LOGIC PROGRAMMING AND LPL

T. Hürlimann

Working Paper No. 216

September 1993

INSTITUT D'INFORMATIQUE, UNIVERSITE DE FRIBOURG

INSTITUT FÜR INFORMATIK DER UNIVERSITÄT FREIBURG

Institute of Informatics, University of Fribourg, site Regina Mundi

rue de Faucigny 2

CH-1700 Fribourg / Switzerland

Bitnet: HURLIMANN@CFRUNI51

phone: (41) 37 21 95 60 fax: 37 21 96 70

Logic Programming and LPL

Tony Hürlimann, Dr. rer. pol.

Key-words: logic programming, language design

Abstract: This paper gives an survey of logic programming. It is supposed that the reader is familiar with the vocabulary of first order logic and especially with resolution from a logical point of view. A concise summary can be found in *Logic: a Survey* [Hürlimann 93b]

Stichworte: Logische Programmierung, Sprachenentwurf

Zusammenfassung:

"...Science as well as technology,
will in the near and in the farther future
increasingly turn from problems of intensity,
substance, and energy, to problems of structure,
organization, information, and control....."
J.v.Neumann

INTRODUCTION

This paper gives a somewhat biased survey of logic programming. The bias is to make the connection between predicate logic and logic programming very clear. It is, therefore, necessary to read first my paper Logic, a Survey [Hürlimann 93b]. Several examples will explain the connections as well as the link to Prolog, the classical language in logic programming.

In an imperative programming language such as Pascal one needs to write down *how* a task is executed. The path to the solution is then written as a sequence of procedures and functions. In logic programming, one must only write down *what* the problem is. To find a solution is the problem of its inference machine. Therefore, we may say that a logic programming language is a *declarative* language and the others are *imperative* languages. We may also say that "a language is called declarative when the programming activity does not require particular knowledge about the order of evaluation." [Lock p 22]. To illustrate the point take the problem of calculating the faculty of an integer. The faculty is defined as

faculty of zero is one

faculty of n > 1 is faculty of n–1 times n.

Formally, the definition can be stated as

fac(0) = 1

fac(n) = fac(n–1)*n (for all n > 0)

In logic programming we would just enter this definition without any indication how to manipulate it. So in logic programming we may write

```
fac(X,0) :- X=1
fac(X,N) :- N*fac(X,N-1)
```

In an imperative language such as Lisp, one may write the function as

```
(define fac (n)
   (if (= n 0) 1
       (* n fac(n-1)))
```

or in Pascal the function may be written as

```
function fac(n:integer):integer;
begin
  if n=0 then fac:=1
  else fac:=n*fac(n-1);
end;
```

both functions are essentially the same. Both are called with an argument n. If the argument is bigger than zero the function is called recursively with an argument n–1. In spite of its resemblance with the Lisp and Pascal function, the logic program, however, has a very different semantic and execution path. It does not indicate how to solve the problem. It just states the definition of faculty, much like we define a term in mathematics. To see the point, one may put the query in logic programming:

```
:- fac(5040,N)
```

which means “what is the number such that the faculty is 5040?” This is very different from what the imperative function could do. While the imperative functions have a defined *direction* in its calculation, the logic program does not. It don't even matter what the input is and what the output is.

Another way to say the difference is to note that an imperative program implements mappings, that is having implemented a mapping m(x), we can make the following request:

given a, return the value of m(a).

Logic programming, however, implements relations, that is, having implemented a relation R(x,y), we can make the following requests:

given a and b, determine whether R(a,b) is true
given a, find all x such that R(a,x) is true
given b, find all x such that R(x,b) is true
find all x and y such that R(x,y) is true.

Where does this power of logical programming come from? A logic program is nothing else than a formula in predicate logic. (Almost) all that can be written in first order logic can also be expressed in a logic program. The main objective of this paper is to make this connection clear.

DEFINITIONS

A *logic program* is defined as a set of clauses (=Skolem normal form). Normally in practical implementation, there are limitations on the form of the clauses allowed. Prolog is essentially based on Horn-clauses, more precisely, all clauses must be *definite* clauses (=clauses containing exactly one positive literal) except one clause which has no positive literal at all. The definite

clauses are called *rules*, if they contain at least two literals. They are called *facts*, if they contain only one literal (a *unit clause*). The single clause without any positive literal is called *goal* (or *query*). Rules, facts, and goals are compared in the following table (A, B, C, and D are literals, T means true, and F means false):

	logic prog. syntax	first order syntax	clausal form
rules	A :- B C D ...	A ♦ B □ C □ D □ ...	A Δ ¬B Δ ¬C Δ ¬D Δ ¬...
facts	A :-	A ♦ T	A
goals	:- B C D ...	F ♦ B □ C □ D □ ...	¬B Δ ¬C Δ ¬D Δ ¬...

The AND-connector is replaced a space (or sometimes by a comma), and the implication (♦) is replaced by the symbol :- (in a goal the symbol ?: is sometimes used). Since a logic program is a Skolem normal form, all variables are bounded by all-quantifiers (which are omitted within a logic program). Hence, we can write the clauses in first order logic as

rules : $\forall xy\ldots(A \vee \neg B \vee \neg C \vee \neg D \vee \neg \mathrm{K})$

facts : $\forall xy\ldots(A)$

goals: $\forall xy\ldots(\neg B \vee \neg C \vee \neg D \vee \neg \mathrm{K})$ which is the same as

$\neg(\exists xy\ldots(B \wedge C \wedge D \wedge \mathrm{K})$

Note that the query (goal) is the negation of an existence theorem converted to logic programming form.

There are also syntax prescriptions concerning the atoms (literals): function names and predicate names normally must begin with a lower-case letter, whereas variables begin with a upper-case letter.

A rule has two parts, the *head* (left from the :- symbol) and the *tail* (right to the :- symbol). A fact consists only of a head, whereas a goal consists only of a tail. The head contains only one (positive) literal (atom), and the tail is a conjunction of (positive) literals.

The whole syntax of a logic program is as following (where f is a functor-name, p is a predicate-name, "|" means select, and "{ ... }" means repeat any times):

```
program ::= {clause}
clause ::= fact | rule | goal
fact ::= atom :-
rule ::= atom :- {atom}
goal ::= :- {atom}
atom ::= p | p ( {term} )
term ::= f | f ( {term} ) | Variable
```

Since a logic program is the same as a set of clauses, the objective is to prove the validity of a formula: given a knowledge base (a set of clauses) we have to prove an arbitrary formula. This is done using resolution. We use a simple example to explain how resolution can be viewed in a logic program. Suppose the knowledge base consists of the two facts (=atoms) *r* and *s* (note that *r* and *s* are predicates) and the single rule *t :- r s*. The program is

```
r :-
s :-
t :- r s
```

Suppose the clause to prove is *t*. The goal to add to the knowledge base is

```
:- t
```

Our whole logic program is now

```
r :-
s :-
t :- r s
:- t
```

We can express this program in first order logic (actually in propositional logic since no variables appear) as following

```
r ♦ T
s ♦ T
t ♦ r □ s
F ♦ t
```

Or expressed in clausal form we have:

```
r
s
t Δ ¬r Δ ¬s
¬t
```

To use resolution, the negation of the formula to prove has to be added to the knowledge base to prove the inconsistency of the system. By defining a goal in the logic program, the negation *is* automatically added to the knowledge base as we see by the example. Resolution now consists to find the same literal – once positively and once negatively – in two different clauses and to produce the resolvent (see Hürlimann 1993b). From the point of view of a logic program, this means to unify a head of a clause with an atom of a tail, since the head-atom is a positive literal and the tail-atoms are negative literals in the clausal form. In our example we have the three possible matches:

1 The head of the first clause can be unified with the first tail-atom of the third rule (r :- and t :- r s)

2 The head of the second clause can be unified with the second tail-

atom of the third rule (s :- and t :- r s)

3 The head of the third clause can be unified with the goal

(:- t and t :- r s)

In logic programming, resolution is goal directed, which means that to trigger the resolution procedure the goal is resolved with any clause. A resolvent can then be interpreted as a new goal from which the search goes on. In our example, the goal will be unified with the head of the third clause which leaves us with the new goal (that is the tail of the third clause):

```
:- r s
```

To prove *t*, we have now reduced the goal to prove *r* and *s*. Now the proof has to branch: first we have to prove *r* and later on we have to prove *s*. Let's begin with *r*. The goal (:-r) can be unified with the head of the first clause (r:-). Since we obtain the empty clause we are finished, and we are left with the second half to prove *s*. The (new goal (:-s) can be unified with the second clause (s:-) which again produces the empty clause. Since we have proved both r and s which was needed to prove t we are finished. The computation of the proof can be visualized by the computation tree (or *deduction tree*) see Figure 1: To prove t, prove r **and** s.

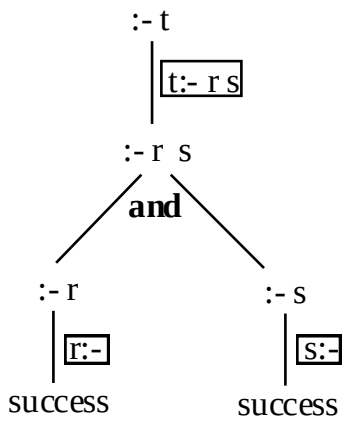


Figure 1

The resolution just described is called SLD-resolution. It is a special case of linear resolution which

1) is goal-directed: The goal is matched with the head of an input-clause.
2) clause-ordered: The goal is unified with the first possible clause from top to bottom.
3) goal-left-to-right-ordered: the atoms of the goal are unified from left to right.

