23

PROLOG: a language for implementing expert systems

K. L. Clark and F. G. McCabe
Department of Computing
Imperial College, London

Abstract

We briefly describe the logic programming language PROLOG concentrating on those aspects of the language that make it suitable for implementing expert systems. We show how features of expert systems such as:

(1) inference generated requests for data,
(2) probabilistic reasoning,
(3) explanation of behaviour

can be easily programmed in PROLOG. We illustrate each of these features by showing how a fault finder expert could be programmed in PROLOG.

1.0 AN INTRODUCTION TO PROLOG

1.1 A Brief History

PROLOG is a programming language based on Kowalski's procedural interpretation of Horn clause predicate logic (Kowalski 1974, 1979). The language was developed and first implemented by Alain Colmerauer's research group in Marseilles (Colmerauer *et al.* 1973, Roussel 1975). There are now many implementations of the language, of which the most well known is the Edinburgh DEC-10 compiler/interpreter (Pereira *et al.* 1979). There are also machine coded interpreters for the IBM 370 (Roberts 1977), for the PDP-11 under UNIX (Mellish & Cross 1979), for any Z80 micro running CP/M (McCabe 1980) and for the Apple micro-computer (Kanoui and van Canaghem). Most of the interpreters give execution times of the same order as interpreted LISP, and DEC-10 compiled PROLOG compares favourably with DEC-10 compiled LISP (see Warren *et al.* 1977 for details). So we are talking about a programming language on a par with LISP with respect to efficiency. At the same time PROLOG provides much of the inference machinery that the expert system implementer has to program up in LISP.

In Marseilles and Edinburgh PROLOG is the main AI programming language. In Marseilles its major use has been for natural language understanding (Colmerauer 1978), but there was an early application to symbolic integration (Bergman & Kanoui 1973). In Edinburgh it has been used for plan formation (Warren 1974, 1976), problem solving (Bundy *et al.* 1979) and natural language understanding (Pereira 1980). It is also a handy language for compiler writing (Warren 1977). The DEC-10 PROLOG compiler is written in PROLOG.

In Hungary, where there are several implementations of PROLOG (see Santane-Toth & Szeredi 1980), the language is widely used for implementing expert systems. There are several concerning pharmacology: prediction of biological activity of peptides (Darvas *et al.* 1979), drug design aids (Darvas 1978), prediction of drug interactions (Futo *et al.* 1978). It is also been used in architecture to aid the design of complexes of apartments from modular units (Markusz 1977) and for many other applications (see Santane-Toth & Szeredi 1980). Outside Hungary its use to develop expert systems has been slow to start. At Imperial College it has been used to build a small fault finder system (Hammond 1980). The authors intend to explore more fully the use of the Z80 implementation for building desk-top expert systems. We also hope that this paper acts as a stimulus for others to use PROLOG to implement expert systems.

1.2 PROLOG Program = A Set of Rules + A Data Base

A PROLOG program comprises a sequence of clauses. The clauses are implications of the form

$$R(t1, .., tn) \text{ if } A1 \& \ldots \& Ak, k \geqslant 0.$$

Each Ai, like the $R(t1, .., tn)$, is a relation name R followed by a list of argument terms $t1, .., tn$. A term is a constant, a variable, or a function name f followed by a list of argument terms, $f(t1, .., tn)$.

The variable-free terms are the data structures of a logic program. Constants are like LISP atoms. A LISP list is a term such as $\text{cons}(A, \text{cons}(B, \text{Nil}))$, which can be written as $A.B.C.\text{Nil}$ in infix notation, or as $[A\ B\ C]$ in some PROLOGs. Different PROLOGs use different conventions to distinguish constants and variables. We shall use the convention that constants begin with an upper case letter (or are quoted), and that variables begin with a lower case letter.

Declarative reading

For the most part the clauses can be read as universally quantified implications, the quantification applying to all the variables of the clause. For example,

$$\text{fault}(x, \text{in}(u, y)) \text{ if } \text{part}(y, x) \& \text{fault}(y, u)$$

can be read as,

for all x, y, u
u in y is a fault with x if y is part of x and u is a fault with y.

Syntactic sugar

Several of the PROLOGs allow the programmer to declare that certain predicate and function names will be written as operators with a particular precedence. By making "in", "is-fault-with" and "is-part-of" operators we can write the above clause as

u in y is-fault-with x **if** y is-part-of x & u is-fault-with y.

This ability to extend the syntax with operators, or more generally to write a front-end program in PROLOG that accepts clauses written in any notation, makes it very easy to develop a user interface congenial to a specific use of PROLOG. A dramatic example of this is Colmerauer's use of the language for natural language understanding. The PROLOG program for parsing some subset of natural language is written as a set of productions of his metamorphosis grammars.

Assertions and rules

The clauses with an empty antecedent, which are written without the **if**, are *assertions.* The set of all variable free assertions in a program can be thought of as the database of the program. The other clauses are the *rules* that will be used to interrogate and/or modify the data base.

