Prolog and expert systems

Davies, Peter Leslie

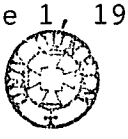
Prolog and Expert Systems

Peter Leslie Davies

A thesis submitted for the Degree of
Doctor of Philosophy
of the University of Durham

School of Engineering and Applied Science
(Computer Science)
University of Durham

June 1, 1987



Prolog and Expert Systems

P. L. Davies

School of Engineering and Applied Science

(Computer Science)

University of Durham.

ABSTRACT

The first part of the thesis provides an introduction to the logic programming language Prolog and some areas of current research. The use of compilation to make Prolog faster and more efficient is studied and a modified representation for complex structures is presented. Two programming tools are also presented. The second part of the thesis focuses on one problem which arises when implementing an Expert System using Prolog. A practical three-valued Prolog implementation is described. An interpreter accepts three-valued formulae and converts these into a Prolog representation. Formulae are in clausal form which allows disjunctive conclusions to rules. True and false formulae are stated explicitly and therefore the interpreter is able to perform useful consistency checks when information is added to the data base.

June 1, 1987

Acknowledgements.

This work was supported by the Science and Engineering Research Council.

I wish to thank my supervisor, Mr A. J. Slade, for his invaluable advice and timely prompting and also for allowing me the freedom to pursue my own interests. Thanks are also due to several anonymous referees who provided comments on the earlier versions of the three-valued Prolog implementation. In addition, I should like to thank the department for allowing me to use the text processing facilities and the laser printer.

June 1, 1987

Table of Contents

June 1, 1987

Dedicated to my future wife, Sarah.

June 1, 1987

Prolog and Expert Systems

1. Introduction.

This thesis is based on the logic programming language Prolog and the implementation of Expert Systems. The thesis is split into two parts. The first part will provide an introduction to Prolog, its syntax, semantic and its implementation. This includes some new ideas on the implementation of Prolog through an experimental compiler and the design of two programming tools. The second part focuses on a particular part of Prolog relevant to the implementation of Expert Systems. The problems are presented first and a three-valued implementation proposed. The first section in the second part gives an introduction to Expert Systems and outlines some previous work on three-valued logic. The next section puts forward the proposed three-valued system and how it is to be used with Prolog. The next section is the implementation details and presents several demonstration examples and expert systems using the three-valued Prolog implementation. The last section is the conclusion and further work.

June 1, 1987

2. Introduction to Prolog.

The logic programming language Prolog was developed around 1972 at Marseille by Alan Colmerauer as a system that incorporated Robinson's Resolution principle [Robinson1965] and a fixed backtracking strategy. It was originally designed for Natural Language processing but has now been used for symbolic integration, plan formation, computer aided building design, compiler construction, data base description and query, mechanics problems and drug analysis. (See [Kowalski1982] for details and references). There are now many research implementations of Prolog, for example,

micro-Prolog, [Clark1984]
IC-Prolog, [Clark1981]
C-Prolog, [Pereira1984a]
Waterloo Prolog, [Roberts1977]
York Prolog, [Spivey1982]
MProlog, [Domolki1983]
and DEC 10 Prolog, [Warren1979,Bowen1982a].

There are also many commercial implementations, such as,

Turbo Prolog, Arity Prolog, Prolog 1, Prolog 2, Quintus Prolog, UNSW Prolog and Prolog-86.

A reasonably complete list of implementations can be found in [Smith1986] and an introduction to Prolog can be found in several texts [Clocksin1981,Kluzniak1985,Colmerauer1985].

2.1. Syntax.

The syntax presented here is the DEC 10 syntax which is

used in [Clocksin1981,Kluzniak1985], and many implementations. The basic objects in Prolog are called terms. A term can be defined from a constant, a variable or a compound term. The definition of a constant includes integers such as 0, 1, 1000, -10 and atoms. Atoms are defined as a sequence of alphanumeric characters starting with a lower case letter or a sequence of characters delimited by single quotes. For example,

atom, void, a, 'A String', a987,

are all atoms.

A variable in Prolog as a sequence of alphanumeric characters starting with an upper case letter. A variable can also start with "_" and the special case of "_" by itself is called the anonymous variable. For example,

X, Type, A2, _, _Num,

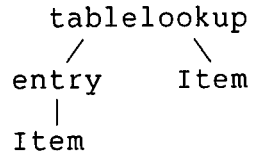
are all variables. The anonymous variable is used to match anything where the valued returned is not required by the programmer.

A compound term in Prolog represents a structured object and is made up of a functor and its arguments. A functor comprises its name, which is an atom, and its arity, which is the number of arguments for the functor. So

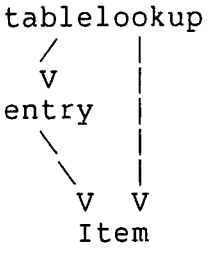
tablelookup(entry(Item), Item).

is a functor 'tablelookup' of arity 2 that has arguments 'entry(Item)' and 'Item'. The first argument is another

functor, 'entry', of arity 1 and argument 'Item' that is a variable. The second argument is a variable 'Item'. Generally, compound terms can be pictured as trees.



or more precisely as a directed graph since 'Item' corresponds to the same object.



If a compound term contains no variables it is referred to as a ground compound term.

To add to the readability of Prolog functors can be declared to be operators so that they can be written in prefix, infix or postfix form. This merely provides a more convenient and readable form. Standard operators such as '+', '*' '~' can then be written as,

```
X + Y,  P * Q,  ~ N
```

instead of the bracketed form,

```
'+'(X, Y),  '*'(P, Q),  '~'(N).
```

A Prolog program is made up of one or more sentences. A sentence is either a clause or a directive where a clause

can be with or without a body, such as,

```
head :- body.
head.
```

The 'head' is a term but excluding a variable or an integer and 'body' is a comma separated list of terms (where term is not an integer). Some examples are,

```
descendant(X, Y) :- offspring(X, Y).
descendant(X, Z) :- offspring(X, Y), descendant(Y, Z).

offspring(abraham, isaac).
```

A directive is a headless clause, such as,

```
?- body.
:- body.
```

Since lists are frequently used in Prolog an alternative syntax for a list has been developed. The original Prolog implementations used '.' as the constructor of a list and 'nil' to represent an empty list. The list a,b,c would be represented by,

```
'.'(a,'.'(b,'.'(c,nil).
```

or in infix form by,

```
a.b.c.nil
```

The DEC 10 Prolog introduced an alternative syntax which used square brackets and '[]' for the empty list. The above list would be written as,

```
[ a, b, c ].
```

Also for constructing lists the form '[A|B]' was introduced,

where A is the head of the list and B is the tail. This can give rise to strange expressions such as '[a,b|c,d]'. Work on a new standard is in progress. [Moss1986]. The DEC 10 Prolog also introduced the form "abc", (including the double quotes), to represent the list of ASCII characters, [97,98,99]. The exact syntax of certain implementations should be found in the appropriate user's manual. The syntax of micro-Prolog, [Clark1984], is significantly different since it is not based on the DEC 10 syntax, [Bowen1982a].

2.2. Semantics.

There are two ways of considering a Prolog program: declaratively and procedurally. The most appropriate way depends on the application. The simplest sentence to consider first is a clause that has no body, such as,

```
P.
```

(Here, uppercase is used to indicate any term). This is read declaratively as 'P is true' and procedurally as 'Goal P is satisfied'. Clauses with several goals, such as,

```
P :- Q, R, S.
```

can be read declaratively as

P is true if Q and R and S are true.

or procedurally as

To satisfy goal P, satisfy goals Q, R and S.

The declarative interpretation of the directive

?- P, Q.

is

Are P and Q true ?

and procedurally as

Satisfy goals P and Q.

The form ':- P,Q' is executed in the same way as '?- P,Q' except an answer is not expected.

The procedural interpretation of Prolog views the language as a program whereas the declarative interpretation treats it as a database. There is, therefore, a duality between program and database. A program can be seen as data and be manipulated by other Prolog programs. Likewise, a database can be executed where the query provides the execution entry point. The ability to manipulate a program as data to another program makes it extremely easy to define a meta language and write control programs for executing Prolog. A Prolog interpreter can be written in Prolog in as little as four lines. (Page 86, [Bowen1982a].)

2.2.1. Unification.

2.2.1.1. Resolution.

The resolution principle was put forward by [Robinson1965], as a new single inference process to prove sets of logical statements. He also developed the unification algorithm that generates the most general unifier between two

formulae. Using these two techniques a machine oriented logic was developed. The starting point of the inference is to deny the alleged conclusion and then prove that this is inconsistent with the initial statements. If S is the set of clauses that includes the denied conclusion, then S is unsatisfiable if and only if Rn(S) contains {}, for some n >= 0 where R(S) is the result of applying resolution to the set S.

Unification in Prolog is based on this work and provides a single mechanism for :-

1. passing parameters into and out of procedures,

2. constructing and accessing compound terms,

3. comparing and assigning variables.

The unification process finds the most general unifier between two terms i.e. the most general substitution for two terms to make them the same. For example, if the following was given,

```
?- goal(X, Y).
goal(A, 20) :- newgoal(A).
newgoal(15).
```

then Prolog would try to unify 'goal(X, Y)' with 'goal(A, 20)' and find a substitution where 'A' and 'X' are equivalent and 'Y' is the integer 20. This unification provides a connection between 'X' and 'A' so if 'newgoal' uni-

fies 'A' with the integer 15 then the values of 'X' and 'Y' are 15 and 20. In terms of a PASCAL type language, [Jensen1975], after unification 'X' and 'A' can be considered as names of pointers pointing to the same object.

Unification does not just deal with variables but any compound term. For example,

```
?- goal(structure(A, Rest)).

goal(structure(atom, substructure(A, B))) :-
        newgoal(item(B)).

newgoal(item(book)).
```

In this example, 'structure(A, Rest)' will be unified with the head of 'goal' matching 'A' with 'atom' and 'Rest' with 'substructure(A, B)'. Here, a new structure has been created that is referred to by 'Rest'. (The two occurrences of 'A' refer to different items since the scope of a variable is local to a clause and unification provides automatic renaming to avoid name clashes when unifying a goal with a head). The call on 'newgoal' unifies 'B' with 'book' after creating a new structure 'item' and then accessing the structure to unify it with 'book'. The result after both unifications is that 'A' is unified with 'atom' and 'Rest' is unified with 'substructure(X, book)'. (Where 'X' is a new unique variable name).

Variables in Prolog provide a powerful mechanism to refer to any object and therefore removes the need for low level concepts such as pointers. Effectively, unification

provides one mechanism for referencing, constructing, comparing and assigning objects. The assignment is non-destructive in that a variable is either unassigned or it has a value that cannot be overwritten. The assignment can be undone if required. (See backtracking). When a variable first occurs in a program it is uninstantiated, that is, it has no value or instance attached to it. When it is unified against another object it is said to be instantiated.

2.2.1.2. Occur check.

Robinson's definition of unification includes an occur check so that a term unified with another term that has a reference to itself, will fail. For example,

```
?- goal(X, func(X)).
goal(A, A).
```

should fail since a circular term will result. Most Prolog systems do not include an occur check so the unification creates a circular term. Once the circular term has been created great care must be taken not to cause an infinite loop. Writing out the circular term will cause an infinite loop.

The omission of the occur check in most Prolog systems was done deliberately. This is because unification with the occur check is linear on the sum of the sizes of the terms, whereas without, it is linear on the size of the smallest

term. This means that unification against a variable can only be made constant if the occur check is ignored. Since unification against a variable is common the occur check has to be ignored to make Prolog a practical programming language. Unfortunately, this does mean that Prolog has limited applications in theorem proving and can not solve some examples given in [Robinson1979b].

2.2.2. Backtracking.

During the unification of two terms it is possible that the unification fails, this requires the concept of backtracking. Also, Prolog procedures can be non-deterministic, that is, a procedure can return a result that is later rejected. The system must then backtrack to the procedure so it can return a different result.

When Prolog executes a goal, the search strategy is to search the list of clauses from the top, for a head that unifies with the goal. This could involve several unsuccessful attempts to unify with clause heads. When a successful head match is found the point of the match is noted so that if future computations fail the search can continue from this point.

The goals in the body of the successfully matched head are then executed from left to right. Each goal could introduce its own backtrack points so generally there is a stack of backtrack points. When a unification fails the

latest backtrack point is found and all the unifications done since this point are undone. (i.e. set the variables back to uninstantiated). For example,

```
?- goal(X).

goal(Y) :- a(Y), b(Y).
goal(Z) :- b(Z), c(Z).

a(1).
b(2).
b(3).
c(3).
```

The initial goal 'goal(X)' is matched against the clauses and matches 'goal(Y) :-..'. If this fails matching will start at 'goal(Z) :-..'. The leftmost goal for 'goal(Y)' is then executed and 'a(Y)' is matched with the clauses. The first clause that matches is 'a(1)' so 'Y' is unified with '1'. The next goal for the clause 'goal(Y) :-..' is then executed. This is the goal 'b(Y)' but 'Y' is already instantiated to '1' so the goal is 'b(1)'. The goal 'b(1)' does not match anything so the system backtracks to where 'a(1)' was matched. There are no other matches for this so the system backtracks to where 'goal(Y)' was matched. This backtracking has undone the unification of 'Y' with '1' so 'Y' is now uninstantiated.

The next clause to match 'goal(X)' is 'goal(Z)'. This then executes 'b(Z)' that matches 'b(2)'. The next goal is 'c(2)' that fails to match so the system backtracks to where 'b(Z)' matched, undoing the unification of 'Z' with '2'. The next clause to match 'b(Z)' is 'b(3)' so this is

returned and the goal 'c(3)' is executed. This matches 'c(3)' and there are no more goals left to execute so '?- goal(X)' returns with 'X' unified with '3'.

The backtracking used in Prolog is naive in that it always returns to the most recent backtracking point first. This can cause a considerable amount of unnecessary work in finding a solution. In the following, the computation for 'b(Y)' is repeated five times which is totally unnecessary.

```
?- back(X, Y).

back(X, Y) :- a(X), b(Y), c(X).

a(1).
a(2).
a(3).
a(4).
a(5).
b(Y) :- complex_calculation.
c(5).
```

This type of unnecessary repetition of calculations can be avoided by careful ordering of the goals. The work of [Warren1981] shows how goals can be ordered in Prolog queries so that the goal which will reduce the size of the search space most, is executed first. In the above example, a solution can be found immediately with no extra calculations if the goals of 'back' are ordered,

```
back(X, Y) :- b(Y), c(X), a(X).
```

Backtracking can also re-calculate goals that have backtracking points but which have nothing to do with the reason for the failure. A good example is taken from

[Bruynooghe1984b].

```
?- p(X), q(Y).

p(a).
p(b).
q(U) :- r(U, U).
r(V, W) :- s(V), t(W).
s(a).
s(b).
t(c).
t(d).
```

This example will fail but only after evaluating the goal 'q(Y)' twice. This type of behaviour is inefficient and has prompted work on intelligent backtracking. The work by [Bruynooghe1984b] attempts to find the culprit of the failure and then backtracks to that point. This involves finding minimally inconsistent deduction trees from the whole deduction tree. The work by [Cox1984] presents a method for finding alternative maximally consistent trees which then provides the information about an appropriate backtrack point.

2.2.3. Extra features.

Prolog provides the programmer with many built-in procedures or evaluable predicate. These can be grouped into input/output, arithmetic, comparisons, program manipulation, sets, debugging, meta-logical, extra control and definite clause grammars. The two areas that will be considered in more depth are the 'cut' which provides extra control information and definite clause grammars. The other areas are not considered relevant to this thesis.

2.2.3.1. Cut.

The 'cut' is used by Prolog to provide purely control information and therefore has no declarative interpretation. When a 'cut' is executed as a goal it always succeeds but if backtracking returns to the 'cut' it causes the parent goal to fail. It effectively 'cuts' out other solutions for the parent goal. For example, (The 'cut' is written as '!'),

```
a :- b, c, !, d.
a :- e, f.
```

If the first clause matches, and 'b' and 'c' produce a result, then the 'cut' removes any other alternatives for 'b', 'c' and 'a' (the parent goal). Therefore, after the 'cut', if 'd' fails then 'a' will fail. If, on the other hand, either of 'b' or 'c' had failed then the 'cut' would not have been reached and 'e' and 'f' could generate solutions.

The use of 'cut' can cause a program to have no suitable declarative reading. For example,

```
not(X) :- call(X), !, fail.
not(X).
```

In this example the ordering of the clauses is important and the second clause does not have a declarative reading of 'not(X) is true' because it depends on whether 'call(X)' is true or not. As far as an implementation is concerned it provides a useful way of removing unwanted alternatives for a goal. If a goal is found to have no other alternatives

then an implementation can make good use of this fact to recover storage used in the goal since it is known that this goal will never be reactivated.

2.2.3.2. Definite Clause Grammars.

This type of grammar was proposed by [Colmerauer1978] for use with a Prolog system. The grammar is formed from a 5 tuple relation of the form { F, Vt, Vn, Vs, --> } where F is a set of functional symbols containing "." and "nil", Vt is the set of terminal symbols, Vn is the set of non-terminal symbols satisfying Vn intersection Vt = 0, Vs is the set of starting non-terminals such that Vs subset-of Vn and --> is a rewriting relation on V*. This grammar can be expressed in different forms depending on the syntax of the Prolog in use and also the grammar can be provided as an in built part of Prolog or a routine can be written in Prolog to translate the grammar into Prolog clauses. (See [Clocksin1981]). As shown in [Colmerauer1978] and [Clocksin1981] this form of grammar provides more than a context free grammar and can be very useful in writing grammars for compilers and natural language processors. Three separate grammars are used in the compiler to perform the lexical analysis, the syntax analysis and then the synthesis of the machine code. The extensions of definite clause grammars from context-free grammars are :-

1. Any terminal or non-terminal symbol can be any Prolog term,

2. True Prolog goals can be interspersed in the grammar rule body,

3. The left hand side of the grammar rule can contain a non-terminal followed by a sequence of terminals.

The syntax of a grammar rule is similar to Prolog clauses except ':-' is replaced by '-->'. For example,

```
head --> body.
```

or

```
variable(Table, Info, Id) -->
   space, uppercase(C), reststr(Rest),
   { tablelookup(Table, string(C, Rest), Info, Id) }.

reststr(string(C, Rest)) -->
   alphanumeric(C), reststr(Rest, !.
reststr(eos) --> [].

uppercase('A') --> "A".
uppercase('B') --> "B".
 etc
```

The second example is taken from the Prolog compiler, (See next section), and is the definition of a variable. The goal between '{}' is an extension of type 2 and is not part of the grammar but a call on a normal Prolog clause. Terminals are written in list form, such as "terminal" or [a].

The grammar rule notation is a syntactic extension of Prolog clauses and can be translated directly into Prolog clauses. A grammar rule implicitly takes two extra arguments which are the input string and the output string. Terminals match items on the input string to leave the output string. Translating the grammar rules involves adding

the extra arguments. For example,

```
p(X, Y) --> q(X), r(X, Y).
```

becomes

```
p(X, Y, S0, S) :- q(X, S0, S1), r(X, Y, S1, S).
```

Terminals make use of a clause defined as

```
'C'([X|S], X, S).
```

that takes item 'X' from list '[X|S]' to leave 'S'. So, for example,

```
uppercase('A') --> "A".
```

becomes

```
uppercase('A', S0, S) :- 'C'(S0, 65, S).
```

Any goal that appears between '{}' is just left untouched.

2.3. Modularity.

Many Prologs lack any form of modularity so programs are considered as one large flat program with all names visible. Work has been done to introduce modularity into Prolog, [Domolki1983]. The version of Prolog presented in this paper allows modules to be created with import and export lists. It also allows data names to be either symbolic or not. If a data name is symbolic then the sequence of characters that make up the name is significant, otherwise the name has no character representation. A more theoretical approach to providing modules and generics is given in [Goguen1984]. Micro-Prolog, [Clark1984], also

provides a module facility. Turbo Prolog, [Borland1986], provides a mechanism for declaring predicates as global, all others being local to the module being compiled. This allows a program to be split up into several modules which are compiled separately.

2.4. Typing in Prolog.

Another feature of Turbo Prolog is that it requires programs to be explicitly typed. These types are call domains. Turbo Prolog provides six basic domains: integer, real, character, string, symbolic and file. All other domains are built up from these and declared in a 'domains' declaration section. The declarations are similar to the 'type' declarations in PASCAL, [Jensen1975]. To declare a list of integers Turbo Prolog would require

```
domains
        integerlist = integer*
```

where '*' represents zero or more of the preceding domain. Turbo Prolog then has a 'predicates' declaration section where each predicate is declared to use arguments from a given domain. For example,

```
predicates
        append(integerlist, integerlist, integerlist),
```

declares the predicate 'append' to take three arguments which are lists of integers.

From the programming point of view this typing enforces a more strict design of predicates and more information is

present in the source code, aiding readability. From the implementation point of view parameters can be passed directly and space saved by compacting the representation. Speed can be improved by direct manipulation of the data instead of always consulting the type of data first. Since the types are stated explicitly it simplifies the interface to other languages. If a predicate is declared to have two arguments of type integer then the external routine can expect to receive two integers on the stack.

Also from the implementation point of view, a predicate can be declared to have alternative types. (append could be declared to work on lists of integers and lists of symbols). In this case the compiler can generate specific code for each alternative type and compile in the calls to the appropriately typed predicate.

2.5. Parallelism.

There is much interest in the execution of Prolog on parallel architectures and as a concurrent language. Parallel execution can be divided into two sections: 'and' parallelism and 'or' parallelism. 'And' parallelism executes processes that will produce results relevant for one solution. 'Or' parallelism executes processes that will produce results for alternative solutions. [Conery1981]. The work by [Clark1985,Gregory1986], provides an implementation of a language called PARLOG. This language features both 'and' and 'or' parallelism and differs from Prolog in three

respects :-

1. Don't care non-determinism.

2. 'and' parallel evaluation.

3. Mode declarations for shared variables.

There is also work on Flat Concurrent Prolog [Shapiro1986].

2.6. Definition of Horn and Clausal form.

Prolog is based on Horn clause logic where rules and facts are expressed by a set of Horn clauses. Horn clauses are a subset of clausal form which will be defined first since it is used later in the thesis.

2.6.1. Clausal form.

Clauses in clausal form are expressions of the following form

A1 or A2 or ... or Am if B1 and B2 and ... and Bn.

'A1 ... Am' are the head of the clause and 'B1 ... Bn' the body of the clause where 'A1 ... Am, B1 ... Bn' are atomic formulae and m >= 0, n >= 0. An atomic formula is defined by

p(t1, ... , tq)

where 'p' is a q-ary predicate, q >= 0 and 't1, ... , tq' are terms. A term is defined by

f(t1, ... , tr)

where 'f' is a r-ary function symbol, r >= 0 and 't1, ... , tr' are terms. When m = 0 the clause is a headless clause and can be read as

attempt to satisfy B1 and B2 and B3 ....

2.6.2. Horn Clauses.

Horn Clauses are defined as clausal form where 'm' can only have values 0 and 1. (Horn Clauses will be written using a different syntax, which is like Prolog, to distinguish between the two forms. 'if' = ':-', 'and' = ','). A Horn Clause is an expression of the form

```
A :- B1, ... , Bn
        or
:- B1, ... , Bn.
```

'A, B1, ... , Bn' are atomic formulae as defined above.

Considering the form 'A :- B1, ... , Bn'. When 'n = 0', the clause reduces to 'A' and is called a fact. When 'n > 0' the clause is a rule which has a body that must be satisfied before the head is satisfied. The headless clause ':- B1, ... , Bn' is a request to satisfy the body immediately. This form is used to query the set of clauses.

2.7. Converting Standard to Clausal Form

The book by [Clocksin1981] presents an algorithm to convert standard first order logic into clausal form which

can then be used by a Prolog interpreter. The first stage is to remove the implications using the equivalences

```
A -> B  equivalent to  (~A) or B
A <-> B equivalent to  (A and B) or (~A and ~B)
```

The next stage is to move all negations inwards so that negation only appears in front of a literal.

```
~exists(A, B) becomes all(A, ~B)
~all(A, B) becomes exists(A, ~B)
~(A and B) becomes ~A or ~B
~(A or B) becomes ~A and ~B
```

The next stage is to remove existential qualifiers by introducing Skolem constants

```
exists(X, A(X) and B(X, C)) becomes
A(g1) and B(g1, C) where g1 is a unique constant.
```

Now the universal quantifiers are moved outwards so that every variable is universally quantified and therefore the quantifiers can be removed and all variables are assumed to be quantified. The next stage is to distribute "and" over "or" so that the clauses are in conjunctive normal form and the last stage is to put the formula into clausal form by putting all the literals into cl(A, B) where A is the list of literals that are not negated and B is the list of negated literals but without their negation. Then using the equivalence :-

```
A <- B  equivalent to  A or ~B
```

this puts the formulae in clausal form. Provided the list A contains either one element or no elements then the clause can be converted into Prolog by writing the list A first,

followed by a ":-" and then the B list separated by ",".

2.7.1. Quantifiers

When the standard form of logic is converted into clausal form then at some point the existential quantifiers have to be removed and replaced by Skolem constants or functions. Sometimes valuable information is lost when Skolemising occurs and so some systems use logic stored in an existentially quantified form. [Shapiro1979] takes this one step further and introduces numerical quantifiers as well. The first quantifier introduced is the maximal quantifier which is used to represent statements like "every one has a maximum of one mother". The minimal quantifier is then introduced but to be of use the world size must be known otherwise the situation where every condition except the minimum number is known, and therefore other objects must satisfy the formula, would not be detected. These two forms are then incorporated into a general numerical quantifier which also includes the special case of the existential quantifier.

3. Implementation issues.

The following sections present methods for representing complex terms, variable classification, indexing, tail recursion, compilation, garbage collection and speed tests.

3.1. Structure representation.

When implementing a Prolog interpreter or compiler a suitable representation of complex terms has to be chosen. The basic requirement is that multiple instances of a term appearing in the source text can be created and undone rapidly. For example, if the following clause was called several times there would be several instances of the structure but with possibly different variable values.

```
fact(structure(X, str(Y, Z))).
```

There seem to be two different approaches to this problem. One is structure sharing, [Warren1977], and the other makes use of concrete copying, [Mellish1980].

3.1.1. Structure sharing.

The structure sharing approach represents a structure as a pair

```
< source term, frame >
```

where 'source term' is a pointer to a representation of the source term that has relative offsets for the values of variables. The variables are stored in an area pointed to by 'frame'. For example, the above structure in 'fact'

would be represented by the source term

```
ptr1:   structure/2
        X1
ptr2:   str/2
        X2
        X3
```

where Xn is an offset for variable 'n'. An instance of this structure would be represented as

```
< ptr1, . >
        |
        V
        X, < ptr2, . >
                   |
                   V
                   Y, Z
```

where X, Y and Z are storage for instances of X, Y and Z. The main advantage of a structure sharing method is the concise representation of complex structures and the single occurrence of constant data for multiple occurrences of the structure. The cost of constructing new instances of a structure is proportional to the number of distinct variables in the source terms. The disadvantage is that references to the variables of a structure must go through an extra level of indirection whereas a direct representation would not.

