ignition  is turned on.?
(yes/no) ==> yes.
electrical_system-1
Does the  electrical system  go dead  when the starter
switch  is turned on.?            (yes/no) ==> no.
Fuel system needs to be checked !! ?
----fuel_system-1----
fuel_system-1
Move the throttle manually, do you see a spray of fuel
mixture in  the carburator throat. ?      (yes/no) ==>
yes.

The cause of the stalled engine problem is :
battery
with the certainity : 0.8



 -------  CONCLUSIONS MADE DURING THE  CONSULTATION --
parameter / value / certainity / context instance
-----------------------------------------------------
year --- 86 --- 1--- car--1
model --- new --- 1--- car--1
dimming_light --- yes --- 1--- electrical_system--1
hydrometer --- 1200 --- 1--- electrical_system--1
battery_volt --- 10 --- 1--- electrical_system--1
battery --- weak --- 1--- electrical_system--1
```

```
electrical --- battery --- Ø.8--- electrical_system--1

ammeter --- yes --- 1--- electrical_system--1

starting_motor --- no --- 1--- electrical_system--1

stalled_engine ---battery---Ø.8---electrical_system--1

throttle_test --- yes --- 1--- fuel_system--1



yes

| ?- begin.

Enter  the   name  of  the  root  context  (CAR,LEASE)

==>car.

This is a car diagnoses program

----car-1----

car-1

What is the year of the car ? ==> 65.

car-1

What is the model of the car ? ==> old.

What is/are the problem(s) ?
1.Stalled engine

2.Dieseling

3.Engine noise

4.Slow cranking

5.Hard starting

6.Rough idle

Enter the number which corresponds to the problem  ==>

1.
```

```
Electrical system needs to be checked !! ?
----electrical_system-1----
electrical_system-1
Turn on your lights and operate the starter
Do the  lights go  out or  become dim ?       (yes/no)
==> yes.
electrical_system-1
What is the specific gravity measured  by hydrometer ?
==> 1300.
electrical_system-1
Disconnect  the  battery  connections  and measure the
voltage
What is the voltage measured on battery ? ==> 12.
electrical_system-1
Does the ammeter shows a slight discharge (or does the
telltale lamp  light) when the ignition is turned on.?
(yes/no) ==> yes.
electrical_system-1
Does the electrical system go dead when the starter

switch  is turned on.?          (yes/no) ==> no.
Fuel system needs to be checked !! ?
----fuel_system-1----
fuel_system-1
Move the throttle manually, do you see a spray of fuel
mixture in  the carburator throat. ?     (yes/no) ==>
no.
```

```
fuel_system-1
 Disconnect the fuel line at the carburator.
 Crank the engine.Do you see fuel pulsating out
of the line. ?  (yes/no)  ==> yes.


The cause of the stalled engine problem is :
carburator
with the certainity : 1



------  CONCLUSIONS MADE DURING THE CONSULTATION  ---
 parameter / value / certainity / context instance
-----------------------------------------------------
year --- 65 --- 1--- car--1
model --- old --- 1--- car--1
dimming_light --- yes --- 1--- electrical_system--1
hydrometer --- 1300 --- 1--- electrical_system--1
battery_volt --- 12 --- 1--- electrical_system--1
battery     ---     bad_connections     ---     0.8---
electrical_system--1
ammeter --- yes --- 1--- electrical_system--1
starting_motor --- no --- 1--- electrical_system--1
throttle_test --- no --- 1--- fuel_system--1
fuel_pump --- yes --- 1--- fuel_system--1
```

```
fuel --- carburator ---1--- fuel_system--1
stalled_engine---carburator---1---electrical_system--1




| ?- halt.

[ Prolog execution halted ]
```

2. FINANCE ANALYSIS CONSULTATIONS

The consultation results obtained for the FINANCE analysis system are the same with the original FINANCE analysis system which is built elsewhere [13]. Following the second consultation original consultation results are also given for the comparison purpose.

```
% prolog
C-Prolog version 1.5
| ?- [engine,func,utilities,financerules].
engine consulted 12344 bytes 3.06667 sec.
func consulted 6572 bytes 1.9 sec.
utilities consulted 7020 bytes 1.63333 sec.
```

```
financerules consulted 9788 bytes 2.65 sec.
 WELCOME TO EMYCIN-PROLOG CONSULTATION PROGRAM
 Please enter "begin" to start the consultation
yes
| ?- begin.
Enter  the  name  of  the  root   context  (CAR,LEASE)
==>lease.
The following  is a part of a lease/acquire_by/finance
DSS
----lease-1----
Do you want to analyze  the  financing  for  asset  ?
==>y.
----finance-1----
Do you have any other finance to analyze ?   ==>n.
lease-1
How would  you describe  your current  credit rating ?
(good/fair/poor)  ==>good.
My recommendation is buy the asset
finance-1
What is the asset cost  ==>3000.
finance-1
Percent charged by the leasing firm ?  ==>12.
finance-1
Lease period ?  ==>2.
finance-1
What is the amount of down payment ?  ==>500.
```

```
Is there any other lease problem you want to solve  ?
==>n.

Payment on the asset for the asset is ($) :
1400
with the certainity : 1

------  CONCLUSIONS MADE DURING THE CONSULTATION  ---
parameter / value / certainity / context instance
------------------------------------------------------
lessee_credit --- good --- 1--- lease--1
acquire_by --- purchase --- 1--- lease--1
asset_cost --- 3000 --- 1--- finance--1
finance_interest --- 12 --- 1--- finance--1
finance_period --- 2 --- 1--- finance--1
down_payment --- 500 --- 1--- finance--1
finance_it --- 1400 --- 1--- finance--1
payment --- 1400 --- 1--- lease--1



yes

| ?- begin.
Enter  the  name  of  the  root  context  (CAR,LEASE)
==>lease.
The following is a part of  a lease/acquire_by/finance
DSS
```

```
----lease-1----
Do  you  want  to  analyze  the financing for asset ?
==>y.
----finance-1----
Do you have any other finance to analyze ?   ==>y.
----finance-2----
Do you have any other finance to analyze ?   ==>n.
lease-1
How would you describe your current credit rating ?
(good/fair/poor)  ==>poor.
Your credit is not adequate.You cannot borrow money to
acquire_by the asset
Therefore LEASE the asset
My recommendation is lease the asset
finance-1
Do you want a straight lease ?  ==>yes.
finance-1
What is the asset cost  ==>4000.
finance-1
Percent charged by the leasing firm ?  ==>13.
finance-1
Lease period ?  ==>3.
finance-1
Do you  want a  lease with  the option  to terminate ?
==>no.
Is there any other lease problem you want to solve  ?
```

```
==>n.