Queries

A query is a conjunction of atoms $B1\&\ldots\&Bn$. It invokes a backward chaining use of the rules to interrogate the data base. The interrogation is successfully completed when a substitution s has been found such that $[B1\&\ldots\&Bn]s$ 'follows from' the rules and the assertions. We have quoted 'follows from' since the substitution instance is not a logical consequence of the clauses viewed as assertions and implications when the data base of assertions has been modified during the interrogation. In expert systems applications we certainly want to be able to modify the data base, to accept new facts supplied by a user. If we are careful in the way we use the data base modifying facilities we can ensure that the $B1\&\ldots\&Bn$ substitution instance is a logical consequence of the final state of the data base and the 'pure' rules of the program, the rules that do not modify the data base.

As an example of a query evaluation let us suppose that the above rule about "is-fault-with" can access the assertions:

Engine is-part-of Car
Carburettor is-part-of Engine
Dirt is-fault-with Carburettor

The query

y is-fault-with Car

can be answered by a backward chaining deduction that binds y to (Dirt in Carburettor in Engine). Notice how the nested 'in's' give us the path to the fault.

Meta variable feature

Just as LISP can evaluate a function named by a list structure, so a PROLOG clause can use as a precondition an atom named by a term. This is usually referred to as the meta-variable feature since any clause that uses it can be thought of as an axiom schema. It means that we can write clauses of the form

$$R(\ldots, c) \text{ if } \ldots .\&c\&\ldots$$

in which a precondition is named by an argument term. We can think of the clause as an abbreviation for

$$R(\ldots, P(x1, .., xk)) \text{ if } \ldots .\&P(x1, .., xk)\&..$$
$$R(\ldots, Q(y1, .., yn)) \text{ if } \ldots .\&Q(y1, .., yn)\&..$$
$$\vdots$$

in which there is a specific clause for each relation name used in the program. The "$P(x1, \ldots, xk)$" appearing as a term is the name of the "$P(x1, .., xk)$" appearing as a precondition. This axiom schema explanation of the meta-variable feature is due to Phillippe Roussel.

1.3 Procedural Semantics

An evaluation of the query $B1\&\ldots\&Bn$ is broken down into the evaluation of the query $B1$, to find a substitution instance $[B1]s1$, followed by an evaluation of the query $[B2\&\ldots\&Bn]s1$. The $s1$ is a set of bindings for the variables of $B1$ which is passed on to the remaining conditions.

The *atomic query* $B1$ will be of the form $R(t1, .., tn)$ for some predicate name R. It is answered by trying each of the clauses of the form

$$R(t'1, .., t'n) \text{ if } A1\& .. \&Ak,$$

until one is found that applies to find an instance of the query. It applies if there are substitutions s and s' for the variables of $R(t1, .., tn)$ and $R(t'1, .., t'n)$, respectively, that will make them syntactically identical. Technically, the two formulae *unify*. All the PROLOGs use a simplified form of Robinson's unification algorithm (see Robinson 1979) to test for unifiability. The algorithm returns a pair of substitutions s and s' when the test succeeds. These are most general substitutions in the sense that any other pair of unifying substitutions are essentially specialisations of s and s'. It is in this rule application by unification that much of the power of PROLOG resides.

If $k = 0$, i.e. the program clause is an assertion, the evaluation of $B1$ is completed with s as the answer substitution. Otherwise the evaluation of $B1$ reduces to the evaluation of the new query $[A1\& .. \&Ak]s'$. The answer for $B1$ is then s modified by the answer substitution s'' for this new query.

An example evaluation

The atomic query

z is-fault-with Car

can be solved using the program clause

u in y is-fault-with x **if** y is-part-of x & u is-fault-with y.

Under the substitutions

$s = \{z/u \text{ in } y\}$
$s' = \{x/\text{Car}\}$
z is-fault-with Car, u in y is-fault-with x

become identical. The evaluation of the query reduces to the evaluation of the derived query

y is-part-of Car & u is-fault-with y.

With the assertions given above this will be solved with the answer

$s'' = \{y/\text{Carburettor in Engine}, u/\text{Dirt}\}$

The answer to the original query is

z/u in y where y/Carburettor in Engine and u/Dirt,

i.e. it is z/Dirt in Carburettor in Engine. For more information on the procedural semantics we refer the reader to Kowalski (1979).

Backtracking

The above recursively described evaluation process always takes the first program clause for which a matching substitution can be found. There may be other program clauses which could also be used. These will be tried should a subsequent evaluation step *fail*.

A failure occurs when none of the program clauses will unify with the atomic query currently being attempted. When this happens the evaluation *backtracks* to the most recent previous evaluation step for which there are untried program clauses.

Controlling the search

The program clauses for a relation are always tried in the order in which they appear in the sequence of clauses. PROLOG programmers often exploit this order of use to give ordinary rules for a relation first, followed by a default rule that should only be used if the ordinary rules have failed. There is a backtracking control device, the "/", which can be used to prevent a later rule being used if an earlier rule has been successfully applied.

For example, suppose that the rule

B **if** $A1\&..\&Ai/Ai+1\&..\&An$

has been invoked to find an answer to an atomic query B'. If the preconditions $A1,..,Ai$ that appear before the "/" can be solved then no later clause will be used to try to solve B'. The "/" expresses the control information that the successful evaluation of $A1$ to Ai indicates that the later rules that might be used to solve B' will either fail or give the same answer. It also prevents backtracking on the proof of $A1\&..\&Ai$.