3.1.2. Non structure sharing.

An alternative to structure sharing for Prolog interpreters has been suggested by [Mellish1980]. This method involves taking a concrete copy of the structure, when a new one is created, with the appropriate variables set. For

example, the above structure would have storage for all the variables allocated together. When one variable was instantiated, the matching structure would be copied to a new storage area with the variables linked. Using the example above and unifying it with '?- fact(A).' the structure sharing approach would produce a structure such as,

```
/* fact(structure(X, str(Y, Z))). */
/* ?- fact(A). */

Local            Global            Heap

A * *-------------------------> structure/2
  |                  source term  0
  ------------> X                 str/2
      frame     Y                 1
                Z                 2
```

Whereas the alternative would produce

```
/* fact(structure(X, str(Y, Z))). */
/* ?- fact(A). */

Local            Global
A *------------> structure/2
X *------------> nil
                 str/2
Y *------------> nil
Z *------------> nil
```

The main reason for using this alternative approach is that the storage of a <source term,frame> pair requires space for two addresses. On an address wide machine where only one pointer will fit in a word, this is wasteful. The alternative method only needs one address per word so the basic variable can be stored more efficiently. The advantages of this alternative method are that structures are not created unless it is necessary. With the structure sharing approach it can not be determined whether an operation will access an

existing structure or construct a new one. Therefore, structure sharing always allocates space for a new structure even if one is not created. Also, because the structure is directly represented then the speed of access should be improved. The disadvantages are that there needs to be a 'local' variable for every variable in the structure including multiple copies of the same variable. Also, there can be multiple copies of the functor name and other constant information when multiple copies of the structure are created. A more complete analysis of the two approaches is given in, [Mellish1980].

3.2. Variable Classification.

If a Prolog clause exits deterministically then the local storage used in the clause can be reclaimed. If, however, the clause is non-deterministic then the storage can not be recovered because computation may return to this clause and the variables should still be accessible. This means that it is important to know when a clause is deterministic so space can be recovered. This is part of the power of the 'cut'.

If a structure is created in a clause and this is passed out of the clause then the variables that appear inside the structure can not be considered local because they are still accessible from the rest of the program. These variables are usually allocated space on a different stack called the global stack. The other variables are

allocated on a local stack. Space used by a clause on the global stack can not be recovered when the clause exits deterministically, but space is recovered on both stacks when backtracking occurs. This means it is very important to classify as many variables as possible as local. The structure sharing approach needs to classify variables as either local or global at compile time since the offsets to the variables in the source term are calculated at compile time. Generally, any variable that appears inside a structure must be made global. Although, using mode declarations, this can be reduced to variables that appear inside the literals for constructed source terms. The alternative approach can classify variables at run time since all the variables have fixed locations on the local stack. This is an improvement, although variables are still made global in a constructed structure even if it is not passed out of the clause.

To allow backtracking to be implemented, variables that have been assigned values should be noted so that backtracking can reset these variables to uninstantiated. The area of storage used for this purpose is referred to as the trail. It is normally a stack structure since backtracking removes one execution frame at a time and therefore the corresponding trail frame is popped off. If variables are classified as local and global then some simple tests on the variables can be used to reduce the amount of trailing necessary. If it is known that the variable just assigned

will be removed on backtracking anyway, there is no need to trail the assignment. This occurs if a value is assign to a local variable in a deterministic clause because when the clause fails the storage for the variable will be removed anyway. Also, to reduce the amount of trailing a seniority is assigned to variables. The most senior variable is the one that will be removed last on backtracking. This is normally determined by the relative position on the stack. Since local variables could be removed on deterministic exit they are always more junior than global variables. When two variables are unified it is important to assign the junior to the senior and not the reverse. This minimises the amount of trailing and stops long chains of references to a single variable.

3.3. Indexing.

To increase the performance of some Prolog systems indexing has been introduced so that large databases can be searched efficiently. This facility is transparent to the user since it appears that the database is search sequentially. The Prolog system by [Warren1977], indexes on the predicate name and the first argument. Therefore, if there is a large database of names and telephone numbers with a query to find the telephone number of a given name, then the Prolog system can index on the predicate name and the name of the person. This will probably only give a few matches and should be found in a time independent of the name. A

non indexing search would be dependent on whether the name was at the top or the bottom of the database. As a side effect of the indexing it is easier to detect when the last match has occurred so the procedure can return deterministically recovering its local storage. The Prolog interpreter written by [Clark1979], has indexing on the predicate name and all of the arguments.

3.4. Tail Recursion.

Prolog relies heavily on recursion so the overheads due to this must be kept to a minimum. When a procedure exits deterministically the local stack frame can be recovered on exit. The technique of tail recursion optimisation makes use of the fact that the space can actually be recovered before the last call on the procedure provided the previous goals have no backtracking points left. This means that a recursive procedure such as concatenate,

```
concatenate([],L,L)
concatenate([X|L1],L2,[X|L3]) :- concatenate(L1,L2,L3).
```

can recover storage before the recursive call on concatenate, instead of waiting until the procedure returns. Although only the local stack frame is recovered it does means that any references to global variables that would disappear when the procedure exited, will disappear before the last call. Therefore garbage collection will be able to recover the unreferenced storage earlier. The tail recursion optimisation provides not only a decrease in the

storage required but it can also provide some speed improvements. This is because some of the stack frame information for the current goal can be left in place when the tail procedure overwrites the frame. Some work in creating a new frame can therefore be avoided. The paper by [Warren1980], explains how tail recursion optimisation is built into the DEC 10 Prolog system. [Bowen1982a].

As [Warren1980] points out, this technique effectively transforms recursive procedures into an iterative implementation. The programmer can therefore write recursive procedures without bothering about the overheads introduced by recursion.

3.5. Compilation.

The idea behind compilation is that the full power of unification may not be needed to match a goal against a given head of a clause. The general algorithm for unification can therefore be specialised for a particular head. This specialised code can then be made more efficient. For example, when matching any goal against the first occurrence of a variable all that is necessary is to assign the matching term to the variable. For compound terms there are two cases. The first is accessing an already instantiated structure and the second is building a new structure. If information is provided about how the clause will be called, (in particular which arguments are guaranteed to be instantiated and which are guaranteed to be uninstantiated), then

the compiler can specialise the structure accessing/constructing code.

3.6. Mode Declarations

When writing a compiler, [Warren1977], found it useful to add mode declaration information to a program so the compiler could determine whether an argument to a predicate was always instantiated or always uninstantiated. He used information such as,

```
:- mode pred(+,+,-,?).
```

to declare a predicate called 'pred' of arity four with the first and second arguments as input (always instantiated), the third argument as output (always uninstantiated) and the last as either input or output. Without this information a compiler must create code which can access an existing structure or create a new structure. The information allows the compiler to generate only one alternative and also avoid code which tests the state of the argument. An added advantage of the mode information is that less global variables are needed. This is because a variable occurring inside a structure which would normally be global can be made local if the structure is known to be instantiated insuring a new structure will not be created.

In his paper, [Mellish1981] argues that providing this information by hand can be difficult, involve considerable work if the code is modified and error prone. To allow the

compiler to still have access to this information he proposes an automatic generator of mode declarations. The generation of modes is a two stage process. The first analyses the clauses and builds up a dependency graph for all the argument positions. The second stage uses this dependency graph to propagate constraints about argument positions through the graph. When the graph reaches equilibrium the constraints at each argument position are converted into mode declarations.

To create the dependency graph for one clause a complete history of each variable is generated and the argument positions analyses before and after the call on the predicate. This is so information is available about variables appearing in predicates after the call on that predicate. The mode information will be derived from the information about argument states before the call.

Once the dependency graph has been generated for each clause the state of each argument is unknown (corresponds to mode '?'). Starting from one particular constraint, such as argument X is an integer, this information is propagated through the graph using a set of rules for combining the constraints. This continues until the graph stabilises. Once all the constraint information has been processed the information at each argument position is mapped into the three modes useful for the compiler.

3.7. Garbage Collection.

The life time of storage on the local stack is either until determinate exit or until backtracking. On the global stack, storage is only recovered on backtracking, so it is sometimes useful to perform some kind of garbage collection. Global storage can become inaccessible if all the references to it are removed due to local storage being recovered on exit. For example, the following will produce an inaccessible global structure.

```
create :- struc(X).
struc(func(X,name)).
```

Methods for recovering global storage are discussed in [Bruynooghe1984a] and [Warren1977]. The basic idea is to trace and mark all the accessible locations and then compact the storage by moving the accessible locations to the bottom part of the stack. All the references to the global locations have to be remapped to point to the new locations.

Some implementations do not include garbage collection, [Pereira1984a,Spivey1982] but rely on the user using programming tricks such as,

```
not(not(X)).
```

to check a call on 'X'. 'not' is implemented by failure so all the storage is recovered after goal 'X' returns. See [Kluzniak1985] for other techniques.

3.8. Speed.

To test the speed of Prolog implementations a benchmark has been developed which attempts to measure the number of logical inferences per second an implementation can achieve. This is commonly called LIPS. Some rough estimates of speed for some implementations are :- (The tests run using UNIX† were when the operating system was not being used by other users)

micro-Prolog on Z80 CP/M	300 LIPS.
C-Prolog on VAX UNIX	800 LIPS.
Own Test using LIPS program	
C-Prolog on SUN UNIX	2000 LIPS.
Own Test using LIPS program	
Quintus Prolog on SUN UNIX	22000 LIPS.
["Ted%nmsu.csnet@CSNET-RELAY.ARPA"1986]	
Turbo Prolog on IBM PC/AT	9000 LIPS.
Own Test using LIPS program	
UNSW Prolog on IBM PC/AT	500 LIPS.
[Fischer1986]	

Test were run using the LIPS program given in [Meier1986]. (See appendix).

Apart from the raw speed of an implementation, once a program has been written critical areas can be improved by paying attention to the structures used. The following demonstrates how a list reverse program can be dramatically improved.

```
/* Reverse program for lists taken from
   "Prolog for programmers"
   by Kluzniak and Szpakowicz chapter 4 pp132
   Represent lists as 'nil' and list(Head,Tail).
 */
```

† UNIX is a trademark of Bell Laboratories.

```
/* Naive reverse program */
reverse(nil, nil).
reverse(list(A,Tail), TA) :-
        reverse(Tail, T), attach(T, A, TA).

attach(nil, A, list(A,nil)).
attach(list(B,Tail), A, list(B,TA)) :-
        attach(Tail, A, TA).


/* Reverse using a stack structure */
reverse2(L, LReversed) :-
        reverse2a(L, nil, LReversed).

reverse2a(nil, Stack2, Stack2).
reverse2a(list(A,Tail), Stack2, Final) :-
        reverse2a(Tail, list(A,Stack2), Final).


/* Reverse using difference lists */
reverse_d(d_list(nil,nil), d_list(Y,Y)).
reverse_d(d_list(L,nil), d_list(list(An , X), Y)) :-
        reverse_d(d_list(L,list(An,nil)), d_list(X,Y)).
reverse_d(d_list(Z,Z), d_list(Y,Y)).
reverse_d(d_list(L,Z), d_list(list(An , X), Y)) :-
        reverse_d(d_list(L,list(An,Z)), d_list(X,Y)).
```

Some example test figures for a 60 element list

	reverse	reverse2	reverse_d
trail	496	16	264 bytes
global	37072	1672	2156 bytes
local	68072	2232	1952 bytes
time	2.75	0.14	0.54 seconds
	2.62	0.15	0.55 seconds

From these figures it can be seen that reverse2 is considerably better than the naive reverse for both time and space.

4. Implementation of a Z80 Prolog compiler.

This section describes the implementation details of a Prolog compiler written for a Z80 based microprocessor system, [ZILOG1980]. It was written to provide in-depth knowledge of a Prolog implementation so that work on possible language extensions were feasible. The Z80 is powerful 8-bit machine and is a very widely used microprocessor. An idle Z80 machine was available with 10M bytes of hard disc and a suitable assembler and loader. It was decided that the compiler should be able to bootstrap itself so the development could be carried out on another machine and then downloaded. The code must therefore be fast so that the compiler can be bootstrapped and library programs can be developed in Prolog which would usually need to be written in a lower level language. The space requirement is critical since most Prolog programs require over the 64K bytes of space provided by the Z80 address range.

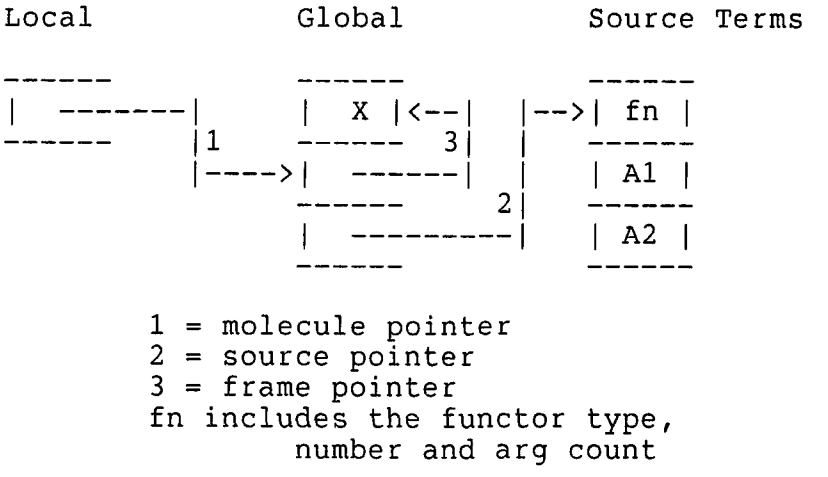
4.1. Structure Sharing.

Two possible ways of representing complex terms created in a Prolog program will be considered. These are the approaches outlined in the previous section on structure representation.

The paper by [Mellish1980] provides a comparison between the two methods for a small word machine. The result is that the two methods are reasonably similar in

space requirements with a slight bias towards direct representation. A more compact method of structure sharing than the method used by [Mellish1980] was developed so structure sharing was used. No comparison of the finished code was made with the direct representation method. Also the work by [Warren1977] on Prolog compilers used a structure sharing approach so the design of a new compiler would be easier with this method.

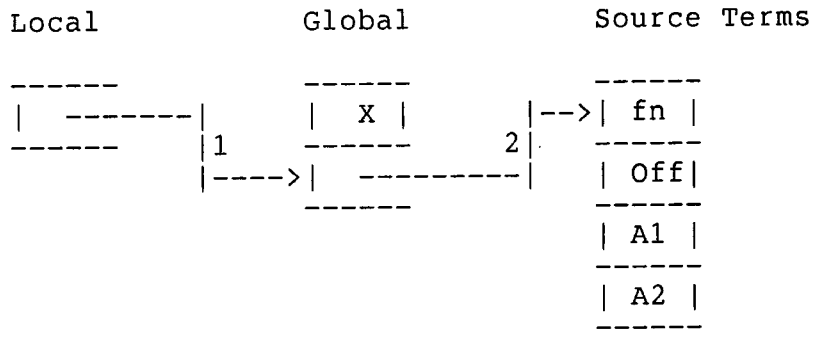
The Z80 is an address wide machine in that one address will fit into a machine word. This causes some problems in representing complex terms as described in [Mellish1980]. The structure sharing approach that he uses is as follows :-



1 = molecule pointer
2 = source pointer
3 = frame pointer
fn includes the functor type, number and arg count

This uses two global stack locations to hold the frame and source pointers. The new method only puts one item on the global stack for each molecule and stores an offset in the source term. The offset gives the position of the beginning of the frame relative to the molecule pointer. This monopolises on the fact that the relative position of

the start of the frame is always the same for a given source term.

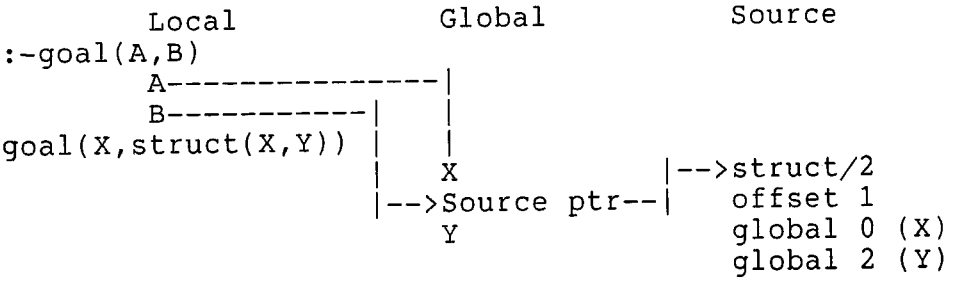


Off is the offset from 1 to give the frame pointer

For example,

```
:- goal(A, B).
goal(X, struct(X, Y)) :- next(X).
```

The frame for the call on 'goal' would just contain space for two local variables. The frame for the clause with 'goal' at its head would have space for three global variables, the second being the source pointer, and a source term with offset of one. Every time 'goal' is called the positions of X and Y in the frame are always fixed hence the offset of one from the molecule pointer. The following is the stack frame after unification.



If the offset requires the same amount of storage as

the frame pointer in the previous method and only one molecule is generated from each source term then the space required by each method would be the same. In practise there are usually several molecules created from each source term so the saving is one global stack location for each molecule minus one. As [Mellish1980] points out in his paper, for the examples he tested the number of variables allocated at any one time is generally greater than twice the number of molecules. It therefore takes more space to make a variable's space large enough to hold two addresses than to allocate two extra locations on the global stack for molecules and have one address per variable. The offset approach is more efficient than the two addresses per variable if the number of variables allocated is greater than twice the number of source terms. For the 'goal' example above, [Warren1977] would require four variables at two addresses each i.e. eight, the structure sharing by [Mellish1980] would require four variables plus two extra for the molecule i.e. six and the offset approach would require four variables plus one for the molecule and one for the offset i.e. six. If there were two active calls on 'goal' the Warren approach would require (4*2)*2, Mellish approach would require (4+2)*2, and the offset approach would require (4+1)*2+1. This space efficiency is for the representation of complex terms and it may be the case that the representation of atoms and integers requires two address size variables. In which case, the analysis is more complex.

On the Z80 implementation, the offset is stored in an 8-bit location whereas the frame pointer would have to be a 16-bit location. This means each frame must be only 255 bytes but this has not proved a limitation with practical programs. These two factors together should provide a space improvement for some programs over the direct representation method used by [Mellish1980].

The choice of representation for atoms and integers was made so that each item would fit in 16-bits. Since it must be possible to distinguish between a reference and a basic type the top bit in a 16-bit word was used to represent a reference. This therefore limited the data areas (global and local stack) to 32K bytes with the program code, trail and literals in the other 32K bytes.

4.2. Compiler Description.

To keep the space requirements down for the executable code it was decided that the code produced by the compiler should be basic Prolog machine code instructions not in-line Z80 code. This could then be interpreted by a small (~1K bytes) interpreter written in assembler. This approach would also make it possible to produce code for a different machine and run it using a different backend interpreter. The basic code chosen was the PLM instructions defined by [Warren1977] since this defined a working system in reasonable detail. The implementation of the PLM instructions in assembler was found to be reasonably easy and the whole

interpreter is under 1500 bytes. This is the only part that will need to be in memory to execute a compiled program.

A full description of the PLM instructions are given in [Warren1977] but a brief outline will be given here. A group of clauses such as :-

```
a :- b , c.
a :- d.
```

would be compiled to the following code.

```
        enter
        try(label(a1))
        trylast(label(a2))

label(a1):
        neck(0,0)
        call(b)
        call(c)
        foot(0)
label(a2):
        neck(0,0)
        call(d)
        foot(0)
```

The 'enter' routine is responsible for setting up part of the new environment and the 'neck' instruction completes the environment. 'enter' only creates the parts of the environment that are necessary for the unification of the head. The rest of the environment necessary for the body is postponed until the head is unified in the hope that it will be unnecessary if the unification of the head fails. The 'foot' instruction discards the current environment if it is no longer needed (i.e. the clause is deterministic) and continues at the 'continuation point' which is set by a 'call()'. The 'try' instruction creates a 'backtracking

point' so if the call fails the next instruction after the 'try' is executed. The 'trylast' is an optimisation of 'try' for the last clause. If the last clause fails the 'trylast' instruction has arranged for the previous back-track point to be the one that is used. The unification instructions are placed between the 'label()' and the 'neck' instructions. For example,

```
equals(X,X).
```

would be

```
        enter
        trylast(label(enter1))
label(enter1):
        uvar(0,local,0)
        uref(1,local,0)
        neck(1,0)
        foot(2)
```

The arguments to 'uvar' and 'uref' (unify variable and unify reference) are the argument number, the type of variable (local or global) and the variable number. Here the variable 'X' is number zero and of type 'local'. The 'uvar' instruction is a special case of 'uref' that is used when the variable is unified for the first time. In this case it is known that the variable is undefined and therefore some initialisation and testing can be avoided. Other unification instructions are 'uatom(), uint(), and uskel()'. These unify an atom, an integer and a skeleton.

The compiler itself was written in Prolog so that bootstrapping would be possible. It would also provide a useful test for the compiler if it could compile itself and

run correctly. The compiler structure is built around the grammar of Prolog expressed in definite clause grammar notation. The code is produced by a three stage process,

[1] The source code is translated to PLM code in a single pass. This means that the code has no optimisation for the last call in a group of clauses.

[2] The second stage scans the PLM code and converts the last call for a clause. The PLM instruction 'try(label)' is converted to 'trylast(label)'.

[3] The third stage is the backend which either outputs the code as PLM instructions or converts the instructions to Z80 code and calls on the Z80 PLM interpreter subroutines.

The code is built up in an internal list that is then output by a machine dependent backend. The machine dependent code is therefore collected into one section.

Although the first stage of the compiler produces code in a single pass, a considerable amount of code is generated after the clause has been parsed. For example, when the clause,

```
clause(X, ....) :- ...
```

is parsed, at the point when the variable 'X' is processed the compiler does not know whether to produce code for a local, global, temporary, or void variable. This is only

known when the end of the clause has been reached. A similar problem occurs when generating the 'neck' instruction since this must know how many variables are in the clause, including the body that has not been parsed yet. The symbol table therefore has an information field for each variable which indicates its type. The code for a variable is then generated using this type. At the end of a clause the variable symbol table is scanned and the types determined. For the 'neck' instruction markers are inserted into the symbol table which have references to the maximum number of local and global variables. When the table is scanned at the end of a clause the number of each type can be determined. This method is relatively easy using Prolog variables but would be complex in a language that used explicit pointers.

Once the grammar had been defined it was transformed to make the parsing more efficient. The improvements are achieved by moving multiple calls that would fail later into one call. For example,

```
term --> constant.
term --> predicate.
term --> variable.

predicate --> constant, arguments.

arguments --> argument, restofarguments.

restofarguments --> [].
restofarguments --> andop, argument, restofarguments.
```

In this syntax the two calls on 'constant' are not needed. If a predicate term was to be matched, 'term' would initially find the constant but then fail later because there

were some arguments. 'term' would then have to match 'constant' again before matching the arguments. A better syntax would allow 'constant' to be matched and then match zero or more arguments.

```
term --> predicate.
term --> variable.

predicate --> constant, arguments.

arguments --> [].
arguments --> argument, restofarguments.

restofarguments --> [].
restofarguments --> andop, argument, restofarguments.
```

The library routines for the compiler were written in Z80 but kept as small as possible. The routines included only basic operations like character input/output, addition/subtraction and var/nonvar. These were found sufficient to bootstrap the compiler and more complex routines written in Prolog could be built from these. For example, writing out integers can be written in terms of single character output.

When the compiler produced code, the names of atoms and predicates were defined as local to the compilation. This meant that to interface to the library routines an explicit interface description had to be provided. For predicates it was decided that the names of the external routines would be provided in an external declaration of the form,

```
external(plc, get, put, add, var, nonvar, ... ).
```

The syntax could have been written in any form but a

description that was also a valid Prolog fact was considered an advantage. The program could then be 'consulted' by any other Prolog implementation without removing the interface description. The interface was kept as simple as possible since no other information about the predicate was provided. (such as arity, mode of arguments, argument types, etc). The only information provided by the external declaration is the relation between the predicate name and the internal value given to the predicate.

To write out atoms and complex terms the external routines must be able to find the source name that corresponds to the internal value. To overcome this problem the compiler builds a symbol table at the end of the code so external routines can find the source name. Because the compiler does not know which atoms and functors are needed externally, the symbol table contains all the strings used in the source code. This is wasteful and with hindsight it would have been better to include the atoms and functors explicitly in the external declarations so only those that are required are put in the symbol table. This new external declaration would take the form,

```
external(plc(_,_), get(_), put(_), atom1, ... ).
```

This now also includes the arity of the predicates which should help in error checking between routines. It should be possible to extend this form to allow separate compilation of different modules.

4.3. Testing and Evaluation of the Compiler.

The compiler that ran under C-Prolog on a VAX 11/750 was tested with a suite of 24 test programs. These tried to test most aspects of the implementation. The code produced by the VAX compiler was down loaded onto the Z80 machine which then assembled, loaded and ran the code. After testing and correcting with all these programs the compiler itself (with slight modifications for file handling) was compiled and this down loaded onto the Z80. When this was running successfully the same suite of test programs were compiled on the Z80 and run. This provided a double test in that it was testing the compiler implementation and the code it produced. Nine of the test programs ran correctly but 15 of the test programs were too big to compile on the Z80.

The space required by the implementation seems to be reasonably efficient although the space required by the compiler when running on the Z80 is disappointingly high. This might be reduced by re-writing the way the compiler produces code, for example, compile the code clause-at-a-time instead of reading all the source, then compiling all the code, then outputting all the code.

To give some indication of the implementation's performance the tests run on Micro-Prolog by [Liardet] were used. The first test program does list reversal using a 30 element list.

Space requirements in bytes

	Z80 Prolog	Micro-Prolog	C-Prolog
global stack	4652		9424
local stack	5664		16896
trail	1830		1768
total data	12146	19K	28088
code	712	1K	796
literals	272		
library	1410		57560
interpreter	1418	29K	68676
total code	3812	30K	127032

The second test program also comes from [Liardet] and performs a quick sort on a 50 element list.

	Z80 Prolog	Micro-Prolog	C-Prolog
global stack	2892		5984
local stack	3683		10316
trail	734		852
total data	8309	12K	17152
code	1062	1K	1316
literals	432		
library	1410		57560
interpreter	1418	29K	68676
total code	4332	30K	127552

The library and interpreter sizes are not very useful for comparison because the Z80 compiler only provides simple I/O and arithmetic where as C-Prolog provides numerous built in predicates.

The speed tests use the same two programs as used for the space test. The Micro-Prolog tests were run using a 2 MHz Z80 cpu and the Z80 Prolog compiler tests were run with a 2.4 MHz Z80 cpu.

	Z80 Prolog	Micro-Prolog	C-Prolog
reverse	1.5 sec	3.5 sec	1 sec
quick sort	2 sec	5 sec	2 sec

Since the times for the Z80 compiler and C-Prolog are small

and therefore inaccurate when timed by hand another test was run comparing the times to compile one of the test programs

	Z80 Prolog	C-Prolog
compile program 'struct.pl'	18 seconds	10 seconds (*)

(*) This is with a VAX 11/750 running UNIX 4.1 in multiuser mode but with no other jobs running.

This therefore puts the speed of the Z80 Prolog compiler at approximately 450 LIPS.

5. Programmer's Tools.

This section will describe two tools developed for use with Prolog. Both tools can be used when developing Expert Systems and also for general Prolog programming. The first tool aids diagnosing a program and the second provides some code improvements.

5.1. Connectivity.

The complete set of clauses in a database can be considered as a connected graph with the connections between clauses representing possible unifications. (See [Kowalski1979]). If such as a graph is constructed, clauses with no connections can be detected. These isolated clauses can usefully be used in generating information about possible errors in the set of clauses (or program). A Prolog program was written that generated a list of every clause head and a list of every goal. It then scans through these lists in two passes. The first pass detects any goals that have no head to match and the second pass detects any clause heads that are never called. As a side effect of pass one, a list of all the system calls used by the program is generated. When considering whether a Prolog program is portable between one implementation and another this list should be useful.

This information can be used to detect several types of errors.