Payment on the asset for the asset is ($) :

1853.3

with the certainity : 1


------  CONCLUSIONS MADE DURING THE CONSULTATION  ---

parameter / value / certainity / context instance

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

lessee_credit --- poor --- 1--- lease--1

cannot_borrow --- yes --- 1--- lease--1

how_to_acquire --- lease --- 1--- lease--1

acquire_by --- lease --- 1--- lease--1

straight_lease --- yes --- 1--- finance--1

asset_cost --- 4000 --- 1--- finance--1

finance_interest --- 13 --- 1--- finance--1

finance_period --- 3 --- 1--- finance--1

finance_it --- 1853.3 --- 1--- finance--1

option_lease --- no --- 1--- finance--1

payment --- 1853.3 --- 1--- lease--1


yes

| ?- halt.

[ Prolog execution halted ]
```

Consultation results of the Personal Consultant Plus expert system shell for the FINANCE analysis system [14].

The knowledge-base for these results are the same with the one in appendix.C.2. In parantheses are the corresponding terms of our FINANCE analysis system.

```
Consultation  record  for:   DEMO    :  LEASE  OR BUY
DECISION SYSTEM


your credit rating                           :: \POOR\
lease is to be a modifiable option 1...      :: \YES\
( straight lease ?)
ASSET-COST                                   :: \4000\
FINANCE-INTEREST                             :: \13\
FINANCE-PERIOD                               :: \3\
analyzing the financing                      :: \NO\
( any other lease problem ?)