2.0 FEATURES USEFUL FOR EXPERT SYSTEMS

Inputs and updates

During a query evaluation data can be input from a user, and assertions and rules can be added to the program, using special primitive relations. The evaluation of **read**(x) will cause x to be bound to the next term typed at the input terminal. The evaluation of **assert**(x) will cause the term which is the current binding of x to be added as a new rule. Thus, the rule

Ask-about(c) **if** print$(c,$ "?") & **read** (ans)
& ans = Yes & **assert** (c),

used to try to answer the query,

Ask-about(Dirt is-fault-with Carburettor),

will print the message

Dirt is-fault-with Carburettor?,

read the next term, and if it is the constant Yes it will add

Dirt is-fault-with Carburettor

as a new assertion about the is-fault-with relation. Where this assertion is added is important. It can be added at the beginning of the list of clauses for the relation, or the end, or at some intermediate position. In this situation we would like it added at the beginning. Where it is added is an option that can be specified by the programmer. We shall not go into the details of this.

Dynamic data base

The rule

u in y is-fault-with x **if** y is-part-of x & u is-fault-with y

must access assertions giving components and assertions about faults with components. We can use the Ask-about rule to allow the assertions about faults to be added dynamically as we try to solve the problem of finding a fault.

Instead of assertions about known faults with components we include in the initial data base only assertions about possible faults, knowledge that expert should have. We then include the rule

u is-a-fault-with y **if** u is-a-poss-fault-with y & Ask-about(u is-a-fault-with y)

Let us pause for a moment to consider the behaviour of our fault finder. When asked to find a fault with some device with a query

u is-fault-with Device

the use of the first rule for faults will cause the fault finder to walk over the structure of Device as described by the is-part-of assertions. When it reaches an atomic part it will query the user concerning possible faults with this component as listed in the is-poss- fault-with assertions. It will continue in this way, back-tracking up and down the structure, until a fault is reported. As it currently stands, our expert sytem helps the user to look for faults.

Generation of lemmas

Sometimes it is useful to record the successful evaluation of an atomic query B by adding its derived substitution instance as a new assertion. Thus, suppose we have a rule

$R(t1,..,tn)$ **if** $A1\&..\&Ak$

and we want to generate a lemma each time it is successfully used to answer a query $R(t'1, .., t'n)$. We add an extra assert condition at the end of the list of preconditions of the rule.

$R(t1,..,tn)$ **if** $A1\&..\&Ak\&\text{assert}(R(t1,..,tn))$

If this solves the atomic query with answer substitution s then $[R(t'1,..,t'n)]s$ will be added as new assertion. It will be added at the front of the sequence of clauses for R.

By adding asserts we can also simulate the use of rules with conjunctive consequences. Suppose that we know that both B and B' are implied by the conjunction $A1\&..\&An$. Normally we would just include the two rules:

B **if** $A1\&..\&An$
B' **if** $A1\&..\&An$

in the program. The drawback is that we need to solve $A1\&..\&An$ twice in derivations where both B and B' are needed. We can avoid the duplication by writing the two rules as:

B **if** $A1\&..\&An\&\text{assert}(B')$
B' **if** $A1\&..\&An\&\text{assert}(B)$

The successful use of either rule will add the other consequent as a lemma.

By developing a suitable front end to PROLOG we can shield the programmer from the details of the lemma generation. We could allow him to write rules with conjunctive consequents and to specify which rules were lemma generation rules. The front end program would expand rules with conjunctive consequents into several rules and add the appropriate asserts to the end of each of these rules. It would also add an assert to the end of each of the lemma rules.

All solutions

Sometimes we want to find all the answers to a query, not just one answer. This is an option in some of the PROLOGs. Where it is not we can make use of a meta-rule such as

All(query, term) **if** query & print(term) & fail.

This will print out [term]s for each answer substitution *s* to query. The "fail" is a condition for which we assume there are no clauses. With a slightly modified rule, we can define the relation

l is-all term such-that query

that holds when *l* is the list of all the instantiations of "term" for answer substitutions to "query". In DEC-10 PROLOG such a relation is now a primitive of the language. Using it we can write rules such as

l is-a-list-of-faults-with *x* **if** *l* is-all *u* such-that
u is-fault-with *x*

The use of this rule will result in *l* being bound to a list of all the faults with *x*.

We can now consider a very simple extension to our fault finder. Instead of asking for a single fault we can ask for a list of all the reported faults with corrective actions. We do this with a query of the form

l is-all [*u a*] such-that
u is-fault-with *D* & *a* is-action-for *u*.

To handle this query we must also include in our database a set of assertions giving actions for faults, information supplied by the expert. An evaluation of this new query will guide the user through the structure of device *D*, asking about faults in components. Each reported fault will be paired with its corrective action. Finally the list of pairs [reported-fault corrective-action] will be printed.

3.0 INEXACT REASONING

MYCIN(Shortliffe 1976) and PROSPECTOR (Duda *et al.* 1979) use inexact or probabalistic reasoning. Probabilities are associated with assertions and rules. Conclusions are derived with associated probabilities. To implement this in PROLOG we augment all those relations about which we want to make probabilistic assertions with an extra argument. Instead of $R(x,y)$ we use $R(x,y,p)$. We read this as

$R(x,y)$ with certainty p.

