THE McESE PROJECT

F. Franek and I. Bruha

Technical Report no TR-21/88

Department of Computer Science and Systems

McMaster University 1988

THE McESE PROJECT

F. Franek 1) and I. Bruha 2)

Dept. of Comp. Sci. & Systems
McMaster University
Hamilton, Ontario
L8S 4K1 Canada

TABLE OF CONTENTS:

Acknowledgement:

1) Research supported by SERB 5-26397 and NSERC OGP0025112 research grants.
2) Research supported by NSERC A20037 research grant.

ABSTRACT

McESE is an expert system environment (a software tool) designed to help create problem-specific shells with incomplete and uncertain knowledge, fast and compact expert system applications in a particular programming language. Specialized software of McESE is written in C and facilitates handling of all aspects of dealing with rule-based knowledge bases. Practical and theoretical aspects of McESE are discussed.

(1) INTRODUCTION.

McESE (McMaster Expert System Environment) is a software tool to build problem-specific shells and create expert system applications. It is designed to satisfy the goals listed below (not in the order of their significance):

(1.1) allow the user to deal with imprecise and incomplete knowledge in McESE knowledge bases with a declarative formalism that has a satisfactory degree of expressive power;

(1.2) allow the user to customize the shell as so it handles uncertainty in the way of his preference;

(1.3) allow the user to create expert system applications in a particular programming language (C, FranzLISP, and SCHEME are available at the moment), with a point of reference being the

application rather than the knowledge base (so the creation of such an application resembles ordinary programming as much as possible);

(1.4) allow the user a natural (hierarchical) connection of different knowledge bases in an application;

(1.5) allow rapid prototyping;

(1.6) allow fast inferring.

(2) HOW THE GOALS OF McESE PROJECT ARE ATTAINED.

(2.1) In McESE the user can encode the domain knowledge in rules of the following form:

TERM1 & TERM2 & ... & TERMn ==CVPF==> TERM

(cvpf abbreviate "certainty value propagation function".) The "meaning" of a simple rule "TERM1 & TERM2 ==F==> TERM3" is: if we are certain with value v1 that TERM1 is true, and if we are certain with value v2 that TERM2 is true, then we are certain with value F(v1,v2) that the left hand side (LHS for short) holds, and so we are certain with that value that TERM3 holds.

An (meaningless) example of a McESE rule:

R1: .8*P1(x,y)[>=.3] & -P2(z) ==F2==> P3(x,y,z)[<.5]

where R1 is the rule's id, P1, P2, and P3 are predicates, F2 is a cvpf, x,y, and z are predicate variables, "-" stands for negation, .8 preceding P1 is the weight of the first term (must be a real number between 0 and 1 inclusive; if omitted, it is assumed to be 1), [>=.3], [<.5] are threshold directives (>= and > in [ ] are threshold operators, and .3 and .5 in [ ] are threshold values, must be real values between 0 and 1 inclusive).

A predicate, possibly preceded by a weight, possibly preceded by "-" or "~" (denoting negation), and possibly followed by a threshold directive, is called a term.

The "firing" of the above mentioned rule consists of: first, for the rule to be "fired", all predicate variables in the rule must be bound to some data structures, called objects. Let x be bound to the object X, let y be bound to the object Y and let z be bound to the object Z. Second, the certainty values (real values between 0 and 1 inclusive) of all LHS terms must be known. Then the certainty value of the right hand side (RHS for short) predicate can be computed as: Let v1 be the value of the first term of the LHS of the rule R1 (i.e. the term .8*P1(X,Y)[>=.3]), let v2 be the value of the second LHS term of the rule R1 (i.e. the term ~P2(Z)). Then the value of the LHS is F2(v1,v2). (F2 must be a function of two real arguments returning a real value between 0 and 1 inclusive, or -1.) From this the value of the RHS predicate P3(X,Y,Z) is determined by the threshold directive. In this case, if the value of the LHS is strictly less than .5, the value of P3(X,Y,Z) will be set to 1, otherwise it will be set to 0.

The value of a LHS term is computed from the value of the term's predicate according to the weight and the

threshold directive. E.g. the value of P1(X,Y)[>=.3] will be 1 if the value of P1(X,Y) is greater or equal to .3, otherwise it will be 0. If the weight is not specified, it is assumed to be 1, and so the final value of the term is completed at this point. On the other hand when the weight is specified, as in this case, the final value of the term is obtained by multiplying by the weight.

The value of ~P2(Z) will be (1-value of P2(Z)).

If the cvpf F2 returns -1, then the rule is considered not "fired".

Rules in this form allow to capture an imprecise, uncertain and incomplete knowledge, since the rules are guaranteed to "fire" for any values of LHS terms (except some situations when the firing is prevented by the cvpf), and only the resulting value of the RHS predicate is affected by the values of LHS terms. Thus, we can formulate our rules in vague terms, as in this example from an expert system to play a card game Canasta:

"opponent_collect(x) & used_stck_high =F=> ~discard(x)" where we can never be sure if the opponent really collects x, and when the used stack is high. But we can build into the knowledge base enough information to estimate these facts numerically (based on current input data) and these numbers project via cvpf F into the value of discard(x). Even in the case of complete lack of information, say if the value of opponent_collect(x) is 0 we may want to associate the value of .25 with discard(x) (since there are 4 possible types the opponent may be collecting and so in the absence of any relevant information a good guess is that there is .25 chance of the opponent collecting x) and that's what cvpf F can do.

The predicates serve as simple statements about entities the knowledge is "talking" about. For example "ontop(X,Y)" is meant to indicate that the object X is on top of the object Y, or "allyounger(10)" is meant to indicate that all involved were younger than 10. How well they really reflex the reality is a different matter.

(2.2) If no cvp function in a rule is stipulated, the default one is used. If unchanged by the user, it is so-called weighted cumulative evidence computed according the following formula: let w = w1 + w2 + w3 + .. + wn, where w1 is the weight of term1, w2 the weight of term2, ... , wn is the of termn. Let v1 be the value of LHS TERM1, v2 the value of LHS TERM2,..., vn the value of LHS TERMn. Then (v1 + v2 + ... + vn)/w is the value of the LHS.

As any cvpf can be defined as the default choice, one can pre-determine that all rules in the knowledge base will be handled uniformly, in essence fixing a particular method of the treatment of uncertainty in the whole knowledge base.

(2.3) Most of expert systems shells are either presented with the knowledge representation language as the main language of the application, and hence the application is "centered" around the "model" (knowledge base), and the procedural parts are connected to it by different means (in the case of OPS languages and PROLOG it is the only language), or they themselves are written in the language of application (for example KEE in LISP). We tried to give the user a possibility to write an application in the usual way, at least the

procedural parts, and in the programing language of his choice, but still preserve the possibility of having access to a declarative knowledge base when needed. This is achieved by "extending" a particular programming language by McESE commands to facilitate all required communication between the application and the knowledge bases. The software to performer the communication is written in C, but is transparent to the user. Thus, a particular application is completely built using a single programming language and the language of McESE rules. At this point, McESE extensions of C, FranzLISP, and SCHEME are available. Note that this shift in emphasis changes the focal point from knowledge base to the application in an effort to allow for ordinary programming techniques, methods, and experience to be utilized.