To illustrate SLD-resolution with unification we use a second simple example: “John needs vitamin C. Carrots contain vitamin A and oranges contain vitamin C. Any person should eat what he or she needs”. The knowledge could be modeled as a logic program as following:

```
needs(john,vitamin_C) :-
found_in(vitamin_A,carrot) :-
found_in(vitamin_C,oranges) :-
should_eat(X,Y) :- found_in(Z,Y) needs(X,Z)
```

The goal now is to infer whether John should eat oranges. The query, therefore, is:

```
:- should_eat(john,oranges)
```

In clausal form, we have the following clauses where the goal is negated:

```
needs(john,vitamin_C)
found_in(vitamin_A,carrot)
found_in(vitamin_C,oranges)
should_eat(X,Y) Δ ¬found_in(Z,Y) Δ ¬needs(X,Z)
¬should_eat(john,oranges)
```

The goal is unified with the head of the forth clause which produces the new goal using the unificator {john/X,oranges/Y} (see Hürlimann 1993b):

```
:- found_in(Z,oranges) needs(john,Z)
```

In clausal form this is written as:

```
¬found_in(Z,oranges) Δ ¬needs(john,Z)
```

The new goal is evaluated from left to right, therefore, we take first *found_in(Z,oranges)*. This subgoal is tried to unify with the second clause *found_in(vitamin_A,carrot)*. Since the unification fails, we try to unify *found_in(Z,oranges)* with the third clause *found_in(vitamin_C,oranges)*. Unification is successful and *Z* is bounded to *vitamin_C*. Hence, we are left with the second subgoal *needs(john,vitamin_C)*. This goal can be unified with the first clause. The empty clause results and the inconsistency is proved. The deduction tree is shown in Figure 2.

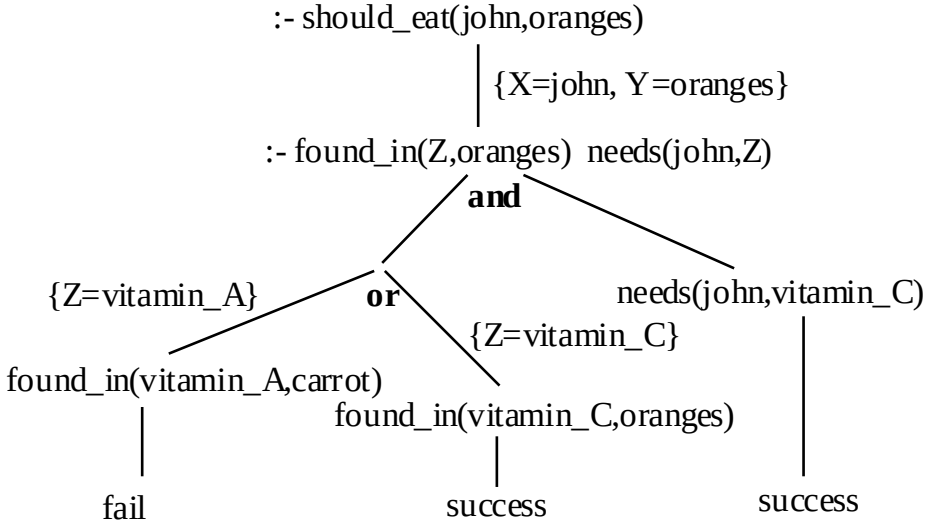


Figure 2

The execution mechanism is a top down procedure which reduces the goal to further subgoals until the empty clause (success) is found. If several causes contain the same head we can take either of them. This corresponds to an OR-node in the deduction tree. If the goal contains several subgoals, we need to prove *all* of them. This corresponds to an AND-node in the tree. To prove the formula the deduction tree must contain at least one success node beneath every branch of each AND-node. If we reach a fail node, we needs to *backtrack* to the next OR-node and to try another branch which has not yet tried. If none of them is left, we needs to backtrack even further up the tree to the next OR-node. When backtracking bounded variables beneath the backtracking point (OR-node) must be freed.

PROLOG II

A logic program does not determinate how the deduction tree is traversed, a logic program is intrinsically *non-deterministic*. We might imagine a parallel machine that executes the different branches in parallel: an OR-node succeeds if at least one branch succeeds, and an AND-node succeeds if all branches succeed. On a sequential machine, however, the tree is normally executed in prefix-order. This corresponds to the SLD-resolution procedure. And Prolog is based on SLD-resolution.

The prefix-order evaluation (of Prolog) has several serious consequences on the execution time: the order at which the rules and the sub-goals are written within a program matters. This introduces a procedural element into the program which is not necessarily a disadvantage. On the contrary, the break-though in logic programming was the fact that rules can be interpreted as procedures, and terms as data structures [Kowalski 1979] who coined the statement: *Algorithm = Logic + Control.* The logic is stuffed in the rules and the control is the knowledge of the traversal information of the tree.
But this restriction toa specified traversal destroyed the soundness of a program eventually. Sometimes, the evaluation will even loop for ever. To illustrate this point, we may take another example. Suppose we have the following program

```
p(f(X)) :- p(X)
p(a) :-
```

and we wanted to prove

```
:- p((a)
```

Resolving the goal with the first rule will bind the variable X to f(a) and the new goal to prove is p(f(a)). Resolving again with the first rule will bind the variable to f(f(a)) and the new goal to prove is p(f(f(a))), etc. If, however, we would have reversed the two rules as in

```
p(a) :-
p(f(X)) :- p(X)
```

then a single match would have produced the empty clause. Therefore, the order of the clauses matters for the SLD-resolution.
But also the order of the literals within the goal is important. In comparison with the order of the clauses, however, his second indeterminismn is harmless, because it does not destroy the soundness. It can have an influence on the running time as the following example shows.

```
; this is inefficient
query1(X,Y) :-
  father_of(Y,Z)
  wife_of(Z,W),
  brother_of(W,X)

; this is more efficient
query2(X,Y) :-
  brother_of(W,X),
  wife_of(Z,W),
  father_of(Y,Z)
```

Every person has a father but much less persons has a brother. So it might be more efficient to filter all brother which reduces the search greatly.

There are some other – from the point of view of a logic program – “impure” constructs which are necessary to have a versatile programming language:
1) The occur check: Depending on how the occur test is done within the unification algorithm, the evaluation can even produce a false answer! The reason is the following: We know that the two clauses $P(x,x)$, $\neg P(x, f(x))$ cannot be unified. Some Prolog, however, unify this to give $P(f(x), f(x))$, which is false!
2) Since Prolog is restricted to Horn-clauses,no negated literal is allowed. Several Prolog implementation, however, have a not(X) predicate, which succeeds iff an atempt to solve the given atom X fails. But this predicate must be used with care or in a cerain context only. An example is the following rule.

```
IsRichAndUnhappy(X) :- not(happy(X))  rich(X)
```

will not be accepted by most Prolog interpreter, whereas with

```
IsRichAndUnhappy(X) :- rich(X)  not(happy(X))
```

most Prolog do not have any problem. The reason is that in the second version *X* was bound – say to *a* – when unifying `rich(X)` with `rich(a)`, so `not(happy(a))` is a constant now. The expression `not(happy(a))` fails if `happy(a)` is a fact in the knowledge base otherwise it succeeds. We call this mechanism *negation by failure*, which adopts the heuristics that evey fact that is not known is also false. Clearly the not predicate is not the same as the negation. To see this consider the following example:

```
happy(X) :- not(sad(X))
sad(fred) :-
```

the query (of the unknown fact that jane is happy) `:-happy(jane)` will succeed, but the queries `:-happy(fred)` and `:-happy(X)` both will succeed.

3) Two build-in predicates are *cut* and *fail*. The cut predicate is a powerful method to prune the search tree. When cut appears in the tail of a goal then it immediately succeeds, but if the computation must backtrack to this call of cut it fails and thus any part of the tree that lay beyond that point may not be searched. The fail predicate fails all the time. To illustrate these predicate consider the following rule that define the not predicate.

```
not(X) :- call(X) cut fail
not(X) :-
```

We suppose that call(X) is a build-in predicates that takes an atom as argument and simply calls a procedure whose head unifies with this atom. If we evaluate the goal `:-not(a)` then a will be unified with X in `call(X)`. If this fails, the resolution tries to unify with the second rule (fact) and succeeds. If the unification of `call(X)` succeeds then `cut` will succeeds too but `fail` will fail. Since the procedure cannot backtrack beyond the `cut`, the second rule cannot be executed and so our original goal fails. Hence, we have the following, if `a` succeeds then `not(a)` fails and if `a` fails then `not(a)` succeeds.

4) Any arithmetic, computable functions can be describted by a definite program. That means that from a theoretical point of view the paradigm of logic programming is not less expressive than any other programming language. But this can be very inefficient. Therefore, several arithmetic functions are “hardwired” within the Prolog-Interpreter. The problem is, whether this does not destroy the declarative manner of logic programming. Fortunately, the operators can be expressed as “predicates”. Hence, instead of writing “4+5”, we may adopt the syntax “plus(4,5,X)” which returns the result in *X*. Most prolog allow also the infix notation using the *is* construct (such as “X is 4+5”. Such “predicates” must not be confounded with the l,ogic predicate plus() that could be used for substraction too.