We now add to the rules that refer to these relations extra conditions to deal with the transfer of certainties from the preconditions to the conclusion.

Let us elaborate our fault finder program to deal with uncertainties. To

make it more of an expert we should have rules for detecting faults by their symptoms. We should also allow that symptoms only partly correlate with faults. This means that the is-fault-with relation should have an extra argument to carry a certainty measure. We shall not go into details of what this is. Whatever measure is used must be handled by the probability transfer relations. The definition of these can implement whatever probability theory we choose.

Instead of the rule that queries the user about possible faults we can now have the rule

u is-fault-with *x* certainty *p* **if**
s is-symptom-for-fault *u* of *x* strength *q*
& *s* certainty *r*
& *q* and *r* give *p*.

This rule accesses assertions such as

Stalling is-symptom-for-fault Dirt of Carburettor strength .3.

The strength measure is the degree to which the symptom correlates with the fault. We also need assertions about the certainty of the symptoms, or more usefully, a rule such as

u certainty *r* **if** print("Give the certainty of", *u*)
& **read**(*r*) & **assert**(*u* certainty *r*)

The rules for "*q* and *r* give *p*" implement our probability theory.

When invoked by a query to find all the faults with some device our new probabilistic rule will query the user about the likely occurrence of each symptom that correlates with a fault in the device. It will use the user supplied certainty that the symptom is present, and the degree to which it correlates with the fault, to give a certainty for the associated fault.

From single symptoms to sets of syndromes

To compute a certainty for a fault on the basis of a single symptom is a little too simple. More realistically a fault will be signalled by one or other of several clusters of symptoms, that is, by one or other of several syndromes. A better version of our probabilistic is-fault-with rule is

u is-fault-with *x* certainty *p* **if**
l is-all [*s q*] such-that
s is-syndrome-for *u* of *x* strength *q*
& *l* gives *p*.

The rules for "*l* gives *p*" must recurse down the list of syndrome-correlation pairs. In doing this it must compute the certainty of each syndrome using the certainties of the individual symptoms supplied by the user. It then computes a certainty for the fault using the certainties and strengths of all the syndromes.

A syndrome can be quite complex. The syndrome

symptom $S1$ and either symptom $S2$ or symptom $S3$ in the absence of symptom $S4$

can be denoted by the term

$S1$ & ($S2$ or $S3$) & not $S4$

where "&", "or" & "not" are operators. To handle syndromes denoted by such terms the rules for computing the certainties of syndromes would look like:

certainty-of-syndrome u is p **if** symptom(u) & u certainty p

certainty-of-syndrome(u & v) is p **if**
certainty-of-syndrome u is q
& certainty-of-syndrome v is r
& anding [q r] gives p

The first rule deals with the case of a syndrome which is a symptom. The second deals with the case of one that is a conjunction of syndromes. The rules for

anding [q r] gives p

compute the certainty of a conjunction of two syndromes given their individual certainties. How this is done is determined by the choice of probability theory. If we want to take into account such subtleties as dependencies between symptoms we would add an extra precondition

dependency-of u and v is d

to the rule and include d as an argument to the "anding" condition. We now also include in the data base assertions and rules about the interdependence of symptoms and syndromes.

We would have analogous rules for the case of a "u or v" syndrome and the case of a "not u" syndrome.

Symptom nets

The set of alternative syndromes for a fault F in a component C will be described by a set of assertions of the form

($S1$ & ($S2$ or $S3$) & not $S4$) is-syndrome-for F of C strength Q.

The set of all the assertions about a fault F are a description of a symptom net for F as depicted in Fig. 1.

This is just a shallow inference net. The movement of probabilities along the arcs of this is a special case of what PROSPECTOR does.

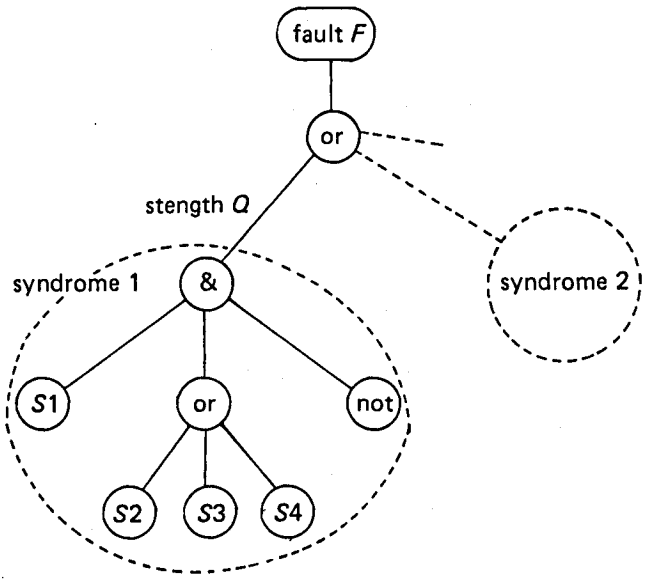


Fig. 1.

