MINISTÉRIO DA HABITAÇÃO E OBRAS PÚBLICAS
Laboratório Nacional
de Engenharia Civil

HOW TO SOLVE IT WITH PROLOG

HELDER COELHO
JOSÉ CARLOS COTTA
LUÍS MONIZ PEREIRA

3rd. edition

Lisboa, Julho de 1982

MINISTÉRIO DA HABITAÇÃO E OBRAS PÚBLICAS

LABORATÓRIO NACIONAL DE ENGENHARIA CIVIL

HOW TO SOLVE IT WITH PROLOG

Helder Coelho

José Carlos Cotta

Luís Moniz Pereira

3nd. edition

LISBOA, 1982

ABSTRACT

Our purpose is to present the outstanding features of PROLOG through a collection of small problems and exercises, divided in general application areas such as deductive reasonins over data bases, natural languase, symbolic calculus, etc.

KEY WORDS AND PHRASES: PROLOG, Logic Programmins, Problem Solvins

SUMÁRIO

O nosso objectivo consiste em apresentar as características proeminentes da linguasem de programação PROLOG, através de uma colectânea de pequenos problemas e exercícios divididos em áreas gerais de aplicação, tais como raciocínio dedutivo sobre bases de dados, língua natural, cálculo simbólico, etc.

PALAVRAS E FRASES CHAVE: PROLOG, Programação em Lógica, Resolução de Problemas

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Over the past nine years there has been a great deal of research work in logic programming (see Bibliography). A programming language, PROLOG, based on predicate calculus was developed, and several interpreters and two compilers were written for the main computer systems (see Appendix 1). A number of practical working systems have been implemented (see Appendix 2).

Our purpose is to present the outstanding features of PROLOG, and of the logic by the clause approach, through a collection of small problems and exercises, divided in general application areas such as deductive reasoning over data bases, natural language, theorem proving, planning and symbolic calculus. Our aim was motivated by the inexistance of a document to help people become initiated in PROLOG, and to learn it by the example.

The present text is not intended to be a complete and definite collection, although we consider that the major examples, taken from literature and duly referenced, have been included. The respective programs were adapted to the PROLOG

running on the DEC-10 systems. The programs for the problems and exercises not referenced were written by the authors.

All logic programs were written and tested on the DECsystem-10 PROLOG (Warren,1977;Pereira et al.,1978).

For your convenience we have included a list of PROLOG relevant addresses (Appendix 3), as well as a specification of system predicates (see Appendix 4).

CHAPTER 2

A BRIEF SURVEY OF PROLOG

PROLOG is a simple but powerful programming language founded on symbolic logic, developed at the University of Marseille (Roussel,1975), as a practical tool for logic programming (Kowalski,1974;1979) (Colmerauer,1975) (Emden,1975). A major attraction of the language, from a user's point of view, is ease of programming. Clear, readable, concise programs can be written quickly with minimum error. Recently, an efficient compiler and an interpreter were implemented on the DECsystem-10 (Warren,1977a;1979). A user's guide is available (Pereira et al,1978).

Like LISP, PROLOG is an interactive language designed primarily for symbolic data processing. Both are founded on formal mathematical systems -- LISP, on the lambda calculus, is typically used for the definition of functions, PROLOG, on a powerful subset of classical logic (Tarnlund,1977), is typically used for the definition of relations. Pure LISP in fact can be viewed as a specialization of PROLOG (Warren et al,1977).

2.1 SYNTAX

Here is a PROLOG program, consisting of two clauses, for specifying the concatenation relation of two lists:

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

In general, a PROLOG program consists of a set of procedures, where each procedure comprises a number of clauses. The procedure name is called a predicate ("concatenate" above), and has an arity which is the number of its arguments (3 above). A clause begins with a head or procedure entry point, and continues with a body. If the body is not empty it is separated from the head by ":-" (2nd clause above). Every clause terminates with a ".". The head displays a possible form of the arguments to the procedure's predicate. The body consists of a number (possibly zero) of goals or procedure calls, which impose conditions for the head to be true. If the body is empty we speak of a unit clause (1st clause above).

In general, all PROLOG objects are terms. A clause is a term, a predicate or a goal with their respective arguments, and the arguments themselves are terms. For example, a tree of 5 categories of a classification system is represented by the following term,

```
t(t(void,linguistics,void),
  'computing sciences',
  t(t(void,ai,void),applications,t(void,software,void)))
```

A term is either a variable (distinguished by an initial capital letter), an atom ('void' above), or a compound term. A compound term comprises a functor ('concatenate' or 't' above) of some arity N >= 1, and a sequence of N terms as its arguments ('t(void,ai,void)' above). An atom is treated as a functor of arity 0. A term of the form [H|T] stands for the list .(H,T), whose head is H and tail is T. The empty list is denoted [], and a list with exactly two elements by [A,B].

The second clause above is just infix notation for the term

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

where ':-' is a binary functor. The functor ':-' takes as arguments the head and the body of the clause. If a body has more than one goal the comma separating the goals is just another binary functor used in infix notation. The above term stands for a clause because it figures in the set of clauses for a procedure. It is dintinguished by a final '.'.

Apart from syntax conventions, the names and arities of terms (and their number) are arbitrary, except for a pre-defined set of procedures which are built into the implementation of the language, and which achieve input, output, arithmetic, etc (a listing of system procedures will be found in Appendix 4).

2.2 SEMANTICS

PROLOG differs from most programming languages in that there are two quite distinct ways to understand its semantics. The procedural or operational semantics is the more conventional, and describes as usual the sequence of states passed through when executing a program. In addition a PROLOG program can be understood as a set of descritive statements (one for each clause) about a problem. The declarative or denotational semantics, which PROLOG inherits from logic, provides a formal basis for such an understanding. Informally, one interprets terms as shorthand for natural language phrases by applying a uniform translation of each functor, e.g.,

void = "the empty tree"

t(L,N,R) = "the binary tree with root N, left subtree L and right subtree R"

A clause "E :- Q,R,S." where E,Q,R and S are metavariables standing for terms, is interpreted as

"E if Q and R and S"

A clause "E." is interpreted as "E is true".

Each variable in a clause should be interpreted as some arbitrary object (i.e. variables are universally quantified). The type of the object conveyed by a variable will be appropriate to the functor(s) where the variable figures by using terms in a consistent way throughout the program.

The declarative semantics simply defines (recursively) the set of terms which are asserted to be true according to a program. A term is true if it is the head of some clause instance and each of the goals (if any) of that clause instance is true, where an instance of a clause (or term) is obtained by substituting, for each of zero or more of its variables, some term for all occurences of the variable.

Thus the only instance of the goal:

```
concatenate([a],[b],L).
```

is

```
concatenate([a],[b],[a,b]).
```

It is the declarative aspect of PROLOG which is responsible for promoting clear, rapid, accurate programming. It allows a program to be broken down into small, independently meaningful units (clauses), and it allows some understanding of a program without looking into the details of how it is executed.

2.3 PROCEDURAL SEMANTICS

It is the procedural semantics that describes the way a goal is executed. The objective of execution is to produce true instances of the goal. It then becomes important to know that the ordering of clauses in a program, and of goals within a clause, which are irrelevant as far as the declarative semantics is concerned, constitute crucial control information for the procedural semantics.

To execute a goal, the system searches for the first clause whose head matches or unifies with the goal. The unification process (Robinson,1965) finds the most general common instance of the two terms, which is unique if it exists. If a match is found, the matching clause instance is then activated by executing in turn, from left to right, each of the goals of its body (if any). If at any time the system fails to find a match for a goal it backtracks, i.e. it rejects the most recently activated clause, undoing any substitutions made by the match with the head of the clause. Next it reconsiders the original goal which activated the rejected clause, and tries to find a subsequent clause which also matches the goal. Execution terminates if no goals remain to be executed (the system has then found a true instance of the original goal). Backtracking may then be invoked to find other true instances of the goal. Execution fails when no true instances of the original goal are found, and terminates if it cannot find any more true instances. Termination however cannot be guaranteed, even if there are no

more true instances (e.g. if there are infinite branches).

Note that the execution just defined is a left to right depth-first process. Note also that because unification always provides the most general common instance between a goal and a matching clause, all the most general true instances of a goal can potentially be found (i.e. apart from termination issues).

Basically, each execution step is justified by Robinson's Resolution Principle (Robinson,1965). This principle subsumes in a single inference rule the classical rules of "modus ponens" and "generalization" in formulations of first order predicate calculus. For example, from

```
p(X):-q(a,X),r(X).
```

and

```
q(Y,f(Y,Z)):-s(a,Z).
```

it allows to conclude

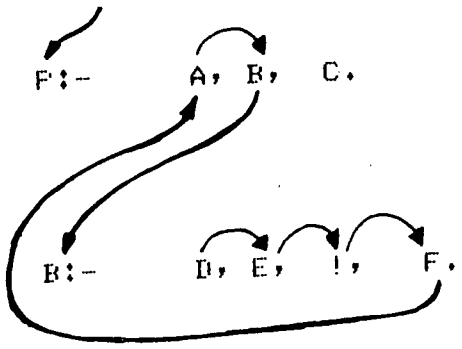
```
p(f(a,Z)):-s(a,Z),r(f(a,Z)).
```

by "execution" of q(a,X).

Besides the ordering of clauses and the sequencing of goals within clauses PROLOG provides just one other essential mechanism for specifying control information. This is the "cut" symbol, written "!". It is inserted in a program just like a goal, but it is not to be regarded as part of the logic of the program and should be ignored as far as the declarative semantics is concerned.

The effect of the "cut" is as follows: when first encountered, as a goal, "cut" succeeds immediately. If backtracking should later return to the "cut", the effect is to fail the goal which caused the clause containing the "cut" to be activated. In other words, the "cut" operation commits the system to all choices made since execution of the goal activating the clause begun, i.e. other alternatives for that goal are not considered, as well as for all goals occurring in the matching clause before the "cut". By means of a "cut" one can ensure that some goals, once partly executed by a clause up to a "cut", either must continue that partial execution or fail. The "cut" renders deterministic the whole partial execution made by the activated clause up to it.

Example of the effect of a "cut" in the flow of control, when goal F fails,



where A, B, C, D, E, F, and P are metavariables standing for predicate instances.

IF F fails, backtracking returns to goal A immediately before B, the goal that activated the clause with the "cut".

2.4 OUTSTANDING FEATURES OF PROLOG

Let us briefly review the combination of features which make PROLOG a powerful but simple to use programming language.

(1) A declarative semantics inherited from logic in addition to the usual procedural semantics.

(2) Identity of form of program and data - clauses can be employed for expressing data, and can be manipulated as terms by interpreters written in Prolog.

(3) The input and output arguments of a procedure do not have to be distinguished in advance, but may vary from one call to another. Procedures can be multi-purpose.

(4) Procedures may have multiple outputs as well as multiple inputs.

(5) Procedures may generate, through backtracking, a sequence of alternative results. This amounts to a high level form of iteration.

(6) Terms provide general record structures with any number of fields. An unlimited number of record types may be used, and there are no type restrictions on the fields of a record.

(7) Pattern matching replaces the use of selector and constructor functions for operating on structured data.

(8) Incomplete data structures may be returned (i.e. structures containing free variables) which may later be filled in by other procedures.

(9) Prolog dispenses with go to, do for and while loops, assignment, and references (pointers).

(10) The procedural semantics of a syntactically correct program is totally defined. It is impossible for an error condition to arise or for an undefined operation to be performed. This totally defined semantics ensures that programming errors do not result in bizarre program behaviour or incomprehensible error messages.

(11) No part of the program is concerned with the details of the underlying machine or implementation.

CHAPTER 3

PROBLEMS AND EXERCISES : LOGIC PROGRAMS

3.1 UTILITIES (nos. 1 to 41)

PROBLEM 1

Verbal statement:

Check whether H is a member of a list L; search all members of a list L.

Logic program:

```
member(H,[H|_]).
member(I,[_|T]):- member(I,T).
```

PROBLEM 2

Verbal statement:

Check whether H is a member of a list L; search the first member of a list L. Consider L as [a,b,c].

Logic program:

```
member(H,[H|_]):- !.
member(I,[_|T]):- member(I,T).
```

Execution:

The only result produced by executing:

```
?-member(X,[a,b,c]).
```

is X=a. The other two potential solutions are discarded on account of the cut symbol.

PROBLEM 3

Verbal statement:

Check whether U is the intersection of lists L1 and L2 ; search the intersection of lists L1 and L2.

Logic program:

```
intersect([H|T],L,[H|U]):- member(H,L),intersect(T,L,U).
intersect([_|T],L,U):- intersect(T,L,U).
intersect(_,_,[]).

member(H,[H|_]).
member(I,[_|T]):- member(I,T).
```

PROBLEM 4

Verbal statement:

Write the 'pickup the first n members of a list' relation.

Logic program:

```
set_till_n([],[],_,_).
set_till_n([X|L],[X|P],C,N):- (var(C),C=1;true),
                              C=<N,C1 is C+1,
                              set_till_n(L,P,C1,N).

set_till_n(_,[],_,_).
```

PROBLEM 5

Verbal statement:

Write the set equality relation.

Logic program:

```
set_equal(X,X):- !.
set_equal(X,Y):- equal_lists(X,Y).

equal_lists([],[]).
equal_lists([X|L1],[Y|L2]):- delete(X,L2,L3),
                             equal_lists(L1,L3).

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

PROBLEM 6

Verbal statement:

Write the subtraction relation of two lists.

Logic program:

```
subtract(L,[],L):- !.
subtract([H|T],L,U):- member(H,L),!,subtract(T,L,U).
subtract([H|T],L,[H|U]):- !,subtract(T,L,U).
subtract(_,_,[]).

member(H,[H|_]).
member(I,[_|T]):- member(I,T).
```

PROBLEM 7

Verbal statement:

Specify the relation append (list concatenation) between three lists which holds if the last one is the result of appending the first two. Consider the last list as [a,b].

Logic program:

```
append([],L,L).
append([H|T],L,[H|U]):- append(T,L,U).
```

Execution:

For the example proposed, if we execute the goal expressed by the question:

```
?-append(X,Y,[a,b]).
```

the first solution is: Y=[a,b] and X=[]

If this solution is rejected by the user, backtracking will generate the further solutions:

```
X=[a]                    Y=[]
Y=[b]                    X=[a,b]
```

PROBLEM 8

Verbal statement:

Check whether T is a sublist of U; search all incompletely specified sublists (patterns).

Logic program:

```
sublist(T,U):- append(H,T,V),append(V,W,U).

append([],L,L).
append([H|T],L,[H|U]):- append(T,L,U).
```

PROBLEM 9 [MELONI,1976]

Verbal statement:

This problem is a sophisticated application of list concatenation. It is pretended to build, on basis of a list of French words (for instance, animal names), a list of new words, each one obtained from the ancestors by juxtaposition of the words whose final characters are the first ones of another. For example the words VACHE (cow) and CHEVAL (horse) give the word VACHEVAL (mutation problem).

Logic program

```
begin:- mutation(X),name(Nn,X),write(Nn),nl,fail.
begin:- nl,write('Done.'),nl.

mutation(X):- animal(Y),name(Y,Ny),animal(Z),name(Z,Nz),
              append(Y1,Y2,Ny),Y1==[],append(Y2,Z2,Nz),
              Y2==[],append(Y1,Nz,X).

append([],X,X).
append([A|X],Y,[A|Z]):- append(X,Y,Z).

animal(alligator).          /* crocodile */
animal(tortue).             /* turtle    */
animal(caribou).            /* caribou   */
animal(ours).               /* bear      */
animal(cheval).             /* horse     */
animal(vache).              /* cow       */
animal(lapin).              /* rabbit    */
```

Execution:

```
:-begin

alligatortue
caribours
chevalligator
chevalapin
vacheval

Done.
```

PROBLEM 10

Verbal statement:

Check whether L1 is the reverse of list L; search the reverse of list L.

Logic programs:

```
/* program 1 */

reverse([],[]).
reverse([H|T],L):- reverse(T,R),append(R,[H],L).

append([],L,L).
append([H|T],L,[H|U]):- append(T,L,U).