[1] goals or heads where there is a difference in the number of arguments. For example,

```
:- checkswitch(OldPos, NewPos).
checkswitch(on) :- ...
checkswitch(off) :- ...
```

Since the program has definitions of all the system calls it can find where system calls are called with the wrong number of arguments. For example,

```
:- write('Debug level ', Level).
```

[2] goals or heads that contain non unifiable arguments due to incorrect types. For example,

```
:- enter(item(Info), Table).
enter(entry(Info,Ident), Table).
```

[3] it can be used to detect rules in the three-valued implementation, (defined in the second part of this thesis), that have been specified as three-valued but are actually being used in a two-valued way. For example,

```
true((rule1 if part1 and part2(T)
        and writetext(T))).
true(writetext(T)) :- write(T).

        transforms to

true(rule1) :- true(part1),
        true(part2(T)), true(writetext(T)).
false(part1) :- false(rule1),
        true(part2(T)), true(writetext(T)).
false(part2(T)) :- false(rule1),
        true(part1), true(writetext(T)).
false(writetext(T)) :- false(rule1),
        true(part1), true(part2(T)). (*)
true(writetext(T)) :- write(T).
```

After the transformation the fourth rule (*) is redundant and not connected to any other rule because 'writetext' is a two-valued rule.

The list of clauses that are never called can indicate a group that is not called because of an error or that the clause is to be used at the top level only. If the clauses are not to be called at the top level and the program is error free then these clauses are superfluous and can be removed. This is used to good effect by the next program.

5.1.1. Example.

Test code used

```
test :- write(hello,world),
        systemcall([X,Y,Z]),
        test2(clause(func,X)).

test2(clause(func2,X)).
```

Sample output when processing the above code.

```
Script started on Tue Jul  8 11:03:46 1986
% prolog
C-Prolog version 1.5
| ?- ['connect.full'].
bagofall consulted 896 bytes 0.383334 sec.
connect.full consulted 7328 bytes 3.66667 sec.

yes
| ?- listing(test).

test :-
   write(hello,world),
   systemcall([_7,_8,_9]),
   test2(clause(func,_7)).

yes
| ?- listing(test2).

test2(clause(func2,_10)).
```

```
yes
| ?- connect.

System calls used :-
!/0
==/2
+/1
==/2
call/1
erase/1
fail/0
functor/3
nl/0
read/1
recorda/3
recorded/3
retract/1
setof/3
sort/2
true/0
write/1
No clause found for goal write(hello,world)
No clause found for goal systemcall([_3658,_3659,_3660])
No clause found for goal test2(clause(func,_3658))
No calls found for clause connect
Retract connect ?no.
No calls found for clause test
Retract test ?yes.
test:-write(hello,world),systemcall([_37937,_37938,_37939]),
        test2(clause(func,_37937)) retracted.
No calls found for clause test2(clause(func2,_3761))
Retract test2(clause(func2,_3761)) ?yes.
test2(clause(func2,_3761)):-true retracted.

yes
| ?- connect.

System calls used :-
!/0
==/2
+/1
==/2
call/1
erase/1
fail/0
functor/3
nl/0
read/1
recorda/3
recorded/3
retract/1
setof/3
sort/2
true/0
```

```
write/1
No calls found for clause connect
Retract connect ?no.

yes
| ?- halt.

[ Prolog execution halted ]
%
script done on Tue Jul  8 11:06:26 1986
```

5.2. Middle-Out Processing.

In a similar way to the connectivity program this processing can be understood by considering the set of clauses as a connected graph. (See [Kowalski1979]). In his book, Kowalski describes top-down, bottom-up and middle-out processing as a method for solving problems. In this section a program is described that performs a defined subset of the middle-out processing before execution of any query or program takes place. It can therefore be viewed as pre-execution code improvement. Initially, the program was written to perform macro-processing so that uniquely defined facts could be preprocessed. So, for example,

```
maxlines(20).

input(Data,Lines) :-
        maxlines(Max), Lines < Max, readline(Data).
```

would be transformed into,

```
maxlines(20).

input(Data,Lines) :-
        Lines < 20, readline(Data).
```

The criterion used to define the transformations is that the fact only appears once in the database so during

execution it would be matched deterministically. Note that 'maxline(20)' has now become superfluous and this will be detected by the connectivity program which can then remove it.

The macro processor was then extended to expand any rules where the head of the clause is matched deterministically. The call is then expanded to the list of goals in the deterministic clause. For example,

```
a :- b, c, d.
c :- f, g, h.
```

would be expanded to

```
a :- b, f, g, h, d.
c :- f, g, h.
```

After this expansion the clause for 'c' could become disconnected and could be removed by the connectivity program. When variables are involved in the clauses then the expansion is more difficult but unification takes care of most of the details. The program makes sure that the clause it is working on is a copy of the clause in the data base and has not been modified by any previous unifications. Otherwise, the program would be trying to solve a clause rather than just match two isolated clauses. An example of clauses with variables follows,

```
body(Ctbl,Vartbl,Neck,[],HiLoc,HiGlb,L,L) -->
   emptyifop(Neck,HiLoc,HiGlb,Vartbl).

emptyifop([init(Gstart,Gend)|[localinit(Lstart,Lend)|
   neck(HiLoc,HiGlb)]],HiLoc,HiGlb,Table) -->
      { tablelookup(Table,marker(neck),
        info(Gstart,Gend,Lstart,Lend),_) } .
```

is transformed to

```
body(Ctbl,Vartbl,[init(Gstart,Gend),localinit(Lstart,
   Lend),neck(HiLoc,HiGlb)],[],HiLoc,HiGlb,L,L) -->
      { tablelookup(Vartbl,marker(neck),
        info(Gstart,Gend,Lstart,Lend),_) } .
```

Prolog includes as part of the language several "non logical" features such as cut, var, assert etc. These features are very difficult to handle properly in the above programs. For the middle-out processor the 'cut' causes problems because its effect is local to the clause it is in, so altering the goals in a clause alters the scope of the 'cut'. For example,

```
a :- b, c, d.
c :- f, g, !, h.
```

is not equivalent to,

```
a :- b, f, g, !, h, d.
```

because the 'cut' now effects the solutions returned by 'b'.

Another example of incorrect processing is the 'var' predicate when it is used for manipulating open lists. (See [Kluzniak1985]). If the code for entering an item into the last free location in a list is defined as,

```
lookup(Item,Entry) :- var(Entry), equals(Item,Entry).
equals(X,X).
```

Then this will be processed to produce

```
lookup(Item,Item) :- var(Item).
```

which attempts to unify argument one with argument two before there is any test to see if argument two is a variable. The initial clause will succeed if argument two is variable and argument one is anything, whereas the processed clause will only succeed if argument one and argument two are variables. A solution to this problem is to check if the clause about to be moved contains any "non logical" goals and if it does, abandon the current clause.

5.2.1. Performance.

To test the performance of the middle-out program a large Prolog program was selected which was written before the middle-out program. If the program was written before the middle-out program then the style will not have been influenced by any thought of how the program might be improved by an automatic process. The program that was selected was the Z80 Prolog compiler written to produce intermediate code as defined in [Warren1977]. This program is split into two sections. The first produces the intermediate code and the second converts this to Z80 microprocessor machine code, [ZILOG1980]. Code improvements concentrated on the 420 lines of code that comprised the first section. The compiler was run compiling the quick sort

program written by [Warren1977]. The original compiler and the modified compiler produced exactly the same output code so there was no functional change in the compiler.

5.2.1.1. Original Compiler.

Clause space (Heap + Atom)
= 22624+6184 = 28808 bytes

Stack space (Global+Local+Trail)
= 41594+14308+1612
= 57514 bytes

Runtime (Three interleaved runs)
= 19.55, 20.31, 19.65 sec

5.2.1.2. Modified Compiler.

Clause space (Heap + Atom)
= 22212+5180 = 27392 bytes

Stack space (Global+Local+Trail)
= 43640+14308+492
= 58440 bytes

Runtime (Three interleaved runs)
= 16.97, 16.85, 17.00 sec

For the runtimes there is a constant time for the second pass which was timed at

7.10, 7.05, 7.74 seconds

Therefore the average time for the original first pass is 12.54 seconds and for the modified first pass 9.64 seconds. This represents approximately a 25% improvement in speed. The space required to store the clauses has been slightly reduced but the runtime storage is slightly increased. The total storage requirement for the modified compiler is actu-

ally slightly less (86322 vs 85832).

Some interesting features are that the amount of trail storage has been cut by a third which is presumably due to fewer clauses being called. The amount of global storage has increased which is probably due to the interpreter classifying more of the variables in the larger clause bodies as global. Generally, the space differences between modified and unmodified program will be dependent on the type of program but a speed improvement should be achieved. A class of programs which will cause the middle-out program to significantly increase the size of the program are the ones which are deterministic and have many subroutines used from many places. For example,

```
a :- b, c, d.
a :- c, d.

b :- g, h, i.

c :- j, k, l.

d :- g, h, j, k.

g :- m, n, o, p.

h :- m, n, p.

i :- p, n, m.

j :- m, n.

k :- o, p.
```

Since every clause except 'a' is deterministic this will be expanded to

```
a :- m, n, o, p, m, n, p, p, n, m, m, n, o,
     p, l, m, n, o, p, m, n, p, m, n, o, p.
a :- m, n, o, p, l, m, n, o, p, m, n, p, m, n, o, p.
```

The real run time for the middle-out program to modify the compiler was 1 hour 19 minutes and then the connect program was run to remove disconnected clauses. This requires the operator to know which clauses are to be executed from the top level and takes approximately ten minutes of operator time. Since the middle-out program is run by the C-Prolog interpreter, [Pereira1984a], which has a speed of 700-1000 LIPS and there are Prolog compilers with speeds of at least 10000 LIPS then the execution time for processing the compiler could be cut by a factor of ten to 8 minutes. This makes the processing time much more acceptable.

5.3. Bagofall.

In [Warren1982b], he suggests an extension to Prolog so that all solutions to a predicate can be collected in a list. This has been implemented in C-Prolog as two built-in predicates, 'setof' and 'bagof'. The 'setof' predicate, 'setof(X,P,S)', is read as "'S' is the set of all instances of 'X' such that 'P' is provable". In this case any variables which are not specified in 'X' are assumed to be free and therefore 'setof' can produce several solutions on backtracking. The 'bagof' predicate is similar to 'setof' except it is unordered and can have duplicates. During the writing of the above utilities it was found that an exten-

sion to 'setof' and 'bagof' would be very useful. This is the ability to generate the set or bag of all the solutions to a query without knowing the variables contained in the query.

The utilities required the program to find a head of a clause and then find all the clauses that match that head. i.e. all the solutions of 'clause(X,Y)'. To be able to use 'setof' or 'bagof' the variables appearing in the head of any clause would have to be known in advance which is impossible. The new variant of 'setof'/'bagof' that was defined using similar low level code was 'bagofall(P,S)' where 'S' is the set of solutions to 'P' assuming all variables in 'P' are bound. Taking the example of 'setof' from [Pereira1984a] to demonstrate 'bagofall'.

```
Example of Setof

:- setof(X, X likes Y, S)

gives

Y = beer,  S = [dick, harry, tom]
Y = cider, S = [bill, jan, tom]
```

whereas the bagofall predicate gives

```
:- bagofall(X likes Y, S)

S = [dick likes beer, harry likes beer,
        tom likes beer, bill likes cider,
        jan likes cider, tom likes cider]
```

Since there is no way of connecting items in the resulting bag with any variables in the query, 'bagofall' returns the set of whole clauses with any variables instantiated.

6. Introduction to Expert Systems.

This second part of the thesis moves on from the implementation of Prolog to using Prolog. The area studied is Expert Systems. This section gives some background to Expert Systems and gives some indications where Logic Programming, and in particular Prolog, fit in to the Expert System area. There are eight subsections : Current Expert systems, Requirements for an Expert systems, Knowledge Representation, The Knowledge Base, Inference techniques, Inference with Uncertainty, Searching Techniques, and Learning Techniques.

6.1. Current Expert systems.

Examples of Expert systems are many and varied but as explained in [Buchanan1982] the present systems are quite limited. His report on the subject gives a list of current Expert systems together with his views on the current state of the art and areas for future research. The article by [Gevarter1982] gives a list of 17 Expert systems and the purpose and methods used by each. From this list it can be seen that Expert systems are used for a very wide range of diverse applications. He also gives examples of Expert system tools which can be used in the construction and maintenance of an Expert system and gives a list of the research he thinks is required in the area.

6.1.1. Knowledge Transfer from an expert.

This is commonly recognised as the bottleneck in the development of an Expert system, taking around 5 to 10 man-years, and several approaches have evolved. The first is to develop a program that interactively extracts the knowledge from the domain expert and fits it into the knowledge (or data) base. An example is the TEIRESIAS system. (See [Suwa1982]). Another approach is to acquire the knowledge through self learning or discovery. (See Learning Techniques). When transferring knowledge from the domain expert to a knowledge base the expert must provide explicitly the context that the data is to be used in. [Spiers1983] explains that what is explicit today for one person might be totally incomprehensible tomorrow because the conceptual framework which was implicitly assumed before has now been lost. A system which could construct and adapt its own conceptual framework would obviously be advantageous.

6.2. Requirements for an Expert system.

6.2.1. Explanatory powers.

A key feature of Expert systems is their ability to explain their reasoning. This feature is of great importance since in making decisions the operator must have confidence in the results that the system produces. This confidence will be gained if the system can explain how it

arrived at the decisions. Being able to explain its reasoning requires a control structure that is sufficiently complex to solve the problem satisfactorily but also not too complex for the explanation mechanism to explain or for the domain expert to understand. The domain expert must be able to understand how the system will perform to be able to integrate his knowledge into the system successfully. The work of [Wallis1982] is concerned with providing satisfactory explanation powers for MYCIN, [Shortliffe1976], which could previously only cite the appropriate rule. He argues that explanation powers are necessary for four reasons:-

(i) to be able to examine the system if errors arise,

(ii) to assure the user that reasoning is logical,

(iii) to persuade the user that unexpected advice is appropriate and

(iv) to educate the user.

To be able to provide reasonable explanation the system must have some concept of the users ability so that the explanation can be geared to the users expertise. This includes the users goals and social role. The extra information that was found to be necessary includes the systems ability to judge a rules complexity and importance. This is so rules that are complex but unimportant need not be explained and rules which are important but not complex should be explained first. The areas the author suggests for further

work include the systems ability to determine the reason for the users enquiry and reasoning with the context of the dialogue.

6.2.2. Transparency.

This implies the user can have access to the knowledge base and see what knowledge is being used. The transparency of a knowledge base is linked to the explanatory power in that it is an aid to user confidence and understanding of the system.

6.2.3. Structure.

6.2.3.1. Separate Knowledge base and Problem Solver.

One of the first lessons learned from the study of Expert systems was that the domain specific knowledge should be separated from the problem solving mechanism (or Inference Engine.). The introduction to the work by [Leith1983] expands on this idea and puts forward the reasons.

6.2.3.2. Knowledge sources.

This is a possible structure where several cooperating Experts work together. [Drazovich1982] uses this structure in his Expert for object identification. The hypothesis for the current object is represented in a hierarchical structure and for each level in the hierarchy there is a cooperating Expert. For each Expert to function correctly it must be able to access knowledge used and created by

other Experts. This gives rise to the blackboard structure for knowledge representation where global knowledge is put on a "blackboard" where every Expert can access it. This system uses data driven inference to build conclusions from arriving data but it can also use goal driven inference by placing the goal on a data-based call-back list. If the data arrives the goal is activated, if not the goal is never activated.

6.3. Knowledge Representation.

6.3.1. First order Predicate Logic.

Knowledge in the knowledge base can be represented by 1st order predicate logic. The usefulness of this form is explored in [Kowalski1979], where many arguments are put forward in favour of logic. Hayes, Deliyanni and Kowalski (Referenced in [Kowalski1982]) have argued that natural language representation schemes based on such structures as semantic networks, frames and scripts can usefully be reformulated in symbolic logic.

6.3.2. Semantic Nets.

This is a general structure where the knowledge is organised around objects. These objects are represented by the nodes in the semantic net and the arcs between the nodes represent the relation between the objects. In his book, [Kowalski1979] points out that semantic nets can be represented in logic by an N-ary relation representing the

arc with the nodes as arguments to the relation. This has a disadvantage in that the information about one object is spread about the data base but all the relations can be kept together. [De1982] use semantic nets to represent knowledge for an office environment. The semantic net structure is expanded by the use of views, activities and hierarchies. These additions help in the retrieval in that similar concepts are grouped together. The authors propose a semantic net represented by a 5 tuple relation, (C, F, V, A, H). C is a finite set of nodes or concepts, F is a finite set of arcs or modeling functions, V is a finite set of views, A is a finite set of activities and H is a finite set of hierarchies. To access the semantic net the operator will be involved with certain activities at a certain level in the managerial hierarchy and from there he will have a certain viewpoint. The semantic net is therefore structured to group together the information that is relevant to each operator.

6.3.2.1. Viewpoints.

In his paper, [Barber1982] describes the viewpoint mechanism, which he considers as similar to situational calculus and contexts but with the advantage of being objects within the system that can be reasoned about. Information is only ever added to viewpoints so to change a viewpoint a new one is created with the changed information. The changed information is then recorded in a data structure (

E.g. property lists - LISP or records - PASCAL). When an inconsistency occurs in the system the inconsistency is quarantined to one viewpoint by explicitly keeping track of what is believed to be true.

6.3.3. Production Rules.

The structure of a production rule is most easily likened to an "IF condition THEN action" structure. This is the form that is used in MYCIN. The whole knowledge base consists of these production rules which are then invoked by pattern matching. Pattern matching is the invocation of the action part of the rule when the condition part can be satisfied. [Leith1983] proposes that the production rules should be hierarchically structured so that some form of context for the invocation can be achieved. This is important for explanation mechanisms since the context is presented explicitly. Also the interpreter can search the rules as if they were in a tree and therefore have less rules to consider at one time. The advantages of production rules are that the whole system is very modular with knowledge represented in a consistent manner. Also each rule represents a single chunk of knowledge and therefore helps in maintenance and changing. The disadvantages are,

(i) the rules are not called explicitly and therefore there may be unexpected side effects,

(ii) knowledge cannot always be easily expressed as a

production rule and therefore the simplicity sacrifices the easy of expression and

(iii) the desired sequence of consultation can not always be easily mapped into production rules because of backward searching.

6.3.4. Frames - deficiencies of Production Rules.

Frames were developed in response to the concept that thinking is driven by expected structures. The frame holds the expected values and the rule it is associated with. Frames are similar to scripts developed by [Lehnert1980]. As stated in [Nau1983], Nilsson has pointed out that frames can be translated into 1st order logic. Explicit representation of knowledge has become an important part of Expert systems and [Aikens1983] found that the production rules used in MYCIN were not explicit enough so developed a system that uses frames and production rules. The system, called CENTAUR, is a pulmonary physiology Expert designed to demonstrate the structures. The frame system is able to provide the context and function of the production rule and also provide a control structure which is sensitive to the initial data. This results in a more focused consultation. It was found in an earlier system, called PUFF, which only used production rules that

(i) it was difficult to represent prototypical knowledge,

(ii) adding and modifying rules was difficult,

(iii) altering the order of the requested information during a consultation was problematic and

(iv) the system could not explain its reasoning satisfactorily.

The control structure for the consultation is also expressed in frames and can therefore be used to direct the consultation. The explanation mechanism can also explain the control structure, if necessary, since it is explicit.

6.3.5. Data Pools.

This concept is explained in [McDermott1983] and is similar in many respects to the Frame representation. The data pools hold information that can be copied into new data pools but changes to the original data pool are also inherited by the copy. This structure helps in maintaining the consistency and completeness of the knowledge base.

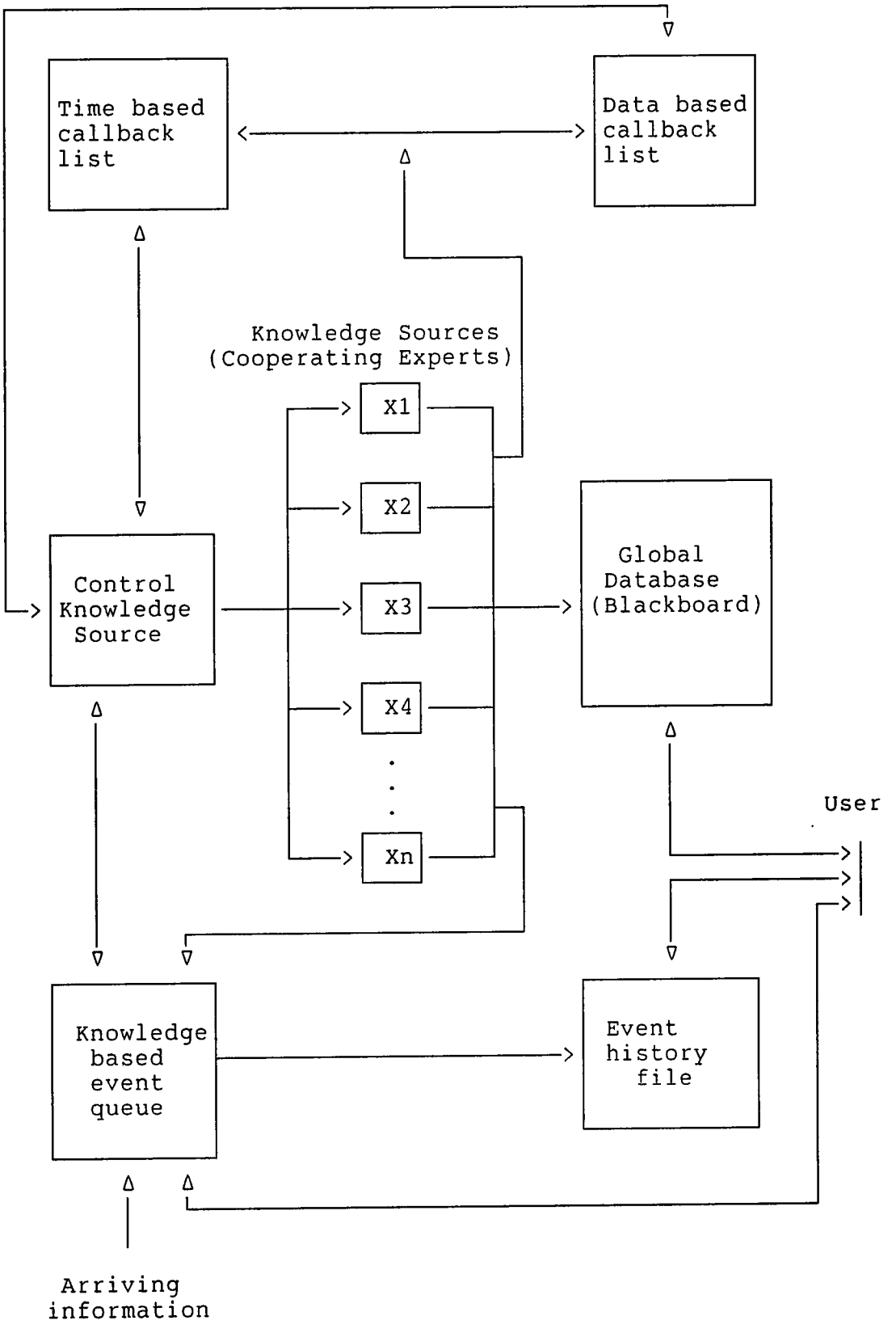
6.3.6. Blackboard scheme.

This is the framework used in HEARSAY II (See [Nau1983]) and the system developed by [Drazovich1982]. A possible structure is given in the diagram on the next page.

6.3.7. Fuzzy sets.

Zadeh, referenced in [Efstathiou1979], suggests that humans think in fuzzy sets, for example a "set of chairs" or

Possible multi expert and blackboard scheme structure



a "set of small chairs". The operator on the set, (small in this case), is called a hedge. These hedges can be of two types which are operators such as very, more, less, etc or descriptors such as technically, strictly, etc. Using these sets truth functions can be worked out and the system used to represent knowledge. At present there are several problems in this area of which the method used to combine sets is one. When the fuzzy sets are large the storage requirements for calculating the fuzzy result set is massive. [Kopriva1983] proposes an algorithm that does the calculation more efficiently, using similarities between the calculation and tree pruning. The evaluation tree becomes a minimax tree which can then be pruned using alpha - beta pruning. (See Searching Techniques).

6.4. The Knowledge Base.

This is the collection of information that the problem solver will use in the course of its inference. Using one of the representations described above knowledge is collected together and structured. One of the main features of Expert systems is that they work with many different types of knowledge and the degree of structure required is low. The types of knowledge that can be used are, for example, facts, theorems, heuristics, equations, rules of thumb, assumptions, strategies, tactics, probabilities, advice and causal laws. This is in contrast to conventional programming method that rely on more concrete facts.

6.4.1. Completeness and Consistency.

As a knowledge base increases in size it becomes increasingly important to have a structured approach to the completeness and consistency of that knowledge base. For large knowledge bases it might be considerable time before the errors are noted by day to day use so a formal approach is required. The article by [Suwa1982] is an attempt to clarify some of the issues and suggest relevant directions. The area covered only considers the completeness of the information and not that the program interprets the data correctly so the correct interpretation must still be checked carefully. The checks that are suggested are

(i) logical conflict, where two rules succeed with different results,

(ii) redundancy, where two rules succeed with the same result and

(iii) subsumption, where two rules succeed but one has additional restrictions on its use.

For completeness missing rules have to be checked for. The program TEIRESIAS by Davis, R. is used to check the rules used by MYCIN, [Shortliffe1976]. An added aid in this area is the explanatory power of the Expert system (See Explanatory powers).

6.4.2. Truth Maintenance under dynamic change.

When a knowledge base is changed dynamically then the problem of truth maintenance must be considered. If, for example, a section of knowledge is changed and many other sections are dependent on the changed knowledge then there will be a finite time when the propagation of this change is not completed and the knowledge base is therefore inconsistent. Work on truth maintenance is considered in Doyle, J. which is cited in [McDermott1983] under his work on Data Pools (See above).

6.5. Inference techniques.

Inference describes the process whereby the input data is used in conjunction with the knowledge base to produce the Expert system's conclusions. In conventional programming the inference process is the sequential execution of the instruction statements with loops and calls. For Expert systems a new approach is used whereby inference is carried out by a search and a pattern matching routine. The search is used to find the part of the program that matches a certain pattern and when a match is found that part is executed. [Nau1983] gives several examples of programming languages that use pattern driven invocation of programs. These are Planner, Conniver, Prolog and ARS. The inference can be seen as the chaining of rules to form a line of reasoning. This chaining can be forward from the set of conditions to the conclusions, called forward chaining, or it can

be from the conclusion back to the conditions, called backward chaining. (See Searching Techniques).

6.5.1. Goal driven.

This is basically a top down inferencing approach where the starting point is an initial goal. This is then hierarchically subdivided down until a match with the input data is found. This goal (or model) driven approach is termed as a backward search (See Searching Techniques). An example of an Expert system that uses this approach is MYCIN. The HEARSAY II system referenced in [Nau1983] uses both goal and data driven inference. It achieves this by using multiple knowledge sources which each act as a problem solver. The knowledge sources communicate with each other via a blackboard scheme. (See Blackboard scheme.)

6.5.2. Data driven.

This is basically a bottom up inferencing approach where the starting point is the input data. This data (or antecedent) approach is a forward search producing new concepts from the old ones. This is then hierarchically abstracted to produce higher level concepts. An example of a program that uses this approach is BACON by [Langley1982]. It also uses expectation driven inference to decided what sort of structure it is expecting in the data. The program is used to discover empirical laws for summarising data such as the ideal gas law from data relating pressure, tempera-

ture and volume. Initially, the program finds relations between two variables then recurses to a higher level. To keep the system as general as possible the expectation heuristics are derived from previous discoveries the system has made and not domain dependent knowledge. The author hopes to extend the program to discover qualitative laws as well as quantitative laws. The DENDRAL system referenced in [Nau1983] is another example of a system that uses data driven (forward search) inference. The system creates plans which are used to generate possible solutions which are then tested for validity. (See Generate and Test.) It also uses problem reduction to reduce the search space. (See State space versus Problem reduction.)