(2.4) Predicates which never occur on RHS of any rule correspond to facts and observations; we shall call them level 0 predicates for they will be on level 0 of the knowledge tree (see 4.2). They represent data input nodes of the knowledge tree. Their values are not derived (inferred) using rules, they must be obtained from so-called predicate service procedures. These may be ordinary procedures to supply the facts and/or observations, or they may in fact be other expert systems. This mechanism allows for convenient partitioning of the domain knowledge into a hierarchy of knowledge bases (or more precisely expert systems), see Fig. 1.

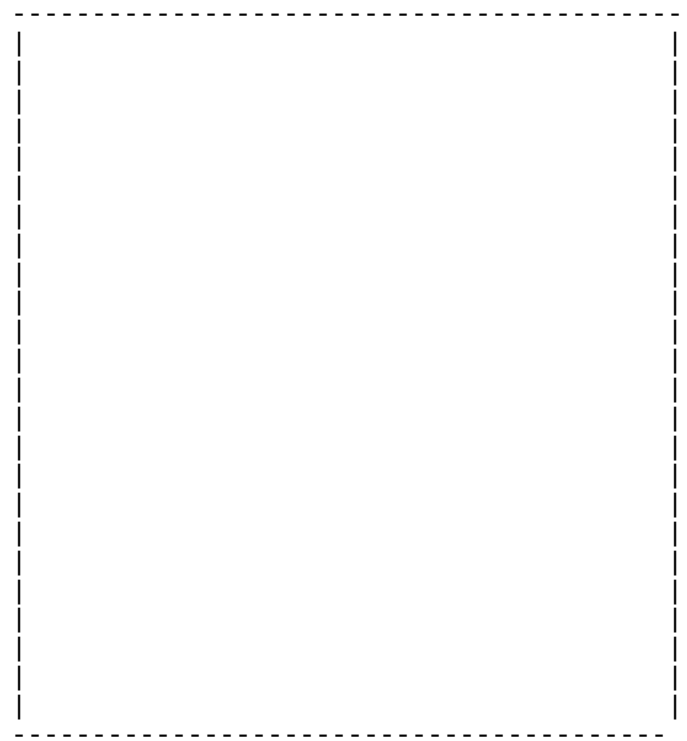


Fig. 1

On the other hand, predicates on higher levels represent conclusions based on facts and/or other conclusions. Their value must be obtained by inferring.

(2.5) Since McESE built-in inference engine automatically prompts the user for the result of the invocation of a predicate service procedure in the case that the predicate service

procedure is not available to the system (and similarly for cvpf's), one can just test and modify rules in the knowledge base without the overhead of building the complete application. Moreover since McESE interactions and inferences are identical in McESE-C, McESE-FranzLISP, and McESE-SCHEME, one can quickly build a prototype in McESE-FranzLISP (utilizing versatility and flexibility of FranzLISP) to verify the methods and approaches, and when satisfied, the knowledge bases can be used as they are for the McESE-C application.

(2.6) McESE knowledge bases are first compiled before they can be used in an application. The compiled knowledge base (for short called knowledge tree) allows for direct linking of relevant predicates, so only relevant rules are in fact considered when a predicate must be evaluated. Thus inferring with such a knowledge tree amounts to a "walk" through the tree, and hence the speed of inferring depends entirely on the depth of the knowledge tree rather than on its size. The result is a fast performance, knowledge base queries are quickly evaluated and returned to the application program.

(3) SYNTACTICAL DESCRIPTION OF RULES IN RSET.

List of all tokens and there definitions follows. As usual, a space represents any number (none included) of so-called white spaces (blank, carriage return, newline, and comment). { } indicate an optional part.

comment <comment> is any text enclosed between comment brackets - left bracket /* - and - right bracket */ - it is treated as a white space McESE compiler, and hence not part of compiled RSET.

rule <rule> is a block <ruleid> <lhs> <arrow> <rhs>

rule identifier <ruleid> is a block <rulename> { <tdir> } :

rulename <rulename> is a block of characters beginning with a letter, can contain lower and upper case letters, digits, or '_'

threshold directive <tdir> is a block [<top> <tval>]

threshold operator <top> is a block of 1 to 2 characters, either '>' or '>='

threshold value <tval> is a decimal constant value between 0 and 1 inclusive

left hand side <lhs> is a block <term> & <term> ... & <term>

term <term> is a block { <weight> * } { <sin> } <pred> { <tdir> }

weight <weight> is a decimal constant value between 0 and 1 inclusive

sign <sign> is a block of one character, either '-', or '~'

predicate <pred> is a block <predname> { (<varlist>) }

predicate name <predname> is a block of characters beginning with a letter, can contain lower and upper case letters, digits, or '_'

list of variables <varlist> is a block <varname> , ... ,

<varname>

variable name <varname> is a block of characters beginning with a letter, can contain lower and upper case letters, digits, or '_'

arrow <arrow> is a block { <larrow> <cvpf>} <rarrow>

left arrow <larrow> is a block of one or more '='

right arrow <rarrow> is a block of one or more '=' followed by '>'

certainty value propagation function <cvpf> is a block of characters beginning with a letter, can contain lower and upper case letters, digits, or '_'

right hand side <rhs> is a block { <sign> } <pred> { <tdir> }

(4) McESE COMPONENTS.

(4.1) McESE source knowledge base:

McESE source knowledge base consists of two separate sets: the set (RSET) of McESE rules (in descriptive form), and the set (FSET) of corresponding cvpf's (in procedural form). In addition to the above mentioned syntax of McESE rules, each rule has to satisfy the condition that all predicate variables occurring in predicates of the LHS, must be variables of the RHS predicate, and vice versa, all RHS predicate variables must occur as predicate variables of some predicate of the LHS. There also are restrictions on the whole RSET, namely that

(a) There should be no subset of rules forming a close cycle, e.g. subset like this: A & B => C, C & D => E, E & F => A, since the inferring might be going on in this cycle without getting anywhere. Besides, cycles like this are cousins of tautologies like "A implies A", which cannot carry any relevant information and thus are useless as a knowledge representation vehicle; and

(b) The same predicate can occur in more than one rule as the RHS or the LHS predicate. But each occurrence of the predicate must have the same number of variables (though the variables can have different names).

Why no rules with disjunctive LHS were considered?

Disjunction on the LHS of rules can be easily emulated by a simple set of McESE rules ("A v B => C" can be emulated by a set of two McESE rules, "A => C" and "B => C") with no loss of the processing speed and/or expressiveness of the formalism. On the other hand, a McESE set of rules with the same RHS predicate should be viewed as a case of a disjunction. The price to pay for this approach is the need for rule resolution with all implications it has for the certainty value of the predicate being evaluated.