/* program 2 */

reverse2(L1,L):- reverse_append(L1,[],L).

reverse_append([H|T],L,M):- reverse_append(T,[H|L],M).
reverse_append([],L,L).
```

PROBLEM 11

Verbal statement:

Write a program where the main operations on sets are defined.

Logic program:

```
/* member of a set */

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

/* subset */

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

/* disjoint */

disjoint(X,Y):- member(Z,X),not(member(Z,Y)).

/* intersection */

intersection([],X,[]).
intersection([X|R],Y,[X|Z]):- member(X,Y),!,
                              intersection(R,Y,Z).
```

```
intersection([X|R],Y,Z):- intersection(R,Y,Z).

/* union */

union([],X,X).
union([X|R],Y,Z):- member(X,Y),!,union(R,Y,Z).
union([X|R],Y,[X|Z]):- union(R,Y,Z).
```

PROBLEM 12 [EMDEN, personal communication]

Verbal statement:

Split a list with head H and tail [H1|T1] into two lists U1 and U2, where all the elements of U1 are = <H, all the elements of U2 are >H, and the orisinal order is preserved.

Logic program:

```
split(H,[H1|T1],[H1|U1],U2):- H1=<H,split(H,T1,U1,U2).
split(H,[H1|T1],U1,[H1|U2]):- H1>H,split(H,T1,U1,U2).
split(_,[],[],[]).
```

PROBLEM 13 [EMDEN, personal communication]

Verbal statement:

Define a quicksort relation using append.

Logic program:

```
sort([H|T],S):- split(H,T,U1,U2),
                sort(U1,V1),
                sort(U2,V2),
                append(V1,[H|V2],S).

sort([],[]).

append([],L,L).
append([H|T],L,[H|U]):- append(T,L,U).
```

Execution:

```
:-sort([C,Q,W,E,R,T,Y,U,I,O,P,A,S,D,F,G,H,J,K,L,Z,X,C,
        V,B,N,M],S), write(S).

[A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z]
```

PROBLEM 14 [EMDEN, personal communication]

Verbal statement:

Define the quicksort relation without append (make X=[] in the input).

Logic program:

```
sort2([H|T],S,X):- split(H,T,U1,U2),
                   sort2(U1,S,[H|Y]),
                   sort2(U2,Y,X).
sort2([],X,X).

split(H,[H1|T1],[H1|U1],U2):- H1<H,split(H,T1,U1,U2).
split(H,[H1|T1],U1,[H1|U2]):- H1>H,split(H,T1,U1,U2).
split(_,[],[],[]).
```

PROBLEM 15 [EMDEN, personal communication]

Verbal statement:

Define the bubble sort relation.

Logic program:

```
sort3(L,S):- append(U,[A,B|V],L),
             B<A,
             append(U,[B,A|V],M),
             sort3(M,S).
sort3(L,L).

append([],L,L).
append([H|T],L,[H|V]):- append(T,L,U).
```

PROBLEM 16 [EMDEN, personal communication]

Verbal statement:

Write a program to be compiled which defines the quicksort relation.

Apply your program to the list of 50 elements : 27, 74, 17, 33, 94, 18, 46, 83, 65, 2, 32, 53, 28, 85, 99, 47, 28, 82, 6, 11, 55, 29, 39, 81, 90, 37, 10, 0, 66, 51, 7, 21, 85, 27, 31, 63, 75, 4, 95, 99, 11, 28, 61, 74, 18, 92, 40, 53, 59, 8.

Logic program:

```
:- mode qsort(+,-,+).
:- mode partition(+,+,-,-).

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([],_,[],[]).

list_50([27,74,17,33,94,18,46,83,65,2,
         32,53,28,85,99,47,28,82,6,11,
         55,29,39,81,90,37,10,0,66,51,
         7,21,85,27,31,63,75,4,95,99,
         11,28,61,74,18,92,40,53,59,8]).
```

Execution:

```
:-list_50(L),qsort(L,X,[]).
```

PROBLEM 17 [EMDEN, personal communication]

Verbal statement:

Define the insert sort relation.

Logic program:

```
sort4([H|T],S):- sort4(T,L),insert(H,L,S).
sort4([],[]).

insert(X,[H|T],[H|L]):- H<X,!,insert(X,T,L).
insert(X,L,[X|L]).
```

PROBLEM 18

Verbal statement:

Write a program for sorting a list as an ordered permutation of that list.

Logic program:

```
sort(L,S):- permute(L,S),ordered(S).

permute([],[]).
```

```
permute(L,[H|S]):- delete(H,L,N2),permute(NL,S).

delete(H,[H|L],L).
delete(X,[H|L],[H|NL]):- delete(X,L,NL).

ordered([]).
ordere([X]).
ordered([X,Y|Z]):- X =< Y, ordered([Y|Z]).
```

PROBLEM 19 [EMDEN,Personal communication]

Verbal statement:

Define combinations of a list, where s(K) indicates the successor of K. For example, 3 is written s(s(s(0))).

Logic program:

```
combinations(s(K),[H|T],[H|U]):- combinations(K,T,U).
combinations(s(K),[_|T],U):- combinations(s(K),T,U).
combinations(0,_,[]).
```

PROBLEM 20 [EMDEN,Personal communication]

Verbal statement:

Define permutations of a list.

Logic program:

```
permutations(L,[H|T]):- append(V,[H|U],L),
                        append(V,U,W),
                        permutations(W,T).
permutations([],[]).

append([],L,L).
append([H|T],L,[H|U]):- append(T,L,U).
```

PROBLEM 21

Verbal statement:

Check whether a word is a palindrome, i.e., the same if read backwards, as for example "MADAM". Write a program able to check a palindrome for each word you give.

Logic program:

```
begin(X):- read(X),(X=stop ; test_palindrome(X),begin(Y)).

test_palindrome(X):- name(X,Nx),palindrome(Nn),write(X),
                     write(' is a palindrome'),nl,!.
test_palindrome(X):- write(X),
                     write(' is not a palindrome'),nl.

palindrome(X):- reverse2(X,X).

reverse2(L1,L):- reverse_append(L1,[],L).

reverse_append([H|T],L,M):- reverse_append(T,[H|L],M).
reverse_append([],L,L).
```

Execution:

```
:-begin(X).

madam.   john.   astyuytsa.   horse.   bull.   stop.

madam is a palindrome
john is not a palindrome
astyuytsa is a palindrome
horse is not a palindrome
bull is not a palindrome
```

PROBLEM 22

Verbal statement:

Given two lists, where the second one is the concatenation of a third one with the first one, find this third list and build up a palindrome with it.

Logic program:

```
palindrome(L1,L2):- list(L1,Y),list(L2,Z),
                    concatenate(X,Y,Z),
                    write('Third list:  '),write(X),nl,
                    reverse(X,W),
                    concatenate(X,W,K),
                    write('Palindrome:  '),write(K).

concatenate([],L,L).
concatenate([E|R],L2,[E|L]):- concatenate(R,L2,L).

reverse([],[]).
reverse([E|R],L):- reverse(R,R1),
                   concatenate(R1,[E],L).
```

```
list(a,[1,3,5,7,9]).
list(b,[a,b,c,d,e,1,3,5,7,9]).
```

Execution:

The execution of :-palindrome(a,b).

gives us the followins result:

```
Third list:  [a,b,c,d,e]
Palindrome:  [a,b,c,d,e,e,d,c,b,a]
```

PROBLEM 23 [KOWALSKI,1974b]

Verbal statement:

Write a program for implementing the relation 'admissible' between two lists a and b,

bi=2ai and a(i+1)=3bi for i<n

where ai and bi are the elements number i of lists a and b respectively.

Logic programs:

```
/* program 1 */

admissible(X,Y):- double(X,Y),triple(X,Y).

double([],[]).
double([X|Y],[U|V]):- U is 2*X, double(Y,V).

triple([X],[U]).
triple([X,Y|Z],[U|V]):- Y is 3*U,triple([Y|Z],V).


/* program 2 */

admissible([],[]).
admissible([X,Y|Z],[U|V]):- U is 2*X,Y is 3*U,
                            admissible([Y|Z],V).
admissible([X],[U]):- U is 2*X.
```

Execution:

The execution of

```
?-admissible([1|U],V),write([1|U]),write(','),write(V),fail.
```

gives us the followins list pairs:

```
[1],[2]
[1,6],[2,12]
[1,6,36],[2,12,72]
      .
      .
      .
```

PROBLEM 24

Verbal statement:

Write a program to simplify a list whose members are lists, and to transform it into a list of single members. Note that each member is a node of a tree and that simplification means to extract all redundancies, even those brought up by domination of a node over several nodes. In this case, the dominant node is redundant.

Logic program:

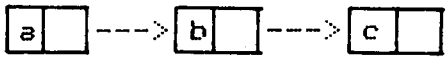
```
simplify(L,NL):- compact(L,L1),simplify1(l1,L2),
                 simplify2(L2,NL).

compact([L,[]],L).
compact([L],L).
compact([L1,[L2]],L):- concatenate(L1,L2,L).
compact(L1,L2|LN],L):- concatenate(L1,L2,X),
                       compact([X,LN],L).

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

simplify1([X|L],NL):- member(X,L),!,simplify1(L,NL).
simplify1([X|L],[X|NL]):- simplify1(L,NL).
simplify1([],[]).

simplify2(L,NL):- simplify3(L,L1),subtract(L,L1,NL).

simplify3([X|L],NL):- compare(L,X,[]),simplify3(L,NL).
simplify3([X|L],[Y|NL]):- compare(L,X,Y),simplify3(L,NL).
simplify3([],[]).

compare(_,[],[]):- !.
compare([Y|L],X,[Z|NL]):- (linked(X,Y),Z=X ; linked(Y,X),Z=Y),
                          compare(L,X,NL).

compare([Y|L],X,NL):- compare(L,X,NL).
compare([],_,[]).

subtract(L,[],L):- !.
subtract([H|T],L,U):- member(H,L),!,subtract(T,L,U).
subtract([H|T],L,[H|U]):- !,subtract(T,L,U).
subtract(_,_,[]).
```

```
member(A,[A|_]).
member(A,[_|L]):- member(A,L).
```

PROBLEM 25 [TÄRNLUND,1976b]

Verbal statement:

Define a procedure for manipulating stacks. Consider a stack such as,



Write the goal statements for deleting an element from the stack, and for inserting a new element e.

Logic program:

```
stack(s(X,Y),X,Y).
```

Execution:

This program pops-up a stack.
The goal statement for deleting is:

```
:-stack(s(a,s(b,s(c,0))),X,Y).
```

The result is X=a and Y=s(b,s(c,0))

The goal statement for inserting is:

```
:-stack(Z,e,s(a,s(b,s(c,0)))).
```

The result is: Z=s(e,s(a,s(b,s(c,0))))

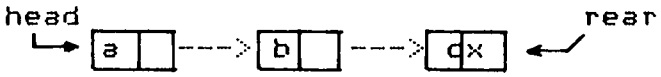
The goal statement for inserting is obtained by dual programming, and it is achieved by shifting the roles between input and output variables in a goal statement.

PROBLEM 26 [TÄRNLUND,1976b]

Verbal statement:

Define a procedure for manipulating queues.

Consider the following queue:



Write the goal statements for deleting the head element of a queue, for inserting an element in the rear of a queue, and for inserting a new element into the queue.

Logic program:

```
delete(q(X,Y),X,Y).

insert(Y,q(W,X1),W,Y).
insert(Y,q(W,X1),W,X1,Y).
```

Execution:

The goal statement for deleting the first element of queue q(a,q(b,q(c,X))) is:

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

The result is: Y=q(b,q(c,Z))

The goal statement for inserting an element d in the rear of the queue is:

```
:-insert(q(a,q(b,q(c,X))),X,d,Z).
```

The result is: Z=q(a,q(b,q(c,q(d,X1))))

The goal statement for inserting a new element into the queue is:

```
:-insert(q(a,q(b,q(c,X))),X,d,X1,Z).
```

PROBLEM 27 [TÄRNLUND,1976b]

Verbal statement:

A fundamental data base operation is to search for a record Ri (node) in a tree given a key Ki, and insert it as a new record if the record is not in the tree. Define an algorithm for binary tree search.

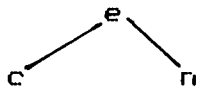
Logic program:

```
binary(t(X,K,Z),K,t(X,K,Z)).
binary(t(X,Y,Z),K,t(X1,Y,Z)):- name(K,NK), name(Y,NY), NK<NY,
                               binary(X,K,X1).
binary(t(X,Y,Z),K,t(X,Y,Z1)):- name(K,NK), name(Y,NY), NK>NY,
                                binary(Z,K,Z1).

binary(void,K,t(void,K,void)).
```

PROBLEM 28 [TÄRNLUND,1976b]

Verbal statement:

Suppose we have the following binary tree:



Construct an algorithm able to search for a key m, and able to insert it if it is not in the tree.

Execution:

the goal statement,

```
:-binary(t(t(void,c,void),e,t(void,n,void)),m,Z).
```

consists in the call of the previous program. The result is a new tree Z,

```
t(t(void,c,void),e,t(t(void,m,void),n,void))
```

PROBLEM 29 [TÄRNLUND,1976b]

Verbal statement:

A frequent data base operation is to delete a node in a tree. Define an algorithm for binary tree deletion.

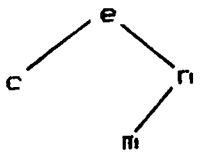
Logic program:

```
delete(t(X,K,void),K,X).
delete(t(X,K,t(void,Y1,Z1)),K,t(X,Y1,Z1)).
delete(t(X,K,t(X1,Y1,Z1)),K,t(X,Y,t(X2,Y1,Z1))):-
                              successor(X1,Y,X2).
delete(t(X,Y,Z)K,t(X1,Y,Z)):- K<Y,delete(X,K,X1).
delete(t(X,Y,Z),K,t(X,Y,Z1)):- K>Y,delete(Z,K,Z1).
delete(void,K,void).