ASSET-1 CONCLUSIONS
My recommandation is as follows: LEASE the asset
Payment for the (ASSET-1) is as follows:  $ 1853.3 per
year
```

APPENDIX E

FIGURES

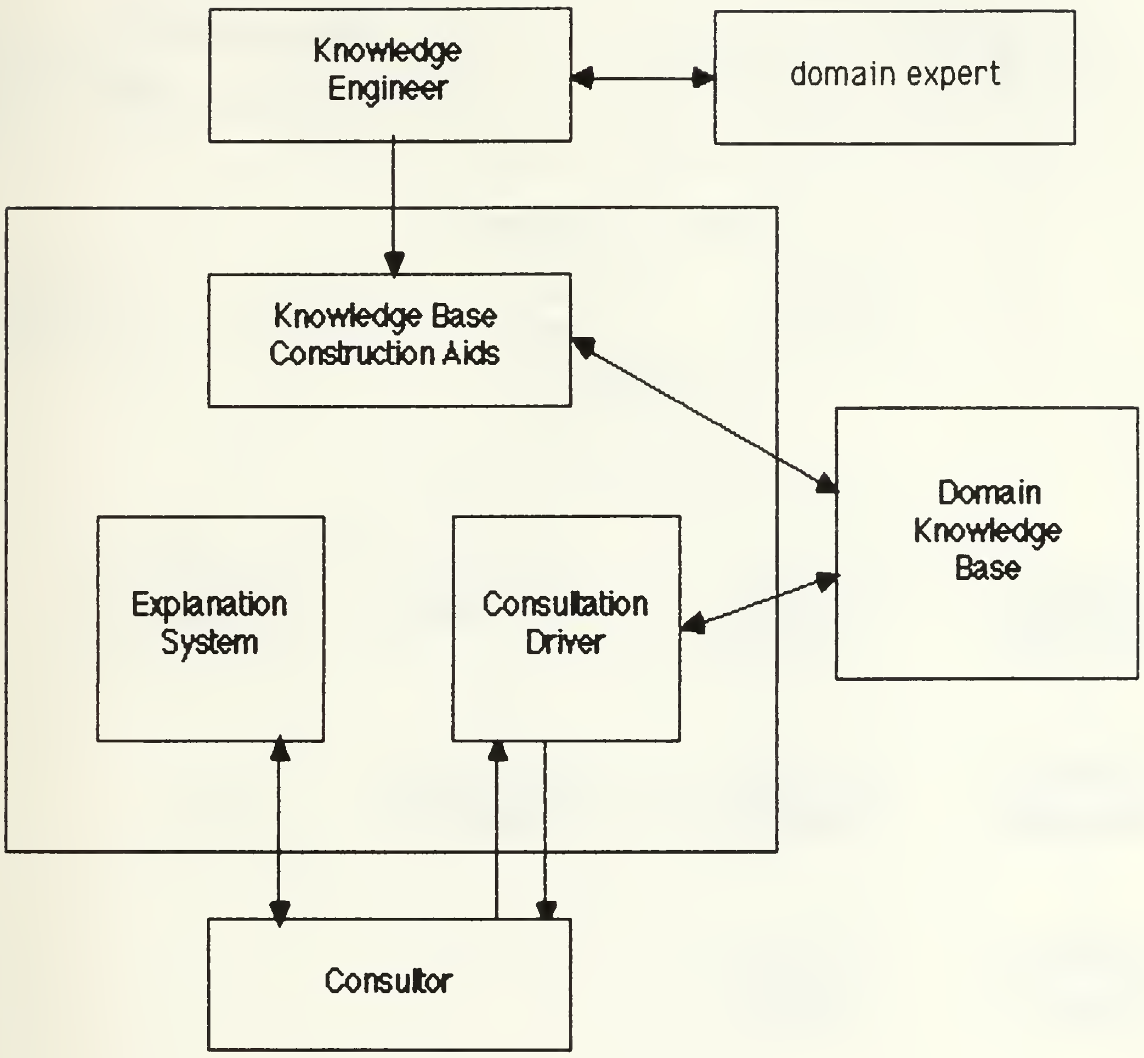

Figure 1 EMYCIN's Overall Organization

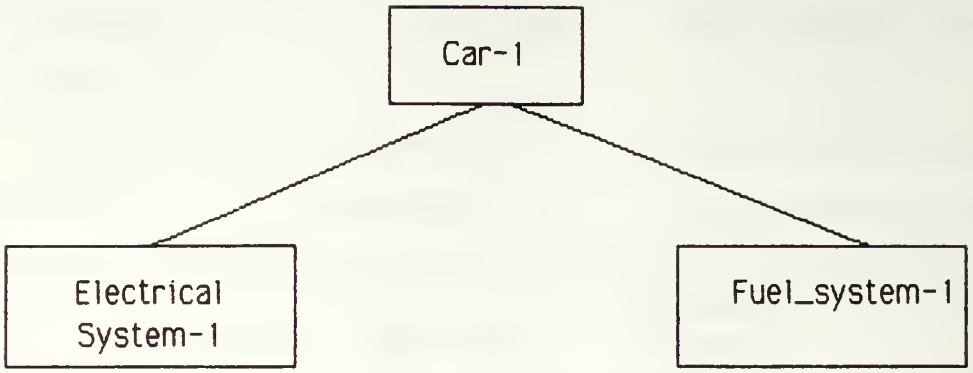

Figure 2 A Context Tree

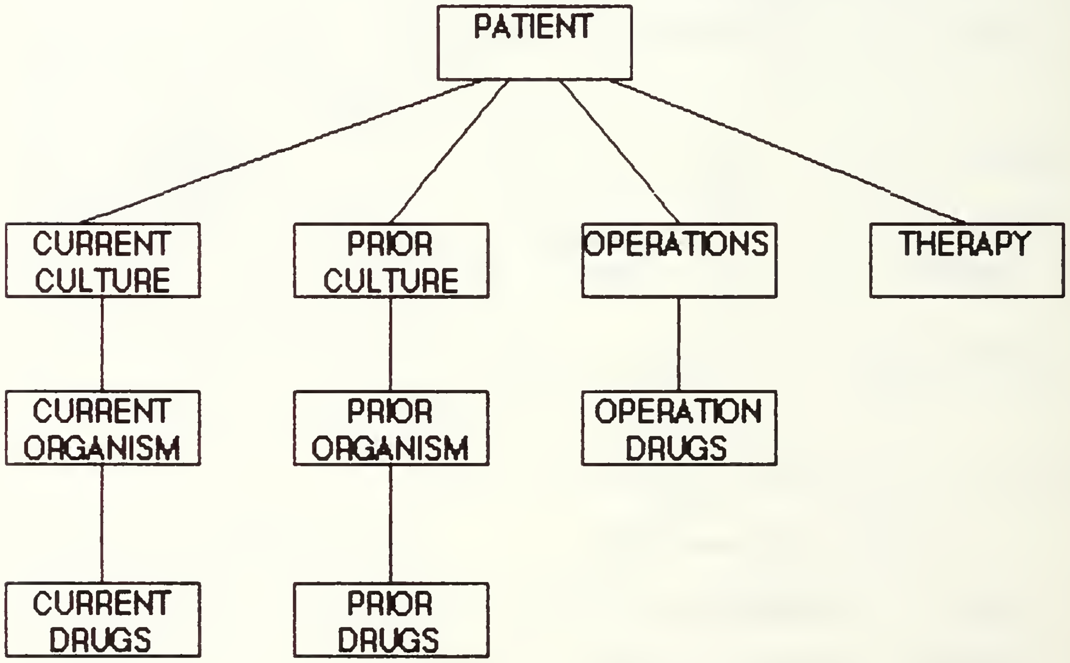

Figure 3 MYCIN's Static Tree Of Context Types

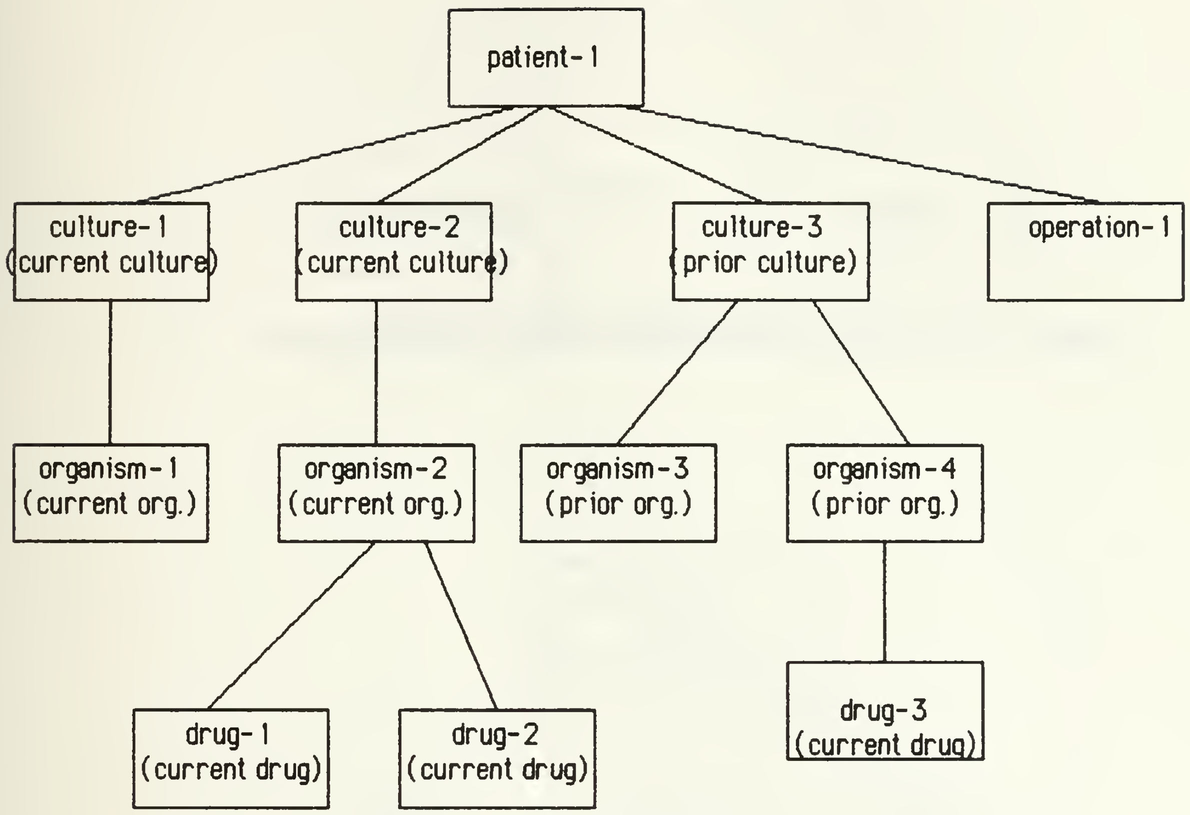
patient-1
culture-1 (current culture)
culture-2 (current culture)
culture-3 (prior culture)
operation-1
organism-1 (current org.)
organism-2 (current org.)
organism-3 (prior org.)
organism-4 (prior org.)
drug-1 (current drug)
drug-2 (current drug)
drug-3 (current drug)

Figure 4 A Context Instance Tree From MYCIN