6.5.3. Expectation driven.

This term is explained in [Gevarter1982] and refers to the inference technique where the inference moves from an abstract concept to a less abstract concept and is therefore generating an expectation of the hierarchy of the concept.

6.5.4. Event driven.

When the choice for the next step in the inference depends on the new data or the last problem solving step then the inference is called event driven. This is similar in some respects to forward chaining except the data or situation is evolving over time and therefore must take account of data as it arrives. This is used for real time

operations.

6.5.5. Generate and Test.

This is a principle by which the solutions to the problem are created. In abstract terms there are two communicating processes one of which generates a candidate solution to the problem and the other which tests the validity of the suggestion. If the suggestion is not appropriate then another candidate solution is generated. This method is very easy to implement in Prolog since the backtracking method used to get the next candidate solution is built into the system as the control strategy.

6.5.6. Inference as a Search.

Since most problem solvers use non-deterministic methods the process of finding the correct solution can be considered as a search for the solution. This therefore draws together the methods of problem solving and tree searching. The methods used to search the tree can be expressed in the knowledge base and therefore vague rules of thumb can be used by the problem solver to reduce the search space. This is called a heuristic search.

6.6. Inference with Uncertainty.

There can be several sources of uncertainty in a knowledge base which should be taken care of by the problem solver. Firstly, the actual knowledge base can have errone-

ous rules or some necessary rules could be missing. This should be overcome once the knowledge base is well tried and tested. Secondly, the input data to the problem solver can be erroneous to a certain degree or the data could be unavailable. This type of uncertainty is quite common and methods of treating it are put forward in several Expert systems (MYCIN [Shortliffe1976], PROSPECTOR [Gaschnig1982] and INFERNO [Quinlan1983]).

6.6.1. Subjective Bayesian reasoning.

The Bayesian reasoning is based on Bayes Theorem which is:-

$$P(Ej|D) = \frac{P(Ej).P(D|Ej)}{\sum_{j=1}^{m} P(Ej).P(D|Ej)}$$

where
- Ej is a state
- P(Ej) denote the prior probability
- P(D|Ej) probability of D occurring if the state is Ej
- P(Ej|D) the posterior probability after receiving the information D

6.6.2. Belief / Disbelief measures.

6.6.2.1. Certainty Factors (CF).

This is the method used in MYCIN [Shortliffe1976] to keep track of the validity of the data that the problem solver is working with. The CF is a single value in a given range (e.g. -1 to 1) which represents the validity. The CF of a conjunction of several facts is the minimum of the

individual facts. For the conclusion, it is the CF of the premise multiplied by the CF for the rule. The CF of a fact produced from more than one rule is the maximum of the rules yielding that conclusion.

6.6.3. INFERNO.

This is a probabilistic system developed by [Quinlan1983] and tries to overcome the deficiences of the methods used in the Expert systems MYCIN and PROSPECTOR. In his article he points out that, with the certainty factors used in MYCIN, there is a problem with the accuracy of the numbers: a certainty value of 0.5 could mean 0.5 +0.001 or +0.3. Other deficiences that he points out are the combining of probabilities that are not shown to be independent and, in most cases, are usually not independent. The system he proposes is based purely on probabilities, so that accuracy is taken care of implicitly, and well-founded inferences so that all probabilities are assumed to be dependent unless explicitly stated. This approach can be thought of as cautious but all results are mathematically sound. The system uses well-founded inferences so it is possible to show when inconsistences arise in the data. This gives rise to a system which deals with propagating constraints.

6.6.3.1. Propagation constraints.

The INFERNO system uses two values to represent the certainty of a value and these values are propagated to the

other associated relations. This then propagates through the system and inconsistency is detected if the true and false values add up to be greater than one. The system can then work back to suggest where the inconsistency came from and what is necessary to rectify it.

6.6.4. Truth maintenance.

When a contradiction occurs in a knowledge base there must be some way of undoing the inferences made from the knowledge involved so that when the contradiction is solved the knowledge base can be corrected. One method is to keep a record of the beliefs made from lines of reasoning so that the beliefs can be backed up and removed if a contradiction occurs.

6.7. Searching Techniques.

When a problem has a small search space then an exhaustive search of the whole tree can be used to find the solution but when the problem has a larger search space an efficient searching technique is necessary to overcome the combinatorial explosion which is found in most real applications. There are two basic methods which either involve an efficient way of searching the state space or a way of transforming the state space into smaller manageable chunks which can be searched efficiently.

6.7.1. Trees.

Two types of search strategies for game playing trees have been proposed. The first method (A) is to establish the whole game tree to a certain depth, evaluate the bottom nodes and then use a minimax algorithm. The second method (B) is to evaluate each successor node and establish subtrees starting with the n best successors. Then perform minimax on the subtrees. Method B is often referred to as Ni best forward pruning which chooses at level i the best Ni branches. The advantages and disadvantages of each, that are put forward in [Mero1983], are:- method B offers the only possible way to diminish the combinatorial explosion, method A can be modified by alpha - beta pruning and method B is spoiled to a greater extent than A by unreliable evaluation functions.

6.7.1.1. SSS*.

The paper by [Roizen1983] proposes a new minimax algorithm called SSS* which is compared with the alpha - beta algorithm. The SSS* algorithm is a non-directional algorithm which traverse the nodes in best first fashion similar to A* and never evaluates a node skipped by alpha - beta. The paper basically shows that SSS* and alpha - beta have the same growth rate and can therefore be regarded as asymptotically equivalent. Because of the only meager improvement in pruning power and the substantial bookkeeping required the author speculates that alpha - beta will still monopolise applications of minimax search.

6.7.1.2. Bidirectional.

Bidirectional search is a technique for searching a graph where the search direction is not limited to one direction. It basically involves methods for Backward and Forward chaining, so that improved efficiency can be achieved. The search heuristics for one such method are explained in [Champeaux1983].

6.7.2. Forward chaining.

This technique is used when the starting point is either a basic concept or data. This is then hierarchically abstracted to the conclusion. The search tree therefore has the initial data or basic concept at the root of the tree and the possible conclusions are at the leaves. The search involves finding a path from the root to a leaf which fits the constraints of the problem.

6.7.3. Backward chaining.

When the starting point is a goal or a hypothesis then the technique that is used is backward chaining which involves breaking down the concept into more basic parts until a match with the input data is found. Using a tree to represent the search, the goal or hypothesis is the root of the tree and the possible input data values are at the leaves. The search then finds the path from the root to the leaf which matches the input data.

6.7.4. State space versus Problem reduction.

The methods of state space searching are described above but an alternative approach is to reduce the problem by methods including divide-and-conquer. Systems that use this approach include GPS and STRIPS (Stanford Research Institute Problem Solver). When a problem is reduced into sub problems, the interactions between the sub problems must be catered for. Using constraint propagation introduces a method of moving the information between the sub problems and can therefore help the problem solver in deciding where to search next by following a line of least commitment. The least commitment policy is to move the focus of the problem solver between sub problems as the data becomes available. When there is insufficient information then an heuristic must be used to continue the search and if that line of reasoning is inappropriate then dependency directed backtracking, i.e. backtracking to where the decision was made, is used to follow other lines of reasoning.

6.7.5. Depth first versus breadth first.

When a tree is searched, nodes can be selected in either a depth first or breadth first manner. The depth first search picks one node at the level below the root and follows that to the next level down and repeats until it reaches a leaf or a specified depth. That node is then evaluated. In breadth first search, nodes at the level below the root are evaluated before nodes of a lower level.

The two methods can be interspersed if a depth is specified for the depth first search and when that is reached then nodes at that level are searched in breadth first manner. A special form of this kind of search is the depth first iterative deepening search. This form does a depth first search at maximum depth of one first. It then starts again with a depth first search at depth two. Then depth three and so on. This could seem inefficient but [Korf1985] has proved that it is asymptotically optimal in terms of time, space and cost of solution path for exponential tree searches.

6.7.6. Best first search.

This method relies on the presence of some measure of merit for each of the nodes in the tree. The search strategy is to search the node which has the highest merit first. This can cause problems if the measure of merit is inaccurate which is often the case in real applications. Combinations of this method and depth and breadth first methods are possible such as finding the best node at the level below the root and performing a depth first search on that subtree.

6.8. Learning Techniques.

In his paper, [Araya1982] puts forward the idea that an intelligent system must have two fundamental capabilities: problem solving and learning. He proposes that the system

must have separate problem solving and learning components called PSC and LC. There is also the external source, ES. The basic system can therefore be represented by the diagram on the following page where lines 1 to 6 represent data channels. The distinction between the LC and the PSC is theoretical and could in practice be one unit. The learning process could then make changes to both the PSC and the LC and therefore make it possible to have a system that improves its learning. The end of the article contains a brief description of 10 learning systems and the modes that they use.

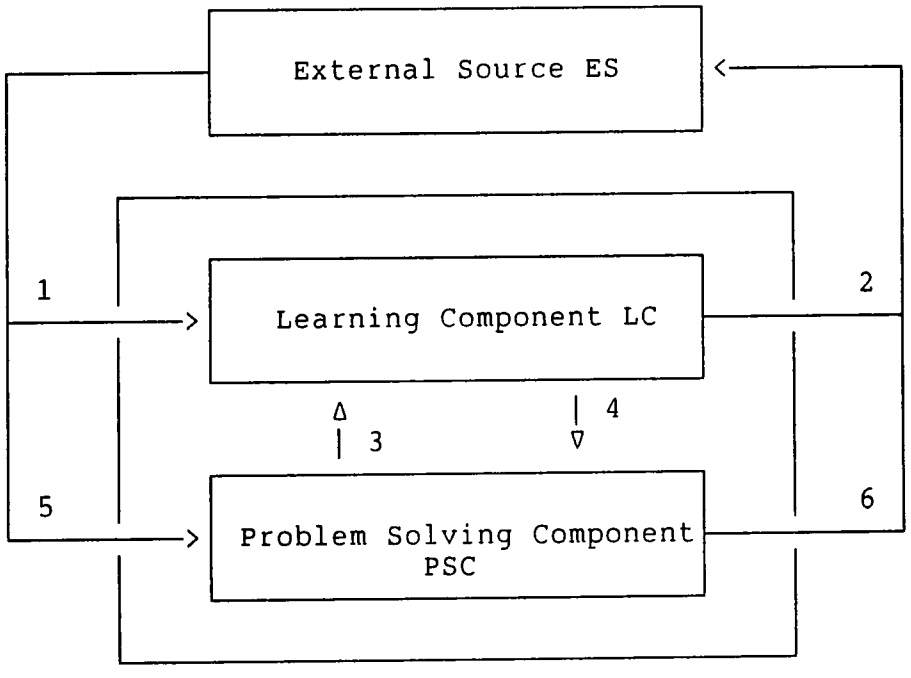
6.8.1. by Examples.

Systems that learn by example accept data for the LC via channel 1. The PSC works with the example and output is sent via channel 6. With this method the important mechanisms are generalisation and discrimination although over generalisation can be a problem. The generalisation process can be single or multi step and can involve the use of domain knowledge or not. Most of the work on learning has been done in this area.

6.8.2. by Instruction.

The input instructions are accepted via channel 1 and then the PSC and LC communicate with each other via channels 3 and 4. This is different from learning by example in that the system is given the concept and the problem then is to

Intelligent System



integrate it into the rest of the knowledge base. A special form of learning by instruction is analogy where the desired property is extracted by finding similarities and differencies.

6.8.2.1. Analogy.

The article by [Winston1982] is a study of the mechanics behind learning by analogy. The basic method used is to have a precedent and an exercise then extract the rule from the matching of the casual structure in the precedent to a corresponding structure in the exercise. The extracted rule is of a production rule type which is grouped by context.

6.8.3. by Practice.

For this learning mode the input is sent via channel 5 to the PSC which tries to solve the problem while the LC observes via channel 3. Any modifications are sent via channel 4 and a trained observer can watch via channel 6. The main problem with learning systems is obtaining the valuable information needed. This mode reduces the problem slightly by only requiring a problem which it can learn from. Once a path to the solution is found the system can keep relevant information so that similar problems can benefit from past experience.

6.8.4. by Exploration.

In this mode the LC and the PSC communicate via chan-

nels 3 and 4 with little external intervention. The lack of dominating external intervention makes this mode different from the rest. The system must be intelligent to create its own observations, examples and problems. One of the main problems with this mode is how to make the system create "interesting" information. One possible criteria is that a concept can be considered interesting if it is closely related to other interesting concepts. AM by [Lenat1977] is an example of this mode which explored the domain of elementary number theory. After a certain amount of success the system was unable to develop new heuristics to keep the search space small. The successor to AM, EURISKO, can devise new heuristics to associate with new concepts as it discovers them. This solves the problem of AM searching too large a search space. To discover new concepts the search must be forward since the goals are very vague. The search is conducted in a highly selective best first manner.

7. Prolog for an Expert System.

The idea of using Prolog to develop expert systems is not new. Two examples are, [Clark1982] and [Sergot1982]. As a language for implementing an expert system Prolog provides a knowledge base and an inference technique. The two are separate and since Prolog can also be used as a programming language, the implementation of an explanation mechanism is relatively easy. The inference technique used by Prolog is resolution with a depth first search strategy. Although depth first search has drawbacks, meta level Prolog programs can be written which implement other search strategies. The single method of representing the database and the program is an added advantage when manipulating any part of the expert system. (e.g. explanation).

The rest of this section will outline a particular problem with Prolog when trying to create an Expert System. The problem considered is distinguishing between true, false and unknown facts. Also considered are the deductions that can be drawn for true, false and unknown facts.

Consider the domain of simple positional information.

```
on(book, table).
on(pen, book).
under(X, Y) :- on(Y, X).
:- under(book, pen).
```

The first two lines are facts that can be read as

"the book is on the table"

and

"the pen is on the book".

The third line is the rule

"X is under Y if Y is on X".

The last line is a query to find whether the book is under the pen. Using this data base it is possible to prove that

the book is on the table,
the pen is on the book,
the book is under the pen,
the table is under the book.

All other queries are given as false since the data base is assumed to be a complete closed world. The query to find whether the cup is on the table is therefore false.

A more flexible approach to this query would respond with the answer "don't know" to show that it is either true or false that the cup is on the table, but it is unknown which.

One partial solution is to keep a list of objects and relations that are known in the given domain. Then, use this list to find if the query references any unknown objects or relations. In this example, the domain concerns "book", "table" and "pen" so a reference to a "cup" would be unknown. This solution provides an improvement but does not cope with a query like

"Is the pen on the table ?"

which is trying to find out whether the pen is half on the

book and half on the table. Here, 'on' is describing the relationship of objects being directly on top of each other. The objects and the relations are within the given domain but the information is still unknown.

Another alternative is suggested by [Forsyth1984]. His approach is to add the facts to the database by splitting them into true and false facts. So, for example, the above database would be :-

```
yes(on(book, table)).
yes(on(pen, book)).
under(X, Y) :- yes(on(Y, X)).
:- under(book, pen).
```

This could then be extended to include false facts, for example,

```
no(on(cup, table)).
```

The limitation of this solution is that the rules manipulating these facts are two-valued. This means that if the fact 'no(under(pen, table))' was added, it would not be possible to conclude that the table was not on the pen. This should be possible since the original database had the rule

```
under(X, Y) :- on(Y, X).
```

and if 'under(X,Y)' is false then 'on(Y,X)' must also be false otherwise the rule does not hold. Therefore, this approach does not handle the connection between three-valued facts and rules correctly. This requires a three-valued implementation for rules and facts where queries that cannot be proved to be true or false are assumed to be 'Not Known'.

If disjunctive clauses are required in a data base then using two-valued rules and negation by failure can cause contradictions. For example, if the only clause in a data base is

switch(on) or switch(off)

then both the queries "switch(on) ?" and "switch(off) ?" are neither true nor false. In a Prolog data base the disjunctive clause could be represented by using the 'not' operator which is defined by failure. (See [Clark1978]). The clause could then be represented by either

```
switch(on) :- not switch(off).
```

or

```
switch(off) :- not switch(on).
```

But, the first incorrectly concludes that the switch is on. The second incorrectly concludes that the switch is off. Prolog does not allow disjunctive conclusions to rules and transforming the rule to use 'not' in the body can only be used if there is a closed world. The constraint that is being violated above is the closed world assumption where all facts not present in the data base are assumed false.

7.1. Previous Work on three-valued logic.

In the paper about Natural logic, [Colmerauer1981a], the authors recognise the need to have three-valued logic to formulate Natural Language queries. This is to allow the correct interpretation of presuppositions. Their three

logic values are true, false and undefined or meaningless. Their basic operators are defined so that when any part of a clause is evaluated to meaningless, the whole clause becomes meaningless. For example

John sells or eats cars

is meaningless, regardless of whether John sells cars or not, since in Colmerauer's data base 'to eat cars' is meaningless. The implementation proposed in the next section uses a different interpretation. Instead, if it is true that John sells cars then the whole clause is true, but if it is false, and John eats cars is not referred to in the data base, then the whole clause is unknown. The use of a three-valued system for natural language processes is also investigated in [Dahl1979]. The subject of her paper is mainly concerned with quantification for representing natural language statements but to represent presupposition a three-valued system similar to [Colmerauer1981a] is used.

The work by [Bossu1985], provides a mathematical reasoning process to deal with implicit negative information in a data base. They define a special type of implication which they call sub-implication which is used on two-valued clauses to evaluate a query. The evaluation of a query results in one of three possible values { 0, 1/2, 1 } which are interpreted as 'No', 'Indefinite', and 'Yes'. The use of sub-implication requires extra 'characteristic' clauses to be added to the data base which are then used in the sub-

sequent query evaluation. This mathematical model seems to have no practical implementation yet.

In his paper, [Shimura1979], proposes a new form of modal logic called manner logic. This logic has four values: true, true*, false* and false. The "*" is taken to mean "in the sense of ... ". The author gives a basic truth table and then the standard laws and some new laws that the logic obeys. From these the author introduces some extra conditions that are needed for Robinson's refutation process using the new logic. The result of the refutation is that the formulae are satisfiable, semi satisfiable, semi unsatisfiable or unsatisfiable. This system requires modifications to the refutation process whereas the system proposed in this thesis does not. The proposed system can be implemented as a separate module that runs on top of standard Prolog.

7.2. An Evaluator.

To aid in the verification of the equivalences used in the next section, Prolog can be used as an automatic theorem prover. The method used is to evaluate one side of the equivalence for every possible value of variables contained in the expression and compare this with the set of values produced by the other side of the equivalence. So, for example, to prove

```
false(A) = true(not(A)).
```

there is a function to evaluate 'false(A)' which gives one set of results for A = 0,w,1 and then the evaluation of 'true(not(A))' gives the same set for A = 0,w,1. The evaluation function can be written easily in Prolog by using clauses such as

```
eval(not(0),1).
eval(not(w),w).
eval(not(1),0).
eval(not(X),A) :- eval(X,A1), eval(not(A1),A).
```

The first three clauses define 'not' and the last shows how to decompose a complex equation involving 'not' to a simple one. For the last clause to work correctly the variable X must be instantiated to some ground formula otherwise 'eval(X,A1)' would match any 'eval' clause in the database. This is insured by giving the variables in a formula one of the three basic values before 'eval' is called. For example,

```
simple(X), eval(not(X),A).
```

The clause 'simple' is defined as,

```
simple(0).
simple(w).
simple(1).
```

This can then be used to generate all the values for variables in the formula. The clauses must behave correctly under backtracking because this is how all the possible solutions to an equation are generated.

Example Proof for

```
unknown(P) = not(or(true(P),false(P))).


right :- simple(P), eval(not(or(true(P),false(P))),Z),
        write(P), write(Z), nl, fail.
left :- simple(P), eval(unknown(P),Z),
        write(P), write(Z), nl, fail.
```

This gives

```
00
w1
10
    and
00
w1
10
```

The basic evaluator was extended to include dynamic definitions of an operator and then to test certain rules with this definition. This was used for the definition of implication. The evaluator has a list which contains the truth values for the dynamic definition and this is scanned when an evaluation involving this operator is performed. For example,

```
        /* The correct defn for X and Y is found */
eval([eval(imp(X,Y),Z)|Tail],imp(X,Y),Z) :- !.
        /* This is not correct defn  -  recurse */
eval([Head|Tail],imp(X,Y),Z) :-
        eval(Tail,imp(X,Y),Z), !.

        /* not in list so eval. sub expressions and
        try again */
eval(L,imp(X,Y),A) :-
        simple(Y), eval(L,X,A1),
        eval(L,imp(A1,Y),A), !.
eval(L,imp(X,Y),A) :-
        simple(X), eval(L,Y,A2),
        eval(L,imp(X,A2),A), !.
eval(L,imp(X,Y),A) :-
        eval(L,X,A1), eval(L,Y,A2),
        eval(L,imp(A1,A2),A), !.
```

8. A Three-valued System - Theoretical Basis.

8.1. Proposed Definitions.

This section outlines the basic values, operators and equivalences used to express three-valued formulae in a two-valued system. The definitions are not meant as a formal system of logic but a justification for the implementation of a three-valued Prolog interpreter. For clearer presentation and to avoid confusion, three-valued formulae will be written in upper case and two-valued formulae will be written in lower case.

The three-valued system that is proposed has three basic values { 1, 0, w } representing true, false and unknown respectively. There are also three basic operators

{ AND, NOT, TRUE },

defined as follows

```
NOT(1) = 0      NOT(0) = 1      NOT(w) = w
AND(P, Q) = MIN{P, Q}  where  0 < w < 1
TRUE(1) = 1     TRUE(0) = 0     TRUE(w) = 0
```

Using these three basic operators the following operators can be built

```
OR(P, Q) = NOT(AND(NOT(P), NOT(Q)))  { = MAX{P, Q}  }
FALSE(P) = TRUE(NOT(P))
UNKNOWN(P) = not(TRUE(NOT(AND(P, NOT(P)))))
```

This collection of operators can be used to express any n-ary three-valued function 'F(P1,..,Pn)' as shown.

```
If n=0  then F is one of 1, 0, w  otherwise
F(P1,..,Pn) = CASE(Pn, F(P1,..,Pn-1, w),
                       F(P1,..,Pn-1, 0),
                       F(P1,..,Pn-1, 1)).

where
        CASE(w, P, Q, R) = P
        CASE(0, P, Q, R) = Q
        CASE(1, P, Q, R) = R

as defined by
        CASE(A, X, Y, Z) = OR(AND(UNKNOWN(A), X),
                              OR(AND( FALSE(A),  Y)),
                                 AND( TRUE(A),   Z)))
```

This means that any n-ary three-valued function where n > 0 can be expressed using the three truth-values and the three basic operators.

The difference between the proposed definitions and Colmerauer's is the way the 'AND' and 'OR' operators are defined.

```
Colmerauer's definition where 'w' represents meaningless
   AND(P,Q) = MIN{P,Q}    where w < 0 < 1
   OR(P,Q)  = MAX{P,Q}    where 0 < 1 < w

Proposed definition where 'w' represents unknown
   AND(P,Q) = MIN{P,Q}    where 0 < w < 1
   OR(P,Q)  = MAX{P,Q}    where 0 < w < 1
```

This difference allows the alternative behaviour explained in the section on Prolog for an Expert system.

At this point we diverge from Colmerauer's work in that we consider three-valued relations instead of two. Colmerauer takes the value of 'DEFINED(R0)' to be true, where 'DEFINED' is equivalent to 'not(UNKNOWN(R0))' and 'R0' is a

relation. In the proposed system facts that are true and facts that are false are stated explicitly in the data base and those facts that are unknown are omitted. This means that 'UNKNOWN(R0)' will be defined by

UNKNOWN(P) = not(or(TRUE(P), FALSE(P)))

The two-valued implication 'a :- b' is equivalent to 'or(a,not(b))' but when this is extended to the three-valued implication 'TRUE(A IF B)', there are several different interpretations, some of which are

OR(A,not(TRUE(B)))
OR(A,NOT(B))
OR(A,TRUE(NOT(B)))

These simple forms are inadequate at expressing the case when both 'A' and 'B' are unknown. To arrive at the interpretation used in this thesis two approaches will be taken.

8.1.1. Method One.

The first approach builds up a truth table from several statements. For the definition of 'TRUE(A IF B)'

(1) If 'A' is true 'B' can take any value and 'TRUE(A IF B)' will be true.

(2) If 'B' is false 'A' can take any value and 'TRUE(A IF B)' will be true.

(3) If 'B' is true and 'A' is not true then 'TRUE(A IF B)'

is false.

(4) If both 'A' and 'B' are unknown the implication holds so 'TRUE(A IF B)' is true.

(5) If 'A' is false 'B' must be false to make 'TRUE(A IF B)' true.

These can be used to define the complete truth table.

A	B	TRUE(A IF B)	from
0	0	1	(2),(5)
0	w	0	(5)
0	1	0	(3),(5)
w	0	1	(2)
w	w	1	(4)
w	1	0	(3)
1	0	1	(1),(2)
1	w	1	(1)
1	1	1	(1)

8.1.2. Method Two.

The second approach to the interpretation of the three-valued implication will use the basic rules of Modus Ponens, Modus Tolens and the Law of Syllogism. Before these can be used the appropriate three-valued representation of these rules must be found. Taking Modus Ponens for example. The two-valued rule is

```
( a and ( a->b ) ) -> b
```

When this is extended to three values it could be

```
( TRUE(A) AND ( A->B ) ) -> TRUE(B),
( TRUE(A) AND TRUE( A->B ) ) -> TRUE(B)
        or
( A AND ( A->B ) ) -> B
```

but the first two forms can be discounted since we are trying to find an interpretation of implication in a three-valued environment where there is a representation of 'A' not 'TRUE(A)'. In other words, if 'A' is represented by 'TRUE(A)' then the rule would be

(TRUE(A) AND (TRUE(A)->TRUE(B)) -> TRUE(B).

This is only a two-valued formula and therefore is not what is required. Also, the first two forms allow strange behaviour for the interpretation of implication, such as

A	B	A->B
0	0	1
0	w	0
0	1	1

The forms used for Modus Ponens, Modus Tolens and the Law of Syllogism are

(A AND (A->B)) -> B
(NOT(B) AND (A->B)) -> NOT(A)
(A->B) AND (B->C) -> (A->C)

8.1.2.1. Possible interpretations for A->B.

The three-valued definition must be compatible with the two-valued form.

a	b	a->b
0	0	1
0	1	1
1	0	0
1	1	1

This gives four of the values for three-valued A->B

A	B	A->B
0	0	1
0	w	
0	1	1
w	0	
w	w	
w	1	
1	0	0
1	w	
1	1	1

The rest of the values will be defined by considering the two conditions Modus Ponens and Modus Tolens.