successor(t(void,Y,Z),Y,Z).
successor(t(X,Y,Z),Y1,t(X1,Y,Z)):- successor(X,Y1,X1).
```

PROBLEM 30 [TÄRNLUND,1976b]

Verbal statement:

Consider the tree,



and delete node m.

Execution:

the goal statement,

```
:-binary(Z,m,t(t(void,c,void),e,t(t(void,m,void),n,void))).
```

consists in the call of the previous program 'binary'. However, the dual programming technique, shift the role of the input and output arguments in the goal statement, does not give a complete solution to the deletion problem, as the program 'delete'.

PROBLEM 31 [TÄRNLUND,1976b]

Verbal statement:

Write an algorithm for inserting a new node at its right position in a three-dimensional tree:

t(X,r(U,U1,U2),Z)

where X and Z are left and right subtrees respectively and r(U,U1,U2) is a root having the attributes U,U1 and U2.

Logic program:

```
insert(t(X,r(U,U1,U2),Z)K,K1,K2,t(X1,r(U,U1,U2),Z):-
                                        K<U,insert(X,K1,K2,K,X1).
insert(t(X,r(U,U1,U2),Z),K,K1,K2,t(X,r(U,U1,U2),Z1):-
                                        K>U,insert(Z,K1,K2,K,Z1).
insert(void,K,K1,K2,t(void,r(K,K1,K2),void).
insert(t(X,r(K,K1,K2),Z),K,K1,K2,t(X,r(K,K1,K2),Z)).
```

PROBLEM 32

Verbal statement:

Get all nodes of a tree above and below a specified node.

Logic program:

```
set_node(X,[M|R],above):- node(M,X),set_node(M,R,_).
set_node(_,[],above).
```

```
set_node(X,L,below):- (down_nodes(X);
                       recorded(c(L),P),erase(P)).

down_nodes(X):- record(c([])),node(X,M),
                replace(c(L),c([M|L])),fail.

replace(X,Y):- recorded(X,P),!,erase(P),record(Y).
replace(_,Y):- record(Y).
```

PROBLEM 33

Verbal statement:

Verify if two nodes of a tree are connected.

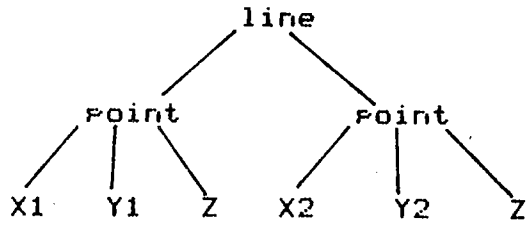
Logic program:

```
set_branched(X,Y):- connected(X,Y);connected(Y,X).

connected(X,Y):- node(X,Y).
connected(X,Z):- node(X,Y),connected(Y,Z).
```

PROBLEM 34

Verbal statement:

Write a term for representing the following structured data object:



Logic program:

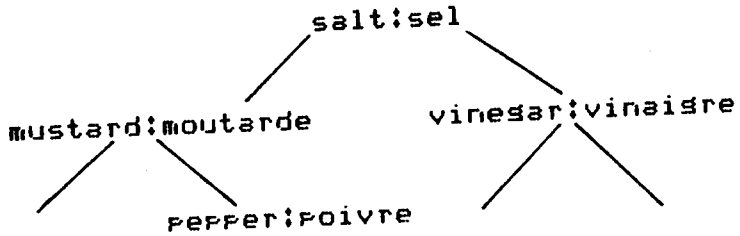
```
line(point(X1,Y1,Z),point(X2,Y2,Z)).
```

PROBLEM 35 [WARREN,1977b]

Verbal statement:

Write a term 'dictionary' for representing an alphabetically ordered dictionary pairing English words with French equivalents.

Consider the following example:



Write also procedures for:

- "look up" a name in a dictionary and find it paired value
- "look up" a name which is before a certain name
- "look up" a name which is after a certain name

and take into account that those procedures may also serve for constructing the dictionary.

Logic program:

```
dictionary(void).
dictionary(dic(X,Y,D1,D2)):- dictionary(D1),
                             dictionary(D2).


dic(salt,sel,
    dic(mustard,moutarde,
        void,
        dic(pepper,poivre,void,void)),
    dic(vinegar,vinaigre,void,void)).

lookup(Name,dic(Name,Value,_,_),Value):- !.
lookup(Name,dic(Name1,_,Before,_),Value):- Name<Name1,
                                lookup(Name,Before,Value).

lookup(Name,dic(Name1,_,_,After),Value):-Name>Name1,
                                lookup(Name,After,Value).
```

Execution:

```
:-lookup(salt,D,X1).
```

is interpreted as : construct a dictionary D such that "salt" is paired with X1.

The execution gives:

```
D=dic(salt,X1,D1,D2)
```

```
:-lookup(mustard,D,X2).
```

gives as result:

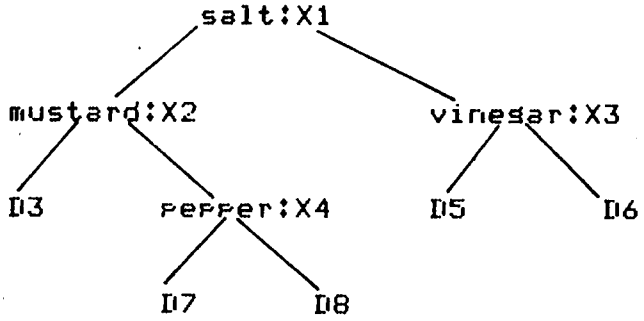
```
D=dic(salt,X1,dic(mustard,X,D,D4),D2).
```

In this way 'lookup' is inserting new entries in a partially specified dictionary.

After executing:

```
:-lookup(vinegar,D,X3),lookup(pepper,D,X4),lookup(salt,D,X5).
```

D is instantiated to a dictionary which may be pictured as:



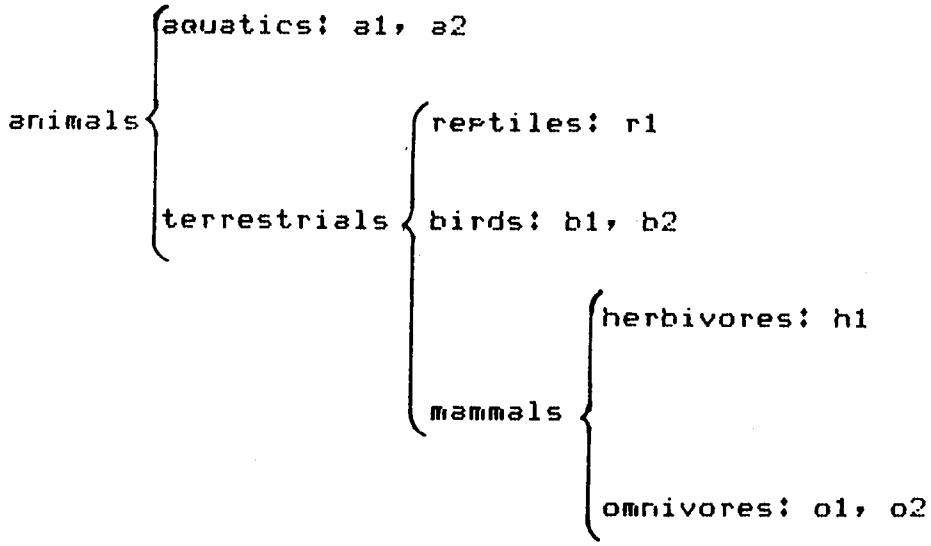
PROBLEM 36

Verbal statement:

Suppose you have an universe of eight animals

U=[a1,a2,r1,b1,b2,h1,o1,o2]

classified in the following way:



Write a data structure representing this classification, and build up a program for answering the following questions:

1) What is the classification of animal X ?

2) What animals have the classification Y ?

Logic program:

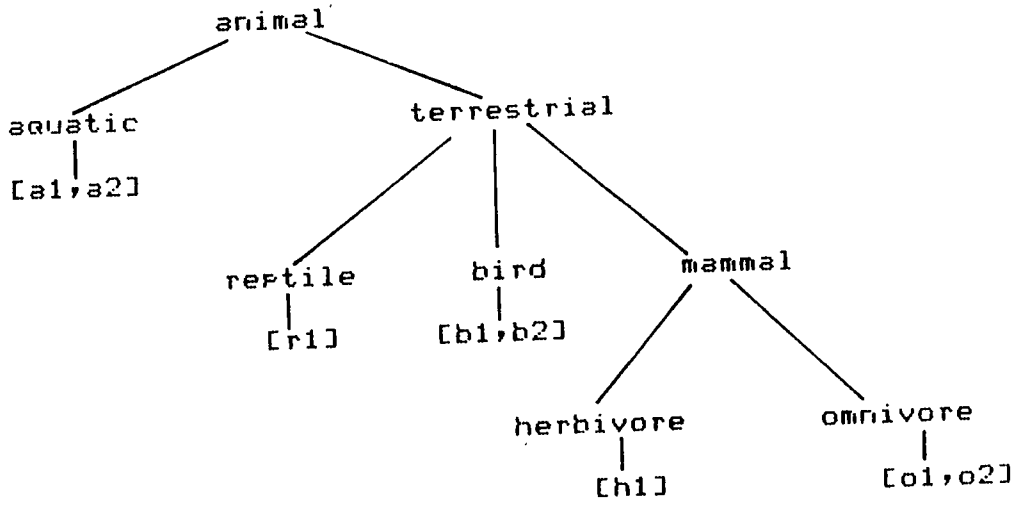
```
classification(A,C):- animal(C,L),member(A,L).

animal(animal(V),X):- aquatic(V,X) ; terrestrial(V,X).

aquatic(aquatic,[a1,a2]).

terrestrial(terrestrial(V),X):- reptile(V,X) ; bird(V,X) ;
                                mammal(V,X).

reptile(reptile,[r1]).

bird(bird,[b1,b2]).

mammal(mammal(V),X):- herbivore(V,X) ; omnivore(V,X).

herbivore(herbivore,[h1]).

omnivore(omnivore,[o1,o2]).
```

Remark: The predicate "member(X,Y)" is defined as in PROBLEM 1, and it tests if X is a member of list Y.

The question "1) What is the classification of animal X ?" where X is instantiated, corresponds to the command ":-classification(X,C).". The other question "2) What animals have the classification Y ?", where Y is instantiated, corresponds to the command ":-animal(Y,X).".

Execution:

The execution is top-down, as represented on the tree:



Remark: This problem was suggested by [Dahl,1977b].

PROBLEM 37 [MELONI,1976]

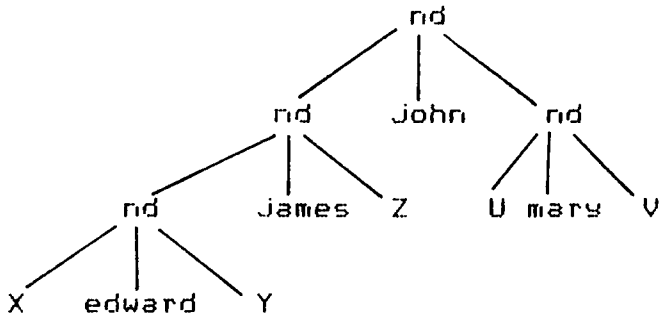
Verbal statement:

Write a program able to sort a list of words by alphabetic order.

The list of words obeys the syntax: word1.word2. ... .wordn.stop. , where 'stop' means that there are no more words to sort.

Sugestion: Imagine a tree having each node as a function of 3 arguments nd(L,W,R), where L represents the left descendent of the node, R, its right descendent and W, the word just read. The input word is read and compared to the word associated to the node, beginning by the root, and afterwards compared recursively to its left or right descendent. The whole process finishes when the descendent is a free variable or when the word already exists in the tree. In the first case, a new node representing the word is created, in the place of the variable.

Example: The list john. mary. james. edward. stop. will generate the tree:



Logic program:

```
order(X):- reading(W),name(W,W1),classify(W1,X,Y),order(Y).
order(X):- nl,write('ordered words:'),nl,write_tree(X),nl,nl,
           write('Done.'),nl.

reading(W):- read(W),(W==stop,!,fail;true).

classify(W,nd(L,W,R),nd(L,W,R)):- !.
classify(W,nd(L,W1,R),nd(U,W1,R)):- lower(W,W1),!,
                                    classify(W,L,U).
classify(W,nd(L,W1,R),nd(L,W1,U)):- !,classify(W,R,U).

lower([],_):- !.
```

```
lower([X|Y],[X|T]):- !,lower(Y,T).
lower([X|Y],[Z|T]):- X<Z.

write_tree(X):- var(X),!.
write_tree(nd(L,W,R)):- !,write_tree(L),name(W1,W),write(W1),
                        write('.  '),write_tree(R).
```

Execution:

To run the program just give the command ":-order(X)." followed by the list of words.