A prototype general fault finder

Let us conclude this section of probabilistic inference by examining the state of our PROLOG fault finder. The three rules:

u in y is-fault-with x certainty p **if** y is-part-of x
& u is-fault-with y certainty p

u is-fault-with x certainty p **if**
l is-all $[s\,q]$ such-that
s is-syndrome-for u of x strength q
& l gives p

help(s,l) **if** l is-all $[u\,a\,p]$ such-that
u is-fault-with s certainty p
& a is-action-for u,

together with the rules that implement the probability theory, form the nucleus of a general fault finder. To use this fault finder program we

(1) add a set of "is-part-of" assertions giving the structure of a new device about which we want help in finding faults,
(2) add a set of assertions giving the correlation of syndromes with faults in atomic components,
(3) add a set of assertions giving the appropriate action for each fault.

The sets of assertions (1), (2) and (3) are what the expert provides. It is the way that he 'programs' the fault finder.

A user of the fault finder asks for help in mending some substructure S of the device with the query help(S,I). He waits to be asked about the certainty of occurrence of various symptoms. He will have printed out a list of possible faults together with the corrective action. Each fault will be described by a path name to its position in S. It will be listed with a likelihood of its presence.

4.0 EXPLAINING

One of the most important abilities of an expert system is to be able to explain itself to a user in intelligible terms. If an expert system produces a fault diagnosis of a car, or an analysis of a chemical compound, the user needs to be convinced that the result is reasonable; that it has not been pulled out of the hat. So an expert system should be able to give an account of how it reaches its conclusions.

Moreover, no expert system produces its answers without some data, which is usually supplied by the user when prompted by the system. It is reasonable for the user to be able to ask why the data is being requested.

There are, then, two kinds of explaining for an expert system, the "HOW" and the "WHY". Let us look at one way in which a set of PROLOG clauses can be modified to cope with these two kinds of explaining.

WHY explanations

Consider again the non-probabilistic version of the PROLOG fault finder. Let us suppose that we want to allow the user to ask why he is being asked if there is some particular fault with a component. To be able to respond to such a "WHY" request the ask-about rule must have access to the context in which it is invoked. One way in which we can do this is to pass a description of the context as an extra argument to each rule that might ultimately lead to the use of the Ask-about rule. For example, instead of the rule

u is-fault-with x **if** u is-poss-fault-with x &
ask-about(u is-fault-with x)

which we shall call rule 2, we could use

u is-problem-with x trace t
if u is-poss-fault-with x &
Ask-about(u is-fault-with x,
2: [u is-poss-fault-with x] . t).

The 2:[u is-poss-fault-with x] added to the front of the current trace t tells us that rule 2 has been invoked and that its precondition (u is-poss-fault-with x) is satisfied. Similarly, we could modify rule

u in y is-fault-with x **if** y is-part-of x & u is-fault-with y,

called rule 1, to the rule

u in y is-fault-with x trace t
if u is-part-of x &
u is-fault-with x trace 1:[u is-part-of x] . t

This adds the fact that rule 1 is being used to the front of the trace and records the single precondition of the rule that is currently satisfied.

The Ask-about rules now need modification to:

Ask-about(c,t) **if** print(c, "?") & **read**(ans) & respond(ans,c,t)
respond(Yes,c,t) **if assert**(c)
respond(Why,$c,u.t$) **if** give-reason(u) & Ask-about(c,t)
respond(Why,c,Nil) **if** print("you should know")
& Ask-about(c,Nil).

The new Ask-about will pick off an explanation u from the trace list for each "Why" repsonse and display it. The rules for give-reason might access a set of assertions of the form

description(n, "some text describing rule n")

in order to give a meaningful description of the rule mentioned in u. Each repeated "Why" leads to a more distant reason for the question being printed out by the give-reason rules. Finally, when the trace is empty, we are back at the top-level query. A "Why" at this point gets the response "you should know" followed by a new prompt.

As an example of the use of these new rules, let us suppose that in solving the query

u is-fault-with Car trace Nil

rule 1 has been used twice because of the assertions

Engine is-part-of Car
Carburettor is-part-of Engine,

and then rule 2 has been used. It has accessed the assertion

Blocked-valve is-poss-fault-with Carburettor

and is now querying the user about this fault. The trace argument passed to Ask-about is

2:[Blocked-valve is-poss-fault-with Carburettor].
1:[Carburettor is-part-of Engine].
1:[Engine is-part-of Car].Nil

Rule transformation

As with lemma generation the modification of the clauses to cater for "Why" explanations can be performed by a front end program. The input to this program would be numbered rules without the trace argument. Along with this program would be a set of directives which specified which rules should be traced and which relations are such that their proofs might lead to the use of "Ask-about". We are assuming that "Ask-about" is being used as though it were a primitive

predicate. The programmer would also include a set of assertions describing the explainable rules. The front end program rewrites the designated rules to include the trace argument.

The rewrite is quite straightforward. Suppose we have a rule of the form

Rule k: $R(t1, .., tn)$ if $A1 \& .. \& Ai \& P(t'1, .., t'm) \& ..$

that must be traced. Let us also suppose that P is a relation the proof of which can lead to the use of the Ask-about rules. We must transform the rule into

Rule k:
$R(t1, ..tn, t)$ if $A1 \& .. \& Ai \& P(t'1, .., t'm, k : [A1, .., Ai].t) \& ..$

The rule passes down to the rules for P the current trace extended by a message to the effect that rule k has been used. The list of terms $[A1, .., Ai]$ that is also passed down will be the preconditions $A1, .., Ai$ instantiated by any substitution generated by the proof of this conjunction of conditions. It tells us exactly what the state of play is in the application of this rule.