The main reason to use rules with purely conjunctive LHS was psychological. We have found that people prefer to list eventualities one by one, rather than in a disjunction. It seems more natural and easier to comprehend. Thus, we have decided to pay the price (rule resolution in the form of inferring modes) and accommodate our human user preferences.

FSET is a set of programs in the language of the chosen extension, the only requirement being that they had as many arguments as the rule they are referred to in has LHS terms, and that their arguments were real values between 0 and 1 inclusive,

and that they returned real values between 0 and 1 inclusive, or -1. From the logical point of view, they must be non-decreasing, i.e. when a certainty value of a term increases, the certainty for the whole LHS (as represented by the cvpf) must not decrease.

Clearly, the cvpf is an important part of the heuristics captured in a rule. Thus, we cannot claim that the domain knowledge stored in a knowledge base is purely declarative. The rules themselves are declarative, but the cvpf's are procedural, even though their purpose is not really procedural.

Although RSET and FSET are maintained in separate files, McESE built-in editor allows the user to edit both parts together in two windows on the screen. This simplifies the task of knowledge and software engineering with McESE.

(4.2) McESE compiled knowledge base:

The McESE source knowledge base as a set of rules (RSET) and a set of cvpf's (FSET) is a structure well suited to store the desired knowledge in its declarative and procedural form respectively, suitable for humans to understand, modify and manipulate easily. But for many reasons it is not a well suited structure for a computer program to access and "infer" with it. Thus, McESE first "compiles" the knowledge base into a data structure which may be visualized as a tree capturing the essential relations between predicates. Let us illustrate this on a small example.

Assume that RSET contains four hypothetical rules (for simplicity with no "meaning"):

```
R1: P1(x,y)[>=.3] & ~P2(x,z)[<.1]&P3(y)=F1=>P6(x,y,z)[>=.7]
R2: P2(x,y)&~P4(y)=F2=>~P7(x,y)
R3:~P7(x,y)&P5(z)[<=.67]=F3=>P8(x,y,z)[>=.2]
R4:P8(a,b,c)&P9(b)=F4=>P6(a,b,c)
```

The compiled version of it can be pictured as the following knowledge tree (see Fig. 2):

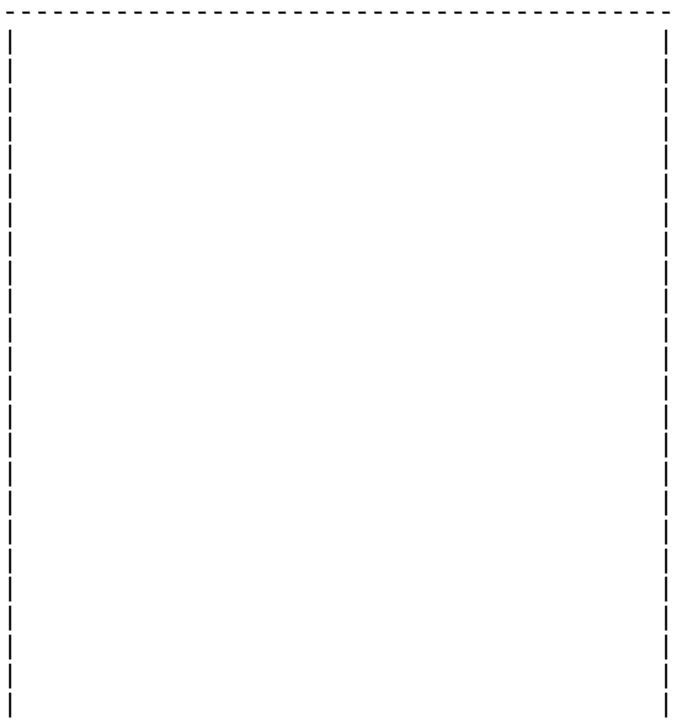


Fig. 2

The predicates are sorted into levels; the level 0 predicates are those which occur only in LHS terms of some rules (as P1,P2,P3,P4,P5,P9). Predicates which occur in RHS terms of some rules are assigned their levels according to this formula: find the LHS predicate with the highest level in the rule the predicate in question is in the RHS term. Than the level of the RHS predicate will be one higher. If the predicate occurs in more rules as the RHS predicate, its level is the maximal one assigned to according to each individual rule. Thus P7 has level 1 as all LHS predicates of rule R2 have level 0. On the other hand, P8 has level 2 as the LHS predicate of rule R3 with the highest level is P7, thus level of P8 is (level of P7 + 1), i.e. 2. From rule R1 follows that level of P6 is 1, since all LHS predicates of R1 have level 0. On the other hand, from rule R4 follows that level of P6 is 3, as P8 as the LHS with the highest level, i.e. 2. Therefore, level of P6 is 3.

Since a McESE source knowledge base must be cycle-free, all predicates can be sorted into levels (as with any acyclic undirected graph). In fact, the procedure for sorting the predicates into levels is used by the compiler to check for possible cycles. See the appendix for complete description of components of knowledge tree.

McESE compiler parses source RSET providing syntactical checking of rules, providing as well other checking as described above, and builds the knowledge tree in main memory with address links relative to the beginning of the knowledge tree. After successful compilation the resulting data structure is recorded in a disk file.

When a knowledge base is open in an application, McESE loader loads from disk into main memory the knowledge tree while absolutizing address links. McESE-FranzLISP and McESE-SCHEME can also load corresponding FSET dynamically. McESE-C does not have an on-line dynamic loader for compiled C code available yet, so for the time being all FSET's required by an application must be linked to it during compilation.

(4.3) McESE inference engine and inferring:

McESE inference engine provides the mechanism for "inferring". It can work in two basic modes, forward chaining and backward chaining.

Backward chaining from a given predicate (node) with given bindings for its variables is performed as depth-first walk down to level 0 nodes (with simultaneous propagation of bindings for predicate variables), "selecting" the appropriate subtree leading to the node being evaluated (there may be more than one such subtree). Only the required 0 level nodes are activated (and so appropriate known facts are fetched and/or appropriate observations are made) and then the resulting certainty values are propagated (and recorded in the knowledge tree, too) back through the selected subtree to the required node. The backward chaining mode has four submodes which amount to rule conflict resolution: max mode, sufficient max mode, min mode, and sufficient min mode. Sufficient max (min) mode searches for a subtree which evaluates the required node to a certainty value bigger (less) or equal to the specified value, then the chaining stops and this value is returned, or if

such a subtree is not found, "failure" is returned. The max (min) mode evaluates all possible subtrees and returns the max, i.e. the highest value (the min, i.e. the lowest value respectively).

Forward chaining is implemented only from 0 level up, to a specified level. Specified nodes from level 0 and their ascendants up to the specified level are evaluated and their values recorded in the knowledge tree.

The inference engine can work in two modes as far as explaining what it is doing: the silent mode when all inferring is transparent to the user and only the resulting value is available, or in trace mode when all inferring is done on the screen, rule by rule, predicate by predicate, with all relevant information being displayed, too. The trace mode is useful mainly for testing and debugging of knowledge bases.