EXAMPLES

EXAMPLE 1: THE PATH IN A GRAPH [SCHMITT 1992]

A path within a graph can be defined as following:

1) for every node X, there is a path from X to X
2) for every node X, Y, and Z, if there exists an edge between X and Z and a path between Z and Y, then there exists also a path between X and Y.

This can be formulated using predicate logic as (where the predicate $P(x,y)$ means "there exists a path from x to y" and $E(x,y)$ "there exists an edge between x and y):

$$\forall x\ P(x,x)$$

$$\forall xy\ (\exists z\ (E(x,z) \wedge P(z,y)) \rightarrow P(x,y))$$

Transforming the formula into a set of clauses gives:

$$\forall x\ P(x,x)$$

$$\forall xyz\ (\neg E(x,z) \vee \neg P(z,y) \vee P(x,y))$$

Since we have only Horn clauses, the logic program is

```
p(X,X) :-
p(X,Y) :- e(X,Z)  p(Z,Y)
```

To get a complete example, we need a concrete (directed) graph. Suppose, Figure 3 is our example graph

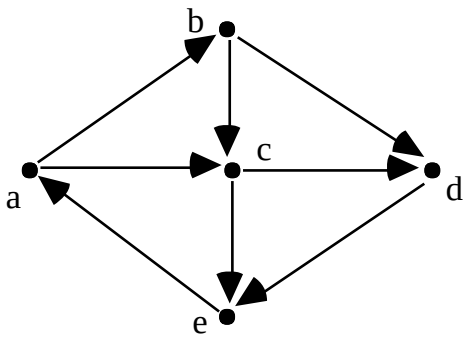


Figure 3

The graph can be added to the knowledge base by adding the following facts to the logic program:

```
e(a,b) :-
```

```
e(a,c) :-
e(b,c) :-
e(b,d) :-
e(c,d) :-
e(c,d) :-
e(d,e) :-
e(e,a) :-
```

Now we may formulate our queries:

1) Is there a path from a to e?
2) Is there a node X that can be reached from a?
3) Is there a node X from which there is a edge going to d and e?

The three queries could be formulated as following:

```
1)  :- p(a,e)
2)  :- p(a,X)
3)  :- e(X,d) e(X,e)
```

The deduction tree of the first query is shown in Figure 4. To prove p(a,e) we try first the rule p(X,X). Since X cannot be bounded this rule fails. Therefore, we try the rule p(X,Y) :- e(a,Z) p(Z,e). Now we need to prove e(a,Z) which succeeds with Z=B or Z=c. If we choose Z=c we need to prove p(c,e). To prove p(c,e) we try again first p(X,X). Since this fails, we try e(c,U) p(U,e). Both goals can only succeed if U is bounded to e. The deduction tree is constructed dynamically. Figure 4 shows only the state of the last binding.

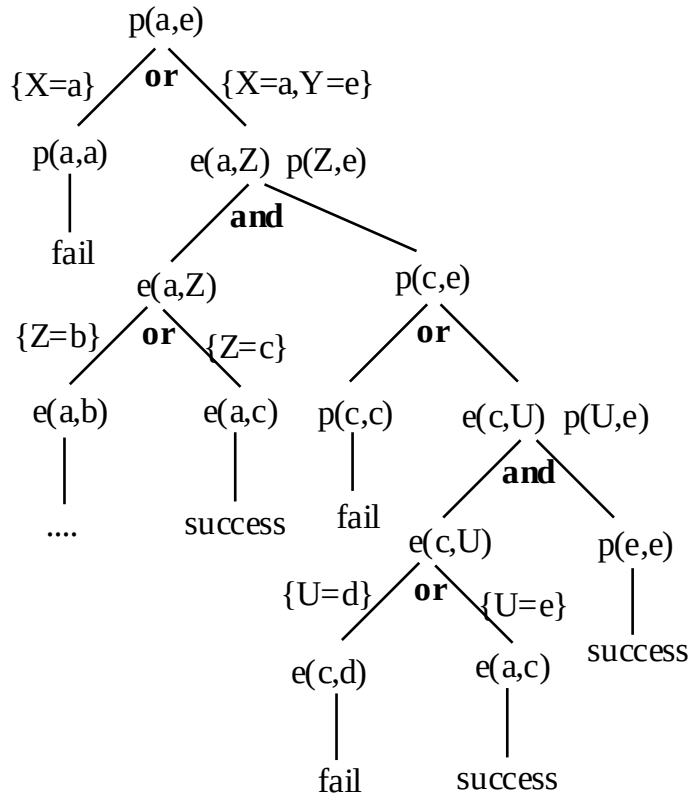


Figure 4

Of course, the logic program gives us only a Yes-or-No answer depending whether the goal was proved or not. The path itself cannot be returned by this program. If, however, we collect the bindings, we would be able to return the path itself (see below).

EXAMPLE 2: ADDITION [DUFFY D., 1991]

The addition can be defined as following

$$x + 0 = x$$
$$\text{if } x + y = z \text{ then } x + y + 1 = z + 1 \qquad (1)$$

In first order logic this may be stated as

$$\forall x\ P(x, 0, x)$$
$$\forall xyz\ (P(x, y, z) \rightarrow P(x, s(y), s(z)) \qquad (2)$$

where the predicate $P(x,y,z)$ means "$x+y=z$", and the function s is the successor function $s(x)=x+1$.

Stated as a logic program, the addition can be defined as

```
plus(X, 0, X).
plus(X, s(Y), s(Z) :- plus(X, Y, Z)
```

(where we use "plus" for the predicate P and "s" for the successor function).

We are now able to query the knowledge base. For example: "Is there any number such that the addition of 3+4 is the same as the addition 2+5". Stated in first order logic, this is:

$$\exists x\ (P(3, 4, x) \wedge P(2, 5, x))$$

In first order logic when using resolution, one has to add the negation of this clause to the knowledge base, which is

$$\neg\exists x\ (P(3, 4, x) \wedge P(2, 5, x)) \text{ or in Skolem normal form}$$
$$\forall x\ (\neg P(3, 4, x) \vee \neg P(2, 5, x))$$

The clause has no positive literal, therefore it can be represented as a goal in logic programming as

```
?: plus(3,4,X), plus(2,5,X)
```

We should keep in mind, what the objective of resolution is in logic: The objective is the prove a formula, that is to prove the inconsistency of the knowledge base together with the negation of the query. Therefore, the proof will only return 'yes' or 'no' saying that the formula was proved or disproved. The query of our example

```
?: plus(3,4,X), plus(2,5,X)
```

would return only "yes: there is a number such that it is the addition of 3+4 and

2+5. What the number actually *is*, cannot be determinate by the proof itself.

EXAMPLES 3: CHEMICAL SYNTHESIS PROBLEM (CHANG/LEE P.21)

The logical formulation of this problem can be found in Hürlimann 1993b. Here is the corresponding logic program:

```
Mg :- MgO, H2
H2O :- MgO, H2
CO2 :- C
CO2 :- O2
H2CO3 :- CO2
H2CO3 :- H2O
MgO :-
H2 :-
O2 :-
C :-
:- H2CO3      (goal)
```

The deduction tree is shown in Figure 5.

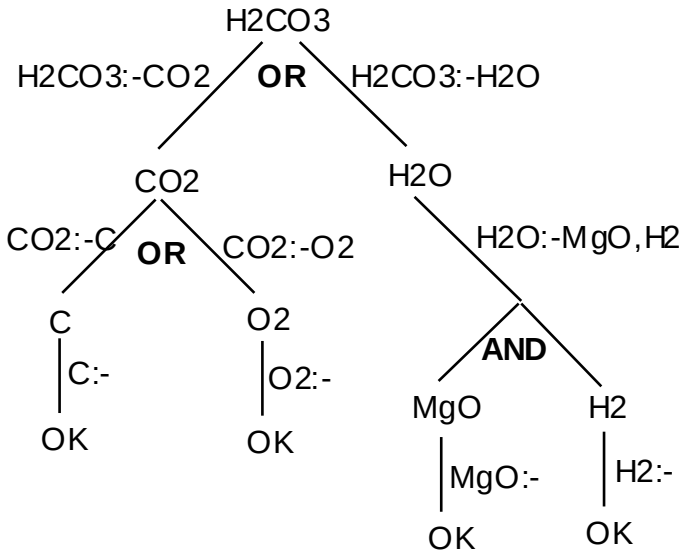


Figure 5: SLD-resolution, deduction tree

EXAMPLE 4: QUICKSORT

The Quicksort can be defined as following:

1) A list of zero elements is always sorted
2) A list X of more than zero elements is sorted if the list is partitioned into two sub-lists sorted U and V such that U contains only elements less or equal to an element Y and V contains only elements greater than Y and the resulting sorted list is a concatenation of U then the element Y and finally the sub-list V.

It is remarkable that the following logic program defines completely the sorting

routine!

```
sortQ(nil, nil) :-
sortQ(X.Y, Z) :- partition(X, Y, U, V)  sortQ(U, U1)  sortQ(V, V1)  append(U1,
Y.V1, Z)
```

(The syntax X.Y is a term. It means that X.Y is a list such that X is the first element and Y is the rest of the list).