```
:-order(X).

helder. antonio. luis. jose. fernando. jorge. carlos. stop.

ordered words:

antonio. carlos. fernando. helder. jorge. jose. luis.
Done.
```

PROBLEM 38

Verbal statement:

Write a program that builds up a tree on a given list of numbers, and then outputs the list in an ordered way.

The way of building the tree is:

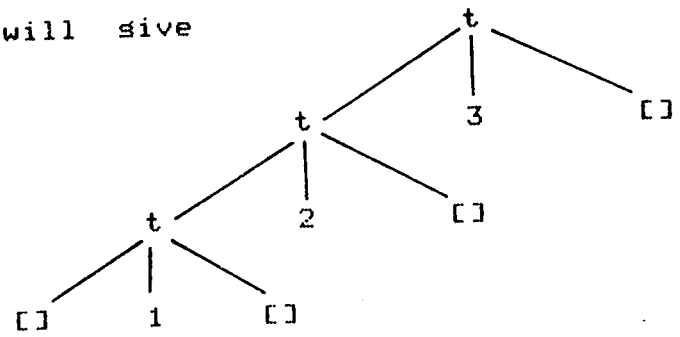
1) The root is the first member of the list.

2) The left branch will consist of the less or equal nodes, and the right branch will consist of the greater or equal ones.

3) Each node is represented by t(L,N,R) where N is the node, L is the left subtree and R is the right subtree.

For example:

[3,2,1] will give



Apply your program to the list:

```
[14,12,16,21,19,9,0,15,13,6,3,2,5,1,4,7,11,18,17].
```

Logic program:

```
goal(X):- make_tree(X,T),nl,write(T),nl,nl,liste_tree(T,L,[]),
          write(L),nl,nl.

make_tree([H|T],t(T1,H,T2)):- split(T,H,U1,U2),
                              make_tree(U1,T1),
                              make_tree(U2,T2).
make_tree([E],t([],E,[])).
make_tree([],[]).

list_tree(t(T1,H,T2),S,X):- list_tree(T1,S,[H|Y]),
                            list_tree(T2,Y,X).
list_tree([],X,X).

split([A|L],C,[A|U1],U2):- A=<C,!,split(L,C,U1,U2).
split([A|L],C,U1,[A|U2]):- A>C,!,split(L,C,U1,U2).
split([],_,[],[]).
```

Execution:

```
  :-goal([14,12,16,21,19,9,0,15,13,6,3,2,1,4,7,11,18,17]).

t(t(t(t([],0,t(t(t(t([],1,[]),2,[]),3,t([],4,[])),6,t([],7,[]))),
  9,t([],11,[])),12,t([],13,[])),14,t(t([],15,[]),16,t(t(t(t([],
  17,[]),18,[],19,[]),21,[])))

[0,1,2,3,4,6,7,9,11,12,13,14,15,16,17,18,19,21]
```

PROBLEM 39 [WARREN et al, 1977]

Verbal statement:

Generate a list of serial numbers for the items of a given list, the members of which are to be numbered in alphabetical order.

Logic program:

```
serialise(L,R):- pairlists(L,R,A),arrange(A,T),
                 numbered(T,1,N).

pairlists([X|L],[Y|R],[pair(X,Y)|A]):- pairlist(L,R,A).
pairlists([],[],[]).

arrange([X|L],tree(T1,X,T2)):- partition(L,X,L1,L2),
                               arrange(L1,T1),
                               arrange(L2,T2).
arrange([],_).
```

```
partition([X|L],X,L1,L2):- partition(L,X,L1,L2).
partition([X|L],Y,[X|L1],L2):- before(X,Y),
                               partition(L,Y,L1,L2).
partition([X|L],Y,L1,[X|L2]):- before(Y,X),
                               partition(L,Y,L1,L2).

partition([],_,[],[]).

before(pair(X1,Y1),pair(X2,Y2)):- X1<X2.

numbered(tree(T1,pair(X,N1),T2),N0,N):- numbered(T1,N0,N1),
                                        N2 is N1+1,
                                        numbered(T2,N2,N).

numbered(void,N,N).
```

PROBLEM 40 [PEREIRA;MONTEIRO,1978]

Verbal statement:

Construct a program for relating a binary tree with the list of its leaves (terminal nodes), and another one for testing whether two binary trees have the same leaves.

Logic program:

```
leaves(t(void,N,void),[N|Z]-Z).
leaves(t(ST1,N,STr),L-Z):- leaves(ST1,L-X),
                           leaves(STr,X-Z).

same_leaves(T1,T2):- leaves(T1,L-[]),
                     leaves(T2,L-[]).
```

Comments:

void='the empty tree'

t(ST1,N,STr)='the binary tree with root N,
left subtree ST1, and right
subtree STr'

The term [N|Z]-Z, where '-' is a binary functor, stands for a 'difference list'. It denotes the list whose sole member is N, obtained from the list [N|Z] 'minus' Z.

The empty difference list is X-X. Difference lists allow for easy concatenation of lists. Because a variable stands in place of the usual 'nil', the list may expand by unifying the variable at the end, which is individuated after the 'minus' sign, with another list. Thus, in the second clause, the (difference) list L-Z is expressed as a concatenation of L-X and X-Z. Since X terminates with Z, L will also terminate with Z. A call to this procedure is

```
leaves(t(t(void,a,void),b,STr),[A,c]-[])
```

Notice that the "[]" in the call has the effect of "closing" the final list obtained.

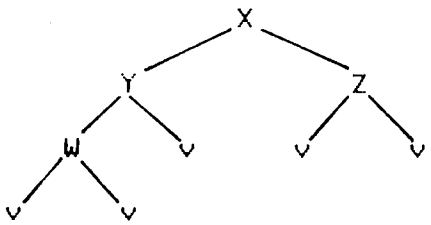
Execution:

Let us now briefly look at how this goal is actually executed. The goal matches only the second clause for "leaves". The body of the matching clause instance is leaves(t(void,a,void),[A,c]-X), leaves(ST1,X-[]). The result of executing the first of these two goals against the only clause which solves it is to instantiate A to a and X to c. The second goal matches the first clause, thereby instantiating ST1 to t(void,c,void), because X is already instantiated to c.

PROBLEM 41

Verbal statement:

Find a subtree ST in a tree T and change it by another subtree CH. Call R the final tree. Apply your program to the tree:



which may be represented by:

```
T=t(T1,X,T2)
 =t(t(T11,Y,T2),X,t(v,Z,v))
 =t(t(t(v,W,v),Y,v),X,t(v,Z,v)) .
```

Logic program:

```
change(CH,ST,ST,CH).
change(CH,ST,t(T1,X,T2),t(R1,X,T2)):- change(CH,ST,T1,R1).
```

Execution:

```
:-change(CH,ST,T,R).
```

where CH stands for the initial tree and R for the result tree.

3.2 DEDUCTIVE REASONING OVER DATA BASES (nos. 42 to 59)

PROBLEM 42 [WARREN et al,1977]

Verbal statement:

Calculate the population density per square mile and find countries of similar population density (differing by less than 5%).

Logic program:

```
density(C,D):- pop(C,P),area(C,A),D is (P*1000)/A.

answer(C1,D1,C2,D2):- density(C1,D1),density(C2,D2),
                      D1>D2, 20*D1<21*D2.

pop(china,825).                 area(china,3380).
pop(india,586).                 area(india,1139).
pop(ussr,252).                  area(ussr,8709).
pop(usa,212).                   area(usa,3609).
```

PROBLEM 43 [EMDEN,1977]

Verbal statement:

Design a deductive information retrieval system concerning some aspects of the operation of a university department, such as those presented under set notation :

Takes		
	M. :Adiri	Math!129
	C.Y.K :Chenk	Math!225
	T.L. :Cook	Math!129
	T.L. :Cook	Math!225

Teaches		
	J.F. :Glotz	Math!129
	J.F. :Glotz	Math!225
	C. :Twill	Math!170

Student			
	M. :Adiri	Biology	1974
	N.A. :Buczek	Recreation	1976
	C.Y.K :Chenk	Physics	1976
	T.L :Cook	Engineering	1975
	G.C :Giusti	Engineering	1976
	A :Hammer	Child Care	1973
	K.L :Mensink	Kinesiology	1973

Scheduled			
	Math!129	Phy@3009	2.30
	Math!301	Phy@3009	2.30

Logic program:

```
:-op(500,xfy,:).
:-op(300,xfy,!).
:-op(300,xfy,@).
:-op(100,xfy,.).

takes(X,Math!129):- year(X,1),program(X,engineering).

year(X,Z):- student(X,Y,Z1),Z is 1977-Z1.

program(X,Y):- student(X,Y,Z).

course_prefix(X!Y,X).

course_number(X!Y,Y).

initials(X:Y,X).

lastname(X:Y,Y).

graduate_course(X):- course_number(X,Y),Y>499.
```

```
conflict(X1,X2):- scheduled(X1,Y,Z),
                  scheduled(X2,Y,Z),
                  X1=\=X2.

teaches(X,Y):- takes(Z,Y).

takes(m:adiri,math!129).
takes(c.y.k:chenk,math!225).
takes(t.l:cook,math!129).
takes(t.l:cook,math!225).

teaches(J.f:slotz,math!129).
teaches(J.f:slotz,math!225).
teaches(c:twill,math!170).

student(m:adiri,biology,1974).
student(n.a:buczek,recreation,1976).
student(c.J.k:chenk,physics,1976).
student(t.l:cook,engineering,1975).
student(s.c:siusti,engineering,1976).
student(a:hammer,child_care,1973).
student(k.l:mensink,kinesiology,1973).

scheduled(math!129,phy@3009,2.30).
scheduled(math!301,phy@3009,2.30).
```

PROBLEM 44 [WARREN,1975a]

Verbal statement:

The number of days in a year is 366 each four years ; otherwis
is 365. How many days have years 1975, 1976 and 2000?

Logic program:

```
:-op(300,fx,'no_of_days_in').
:-op(500,xfy,'IS').

no_of_days_in Y 'IS' 366:- 0 is Y mod 4,
                           not(0 is Y mod 100),!.
no_of_day_in Y 'IS' 365.

:-X 'IS' no_of_days_in Y, write(X).
```

Execution:

```
:-X 'IS' no_of_days_in 1975, write(X).

365

:-X 'IS' no_of_days_in 1976, write(X).

366
```

```
:-X 'IS' no-of-days-in 2000, write(X).

365
```

PROBLEM 45 [WARREN,1975a]

Verbal statement:

John likes Mary. John likes Jane. Mary likes Pete. Pete likes Kate. Jane likes John. Kate likes every one. Who does like someone and is corresponded ?

Logic program:

```
:-op(900,xfy,'&').
:-op(700,xfy,'likes').

john likes mary.
jonh likes jane.
mary likes pete.
pete likes kate.
jane likes john.
kate likes _.

:-X likes Y, Y likes X, write((X&Y)),fail.
```

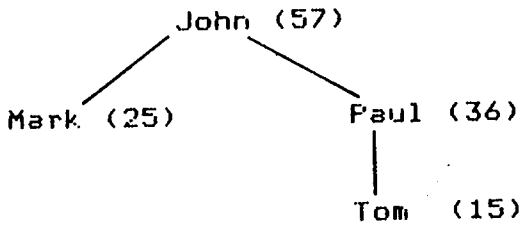
Execution:

```
john & jane
pete & kate
jane & john
kate & pete
kate & kate
?
```

PROBLEM 46 [WARREN,1975a]

Verbal statement:

Suppose we are given the following family tree, with each person's age in brackets:



and we wish to find all solutions to the question "Does John have two descendents whose ages differ by ten years?"

Write a program to solve this problem.

Logic program:

```
has_descendant(X,Z):- begat(X,Z).
has_descendant(X,Z):- has_descendant(X,Y),
                      begat(Y,Z).

begat(john,mark).
begat(john,paul).
begat(paul,tom).

is_aged(john,57).
is_aged(mark,25).
is_aged(paul,36).
is_aged(tom,15).

:-has_descendant(john,X), is_aged(X,N),
  has_descendant(john,Y), is_aged(Y,M), M is N+10.
```

PROBLEM 47 [PEREIRA et al,1978]

Verbal statement:

Construct a small data base obeying to:

"the goal 'descendant(X,Y)' is true if Y is a descendant of X"

Consider that Ishmael and Isaac are descendants of Abraham, and Esau and Jacob are descendants of Isaac.

Logic program:

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

offspring(abraham,ishmael).
offspring(abraham,isaac).
offspring(isaac,esau).
offspring(isaac,jacob).
```

Execution:

If for example the question :

```
?-descendant(abraham,X).
```

is executed, Prolog's backtracking results in different descendants of Abraham being returned as successive instances of the variable X, i.e.,

```
X = ishmael
X = isaac
X = esau
X = Jacob
```

PROBLEM 48

Verbal statement:

There were three departures for a Monte Carlo rally:

London : John (BMW), Tommy (FORD), Fred (BMW), Anne (FORD) and Teddy (BMC)

Paris : Guy (FIAT), Claude (FORD), Jean (FIAT), and Brisitte (BMC)

Lisbon : Luis (FIAT), Carlos (BMW), Lucas (BMC), Pedro (FORD), and Nuno (FIAT).

and only four pilots, Tommy, Fred, Pedro and Nuno, arrived to Monte Carlo.

Which cars did arrive to Monte Carlo?

From where did the pilots that arrived to Monte Carlo leave ?

Logic program:

```
rally:- pilots(A,london),
        pilots(B,lisbon),
        pilots(C,paris),
        pilots(D,bmw),
        pilots(E,fiat),
        pilots(F,ford),
        pilots(G,bmc),
        pilots(H,monte_carlo),
        intersection(A,H,Lon),
        intersection(B,H,Lis),
        intersection(C,H,Par),
        intersection(D,H,Bmw),
        intersection(E,H,Fiat),
        intersection(F,H,Ford),
        intersection(G,H,Bmc),nl,nl
        write('Arrived:'),nl,nl,
        write('From London:  '),write(Lon),nl,
        write('From Lisbon:  '),write(Lis),nl,
        write('From Paris:   '),write(Par),nl,nl,
        write('in BMW:   '),write(Bmw),nl,
        write('in FIAT:  '),write(Fiat),nl,
        write('in FORD:  '),write(Ford),nl,
        write('in BMC:   '),write(Bmc),nl.
```

```
intersection([],X,[]).
intersection([X|R],Y,[X|Z]):- member(X,Y),!
                              intersection(R,Y,Z).
intersection([X|R],Y,Z):- intersection(R,Y,Z).

member(X,[X|_]).
member(I,[_|T]):- member(I,T).

pilots([john,tommy,fred,anne,teddy],london).
pilots([guy,claude,jean,brigitte],paris).
pilots([luis,carlos,lucas,pedro,nuno],lisboa).
pilots([fred,carlos,john],bmw).
pilots([luis,nuno,jean,guy],fiat).
pilots([anne,tommy,claude,pedro],ford).
pilots([teddy,brigitte,lucas],bmc).
pilots([tommy,fred,pedro,nuno],monte_carlo).
```

Execution:

The execution of ?-rally.

gives us the following results:

Arrived:

From London : [tommy,fred]

From Lisbon : [nuno,pedro]

From Paris : []

in BMW : [fred]

in FIAT : [nuno]

in FORD : [pedro,tommy]

in BMC : []

PROBLEM 49

Verbal statement:

Write a program considering the following facts:

- number(X,N) means that person X can be reached by calling phone number N.

- visits(X,Y) means that person X is visiting person Y.

- at(X,Y) means that person X is at the residence of person Y.

- phone(X,N) mean that person X has phone number N.

and the following premises:

1) (∀X) (∀Y) (∀Z) [visits(X,Y) & at(Y,Z) ==> at(X,Z)]

2) (∀U) (∀V) (∀N) [at(U,V) & phone(V,N) ==> number(U,N)]

able to find the way of reaching a person, having a particular phone number base and knowing who visits whom and where is the visited person.

Logic program

```
/* Finding Process */

find(X):- number(X,N),write('phone :  '),write(N),nl.
find(_):- write('Don''t Know.'),nl.

number(X,N):- at(X,Y),phone(Y,N),!.
number(X,N):- phone(X,N).

at(X,Z):- visits(X,Y),at(Y,Z).


/* Phone numbers data base */

phone(coleman,'100001').
phone(gordon,'100002').
phone(wagner,'100003').
phone(smith,'100004').
```

Execution:

Suppose you want to reach Coleman and you know he is visiting with Wagner and that Wagner is at Gordon's house.
You just have to write:

```
visits(coleman,wagner).

at(wagner,gordon).

:-find(coleman).
```

the answer would be:

```
phone :  100002
```

Remark: This problem was suggested from "The Thinking Computer", by B. Raphael.

PROBLEM 50

Verbal statement:

Consider a fact deduction machine which takes a collection of known facts and makes new conclusions, over a domain of animals in a zoo.

Suppose the machine knowledge is composed of 15 recognition productions :

P1
If the animal has hair,
then it is a mammal.

P2
If the animal gives milk
then it is a mammal.

P3
If the animal has feathers
then it is a bird.

P4
If the animal flies,
and it lays eggs,
then it is a bird.