A run time consistency checking takes place: only the minimal and maximal values for a predicate are recorded. The difference between these two is the inconsistency level. McESE allows to preset for the knowledge base what inconsistency level can be tolerated. If the inconsistency level tolerance is exceeded, ALARM is issued and the inferred value is returned to the application.

Also, a run time completness checking takes place: in the case a predicate cannot be evaluated, ALARM is also issued and -1 is returned by the inference engine.

For each McESE knowledge base the user can specify ALARM procedure which is automatically invoked by McESE when ALARM is issued. In this procedure the user can define what should be done.

An invocation of the inference engine from the application constitutes one inference cycle. Any of the values recorded in the knowledge tree as results of backward or forward chaining within the last inference cycle can be fetched to the application.

Let us illustrate some inferring with the small example of a knowledge base from 4.2 (see Fig. 2):

Let us ask for the "max" evaluation of P6(X,Y,Z). First the variables of P6 are bound to X, Y, and Z. Then these bindings are passed to R4. From there the variable of P9 is bound to Y and the variables of P8 are bound to X, Y, and Z. From P8 the bindings are passed to R3. From there the variables of P7 are bound to X, and Y, and the variable of P5 is bound to Z. From P7 the bindings are passed to R2 and from there the variable of P2 is bound to X and the variable of P4 is bound to Y. By now the proper subtree for evaluation has been selected and the bindings were passed to the predicates of level 0 (P9, P2, P4, and P5). Now these (observable) facts are evaluated (by the predicate service procedures) and the numbers (i.e. certainty values) are returned (for example 0.9, 0.8, 0.6, 0.6 respectively). P7 is evaluated to c7 = 1 - F2(0.8,1-0.6) = 1 - F2(0.8,0.4), say c7 = 0.3 . In order to evaluate P8, first all left hand side terms of R3 are evaluated, i.e. 1 -0.3, 0.6 [<= 0.67], i.e. 0.7 and 1. Hence the value of the right hand side term is F3(0.7,1), say 0.12. Thus c8 = 0.12 [>= 0.2] = 0. Now P6 is evaluated via R4 to c6 = F4(0,0.9), say 0.75. That was the first evaluation (via the subtree determined by the rules R4, R3, and R2). But P6 can be evaluated via the the subtree determined by the rule R1 as well. Thus, the bindings of the variables of P6 are passed to R1, from there the variables of P1 are bound to X, and Y, and the variables of P2 are bound to X, and Z, and the variable of P3 is bound to Y. Now these (observable) facts (P1, P2, and P3)

are evaluated (by the predicate service procedures), and their certainty values (for example 0.3, 0.8, and 0.5) returned. (Note, that P2 was used in both evaluations.) Now the LHS terms of R1 are evaluated: 0.3 [>= 0.3], (1 - 0.8) [< 0.1], 0.5, hence 1, 0, 0.5 and so the RHS term is evaluated to F3(1,0,0.5), say 0.6. Then the value of P8 is 0.6 [>= 0.2], i.e. the value of P6 is 1. Since we asked for the "max" evaluation, the value returned to the application program will be 1.

(4.4) Explanation component:

As a simple explanation mechanism the inference engine keeps track of the subtree used for the max evaluation for each predicate evaluated during the last inference cycle (and similarly for the min evaluation), and displays it when asked for, together with the input data which affected the particular values of facts on level 0 at the time of the evaluation. Thus, the user can trace back any inference to the facts and observations of level 0 (more on explanations see [NSH]).

(5) SEMANTICS OF McESE COMMANDS.

compile a source RSET : parses the RSET to provide syntactical check, performs other checks on rules, builds knowledge tree in main memory and stores the final data structure in a disk file.

load loads a compiled RSET from disk to main memory, if successful, a rsetid (rset id) is returned.

discompile recovers textual form of rules from a knowledge tree in main memory, displays it with some vital statistics.

open opens a knowledge, i.e. loads a compiled RSET from disk to main memory, loads compiled FSET, and checks the compatibility of the RSET with the FSET. If successful, a kbid (knowledge base id) is returned.

display displays an open knowledge base, i.e. it is like "discompile" except the statistics includes information from and about FSET.

last-open returns the kbid of the most recently opened knowledge.

last-load returns the rsetid of the most recently loaded RSET.

close closes the specified knowledge base, i.e. compiled RSET is unloaded from main memory, FSET is going to be ignored.

unload unloads a loaded RSET from main memory.

show-inc returns the inconsistency level tolerance for the specified open knowledge base.

set-inc re-sets the inconsistency level tolerance for the specified open knowledge base.

show-alarm returns the name of ALARM procedure for the specified open knowledge base.

set-alarm re-sets the name of ALARM procedure for the specified open knowledge base.

maxinfer performs backward chaining in the max mode on the given open knowledge base for the specified predicate and its arguments.

The max value is returned.

maxinfer-trace is as "maxinfer" except that the recursive evaluation is continuously displayed on screen.

sufmaxinfer performs backward chaining in sufficient max mode with the specified cutoff value on the given open knowledge base for the specified predicate and its arguments . The max value is returned.

sufmaxinfer-trace is as "sufmaxinfer" except that the recursive evaluation is continuously displayed on screen.

mininfer performs backward chaining in the min mode on the given open knowledge base for the specified predicate and its arguments. The min value is returned.

mininfer-trace is as "mininfer" except that the recursive evaluation is continuously displayed on screen.

sufmininfer performs backward chaining in sufficient min mode with the specified cutoff value on the given open knowledge base for the specified predicate and its arguments. The min value is returned.

sufmininfer-trace is as "sufmininfer" except that the recursive evaluation is continuously displayed on screen.

eval returns the value of the given predicate inferred during the last inference cycle.

forward performs forward chaining for the specified group of level 0 predicates to the specified level.

(6) COMPLETENESS OF McESE KNOWLEDGE BASES.

At this point, for the sake of clarity of the following discussion, we'd like to make a few definitions.

By a domain we mean a "piece" of an abstract world corresponding to a "piece" of real world, which we will call the domain realization.

By knowledge we mean the "know how" to solve "problems" in the chosen domain, very often we refer to it as the "domain knowledge". By domain information we mean the information about the state of the current realization of the domain the domain knowledge is supposed to deal with. To illustrate this on an example, recall the rule (from the domain of the card game Canasta) we have discussed before:

"opponent_collect(x) & used_stck_high =F=> ~discard(x)"

This is a piece of the domain knowledge. To use this piece of knowledge, we need some domain information, namely what is the state of affairs in the domain of the current game of Canasta (which is the Canasta domain realization), i.e. what does the opponent collect and how high is the used stack.

There are three different aspects of knowledge bases referred to as "completeness".