EXAMPLE 5: KNAPSACK (SPERSCVHNEIDER AL. P.207)

The predicate packable(List,cap) is true iff a finite list (v_1, v_2, v_3,..., v_n) of numbers and a number cap and a sequence (b_1, b_2, b_3,..., b_n) with $b_i \in \{0,1\}$ fulfill the following equation

$b_1v_1+b_2v_2+...+b_nv_n = cap$

The Knapüsack problem can be formulated as

```
packable([],0)
packable(V.Rest,Cap) :- less(Cap,V) packable(Rest,Cap)
packable(V.Rest,Cap) :- less(V,Cap) packable(Rest,Cap)
packable(V.Rest,V) :- packable(Rest,V)
packable(V.Rest,Cap) :- less(V,Cap) minus(Cap,V,Capnew) packable(Rest,Capnew)
packable(V.Rest,V) :- packable(Rest,0)
```

(In the LPL constraint language we would write:

PROVE knapsack: SUM{i} a*x = b;

or as predicate

knapsack(a{i},x{i},b) : SUM{i} a*x = b; (* or *)

knapsack(a{i},x{i},b) : SUM{i} a*x =~ b; (* or *)

one cannot got it cheaper!)

RELATIONAL ALGEBRA (CALCULUS)

A *database* is a storehouse of associated information about a world. A “world” is made up of *entities* and *relations* between them. Each entity may contain several *attributes*. The *entity-relationship model* (E-R model) is a high-level representation of a database. An example is shown in the Figure 6.

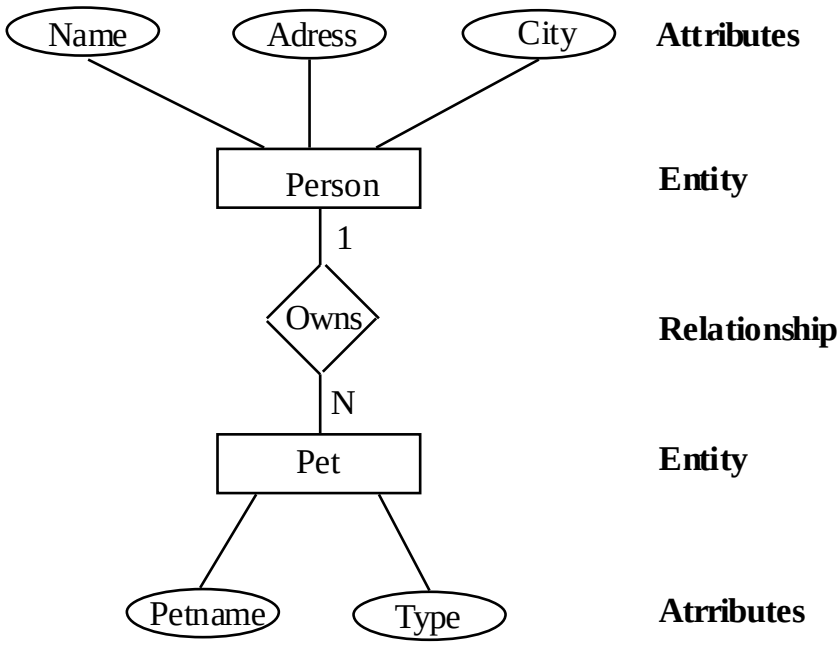


Figure 6

A relational model which is another representation scheme of a database can be developed from a E-R diagram. Both entities and relationship becomes *relations* and the attributes becomes the *fields*. A *relational database* consists of collections of relation *tables*. A relation is a subset of $A_1 \infty A_2 \infty A_2 \infty ... \infty A_n$ where A_i for i={1...n} is the domain of the i-th attribute within the relation. This is called an n-ary relation. An individual row in the table is called a *tuple*. Several operation can manipulate such tables.

1) A *selection* operation creates a new table made up of those tuples (rows) that satisfy a certain property.
2) A *projection* operation selects some columns (attributes) from the table (while eliminating duplicate tuples).
3) The *Cartesian product* operation is performed on two relations. If the arity of the two relations are n and k, then the Cartesian product is a new table with arity n+k.
4) The *set-difference* operation is defined on two relations R and S of the same arity. It is a new table that contains the set of tuples in R but not in S.
5) The *set-union* operation also defined on two relations of the same arity is a new relation of all tuples in both relations.

A calculus that contains at least this five operations is called *complete* (also called the *relational calculus*) (see Ullman for a more extensive discussion).

Several operators extend a database management language which do not enhance the expressive power of the calculus. A example is the *join* operation that can be performed on two relations with a common attribute. It forms the Cartesian product of all n-tuples from the first with all k-tuples in the second relation to form a new relation of n+k-tuples and then selects the subset of those where the common attribute has the same value, writing the result as a set of (n+k-1)-tuples (since the common attribute is written only once).

In the database paradigm one often encounters the concept of *view*. A view is a relation that is not explicitly stored in the database, but that can be produced by means of the operators of relational calculus. The relational calculus is not very powerful. It is, for example, not possible to produce the transitive closure from a table. Relational calculus can be simulated by logic programming. On the other hand, it provides an (often more efficient) alternative to SLD-resolution for a class of logic programs. An example will make the point clear.

EXAMPLE 5: A SMALL DATABASE

a database for a dating agency (Spencer-Smith R. p.108ff). A dating agency has stored the knowledge of the clients in three database tables as following:

PERSON:

name	sex	build	character	age
John	m	large	shy	mature
Mary	f	medium	extrovert	young
.....				

PREFERS:

name	build	character	age
mary	large	extrovert	young
john	small,medium	(any)	young,mature
.....			

LIKES:

name	music	sport	rec
john	jazz	swimming	cinema
mary	classical,jazz	swimming	travel,cimema
.....			

The problem now is to define two queries which extract all suitable persons that meet the requirements for all clients and to define a match relation.
The two SQL queries might be formulated as following:

SQL Queries: (Pius fragen)

suits

match ....

The same knowledge can be formulated in Prolog as following:

```
person(john,m,large,shy,mature).
person(mary,f,medium,extrovert,young).
prefers(mary,large,extrovert,young).
prefers(john,X,_,Y) :-  (X=small;X=medium),(Y=young; Y=mature).
likes(john,jazz,swimming,cinema).
likes(mary,X,Y,Z) :-  (X=classical; X=jazz), Y=swimming, (Z=travel; Z=cinema).
opp(m,f). opp(f,m).

suits(B,A) :-
   person(A,X,_,_,_),
   prefers(A,Build,Char,Age),
   person(B,Y,Build,Char,Age),
   not(X=Y).
match(A,B) :-
   person(A,X,_,_,_),
   likes(A,Music,Sport,Rec),
   opp(X,Y),
   person(B,Y,_,_,_),
   likes(B,Music,Sport,Rec).
ideal(A,B) :- match(A,B), suits(A,B), suits(B,A).
```

The same can be modeled using LPL as following

```
SET p=/john,mary/; s=/m,f/; b=/large,medium/; c=/shy,extrovert/;
a=/mature,young/;
    m=/classical,jazz/; s=/swimming/; r=/travel,cinema/;

    person(p,s,b,c,a) = /john m large  shy       mature
                         mary f medium extrovert young/;
    prefers(p,b,c,a) =  / mary large extrovert young
                          john (small medium) (*) (young mature) /;
    likes(p,m,s,r) = /
         john jazz swimming cinema
         mary (classical jazz) swimming (travel cinema) /;
    opp(s,s) = /m f , f m/;

    suits(p2=p,p1=p) =
      Exist(x=s,y=s,b,c,a)
        (Exist(d1=b,d2=c,d3=a) person(p1,x,d1,d2,d3)
         and prefers(p1,b,c,a) and person(p2,y,b,c,a)
         and not (x=y));
    match(p1=p,p2=p) =
      Exist(x=s,y=s,m,s,r)
         (Exist(d1=b,d2=c,d3=a) person(p1,x,d1,d2,d3)
          and likes(p1,m,s,r) and opp(x,y)
          and Exist(d1=b,d2=c,d3=a) person(p2,y,d1,d2,d3)
          and likes(p2,m,s,r));
    ideal(p1=p,p2=p) = match(p1,p2) and suits(p1,p2) and suits(p2,p1);
```

DEDUCTIVE DATABASES

Lloyd S. 18 Definition. typed first order theory (=many-sorted theory)

RECURSIVE DATA STRUCTURE IN LOGIC PROGRAMMING

A recursive data structure is a data type which contains recursively objects of the same type. A *list* is a typical recursive structure. Logic programming only allow terms as representations of individuals. But it is not very hard to *represent **lists** as **terms***. A syntax of the term is slightly extended. The list of three constant terms is represented by [a,b,c] and the list of variables is written as X.Y where X is the first element of the list and Y is the rest. Using this syntax it is easy to expressed the two LISP functions *car* and the *cdr* of a list:

car(Head,Head.Tail)

cdr(Tail,Head,Tail)

The query `:- cdr(X,[a,b,c])` , for example, is true if and only if X=[b,c].

Let us return to the example of paths in a graph. Above wew defined a path as

```
path(X,X) :-
path(X,Y) :- edge(X,Z) path(Z,Y)
```

This predicate can only indicate whether there is a path from some node to others or not. The path itself cannot be returned. Furthermore, if the path predicates goes into a loop, because the graph has some cycles, this could not be detected, because the path cannot remember nodes that have already been visited. To prevent such a difficulty the predicate path is defined differently as:

```
path(X,Y) :- path(X,Y,[X])
path(X,X,Visited)
path(X,Y,Visited) :- edge(X,Z)
                     not member(Z,Visited)
                     path(Z,Y,Z.Visited)
member(X,X.Y)
member(X,Y.Z) :- member(X,Z)
```

But this program still does not return the pathn itself. To return the path itself as a list we use the following definition

```
path(X,Y,Path) :- path(X,Y,[X],Path)
path(X,X,Visited,Visited)
path(X,Y,Visited,Path) :- edge(X,Z)
                          not member(Z,Visited)
                          path(Z,Y,Z.Visited,Path)
```

(=searching in a state space using a transition graph! (see Nilsson al 1990).