P5
If the animal is a mammal,
and it eats meat,
then it is a carnivore.

P6
If the animal is a mammal,
it has pointed teeth
it has claws,
and its eyes point forward,
then it is a carnivore.

P7
If the animal is a mammal,
and it has hoofs,
then it is an ungulate.

```
P8
If     the animal is a mammal,
and    it chews curd,
then   it is an ungulate,
and    it is even toed.

P9
If     the animal is a carnivore,
       it has a tawny color,
and    it has dark spots,
then   it is a cheetah.

P10
If     the animal is a carnivore,
       it has a tawny color,
and    it has black stripes,
then   it is a tiger.

P11
If     the animal is an ungulate,
       it has long legs and a long neck,
       it has a tawny color,
and    it has dark spots,
then   it is a giraffe.

P12
If     the animal is an ungulate,
       it has a white color,
and    it has black stripes,
then   it is a zebra.

P13
If     the animal is a bird,
       it does not fly,
       it has long legs and a long neck,
and    it is black and white,
then   it is an ostrich.

P14
If     the animal is a bird,
       it does not fly,
       it swims,
and    it is black and white,
then   it is a penguin.
```

P15
If the animal is a bird
and it is a good flyer
then it is an albatross.

Write a program (simulating that machine) able to discover a certain animal by means of asking its characteristics using the recognition productions, and to describe the properties of any animal in its knowledge base.

Logic program:

```
/* Knowledge base */

rule(1,animal,mammal,[c1]).
rule(2,animal,mammal,[c2]).
rule(3,animal,bird,[c3]).
rule(4,animal,bird,[c4,c5]).
rule(5,mammal,carnivore,[c6]).
rule(6,mammal,carnivore,[c7,c8,c9]).
rule(7,mammal,ungulate,[c10]).
rule(8,mammal,ungulate_toed,[c11]).
rule(9,carnivore,cheetah,[c12,c13]).
rule(10,carnivore,tiger,[c12,c14]).
rule(11,ungulate,giraffe,[c15,c16,c12,c13]).
rule(12,ungulate,zebra,[c17,c14]).
rule(13,bird,ostrich,[c18,c15,c16,c19]).
rule(14,bird,penguin,[c18,c20,c19]).
rule(15,bird,albatross,[c21]).


/* Recognition process : discover animal's name */

recognition(X):- rule(N,X,Y,Z),discover(Z),found(rule),
                 conclusion(X,Y,N), recognition(Y),
                 retractall(fact(_,_)).
recognition(_):- retract(rule),write('Done.'),nl.
recognition(_):- write('Don''t know this animal.'),nl.

found(X):- X,!.
found(X):- assert(X).

/* Discovering process */

discover([]).
discover([X|Y]):- ask(X),discover(Y).

ask(X):- fact(X,yes),!.
ask(X):- fact(X,no),!,fail.
ask(c1):- write('has it hair?')nl,!,complete(c1).
ask(c2):- write('does it give milk?'),nl,!,complete(c2).
```

```
ask(c3):- write('has it feathers?'),nl,!,complete(c3).
ask(c4):- write('does it fly?'),nl,!,complete(c4).
ask(c5):- write('does it lay eggs?'),nl,!,complete(c5).
ask(c6):- write('does it eat meat?'),nl,!,complete(c6).
ask(c7):- write('has it pointed teeth?'),nl,!,complete(c7).
ask(c8):- write('has it claws?'),nl,!,complete(c8).
ask(c9):- write('has it eyes pointing forward?'),
          nl,!,complete(c9).
ask(c10):- write('has it hoofs?'),nl,!,complete(c10).
ask(c11):- write('does it chew cud?'),nl,!,complete(c11).
ask(c12):- write('has it a tawny color?'),nl,!,complete(c12).
ask(c13):- write('has it dark spots?'),nl,!,complete(c13).
ask(c14):- write('has it black stripes?'),nl,!,complete(c14).
ask(c15):- write('has it long legs?'),nl,!,complete(c15).
ask(c16):- write('has it a long neck?'),nl,!,complete(c16).
ask(c17):- write('has it a white color?'),nl,!,complete(c17).
ask(c18):- write('does it not fly?'),nl,!,complete(c18).
ask(c19):- write('is it black and white?'),nl,!,complete(c19).
ask(c20):- write('does it swim?'),nl,!,complete(c20).
ask(c21):- write('is it a good flyer?'),nl,!,complete(c21).

complete(X):- read(Y),assert(fact(X,Y)),Y=yes.

/* Conclusion of the recognition process */

conclusion(X,Y,N):- nl,tab(4),write('--- the '),write(X),
                    write(' is a '),write(Y),write(' by rule '),
                    write(N),nl,nl.


/* Description process:  discover animal's properties */

description(X):- rule(N,Y,X,Z),description1(Y,L,[]),
                 conclusion1(X,L,Y,Z,N).
description(_):- nl,write('Don''t know this animal.'),nl.

description1(Y,L,Ls):- rule(_,X,Y,_),description1(X,L,[X|Ls]).
description1(_,L,L).


/* Conclusions of the description process */

conclusion1(X,L,Y,Z,N):- nl,write('a '),write(X),
                         write(' is an '),
                         output(L),write(Y),
                         write('satisfying conditions:  '),nl,
                         output(Z),nl,write('by rule '),write(N),
                         write('.').


output([]).
output([A|B]):- write(A),tab(1),output(B).
```

Execution:

When discovering the animal's name

1) Sucessfull recognition

```
:-recognition(animal).

has it hair?
| yes.

    ---the animal is a mammal by rule 1

does it eat meat?
| yes.

    ---the mammal is a carnivore by rule 5

has it a tawny color?
| yes.
has it dark spots?
| no.
has it black stripes?
| yes.

    ---the carnivore is a tiger by rule 10

Done.
```

2) Unsucessfull recognition

```
:-recognition(animal).

has it hair?
| no.
does it give milk?
| no.
has it feathers?
| no.
does it fly?
| no.
Don't know this animal.
```

When discovering the animal's properties

1) Sucessfull description

```
:- description(tiger).

A tiger is an animal mammal carnivore satisfying conditions:

   c12   c14
```

by rule 10

2) unsucessfull description

```
:- description(horse).
Don't know this animal.
```

Remark:

The program answers only with the numbers of the conditions for comodity of the programmer. A procedure may be written to translate these numbers into the sentences of the conditions.

(This problem was sugested from "Artificial Intelligence", by P. Winston.

PROBLEM 51

Verbal statement:

Write a simple question-answering system for the exploration of a data base with transistor information (transistor name, material, polarity, function, power, collector base voltage, collector emiter voltage, amplification factor and capsula type), and according to the table:

characteristics / Transistor Name	mat	pol	fun	pow	vcb	vce	hfe	cap
2n2219	si	n	hsa	800	60	30	120	TO5
2n2904	si	p	hss	300	60	40	120	TO5
2n3055	si	n	lpa	115000	100	70	70	TO3
2n3904	si	n	hss	310	60	40	300	TO92
2n3906	si	p	hss	310	40	40	300	TO92

The system is requested to deal with user's changes of mind, translation of user's unities, and output of all characteristics for a given transistor, specified by the user.

Logic program:

```
begin:- write('Characteristics(value)'),nl,repeat,nl,
        continue(_).

continue(L):- write('- '),ttyflush,read(C),process(C,L).
```

```
process('Goodbye',_).
process(stop,L):- ( trans(X,L),nl,write(trans(X,L)),nl,
            write('Do you want it(yes or no) ?'),
            nl,read(S),S=yes,!,
            write('New characteristics'),nl,fail ;
            write('No transistors available'),nl,continue(L) ).

process(C,L):- ( subs(C,L),L=L1 ; replace(C,L,L1) ),
                continue(L1).

subst(C,L):- reduct(C,C1),match(C1,L).

reduct(C,C1):-C=..[X,N,mili],C1=..[X,N].
reduct(C,C1):-C=..[X,N,w],C1=..[X,M],M is N*1000.
reduct(C,C1):-C=..[X,N,_],C1=..[X,N].
reduct(C,C).

match(mat(A),[A|_]).
match(pol(B),[_,B|_]).
match(fun(C),[_,_,C|_]).
match(pow(D),[_,_,_,D|_]).
match(vcb(E),[_,_,_,_,E|_]).
match(vce(F),[_,_,_,_,_,F|_]).
match(hfe(G),[_,_,_,_,_,_,G|_]).
match(cap(H),[_,_,_,_,_,_,_,H]).

replace(C,L,L1):- subst(C,L1),replace1(L,L1).

replace1(X,X):- var(X),!.
replace1([],[]):- !.
replace1([H|T],[H1|T1]):- !,replace1(H,H1),replace1(T,T1).
replace1(_,_).


/* Transistor information */

trans(t2n2219,[si,n,hsa,800,60,30,120,'TO5']).
trans(t2n2904,[si,p,hss,300,60,40,120,'TO5']).
trans(t2n3055,[si,n,lpa,'115000',100,70,70,'TO3']).
trans(t2n3904,[si,n,hss,310,60,40,300,'TO92']).
trans(t2n3906,[si,p,hss,310,40,40,300,'TO92']).
```

Execution:

To run the system, just perform the command: ':-begin.'

```
:-begin.
```

Caracteristics(value)

- mat(si).
- pol(p).
- fun(hss).
- pot(300,mili).
- vcb(60).

```
- stop.

trans(t2n2904,[si,p,hss,300,60,40,120,T05])

Do you want it (yes or no) ?
 no.

No transistors available
- fun(hsa).
- pol(n).
- pot(800).
- stop.

trans(t2n2219,[si,n,hsa,800,60,30,120,T05])

Do you want it(yes or no) ?
 yes.
New characteristics

- Goodbye.
```

PROBLEM 52 [MELLISH,1977]

Verbal statement:

Write a program that behaves as a travel agent and holds an interface with an user in order to determine his precise travel requirements and to present him with information enabling him to choose air flights.

Imagine a question-asking strategy determined by the requirements to fill in slots in a system of frames in a depth-first manner. To prevent unnecessary questions imagine a system of default values and a mechanism that can avoid asking questions whose answers have been provided either implicitly or explicitly at some earlier time.

The program must also be able to cope with the client changing his mind by means of dealing with the contradictions arising from his inputs.

Suppose you have the following flight information, already written as a PROLOG data base :

```
flight(edinburgh,paris,jan,morn,f1).
flight(edinburgh,paris,jan,aft,f2).
flight(edinburgh,paris,jan,aft,f3).
flight(paris,rome,feb,morn,f4).
flight(paris,nome,feb,morn,f5).
flight(paris,rome,feb,aft,f6).
flight(rome,edinburgh,mar,morn,f7).
```

where flight(F,T,D,Ti,Fl) means a flight from F to T in month D at time Ti with number Fl.

Logic program:

```
:-op(410,xfy,',').


/* Overall control of the dialogue */

talk:- default(trip(1),exists),default(trip(2),exists),
       default(homeport(1),edinburgh),
       repeat,dialogue(X), nl,nl,
       display('trips booked :'),nl,nl,
       output(X),nl.

dialogue(X):- nextafter(0,M),tripspecification(M,X).


/* Gathering of trip information */

tripspecification(N,tr(d(A),t(B),f(C),to(D),f1(E),tr(F)).Ts):-
    discover(date(N),A), discover(time(N),B),
    discover(homeport(N),C), discover(foreignport(N),D),
    setflight(C,D,A,B,E,N), discover(traveller(N),F),!,
    continue(D,F,N,Ts).

continue(Dest,Trav,N,Ts):- nextafter(N,M),!,
    default(homeport(M),Dest),
    recall(homeport(1),H), default(foreignport(M),H),
    default(traveller(M),Trav),!,
    tripspecification(M,Ts).
continue(_,_,_,[]).

setflight(F,T,D,Ti,F1,N):- fact(ok(F1,N)yes),
    feasible(F,T,D,Ti,F1,N) , !.
setflight(F,T,D,Ti,F1,N):- flight(F,T,D,Ti,F1),
    flightok(F1,N,Con), !, Con = 0.
setflight(_,_,_,_,_,_):- display('no flights available'),
    nl,discover(change,Z).

flightok(F1,N,0):- discover(ok(F1,N),R), !, R = yes.
flightok(_,_,1).

feasible(F,T,D,Ti,F1,N):- flight(F,T,D,Ti,F1),!.
feasible(_,_,_,_,_,N):- retract(fact(ok(F,N),X)),fail.


/* Manipulation of facts */

discover(lit,Val):- fact(lit,Val), !.
discover(lit,Val):- repeat,askclient(lit,Val,Con), !,
            Con = 0.

default(Lit,Val):- fact(Lit,Newval), !.
default(Lit,Val):- assert(fact(Lit,Val)), !.

recall(Lit,Val):- fact(Lit,Val), !.
```

```
/* Communication with user */

askclient(Lit,Val,Con):-
    display(Lit), display(' ?'),nl,
    read(Input), interpret(Input,Con), !,
    notaskagain(Lit,Val,Con), !.

notaskagain(Lit,Val,0):- !, fact(Lit,Val).
notaskagain(_,_,_).

interpret([],0).
interpret(A,B,R):- dealwith(A,S), interpret(B,T),
                   R is S + T.

dealwith(A,B,0):- fact(A,B), !.
dealwith(A,B,1):- fact(A,C), retract(fact(A,C)),
                  assert(fact(A,B)), !.
dealwith(A,B,_):- assert(fact(A,B)).


/* Miscellaneous */

nextafter(N,M):- fact(trip(M),exists), N < M,
                 P is N + 1, nonebetween(P,M), !.

nonebetween(X,X):- !.
nonebetween(X,Y):- fact(trip(X),exists), !, fail.
nonebetween(X,Y):- Z is X + 1, nonebetween(Z,Y).

output(A,B):- display(A),nl,output(B).


/* Flight information */

flight(edinburgh,paris,jan,morn,f1).
flight(edinburgh,paris,jan,aft,f2).
flight(edinburgh,paris,jan,aft,f3).
flight(paris,rome,feb,morn,f4).
flight(paris,rome,feb,morn,f5).
flight(paris,rome,feb,aft,f6).
flight(rome,edinburgh,mar,morn,f7).
```

Execution:

```
:-talk.
date(1)?
: [trip(3).exists,foreignport(1).paris,foreignport(2).rome].
date(1)?
: [date(1).jan, traveller(1).me].
time(1)?
: [time(1).morn].
ok(f1,1)?
: [ok(f1,1).no].
no flights available
change?
```

```
 ! [time(1).aft,date(2).feb].
ok(f2,1)?
 ! [ok(f2,1).no].
ok(f3,1)?
 ! [ok(f3,1).yes].
time(2)?
 ! [time(2).morn,traveller(2).fred].
ok(f4,2)?
 ! [ok(f4,2).yes].
date(3)?
 ! [date(3).mar,trip(1).notexists,ok(f4,2)no].
ok(f5,2)?
 ! [time(2).aft].
ok(f6,2)?
 ! [ok(f6,2).yes].
time(3)?
 ! [time(3).morn].
ok(f7,3)?
 ! [ok(f7,3).yes].

 trips booked:

tr(d(feb),t(aft),f(paris),to(rome),fl(f6),tr(fred))
tr(d(mar),t(morn),f(rome),to(edinburgh),fl(f7),tr(fred))
```

PROBLEM 53 [SILVA&COTTA,1978]

Verbal statement:

In the preceding problem, the program has two ways of achieving a situation in which there are "no flight available".