How explanations

We can use the same extra argument trick to save proofs. We can also *hide it* from the expert system programmer. He simply states that certain relations should be proof traced, these being the relations that he wants mentioned in the "How" explanations. The front end program then transforms the clauses for these relations as follows.

An assertion

$P(t1, .., tn)$

becomes

$P(t1, .., tn,\ P(t1, .., tn))$

The extra argument is the trace of the proof of the instance of P found using the assertion. It will be a copy of the proven instance.

A rule of the form

$P(t1, .., tn)$ if $..\&Q(..)\&...\&R(..)\&..,$

in which Q and R are the only proof trace relations in the antecedent, is transformed into

$P(t1, .., tn, \text{proof1} \,\&\, \text{proof2 implies } P(t1, .., tn))$ if
$..\&Q(.., \text{proof1}) \& ... \& R(.., \text{proof2}) \& ..$

This constructs a description of the proof of the instance of P that it generates in terms of the descriptions of the proof trace relations that it accesses. If this rule is also declared a lemma generating rule we can transform it into

$P(t1, .., tn, P(t1, .., tn))$ if
$..\&Q(.., \text{proof1}) \& .... \& R(.., \text{proof2}) \& ..$
$\& \text{ assert}(\text{proof1} \,\&\, \text{proof2 implies } P(t1, .., tn))$
$\& \text{ assert}(P(t1, .., tn))$

This asserts the explanation as well as the lemma. The proof trace returned is a reference to this lemma.

The following rule now defines a relation that the user can invoke to seek an explanation of some asserted lemma $P(t1, .. tn)$ with a query Explain($P(t1, .., tn)$).

Explain(lemma) if proof implies lemma & display(proof)

This retrieves the term that describes the proof of the lemma from the "implies" assertion for the lemma and then displays it in a suitable format. The explanation will be a proof structure down to other asserted lemmas. If the user wants to see more he asks for an explanation of these other lemmas.

5.0 CONCLUSIONS

We hope that we have convinced the potential expert system implementer that he should look at PROLOG more closely. We do not claim that the techniques for programming expert systems that we have sketched are the best. They represent just one possibility. A fully fledged implementation of our fault finder would undoubtedly reveal shortcomings. Nonetheless we are convinced that PROLOG offers a programming medium in which these could be easily overcome. We invite the reader to do his own experimenting with PROLOG.

6.0 ACKNOWLEDGEMENTS

The main ideas in this paper evolved through discussions with P. Hammond. Our fault finder example was inspired by his PROLOG fault finder, although he tackles the problem in a slightly different way.

The paper was written whilst Keith Clark was visiting Syracuse University. Comments on the ideas by K. Bowen, J. A. Robinson and E. Sibert were much appreciated.

We would also like to thank Diane Zimmerman. She patiently typed the paper into a text formatter using an often overloaded computer system.

7.0 REFERENCES

Bergman, M., Kanoui, H. (1973), Application of mechanical theorem proving to symbolic calculus, *3rd Int. Symp. on Adv. Methods in Physics*, C.N.R.S., Marseilles.

Bundy, A., Byrd, L., Luger, G., Mellish, C., Palmer, M., (1979), Solving mechanics problems using meta-level inference, *Expert systems in Micro Electronic Age*, pp. 50–64 (ed. Michie, D.). Edinburgh: Edinburgh University Press.

Colmerauer, A., Kanoui, H., Pasero, R., Roussel, P. (1973), Un systeme de communication homme-machine en francais. Rapport, Groupe d'Intelligence Artificielle, Marseille: Univ. d'Aix, Luminy.

Colmerauer, A. (1978), Metamorphosis Grammars. *Natural language communication with computers*, pp. 133–189 (ed. Bolc, L.) *Lect. Notes in Comp. Sci. No. 63*, Springer Verlag.

Darvas, F., (1978), Computer analysis of the relationship between biological effect and chemical structure. *Kemiai Kozlemenyek*, 50, (Hungarian).

Darvas, F., Lopata, A., Gy. Matrai, (1979), A specific QSAR model for peptides. In *Quantitative Structure Activity Analysis*, (ed. Darvas, F.), Budapest: Akademiai Kiado.

Duda, R., Gashnig, J., Hart, P., (1979), Model design in the prospector consultant system for mineral exploration, *Expert Systems in the Micro Electronic Age*, pp. 153–167 (ed. Michie, D.). Edinburgh: Edinburgh University Press.

Futo, I., Darvas, F., Szeredi, P., (1978), Application of PROLOG to development of QA and DBM systems, *Logic and Data Bases*, (ed. Gallaire, H. and Minker, J.). Plenum Press.

Hammond, P. (1980), Logic programming for expert systems, MSc Thesis, London: Imperial College, University of London.

Kanoui, van Caneghem, (1980), Implementing a very high level language on a very low cost machine, Rapport, Marseille: Group d'Intelligence Artificielle, University d'Aix-Marseille, Luminy.