One really means "completeness of domain information available" and is intimately tied to the issue of rule firing with matching of their LHS's. This type of incompleteness is accommodated in inferring with McESE knowledge bases by the

mechanism of uncertainty (cvpf's and threshold directives). Every matching of a left hand side is "complete" in this sense, but varies in degree (reflecting the degree of available domain information about the current state of the domain realization), this projects into the degree (certainty value) of the RHS side term.

Another "completeness" refers to "completeness of the domain knowledge" and may be expressed as "is the knowledge represented in the knowledge base complete for the task?". There are no methods short of evaluating all possible problems in the domain to verify this kind of completeness (though its negation, i.e. incompleteness, may be verified by a problem not solvable by the represented knowledge). McESE does not address this issue and leaves it to the user to make sure his knowledge base is "adequate" to the task.

Since some people like to view knowledge bases as sets of logic formulae (see e.g. [LMP]), a term "completeness" may be then used in the logic sense, meaning that either every formula or its negation may be deduced from it. These types of knowledge bases are considered most often in the context of logic programming. McESE knowledge bases cannot be viewed as sets of logic formulae. The issue is not the strength of expressibility of McESE rules and how they reflect to logical formulae, but the fact that no deduction with the rules is done at all. In the logic sense one may deduce thinks which are not explicitly contained in the knowledge base itself, i.e. they are implicit with respect to the knowledge base (or better yet, they follow implicitly by deduction from the assertions in the knowledge base). In McESE we can "reason" only about things which are explicitly asserted and contained in the knowledge base. With respect to this explicitness, there is no point to talk about logical completeness of McESE knowledge bases. On the other hand, McESE knowledge base should be able to evaluate any predicate. Since the evaluation depends on input data, static check is practically impossible. Thus we opted for a simple run time check (see (4.3)).

(7) CONSISTENCY OF McESE KNOWLEDGE BASES.

The term "consistency" is also used with different meanings, though they are much closer. Most often it means that given the knowledge base and the input data, one cannot "arrive" to a contradiction. This is a weaker form than the logic consistency, meaning that one cannot deduce a formula and its negation from the given set of formulae. As discussed before, with respect to the explicitness of McESE knowledge bases, there is no point to consider the logic completeness of these. But in any case, static check for consistency is hard and computationally expensive. To check if a propositional formula together with a consistent set of propositions form a consistent set is NP-complete (see e.g. [LMG]). The consistency of explicit knowledge bases (as in case of McESE knowledge bases) may not be that expensive to check, for one does not have to go beyond the terms explicitly in the knowledge base, but it may be (and usually is) complicated by the input data giving the terms in the knowledge base different values. In McESE knowledge bases, thanks to their stratification into levels and their explicitness, the only source of a possible (explicit) contradiction may be an occurrence of a set of rules with the same predicate on the right hand side. The static check (i.e. in compilation time), which is more or less syntactical, cannot detect the possibility of a predicate being evaluated differently by different rules as these evaluations depend on input data. Thus, we opted for a run time check, when the user may specify how

"inconsistent" the knowledge base can be and in case the "inconsistency" level is crossed, an alarm is triggered (see (4.3)).

(8) "DEEP KNOWLEDGE" VERSUS "SHALLOW KNOWLEDGE".

Expert systems of the so-called first generation deal with what is called "shallow knowledge", i.e. the knowledge being just a set of heuristics, without any "deep" understanding of the domain, where all concepts are treated uniformly in a homogeneous way. On the other hand, so-called "deep knowledge" calls for explicit models of the domain embodying "understanding" of different concepts within the domain. There have been quite strong claims with regard to expert systems of the second generation using this "deep knowledge" (see e.g. [NSM]), but we tend to agree with [S] that these are mostly unsupported by substantial practical demonstrations yet. Being guided by our pragmatical goals, McESE belongs among first generation expert systems, with all the consequences for explanations and knowledge acquisition.

(9) EXPLANATIONS IN McESE.

McESE addresses this issue only superficially as most of the first generation expert systems do, i.e. the explanation consists of the trace of which rules have been "fired" to arrive to the conclusion together with the input data responsible for the particular evaluation. There have been numerous discussions about the fact that explanations of this nature are not very convincing (see e.g. [S], but the fact is that without "deeper understanding" being somehow embodied in the knowledge base, the explanations cannot really go beyond this. One may, though, treat the explanations as a domain of its own and try to formalize the knowledge needed to do so more convincingly for a knowledge base accompanying the one used for solving problems. This is the current interest in our research and we do plan to enhance McESE in this fashion.

(10) KNOWLEDGE ACQUISITION IN McESE.

As stated many many times, knowledge acquisition is the real bottleneck of expert systems development. It is very hard to get and formalize domain knowledge, and the task is time consuming and seemingly endless. Besides that, the domain knowledge may (and will) differ from one human expert to another. There are hopes (and claims) that the use of "deep knowledge" expert systems will alleviate this problem and will help facilitate in some way "learning" (see [S]). Given the current state of the research into these problems, McESE does not address it in any way and leaves it to the poor user (i.e. knowledge engineer) to deal with it in the old fashion way. There are, though, some interesting possibilities for "learning by observation" the cvpf's, given the concepts and their relations. This is in the plans for future research concerning McESE.

(11) DISCUSSION OF THE TREATMENT OF UNCERTAINTY IN McESE.

Presently there are two basic approaches to uncertainty in expert systems.

The first, linguistic approach, is what one may describe as "reasoning about uncertainty", where linguistic terms are used to deal with uncertain aspects of the knowledge (as we humans do), and their interpretation and verification is handled according to the "context" in which they were posed, i.e. close to have

different "scales" depending on what we talk about. Thus one may be close to something if it is just few centimeters afar, or the Earth may be close to the Sun, with respect to the size of our galaxy. The argument of people following this line of thought is a plausible one, namely that the knowledge representation we use ought to reflect the way we communicate our ideas as much as possible. But, from the pragmatical (computational) point of view this is not (at least at this stage) a very practical approach due to the need of a very sophisticated mechanism to handle this interpretation and verification of linguistic terms of uncertainty. We suspect that this approach may be feasible when systems for representation and reasoning with common sense are developed. For example see [CG] for a heuristic treatment of this type. There are other schemes in this category, like default reasoning, but we think they address more the problem of reasoning with incomplete and/or unreliable knowledge.

The second approach is a numerical one, where numbers are employed to indicate the degree of certainty, or uncertainty, or whatever other term may be used. This seems to be more natural (at least at this stage) and the field is richer in different schemes how to do it. We are inclined to view the problem of uncertainty in this light, given the fact that even we, humans, very often have to use numerical specifications to clear some aspects of our transmission of ideas (e.g. the Earth is close to the Sun, With respect to our galaxy, where we give the frame of reference and/or scale, which is always numerical in its principle). When we were deciding how to deal with uncertainty in McESE, we were mainly guided by admittedly pragmatical aspects of the proposed system. Thus the question was which of many schemes to employ. For an excellent overview of the field, see [G]. Briefly, there are about four main ways to deal with uncertainty numerically.