The first way corresponds to the inexistance of a flight in the data base for the user's specifications, and the other corresponds to a disagreement about all the flights the program knows for the user's specifications.

Write a new version of that program where these situations are distinguished. For the first situation output the literals whose values are preventing the attainment of a flight.

Logic program:

The procedure "setflight" has now the following form:

```
setflight(F,T,D,Ti,Fl,N):- fact(ok(Fl,N),yes),
                           feasible(F,T,D,Ti,Fl,N),!.
setflight(F,T,D,Ti,Fl,N):- flight(F,T,D,Ti,Fl),
                           flightok(Fl,N,Con),!, Con=0.
```

```
setflight(F,T,D,Ti,Fl,N):- (flight(F,T,D,Ti,_),
     display('no more flights available'),nl,!,
     discover(change,Z)).
setflight(F,T,D,Ti,Fl,N):- display('no flights available'),nl,
    display('possible changes:'),nl,alterations(F,T,D,Ti).
setflight(_,_,_,_,_,N):- read(Input),interpret(Input,Con),fail.
```

The new procedure alteration has the form:

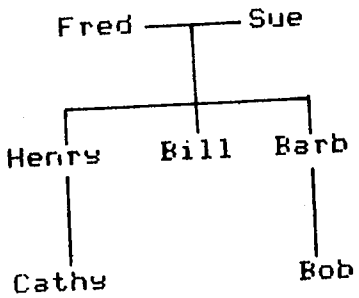
```
alterations(F,T,D,Ti):- first(flight(F,T,D,_,_)),
                        display('time'),nl,fail.
alterations(F,T,D,Ti):- first(flight(F,T,_,Ti,_)),
                        display('date'),nl,fail.
alterations(F,T,D,Ti):- first(flight(F,_,D,Ti,_)),
                        display('foreignport'),nl,fail.
alterations(F,T,D,Ti):- first((flight(F,T,X,Y,_),X\==D,
                        Y\==Ti)),
                        display('time and date'),nl,fail.
alterations(F,T,D,Ti):- first((flight(F,X,D,Y,_),X\==T,
                        Y\==Ti)),
                        display('time and foreignport'),nl,fail.
alterations(F,T,D,Ti):- first((flight(F,X,Y,Ti,_),X\==T,
                        display('date and foreignport'),nl,fail.
alterations(F,T,D,Ti):- first((flight(F,X,Y,Z,_),X\==T,
                        Y\==D, Z\==Ti)),
                        display('time,date and foreignport'),nl,
                        fail.
```

```
first(P):- P, !.
first((P,Q)):- P,first(Q),!.
```

PROBLEM 54

Verbal statement:

Specify a virtual relational data base for the kinship facts.
Take the following family tree for facts:



Logic program:

```
grandparent(X,Z):- grandfather(X,Z);grandmother(X,Z).

grandfather(X,Z):- father(X,Y),parent(Y,Z).

grandmother(X,Z):- mother(X,Y),parent(Y,Z).

parent(X,Y):- father(X,Y);mother(X,Y).

brother(X,Y):- male(X),parent(Z,X),parent(Z,Y),X\==Y.

sister(X,Y):- female(X),parent(Z,X),parent(Z,Y),X\==Y.

uncle(X,Y):- male(X),parent(Z,X),grandparent(Z,Y).

aunt(X,Y):- female(X),parent(Z,X),grandparent(Z,Y).

cousin(X,Y):- parent(Z,X),parent(T,Y),
              (brother(Z,T);sister(Z,T)).

husband(X,Y):- father(X,Z),mother(Y,Z).

wife(X,Y):- husband(Y,X).



father(fred,henry).      mother(sue,henry).
father(fred,bill).       mother(sue,bill).
father(fred,barb).       mother(sue,barb).
father(henry,cat).       mother(barb,bob).

male(fred).              female(barb).
male(bob).               female(sue).
male(henry).             female(cathy).
male(bill).
```

PROBLEM 55 [TOWNSHEND,1979]

Verbal statement:

Write a program to implement a pre-registration appointments matching scheme. It covers the allocation of the pre-registration six-month surgical and medical appointments that the law requires (here called jobs, or "posts" with a particular consultant's firm) to newly qualified doctors (here referred to as applicants).
A general representation of a job might be: the 'job number', the special restrictions on the applicants to be allocated to this job, the numbers of posts available in the six-month periods starting in August and February, the 'job preference list', the list of the reference numbers of applicants for the posts in that job, ranked by the consultant in charge in the

order of his preference and the result lists.
The corresponding record for an applicant might be: the 'job number', the sex, the season, the preference lists and the corresponding result lists.
The problem of assigning the members of two sets to one another is dealt according to the following definition of a 'stable marriage'. "Consider two distinct sets A and B. An assignment of the members of A to the members of B is said to be a stable marriage if and only if there exists no elements a and b (belonging to A and B respectively) who are not assigned to each other but who would both prefer each other to their present partners".

Logic program:

```
   /* PRAMS PART 1 in PROLOG */

program:-jobs(Jobslist),apps(Appslist),
        prepare(Jobslist),
        alloc1(Joblist,Applist,Jobslist).

alloc1([Job|JL],Appslist,Jobslist):-
        offer(Job,Appslist,Jobslist),
        allocc(JL,Appslist,Jobslist).
alloc1([],Appslist,Jobslist):-show(Jobslist).


offer(Job(J,Lim,NA,NF,Jprefs,As,Af,All),Appslist,Jobslist):-
        write(J),
        N is NA+NF,
        offer1(J,Jprefs,N,0,All,Appslidt,Jobslist).

offer1(_,[],_,_,_,_,_).
offer1(_,_,N,N,_,_,_).
offer1(J,[A|Jprefs],N,K,All,Appslist,Jobslist):-
        member(A,All),
        accept(A,J,Appslist,Jobslist),
        K1 is K+1,
        offer1(J,Jprefs,N,K1,All,Appslist,Jobslist).
offer1(J,[A|Jprefs],N,K,All,Appslist,Jobslist):-
        offer1(J,Jprefs,N,K,All,Appslist,Jobslist).


accept(A,J,Appslist,Jobslist):-
        member(app(A,_,_,Maprefs,X,Saprefs,Y),Appslist),
        accept1(A,Maprefs,Saprefs,X,Y,J,Appslist,Jobslist).

accept1(A,Maprefs,Saprefs,X,Y,J,Appslist,Jobslist):-
        member(J,Maprefs),!,
        accept2(A,J,Maprefs,X,Appslist,Jobslist).
accept1(A,Maprefs,Saprefs,X,Y,J,Appslist,Jobslist):-
        member(J,Saprefs),!,
        accept2(A,J,Saprefs,Y,Appslist,Jobslist).
```

```
accept2(_,J,_,X,_,_):-nonvar(X),X=J,!.
accept2(_,J,_,X,_,_):-nonvar(X),!,fail.
accept2(A,J,Prefs,X,Appslist,Jobslist):-
        accept3(A,J,Prefs,X,Appslist,Jobslist).


accept3(_,J,[J|_],J,_,_).
accept3(A,J,[J1|Prefs],X,Appslist,Jobslist):-
        not(ask(A,J1,X,Appslist,Jobslist)),
        accept3(A,J,Prefs,X,Appslist,Jobslist).


ask(A,J,J,Appslist,Jobslist):-
        member(Job(J,Lim,NA,NF,Jprefs,As,Af,All),Jobslist),
        N is NA+NF,
        ask1(A,J,Jprefs,N,All,Appslist,Jobslist).
        ask1(A,J,Jprefs,N,All,Appslist,Jobslist):-
        ismember(A,All).
ask1(A,J,Jprefs,N,All,Appslist,Jobslist):-
        offer1(J,Jprefs,N,0,All,Appslist,Jobslist),!,
        ismember(A,All).


prepare([Job(No,Lim,NA,NF,Prefs,As,Af,All)|JL]):-
        makelist(NA,[],As),
        makelist(NF,[],Af),
        N is NA+NF,
        makelist(N,[],All),
        prepare(JL).
prepare([]).


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

ismember(X,[Y|_]):-nonvar(Y),X=Y,!.
ismember(X,[_|L]):-ismember(X,L).



Jobs([
Job(1,0,1,1,[1,9,4,6,2,7,3,5,8],_,_,_),
Job(2,0,1,1,[3,2,1,5,7,6,10],_,_,_),
Job(3,0,1,0,[9,6,7,5,3,4,1,2],_,_,_),
Job(4,0,1,2,[4,7,1,6,2,5,3,10],_,_,_),
Job(5,0,2,2,[1,4,8,3,2,5,10,7,6],_,_,_),
Job(6,0,3,3,[9,10,4,6,5,3,2,1],_,_,_),
Job(7,0,2,2,[1,2,9,10,6,4,3,5],_,_,_),
Job(8,0,3,0,[7,8,1,2,6,4,3,5],_,_,_),
Job(9,0,0,3,[4,3,8,10,9],_,_,_)
]).
```

```
apps([
app(1,m,0,[1,2,3,4],_,[5,6,7,8],_),
app(2,f,0,[1,4,2,3],_,[6,7,8,5],_),
app(3,m,1,[1,3,2,4],_,[5,8,7,9],_),
app(4,m,0,[1,3,4,9],_,[6,5,8,7],_),
app(5,m,2,[3,1,4,2],_,[5,6,8,7],_),
app(6,f,3,[4,1,2,3],_,[7,5,6,8],_),
app(7,m,0,[2,4,1,3],_,[8,5],_),
app(8,m,0,[1],_,[5,8,9],_),
app(9,f,1,[3,1],_,[6,7,9],_),
app(10,m,0,[2,4,9],_,[5,6,7],_)
]).
```

Execution:

```
:-program.
123456789

RESULTS
1    [ 1 4 ]
2    [ 3 5 ]
3    [ 9 ]
4    [ 7 6 2 ]
5    [ 1 8 3 5 ]
6    [ 9 10 4 2 _ _ ]
7    [ 6 _ _ _ ]
8    [ 7 _ _ ]
9    [ 10 _ _ ]
```

PROBLEM 56 [PEREIRA&PORTO,1980]

Verbal statement:

Write a program for colouring any planar map with at most four colours, such that no two adjacent regions have the same colour.

Logic program:

The program consists of a complete list of pairs of different colours, taken from a collection of form. These constitute the admissible pairs of colours for regions next to each other.

```
next(blue,yellow).

next(blue,red).

next(blue,green).
```

```
next(yellow,blue).

next(yellow,red).

next(yellow,green).

next(red,blue).

next(red,yellow).

next(red,green).

next(green,blue).

next(green,yellow).

next(green,red).
```

To obtain a colouring of the map below, we give as a goal to the program all the pairs of regions that are next to each other. This can be done systematically by first pairing region 1 with higher numbered regions next to it, then region 2, etc.

```
goal(R1,R2,R3,R4,R5,R6):- next(R1,R2),next(R1,R3),
                          next(R1,R5),next(R1,R6),
                          next(R2,R3),next(R2,R4),
                          next(R2,R5),next(R2,R6),
                          next(R3,R4),next(R3,R6),
                          next(R5,R6).
```

PROBLEM 57 [MÁRKUSZ,1978]

Verbal statement:

Write a program for designing an architectural unit obeying to the following specifications:

- Two rectangular rooms.
- Each room has a window and interior door.
- Rooms are conected by interior door.
- One room also has an exterior door.
- A wall can have only one door or window.
- No window can face north.
- Windows cannot be on opposite sides of the unit.

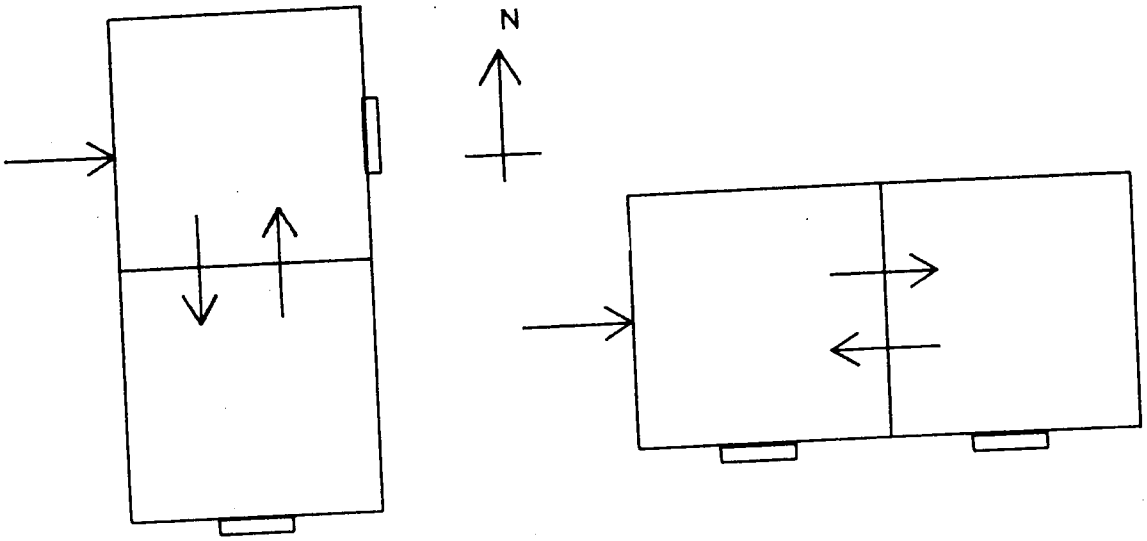


Figure 1
Sample Plans

Logic program:

```
plan(FD,D1,W1,D2,W2):- frontroom(FD,D1,W1),
                       opposite(D1,D2),
                       room(D2,W2),
                       notopposite(W1,W2).

frontroom(FD,D,W):- room(D,W),
                    direction(FD),
                    FD=\=,FD=\=W.

room(D,W):- direction(D),
            direction(W),
            D=\=W, W=\=north.

direction(north).
direction(south).
direction(east).
direction(west).

opposite(north,south).
opposite(south,north).
opposite(east,west).
opposite(west,east).

notopposite(D1,D2):- opposite(D1,D3),
                     D2=\=D3.
```

Execution:

Planning a unit entered from the west:

```
?-plan(west,D1,W1,D2,W2).
```

```
D1 = east,
W1 = south,
D2 = west,
W2 = south
```

PROBLEM 58 [PEREIRA&PORTO,1980]

Verbal statement:

Write the following query:

> 'Is there a student such that a professor teaches him two different courses in the same room?'

for a data base of students who take courses, professors who teach courses, and courses held on certain weekdays and rooms.