Modus Ponens (A AND (A -> B)) -> B
Modus Tolens (~B AND (A -> B)) -> ~A

Considering Modus Ponens for A=0 and B=w

(0 AND (0 -> w)) -> w = 1
but 0 AND X = 0 so
0 -> w = 1

Similarly, considering Modus Tolens for A=w and B=1

(0 AND (w -> 1)) -> w = 1
but 0 AND X = 0 so
0 -> w = 1

Considering Modus Tolens for A=0 and B=w

(w AND (0 -> w)) -> 1 = 1
but 0 -> w = 1 so
(w AND 1) -> 1 = 1
so w -> 1 = 1

This gives

A	B	A->B
0	0	1
0	w	1
0	1	1
w	0	
w	w	
w	1	1
1	0	0
1	w	
1	1	1

When A=w and B=w then (w AND (w -> w)) -> w = 1.
Therefore w -> w <> w
When A=w and B=0 then (w AND (w -> 0)) -> 0 = 1.
Therefore w -> 0 <> w
When A=1 and B=w then (w AND (1 -> w)) -> 0 = 1
If 1 -> w = w then w -> 0 = 1
but when A=w and B=0 then (1 AND (w -> 0)) -> w.
with w ->0 = 1 gives 1 -> w = 1
Therefore 1 -> w <> w

This gives 8 possible interpretations

A	B	1	2	3	4	5	6	7	8
0	0	1	1	1	1	1	1	1	1
0	w	1	1	1	1	1	1	1	1
0	1	1	1	1	1	1	1	1	1
w	0	0	1	0	1	0	1	0	1
w	w	0	0	1	1	0	0	1	1
w	1	1	1	1	1	1	1	1	1
1	0	0	0	0	0	0	0	0	0
1	w	0	0	0	0	1	1	1	1
1	1	1	1	1	1	1	1	1	1

2 and 4 can be ruled out because
when A=w and B=0 then
(1 AND (w -> 0)) -> w
so if w -> 0 = 1 then 1 -> w = 1
5 and 7 can be ruled out because
when A=1 and B=w then
(w AND (1 -> w)) -> 0
so if 1 -> w = 1 then w -> 0 = 1

This leaves four possible interpretations. We now introduce the Law of Syllogism.

```
( (A -> B) AND (B -> C) ) -> (A -> C)
8 can be ruled out because
   when A=1 , B=w and C=0 then
      ( 1 AND 1 ) -> 0 = 0
Similarly for 6,  when A=1 , B=w and C=0 then
( 1 AND 1 ) -> 0 = 0
```

This now leaves two possible interpretations; 1 and 3.

A	B	1	3
0	0	1	1
0	w	1	1
0	1	1	1
w	0	0	0
w	w	0	1
w	1	1	1
1	0	0	0
1	w	0	0
1	1	1	1

To decide between these two possibilities a new rule will be used and that is self implication

A -> A

This holds for two-valued implication and for the case when A=w it seems logical that when A is unknown that implies A is unknown. The interpretation used is therefore three.

A	B	A->B
0	0	1
0	w	1
0	1	1
w	0	0
w	w	1
w	1	1
1	0	0
1	w	0
1	1	1

This interpretation has been verified using the Prolog

theorem prover with the four rules (Ponens, Tolens, Law of Syllogism and Self implication) to show that only one possible interpretation satisfies these rules out of the 19683 (i.e. 3 ^ 3 ^ 2) combinations. This table can be rewritten to give the same table as before using,

TRUE(A IF B) = B -> A.

The truth table is similar to the one produced by 'TRUE(OR(A,NOT(B)))' except for the case when both 'A' and 'B' are unknown. Although this truth table seems to produce a result which is difficult to express in a simple three-valued statement it can be expressed using the ordering of the truth values. This definition of implication can be defined as

TRUE(A IF B) = A >= B

where '>=' is the greater-than-or-equal operator when the truth values are ordered $0 < w < 1$. From this it can be seen that the order of truth values is important since, not only is it used to define implication, but also 'AND' and 'OR'. (AND(P,Q) = MIN{P,Q}). The work by [Bossu1985] also depends on the ordering of interpretations. Their definition of sub-implication depends on minimal models which are defined from the ordering relation '<'.

A three-valued logic system has been proposed by [Lukasiewicz1970] which has the following truth table for implication.

A	B	A IMPLIES B
0	0	1
0	1/2	1
0	1	1
1/2	0	1/2
1/2	1/2	1
1/2	1	1
1	0	0
1	1/2	1/2
1	1	1

In his definition the value of 1/2 is taken to mean "possibility" or "doubtful". This truth table translates directly into the definition derived above for TRUE(A IF B). He also gives a definition of the principles of identity.

The system that he proposes obeys most of the classical two-valued logic statements but he notes that some laws such as

```
( A = NOT(A) ) = 0
```

do not hold. [Lukasiewicz1970] also defines a general definition for a n-valued logic system where the values are in the interval (0,1) as

```
P IMPLIES Q  = 1        for P less than or equal Q
P IMPLIES Q  = 1-P+Q    for P greater than Q
```

and a definition of 'NOT' as

```
NOT P  = 1-P
```

The work of [Lukasiewicz1970] is commented on in [Lewis1959] where a list of elementary laws which hold in Lukasiewicz's three-valued logic is shown. The list includes the self implication rule 'P IMPLIES P'.

[Lewis1959] also provides definitions for AND and OR as

```
P OR Q  =  ( P IMPLIES Q ) IMPLIES Q
P AND Q  =  NOT( NOT(P) OR NOT(Q))
```

These two definitions are the same as the definitions in terms of MIN and MAX. Since 'AND' and 'OR' can be defined from 'IMPLIES' and 'NOT' the definitions that were presented at the beginning of this section could have been defined from the two operators 'IMPLIES' and 'NOT' instead of 'NOT', 'AND' and 'TRUE'.

The three-valued logic defined in [Kleene1962] uses a different form of implication and equivalence where implication has the value 'w' when 'P = w' and 'Q = w'. This definition has been tried with the transformations that appear in the next section but results in the system being unusable because of unknown variables.

In summary, the definition of three-valued implication is difficult and several approaches have been adopted. If self implication (A -> A) is considered valid the definition should be

```
TRUE(A IF B) = A >= B.
```

This is the definition used in the next section.

8.2. Transformations.

We now look at how these three-valued definitions can be used with Prolog. Clauses will be split into sections dealing with facts and rules.

8.2.1. Facts.

The fact 'A' will have one of the three truth values which can be represented by using the two-valued property of TRUE, FALSE and UNKNOWN. 'A' is therefore represented by

```
When A has truth value 1   i.e. TRUE(A) = 1
        the presence of 'true(a)' in the data base
When A has truth value 0   i.e. FALSE(A) = 1
        the presence of 'false(a)' in the data base
When A has truth value w   i.e. UNKNOWN(A) = 1
        the absence of both the above forms
```

N.B. When clauses are expressed in Prolog they are in lower case since the Prolog system used requires clauses to be in lower case and variables to be in upper case.

8.2.2. Rules.

This section will develop a representation for three-valued clausal form rules with the following syntax

```
TRUE( X1 OR X2 OR ... OR Xn IF Y1 AND Y2 AND ... AND Ym )
        provided n > 0 and m >= 0.
```

or

```
FALSE( Y1 AND Y2 AND ... AND Yq )
        provided q > 0.
```

The first version is clausal form defined in the intro-

duction and the second is an extended form which allows clauses to be added to the data base when 'n' above is zero.

Considering 'TRUE(A IF B)' first, it can be shown (See Appendix) that the following equivalence holds.

```
TRUE(A IF B) =
        and(or( TRUE(A),not( TRUE(B))),
            or(FALSE(B),not(FALSE(A)))
           )
```

Since the operators TRUE and FALSE can only take two values the statement can be directly represented. Thus, using 'or(a,not(b)) = a :- b' we get the following clauses

```
TRUE(A)  :- TRUE(B)    and
FALSE(B) :- FALSE(A).
```

The 'and' operator is represented by both rules being added to the data base. The data base for the three-valued rule 'TRUE(A IF B)' is therefore

```
true(a) :- true(b).
false(b) :- false(a).
```

This can now be extended to the three-valued rule 'TRUE(A IF B AND C') as follows.

```
TRUE(A IF B AND C)  =
TRUE(A IF (B AND C)) =
     and(or(TRUE(A),not(TRUE(B AND C))),
         or(FALSE(B AND C),not(FALSE(A)))
        )
```

Using the following equivalences

```
TRUE(B AND C) = and(TRUE(B),TRUE(C)).
FALSE(B AND C) =
        and(or(not(TRUE(B)),FALSE(C)),
            or(not(TRUE(C)),FALSE(B)),
            or(not(UNKNOWN(B)),not(UNKNOWN(C)))
           )
X or (Y and Z) = (X or Y) and (X or Z).
```

gives

```
TRUE(A IF B AND C) =
     and(or(TRUE(A),not(and(TRUE(B),TRUE(C)))),
      and(or(FALSE(B),not(and(FALSE(A),TRUE(C)))),
       and(or(FALSE(C),not(and(FALSE(A),TRUE(B)))),
           or(not(FALSE(A)),
              not(and(UNKNOWN(B),UNKNOWN(C))))
        )))
```

If it can be assured that

```
or(not(FALSE(A)),not(and(UNKNOWN(B),UNKNOWN(C))))
```

is always true (See next section on consistency and limitations) then the equivalence reduces to

```
TRUE(A IF B AND C) =
     and(or(TRUE(A),not(and(TRUE(B),TRUE(C)))),
      and(or(FALSE(B),not(and(FALSE(A),TRUE(C)))),
       and(or(FALSE(C),not(and(FALSE(A),TRUE(B))))
        )))
```

In a similar way to the treatment of 'TRUE(A IF B)', the equivalence can be put into a similar form.

```
TRUE(A IF B AND C)  therefore becomes
true(a) :- true(b), true(c).
false(b) :- false(a), true(c).
false(c) :- false(a), true(b).
```

We now consider another form of rule with disjunctive conclusions, TRUE(A OR B IF C). This can also be converted in a similar manner.

```
TRUE(A OR B IF C) =
TRUE((A OR B) IF C) =
     and(or(TRUE(A OR B),not(TRUE(C))),
         or(FALSE(C),not(FALSE(A OR B)))
        )
```

Using the following equivalences

```
TRUE(A OR B) =
     and(or(TRUE(A),not(FALSE(B))),
         or(TRUE(B),not(FALSE(A))),
         or(not(UNKNOWN(A),not(UNKNOWN(B))))
        )
FALSE(A OR B) = not(and(FALSE(A),FALSE(B))).
X or (Y and Z) = (X or Y) and (X or Z)
```

gives

```
TRUE(A OR B IF C) =
     and(or(TRUE(A),not(and(FALSE(B),TRUE(C)))),
      and(or(TRUE(B),not(and(FALSE(A),TRUE(C)))),
       and(or(FALSE(C),not(and(FALSE(A),FALSE(B)))),
           or(not(TRUE(C)),
              not(and(UNKNOWN(A),UNKNOWN(B))))
       )))
```

Again, if it can be assured that

```
or(not(TRUE(C)),not(and(UNKNOWN(A),UNKNOWN(B))))
```

is always true (See next section) then the equivalence reduces to

```
TRUE(A OR B IF C) =
     and(or(TRUE(A),not(and(FALSE(B),TRUE(C)))),
      and(or(TRUE(B),not(and(FALSE(A),TRUE(C)))),
          or(FALSE(C),not(and(FALSE(A),FALSE(B)))),
       ))
```

This can then be expressed as the Prolog rules

```
true(a) :- false(b), true(c).
true(b) :- false(a), true(c).
false(c) :- false(a), false(b).
```

8.2.2.1. General Rules.

The above gives representations for the three forms TRUE(A IF B), TRUE(A IF B AND C) and TRUE(A OR B IF C). We will now consider the general case of

TRUE(X1 OR X2 OR ... OR Xn IF Y1 AND Y2 AND ... AND Ym)

Provided the appropriate clauses are not Unknown the pattern which has emerged is to put each Xi at the head of the clause in turn and the rest of the Xi s inverted in the body and then each Yi inverted at the head and the rest in the body. For example, the rule

TRUE(A OR B OR C IF D AND E AND F)

would be represented in Prolog by

```
true(a) :- false(b), false(c),
           true(d), true(e), true(f).
true(b) :- false(a), false(c),
           true(d), true(e), true(f).
true(c) :- false(a), false(b),
           true(d), true(e), true(f).
false(d) :- false(a), false(b),
           false(c), true(e), true(f).
false(e) :- false(a), false(b),
           false(c), true(d), true(f).
false(f) :- false(a), false(b),
           false(c), true(d), true(e).
```

with the corresponding extra conditions outlined in the section on consistency.

Since there is no constraint on putting 'false(X)' at the head of the clause it is possible to represent the three-valued rule

FALSE(Y1 AND Y2 AND ... AND Yn)

by using the equivalence for FALSE(A AND B) used above. This can be combined with the above pattern. For example

FALSE(A AND B AND C)

would give

```
and(or(FALSE(A),not(and(TRUE(B),TRUE(C)))),
 and(or(FALSE(B),not(and(TRUE(A),TRUE(C)))),
  and(or(FALSE(C),not(and(TRUE(A),TRUE(B))))
   )))
```

which would be represented by

```
false(a) :- true(b), true(c).
false(b) :- true(a), true(c).
false(c) :- true(a), true(b).
```

8.2.3. Queries.

The form ':- A' can be implemented by attempting to satisfy 'true(a)' and then if that is unsatisfiable attempt to satisfy 'false(a)'. If that is also unsatisfiable then 'A' is unknown. However, if 'A' starts with a basic operator then the following should be used

```
true(TRUE(P)) = TRUE(P)
TRUE(NOT(P)) = FALSE(P)
TRUE(AND(A,B)) = and(TRUE(A),TRUE(B))

false(TRUE(P)) = not(TRUE(P))
FALSE(NOT(P)) = TRUE(P)
FALSE(AND(A,B)) = or(FALSE(A),FALSE(B))
```

To enable the use of a goal or query of the form 'unknown(A)' it is necessary to add the following new rules to the data base.

```
unknown(X) :- not(true(X); false(X)).
true(unknown(X)) :- unknown(X).
false(unknown(X)) :- true(X); false(X).
```

Note that the 'or' operator ';' is defined in Prolog for convenience. The basic form can now be easily extended to any form ':- B1, ... ,Bn' by using

```
TRUE(AND(A,B))  = and(TRUE(A),TRUE(B))
                = and(true(a), true(b))
FALSE(AND(A,B)) = or(FALSE(A),FALSE(B))
                = or(false(a), false(b))
```

For example the query ':- A,B,C' would be transformed into the query

```
:- true(a), true(b), true(c).
```

that would give the three-valued query the value true if it is satisfiable. If it is unsatisfiable the query

```
:- false(a); false(b); false(c).
```

would be executed. This would give the three-valued query the value false if it is satisfiable or unknown if it is not.

8.3. Consistency and Limitations.

Considering the statements that must be false when a rule or fact is added to a data base, leads to the conditions that must be satisfied for the three-valued equivalences used in the previous section to be true. For example

```
TRUE(A) = false(or(FALSE(A), UNKNOWN(A)))
        = and(false(FALSE(A)),false(UNKNOWN(A)))
```

Since by virtue of adding 'TRUE(A)' to the data base 'UNKNOWN(A)' will be false, it is only necessary to prove 'FALSE(A)' is false to prove the data base is consistent with 'TRUE(A)'. Moving on to the example of 'TRUE(A IF B)'. This is equivalent to

```
          false(or(
[1]                and(FALSE(A),UNKNOWN(B)),
                   or(
[2]                    and(FALSE(A),TRUE(B)),
[3]                    and(UNKNOWN(A), TRUE(B))
               ) ) )
```

Statement [1] must be false, when the rule 'false(b) :- false(a)' is added, since 'false(a)' is implying 'false(b)' and therefore 'UNKNOWN(B)' is false. Statement [3] must be false when the rule 'true(a) :- true(b)' is added for a similar reason. Statement [2] can be tested when the rule is added by executing the query

```
:- false(a), true(b).
```

This must be false and is a consistency check. It is assumed by executing the query when the rule is added that

the data base will remain consistent using some form of consistency maintenance. To perform this check correctly the rules must be added to the data base before the check is made. This is because the rules themselves might be necessary for the contradiction to be found. For example, considering the definition of integers where the integer 3 is defined erroneously to be not an integer.

```
true(integer(0)).
false(integer(s(s(s(0))))).
```

If the rule 'true(integer(s(X)) if integer(X))' is to be added, the consistency check ':- false(integer(s(X))), true(integer(X))' is correct until the definition of integers themselves is added. The inconsistency is then detected showing the rule for integers is inconsistent with 3 not being an integer. Adding the rule first makes recovery after a contradiction has arisen much more difficult but, as in the case above, this added complexity is necessary.

Now considering the statement

TRUE(A IF B AND C)

which is equivalent to all the following being false.

```
[1]     :- false(a), unknown(b), unknown(c).
[2]     :- false(a), unknown(b), true(c).
[3]     :- false(a), true(b), unknown(c).
[4]     :- false(a), true(b), true(c).
[5]     :- unknown(a), true(b), true(c).
```

Condition [2] must be false because of 'false(b) :-

false(a), true(c)' ; condition [3] must be false because of 'false(c) :- false(a), true(b)' and condition [5] must be false because of 'true(a) :- true(b), true(c)'. Condition [4] is a consistency check and condition [1] is the unknown check. The conditions that are tested to be false are therefore

```
:- false(a), true(b), true(c).
:- false(a), unknown(b), unknown(c).
```

This is the condition that is required so that the equivalence above can be used. The unknown check is only partially checked by trying to execute

```
:- false(a), unknown(b), unknown(c).
```

which will work when 'a', 'b' and 'c' are clauses which do not have variables. When they do have variables the unknown check will try to find a case when the clause is known but does not guarantee that for every case the clause is unknown. For example,

```
false(b(m)).
false(c(m)).
false(a(m)).
true(a(X) if b(X) and c(X)).
```

The test above is correct at this point because there are no values of X for which a(X) is false and b(X) and c(X) are unknown. If the fact

```
false(a(n)).
```

is now added there is a value of X (i.e. when X = n) where the unknown check is now invalid. This means that the rule

will not be able to give the correct answer to a query such as ':- false(b(n) and c(n)).' The answer will be 'DONT KNOW' instead of 'YES'. This limitation arises because 'unknown(a)' is defined as

```
not(true(a); false(a))
```

which can not be executed correctly by Prolog when the argument to 'not' contains variables. See [Clark1978] for an explanation of how 'not' is implemented in Prolog. The user must be aware of this limitation which is caused by the way the proposed definitions are implemented in Prolog.

Now considering

```
TRUE(A OR B IF C)
```

the following must be false

```
[1]     :- false(a), false(b), unknown(c).
[2]     :- false(a), false(b), true(c).
[3]     :- false(a), unknown(b), true(c).
[4]     :- unknown(a), false(b), true(c).
[5]     :- unknown(a), unknown(b), true(c).
```

[1], [3] and [4] are assured to be false when the rules

```
true(a) :- false(b), true(c).
true(b) :- false(a), true(c).
false(c) :- false(a), false(b).
```

are added to the data base. [2] is the consistency check and [5] the unknown check. Again, a limitation of this system is that it can not be guaranteed that [5] will always be false.

Following the pattern that has developed in these exam-

ples the checks that are made for a rule of the form

TRUE(A OR B OR C IF D AND E AND F)

would be :-

[1] Consistency check with all formulae to the right of ':-' i.e. invert operator

```
:- false(a), false(b), false(c),
        true(d), true(e), true(f).
```

[2] Group of checks for unknown goals with pattern, "false conclusions and two or more unknown conditions, others true"

```
:- false(a), false(b), false(c),
        unknown(d), unknown(e), unknown(f).
:- false(a), false(b), false(c),
        unknown(d), true(e), unknown(f).
:- false(a), false(b), false(c),
        true(d), unknown(e), unknown(f).
:- false(a), false(b), false(c),
        unknown(d), unknown(e), true(f).
```

[3] True conditions and two or more unknown conclusions, others false

```
:- false(a), unknown(b), unknown(c),
        true(d), true(e), true(f).
:- unknown(a), false(b), unknown(c),
        true(d), true(e), true(f).
:- unknown(a), unknown(b), false(c),
        true(d), true(e), true(f).
:- unknown(a), unknown(b), unknown(c),
        true(d), true(e), true(f).
```

To put this into a simpler form it can be summarised as: if there is more than one unknown in a rule, the rule must be examined more carefully to see if information will

be lost.

8.3.1. Query Variables.

The proof strategy of Prolog executes a query in a top-down left to right fashion which has serious drawbacks when it is used with this three-valued implementation. The main problem is that the query itself can not be used in the construction of the proof. Considering the example given in [Kowalski1979] for his Connection Graph Proof Procedure.

```
true(happy(X) if playing(X))
true(happy(X) if working(X))
true(playing(bob) or working(bob))
```

These rules are translated to

```
true(happy(X)) :- true(playing(X)).
false(playing(X)) :- false(happy(X)).

true(happy(X)) :- true(working(X)).
false(working(X)) :- false(happy(X)).

true(playing(bob)) :- false(working(bob)).
true(working(bob)) :- false(playing(bob)).
```

From these rules it should be possible to conclude 'true(happy(bob))' but this is only possible if the negated query is included in the set of clauses. This allows the proof strategy to use the query in the proof. This can be partially overcome by adding 'false(happy(bob))' to the data base while the query 'true(happy(bob))' is being evaluated. This method of resolution stems from the work by [Robinson1965] where the negated query is added to the set of clauses and resolution tries to find a contradiction.

Unfortunately, adding the negated clause to the data base does not work for all cases since the connection between any variables in the query and the clause added to the data base is lost. For example, the rule

```
true(man(lesley) or man(leslie)).
```

with the query

```
:- man(X).
```

When executing 'true(man(X))', 'false(man(X))' is added to the data base so the rule 'true(man(lesley)) :- false(man(leslie)).' will succeed with the variable X in the data base equal to 'leslie' and the variable X in the query equal to 'lesley'. This therefore incorrectly concludes that 'lesley' is a man.

One solution to this problem is to add the negated query to the data base, execute the query and then after the variables have been instantiated the old query is removed, the new one added and the query evaluated again. The second execution insures that any variables that are instantiated are consistent with the clause put in the database. This method must be used carefully when more than one query is on the command line and when backtracking is involved. For example

```
:- a(X,Y), b(Y,Z).
```

Initially, the clause 'false(a(X,Y))' would be added to the database and say, the query returned 'a(1,2)'. This would

then have to be checked by removing 'false(a(X,Y))', adding 'false(a(1,2))' and executing the query again. Assume this was correct and execution continued with 'b(2,Z)'. If this fails, Prolog must be able to backtrack to the point where 'false(a(X,Y))' is in the database and 'false(a(1,2))' is not present. To achieve this the clauses that assert and retract the query clauses must be backtrackable. This could be written as follows

```
assertquery(X) :- assert(X).
assertquery(X) :- retract(X), fail.
        On backtracking remove the clause and fail.
retractquery(X) :- retract(X).
retractquery(X) :- assert(X), fail.
        On backtracking add clause that was
        removed and fail.
```

This extension to Prolog's proof strategy can therefore only be used when the query does not include any variables. Otherwise only proofs which do not need the query can be solved.

9. A Three-valued System - Practical System.

9.1. Implementation.

The experimental implementation that was developed to test the three-valued system is based on C-Prolog as developed by Fernando Pereira at Edinburgh Computer Aided Architectural Design, University of Edinburgh, [Pereira1984a]. The main code is written in the language 'C', [Kernighan1978], but the top level interpreter is written in Prolog. The three-valued system can be implemented by modifications at the top level only, the inferencing steps of Prolog remain unaltered. Hence, most of the modifications were made to the Prolog code to create a new top level. The C-Prolog predicate '$dogoal'(X, Q) tries to satisfy the goal 'Q' at the top level. Another rule is therefore added after this one to try to satisfy the goal 'false(Q)' and change the original to look for 'true(Q)' or 'Q'. This can be visualised as a three-valued evaluator called at the top level.

The Prolog system developed by [Spivey1982] uses a similar structure except the main code is written in Pascal therefore the Prolog modifications made to C-Prolog should be easily adaptable to York Prolog. The C-Prolog messages written out when a goal is satisfied and variables are instantiated were also changed. The three-valued implementation was designed to run on top of standard Prolog but also provide an environment where the two and three-valued

systems exist together. See diagram on the next page.

The three-valued assert provides a transparent mechanism to add three-valued rules to the data base. If the rule is in three-value syntax then it is automatically converted to two-valued rules and added to the database. This also provides the consistency and unknown checks automatically.

The three-valued execute provided an evaluator for three-valued queries and provided a suitable control mechanism.

9.1.1. Control.

There are many cases in standard Prolog where rules can be added which cause the evaluation algorithm to go into an infinite loop. Using the three-valued system also causes this to happen. For example,

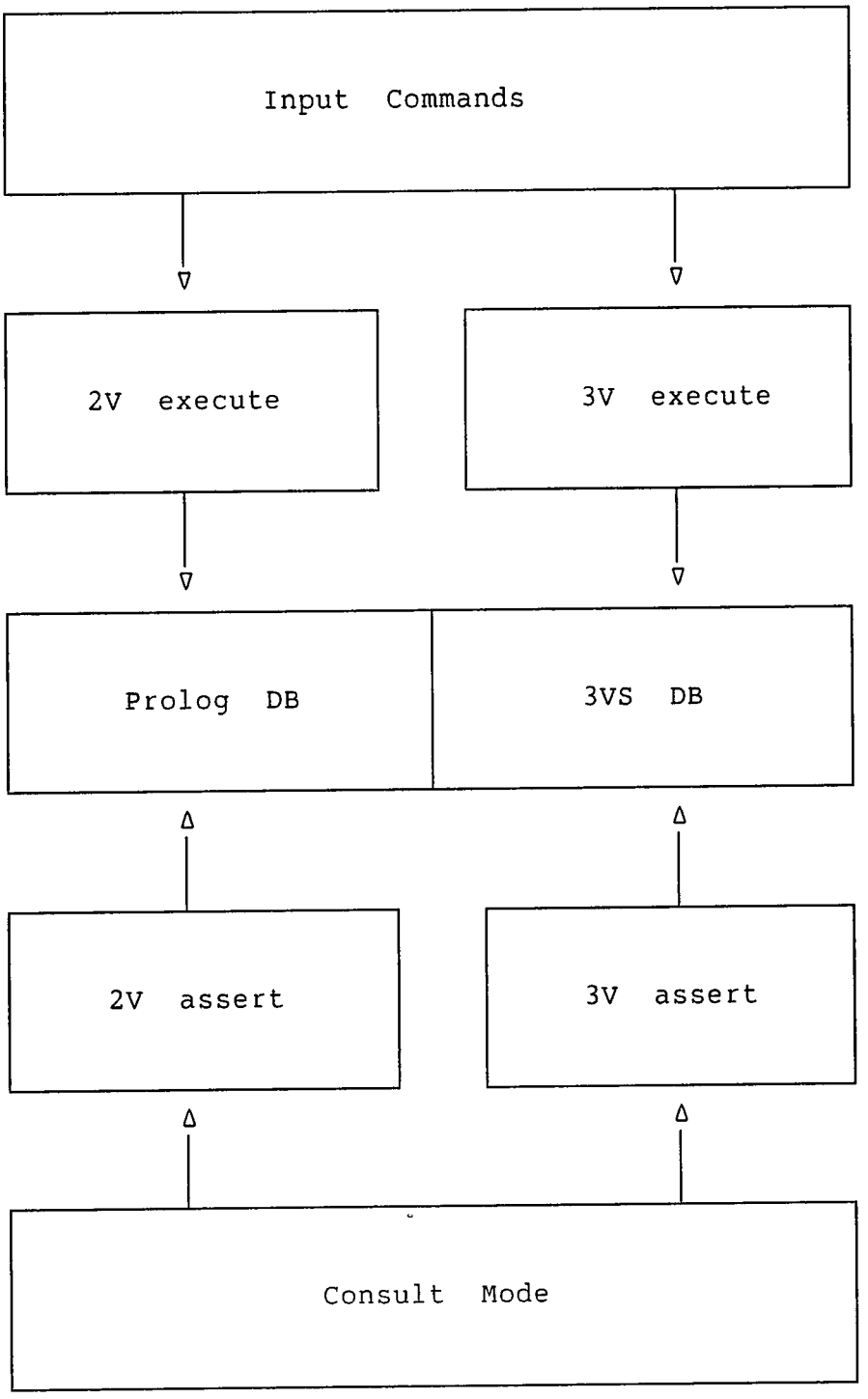
```
true(switch(on) or switch(off)).
false(switch(on) and switch(off)).
```

is translated into

```
true(switch(on)) :- false(switch(off)).
true(switch(off)) :- false(switch(on)).
false(switch(on)) :- true(switch(off)).
false(switch(off)) :- true(switch(on)).
```

Prolog's control strategy searches for a goal in a top down manner i.e. searches from the requested goal until a fact is found instead of searching from the given facts to generate the goal. The current goal is matched against the data base from top to bottom and when the goal is matched the new goal

Two and Three valued systems



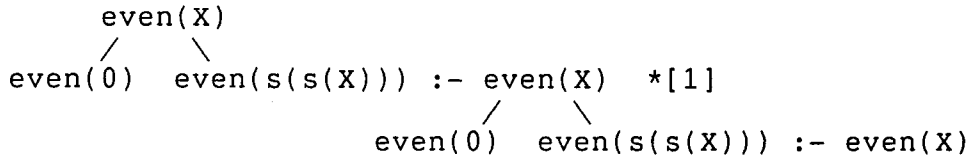
is the left most formula. This approach causes infinite loops to occur when clauses such as :-

```
true(a) :- false(b).      [1]
false(b) :- true(a).      [2]
```

appear in the data base. If the goal is 'true(a)' the first clause to be matched is [1] and the new current goal is 'false(b)'. The data base is then searched from top to bottom and [2] is the first to match. The new current goal is therefore 'true(a)'. We are now back in the same state as the beginning and the search will continue indefinitely (until stack space runs out). One solution to the problem is to keep a list of the goals that are currently being solved. When a new goal is tried it is first checked against all the previous goal to see if there is a match. If there is the search is looping and the new goal fails.