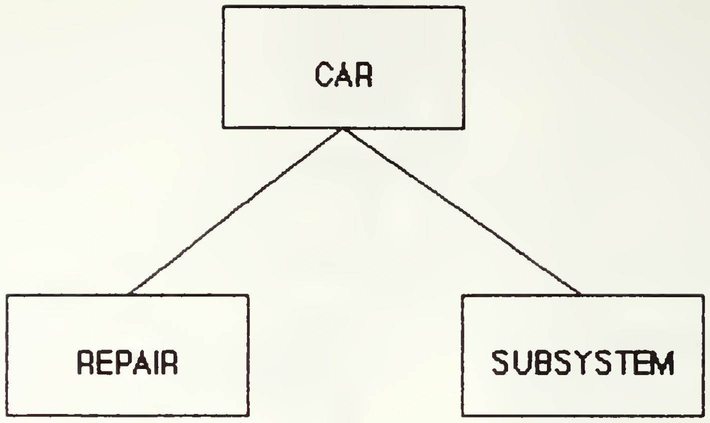

Figure. 5 Static Tree of Context Types of CAR Diagnose System

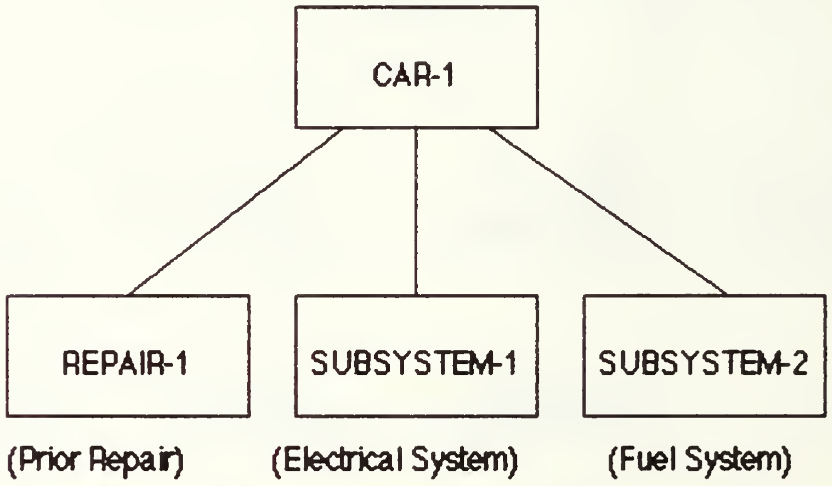

Figure 6 Dynamic Tree of CAR Diagnose System.
(Instance names are in parentheses.)

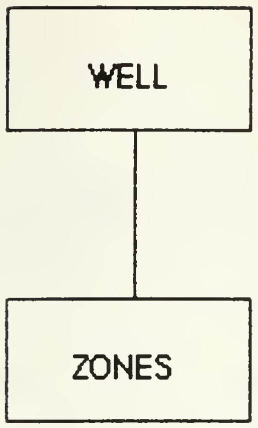

Figure. 7.a. Static Context Tree of LITHO.

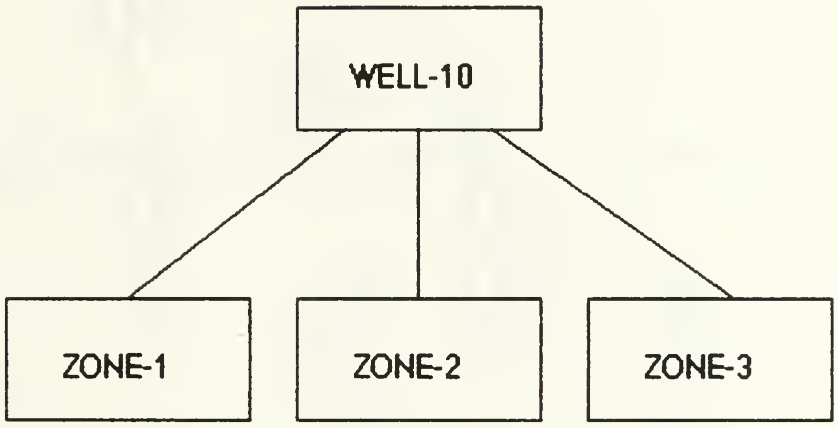

Figure 7.b. Dynamic Context Tree of LITHO.

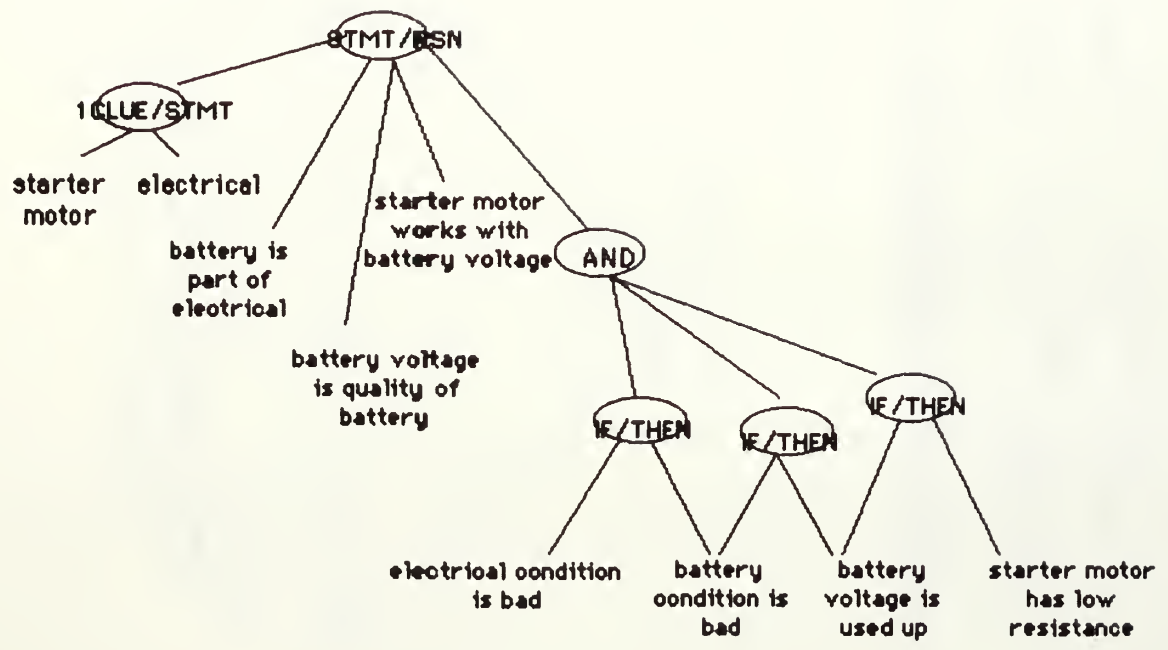
STMT/RSN
1CLUE/STMT
starter motor
electrical
battery is part of electrical
battery voltage is quality of battery
starter motor works with battery voltage
AND
IF/THEN
IF/THEN
IF/THEN
electrical condition is bad
battery condition is bad
battery voltage is used up
starter motor has low resistance

Figure 8 Explanation Tree Of Starter_motor/electrical