Kowalski, R. (1974), Predicate logic as programming language, *Proceedings IFIP 74.*

Kowalski, R. (1979), *Logic for problem solving,* Amsterdam and New York: North Holland.

McCabe, F. G. (1980). Micro-PROLOG programmers reference manual, 36 Gorst Rd., London: LPA Ltd.

Markusz, Z. (1977), How to design variants of flats using programming language PROLOG, based on mathematical logic, *Proc. Information Processing 77*, pp. 885–889 (ed. Gucheist, B.) Amsterdam, New York, Oxford: North Holland.

Pereira, L., Pereira, F., Warren, D., (1978), User's guide to DEC-system 10 PROLOG, Edinburgh: Dept. of A.I., Edinburgh University.

Pereira, F. (1980), Extraposition grammars, *Proceedings of Logic Programming Workshop*, (Budapest).

Roberts, G. M. (1977). An implementation of PROLOG, M.Sc. Thesis, Waterloo: Dept. of Computer Science, University of Waterloo, Canada.

Robinson, J. (1979), *Logic: Form and Function*, Edinburgh: Edinburgh University Press.

Roussel, P. (1975), PROLOG, Manuel de Reference et d'Utilisation, Marseille: Groupe d'Intelligence Artificielle, U.E.R. de Luminy, University d'Aix.

Shortliffe, E. H. (1976). *Computer Based Medical Consultations: MYCIN*, Americal Elsevier, New York.

Santane-Toth, E., Szeredi, P., (1981), PROLOG applications in Hungary, In *Logic Programming* (eds. Clark, K. L. and McCabe, F. G.), London: Academic Press.

Warren, D., Pereira, L., Pereira, F. (1977), PROLOG – the language and its implementation compared with LISP, *Proc. Symp. on AI and Prog. languages, SIGPLAN notices*, 12, No. 18.

Warren, D. (1974), WARPLAN: A system for generating plans, Memo, Dept. of A.I., Edin. Univ.

Warren, D. (1976), Generating conditional plans and programs, *Proc. AISB Summer Conference*, (Edinburgh).

Warren, D. (1977), Logic programming and compiler writing, Report, Edinburgh: A.I. Dept., Edinburgh University.

Appendix to PROLOG: a language for implementing expert systems

P. Hammond
Department of Computing
Imperial College, London

INTRODUCTION

An expert system, which is domain-independent, has been implemented in Micro-PROLOG* with many of the features described above. It interacts with a knowledge-base and a set of rules for handling uncertianty to become a domain expert. The facilities that have been built into the system include the following:

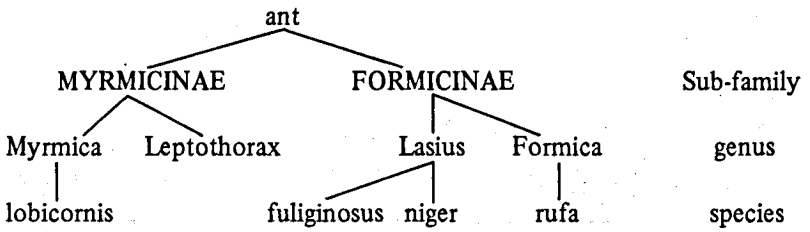
(i) the handling of degrees of belief in evidence and their subsequent effect on related deductions;
(ii) the explanations of *why* a particular question is being asked and *how* a deduction has been made;
(iii) the use of key symptoms to direct the problem-solving activity to a narrower solution set;
(iv) the ability for the user to query the knowledge-base as a database, e.g., to see the rules;
(v) the inclusion of symptoms that have values e.g., age and temperature;
(vi) the use of external files to store relations to avoid loading the entire knowledge base into machine memory.

CHANGES IN THE DATA REPRESENTATION AND METHODOLOGY

The system has been used with knowledge-bases on car fault-finding, skin disease diagnosis, ant identification, personal investment analysis and pipe corrosion diagnosis.

To accommodate such a wide range of use, which goes beyond simple fault-finding in a structured object, the knowledge representation framework was generalised so that each application is just a special case of a more general scheme. For example, the "part of" structure of the fault finder has now become a partition of the solution space into a collection of nested subsets rather like the taxonomical divisions used in botany and zoology. In fact, the ants data-base

is a good example of this subclassification process. The following diagram illustrates the divisions of a collection of British ants into sub-family, genus and species:



Now, instead of having one complex syndrome for indentifying "fuliginosus" such as

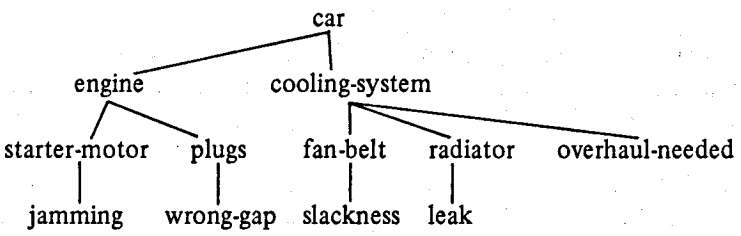
Deduction	*Syndrome*
fuliginosus in ants	number of waist segments is 1 & leg-length is short & colour is black & head-shape is heart-like.

we can define a hierarchy of rules corresponding to the partition above:

Deduction	*Sydrome*
MYRMICINAE in ants	number of waist segments is 2
FORMICINAE in ants	number of waist segments is 1
Lasius in FORMICINAE	leg-length is short
Formica in FORMICINAE	leg-length is long
fuliginosus in Lasius	colour is black & head-shape is heart-like
niger in Lasius	colour is black & head-shape is normal

Thus the rules for differentiating within Lasius are only tested if we have established the syndrome in the rule for identifying Lasius in FORMICINAE.