Robinson's resolution principle has a subsumption check where any clause that is subsumed by another clause is removed from the set of clauses. The subsumption is checked by trying to unify the clause with any other clause in the data base. This can not be used in our system because although it does not alter the satisfiability of the set of clauses it does stop some solutions from being generated. For example :-

```
even(0).
even(s(s(X))) :- even(X).
?- even(X).
```



At point *[1] the new goal is unifiable with the original goal and the new goal would fail. This stops the generation of more solutions. In Robinson's resolution principle unification failed if a variable was unified against a term that also contained the same variable. This is called the 'occur' check which is not implemented in most Prolog systems. Since this is the case, to check whether a goal has occurred before, the list is compared with the current goal using the '==' operator that tests to see if the two items are literally the same. (X == Y fails , X == X succeeds).

This strategy can still cause looping but only when the programmer requires it. For example,

```
:- append(X,Y,Z).

append([A|R], X, [A|R1]) :- append(R,X,R1).
append([],A,A).
```

This is creating an infinite list and goals are never literally equal so an infinite loop will occur.

To cope with certain aspects of problem solving when there are many paths in the search space to explore but the solution is not at a great depth, the evaluator includes a depth count which will cause the search to fail at a given

depth of the tree. This can cause the search to continue at a higher level and hopefully find a solution. This search is similar to the hybrid depth first / breadth first search described in the introduction to expert systems. The evaluator can also cause failure when a stack is full rather than aborting the program. This can be useful in limiting the depth of a search.

9.1.2. Efficiency considerations.

This section will analyse the efficiency of the three-valued system. Efficiency will be considered in terms of execution time, space taken for clauses and dynamic stack space.

9.1.2.1. Speed.

When a three-valued clause is added to the data base it is transformed in to several two-valued rules each with a different head. Therefore, when a two-valued goal is executed it can only possibly match one of the rules provided by the three-valued system. For example,

```
TRUE(on(X,Y) :- under(Y,X)).
```

becomes

```
true(on(X,Y)) :- true(under(Y,X)).
false(under(Y,X)) :- false(on(X,Y)).
```

so with a two-valued query such as,

```
?- true(on(book, table)).
```

only one clause can possibly match. Since the internal structure of the clauses remain unaltered the speed of unification is the same as for standard Prolog. The ability to find the matching clause in a similar time will be dependent on the use of indexing so all clauses with 'true(...)' are not tried.

When a three-valued query is executed at the top level it is split up into two two-valued queries to search for the true and false queries. If the two queries are of roughly equal complexity solutions which are found to be true will be found in a time comparable with standard Prolog, solutions which are false will take up to twice as long (due to two searches from the top level) and unknown solutions will take approximately twice as long.

If Prolog was run on dual parallel processors then the top level query could easily be split between the two processors since there are no shared variables. The true and false solutions would then be found in easiest-to-prove order and with virtually no speed penalty.

This analysis has ignored any use of a loop detection mechanism. In the developed system a meta level Prolog program was written to keep a list of the goals executed so far and tested to see if goals were being repeated. This three-valued interpreter was significantly slower because it involved a Prolog program interpreting another Prolog program. If the loop detection was built into the object level

Prolog interpreter it should have a better performance than the method used.

9.1.2.2. Space.

9.1.2.2.1. Static.

The static space required to store the clauses in the three-valued system is obviously greater than that required for two-valued clauses. The space required is dependent on the number of goals in the body of a clause. This will be represented by NG. The number of clauses required in the database will be called NC.

For facts, when NG = 0, only one clause is added to the database depending on whether the fact is true or false. Therefore NC = 1. For a rule with one goal in its body, NG = 1, clauses are added with each possible goal or head at the head of the clause. Therefore NC = 2. Similarly, when NG = 2, the number of clauses added is 3. Therefore

$$NC = NG + 1.$$

If the number of goals in a clause is averaged over the whole database the average increase in static space requirements can be calculated. So, for example, if the average number of goals was 1 the expected increase in static space would be twice that required for two-valued clauses.

9.1.2.2.2. Dynamic.

At the top level the true and false goals are executed sequentially and so the proof tree for the false goal is not constructed until the true goal has failed. Since all dynamic space is recovered on backtracking only space for one proof tree will exist at once. From the argument for speed, the number of clauses used in the proof is not increased by the three-valued system so therefore there will be no significant increase in the dynamic space requirements for this system.

9.2. Demonstration Systems

9.2.1. Example.

The following is an example of the three-valued implementation running. In C-Prolog the syntax of the headless clause uses a '| ?-' instead of ':-'. The text to the right within '{}' are comments added later to explain the example. The text typed by the user appears after a '| ?-', '| ' or '%' prompt and in reply to 'Enter list of ...'. The first example shows the debugging mode which prints the query executed to find contradictions and the clauses asserted. The example also shows what happens when a contradiction is found and when a rule has more than one unknown.

```
% prolog2                                  {Start Prolog version 2}
C-Prolog version 2.0
| ?- [user].                               {Add new 2 or 3V rules }
| debug3vs.                                {Switches debugging on }
| true(on(book,table)).                    { the book is on table }
Execute false(on(book,table))              {Query executed to find}
                                           { contradiction        }

true(on(book,table))  Asserted
| true(on(pen,book)).                      {the pen is on the book}
Execute false(on(pen,book))
true(on(pen,book))  Asserted
| true((under(X,Y) if on(Y,X))).           { Rule that if X is on }
                                           { Y then Y is under X  }
Execute false(under(_22,_23)) and true(on(_23,_22))
Execute false(under(_22,_23))              { execution for contra-}
                                           { diction fails at this}
                                           { goal                 }
true(under(_22,_23)):-true(on(_23,_22))  Asserted
false(on(_23,_22)):-false(under(_22,_23))  Asserted
                                           { Two rules added as   }
                                           { required             }
| false(on(X,table)).                      { Nothing is on the    }
                                           { table                }

Execute true(on(_22,table))
! Contradiction found                      { But this is not true }
true(on(book,table))                       { the book is on the   }
                                           { table.               }
```

```
| true((above(X,Y) if below(Y,X))).
                                    { Try to add a rule    }
                                    { which will fail the  }
                                    { unknown tests.       }
Execute false(above(_22,_23)) and true(below(_23,_22))
Execute false(above(_22,_23))
! Rule has more than one unknown fact
| false((under(X,Y) and on(X,Y))).
                                    { If X is on Y then it }
                                    { cannot also be under }
Execute true(under(_22,_23)) and true(on(_22,_23))
Execute true(under(_22,_23))
Execute true(on(_23,_22))
Execute true(on(table,book))
Execute true(on(book,pen))
false(under(_22,_23)):-true(on(_22,_23))  Asserted
false(on(_22,_23)):-true(under(_22,_23))  Asserted
                                    { exit consult mode    }
| ^Duser consulted 640 bytes 1.03333 sec.

YES
| ?- abolish(debug3vs,0).           { Switch off debugging }
Execute true(abolish(debug3vs,0))
Execute2v abolish(debug3vs,0)

YES
| ?- abolish(true,1).               { Clear database to    }
                                    { remove clauses added }
YES                                 { by rules which failed}
| ?- abolish(false,1).

YES
```

The second example shows a small database added and then the execution of several types of queries.

```
| ?- [-user].                       { reconsult the user   }
| true(on(book,table)).             { Add correct rules and}
| true(on(pen,book)).               { facts                }
| true((under(X,Y) if on(Y,X))).
| false((under(X,Y) and on(X,Y))).
| ^Duser reconsulted 344 bytes 0.366668 sec.

YES
| ?- on(cup,table).                 { unknown query        }

DONT KNOW
| ?- under(pen,book).               { known to be false    }

NO
| ?- under(X,Y).                    { Find all the under   }
```

```
                                        { relations.          }
X = table                               { These are the true  }
Y = book ;                              { facts               }

X = book
Y = pen ;

X \= book                               { These are the facts }
Y \= table ;                            { that are proved to be}
                                        { false               }

X \= pen
Y \= book ;

DONT KNOW                               { Dont know any other }
                                        { facts               }
```

The last example uses a utility called 'questions' written to find all the unknown relations given a list of predicates and a list of atoms.

```
| ?- ['3vs'].                           {Load some three-valued}
3vs consulted 4872 bytes 2.31667 sec.
                                        { utilities           }

YES
| ?- questions.                         { Print out all the   }
                                        { relations that are  }
                                        { unknown             }
Enter list of predicate names..[on(X,Y),under(A,B)].
Enter list of atoms..[table,book,pen].
Dont know about on(table,table)
Dont know about on(table,pen)
Dont know about on(book,book)
Dont know about on(pen,table)
Dont know about on(pen,pen)
Dont know about under(table,table)
Dont know about under(table,pen)    { data base does not  }
Dont know about under(book,book)    { contain rules about }
Dont know about under(pen,table)    { things being on or  }
Dont know about under(pen,pen)      { under themselves    }

YES
| ?- [user].
| false(on(X,X)).                     { Add required rules  }
| false(under(X,X)).
| ^Duser consulted 88 bytes 0.15 sec.

YES
| ?- questions.
Enter list of predicate names..[on(X,Y),under(A,B)].
Enter list of atoms..[table,book,pen].
```

```
Dont know about on(table,pen)
Dont know about on(pen,table)        { Now only wants to     }
Dont know about under(table,pen)     { know about the table  }
Dont know about under(pen,table)     { and the pen.          }
                                     { Points out that it    }
                                     { doesn't know what is  }
                                     { on the pen or under   }
                                     { the table.            }

YES
| ?- halt.

[ Prolog execution halted ]
%
```

9.2.2. An Expert System Shell.

In order to provide a consistent interface between the user and the three-valued Prolog implementation an Expert System Shell was written. This provides several features such as explanation, reasoning and mode of operation. The shell also helped in the development of two experimental systems providing consultations for VAX 11/750 booting and a law database.

The modes of operation provided by the shell are either consultation or search mode. The consultation mode displays an introduction to the expert system and then starts asking the user questions until the goal has been reached or the consultation fails. During the consultation the system will acquire facts from the result of questions and these can be displayed in search mode. In this mode the shell will run up a Prolog interface so that any Prolog queries can be entered. It is then possible to list all the facts that were acquired previously or list the rules that make up the expert system. This can be useful in, for example, the law

database to find all offences that carry a maximum fine of greater than 50 pounds. It can also be used to find all the possible devices that can be used to boot the VAX. This provides transparency and limited explanations which were detailed in the introduction to expert systems.

9.2.3. Booting Expert.

This section describes an expert system developed using the three-valued Prolog implementation. The expert system was first developed using standard Prolog but had several drawbacks. It was then rewritten using the three-valued system which greatly enhanced its performance.

The expert system contains 20 three-valued rules and 56 two-valued rules about the boot procedure on a VAX 11/750 running UNIX 4.1BSD. This was chosen because :-

[1] there was little help available,

[2] the procedure was non standard because of the non standard devices attached to the VAX,

[3] there are many alternative ways to boot,

[4] the possibility of different people rebooting the VAX

[5] and the transfer of knowledge from the domain expert to the knowledge engineer was avoided since the author was familiar with the booting procedure.

The original expert system ran into problems when facts

were added to the database after questions were asked. Originally, when a question was answered 'yes' the fact was asserted. When a question was answered 'no' the database remained unaltered. The problem with this representation is that there is no distinction between unknown facts and facts that are known to be false. In the expert system this is shown by the repeated asking of the same question that was originally answered 'no'. The expert system therefore needs a representation of negative facts and must use some form of three-valued reasoning. In his book, [Forsyth1984] recognises the need to add negative facts but does not extend this idea to incorporate three-valued rules.

The query form of the three-valued system tries to solve the true query first and then if that fails tries to solve the false query. This order is acceptable if the database does not change between the two evaluations. When used in the expert system, before a query is asked both evaluations must be done before new information is added. This is achieved by calling the three-valued evaluator ('eval') as a goal of the current clause instead of just for the top level query. For example

```
true(poweronaction(P)) :-
        eval(fact(poweronaction(P)),State),
        check(poweronaction(P),State).

check(poweronaction(P),true) :-
        /* the switch is already correct */
check(poweronaction(P),false) :-
        /* find old position and request change */
check(poweronaction(P),unknown) :-
        /* request switch put in position P */
```

The best example of three-valued rules in the booting expert system is the representation of the power-on-action switch. This switch can be in either the 'boot' position or the 'halt' position. Initially the position of the switch is unknown but when it is needed a question is asked and the position added to the database. Also in the database is the three-valued rule

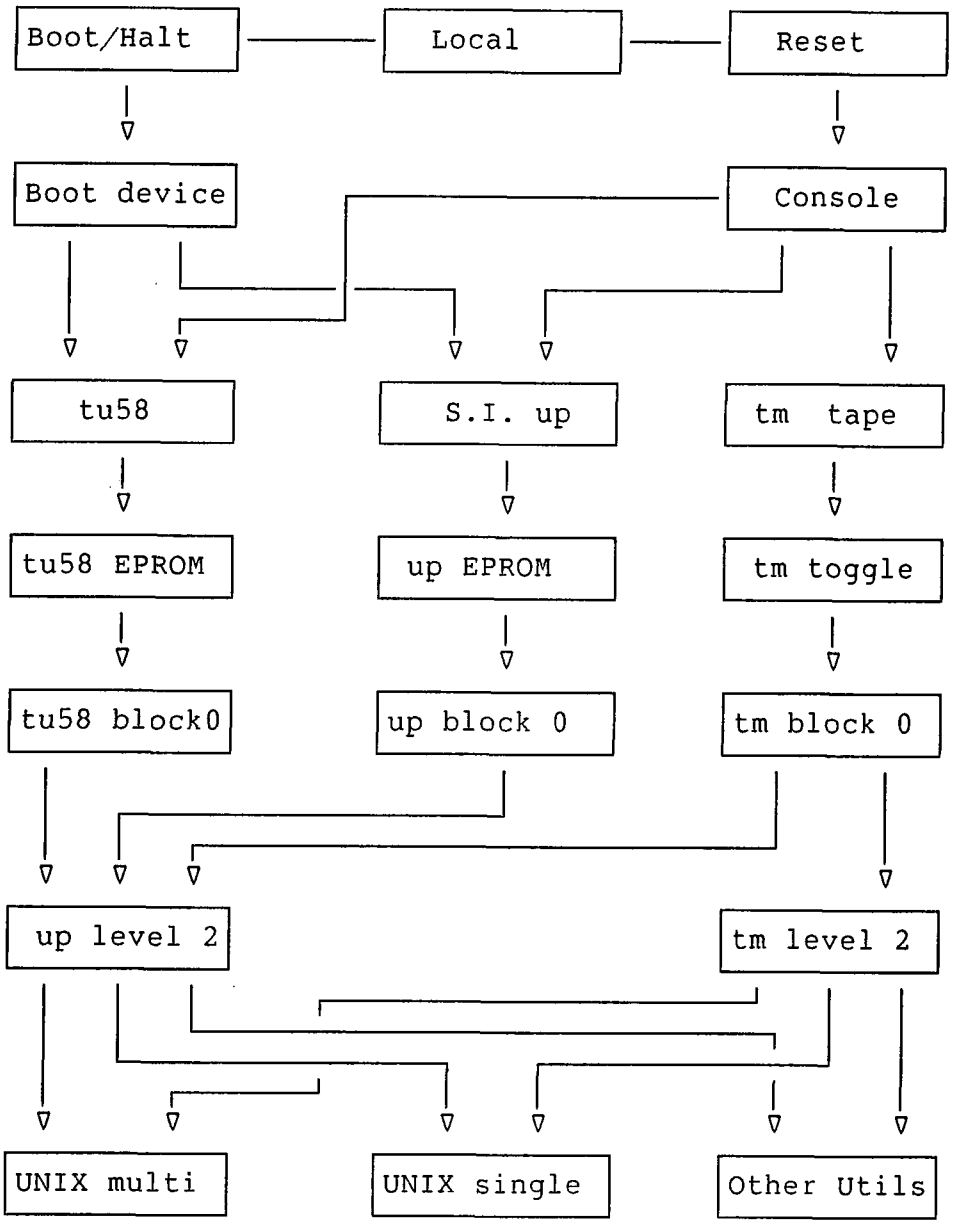
```
false((fact(poweronaction(halt)) and
       fact(poweronaction(boot)))).
```

This rule allows the system to know the switch is not in the other position without explicitly asserting the fact. When the switch is changed the rule can be used to find what must be retracted before the new switch position can be asserted. To insure that the data base always remains consisted only facts are added based on input data. This means that the retraction of a fact represents a change in the input data not a change in the rules of the data base.

This type of alteration to the database can be viewed as learning by instruction. The learning component is given the required data which is integrated into the database. The problem solver then uses this new information like any other information in the database.

The diagram on the next page shows the basic paths that could be taken by the booting expert to get UNIX running. Each of the connections between the boxes also carries flag values which will influence the choices taken later. For

Boot Path for UNIX



example, if the Boot/Halt switch is in the boot position then when the level two boot is reached it will try to bring up UNIX in multi user mode. The following is an example consultation with the 'boot' expert system. The consultation proceeds positively until the level one boot fails from the TU58 tape drive. It then starts again trying to auto boot from the disc drives. This fails to load UNIX properly so the expert system tries to boot manually. It then successfully boots from the TU58 tape. Lines starting ':' are reasoning messages.

Example consultation.
    Comment added later are between '{}'.

```
% prolog2
C-Prolog version 2.0
| ?- [boot].
bootdevice consulted 11912 bytes 19.5 sec.
bootloader consulted 84 bytes 0.200012 sec.
leveloneboot consulted 292 bytes 0.350008 sec.
leveltwoboot consulted 704 bytes 0.483336 sec.
unix consulted 168 bytes 0.100011 sec.

This demonstration Expert will try to perform the boot
procedure on a VAX 11/750 running UNIX 4.1bsd.
The questions asked should be answered with "yes." or "no."
including the full-stop.  When the consultation is finished
the facts acquired during the consultation can be found
using the search mode and the query "facts".

Do you want to use the expert system for a consultation
or do you want to search for rules.
( consult/search ) ? consult.
Do you want reasoning to be printed ? yes.
: Booting is performed using a boot loader and a two level
: process.  The boot loader is small program that can read
: the first block on the selected device.  This program is
: either in ROM or typed in by hand.
: The first level boot is a program that
: is at block zero which can search the root directory on
: the device and find a program called "boot".  The second
: level is this program called boot which can scan the root
: directory and load in the UNIX operating system.
:
: Selecting the boot device
```

```
: First try to boot automatically
: Set up to auto boot from tu58
Check that the VAX bootstrap tape is loaded in the tu58 tape
drive. Done ? yes.
Check that the power-on-action switch is turned to
boot. Done ? yes.
Check that the Boot Device switch is turned to A. Done ? yes.
Turn Key Switch to local position. Done ? yes.
Reset the VAX using the Reset button. Done ? yes.
: Trying to execute a boot loader
: Boot loader has succeeded. Now trying to execute a level
: one boot program
Check that System Industries controller and drives are
powered up and online.  A reset using the reset switch on
the controller will ensure the drives are on line. Done ? yes.
: Level one boot has succeeded. Now trying level two boot
Has the following been output: "Boot". Done ? no.
                          { The tu58 tape is corrupted ?? }
: tu58 auto boot has failed so set up to auto boot
: from System Industries drives
: The System Industries controller and drives have
: already been checked
: The power on action switch is correct
Change the Boot Device switch from A to D. Done ? yes.
    { Note that the expert knew the previous switch position }
    { Also switch in position A has been removed }
: Key Switch is already in local position
Reset the VAX using the Reset button. Done ? yes.
: Trying to execute a boot loader
: Boot loader has succeeded. Now trying to execute a level
: one boot program
: Level one boot has succeeded. Now trying level two boot
Has the following been output: "Boot". Done ? yes.
Has the following been output: ": up(0,0)vmunix". Done ? yes.
: Checking that UNIX has been loaded properly by the
: level two boot
Has the following been output:
"integer+integer+integer start hexadecimal". Done ? no.
                  { UNIX hasn't loaded properly }
: Auto boot has failed so now try manually
Change the power-on-action switch from boot to halt. Done ? yes.
: Key Switch is already in local position
Reset the VAX using the Reset button. Done ? yes.
: The console emulator should now be running
: First try the tu58
: The tu58 tape is already loaded
: Set up so that UNIX comes up single user
Type 'B/2 DDA0' on the console. Done ? yes.
: Trying to execute a boot loader
: Boot loader has succeeded. Now trying to execute a level
: one boot program
: The System Industries controller and drives have
: already been checked
: Level one boot has succeeded. Now trying level two boot
```

```
Has the following been output: "Boot". Done ? yes.
Has the following been output: ": up(0,0)vmunix". Done ? yes.
: Checking that UNIX has been loaded properly by the
: level two boot
Has the following been output:
"integer+integer+integer start hexadecimal". Done ? yes.
: Level two boot has succeeded. Now trying to run UNIX
Is UNIX (single) up and running ? yes.
Do you want to continue. ( yes/no ) ? yes.
Do you want to use the expert system for a consultation
or do you want to search for rules.
( consult/search ) ? search.
[ Break (level 1) ]
| ?- facts.
    { These are the facts learned during the consultation }
true(fact(tu58loaded))
true(fact(keyswitch(local)))
true(fact(uponline))
true(fact(bootdevice(D)))
true(fact(poweronaction(halt)))

[1] YES
| ?- ^D[ End break (level 1) ]
Do you want to continue. ( yes/no ) ? no.

[ Prolog execution halted ]
%
```

9.2.4. Law Expert.

This section contains a description of a law expert system written to use the three-valued Prolog implementation. [Sergot1982] has explored the prospects of using Prolog to represent the law. His initial work is to represent the bare facts but then this is extended to represent norms of conduct. He presents a system that implements library regulations which is extended to provide a way of keeping a consistent data base of facts.

The Expert System was written to investigate the applicability of three-valued Prolog for developing a law expert.

The raw data was provided in the form of several thousand statements of law which were created for a data base program. A small area from these statement was selected which dealt with lamps on vehicles and sale of liquor. The two areas were deliberately chosen to be unconnected. The raw statements are indexed by several fields containing the type of the offence, the year of the statute, the maximum fine etc. The statements also contain text describing the offence. To illustrate the way these statements were translated into rules for the expert system an example will be used. The example is six statements concerning the position of lamps on a horse drawn vehicle. The raw data is

972020020500 1050 9731006004012
BY ANY PERSON USING

a horse drawn vehicle whereon the obligatory front lamp is so fixed that where the vehicle has only one axle which is behind the axle.

972020020500 1050 9731006004013
BY ANY PERSON CAUSING TO BE USED

a horse drawn vehicle whereon the obligatory front lamp is so fixed that where the vehicle has only one axle which is behind the axle.

972020020500 1050 9731006004014
BY ANY PERSON PERMITTING TO BE USED

a horse drawn vehicle whereon the obligatory front lamp is so fixed that where the vehicle has only one axle

which is behind the axle.

972020020500 1050 9731006004015
BY ANY PERSON USING

a horse drawn vehicle having more than one axle whereon the obligatory front lamp is more than 1 ft 6 inches behind the front axle

972020020500 1050 9731006004016
BY ANY PERSON CAUSING TO BE USED

a horse drawn vehicle having more than one axle whereon the obligatory front lamp is more than 1 ft 6 inches behind the front axle

972020020500 1050 9731006004017
BY ANY PERSON PERMITTING TO BE USED

a horse drawn vehicle having more than one axle whereon the obligatory front lamp is more than 1 ft 6 inches behind the front axle

Since the statements are already split into three categories this will form the first set of rules.

```
true((offence(index(N,using),S,I,T,F,P,C) if
    offenceusing(N,S,I,T,F,P,C))).
true((offence(index(N,permitting),S,I,T,F,P,C) if
    offencepermitting(N,S,I,T,F,P,C))).
true((offence(index(N,causing),S,I,T,F,P,C) if
    offencecausing(N,S,I,T,F,P,C))).

true((offenceusing(N,S,I,T,F,P,C) if
    offenceupc(N,S,I,T,F,P,C) and
    query('any person using the vehicle'))).
true((offencepermitting(N,S,I,T,F,P,C) if
    offenceupc(N,S,I,T,F,P,C) and
    query('any person permitting the vehicle to be used')
).

true((offencecausing(N,S,I,T,F,P,C) if
    offenceupc(N,S,I,T,F,P,C) and
    query('any person causing the vehicle to be used'))).
```

The text of the statements are analysed and split up into groups of conditions. This converts the statements into the following rules.

```
true((offenceupc(N,S,I,T,F,P,Ty) if
    query('vehicle horse drawn') and
    wronglampposition(N,S,I,T,F,P,Ty))).

true((wronglampposition(25867,statute(1972,20,205,0),
 instrument(1973,1006,4,12), summary,50,none,
 ['Protection of public safety',
 'Avoidance of non-economic injury to offender or others']) if
    query('vehicle one axled') and
    query('front lamp behind axle'))).

true((wronglampposition(25879,statute(1972,20,205,0),
 instrument(1973,1006,4,15), summary,50,none,
 ['Protection of public safety',
 'Avoidance of non-economic injury to offender or others']) if
    query('front lamp greater than 1 ft 6 behind front axle'))).

false((fact('vehicle one axled') and
    fact('more than one axle on vehicle'))).
```

Note that some constrains are added which will stop the expert system from asking questions where a mutually exclusive question is true.

Using the shell allows two modes of operations. The first is consultation mode where the user is asked a series

of question to ascertain whether an offence is being committed. The second mode of operation is search mode where the description of offences satisfying several criteria are printed out.

9.2.4.1. Consultation Mode.

The program starts by printing a menu of the areas that are available. Execution is then continued from the procedure dealing with the selected area. The procedure scans all the offences asking questions - usually 2 or 3 per offence. It would be totally unacceptable to ask this many questions in a consultation so the three-valued Prolog implementation is used. This allows general rules such as :-

```
false((fact('two front lamps') and
       fact('one front lamp'))).
```

so that offences involving two front lamps are not considered if the user has indicated that the car considered in the consultation has only one front lamp.