Probabilistic (Bayesian statistics), which has the advantage of a well developed terminology, technology and machinery, and also is easily comprehensible to humans as we have incorporated probability into our everyday lives and language. Some of the works in this field clearly influenced our thinking, especially for their clear "theoretical" approach and quite impressive theories (see e.g. [DP], [P], [RP]). But given the fact that one has to establish a host of conditional probabilities in "background" to facilitate the complete probability distribution and/or assume that most of "events" are statistically independent, renders the whole approach computationally almost impossible, or one must fit (and that is the most frequent case) his knowledge representation into a fixed scheme satisfying the underlying (and often implicit and hence "transparent") restrictions. These systems have what we call one degree of ad-hocness in that the conditional probabilities among the "events" are ad-hoc (i.e. supplied by the domain expert), their coherent processing is well-established and well understood process, and so easily interpreted (some mathematicians though argue that the coherence is only a superficial one, that the approach is meaningless as the "events" in question are not really part of a chance system).

Evidential approach, mostly based on variations of what is now commonly called Dempster-Shafer theory of belief functions, presupposes that the pieces of evidence are "independent", so again one has to mold his knowledge representation into a fixed scheme satisfying some (unfortunately implicit and not so obvious) underlying restrictions. Similarly, as with the probabilistic approach, the penalty to pay is that the system becomes incoherent (theoretically, it does not have to be visible on the performance of the system in question) when these underlying restrictions are

violated, with no recourse but re-fitting the knowledge representation into the required framework. Since these schemes are computationally even more taxing than the probabilistic approach, applications in the field tend to simplify matters by considering only the simplest of belief (support) functions. These systems have what we call two degrees of ad-hocness, for these evidential support belief functions must be supplied for each piece of evidence to be considered in order to get a meaningful interaction among all "events". They are propagated using some (slightly ad-hoc) machinery of combining evidence (Dempster's rule), and then the results must be interpreted (what they mean). The violation of underlying constrains is even less obvious than in the probabilistic case. On the other hand, there are common grounds between those two, as in the case of causal trees (see [SS]).

The third approach is "fuzzy logic" or "fuzzy set" approach. There every linguistic (uncertain) term is represented by a membership function and the degree of uncertainty reflects in the membership function of the "conclusion". There are some common grounds with Dempster-Shafer theory (see [Z],[Z1]), but we have not seen a practical application yet going beyond the use of max and min operators to propagate "certainties" through rules (corresponding to conjunction and disjunction respectively). These systems have what we call three degrees of ad-hocness, for one has to supply completely ad-hoc membership functions for each linguistic term, these are processed with an ad-hoc mechanism to obtain some results, whose interpretations are again ad-hoc. More than in the previous two approaches, the underlying constrains (with respect to max and min operators) are so tied, that the system becomes factually incoherent quite fast.

The fourth way to deal with uncertainty may be called ad-hoc systems. There belong systems with different mechanisms of propagating "certainty factors" (MYCIN - see [BS]), and so. The list would be quite long. These systems exhibit three degrees of ad-hocness (ad-hoc numbers to start with, ad-hoc propagation mechanism, and ad-hoc interpretation). However, they proved themselves quite well from the practical point of view.

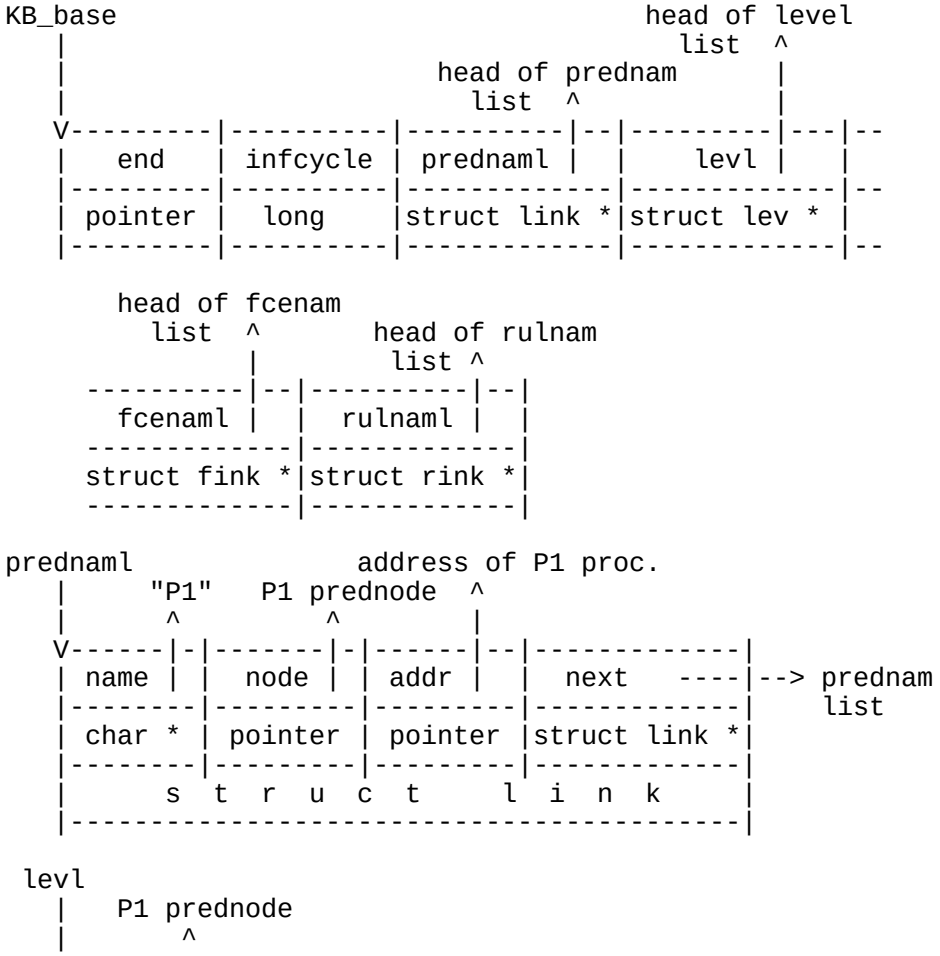
In the light of the above mentioned possibilities, bearing on our mind one of our most important goals (speed of execution, and thus a need of a simple mechanism) we opted for a "hybrid" solution exhibiting also three degrees of ad-hocness. The certainty values coming into the knowledge trees are ad-hoc in that that they are results of "computations" of level 0 predicate procedures and reflect the user's ideas about how certain he is about them. The cvpf's are ad-hoc, for it is up to the user to provide them, according to his "feelings". And finally the interpretation of the resulting values is also ad-hoc, for their is no "theoretical" explanation of what these numbers mean. Why did we opt for a system with three degrees of ad-hocness? Speed itself would not justify it, as having for example one or two functions to propagate the certainty values would be even faster and more convenient for the user. (a) we found systems of all four numerical approaches inherently inflexible. In brief, you have to "mold" the rules to fit the system, you have no freedom to capture the "structural" relationships of the predicates involved, and then work on capturing the "certainty value" characteristic of the rule. Our approach allows you to do it in two separate steps. (b) we found that too often we need different approaches to uncertainty within a single knowledge base (and/or control mechanism). Again, cvpf's allow you to "switch" from "probabilistic" to "evidential" to "fuzzy" to "ad-hoc" on the go. (c) we have been influenced by (very pointed) arguments by