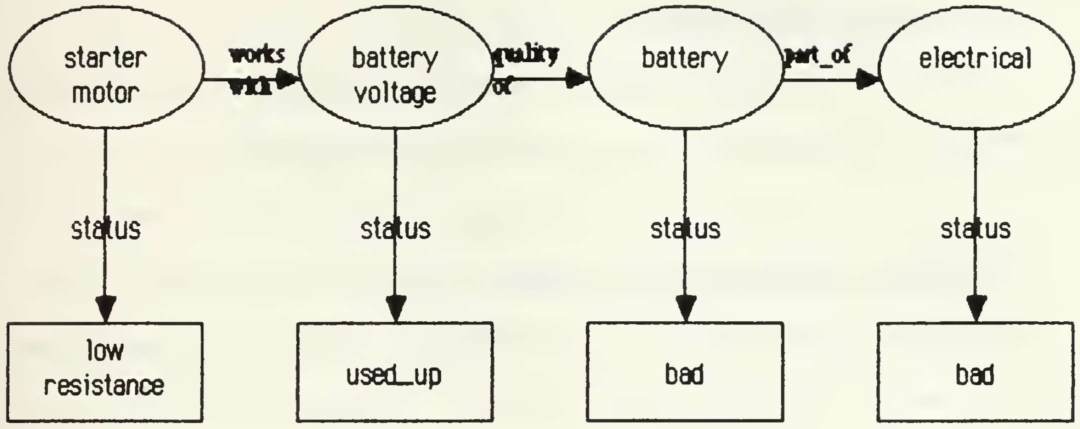

Figure 9 Semantic Network Of Starter_motor/electrical

ammeter/electrical

Electric current is produced by electrical system and ammeter measures the electric current.

dimming light/electrical

Electric current is produced by the electrical system and dimming_light-test measures the electric current.

hydrometer/battery

Hydrometer measures the specific gravity value of electrolyte. Electrolyte is part of the battery and if the specific gravity value is less than 1250 then battery will not function properly.

battery voltage/battery

Battery voltage is the measure of the battery performance and if its value is less than 12 V. then battery will not perform properly.

Figure 10 Natural Explanations