9.2.4.2. Search Mode.

In this mode no questions are asked about unknown information so that all offences can be selected. The queries are formed using the Prolog syntax with a few user friendly extensions. An offence has seven fields which are :-

Index, Statute, Instrument, Type,
Fine, Permission and Category

with form

```
Index      - index(Number,Indextype).
Statute    - statute(Year,Section,Chapter,Part).
Instrument - instrument(Year,Section,Chapter,Part).
Type       - mode of prosecution - summary or crown
Fine       - value e.g. 50
Permission - permission for prosecution - none or dpp
Category   - category of law - [ Type1, Type2 ... ]
```

The offence is printed using the routine 'print_text(Index)' and for relations the following are provided :-

equal, not_equal, less_than, greater_than,
less_than_or_equal, greater_than_or_equal.

9.2.4.3. Examples.

To find and print all the offences which carry a maximum fine of over 50 pounds execute

```
?- offence(Index,_,_,_,Fine,_,_) and
        Fine greater_than 50 and
        print_text(Index).
```

As another example find all the offences created in 1972 and print the category associated with each

```
?- offence(_,statue(1972,_,_,_),_,_,_,_,Category).
```

The following text is output from a consultation to show the system in operation.

```
% prolog2
C-Prolog version 2.0
| ?- [consultlaw].
```

```
../shell consulted 2164 bytes 1.15 sec.
law consulted 6660 bytes 3.6 sec.
index consulted 14244 bytes 1.56667 sec.

***  LAW  demonstration program  ***

Do you want to use the expert system for a consultation
or do you want to search for rules.
( consult/search ) ? consult.
Do you want reasoning to be printed ? yes.
Which area of law is required
    frontlamps
    liquor
    others
Enter option.. frontlamps.
Is front lamp higher than 5 ft ? no.
Is vehicle horse drawn ? no.
Is lamp position not on opposite sides of vehicle ? yes.
Is lamps at different heights ? yes.
Is any person using the vehicle ? no.
Is one front lamp ? no.
front lamp higher than 5 ft is false
vehicle horse drawn is false
lamp position not on opposite sides of vehicle is true
lamps at different heights is true
Is any person permitting the vehicle to be used ? yes.

It is an offence for any person to permit the use of
a vehicle carrying two obligatory front lamps where they
are not fixed on opposite sides of the vehicle and at the
same height from the ground
Do you want to continue. ( yes/no ) ? yes.
Do you want to use the expert system for a consultation
or do you want to search for rules.
( consult/search ) ? search.
[ Break (level 1) ]
| ?- offence(Index,_,_,_,Fine,_,_) and
|       Fine greater_than 50 and
|       print_text(Index).

It is an offence for
a person by himself or by his servant or agent to sell
intoxicating liquor to any person in a licenced canteen
outside the permitted hours

Index = index(14445,statute)
Fine = 100 ;

It is an offence for
a person to consume intoxicating liquor in a licenced
canteen outside the permitted hours

Index = index(14448,statute)
Fine = 100 ;
```

```
[1] DONT KNOW
| ?- halt.

[ Prolog execution halted ]
%
```

10. Conclusion.

The first part of the thesis presented an implementation of a Z80 Prolog compiler. The Z80 implementation is both reasonably fast and space efficient. The main advantage over an interpreter is the speed and space improvements for the resultant code but the disadvantage is the time required to compile the code. If a program has been developed and tested and will therefore only be compiled once the overhead is not significant but for development it would be painful.

The implementation is far from optimum in that it does not include the following features :-

[1] Indexing of clauses

[2] Tail recursive optimisation

[3] Garbage collection

[4] Mode declarations

These improvements are described in [Warren1977,Warren1980,Mellish1981,Bruynooghe1984a].

The new structure sharing method for address wide machines does not incur excessive time penalties but obviously a comparative test is necessary to evaluate the relative time/space trade offs.

One possible improvement for this implementation is the

incorporation of a global stack reclamation algorithm that does not use garbage collection. The global stack would be kept as a doubly linked list of global frames which also included a link count for each frame. When ever a variable was unified with an item on the global stack the link count for the frame used would be increased. This operation would need to be trailed on a new global trail so it could be undone on backtrack. When a local frame is discarded by the determinate exit of the procedure the frame contents would be scanned to see if any references to global variables are being discarded. If so the global frame link count would be decremented. If the count reaches zero the global frame can be unlinked and returned to the free space list. The discarded global frame would also be scanned to find any global references and the process repeated for any link count reaching zero.

One problem with this method is the ordering on the global stack is removed so the test for seniority in unification between two global variables is impossible. The link count guarantees that no dangling references to the discarded frame are left but there is no indication which frame might be discarded first. One possible solution is the look at the link count and assume the frame with the lower link count is more likely to be discarded first. This cannot be guaranteed so the unification must always be trailed.

The second part of the thesis has presented a three-

valued Prolog implementation. The modifications necessary to Prolog are not difficult to implement and the transformation of rules and fact straightforward. The addition of false facts to the data base enables the representation of an open world. This could be used even when relations are mathematically defined as either true or false such as 'addition'. The values true or false would be given when it was possible to prove the relation either way and the value unknown would be given when practical limitations, such as integer overflow or underflow, restrict the calculation. The system is therefore aware of its own limitations and could try the calculation another way. The use of rules that include either true or false goals overcome some of the problems of negation by failure when uninstantiated variables are involved. In IC-Prolog, micro-Prolog, C-Prolog and York Prolog it is not possible to have a rule of the form

```
under(X,Y) :- not on(X,Y).
```

if either of the variables X and Y are uninstantiated. Using the three-valued implementation it is possible to have a rule

```
true((under(X,Y) or on(X,Y))).
```

which translates to

```
true(under(X,Y)) :- false(on(X,Y)).
true(on(X,Y)) :- false(under(X,Y)).
```

with no restrictions. The two forms are not identical how-

ever, since the second form is for an open world data base.

The three-valued implementation has also enabled the use of disjunctive conclusions (as above) which makes it possible to express integrity constraints concisely. Also the 'false' form makes it possible to add clause like

```
false(switch(on) and switch(off)).
```

which were previously impossible.

10.1. Further Work.

Based on the two parts of this thesis the further work falls into two categories both centred around the language Prolog. The first area is the continued development of the Prolog compiler by adding more features such as tail recursion and mode declarations. This would be a prerequisite for developing a non structure sharing compiler to compare with the offset structure sharing. It would be useful to compare the space-time trade offs for each implementation. A third compiler could then be developed that required type declarations so that structures could be treated in a similar manner to other block structured language (e.g. PASCAL). This seems to be where Turbo Prolog gains its good speed and space performance. It would also be interesting to see if a hybrid can be developed which will use the best representation possible given a certain type of program with certain extra information such as types or modes. The development of an automatic mode declaration program based

on the work by Mellish could provide a good starting point for development of an automatic type declaration program. This would be able to deduce types from limited information given in the program. This could be augmented by information given by the programmer.

From the implementations seen it appears that a useful standard for module information is necessary. The work by [Goguen1984] provides useful information about modules and also indicates that generics could be incorporated. The standardisation of module interfaces and the provision of types should ease the interface between Prolog and other languages.

Further investigation and development is needed on the automatic global stack reclamation method. This might be useful for some types of programs but if good mode declarations are provided the amount of stack eligible for reclamation is reduced. The use of this method might be adversely affected by the use of types in a program and therefore the method of representing complex structures.

The second area of further work concerns the three-valued implementation. A full rigorous analysis is needed for the rule of self-implication. This has been used successfully in the three-valued interpreter but might have unseen effects in other areas. It is necessary to provided the distinction between the two possible interpretations but other methods of deriving a representation of three-valued

implication might be possible.

The limitations of the system should be investigated further and analysis of whether these limitations have serious effects on large expert systems. The two expert systems developed show that the three-valued implementation can be used successfully on small projects but usefulness on large systems has still to be investigated. It might be that the limitations become unwieldy in large systems and interact in such a complex way that the knowledge engineer fails to take the limitations into account.

To aid in the analysis and usefulness of the three-valued interpreter it should be made more portable. It could then be tested with other Prolog implementations.

BIBLIOGRAPHY

Ted%nmsu.csnet@CSNET-RELAY.ARPA",, "LIPS," PROLOG Digest, vol. 4, no. 24, RESTIVO@SU-SCORE.ARPA, 23 Jun 1986. Issued via electronic mail

Aikens, J., "Prototypical knowledge for Expert systems," Artificial Intelligence, vol. 20, no. 2, pp. 163-210, Feb 1983.

Araya, Agustin A., "Learning modalities in Artificial Intelligence systems : A framework and a review," Actas de la segunda conferencia internacional en ciencia de la computacion, pp. 53 - 84, August 1982.

Barber, Gerald, "Embedding knowledge in a workstation," in Office information systems, ed. N. Naffah, pp. 341 - 354, North Holland Publishing Company, 1982.

Berry-Rogghe, G. L., M. Kolvenbach, and H. D. Lutz, "Interacting with PLIDIS a deductive question answering system for German," in Natural Language Question Answering systems, ed. Leonard Bolc, 1980.

Borland,, Turbo Prolog Owner's Handbook, Borland International Inc, April 1986.

Bossu, G. and P. Siegel, "Saturation, Nonmonotonic Reasoning and the Closed-World Assumption," Artificial Intelligence, vol. 25, no. 1, January 1985.

Bowen, D. L., "DECsystem-10 PROLOG User's Manual," DAI Occasional Paper, no. 27, Department of Artificial Intelligence, University of Edinburgh, November 1982a.

Bowen, Kenneth A., "Amalgamating language and metalanguage in logic programming," in Logic Programming, ed. K. L. Clark and S. A. Tarnlund, Academic Press, 1982b.

Bruynooghe, M., "Garbage collection in PROLOG interpreters," in Implementations of PROLOG, ed. Campbell, pp. 259 - 267, Ellis Horwood Limited, Chichester, 1984a.

Bruynooghe, M. and L. M. Pereira, "Deduction revision by intelligent backtracking," in Implementations of PROLOG, ed. Campbell, pp. 194 - 215, Ellis Horwood Limited, Chichester, 1984b.

Buchanan, B. G., "New research on expert systems," in Machine Intelligence 10, ed. J. E. Hayes, D. Michie and Y-H. Pao, pp. 269 - 300, Ellis Horwood Limited, 1982.

Campbell, J. A., "Expert systems," IUCC Bulletin, vol. 5, no. 2, pp. 63 - 67, Summer 1983.

Champeaux, D. de, "Bidirectional heuristic search again," Journal of the association for Computing Machinery, vol. 30, pp. 22-32, Jan 1983.

Clark, K. L. and F. G. McCabe, "The Control Facilities of IC-Prolog," in Expert Systems in the Micro-electronic Age, ed. Donald Michie, pp. 122 - 149, Edinburgh University Press, 1979.

Clark, K. L., F. G. McCabe, and S. Gregory, "IC-PROLOG Language Features," Research Report No. DOC 81/31, Department of Computing, Imperial College of Science and Technology, University of London, October 1981.

Clark, K. L. and F. G. McCabe, "Prolog : a language for implementing expert systems," in Machine Intelligence 10, ed. J. E. Hayes, D. Michie, Y-H. Pao, pp. 455 - 475, Ellis Horwood Limited, 1982.

Clark, K. L. and F. G. McCabe, micro-PROLOG : Programming in Logic, Prentice-Hall, 1984.

Clark, Keith and Steve Gregory, "Notes on the implementation of PARLOG," Journal of Logic Programming, vol. 2, no. 1, pp. 17 - 42, 1985.

Clark, Keith L., "Negation as failure," in Logic and Data Bases, ed. Herve Gallaire and Jack Minker, 1978.

Clocksin, W. F. and C. S. Mellish, Programming in PROLOG, Springer-Verlag, 1981.

Colmerauer, A., "Metamorphosis Grammars," in Natural Language Communication with Computers. Lecture Notes in Computer Science no. 63, ed. Leonard Bolc, pp. 133 - 189, 1978.

Colmerauer, A. and J. F. Pique, "About Natural Logic," in Advances in Data Base Theory, ed. H. Gallaire, J. Minker and J. M. Nicolas, 1981a.

Colmerauer, A., H. Kanoui, and M. Van Caneghem, "Last steps towards an ultimate PROLOG," IJCAI, vol. 2, pp. 947 - 948, 1981b.

Colmerauer, A., "An Interesting Subset of Natural Language," in Logic Programming, ed. K. L. Clark and S. A. Tarnlund, Academic Press, 1982.

Colmerauer, Alain, Henry Kanoui, and Michel Van Caneghem, "PROLOG theoretical bases and current developments,"

TSI - Technique et science informatiques, vol. 2, no. 4, pp. 271 - 311, 1983.

Colmerauer, Alain, "PROLOG in 10 figures," Communications of the ACM, vol. 28, no. 12, pp. 1296 - 1324, December 1985.

Conery, John S. and Dennis F. Kibler, "Parallel interpretation of logic programs," Proceeding of ACM conference on functional programming and computer architecture, pp. 163 - 170, 1981.

Cox, P. T., "Finding backtrack points for intelligent backtracking," in Implementations of PROLOG, ed. Campbell, pp. 216 - 233, Ellis Horwood Limited, Chichester, 1984.

Dahl, Veronica, "Quantification in a three-valued logic for natural language question-answering systems," Proceedings of the 6th international joint conference on artificial intelligence, pp. 182 - 187, 1979.

De, Prabuddha and Arun Sen, "Knowledge representation of a data base by a semantic network," International Journal of Policy analysis and Information systems, vol. 6, no. 1, pp. 25 - 45, March 1982.

Domolki, Balint and Peter Szeredi, "PROLOG in practice," in Information processing 83, ed. R. E. A. Mason, pp. 627 - 636, Elsevier Science Publishers, 1983.

Drazovich, Robert J., Brian P. McCune, and J. Roland Payne, "Artificial Intelligence : An emerging military technology," EASCON 82 - 15th annual electronics and aerospace systems conference, pp. 341 - 348, 1982.

Efstathiou, H. J., A practical development of multi attribute decision making using fuzzy set theory, Department of Computing PhD, University of Durham, 1979.

Emden, M. H. van, "An interpreting algorithm for PROLOG programs," in Implementations of PROLOG, ed. Campbell, pp. 93 - 110, Ellis Horwood Limited, Chichester, 1984.

Filgueiras, M., "A PROLOG interpreter working with infinte terms," Implementation of PROLOG, pp. 250 - 258, Ellis Horwood Limited, Chichester, 1984.

Fischer, Herm, "Borland Turbo Prolog," PROLOG Digest, vol. 4, no. 18, RESTIVO@SU-SCORE.ARPA, 10 Jun 1986. Issued via electronic mail

Forsyth, R., Expert Systems - Principles and case studies, 1984.

Gallaire, Herve, Jack Minker, and Jean Marie Nicolas, "An overview and introduction to logic and data bases," in Logic and Data Bases, ed. Herve Gallaire and Jack Minker, 1978.

Gaschnig, J., "Application of the PROSPECTOR system to geological exploration problems," in Machine Intelligence 10, ed. J. E. Hayes, D. Michie and Y-H. Pao, pp. 301 - 324, Ellis Horwood Limited, 1982.

Gevarter, W. B., "An overview of Expert systems," National Bureau of standards report N82-29899, May 1982.

Goguen, Joseph A. and Jose Meseguer, "Equality, type, modules and (why not?) generics for logic programming," Journal of Logic Programming, vol. 1, no. 2, pp. 179 - 210, 1984.

Goto, Shigeki, "DURAL: an extended PROLOG language," in Lecture Notes in Computer Science 147 - RIMS Symposia on software science and engineering, ed. Eiichi Goto et al..

Gregory, Steve, "Sequential PARLOG Machine," PROLOG Digest, vol. 4, no. 14, RESTIVO@SU-SCORE.ARPA, 10 May 1986. Issued via electronic mail

Haridi, S. and D. Sahlin, "Efficient implementation of unification of cyclic structures," Implementation of PROLOG, pp. 234 - 249, Ellis Horwood Limited, Chichester, 1984.

Hart, Peter E., Richard O. Duda, and Kurt Konolige, "A computer-based consultant for mineral exploration," SRI Project 6415, SRI International, April 1978.

Hauptmann, Alexander G. and Bert F. Green, "A comparison of command, menu-selection and natural-language computer programs," Behaviour and Information Technology, vol. 2, no. 2, pp. 163 - 178, 1983.

Hayes-Roth, Frederick, "The knowledge-based expert system: a tutorial," Computer, pp. 11 - 28, September 1984.

Heines, Jesse M., Jonathan Briggs, and Richard Ennals, "Logic and recursion: the PROLOG twist," Creative Computing, pp. 220 - 226, November 1983.

Jensen, Kathleen and Niklaus Wirth, in PASCAL User Manual and Report, Springer-Verlag, 1975.

Kernighan, Brian W. and Dennis M. Ritchie, The C Programming Language, Prentice-Hall, 1978.

Klahr, P., "A Deductive system for Natural Language Question Answering," in Natural Language Question Answering Systems, ed. Leonard Bolc, 1980.

Kleene, Stephen Cole, Introduction to Metamathematics, North Holland, 1962.

Kluzniak, F. and S. Szpakowicz, "PROLOG - a Panacea ?," in Implementations of PROLOG, ed. Campbell, pp. 71 - 84, Ellis Horwood Limited, Chichester, 1984.

Kluzniak, Feliks and Stanislaw Szpakowicz, PROLOG for Programmers, Academic Press, 1985.

Kopriva, Jiri , "Fuzzy logic and superpruning," Computers and Artificial Intelligence, vol. 2, no. 1, pp. 3 - 12, February 1983.

Korf, Richard E., "Depth-first Iterative-deepening : An optimal admissible tree search," Artificial Intelligence, vol. 27, no. 1, pp. 97 - 109, North-Holland, September 1985.

Kowalski, R., "Logic for Problem solving," in Artificial Intelligence Series 7, ed. Nils J. Nilsson, 1979.

Kowalski, R., "Logic as a Computer Language," in Logic Programming, ed. K. L. Clark and S. A. Tarnlund, 1982.

Langley, P. W., G. L. Bradshaw, and H. L. Simon, "Data driven and Expectation driven discovery of Empirical Laws," U.S. Government report AD-A120 950, Carnegie - Mellon University, 1982.

Lehnert, W. G., "Question Answering in Natural Language processing," in Natural Language Question Answering Systems, ed. Leonard Bolc, 1980.

Lehnert, Wendy G., Michael G. Dyer, Peter N. Johnson, C. J. Yang, and Steve Harley, "BORIS - An experiment in in-depth understanding of narratives," Artificial Intelligence, vol. 20, no. 1, pp. 15 - 62, Jan 1983.

Leith, Philip, "Hierarchically structured production rules," The Computer Journal, vol. 26, no. 1, pp. 1 - 5, February 1983.

Lenat, D., "Automated Theory Formation in Mathematics," Proceedings of the 5th International Joint Conference on Artificial Intelligence, 1977.

Lewis, Clarence Irving and Cooper Harold Langford, Symbolic Logic, Dover Publications, 1959.

Liardet, M., The Logical Solution, Review Article.

Lukasiewicz, Jan, Selected Works, North Holland, 1970.

McDermott, D., "Contexts and Data Dependencies: A synthesis," IEEE transactions on Pattern Analysis and Machine Intelligence, vol. 5, no. 3, May 1983.

Meier, Micha, "LIPS again," PROLOG Digest, vol. 4, no. 11, RESTIVO@SU-SCORE.ARPA, 10 Mar 1986. Issued via electronic mail

Mellish, C. S., "An alternative to structure-sharing in the implementation of a PROLOG interpreter," DAI Research Paper, no. 150, Department of Artificial Intelligence, University of Edinburgh, 1980.

Mellish, C. S., "The automatic generation of mode declarations for PROLOG programs," DAI Research Paper, no. 163, Department of Artificial Intelligence, University of Edinburgh, August 1981.

Mellish, C. S., "Some global optimizations for a PROLOG compiler," Journal of Logic Programming, vol. 2, no. 1, pp. 43 - 66, 1985.

Mero, Laszlo, "Experiments with tree-pruning strategies," Computers and Artificial Intelligence, vol. 2, no. 1, pp. 19 - 33, February 1983.

Michie, Donald, "A personal view of expert systems," IUCC Bulletin, vol. 5, no. 2, pp. 68 - 73, 1983.

Moss, Chris, "Alternative notation for lists," PROLOG Digest, vol. 4, no. 24, RESTIVO@SU-SCORE.ARPA, 10 Jul 1986. Issued via electronic mail

Nau, D., "Expert Computer Systems," Computer, vol. 16, no. 2, pp. 63-85, Feb 1983.

Pereira, F., C-PROLOG User's Manual, Version 1.5, Edinburgh Computer Aided Architectural Design, University of Edinburgh, 1984a.

Pereira, L. M., "Logic control with logic," in Implementations of PROLOG, ed. Campbell, pp. 177 - 193, Ellis Horwood Limited, Chichester, 1984b.

Poe, Michael D., Roger Nasr, Janet Potter, and Janet Slinn, "A KWIC (key word in context) bibliography on PROLOG and logic programming," Journal of Logic Programming, vol. 1, no. 1, pp. 81 - 142, 1984.

Porto, A., "EPILOG: a language for extended programming in logic," in Implementations of PROLOG, ed. Campbell, pp. 268 - 278, Ellis Horwood Limited, Chichester, 1984.

Quinlan, J. R., "INFERNO : A Cautious approach to uncertain Inference," The Computer Journal, vol. 26, no. 3, pp. 255-269, Aug 1983.

Roberts, G., An implementation of PROLOG, MSc Thesis in Computer Science, University of Waterloo, Ontario, 1977.

Robinson, J. A., "A Machine-Oriented Logic Based on the Resolution Principle," Association for Computing Machinery, vol. 12, pp. 23-41, 1965.

Robinson, J. A., "The logical basis of programming by assertion and query," in Expert Systems in the Micro Electronic Age, ed. D. Michie, pp. 105 - 111, Edinburgh University Press, 1979a.

Robinson, J. A., Logic : Form and Function, Edinburgh University Press, 1979b.

Roizen, I., "A minimax algorithm better than alpha - beta ? Yes and No," Artificial Intelligence, vol. 21, no. 1,, pp. 199-220, 2 March 1983.

Rosser, J. B. and A. R. Turquette, "Many-valued Logics," in Studies in Logic and the Foundations of Mathematics, ed. L. E. J. Brouwer, E. W. Beth and A. Heyting, 1952.

Sergot, M., "Prospects for representing the law as logic programs," in Logic Programming, ed. K. L. Clark and S. A. Tarnlund, Academic Press, 1982.

Shapiro, Ehud, "The Logix System," PROLOG Digest, vol. 4, no. 27, RESTIVO@SU-SCORE.ARPA, 14 Jul 1986. Issued via electronic mail

Shapiro, S. C., "Numerical Quantifiers and their use in reasoning with negative information," Proceedings of the 6th International Joint Conference on Artificial Intelligence IJCAI, vol. 2, p. 791, 1979.

Shimura, Masamichi, "Resolution in a new modal logic," Proceedings of the 6th International Joint Conference on Artificial Intelligence, vol. 2, pp. 809 - 814, 1979.

Shortliffe, E. H., "Computer-Based Medical Consulations : MYCIN," in Artificial Intelligence Series 2, ed. Nils J. Nilsson, North Holland, May 1976.

Simon, Herbert A., "Artificial intelligence research strategies in the light of AI models of scientific discovery," Naval Research Reviews, pp. 2 - 16, 1982.

Smith, Bruce, "List of Prologs," SCORE:<Prolog>Prolog.NImplementations, April 1986. File available for FTP transfer

Spiers, J. D., G. Stevenson, and K. I. McKenzie, "Informatology - transforming data into information," Information Age, vol. 5, no. 3, pp. 154 - 156, July 1983.

Spivey, J. M., University of York Portable PROLOG System Release 1 User's Guide, March 1982.

Sterling, Leon, "Logical levels of problem solving," Journal of Logic Programming, vol. 1, no. 2, pp. 151 - 163, 1984.

Suwa, M., A. C. Scott, and E. H. Shortliffe, "An approach to verifying completeness and consistency in a rule based Expert System," U.S. Government report AD-A120 806, Stanford University, August 1982.

Swinson, Peter S. G., Fernando C. N. Pereira, and Aart Bijl, "A fact dependency system for the logic programmer," Computer Aided Design, vol. 15, no. 4, pp. 235 - 243, July 1983.

Tick, Evan and David H. D. Warren, "Towards a pipelined PROLOG processor," International symposium on Logic Programming, pp. 29 - 40, 6 February 1984.

Tufis, Dan, "SDLR: a dialogue system for Rumanian language," Computers and Artificial Intelligence, vol. 2, no. 3, pp. 269 - 278, June 1983.

Wallis, J. W. and E. H. Shortliffe, "Explanatory power for medical Expert systems : Studies in the representation of casual relationships for clinical consultation," U.S. Government report AD-A120 936, Stanford University, August 1982.

Warren, D. H. D., "Implementing PROLOG - Compiling predicate logic programs," DAI Research Report, no. 39 and 40, Department of Artificial Intelligence, University of Edinburgh, May 1977.

Warren, D. H. D., "Prolog on the DEC system 10," in Expert Systems in the Micro Electronic Age, ed. Donald Michie, pp. 112 - 121, Edinburgh University Academic Press, 1979.

Warren, D. H. D., "An improved PROLOG implementation which optimises tail recursion," DAI Research Paper, no. 141, Department of Artificial Intelligence, University of Edinburgh, July 1980.

Warren, D. H. D., "Efficient processing of Interactive Relational Database queries expressed in logic," DAI Research Paper, no. 156, Department of Artificial Intelligence, University of Edinburgh, February 1981.

Warren, D. H. D., "Issues in natural language access to databases from a logic programming perspective," 20th annual meeting of the association of computational linguistics, pp. 63 - 66, 1982a.

Warren, D. H. D., "Higher-order extensions to Prolog : are they needed ?," in Machine Intelligence 10, ed. J. E. Hayes, D. Michie, Y-H. Pao, pp. 441 - 454, Ellis Horwood Limited, 1982b.

Warren, David Scott, "Efficient PROLOG memory management for flexible control strategies," International symposium on Logic Programming, pp. 198 - 202, 6 February 1984.

Webster, Robin, "Expert systems," Personal Computer World, pp. 118 - 122, January 1983.

Winston, Patrick H., "Learning new principles from precedents and exercises," Artificial Intelligence, vol. 19, no. 3, pp. 321 - 350, November 1982.

ZILOG,, Z80 Assembly Language Programming Manual, 1980.