LOGIC PROGRAMMING AND FUNCTIONAL PROGRAMMING

The principle of pure functional programming could be stated as following:

“The value of an expression depends only on the values of its subexpressions, if any.” Logic programming (LP) shares many roots with functional programming (FP):

- both manipulate values rather than assignable cells
- both use heavily recursion
- both have ample opportunities for execution parallelism.

But they are also very different on several fundamental aspects:

- concerning the variables: 'call-by-value' binding versus unification
- concerning abstraction: higher order program entities (in FP) versus inherently non-deterministic execution (in LP).
- concerning input/output: one-directional (FP), does matter (LP).

To understand the last difference let's take an example.

EXAMPLE 1: APPEND

The list operator APPEND can be implemented in LP as following

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

In FP we have the following program

```
append(nil, Y) = Y
append(A.X, Y) = A.append(X, Y)
```

The differences might seem cosmetic. There is a fundamental distinction however. While the functional program can only query in one direction, the logic program can be queried in all directions. The following query in LP is just fine in both paradigm

```
:- append([1,2], [3,4,5], L)        returns L = [1,2,3,4,5]
```

But now consider the query which says find the list which produces together with [1,2] the list [1,2,3,4,5]:

```
:- append([1,2], L, [1,2,3,4,5]     returns L = [3,4,5]
```

which in fact is not an 'append' but a split of a list. This query is just fine in LP, but there is no possibility to make the query in FP. In FP a new function must be written. The query

```
:- append(X, Y, [1,2,3,4,5]) returns
```

is even more interesting. It says find all partitions of the list [1,2,3,4,5]. The result is the set of assignments:

```
{X=[], Y=[1,2,3,4,5]}
{X=[1], Y=[2,3,4,5]}
```

```
{X=[1,2], Y=[3,4,5]}
{X=[1,2,3], Y=[4,5]}
{X=[1,2,3,4], Y=[5]}
{X=[1,2,3,4,5], Y=[]}
```

Hence the append predicate in LP can be used to appends or to split two lists. Logic programs are indifferent about the fact what is input and what is output. This is very different in functional programming. Three different functions would have been needed to define it in FL.
A second difference is that in logic programming the results need not be totally grounded. Consider the query append(X, Y, [1,2,3,4,5] again. Another output of the query could have been: {X=[1,2,3,4,5]\Y}.

EXAMPLE 2: ADD

A second example illustrates the point again: define a function (predicate) for addition. In FP (Scheme) we have

```
(define (add x y)
  (+ x y))
```

In LP (Prolog III) we can write (predicate begin with an uppercase, variables with a lowercase letter):

```
Add(x,y,z) -> , {x=y+z};
```

It is easy to query now in all directions. For example the three queries

```
Add(x,2,3);
Add(5,x,3);
Add(5,2,x);
```

yield the results {x=5}, {x=2}, and {x=2} respectively. The query

```
Add(x,y,2);
```

is even more interesting. Prolog III returns the linear system

```
{x = y_1 + 2, y = y_1}
{x = y_2 + 2, y = y_2}
```

BEISPIEL 3: FIBONACCI

Fibonacci numbers are defined recursively as

$$F(1) = 1, \quad F(2)=1, \quad F(n) = F(n-2)+F(n-1) \text{ for } n>2$$

which produces the sequence: 1, 1, 2, 3, 5, 8, 13, 21, ...
1) There exists a closed form which finds the nth Fibonacci number

$$F(n) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

2) the Loop to find the n-th number:

```
F0 := 0;    F1 := 1;
for i := 0 to n do begin
  temp := F1;  F1 := F0 + F1;  F0 := temp;
end;
return(F0);
```

3) as functional program

```
(function Fib (n)
  (if n<=2 then return(1) else return(Fib(n-1)+Fib(n-2))
```

4) logic program

```
fib(1,1) :-
fib(2,1) :-
fib(X,Y) :- Y=V+W, Fib(X-1,V), Fib(X-2,W)
```

5) as linear equation system (using Gaussian elimination):

$F_1 = 1$
$F_2 = 2$
$F_n = F_{n-1} + F_{n-2}$ for n>2

FP is based on the lambda calculus, LP on the first order logic theory. Both have there advantages and disadvantages. But the most striking difference is their treatment of directionality.

example: path in a graph (Schmitt, p.2ff)

Zur Negation:

A :-
B :- A
:- not B

Wie versucht Prolog dieses Problem zu lösen. Wir haben hier keine Hornklausel mehr. Prolog tut also zunächst so als ob not B nicht negiert wäre. Es versucht als B abzuleiten. Da dies gelingt, kann not B offenbar nicht abgeleitet werden. Das bedeutet aber nicht, dass not B falsch (unerfüllbar) ist, es könnte ja auch erfüllbar sein. Da Prolog aber nur eine positive (yes) oder negative Antwort (fail) gibt....
In dieser close world assumtion wird jedes Prädikat, das nicht ableitbar ist, als falsch angenommen. (:- not B is not a horn clause because we have to read it as F :- not B where F is the (positve) False litteral and it reads as F or B which contains two positive literals.

LOGIC PROGRAMMING AND INTEGER PROGRAMMING

Any (pure) logic program can be reformulated as an *integer program*. An integer program is a linear inequation system with a minimizing (linear) objective function which can be formulated as

$$Ax \geq b$$
$$x \geq 0$$
$$\text{minimize } cx$$

where A is a mxn matrix of integers, x is a unknown n-vector where the elements are elements of $\{0,1\}$, b is an m-vector of integers, and c is a n-vector of integers.

The whole reformation procedure is described in another paper (Hürlimann 1993a). To take two simple examples, we compare the formulations.

Example 1: Suppose we have the following knowledge base which contains a simple fact and a rule: p, $p \rightarrow q$. We wanted to prove q. The following table compares the three formulations: logic, logic program, and inter program.

Notation in logic (set of clauses)	logic program (notation in PROLOG)	inter (0-1) programming formulation
p	$p:-$ (a fact)	$p \geq 1$
$q \vee \bar{p}$	$q:-p$ (a rule)	$q \geq p$
$\bar{q}$	$:-q$ (a goal)	minimize q

Using resolution, it is easy to prove the inconsistency: the resolvent of the first and the second clause is q. The resolvent of q and the third clause is the empty clause and we have proved the inconsistency. Using SLD-resolution in the logic program, we unify the tail of the goal with the rule. The new goal is :-p. Unify the new goal with the first fact produces the empty clause. The integer program is trivial: both variables are fixed at 1. Therefore, to minimize q means to assign the minimal possible value (which is 1) to q. Since we interpret the value 1 as TRUE and the value 0 as FALSE. q has turned out TRUE. This can be interpreted as proof of q.

Example 2: Suppose our knowledge base contains the two clauses: $\{p(a), \forall x(p(x) \rightarrow q(x))\}$. We wanted to prove $q(a)$. Again the three formulation are summarized in a table:

Notation in logic (set of clauses)	logic program (notation in PROLOG)	inter (0-1) programming formulation
$p(a)$	$p(a):-$	$p_a \geq 1$
$\forall x(p(x) \rightarrow q(x))$	$q(X):-p(X)$	$q_i \geq p_i$ for all $i \in D$
$\neg q(a)$	$:-q(a)$	minimize q_a $(a \in D)$

Again, resolution is trivial in logic and in logic programming formulation and needs two steps. To formulate the model in inter programming we need *explicitly* a domain *D*. This is the closed world assumption in logic programming. In integer programming, all variables are fixed to 1 and the solution is trivial, since q_q is also fixed to 1.

The two example are instructive but not very interesting. More real live examples can be found in my paper [Hürlimann 1993a]. The translation of logical programs into IP models may have four advantages:

1) If the all clauses of the logic program are Horn-clauses then the corresponding IP model can be solved using the LP-relaxation.
2) The clauses need not to be Horn-clauses, they need even not be clauses, any logical statement can be translated into one or several IP statement.
3) Mathematical and logical knowledge can be mixed within the same paradigm.
4) Non-monodonic logic stetement canb also be treated as linear mathematical statements (Yager).

CONSTRAINT LOGIC PROGRAMMING (CLP)

The paradigm of logic programming has been extended to constraint logic programming by several authors. The development was triggered by the request – or the necessity – to manipulate conveniently some data structures in logic programming. In the first place one should think of numbers. Numbers cannot be manipulated conveniently within logic programming. Several languages such as Prolog have included some ad-hoc predicates (addition, equality, greater, less and some other operators) to handle calculation. But apart from its being a makeshift solution, it cannot be used with free variables. The following Prolog program for faculty makes this point clear

```
fac(0,1).
fac(N,F) :- M is N-1, fac(M,G), F is N*G.
```

Any query of the form fac(n,F) where n is a number can be answered successfully. But the query fac(X,5040) will produces an error message, because the predicate “M is N-1”, which is evaluated first, cannot assign the variable M since N is free.