Logic program:

```
query(S,P):- student(S,C1),
             course(C1,D1,R),
             professor(P,C1),
             student(S,C2),
             course(C2,D2,R),
             professor(P,C2),C1 =\= C2.

student(robert,prolog).
student(john,music).
student(john,prolog).
student(john,surf).
student(mary,science).
student(mary,art).
student(mary,physics).

professor(luis,prolog).
professor(luis,surf).
professor(antonio,prolog).
professor(eureka,music).
professor(eureka,art).
professor(eureka,science).
professor(eureka,physics).


course(prolog,monday,room1).
course(prolog,friday,room1).
course(surf,sunday,beach).
course(maths,tuesday,room1).
course(maths,friday,room2).
```

```
course(science,thursday,room1).
course(science,friday,room2).
course(art,tuesday,room1).
course(physics,thursday,room3).
course(physics,saturday,room2).
```

PROBLEM 59 [DARVAS et al,1978]

Verbal statement:

Write a data base for predicting drug interactions.

Logic program:

```
/* Facts about chemistry */

contains(procaine,amino_group).
base(X):- contains(X,amino_group).

/* Data on various drugs */

ingredient(aspirin,salicylic_acid).
ingredient(whisky,ethanol).

/* General rules about drug interactions */

interaction(Drug1,Drug2,
            increased_absorption_by_ion_pair_formation(Agent1,
            Agent2)):-

        ingredient(Drug1,Agent1),
        ingredient(Drug2,Agent2),
        quaternary_ammonium_salt(Agent1),
        anti_arrihythmic(Agent1),
        salicylate(Agent2).
```

Execution:

```
        ?-interaction(aspirin,whisky,What).
```

3.3 PROBLEM SOLVING (nos. 60 to 65)

PROBLEM 60 [WARREN,1975b]

Verbal statement:

Write a program that receives a scenary as input and produces an interpretation of the scenary as output.

Consider a scenary composed by a certain disposition of solids. The input for the program is a description of the solids in terms of their edges, and the output will be an interpretation in terms of what solids constitute the scenery and how are their faces.

Consider only vertical edges, positive and negative slope edges, and that a face may be left, right or horizontal.

Apply your program to the scenary:

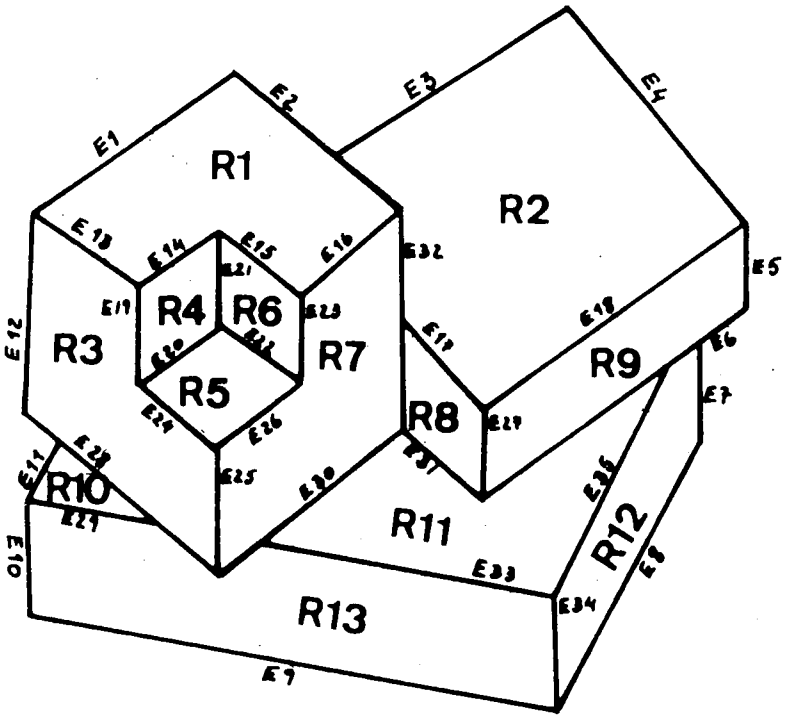


Figure 2

Where the edges and the faces are numbered (E1,...,E35 and R1,...,R13, respectively) in an arbitrary order.

Sugestion: 1) When defining the scenary begin by the edges that limit it, that is, begin by the frontier edges.

2) Define a set of rules caracterizing the different kinds of edges in terms of convexity or concavity of the regions they establish.

3) Label the contour edges clockwise.

Logic program:

```
:- op(500,xfy,';').
:- op(300,xfy,':').

goal:- solution(X),nl,nl,write(X),nl,nl.

/* Set of interpretation rules */

vertical_edge(left:B,right:B,positive).
vertical_edge(right:B1,left:B2,negative).
vertical_edge(right:B,X,down).
vertical_edge(X,left:B,up).

positive_edge(horizontal:B,right:B,positive).
positive_edge(right:B1,horizontal:B2,negative).
positive_edge(X,horizontal:B,up).
positive_edge(right:B,X,down).

negative_edge(left:B,horizontal:B,positive).
negative_edge(horizontal:B1,left:B2,negative).
negative_edge(horizontal:B,X,down).
negative_edge(X,left:B,up).

/* Defining the scenary */

solution(R1;R2;R3;R4;R5;R6;R7;R8;R9;R10;R11;R12;R13):-
          positive_edge(background,R1,E1),
          negative_edge(R1,background,E2),
          positive_edge(background,R2,E3),
          negative_edge(R2,background,E4),
          vertical_edge(R9,background,E5),
          positive_edge(R9,background,E6),
          vertical_edge(R12,background,E7),
          positive_edge(R12,background,E8),
          negative_edge(background,R13,E9),
          vertical_edge(background,R13,E10),
          positive_edge(background,R10,E11),
          vertical_edge(background,R3,E12),
          negative_edge(R3,R1,E13),
          positive_edge(R1,R4,E14),
          negative_edge(R6,R1,E15),
          positive_edge(R1,R7,E16),
          negative_edge(R8,R2,E17),
          positive_edge(R2,R9,E18),
          vertical_edge(R3,R4,E19),
```

```
positive_edge(R4,R5,E20),
vertical_edge(R4,R6,E21),
negative_edge(R5,R6,E22),
vertical_edge(R6,R7,E23),
negative_edge(R3,R5,E24),
vertical_edge(R3,R7,E25),
positive_edge(R5,R7,E26),
vertical_edge(R8,R9,E27),
negative_edge(R10,R3,E28),
negative_edge(R13,R10,E29),
positive_edge(R7,R11,E30),
negative_edge(R11,R8,E31),
vertical_edge(R7,R8,E32),
negative_edge(R13,R11,E33),
vertical_edge(R13,R12,E34),
positive_edge(R11,R12,E35).
```

Execution:

```
:-goal.

horizontal:X1 ; horizontal:X2 ; left:X1 ; right:X1 ;
horizontal:X1 ; left:X1 ; right:X1 ; left:X2 ;
right:X2 ; horizontal:X3 ; horizontal:X3 ; right:X3;
left:X3
```

this interpretation corresponds to the scenary:

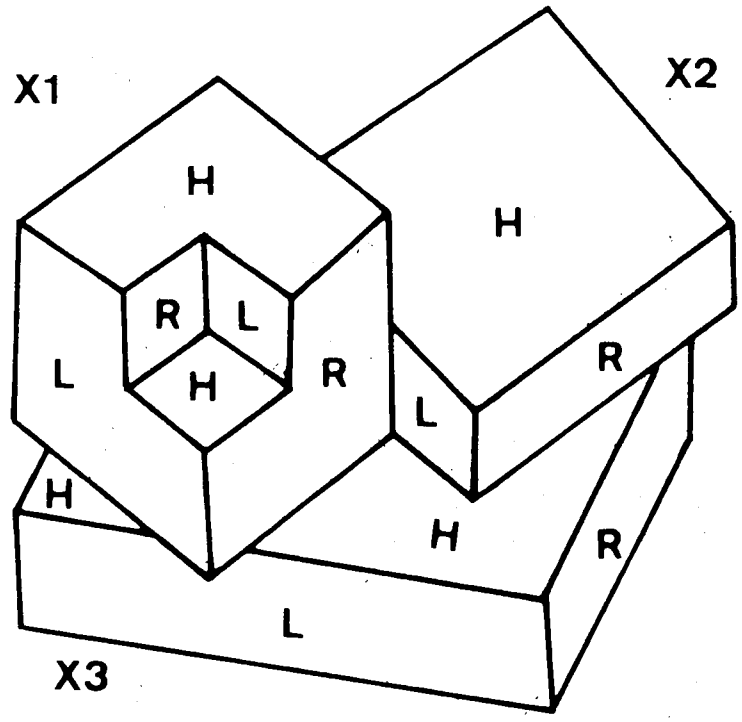


Figure 3

PROBLEM 61 [ROUSSEL,1975]

Verbal statement:

There are three pegs, A,B, and C, and three disks of different sizes a,b, and c. The disks have holes in their centers so that they can be stacked on the pegs.

Initially all the disks are on peg A; the largest, disk c, is on the bottom, and the smallest, disk a, is on the top.

It is desired to transfer all of the disks to peg C by moving disks one at a time. Only the top disk on a peg can be moved, but it can never be placed on top of a smaller disk (Tower-of-Hanoi puzzle).

Logic program:

```
hanoi(N):- hanoi(N,L,[]),nl,write(L),nl.

hanoi(N)--> put(N,a,b,c).

put(0,_,_,_)--> [].
put(N,A,B,C)--> {M is N-1},
                put(M,A,C,B),move(A,B), put(M,C,B,A).

move(X,Y)--> [from,X,to,Y].
```

PROBLEM 62 [EMDEN,1978]

Verbal statement:

Consider the abstract game of Mastermind with 6 colours and 4 places to combine them. Consider that, in the score, a black key peg means strong match (right colour in a right place) and a white key peg means weak match (right colour in a wrong place). Write a program able to perform the Codemaker when you play the Codebreaker and vice-versa.

Sugestion: Consider that the Codebreaker constructs a sequence p1,...,pn of probes with pi =/= C for i =/= n and pn = C, where C means the key code. The Codemaker constructs a sequence s1,...,sn such that si = f(pi,C) where f is the score function. The selection of pi by the Codebreaker may depend on (p1,s1),...,(pi-1,si-1). It is the Codebreaker's objective to make n as small as possible.

Logic program

```
/* Program for the Mastermind game */

:-op(300,xfy,',').
:-op(150,fy,'s').

/* Interactive manager */

play:- write('Mastermind at your service.'), nl,
       write('Enter an arbitrary number between 0 and 164.'),
       nl,read(X), assert(seed(X)),
       write('Example format for entering code :  '),
       write('[yellow,blue,white,black]'), nl,
       write('Example format for the score :  '),
       write('[s s s 0,s 0], nl,
       write('Meaning 3 blacks and 1 white.'), nl,
       repeat, play1.

play1:- write('Do you want to make or to break codes ?'), nl,
        write('Answer make or answer break.'), nl,
        read(X), start(X).

ask:- write('Do you want another game ?'), nl,
      write('Answer yes or answer no.'), nl,
      read(X), answer(X).

answer(yes):- !,fail.
answer(no):- retract(seed(_)), nl,
             write('Mastermind was pleased to serve you.'),
             nl,nl.

/* User performs the Codemaker */

start(make):- write('Enter code;  I promise not to look.'), nl,
              read(C), assert(code(C)),
              generate_code(P), score(P,S),
              write('My first probe is :  '), write(P), nl,
              write('Score :  '), write(S), nl,
              extendable((P,S),[],s s s s s 0),
              retract(code(_)), !, ask.

extendable((P,(s s s s _,_)),_,_):- write('The code must be: '),
                                    write(P), nl.
extendable(Cs,Ns):- can_code(Cs,Cc),
                    write('My next probe is: '), write(Cc), nl,
                    write('Score :  '),
                    score(Cc,S), write(S), nl,
                    extendable((Cc,S),Cs,N).

can_code([],_).
can_code((P1,S1),Ps,P):- mm(P1,P,S1), can_code(Ps,P).
```

```
/* User performs the Codebreaker */

start(break):- generate_code(C), assert(code(C)),
               write('Enter first probe.'), nl,
               read(P), score(P,S), continue_break(S),
               !, ask.

continue_break(s s s s _._):- write('You got it.'), nl,
                              retract(code(_)).

continue_break(S):- write('Your score :  '), write(S), nl,
                    write('Enter next probe or type stop.'), nl,
                    read(X), respond_to(X).

respond_to(stop):- !,write('I assume you give up;'),
                   write(' the code is: '),
                   retract(code(C)), write(C), nl.
respond_to(P):- score(P,S), continue_break(S).

/* Score function */

score(Probe,Score):- code(C), mm(Probe,C,Score).

mm(P,C,S1.S2.[]):- blacks(P,C,P1,C1,S1),
                   whites(P1,C1,S2,C1).

blacks([],[],[],[],0).
blacks(U.P,U.C,P1,C1,s S):- blacks(P,C,P1,C1,S).
blacks(U.P,V.C,U.P1,V.C1,S):- diff(U,V),
                              blacks(P,C,P1,C1,S).

whites([],C,0,Ms):- tuple(C,Ms).
whites(U.P,C,s S,Ms1.Ms):- del(U,C,C1,Ms),
                           whites(P,C1,S,Ms).
whites(U.P,C,S,Ms):- nonmem(U,C,Ms), whites(P,C,S,Ms).

/* Code generator */

generate_code(U.V.W.X.[]):- random_colour(U),
                            random_colour(V),
                            random_colour(W),
                            random_colour(X).

random_colour(X):- random_number(R), N is R mod 6,
                   N1 is N+1, colour(N1,X).

random_number(R1):- retract(seed(R)), X is R*125,
                    Y is X+1, R1 is Y mod 165,
                    assert(seed(R1)).

colour(1,black).                    colour(2,blue).
colour(3,green).                    colour(4,red).
colour(5,white).                    colour(6,yellow).
```

```
/* Miscellaneous */

/* del(U,Y,Y1) is the result of delecting U from list Y */

del(U,U.Y,Y,Ms):- tuple(Y,Ms).
del(U,V.Y,V.Y1,Ms1.Ms):- diff(U,V), del(U,Y,Y1,Ms).

/* nonmem(U,V) if U is not a member of list V */

nonmem(_,[],[]).
nonmem(U,V1.V,W1.W):- diff(U,V1), nonmem(U,V,W).

tuple([],[]).
tuple(U1.U,V1.V):- colour(_,U1), tuple(U,V).

diff(C1,C2):- colour(_,C1), colour(_,C2), not(C1=C2).


Execution:

| ?-play.
Mastermind at your service.
Enter an arbitrary number between 0 and 164.
| 123.
Example format for entering code :  [yellow,blue,white,black]
Example format for the score :  [s s s 0,s 0]
Meaning 3 blacks and 1 white.
Do you want to make or to break codes ?
Answer make or answer break.
| make.
Enter code; I promise not to look.
| [blue,white,blue,white].
My first probe is :  [blue,red,blue,black]
Score :  [s s 0,0]
My next probe is :  [blue,red,green,green]
Score :  [s 0,0]
My next probe is :  [blue,black,black,black]
Score :  [s 0,0]
My next probe is :  [blue,blue,blue,blue]
Score :  [s s 0,0]
My next probe is :  [blue,white,blue,white]
Score :  [s s s s 0,0]
The code must be :  [blue,white,blue,white]
Do you want another game ?
Answer yes or answer no.
| yes.
Do you want to make or to break codes ?
Answer make or answer break.
| break.
Enter first probe.
| [blue,blue,red,red].
Your score :  [0,s 0]
Enter next probe or type stop.
| [white,white,white,white].
Your score :  [s s 0,0]
```

```
Enter next probe or type stop.
| [white,white,blue,black].
Your score :  [s s 0,s 0]
Enter next probe or type stop.
| [white,white,green,blue].
Your score :  [s 0,s 0]
Enter next probe or type stop.
| [white,red,white,black].
You got it.
Do you want another game ?
Answer yes or answer no.
| no.

Mastermind was pleased to serve you.
```

EBOBLEM 63 [BRUYNOOGHE,1978]

Verbal statement:

Write a program to solve the n-queens problem (where n=4).