Appendix A - Proofs

Proof of

```
TRUE(A IF B) =
        and(or(TRUE(A),not(TRUE(B))),
            or(FALSE(B),not(FALSE(A)))
           )
```

{ T = TRUE, F = FALSE U = UNKNOWN }

A	B	T(A IF B)	or(T(A), not(T(B))) (1)	or(F(B), not(F(A))) (2)	(1) and (2)
0	0	1	1	1	1
0	w	0	1	0	0
0	1	0	0	0	0
w	0	1	1	1	1
w	w	1	1	1	1
w	1	0	0	1	0
1	0	1	1	1	1
1	w	1	1	1	1
1	1	1	1	1	1

Proof of

```
FALSE(A AND B) =
        and(or(not(TRUE(A)),FALSE(B)),
            or(not(TRUE(B)),FALSE(A)),
            or(not(UNKNOWN(A)),not(UNKNOWN(B)))
           )
```

A	B	F(A AND B)	or(F(B), not(T(A))) (1)	or(F(A), not(T(B))) (2)	or(not(U(A) not(U(B)))
0	0	1	1	1	1
0	w	1	1	1	1
0	1	1	1	1	1
w	0	1	1	1	1
w	w	0	1	1	0
w	1	0	1	0	1
1	0	1	1	1	1
1	w	0	0	1	1
1	1	0	0	0	1

Appendix B - An Evaluator

```
simple(0).
simple(w).
simple(1).


eval(not(0),1).
eval(not(w),w).
eval(not(1),0).
eval(not(X),A) :-
        eval(X,A1), eval(not(A1),A).


eval(true(0),0).
eval(true(w),0).
eval(true(1),1).
eval(true(X),A) :-
        eval(X,A1), eval(true(A1),A).


eval(and(0,0),0).
eval(and(0,w),0).
eval(and(0,1),0).
eval(and(w,0),0).
eval(and(w,w),w).
eval(and(w,1),w).
eval(and(1,0),0).
eval(and(1,w),w).
eval(and(1,1),1).
eval(and(X,Y),A) :-
        simple(Y), eval(X,A1), eval(and(A1,Y),A).
eval(and(X,Y),A) :-
        simple(X), eval(Y,A2), eval(and(X,A2),A).
eval(and(X,Y),A) :-
        eval(X,A1), eval(Y,A2), eval(and(A1,A2),A).


eval(if(0,0),1).        /* if defined as A >= B */
eval(if(0,w),0).
eval(if(0,1),0).
eval(if(w,0),1).
eval(if(w,w),1).
eval(if(w,1),0).
eval(if(1,0),1).
eval(if(1,w),1).
eval(if(1,1),1).
eval(if(X,Y),A) :-
        simple(Y), eval(X,A1), eval(if(A1,Y),A).
eval(if(X,Y),A) :-
        simple(X), eval(Y,A2), eval(if(X,A2),A).
eval(if(X,Y),A) :-
        eval(X,A1), eval(Y,A2), eval(if(A1,A2),A).
```

```
eval(or(P,Q),A) :-
        eval(not(and(not(P),not(Q))), A).

eval(false(P),A) :-
        eval(true(not(P)),A).

eval(unknown(P),A) :-
        eval(not(or(true(P),false(P))),A).


oner :-
        simple(P), simple(Q),
        eval(not(and(not(P),not(Q))), Z),
        write(P),
        write(Q),
        write(Z),
        nl, fail.


twor :-
        simple(P),
        eval(true(not(P)),Z),
        write(P),
        write(Z),
        nl, fail.


threer :-
        simple(P), simple(Q),
        eval(not(or(true(P),false(Q))),Z),
        write(P),
        write(Q),
        write(Z),
        nl, fail.


fourr :-
        simple(A), simple(B),
        eval(and(true(or(true(A),not(true(B)))),
                  and(true(or(false(B),not(false(A)))),
                      false(and(unknown(A),unknown(B)))
                     )
                ),
              Z),
       write(A),
       write(B),
       write(Z),
       nl,
       fail.
```

```
fourl :-
        simple(A), simple(B),
        eval(true(or(A,not(B))),
             Z),
        write(A),
        write(B),
        write(Z),
        nl,
        fail.


fiver :-
     simple(A), simple(B), simple(C),
     eval(and(true(or(true(A),
                 not(and(true(B),true(C))))),
           and(true(or(false(B),
                 not(and(false(A),true(C))))),
           and(true(or(false(C),
                 not(and(false(A),true(B))))),
           and(false(and(unknown(A),
                 and(unknown(B),not(false(C))))),
           and(false(and(unknown(A),
                 and(unknown(C),not(false(B))))),
               false(and(unknown(B),
                 and(unknown(C),not(true(A))))
                  )
                  )
                 )
                 )
                 ),
               Z),
       write(A),
       write(B),
       write(C),
       write(Z),
       nl,
       fail.


fivel :-
        simple(A), simple(B), simple(C),
        eval(true(or(A,not(and(B,C)))),
             Z),
        write(A),
        write(B),
        write(C),
        write(Z),
        nl,
        fail.
```

```
sixr :-
        simple(A), simple(B),
        eval(and(or(true(A), not(false(B))),
                and(or(false(A), not(true(B))),
                and(or(false(A), not(false(B))),
                and(or(true(B), not(true(A))),
                and(or(true(B), not(false(A))),
                and(or(false(B), not(true(A))),
                false(and(unknown(A),unknown(B)))
                    )
                    )
                   )
                   )
                   )
                ),
             Z),
          write(A),
          write(B),
          write(Z),
          nl,
          fail.


sixl :-
        simple(A), simple(B),
        eval(false(or(A,not(B))),
             Z),
        write(A),
        write(B),
        write(Z),
        nl,
        fail.


sixe :-
        simple(A), simple(B),
        eval(false(or(A,false(B))),Z),
        write(A),
        write(B),
        write(Z),
        nl, fail.


sevenr :-
        simple(A),
        eval(and(false(false(A)),false(unknown(A))),Z),
        write(A),
        write(Z),
        nl, fail.
```

```
eightr :-
        simple(A), simple(B),
        eval(false(or(and(false(A),unknown(B)),
                or(and(false(A),true(B)),
                   or(and(unknown(A),unknown(B)),
                      and(unknown(A),true(B))
                     )
                  )
                )),
                Z),
        write(A),
        write(B),
        write(Z),
        nl, fail.


niner :-
        simple(A), simple(B),
        eval(and(or(false(A),not(true(B))),
                and(or(true(A),not(false(B))),
                   and(or(false(A),not(false(B))),
                          not(unknown(B))
                         )
                  )
                ),
              Z),
         write(A),
         write(B),
         write(Z),
         nl,
         fail.


unknowns :-
        simple(A), simple(B), simple(C),
        simple(D), simple(E), simple(F),
        eval(if(or(or(A,B),C),and(and(D,E),F)),G),
        write(A),
        write(B),
        write(C),
        write(D),
        write(E),
        write(F),
        write(G),
        nl, fail.
```

Evaluator with Dynamic Definition of Implication

```
simple(0).
simple(w).
simple(1).
```

```
eval(L,true(0),0) :- !.
eval(L,true(w),0) :- !.
eval(L,true(1),1) :- !.
eval(L,true(X),A) :-
        eval(L,X,A2), eval(L,true(A2),A), !.


eval(L,false(0),1) :- !.
eval(L,false(w),0) :- !.
eval(L,false(1),0) :- !.
eval(L,false(X),A) :-
        eval(L,X,A2), eval(L,false(A2),A), !.


eval(L,not(0),1) :- !.
eval(L,not(w),w) :- !.
eval(L,not(1),0) :- !.
eval(L,not(X),A) :-
        eval(L,X,A2), eval(L,not(A2),A), !.


eval(L,and(0,0),0) :- !.
eval(L,and(0,w),0) :- !.
eval(L,and(0,1),0) :- !.
eval(L,and(w,0),0) :- !.
eval(L,and(w,w),w) :- !.
eval(L,and(w,1),w) :- !.
eval(L,and(1,0),0) :- !.
eval(L,and(1,w),w) :- !.
eval(L,and(1,1),1) :- !.
eval(L,and(X,Y),A) :-
        simple(Y), eval(L,X,A1), eval(L,and(A1,Y),A), !.
eval(L,and(X,Y),A) :-
        simple(X), eval(L,Y,A2), eval(L,and(X,A2),A), !.
eval(L,and(X,Y),A) :-
        eval(L,X,A1), eval(L,Y,A2),
        eval(L,and(A1,A2),A).


eval([eval(imp(X,Y),Z)|Tail],imp(X,Y),Z) :- !.
eval([Head|Tail],imp(X,Y),Z) :-
        eval(Tail,imp(X,Y),Z), !.
eval(L,imp(X,Y),A) :-
        simple(Y), eval(L,X,A1), eval(L,imp(A1,Y),A), !.
eval(L,imp(X,Y),A) :-
        simple(X), eval(L,Y,A2), eval(L,imp(X,A2),A), !.
eval(L,imp(X,Y),A) :-
        eval(L,X,A1), eval(L,Y,A2),
        eval(L,imp(A1,A2),A), !.
```

```
/* implist(A,B,C,D,E,F,G,H,I,
        [eval(imp(0,0),A),
        eval(imp(0,w),B),
        eval(imp(0,1),C),
        eval(imp(w,0),D),
        eval(imp(w,w),E),
        eval(imp(w,1),F),
        eval(imp(1,0),G),
        eval(imp(1,w),H),
        eval(imp(1,1),I)]). */


implist(B,D,E,F,H,
/* Defined to be compatible with two-valued imp */
        [eval(imp(0,0),1),
        eval(imp(0,w),B),
        eval(imp(0,1),1),
        eval(imp(w,0),D),
        eval(imp(w,w),E),
        eval(imp(w,1),F),
        eval(imp(1,0),0),
        eval(imp(1,w),H),
        eval(imp(1,1),1)]).


find_implication(Num) :-
        ponens(Num,R1,X1,Y1),
        tolens(Num,R2,X2,Y2),
        syllogism(Num,R3,X3,Y3,Z3),
        selfimp(Num,R4,X4),
        write('Defn '), write(R1), nl,
        write(R2), nl, write(R3), nl, write(R4), nl, nl,
        simple(A), simple(B), simple(C),
        simple(D), simple(E), /* simple(F),
        simple(G), simple(H), simple(I),
        implist(A,B,C,D,E,F,G,H,I,List), */
        implist(A,B,C,D,E,List),
        nl, write(List),
        check(List,R1,X1,Y1),
        nl, write('Holds for Modus Ponens'),
        check(List,R2,X2,Y2),
        write(', Modus Tolens'),
        check(List,R3,X3,Y3,Z3),
        write(', Syllogism'),
        check(List,R4,X4),
        write(', Imps'),
        fail.
find_implication(Num).  /* Finished all possibilities */
```

```
check(List,Rule,X) :-
        simple(X),
        eval(List,Rule,Z),
        Z == 1, !, fail.
check(List,Rule,X).


check(List,Rule,X,Y) :-
        simple(X), simple(Y),
        eval(List,Rule,Z),
        /* write(X), write(Y), write(Z), nl, */
        Z == 1, !, fail.
check(List,Rule,X,Y).


check(List,Rule,A,B,C) :-
        simple(A), simple(B), simple(C),
                                /* Law of syllogism */
        eval(List,Rule,Z),
        Z == 1, !, fail.
check(List,Rule,A,B,C).


ponens(1,imp(and(X,imp(X,Y)),Y),X,Y).
ponens(2,imp(and(true(X),true(imp(X,Y))),true(Y)),X,Y).
ponens(3,imp(and(true(X),true(imp(X,Y))),true(Y)),X,Y).


tolens(1,imp(and(not(Y),imp(X,Y)),
        not(X)),X,Y).
tolens(2,imp(and(false(Y),true(imp(X,Y))),
        false(X)),X,Y).
tolens(3,imp(and(not(true(Y)),true(imp(X,Y))),
        not(true(X))),X,Y).


syllogism(1,imp(and(imp(A,B),imp(B,C)),
        imp(A,C)),A,B,C).
syllogism(2,imp(and(true(imp(A,B)),true(imp(B,C))),
        true(imp(A,C))),A,B,C).
syllogism(3,imp(and(true(imp(A,B)),true(imp(B,C))),
        true(imp(A,C))),A,B,C).


selfimp(1,imp(X,X),X).
selfimp(2,imp(X,X),X).
selfimp(3,imp(X,X),X).


one :- system("date > start1"),
        tell(output1), find_implication(1), told,
        system("date >> start1").
```

```
two :- system("date > start2"),
       tell(output2), find_implication(2), told,
       system("date >> start2").


three :- system("date > start3"),
       tell(output3), find_implication(3), told,
       system("date >> start3").
```

Appendix C - Compiler Output

Source Code

```
external(sort,get,put,less,var,nonvar,add,user1,user2,user3).

sort(Lo,L) if qsort(Lo,L,nil).

qsort(cons(X,L),R,Ro) if
    partition(L,X,Li,Lt) and
    qsort(Lt,Ri,Ro) and
    qsort(Li,R,cons(X,Ri)).
qsort(nil,R,R).

partition(cons(X,L),Y,cons(X,Li),Lt) if
    less(X,Y) and ! and partition(L,Y,Li,Lt).
partition(cons(X,L),Y,Li,cons(X,Lt)) if
    partition(L,Y,Li,Lt).
partition(nil,X,nil,nil).
```

PLM Output Code

```
label(0)
        uatom(0,1)
        uatom(1,2)
        uatom(2,3)
        uatom(3,4)
        uatom(4,5)
        uatom(5,6)
        uatom(6,7)
        uatom(7,8)
        uatom(8,9)
        uatom(9,10)
        init(0,0)
        localinit(0,0)
        neck(0,0)
        foot(10)
label(1)
        uvar(0,local,0)
        uvar(1,local,1)
        init(0,0)
        localinit(2,2)
        neck(2,0)
        call(11)
        type(0,local,0)
        type(1,local,1)
        atom(2,12)
```

```
        foot(2)
label(3)
        uskel(0,4,0)
        init(1,3)
        ifdone(5)
        uvar1(0,global,1)
        uvar1(1,global,2)
label(5)
        uvar(1,local,0)
        uvar(2,local,1)
        init(3,5)
        localinit(2,4)
        neck(4,5)
        call(14)
        type(0,global,2)
        type(1,global,1)
        type(2,local,2)
        type(3,local,3)
        call(11)
        type(0,local,3)
        type(1,global,3)
        type(2,local,1)
        setupmol(6,4)
        call(11)
        type(0,local,2)
        type(1,local,0)
        type(2,global,4)
        foot(3)
label(7)
        uatom(0,12)
        uvar(1,local,0)
        uref(2,local,0)
        init(0,0)
        localinit(1,1)
        neck(1,0)
        foot(3)
label(8)
        uskel(0,9,0)
        init(1,3)
        ifdone(10)
        uvar1(0,global,1)
        uvar1(1,global,2)
label(10)
        uvar(1,local,0)
        uskel(2,11,3)
        init(4,5)
        ifdone(12)
        uref1(0,global,1)
        uvar1(1,global,4)
label(12)
        uvar(3,local,1)
        init(5,5)
        localinit(2,2)
        neck(2,5)
```

```
        call(4)
        type(0,global,1)
        type(1,local,0)
        cut(2)
        call(14)
        type(0,global,2)
        type(1,local,0)
        type(2,global,4)
        type(3,local,1)
        foot(4)
label(13)
        uskel(0,14,0)
        init(1,3)
        ifdone(15)
        uvar1(0,global,1)
        uvar1(1,global,2)
label(15)
        uvar(1,local,0)
        uvar(2,local,1)
        uskel(3,16,3)
        init(4,5)
        ifdone(17)
        uref1(0,global,1)
        uvar1(1,global,4)
label(17)
        init(5,5)
        localinit(2,2)
        neck(2,5)
        call(14)
        type(0,global,2)
        type(1,local,0)
        type(2,local,1)
        type(3,global,4)
        foot(4)
label(18)
        uatom(0,12)
        uvar(1,void,0)
        uatom(2,12)
        uatom(3,12)
        init(0,0)
        localinit(0,0)
        neck(0,0)
        foot(4)
labelc(0)
        enter
        trylast(0)
labelc(1)
        enter
        trylast(1)
labelc(11)
        enter
        try(3)
        trylast(7)
labelc(14)
```

```
        enter
        try(8)
        try(13)
        trylast(18)

label(4)
        fn(2,0,13)
        var(0,1)
        var(1,2)
label(6)
        fn(2,4,13)
        var(0,1)
        var(1,3)
label(9)
        fn(2,0,13)
        var(0,1)
        var(1,2)
label(11)
        fn(2,3,13)
        var(0,1)
        var(1,4)
label(14)
        fn(2,0,13)
        var(0,1)
        var(1,2)
label(16)
        fn(2,3,13)
        var(0,1)
        var(1,4)

Symbol table
0        external
1        sort
2        get
3        put
4        less
5        var
6        nonvar
7        add
8        user1
9        user2
10        user3
11        qsort
12        nil
13        cons
14        partition
```

Z80 Output Code

```
        org 02A00H
        call start
```

```
        jp lbc0
lb0:
        ld e,2*0
        ld hl,(Lreg)
        ld c,savebytes+2*0
        call uvarsub
        ld e,2*1
        ld hl,(Lreg)
        ld c,savebytes+2*1
        call uvarsub
        ld b,savebytes+2*2
        call neck2
        call lbc10
        defb savebytes+2*0
        defb VCLreg
        defb savebytes+2*1
        defb VCLreg
        defb 11
        defb Vatom
        ld b,2*2
        jp foot
lb2:
        ld e,2*0
        ld a,2*0
        ld bc,lb3
        call uskelsub
        ld bc,(3-1)*080H*04H+2*1
        call initsub
        ld hl,(Yreg)
        ld a,h
        or l
        jr z,lb4
        ld e,4+2*0
        ld hl,(Greg)
        ld c,2*1
        call uvar1sub
        ld e,4+2*1
        ld hl,(Greg)
        ld c,2*2
        call uvar1sub
lb4:
        ld e,2*1
        ld hl,(Lreg)
        ld c,savebytes+2*0
        call uvarsub
        ld e,2*2
        ld hl,(Lreg)
        ld c,savebytes+2*1
        call uvarsub
        ld bc,(5-3)*080H*04H+2*3
        call initsub
        ld bc,(4-2)*080H*04H+savebytes+2*2
        call localinitsub
        ld b,savebytes+2*4
```

```
        ld c,2*5
        call neck
        call lbc13
        defb 2*2
        defb VCGreg
        defb 2*1
        defb VCGreg
        defb savebytes+2*2
        defb VCLreg
        defb savebytes+2*3
        defb VCLreg
        call lbc10
        defb savebytes+2*3
        defb VCLreg
        defb 2*3
        defb VCGreg
        defb savebytes+2*1
        defb VCLreg
        ld e,2*4
        ld bc,lb5
        call setupsub
        call lbc10
        defb savebytes+2*2
        defb VCLreg
        defb savebytes+2*0
        defb VCLreg
        defb 2*4
        defb VCGreg
        ld b,2*3
        jp foot
lb6:
        ld e,2*0
        ld bc,Vatom*0FFH+Vatom+11
        call uatomsub
        ld e,2*1
        ld hl,(Lreg)
        ld c,savebytes+2*0
        call uvarsub
        ld hl,(Lreg)
        ld e,savebytes+2*0
        ld b,2*2
        call urefsub
        ld b,savebytes+2*1
        call neck2
        ld b,2*3
        jp foot
lb7:
        ld e,2*0
        ld a,2*0
        ld bc,lb8
        call uskelsub
        ld bc,(3-1)*080H*04H+2*1
        call initsub
        ld hl,(Yreg)
```

```
        ld a,h
        or l
        jr z,lb9
        ld e,4+2*0
        ld hl,(Greg)
        ld c,2*1
        call uvar1sub
        ld e,4+2*1
        ld hl,(Greg)
        ld c,2*2
        call uvar1sub
lb9:
        ld e,2*1
        ld hl,(Lreg)
        ld c,savebytes+2*0
        call uvarsub
        ld e,2*3
        ld a,2*2
        ld bc,lb10
        call uskelsub
        ld bc,(5-4)*080H*04H+2*4
        call initsub
        ld hl,(Yreg)
        ld a,h
        or l
        jr z,lb11
        ld hl,(Greg)
        ld e,2*1
        ld b,4+2*0
        call uref1sub
        ld e,4+2*1
        ld hl,(Greg)
        ld c,2*4
        call uvar1sub
lb11:
        ld e,2*3
        ld hl,(Lreg)
        ld c,savebytes+2*1
        call uvarsub
        ld b,savebytes+2*2
        ld c,2*5
        call neck
        call lbc3
        defb 2*1
        defb VCGreg
        defb savebytes+2*0
        defb VCLreg
        ld e,savebytes+2*2
        call cut
        call lbc13
        defb 2*2
        defb VCGreg
        defb savebytes+2*0
        defb VCLreg
```

```
        defb 2*4
        defb VCGreg
        defb savebytes+2*1
        defb VCLreg
        ld b,2*4
        jp foot
lb12:
        ld e,2*0
        ld a,2*0
        ld bc,lb13
        call uskelsub
        ld bc,(3-1)*080H*04H+2*1
        call initsub
        ld hl,(Yreg)
        ld a,h
        or l
        jr z,lb14
        ld e,4+2*0
        ld hl,(Greg)
        ld c,2*1
        call uvar1sub
        ld e,4+2*1
        ld hl,(Greg)
        ld c,2*2
        call uvar1sub
lb14:
        ld e,2*1
        ld hl,(Lreg)
        ld c,savebytes+2*0
        call uvarsub
        ld e,2*2
        ld hl,(Lreg)
        ld c,savebytes+2*1
        call uvarsub
        ld e,2*3
        ld a,2*3
        ld bc,lb15
        call uskelsub
        ld bc,(5-4)*080H*04H+2*4
        call initsub
        ld hl,(Yreg)
        ld a,h
        or l
        jr z,lb16
        ld hl,(Greg)
        ld e,2*1
        ld b,4+2*0
        call uref1sub
        ld e,4+2*1
        ld hl,(Greg)
        ld c,2*4
        call uvar1sub
lb16:
        ld b,savebytes+2*2
```

```
        ld c,2*5
        call neck
        call lbc13
        defb 2*2
        defb VCGreg
        defb savebytes+2*0
        defb VCLreg
        defb savebytes+2*1
        defb VCLreg
        defb 2*4
        defb VCGreg
        ld b,2*4
        jp foot
lb17:
        ld e,2*0
        ld bc,Vatom*0FFH+Vatom+11
        call uatomsub
        ld e,2*2
        ld bc,Vatom*0FFH+Vatom+11
        call uatomsub
        ld e,2*3
        ld bc,Vatom*0FFH+Vatom+11
        call uatomsub
        ld b,savebytes+2*0
        call neck2
        ld b,2*4
        jp foot
lbc0:
        pop hl
        ld (CPreg),hl
        call enter
        call trylastsub
        jp lb0
lbc10:
        pop hl
        ld (CPreg),hl
        call enter
        call lb2
        call trylastsub
        jp lb6
lbc13:
        pop hl
        ld (CPreg),hl
        call enter
        call lb7
        call lb12
        call trylastsub
        jp lb17
symtbl:
        defw lb18
        defw lb19
        defw lb20
        defw lb21
        defw lb22
```

```
        defw lb23
        defw lb24
        defw lb25
        defw lb26
        defw lb27
        defw lb28
        defw lb29
        defw lb30
        defw lb31
lb18:   defm 'sort'
        defb 00H
lb19:   defm 'get'
        defb 00H
lb20:   defm 'put'
        defb 00H
lb21:   defm 'less'
        defb 00H
lb22:   defm 'var'
        defb 00H
lb23:   defm 'nonvar'
        defb 00H
lb24:   defm 'add'
        defb 00H
lb25:   defm 'user1'
        defb 00H
lb26:   defm 'user2'
        defb 00H
lb27:   defm 'user3'
        defb 00H
lb28:   defm 'qsort'
        defb 00H
lb29:   defm 'nil'
        defb 00H
lb30:   defm 'cons'
        defb 00H
lb31:   defm 'partition'
        defb 00H
        org 08000H
lb3:
        defb 12
        defb Vskel
        defb 2*0
        defb 2
        defb 2*1
        defb VYreg
        defb 2*2
        defb VYreg
lb5:
        defb 12
        defb Vskel
        defb 2*4
        defb 2
        defb 2*1
        defb VYreg
```

```
        defb 2*3
        defb VYreg
lb8:
        defb 12
        defb Vskel
        defb 2*0
        defb 2
        defb 2*1
        defb VYreg
        defb 2*2
        defb VYreg
lb10:
        defb 12
        defb Vskel
        defb 2*3
        defb 2
        defb 2*1
        defb VYreg
        defb 2*4
        defb VYreg
lb13:
        defb 12
        defb Vskel
        defb 2*0
        defb 2
        defb 2*1
        defb VYreg
        defb 2*2
        defb VYreg
lb15:
        defb 12
        defb Vskel
        defb 2*3
        defb 2
        defb 2*1
        defb VYreg
        defb 2*4
        defb VYreg
```

Appendix D - LIPS test program

Date: Wed, 5 Mar 86 14:23:09 -0100
From: Micha Meier <unido!ecrcvax!Micha@seismo.CSS.GOV>
Subject: LIPS again

The speed of the current Prolog systems is still measured using the naive reverse example with a list of 30 elements. I guess that anybody who has tried this with a system that runs over 10 kLIPS has seen the inconvenience - the time spent in executing this example is too short to be measured correctly. The other drawback is that many implementors concentrate on optimising this very example and the like, i.e. deterministic procedures processing lists and the results for other types of programs may be totally different (e.g. there is a Prolog system running 'quicksort' 20 times slower than 'nreverse').

Below, I include the listing of a test program which tries to solve these problems: first, it includes a procedure which measures LIPS on naive reverse of an arbitrary list. Second, a procedure that measures LIPS on quicksort of a list in descending order; third, measuring of LIPS by quicksort of an ordered list. I suppose that indexing prevents choices in concatenate and in partition([], _, _, _).

The first case is purely deterministic - no choice points and no failures.

In the second case, the number of inferences is o(n*n/2) and the same for choice points created. In the last example, the number of inferences is o(n*n), for choices and failures it is o(n*n/2).

The quicksort example better reflects the speed of a Prolog system: it creates some choice points, uses the cut and calls an evaluable predicate. When the implementors try to optimise the first clause for 'partition/4' to yield something like

```
partition([X|L], Y, [X|L1], L2) :-
        X < Y, !, partition(L, Y, L1, L2).

        get list A1
        unify variable X5
        unify variable X1
        get list A3
        unify value X5
        unify variable A3
        put value X5, A1
        escape </2
        neckcut
        execute partition/4
```

then the whole system is likely to run fast even on non-deterministic examples with some arithmetic.

-- Micha

```
% File    : LIPS.PL
% Author  : Micha Meier
% Purpose : Testing the speed of naive reverse and quicksort
%           of an arbitrary long list.
%           On systems without reals it is necessary to
%           multiply I (inferences no.) by the time unit,
%           e.g. 1000 if cputime is in milliseconds.


test :- write('list length : '),
        read(X),
        conslist(X, List),
        T1 is cputime,
        nreverse(List, _),
        T2 is cputime,
        T is T2 - T1,
        I is (X*(X+3))/2 + 1,
        LIPS is I/T,
        write('  LIPS of naive reverse:             '),
        write(LIPS),
        nl,
        T3 is cputime,
        qsort(List, _, []),
        T4 is cputime,
        TT is T4 - T3,
        II is (X*(X+5))/2 + 1,
        LIPS1 is II/TT,
        write('  LIPS of quicksort (reverse order): '),
        write(LIPS1),
        nl,
        T5 is cputime,
        qsort1(List, _, []),
```

```
        T6 is cputime,
        TTT is T6 - T5,
        III is (X+1)*(X+1),
        LIPS2 is III/TTT,
        write('  LIPS of quicksort (ordered):       '),
        write(LIPS2),
        nl.

nreverse([], []).
nreverse([X|L0],L) :- nreverse(L0, L1),
    concatenate(L1, [X], L).

concatenate([], L, L).
concatenate([X|L1], L2, [X|L3]) :- concatenate(L1, L2, L3).

conslist(0, []) :- !.
conslist(N, [N|L]) :-
        N1 is N-1,
        conslist(N1, L).

qsort([X|L], R, R0) :-
        partition(L, X, L1, L2),
        qsort(L2, R1, R0),
        qsort(L1, R, [X|R1]).
qsort([], R, R).

partition([X|L], Y, [X|L1], L2) :-
        X < Y,
        !,
        partition(L, Y, L1, L2).
partition([X|L], Y, L1, [X|L2]) :-
        partition(L, Y, L1, L2).
partition([], _, [], []).

qsort1([X|L], R, R0) :-
        partition1(L, X, L1, L2),
        qsort1(L2, R1, R0),
        qsort1(L1, R, [X|R1]).
qsort1([], R, R).

partition1([X|L], Y, [X|L1], L2) :-
        X > Y,
        !,
        partition(L, Y, L1, L2).
partition1([X|L], Y, L1, [X|L2]) :-
        partition1(L, Y, L1, L2).
partition1([], _, [], []).
```