```
starter_motor(works_with,battery_voltage).
battery_voltage(quality_of,battery).
battery(part_of,electrical).
hydrometer(measures,specific_gravity).
specific_gravity(quality_of,electrolyte).
electrolyte(part_of,battery).

ammeter(measures,electric_current).
dimming_light(measure,electric_current).
electric_current(produced_by,electrical).
```

Figure 11.a. Relationship Facts Of The Inference Network

```
path(starter_motor,electrical,[battery_voltage,
battery]).
path(ammeter,electrical,[electric_current]).
path(dimming_light,electrical,[electric_current]).
path(hydrometer,battery,[specific_gravity,
electrolyte]).
path(battery_voltage,battery,[]).
```

Figure 11.b. Path Facts Of The Inference Network

```
battery(status,bad) :- battery_volt(value,< 12).

battery(status,bad) :- hydrometer(value,< 1250).

battery(status,bad) :-

battery_voltage(status,used_up).

battery_voltage(status,used_up) :-

starter_motor(status,low_resistance).

electrical(status,bad) :- battery(status,bad).
```

Figure 11.c. Rules Of The Inference Network

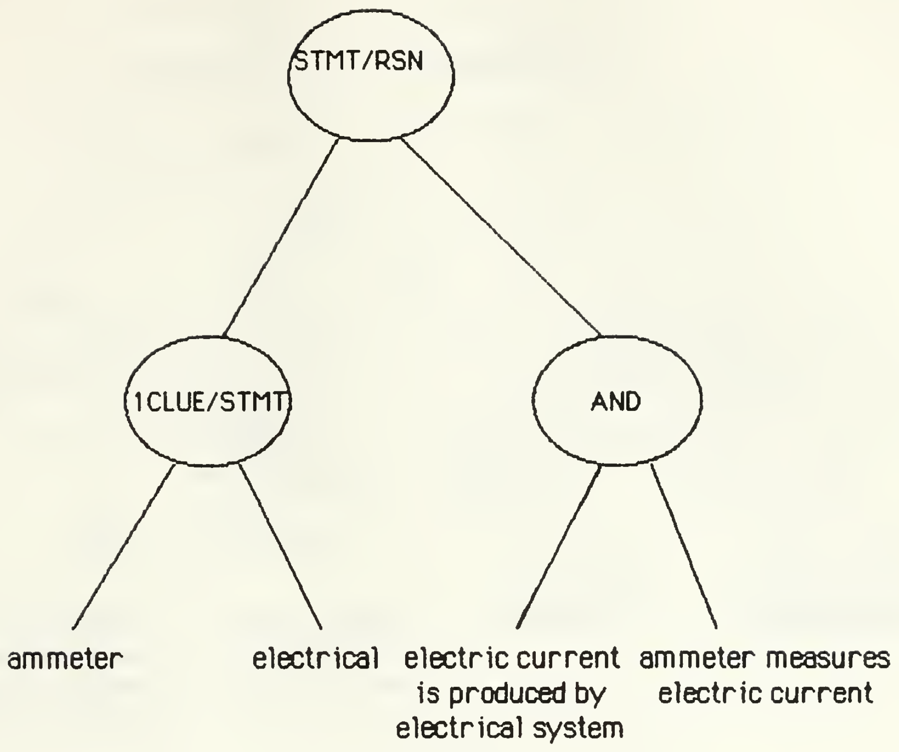
STMT/RSN
1CLUE/STMT
AND
ammeter
electrical
electric current is produced by electrical system
ammeter measures electric current

Figure 12.a. Explanation Tree Of "ammeter/electrical"

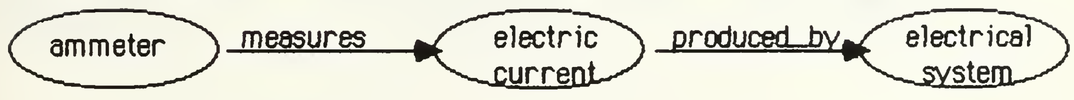

Figure 12.b. Semantic Network Of "ammeter/electrical"

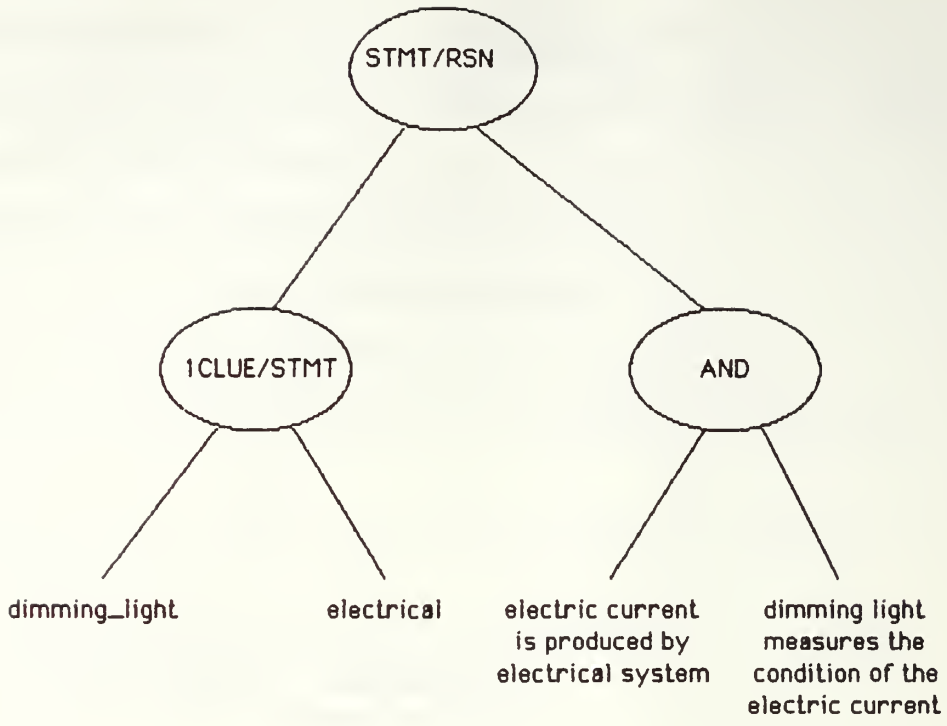
STMT/RSN
1CLUE/STMT
AND
dimming_light
electrical
electric current is produced by electrical system
dimming light measures the condition of the electric current

Figure 13.a. Explanation Tree Of "dimming_light/electrical"

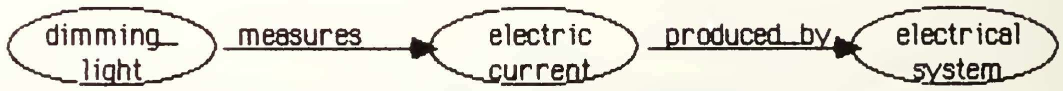

Figure 13.b. Semantic Network Of "dimming_light/electrical"

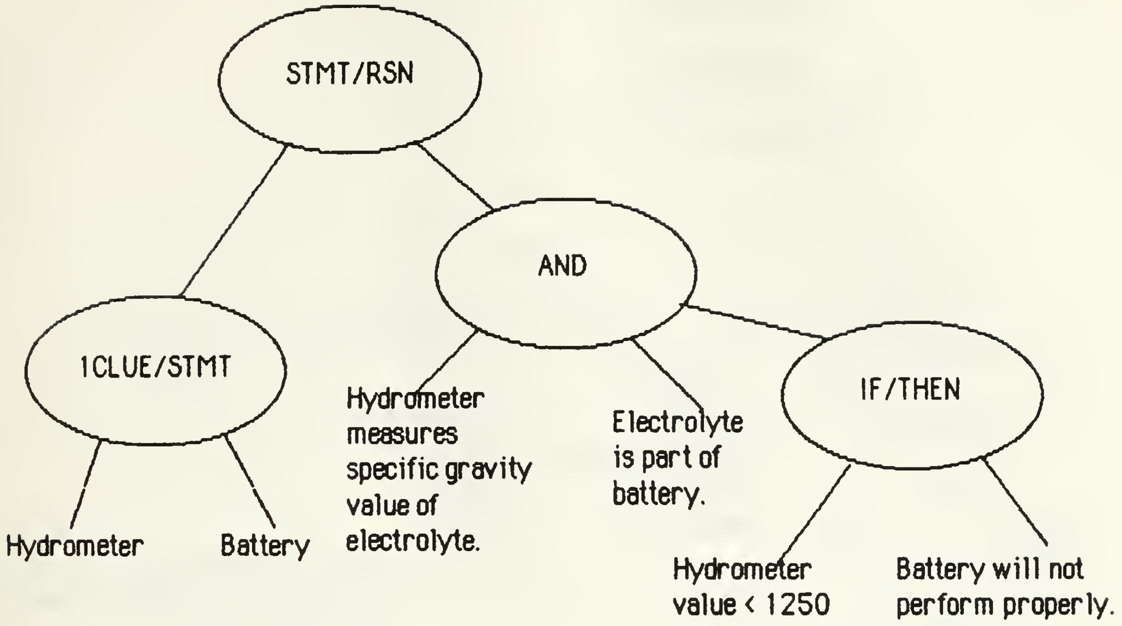
STMT/RSN
AND
1CLUE/STMT
IF/THEN
Hydrometer
Battery
Hydrometer measures specific gravity value of electrolyte.
Electrolyte is part of battery.
Hydrometer value < 1250
Battery will not perform properly.

Figure 14.a. Explanation Tree Of "hydrometer/battery"

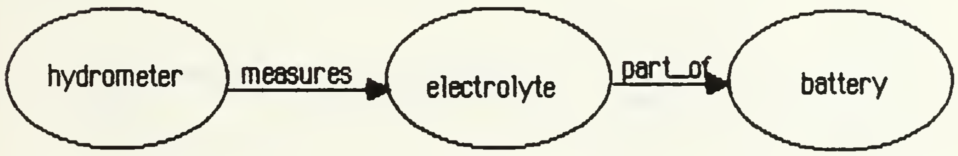

Figure 14.b. Semantic Network Of "hydrometer/battery"

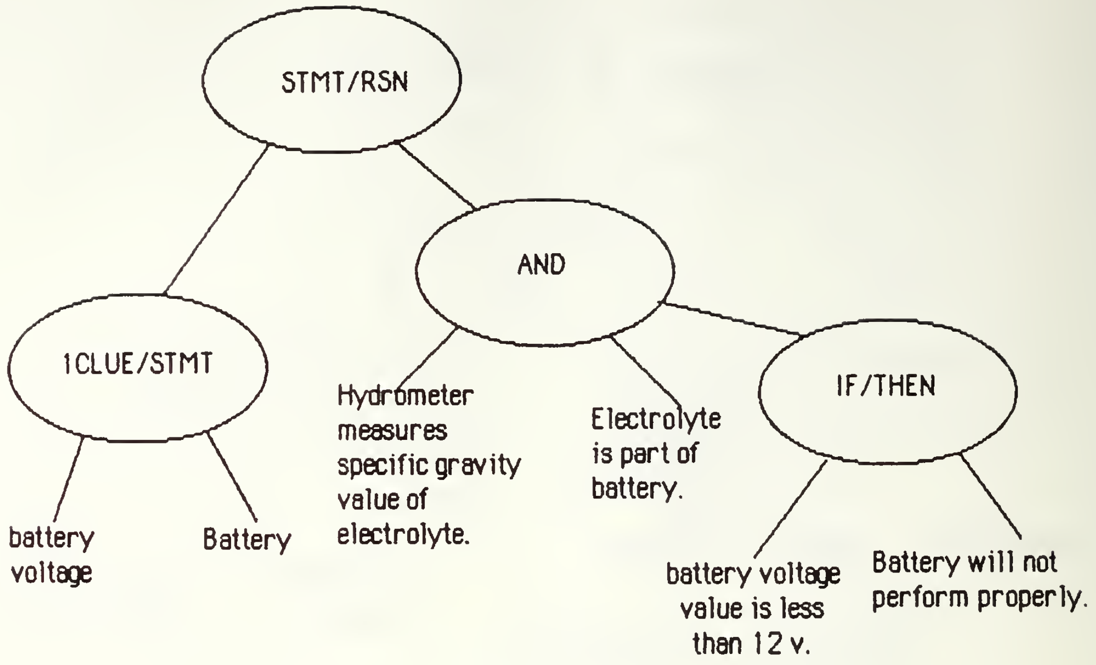