The query of how many rabbits and pigeons are needed to get 12 heads and 34 legs, cannot be answered easily by Prolog. But formulating this problem as a linear program is easy. If the number of rabbits is R and the number of pigeons is P then the linear program is:

```
P ≥ 0
R ≥ 0
P + R = 12
2*P + 4*R = 34
```

An Prolog III program is now

```
atom :- literals '{' constraints '}'
```

PrologIII

CHIP

Finite domain constraint satisfaction problems (CSP) can be described by a set of variables $\mathbf{x} = x_j$ $(j = \{1\text{K } m\})$ to be instantiated on a finite domain D_j and which are subject to a set of constraints $C_i(\mathbf{x})(i = \{1\text{K } n\})$. Finite domain consrait problems are are simple to formulate using mathematical and logical formalism. There are plenty of applications for such problems: graph coloring, VLSI routing, hardware design, many operational research problems (like cutting stock problems, the traveling salesperson problem, the warehouse problem, scheduling problems) and many other combinatorial problems. Most of these problems are NP-complete. CSP-problems can be depicted as labelled hypergraphs ..... Dechter Pearl....

LOGIC PROGRAMMING AND LPL

Deklarative Sprache: what to do? Describe you problem!
Imperative (prozedurale): How to do? Give me the sequence of instructions!
That's the common view! But it is misleading!
Interactive (interpretiert): query Abfrage --- batch: (compiliert)
Prolog only performs a mechanical derivation

EXAMPLE 1: THE SOLAR SYSTEM (SPENCER-SMITH R. P.74FF)

(a simple knowledge base)

natural language formulation

```
Venus is a planet
The Earth is a planet
The sun is a star
The moon orbits around the Earth
All planets orbit around the sun
A moon orbits around a planet
The solar system is geocentric if the sun orbits around the earth
The solar system is heliocentric if the earth orbits around the sun
A satellite is a heavenly body which orbits around another
A heavenly body is a star, a planet, or a moon
```

Queries:

```
Is Venus a planet?
What are the planets?
What orbits the earth?
What orbits around what?
Is the solar system geocentric?
Is there an object that orbits another which orbits the sun?
```

predicate logical formulation

```
planet(Venus)
planet(the_earth)
star(the_sun)
orbits(the_moon,the_earth)
∀x (planet(x) ∅ orbits(x,the_sun))
∀x ((∃y (orbits(x,y)  planet(y))) ∅ moon(x))
      { Skolem form is: ∀xy (¬orbits(x,y) Δ ¬planet(y) Δ moon(x)) }
geocentric(the_solar_system) ♦ orbits(the_sun,the_earth)
heliocentric(the_solar_system) ♦ orbits(the_earth,the_sun)
∀x (heavenly_body(x) ♦ star(x)  planet(x)  moon(x))
∀x (satellite(x) ♦ ∃y (heavenly_body(x)  heavenly_body(y)  orbits(x,y)))
     { Skolem form is: ∀xy (satellite(x) Δ ¬heavenly_body(x) Δ ¬heavenly_body(y)
                            Δ ¬orbits(x,y)) }
```

Prolog formulation

```
planet(venus).
planet(the_earth).
star(the_sun).
orbits(the_moon,the_earth).
orbits(X,the_sun) :- planet(X).
moon(X) :- orbits(X,Y), planet(Y).
geocentric(the_solar_system) :- orbits(the_sun,the_earth).
heliocentric(the_solar_system) :- orbits(the_earth,the_sun).
heavenly_body(X) :- star(X); planet(X); moon(X).
satellite(X) :- heavenly_body(X), heavenly_body(Y), orbits(X,Y).
```

Queries:

```
?-  planet(venus)
?-  planet(X)
?-  orbits(X,the_earth)
?-  orbits(X,Y)
?-  geocentric(the_solar_system)
?-  orbits(X,Y), orbits(Y,the_sun)
```

LPL formulation

```
SET
x; planet(x); star(x); orbits(x,x); moon(x); s; geocentric(s); heliocentric(s);
heavenly_body(x); satellite(i=x);
    ! x = /venus, the_earth, the_sun, the_moon/;  needs not to be defined!

planet(x) = /venus, the_earth/;
star(x) = /the_sun/;
orbits(i,j) = (i='the_moon' and j='the_earth') or (planet(i) and j='the_sun');
     !orbits(i,j) = /the_moon the_earth/ or (planet(i) and j='the_sun');!
moon(i) = Exist(j=x) (orbits(i,j) and planet(j));
s = /the_solar_system/;
geocentric(s) = orbits('the_sun','the_earth');
heliocentric(s) = orbits('the_earth','the_sun');
heavenly_body(x) = star or planet or moon;
satellite(i) = heavenly_body(i) and Exist(j=x) (heavenly_body(j) and
orbits(i,j));
```

Queries:

```
PRINT:     planet('venus');
PRINT(x):  planet;
PRINT(x):  orbits(x,'the_earth');
PRINT(i=x,j=x):  orbits(i,j);
PRINT:       geocentric('the_solar_system');
PRINT(i=x):  Exist(j=x) (orbits(i,j) and orbits(j,'the_sun'));
```

EXAMPLE 2: BLOOD TRANSFER KNOWLEDGE (SPENCER R. P.92FF):

(negation)

Natural language formulation

```
blood from group agglutinates  blood from groups
   A                                B, AB
   B                                A, AB
   AB                               (none)
   O                                A, B, AB

Jones has blood group B.
Smith has blood group O.
A person (the donor) can donate blood to another (the recipient) if the blood
from the recipient does not agglutinate the blood from the donor.
```

Queries:

```
Does A agglutinate O?
Does A not agglutinate O?
Can Jones donate his blood to smith?
```

Predicate logical formulation

```
agglutinates(a,b)
agglutinates(a,ab)
agglutinates(b,a)
agglutinates(b,ab)
agglutinates(o,a)
agglutinates(o,b)
agglutinates(o,ab)
```

```
type(jones,b)
type(smith,o)
∀vw (can_donate(v,w) <-- (∃xy (type(v,x)  type(w,y)  ¬agglutinates(y,x))))
    { Skolem form is : ∀vwxy (can_donate(v,w) Δ ¬type(v,x) Δ ¬type(w,y)
                              Δ agglutinates(y,x))   }
```

Note that the last formula is not a Horn clause!

Prolog formulation

```
agglutinates(a,b).
agglutinates(a,ab).
agglutinates(b,a).
agglutinates(b,ab).
agglutinates(o,a).
agglutinates(o,b).
agglutinates(o,ab).
type(jones,b).
type(smith,o).
can_donate(V,W) :- type(V,X), type(W,Y), not agglutinates(Y,X).
```

Although the last rule is not a Horn-clause, Prolog will accept and evaluate it. The reason lays in the order in which Prolog evaluates (or better: unifies) the predicates: from left to right. Since *type(V,X)* and *type(W,Y)* are unified before *not agglutinates(Y,X)*, the variables *Y* and *X* are bound before *not agglutinates(Y,X)* will be unified: there is no variable left to be bound. Hence, we have do be careful on the order of the literals within the clauses. The same rule declared as

```
can_donate(V,W) :- not agglutinates(Y,X), type(V,X), type(W,Y).
```

would not be accepted by most Prolog.

Queries:

```
?-  agglutinates(a,o)
?-  not agglutinates(a,o)       ; Prolog fails to give the correct answer!
?-  can_donate(jones,smith)
```

LPL formulation

```
SET b=/a,b,ab,o/; p=/jones,smith/;
agglutinates(b,b) = / a (a,ab) , b (a,ab) , o (a,b,ab) /;
type(p,b) = / jones b , smith o /;
can_donate(v=p,w=p) = Exist(x=b,y=b) (type(v,x) and type(w,y) and not
agglutinates(y,x);
```

Queries:

```
PRINT:  agglutinates('a','o');
PRINT:  not agglutinates(a,o);          ! LPL gives the correct answer!
PRINT:  can_donate('jones','smith');
```

EXAMPLE 3: ADVICE ON NUTRITION (SPENCER R. P.102FF):

Prolog

```
pregnant(mary).
pregnant(jane).
has_cold(john).
anaemic(jo).
has_rickets(jim).
deficient_in(fred,vitamin_A).
```

```
found_in(vitamin_A,carrots).
found_in(vitamin_A,X) :- oily_fish(X).
found_in(vitamin_A,X) :- green_leafy_veg(X).
found_in(folic_acid,oranges).
found_in(folic_acid,X) :- pulse(X).
found_in(folic_acid,X :- green_leafy_veg(X).
found_in(vitamin_C,oranges).
found_in(vitamin_D,X) :- oily_fish(X).
found_in(vitamin_D,eggs).
found_in(vitamin_E,X) :- green_leafy_veg(X).
found_in(calcium,X) :- pulse(X).
found_in(iron,spinach).
green_leafy_veg(lettuce).
green_leafy_veg(kale).
green_leafy_veg(parsley).
pulse(lentils).
pulse(red_kidney_beans).
pulse(chick_peas).
oily_fish(mackerel).
oily_fish(sardines).
oily_fish(herring).

needs(X,folic_acid) :- pregnant(X).
needs(X,vitamin_C) :- has_cold(X).
needs(X,iron) :- anaemic(X).
needs(X,Y) :- deficient_in(X,Y).
needs(X,calcium) :- has_rickets(X).
needs(X,vitamin_D :- needs(X,calcium).

should_eat(Person,Food) :- needs(Person,Nutrient), found_in(Nutrient,Food).
```