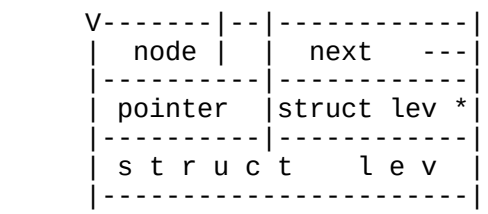
proponents of non-numerical approach, especially "heuristic" approach. Again, cvpf's allow us to do emulate it to a degree. They are nothing else, but (procedural) heuristics about uncertainty in the particular rule. (d) we found that very often we were able to formulate the "structure" of a rule correctly, while the numbers needed frequent correction (tuning). The separation of the structure of the rules (RSET) and the interpretation of uncertainty in the rules (FSET) helped to solve this. If the structure of the rules is satisfactory, one need to fine-tune the cvpf's only. (e) there was an additional bonus in the possibility to run some "experiments" with knowledge bases (RSET) under different interpretations of uncertainty (i.e. with different FSET's attached). This also is on the priority list of our future research topics.

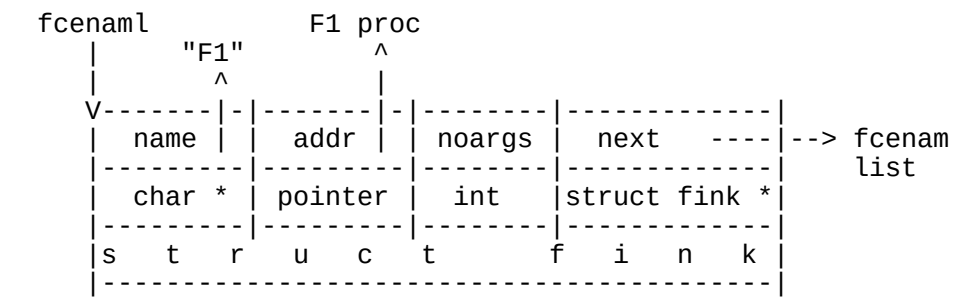
(12) FUTURE EXTENSIONS AND RESEARCH CONCERNING McESE.

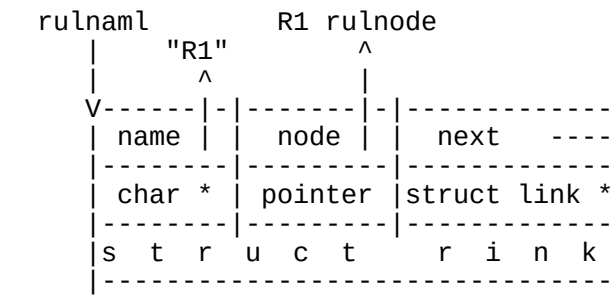
With respect to our quest for the greatest utility value (i.e. practicality of the system), the explanation mechanism must be improved, as we suggested before, via a heuristic approach. If McESE becomes popular and highly used, similar extensions of other languages may be considered. McESE will also be used as a test-bed for different interpretations of uncertainty to help with a development of the most suitable knowledge representation(s) for inferring with, under and about uncertainty. And, last, but not least, we hope, is a frequent use of McESE to build actual expert system applications.

APPENDIX - (compiled) knowledge tree structure and components:

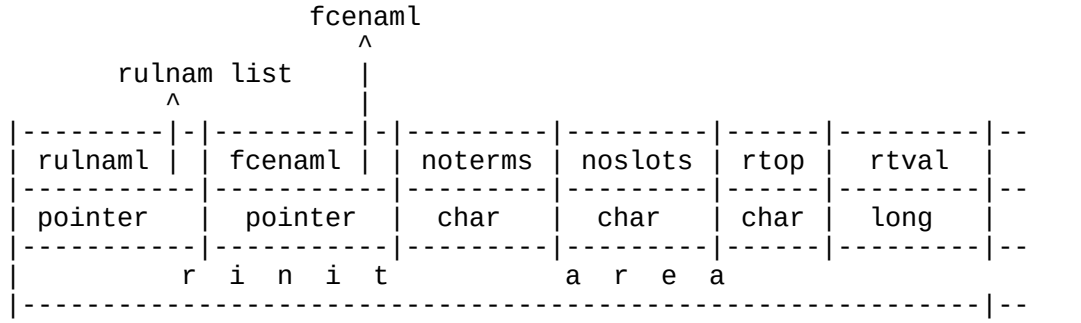


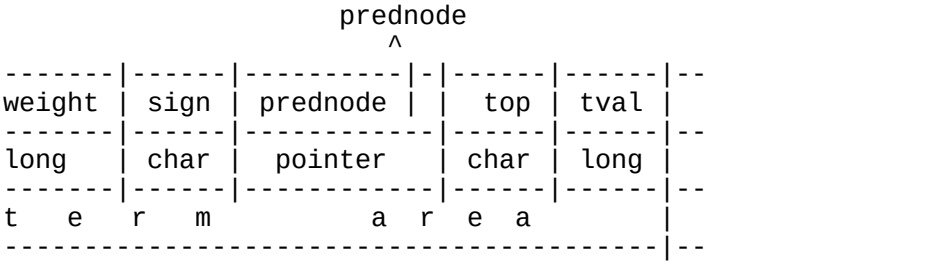




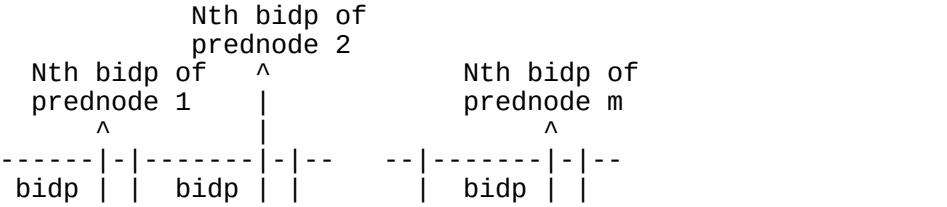


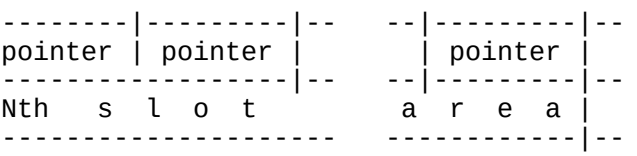
========================== R U L N O D E ==========================