STMT/RSN
1CLUE/STMT
AND
IF/THEN
battery voltage
Battery
Hydrometer measures specific gravity value of electrolyte.
Electrolyte is part of battery.
battery voltage value is less than 12 v.
Battery will not perform properly.

Figure 15.a. Explanation Tree Of "battery_voltage/battery"

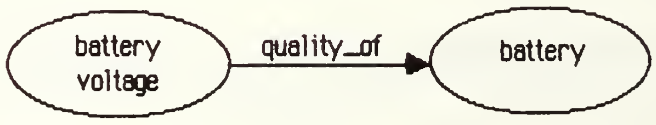

Figure 15.b. Semantic Network Of "battery_voltage/battery"

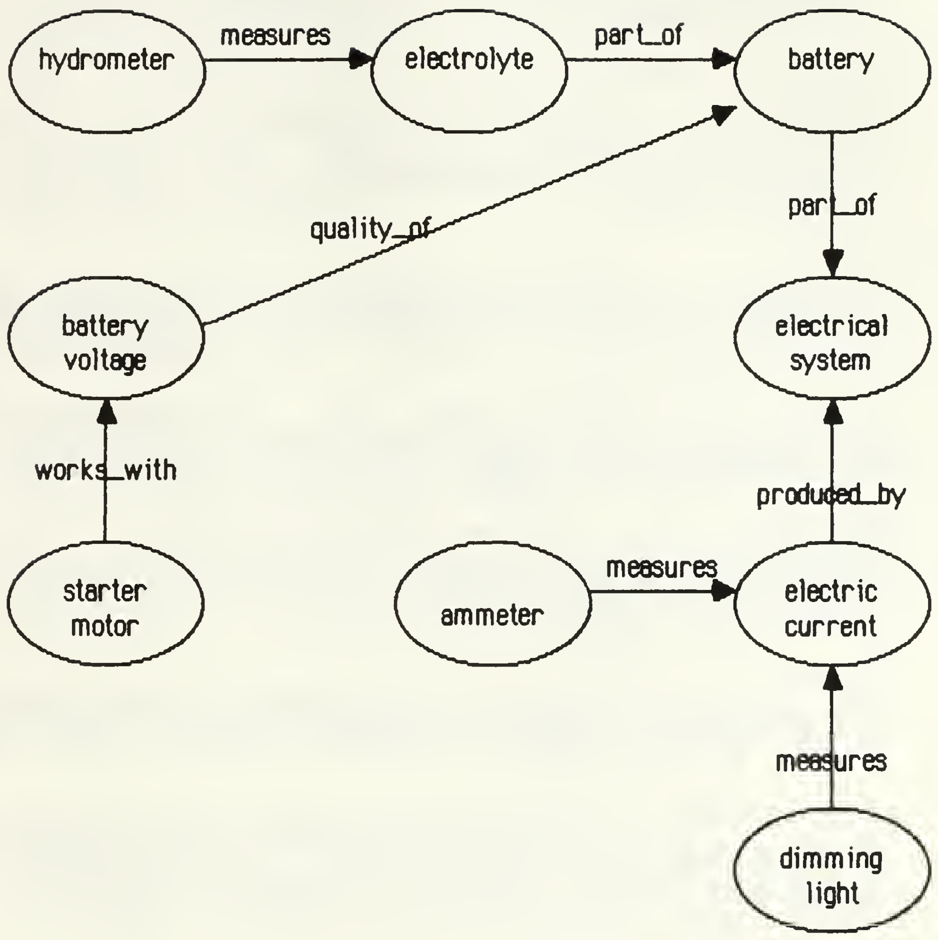

Figure 16. Semantic Network

LIST OF REFERENCES

1. Barr, A., Feigenbaum, E.A. The Handbook Of Artificial Intelligence. Vol.II. William Kaufmann, Inc. 1981.

2. Buchanan, B.G., Shortliffe, E.H. Rule-Based Expert Systems, The MYCIN Experiment Of The Stanford Heuristic Programming Project. Addison-Wesley. 1984.

3. Narain, S. MYCIN: The Expert System And Its Implementation In LOGLISP. Tech. Report, C.I.S., Syracuse University, Syracuse, N.Y. August 1981.

4. Reggia, J. Knowledge Based Decision Support Systems. Ph.D. Dissertation, University Of Maryland. 1981.

5. Barr, A., Feigenbaum, E.A. The Handbook Of Artificial Intelligence. Vol.I. William Kaufmann, Inc. 1981.

6. Mulsant, B., Servan-Schreiber, D. Knowledge Engineering: A Daily Activity On A Hospital Ward. Computers And Biomedical Research, (1984) 17, 71- 91.

7. Harmon, P., King, D. Artificial Intelligence In Business EXPERT SYSTEMS. John Wiley And Sons, Inc. 1985.

8. Van Melle, W. EMYCIN: A Domain Independent Production Rule System For Consultation Programs. Ph.D. Thesis, Computer Science Department, Stanford University. 1980.

9. Davis, R., Buchanan, B.G. Shortliffe, E.H. Production Rules As A Representation For A Knowledge-based Consultation Program. Artificial Intelligence, (February 1977) 8, No. 1

10. Sterling, L., Shapiro, E. The Art Of Prolog. The MIT Press. 1986.

11. Clocksin, W.F. Mellish, C.S. Programming In Prolog. Springer-Verlag. 1984.

12. Robinson, J.A. A Machine-Oriented Logic Based On The Resolution Principle. Journal Of The Association For Computing Machinery, (1965) 12, 23-41.

13. Wong, W.G. PROLOG A Language For Artificial Intelligence. PC Magazine, (October 1986) 14.

14. Texas Instruments. Personal Consultant Plus User's Guide.