LPL formulation

```
SET f = /carrots, oranges, eggs, spinach, lettuce, kale, parsley, lentils,
         red_kidney_beans, chick_peas, mackerel, sardines, herring/;
    p = /mary, jane, john, jo, jim, fred/;
    i = /vitamin_A, vitamin_C, vitamin_D, vitamin_E, calcium, iron, folic_acid/;

    pregnant(p) = /mary, jane/;
    has_cold(p) = /john/;
    anaemic(p) = /jo/;
    has_rickets(p) = /jim/;
    deficient_in(p,i) = /fred vitamin_A/;

    green_leavy_veg(f) = /lettuce, kale, parsley/;
    pulse(f) = /lentils, red_kidney_beans, chick_peas/;
    oily_fish(f) = /mackerel, sardines, herring/;
    found_in(i,f) =
      (i='vitamin_A' and (f='carrots' or oily_fish or green_leafy_veg))
      or (i='folic_acid' and (f='oranges' or pulse or green_leafy_veg)
      or (i='vitamin_C' and f='oranges')
      or (i='vitamin_D' and (oily_fish or f='eggs'))
      or (i='vitamin_E' and green_leafy_veg)
      or (i='calcium' and pulse)
      or (i='iron' and f='spinach');
    needs(p,i) = (pregnant and i='folic_acid)
      or (has_cold and i='vitamin_C')
      or (anaemic and i='iron')
      or deficient_in
      or (has_rickets and i='calcium');
    needs(p,i) = (if,i='vitamin_D',needs(p,'calcium'),needs(p,i));

    should_eat(p,f) = Exist(i) (needs(p,i) and found_in(i,f));
```

EXAMPLE 4: SIMPLE LIST IN PROLOG: A SIMPLE GRAMMAR

Prolog

```
sentence(List) :- noun_phrase(N),
                  verb_phrase(V),
                  append(N,V,List).
noun_phrase(X) :- p_name(X).
noun_phrase(List) :- det(D),
                     c_noun(C),
                     append(D,C,List).
verb_phrase(X) :- intrans(X).
verb_phrase(List) :- trans(V),
                     noun_phrase(N),
                     append(V,N,List).
det([the]).
det([every]).
det([a]).
c_noun([man]).
c_noun([woman]).
c_noun([city]).
c_noun([country]).
intrans([walks]).
intrans([exists]).
trans([is]).
trans([loves]).
p_name([john]).
p_name([mary]).
p_name([london]).
p_name([england]).
```

(Note: the list notation [] is necessary, since the parameters to append are lists: append([],[],[]).

Queries:

```
?-  verb_phrase([loves,every,woman])
?-  sentence([john,is,a,man])
```

LPL

```
SET p = /john mary london england/;
    d = /the every a/;
    c = /man woman city country/;
    t = /is loves/;
    i = /walks exists/;
    n = p union (d,c);
    v = i union (t,n);

    det(d) = /the every a/;
    p_name(p) =  /john mary london england/;
    c_noun(n) = /man woman city country/;
    trans(t) = /is loves/;
    intrans(i) = /walks exists/;

    sentence(n,v) = noun_phrase(N) and verb_phrase(V);
    noun_phrase(n) :- p_name(p) or det(d) and c_noun(c);
    verb_phrase(v) :- intrans(i) or trans(t) and noun_phrase(n);

?:  verb_phrase('loves every woman');
?:  sentence('john is a man');
```

EXAMPLE 5: A NUMERIC EXAMPLE:

EXAMPLE 7A: ON SIMPLE RECURSION:

```
married(X,Y) :- married(Y,X).
married(fred,jane).
?- married(jane,fred)   ; will not return, infinite recursion!!
```

But the following is a correct formulation in Prolog

```
married(fred,jane).
married(X,Y) :- married(Y,X).
?- married(jane,fred)   ; will not return, infinite recursion!!
```

Another solution is

```
wedded(fred,jane).
married(X,Y) :- wedded(X,Y).
married(Y,X) :- wedded(X,Y).
```

LPL

```
SET i;
    married(i,i) = /fred jane/;
    married(i,j) = if(i<=j,married(i,j),married(j,i);
```

or

```
SET
  wedded(i,i) = /fred jane/;
  married(x,y) = wedded(x,y) or wedded(y,x).
```

EXAMPLE 7B: ON SIMPLE RECURSION:

get_to

EXAMPLE 7C: REAL RECURSION: THE TOUR OF HANOI

EXAMPLE MIT TURBO PROLOG

Turbo Prolog formulation

```
/*
   Turbo Prolog 2.0, Answer to first Exercise on page 121.
   Copyright (c) 1986, 88 by Borland International, Inc
*/

Domains
    person = symbol

Predicates
    special_taxpayer(person)
    average_taxpayer(person)
    is_a_citizen(person)
    married(person,person)
    has_kids(person,integer)
    has_two_kids(person,integer)
    makes_bucks(person,integer)
    right_income(person,integer)
```

```
Clauses
    is_a_citizen(tom).
    is_a_citizen(albert).
    is_a_citizen(suzie).
    is_a_citizen(bonnie).
    is_a_citizen(Person):-
        married(Person,Spouse),
        is_a_citizen(Spouse),!.   /* The cut must be placed here to prevent
                                     unnecessary backtracking. To see this,
                                     trace thru the program, first with
                                     and then without the cut.
                                  */
    married(tom,chris).
    married(albert,rachel).
    married(fred,suzie).
    married(duke,joanne).

    has_kids(albert,3).
    has_kids(suzie,2).
    has_kids(fred,2).
    has_kids(bonnie,1).
    has_kids(tom,0).

    has_two_kids(Person,X):-
        has_kids(Person,X),
        X=2.

    makes_bucks(tom,250).
    makes_bucks(fred,3000).
    makes_bucks(albert,1500).
    makes_bucks(suzie,0).

    right_income(Person,N):-
        makes_bucks(Person,N),
        500 <= N,
        N <= 2000.

    average_taxpayer(Person):-
        is_a_citizen(Person),
        right_income(Person,_),
        has_two_kids(Person,_),
        married(Person,_),
        write(Person," is an average taxpayer").

    special_taxpayer(Person):-
        not(average_taxpayer(Person)),
        write(Person," is a special taxpayer").

Goal
    special_taxpayer(fred).
```

LPL formulation

```
set
    person;
    special_taxpayer(person);
    average_taxpayer(person);
    is_a_citizen(person);
    married(person,person);
coef
    has_kids(person);
    has_two_kids(person);
    makes_bucks(person);
    right_income(person);

set
    is_a_citizen = /tom albert suzie bonnie/;
    is_a_citizen(Person)=is_a_citizen and
        Exist(Spouse=Person) (married(Person,Spouse) and
        is_a_citizen(Spouse));
```

```
    married = /tom,chris , albert,rachel , fred,suzie , duke,joanne/;
coef
    has_kids = /albert 3 , suzie 2 , fred 2 , bonnie 1 , tom,0/;

    has_two_kids(Person)=
        has_kids(Person)=2;

    makes_bucks= /tom 250 , fred 3000 , albert 1500 , suzie 0/;

    right_income(Person)=
        500 <= makes_bucks <= 2000;

    average_taxpayer(Person)=
        is_a_citizen(Person) and
        right_income(Person) and
        has_two_kids(Person) and
        Exist(Spouse=Person) married(Person,Spouse);

    special_taxpayer(Person):-
        not(average_taxpayer(Person));

print:
    special_taxpayer(fred);
end
```

PROLOG III Version

```
/*
        Le fameux menu equilibre .
*/

RepasLeger(h,p,d) ->
   HorsDoeuvre(h,i) Plat(p,j) Dessert(d,k)
   {i>=0,j>=0,k>=0,i+j+k<=10};

Plat(p,i) -> Viande(p,i);
Plat(p,i) -> Poisson(p,i);

HorsDoeuvre(radis,1) ->;
HorsDoeuvre(pate,6) ->;

Viande(boeuf,5) ->;
Viande(porc,7) ->;

Poisson(sole,2) ->;
Poisson(thon,4) ->;

Dessert(fruit,2) ->;
Dessert(glace,6) ->;

/*
        Question
*/

RepasLeger(h,p,d);
```

LPL Version 1

```
(*
        Le fameux menu equilibre .
*)

SET p = /radis pate boeuf porc sole thon fruit glace /; "plats"
COEF
  HorsDoeuvre(p) = / radis 1 , pate 6 /;
  Viande(p) = / boeuf 5 , porc 7 /;
  Poisson(p) = /sole 2 , thon 4 /;
  Dessert(p) = / fruit 2 , glace 6 /;
```

```
SET
  Plat(p) = Viande OR Poisson;
  RepasLeger(h=p,p1=p,d=p) =
    HorsDoeuvre[h] AND Plat(p1) AND Dessert(d)
    AND  HorsDoeuvre[h] + Viande(p1) + Poisson(p1) + Dessert(d) <= 10;

(*
        Question
*)

PRINT RepasLeger;
END
```

Die Ausgabe von LPL ist:

```
REPASLEGER(P,P,P)
 {RADIS BOEUF FRUIT}
 {RADIS PORC FRUIT}
 {RADIS SOLE FRUIT}
 {RADIS SOLE GLACE}
 {RADIS THON FRUIT}
 {PATE SOLE FRUIT}
```