To illustrate how this division applies to the familiar fault finder consider the following:



This tree reflects a fault classification scheme where faults are associated with each group of components. Rather than simply reflecting the construction of the car it describes how faults are associated with assemblies and sub-assemblies of components. For example, "cooling-system" now stands for the class of cooling-system faults.

INEXACT REASONING

We have developed two modules for handling inexact reasoning. One uses MYCIN-like and the other PROSPECTOR-like (or BAYESIAN-like) uncertainty. The designer of the knowledge-base can decide which method to use, without affecting the kernal of the expert system or the form of the data-base. Of course each scheme has its own requirements; the MYCIN method has a single number for each rule that describes a syndrome-deduction correlation whereas the PROSPECTOR method involves two such numbers. The possibility also exists for the expert to define his own method of handling uncertainty.

The pipe corrosion data-base uses the BAYESIAN-like uncertainty and each of the other data-bases uses MYCIN-like uncertainty. Some effort is being made by the authors to find a satisfactory way of handling dependency between symptoms.

KEY-SYMPTOMS

An expert frequently makes use of key symptoms to focus attention on a sub-class of problems. To aid this important process the PROLOG classifier allows the knowledge-base constructor to declare certain key symptoms that can be used to concentrate on a particular subclass in the "sub-class" hierarchy. For example, we could declare "number-of-waist-segments", "colour" etc. as key symptoms and define the following:

number-of-waist-segments is 1	suggests-check	class MYRMICINAE
colour in-list (black yellow)	suggests-check	class Lasius
high-petrol-consumption	suggests-check	class carburettor
high-petrol-consumption	suggests-check	identification leak in fuel-system
engine-fails-to-start	suggests-check	class battery

These key symptoms will be requested at the beginning of a consultation. Those that are reported cause the classifier to check the suggested areas.

SYSTEM ARCHITECTURE

The figure above gives a simple-minded view of the structure of the classifier with particular reference to the flow of information between the user, the core program and the knowledge-base. Once the user has indicated which knowledge-base is to be consulted, the core program is able to infer from the rule structure (see under INEXACT REASONING above) which method of handling uncertainty is appropriate.

This diagram is also useful in explaining how the system reacts to special responses from the user. For example, we have already indicated that the response "why" causes the system to re-trace the history of the consultation to explain its current line of reasoning. This is handled within the core program by the assertion

needs-response (why)

and a set of general rules for the re-tracing. If a user wishes to alter the form of the explanantion to make it more domain-specific, he need only re-define the general rules to his own liking and include them in the knowledge-base. Similarly, other responses to handle special queries to the knowledge-base can be defined by assertions of the form

needs-response (deductions) , needs-response (help)

along with rules which print the deductions that can be made or which explain the help facilities which are available.

SYMPTOMS THAT HAVE VALUES

Already from the examples above we have illustrated a fair degree of freedom in symptom description. Currently the system can recognise symptoms which are of the following types:

(i) mnemonic-like strings such as engine-wont-start;
(ii) ⟨name⟩ ⟨relation⟩ ⟨value⟩
where ⟨relation⟩ is a pre-defined binary relation between possible values of ⟨name⟩ and ⟨value⟩; for example

age	in-range	(20 to 40)
colour	in-list	(red yellow)
height	LESS	40
size	is	10

EXTERNALLY DEFINED RELATIONS

The system has so far been used on Z80 based microcomputers with small internal memory (56 to 64 k). As knowledge-bases have grown it has become necessary to store parts of the data in backing store on disc. The system can then access this information as the need arises. Another use of this facility would be to access data-bases related to the knowledge area.

FUTURE DEVELOPMENTS

The work with experts in skin problem diagnosis and pipe corrosion is continuing and it is hoped that these databases will become more realistic as more detailed knowledge is added. Other collaboration with industry is being set up and plans have already been made to implement the classifier on more powerful computers, such as the PERQ, and to experiment with the uses of computer graphics.

The construction of knowledge bases which can interface with the PROLOG classifier was part of a recent MSc thesis by M. Y. Chin (1981) at Imperial College. This project also illustrated how the classifier could be used for structured questionnaires.

The use of quasi-natural language for expert systems has been investigated by another recent M.Sc. student at Imperial College (Steele 1981). The knowledge representation and problem-solving strategy used in this project illustrate alternatives to that used in our general purpose classifier.

REFERENCES

M. Y. Chin (1981), Computer Interrogation Systems: CISP, M.Sc. thesis, London: Department of Computing, Imperial College.

F. G. McCabe (1981), Micro-PROLOG Progammer's Reference Manual, L.P.A. Ltd., 36 Gorst Road, London SW11.

B. D. Steele (1981), EXPERT – The implementation of data-independent expert systems with quasi-natural language information input, M.Sc. thesis, London: Department of Computing, Imperial College.