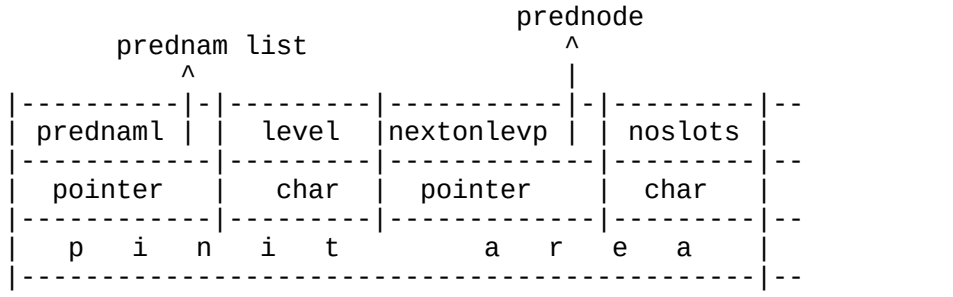
(term area repeats for each term in the rule in the natural order
of terms in the rule, i.e. the RHS term area is the last one)

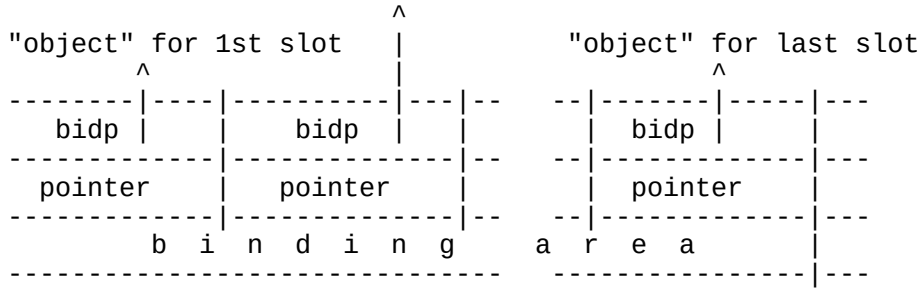




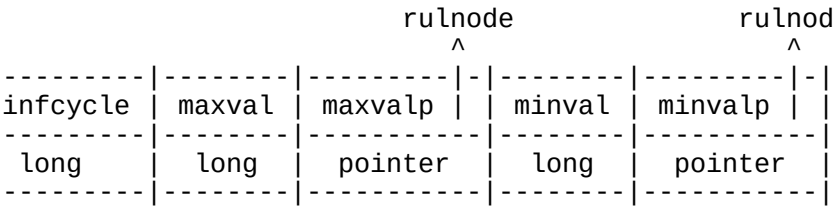
(slot area repeats number of slots times, in the order of slots)
(the order of binding pointers within the slot area follows the natural order of terms in the rule, i.e. the RHS term is the last one)

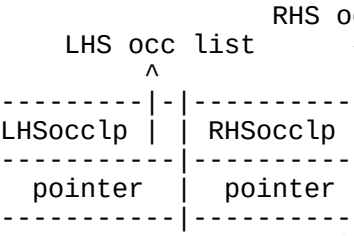
P R E D N O D E



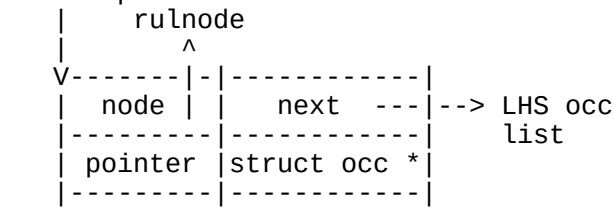


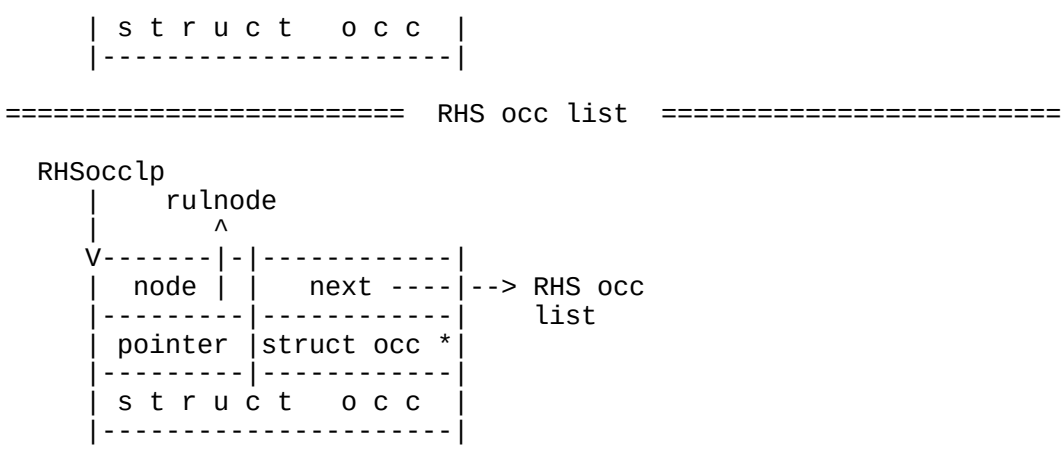
(binding area has number of slots boxes, in the order of slots)





LHS occ list





REFERENCES

[BS] B. Buchanan, E.H. Shortliffe, Rule-based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project, Addison-Wesley, 1984.

[CG] P. R. Cohen, M. R.Grinberg, A Framework for Heuristic Reasoning About Uncertainty, IJCAI 85 Proceedings.

[DP] D. Dubois, H. Prade, Combination and Propagation of Uncertainty with Belief Functions, IJCAI 85 Proceedings.

[D] J. P. Delgrande, An Approach to Default Reasoning Based on a First-Order Conditional Logic: Revised and Extended Report, LCCR Tech. Report 87-9, Simon Frazer University, Burnaby, B.C., Canada, 1988.

[G] Artificial Intelligence and Statistics, edited by W. A. Gale, Addison-Wesley, 1986.

[LMP] C. Lassez, K. McAllon, G. Port, Stratification as a Tool for Interactive Knowledge Base Management, IBM T. J. Watson Research Center Technical Report no. 55416, 1986.

[NSM] R. Neches, W. R. Swartout, J. Moore, Explainable (and Maintainable) Expert Systems, IJCAI 85 Proceedings.

[P] J. Pearl, Fusion, Propagation, and Structuring in Belief Networks, Artificial Intelligence 29 (1986).

[RP] I. Razier, J. Pearl, Learning Link-Probabilities in Causal Trees, Proceedings of the Workshop on Uncertainty in Artificial Intelligence, University of Pennsylvania, 1986.

[S] D. Sharman, A Review of Recent Developments Relating to Deep Knowledge expert Systems, research Report No. 86/236/10, University of Calgary, 1986.

[SS] P. P. Shenoy, G. Shafer, Propagating Belief Functions with Local Computations, IEEE Expert, Fall 86.

[Z] L. A. Zadeh, The Role of Fuzzy Logic in the Management of Uncertainty in Expert Systems, in Fuzzy Sets and Systems, North-Holland, 1983.

[Z1] L. A. Zadeh, Syllogic Reasoning as a Basis for Combination of Evidence in Expert Systems, IJCAI 1985 Proceedings.