LPL Version 2

```
SET p = / radis pate boeuf porc sole thon fruit glace /; "plats"
    HorsDoeuvre(p) = / radis pate /;
    Viande(p) = / boeuf porc /
    Poisson(p) = / sole thon /;
    Dessert(p) = / fruit glace /;
    Plat(p) = Viande OR Poisson;
COEF
  Calories(p) = / 1 6 5 7 2 4 2 6 /;

SET
    RepasLeger(HorsDoevre,Plat,Dessert) =
       SUM(p) Calories <= 10;

(*
        Question
*)

PRINT RepasLeger;
END
```

LPL Version 3

```
SET p = / radis pate /;
    q = / boeuf porc /
    q1 = / sole thon /;
    r = / fruit glace /;
    Pl = / boeuf porc sole thon / (* = q UNION q1;   UNION : nicht implementiert
*)
COEF
  HorsDoeuvre(p) = / radis 1 , pate 6 /;
  Viande(q) = / boeuf 5 , porc 7 /;
  Poisson(q1) = /sole 2 , thon 4 /;
  Dessert(r) = / fruit 2 , glace 6 /;
  Plat(Pl) = Viande(Pl IN q) + Poisson(Pl IN q1);

SET
    RepasLeger(p,Pl,r) =
       HorsDoeuvre[p] + Plat(Pl) + Dessert(r) <= 10;

(*
        Question
```

```
*)

PRINT RepasLeger;
END
```

LPL Version 4

```
SET p = / radis pate boeuf porc sole thon fruit glace /; "plats"
    q = / HorsDoevre Plat Dessert /;
    Rel(q,p) = / HorsDoevre (radis pate) , Plat (boeuf porc sole thon) ,
                 Dessert (fruit glace) /;
COEF   Calories(p) = / 1 6 5 7 2 4 2 6 /;

SET
    RepasLeger(p,q) =
       Rel AND SUM(p) Calories <= 10;

PRINT RepasLeger;
END
```

Predicates can be expressed in LPL at different levels:
1) as model restrictions:

```
MODEL plus(X,Y,Z) :- Z = X + Y;  "variables given in a parameter-list")
MODEL TimeLag{t}(X{t},Y{t}) :- X[t+1] = X[t] + Y[t];
```

2) as (logical) variables

```
VAR path{i,i}; edges{i,i} LOGICAL;
MODEL A1{i}: path[i,i];
      A2{i1,i2} : path[i1,i2]  <-- Exist{i3} (edge[i1,i3]  and  path[i3,i2];
```

3) predicates as sets
example 'plat leger'

Predicates and function
as predicates

```
MODEL plus(X,Y,Z) :- Z = X + Y;
greater(X,Y) :- X > Y;
PROVE : greater(X,Y) and plus(X,1,Y);
```

as function

```
MODEL Z:=plus(X,Y) : Z = X + Y
PROVE greater(plus(X,1),X)
```

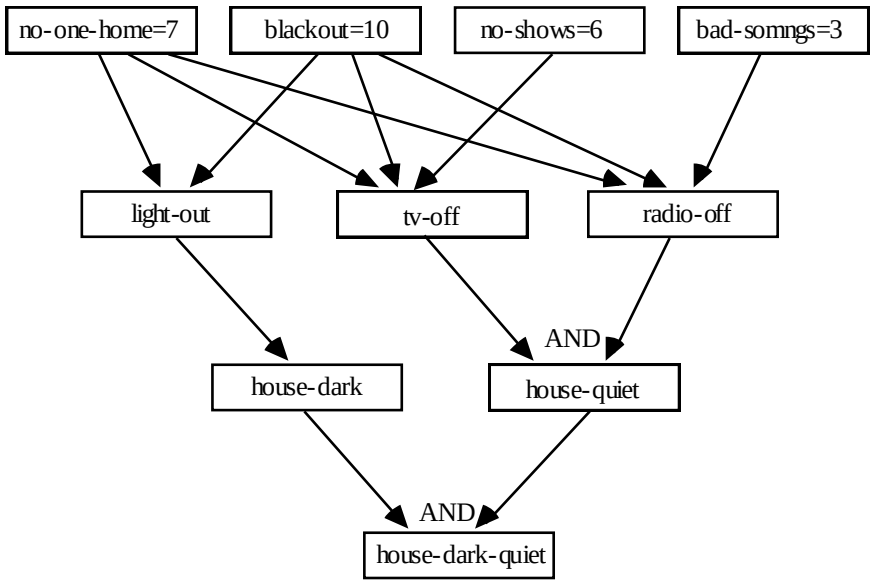
EXAMPLE: COST-BASED ABDUCTION [SANTOS 1994]

Abduction is the problem of finding the best explanation for a given set of observations. Suppose, as an example, the following situation: John visits Mary's house and finds the place quiet and dark. He concludes that Mary is not home. How has he arrived to conclude that? We might formulate his knowledge as the following set of rules

house-dark □ house-quiet □ house-dark-quiet
light-out □ house-dark

no-one-home Δ blackout	□ lights-out
tv-off □ radio-off	□ house-quiet
house-dark □ house-quiet	□ house-dark-quiet
no-one-home Δ no-shows Δ blackout	□ tv-off
no-one-home Δ bad-songs Δ blackout	□ radio-off

The set of rules can also be formulated as a OR-AND directed acyclic graph as following



All sorts of hypotheses (the first row) could be articulated and provided with a 'probability' (or 'cost') (the higher the inprobable). Chains of deduction can be made imposing logical rules to get to the observation. To find the most 'likelihood' expanation, we can minimize the costs over all hypotheses.

It is very easy the formulate the whole model using LPL

```
SET nodes;
  observation{nodes};
  hypotheses{nodes};
  Anodes{nodes};   (* the AND-nodes *)
  Onodes{nodes};  (* the OR-nodes *)
  arcs{nodes,nodes}; (* the arc set of the graph *)
COEF  cost{hypotheses};
VAR x{nodes} BINARY;
MODEL
  obser{i=observation}: x[i]=1;   (* all observations are true *)
```

```
  ANDs{i=Anodes}: AND{j=nodes | arcs[j,i]}  x[j]  IMPL x[i];
  ORs{i=Onodes}: OR{j=nodes | arcs[j,i]}  x[j]  IMPL x[i];
MINIMIZE c: SUM{i=hopotheses} cost[i]*x[i];
END
```

LPL translates this formulation automatically into a MIP-problem as described in Santos [1994].

REFERENCES

BOTHNER P., KAEHLER W.-M., [1991], Programmieren in Prolog, Eine umfassende und praxisgerechte Einführung, (mit Disketten), vieweg, Braunschweig.

DeGROOT D., LINDSTROM G., [1986], Logic Programming, Functions, Relations, and Equations, Prentice-Hall.

DUFFY D., [1991], Principles of Automated Theorem Proving, Wiley, Chichester.

HÜRLIMANN T. [1993a], IP, MIP, and Logical Modeling using LPL, Institute of Informatics (formerly Institute for Automation and Operations Research), Working Paper No 205, last update: September 1993, Fribourg.

HÜRLIMANN T. [1993b], Logic: a Survey, Institute of Informatics (formerly Institute for Automation and Operations Research), Working Paper No 215, September 1993, Fribourg.

KOWALSKI R. [1979], Logic for Problem Solving, The Computer Science Library.

LLOYD J.W., [1987], Foundations of Logic Programming, (2nd edition), Springer.

LOCK H.C.R., [1993], The Implementation of Functional Logic Programming Languages, R. Oldenbourg Verlag, München.

MAIER D., WARREN D.S., [1988], Computing with Logic, Logic Programming with Prolog, The Benjamin/Cummings Publ. Comp.

MIZOGUCHI F. (ed), [1991], Prolog and its applications, a Japanese perspective, Chapman and Hall Computing, London.

NILSSON U., MALUSZYNSKI J., [1990], Logic, Programming and Prolog, John Wiley & Sons, Chichester.

SANTOS E. (Jr), [1994], A linear constraint satisfaction approach to cost-based abduction, in :Artificial Intelligence 65(1) (1994), p1–27.

SPENCER-SMITH R., [1991], Logic and Prolog, Harvester, Wheatsheaf, New York.

SPERSCHNEIDER V., ANTONIOU G., [1991], LOGIC, a Foundation for Computer Science, Addison-Wesley Publ. Comp., Wokingham, (GB).

SCHMITT P.H., [1992], Theorie der logischen Programmierung, Springer-Lehrbuch, Berlin.

ULLMAN J.D., [1982], Principles of Database Systems, Computer Science Press.

ALTY J.L., COOMBS M.J. [1986], systèmes experts, concepts et exemples, Masson.

LEVINE R.I., DRANG D.E., EDELSON B. [1986], A Comprehensive Guide to AI and Expert Systems, McGraw-Hill Book Comp.

YAGER R.R. [1987], A mathematical programming approach to inference with the capability of implementing default rules, in: Gaines B.R., Boose J.H. (eds), Machine Learning and Uncertain Reasoning, Knowledge-Based Systems Vol 3, Academic Press, London, 1990, p261–290.

ZADEH L.A., [1985], The Role of Fuzzy Logic in the Management of Uncertainty in Expert Systems, in: Gupta M.M., Kandel A., Bandler W., Kiska J.B., Approximate Reasoning in Expert Systems, North-Holland, Amsterdam, 1985, p3–31.

POST S.D., BELL C.E., [1991], Default Reasoning Through Integer Linear Programming, in: BROWN D.E., WHITE C.C. III (eds.), [1991], Operations Research and Artificial Intelligence, Kluwer Acad. Publ.