15. Goguen, J.A., Weiner, J.L., Linde, C. Reasoning And Natural Explanation. Int.J.Man-Machine Studies, (1983) 19, 521-559.

16. Micro-AI. Prolog-86 User's Guide And Reference Manual.

17. Hirsch, J.D. The Complete Book Of Car Maintenance And Repair. Charles Scribner's Sons. 1973.

18. Rowe, C.N. Introduction To Artificial Intelligence Through PROLOG. Prentice Hall Englewood Cliffs N.J. 1987.

BIBLIOGRAPHY

Cendrowska, J., Bramer, M.A. A Rational Reconstraction Of The MYCIN Consultation System. Int. J. Man-Machine Studies, (1984) 2Ø, 229-317.

Chandrasekaran, C. Generic Tasks In Knowledge-Based Reasoning:High-Level Building Blocks For Expert System Design. IEEE Expert, (Fall 1986).

Chandrasekaran, C. Towards A Taxonomy Of Problem Solving Types. The AI Magazine, (Winter/Spring 1983).

Hayes-Roth, F., Waterman, A.D., Lenat, D.B. Building Expert Systems. Addison Wesley. 1983.

Keonrad, L., Parker, D.S. Control Over Inexact Reasoning. AI Expert Premier, (1986) 32-43.

Raul, E.V. Inside An Expert Systems Shell. AI Expert, (October 1986) 3Ø-42.

Waterman, D.A. A Guide To Expert Systems. Addison Wesley. 1986.

William, J.C. The Epistemology Of A Rule - Based Expert System - A Framework For Explanation. Artificial Intelligence, (1983) 2Ø, 215-251.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Department Chairman, Code 52 Department Of Computer Science Naval Postgraduate School Monterey, California 93943-5000	1
4.	Associate Professor N.C. Rowe Code 52Rp Department Of Computer Science Naval Postgraduate School Monterey, California 93943-5000	1
5.	Associate Professor B.J. MacLennan Code 52Ml Department Of Computer Science Naval Postgraduate School Monterey, California 93943-5000	1
6.	Associate Professor T.R Sivasankaran Code 54SJ Department Of Administrative Sciences Naval Postgraduate School Monterey, California 93943-5000	1
7.	Yucel Ozin K.K.K'ligi Muh. Elk. ve Bilgi Sis. D. Bsk'ligi Bilgi Sistem Destek Subesi Bakanliklar-ANKARA TURKEY	1
8.	LTJG Fikret Ulug, Turkish NAVY Define Sok. 4/2 Aydinlikevler-ANKARA TURKEY	2

9. Deniz Harp Okulu Kitapligi 1
Deniz Harp Okulu Komutanligi
Tuzla-ISTANBUL
TURKEY