Consider a chess table 4x4 and put four queens on it, obeying to the condition that each of them cannot attack any of the others.

Sugestion: Consider the followins representation of the chess table where each square is 'p(number of line,number of column)'.

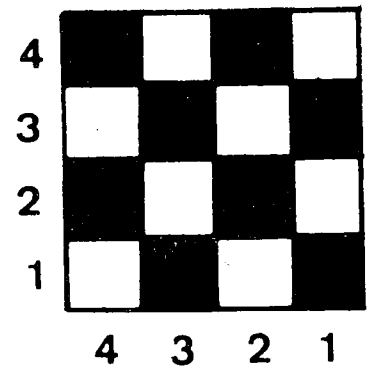


Figure 4

Logic program 1:

```
begin:- perm([4,3,2,1],L),pair([4,3,2,1],L,Q),
        safe(Q),write(Q),nl,nl.

/* Generator of permutations */

perm([],[]).
perm([X|Y],[U|V]):- del(U,[X|Y],W),perm(W,V).

del(X,[X|Y],Y).
del(U,[X|Y],[X|V]):- del(U,Y,V).
```

```
/* Pairing lines with columns */

pair([],[],[]).
pair([X|Y],[U|V],[p(X,U)|W]):- pair(Y,V,W).

/* Check if the queen positions are compatible */

safe([]).
safe([P|Q]):- check(P,Q),safe(Q).

check(P,[]).
check(P,[Q|R]):- diag(P,Q),check(P,R).

/* non_attacking condition on diagonals */

diag(p(X1,Y1),p(X2,Y2)):- Dx is X1-X2,
                          Dy is Y1-Y2,
                          Dx =\= Dy,
                          Dx =\= -Dy.
```

Logic program 2: (more efficient version)

```
queens(L,[C1|RC],R):- del(L1,L,RL),safe(p(L1,C1),R),
                      queens(RL,RC,[p(L1,C1)|R]).
queens([],[],R).
```

Execution:

```
:-queens([1,2,3,4],[1,2,3,4],[]).
```

PROBLEM 64

Verbal statement:

Three missionaries and three cannibals seek to cross a river. A boat is available which holds two people, and which can be navigated by any combination of missionaries and cannibals involving one or two people. If the missionaries on either bank of the river, or 'en route' in the river, are outnumbered at any time by the cannibals, the cannibals will indulge their anthropophagic tendencies and do away with the missionaries. Find the simplest schedule of crossing that will permit all the missionaries and cannibals to cross the river safely.

Sugestion: Consider the representation of the left side of the river (for example) as s(M,C,B) where M is the number of missionaries, C is the number of cannibals, and B takes the value 0 or 1 acordingly with the position of the boat (left side or risht side respectively).

Logic program:

```
:-op(300,xfy,=>).


plan:- movements(s(3,3,1),P),print(P).


movements(s(0,0,0),done).
movements(X,X=>Y=>Z):- move(X,Y),move_back(Y,Y1),
                       test(X,Y1),
                       movements(Y1,Z).

move(X,Y):- move(X,Y,2);move(X,Y,1);move11(X,Y).

move_back(s(0,0,0),s(0,0,0)).
move_back(X,Y):- move_back(X,Y,2);move_back(X,Y,1);
                 moveback11(X,Y).

move(s(M,C,X),s(M1,C,Y),N):- (M=N ;
                              M>N, M-N>C),
                             M1 is M-N, M1=<3,
                             Y is 1-X.
move(s(M,C,X),s(M,C1,Y),N):- C>=N, C1 is C-N,
                             C1=<3, Y is 1-X.

move11(s(M,C,X),s(M1,C1,Y)):- M>=1, C>=1,
                              M1 is M-1, C1 is C-1,
                              Y is 1-X.

move_back(s(M,C,X),s(M1,C,Y),N):- N=<3-M, M+N=C,
                                  M1 is M+N, Y is 1-X.
move_back(s(M,C,X),s(M,C1,Y),N):- N=<3-C,
                                  (M=0 ; M>=C+N),
                                  C1 is C+N, Y is 1-X.

move_back11(s(M,C,X),s(M1,C1,Y)):- M>=1, C>=1,
                                   M1 is M+1, C1 is C+1,
                                   Y is 1-X.


test(s(M,C,_),s(M,C,_)):- !,fail.
test(_,_).

print(X=>Y):- write(X),nl,print(Y).
print(X):- nl,write(X),nl.


Execution:

:-plan.

s(3,3,1)
s(3,1,0)
s(3,2,1)
```

```
s(3,0,0)
s(3,1,1)
s(1,1,0)
s(2,2,1)
s(0,2,0)
s(0,3,1)
s(0,1,0)
s(0,2,1)
s(0,0,0)

done
```

EBOBLEM 65 [EMDEN,1980]

Verbal statement:

Write a program for the game of Nim defined as follows.

A position of the game of Nim can be visualized as a set of heaps of matches. Two players, called US and THEM, alternate making a move. As soon as a player, whose turn it is to move, is unable to make a move, the game is over and that player has lost; the other player is said to have won. A move consists of taking at least one match from exactly one heap.

Logic program:

```
:- op(300,xf,+).


s(X,Y):- move(X,Y),not(t(Y,Z)).
t(X,Y):- move(X,Y),not(s(Y,Z)).

append([],Y,Y).
append([U|X],Y,[U|Z]):- append(X,Y,Z).

move(X,Y):- append(U,[X1|V],X),
            takesome(X1,X2),
            append(U,[X2|V],Y).

takesome(X+,X).
takesome(X+,Y):- takesome(X,Y).
```

3.4 THEOREM PROVING (nos. 66 to 70)

PROBLEM 66

Verbal statement:

Write a simple Propositional Calculus Theorem Prover, covering equivalence, implication, disjunction, conjunction and negation.

Logic program:

```
/* Propositional Calculus Theorem Prover */


:-op(700,xfy,eqv).    /* equivalence */
:-op(650,xfy,imp).    /* implication */
:-op(600,xfy,or).     /* disjunction */
:-op(550,xfy,and).    /* conjunction */
:-op(500,fy,not).     /* negation    */


formulas:- repeat,read(T),( T==stop ;  theorem(T),fail ).

theorem(T):- ttynl,
             ( false(T), display('not valid'),!  ;
                         display('valid')         ),
             ttynl,ttynl.

false( 'FALSE' ):-!.

false( not 'TRUE' ):-!.

false( P eqv Q ):-false(( P imp Q ) and ( Q imp P )).

false( P imp Q ):-false( not P or Q ).

false( P or Q ):-false( P ),false( Q ).

false( P and Q ):-(false( P ) ;  false( Q )).

false( not not P ):-false( P ).

false( not(P eqv Q)):-false( not(P imp Q) or not(Q imp P)).

false( not(P im Q)):-false (not(not P or Q)).

false( not(P or Q)):-false( not P and not Q).

false( not(P and Q)):-false( not P or not Q).
```

PROBLEM 67

Verbal statement:

Program Wang's algorithm for proving theorems of propositional calculus.

The Wang's algorithm consists of writing down a series of lines, each simpler than the previous one, until a proof is completed -- or shown to be impossible. Each line consists of any number of wff's, separated by commas, on each side of an arrow.

Wang's algorithm:

1. As the first line, write the premises to the left of the arrow and the theorem to the right of the arrow:
   premise1,premise2,premise3 --> theorem

2. If the principal connective of a wff is a negation, drop the negation sign and move the wff to the other side of the arrow; for example,

   $P \vee Q, \neg(R \wedge S), \neg Q, P \vee R \rightarrow S, \neg P$

   is changed into

   $P \vee Q, P \vee R, P \rightarrow S, R \wedge S, Q.$

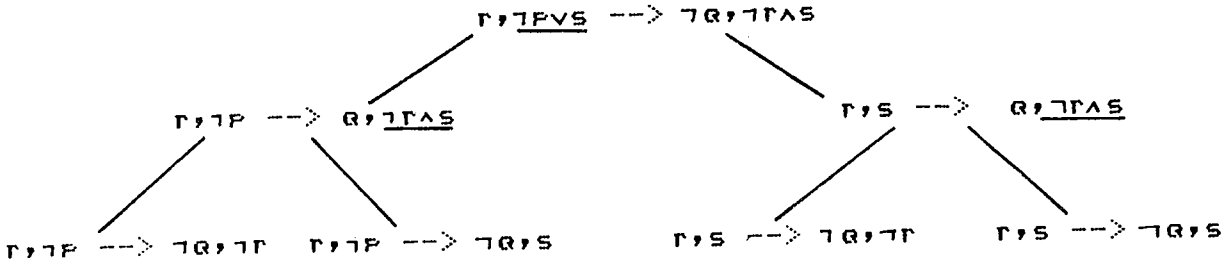
3. If the principal connective of a wff on the left of the arrow is $\wedge$ or on the right of the arrow is $\vee$, replace the connective by a comma; for example,

   $P \wedge Q, R \wedge (\neg P \vee S) \rightarrow \neg Q \vee \neg R$

   is changed into

   $P, Q, R, \neg P \vee S \rightarrow \neg Q, \neg R$

4. If the principal connective of a wff on the left of the arrow is $\vee$ or that on the right of the arrow is $\wedge$, then produce two new lines, each with one of the two sub-wff's replacing the wff; for example,



All the resulting lines must be proved in order to prove the original theorem.

5. If the same wff occurs on both sides of an arrow, the line is proved.

6. If no connectives remain in a line and no propositional variable occurs on both sides of the arrow, the line is not provable. In fact, assigning all the variables on the left to be TRUE and all the variables on the right to be FALSE will provide an example of the falsity of the theorem.

Logic program:

```
/* Operations */

:-op(700,xfy,<=>).        /* equivalence */
:-op(650,xfy,=>).         /* implication */
:-op(600,xfy,V).          /* disjunction */
:-op(550,xfy,&).          /* conjunction */
:-op(500,fy,-).           /* negation    */

/* Read_in and try to prove formula;
   write'valid or' not valid'accordingly */


formulas:- repeat,write('Formula:'),nl,
           read(T),(T==stop;theorem(T),fail).

theorem(T):- nl,nl,
             (prove([]&[]=>[]&[T]),!,nl,
             write('Formula is valid');
             nl,write('Formula is not valid')),nl,nl.

to_prove(T):- write(' prove,'),nl,write(T),nl,nl,
              prove(T).

prove(E1):- rule(E1,E2,Rule),!,
            write(E2),by_rule(Rule),nl,
            prove(E2).

/* Case for V on l.h.s.  */

prove(L & [H V I:T] => R):- !,
            first_branch,to_prove(L & [H:T] => R),
            branch_proved,
            second_branch,to_prove(L & [I:T] => R),
            branch_proved.

/* Case for & on r.h.s.  */

prove(L => R & [H & I:T]):-!,
            first_branch, to_prove(L => R & [H:T]),
            branch_proved,
            second_branch, to_prove(L => R & [I:T]),
            branch_proved.
```

```
/* Case for atom */

prove(L & [H|T] => R):- !,prove([H|L] & T => R).

prove(L => R & [H|T]):- !,prove(L => [H|R]& T).

/* Finally, check whether tautology */

prove(T):- tautology(T),write('Tautology.'),nl.
prove(_):- write('This branch is not provable.'),fail.

/* Cases where => appears in one of the sides */

rule(L & [H => I|T] => R,
     L & [-H V I|T] => R, rule_5).
rule(L => R & [H => I|T],
     L => R & [-H V I|T],rule_6).

/* Cases where <=> appears in one of the sides */

rule(L & [H <=> I|T] => R,
     L & [(H => I)&(I => H)|T] => R, rule_7).
rule((L => R & [H <=> I|T],
      L => R & [(H => I) & (I => H)|T],rule_8).

/* Cases where - appears */

rule(L & [-H|T] => R & R2,
     L & T => R & [H|R2],rule_2).
rule(L1 & L2 => R & [-H & T],
     L1 & [H|L2] => R & T,rule_2).

/* Case for & on l.h.s.  */

rule(L & [H & I|T] => R,
     L & [H,I|T] => R,rule_3).

/* Case for V on r.h.s.  */

rule(L => R & [H V I |T],
     L => R & [H,I|T],rule_3).

tautology(L & [] => R & []):- member(M,L),
                              member(M,R).

branch_proved:- write('This branch has been proved.'),nl.

first_branch:- nl,write('First branch:').

second_branch:- nl,write('Second branch:').

by_rule(R):- write('     by'),write(R),nl,nl.

member(H,[H|_]).
member(I,[_|T]):- member(I,T).
```

```
Execution:

Formula:

a => a.

[] & [] => [] & [-a V a]                                    by rule_6

[] & [] => [] & [-a,a]                                      by rule_3

[] & [a] => [] & [a]                                        by rule_2

Tautology

Formula is valid

Formula:

(a => b) & (b => c) => (a => c).

[] & [] => [] & [-((a => b) & (b => c)) V (a => c)]         by rule_6

[] & [] => [] & [-((a => b) & (b => c)),a => c]             by rule_3

[] & [(a => b) & (b => c)] => [] & [a => c]                 by rule_2

[] & [(a => b) & (b => c)] => [] & [-a V c]                 by rule_6

[] & [a => b,b => c] => [] & [-a V c]                       by rule_3

[] & [-a V b,b => c] => [] & [-a V c]                       by rule_5

[] & [-a V b,b => c] => [] & [-a,c]                         by rule_3

[] & [a,-a V b,b => c] => [] & [c]                          by rule_2

First branch:  Prove,

[a] & [-a, b => c] => [] & [c]

[a] & [b => c] => [] & [a,c]                                 by rule_2

[a] & [-b V c] => [] & [a,c]                                 by rule_5

First branch:  Prove,

[a] & [-b] => [] & [a,c]

[a] & [] => [] & [b,a,c]                                    by rule_2

Tautology.

This branch has been Proved.
```

```
Second branch :  prove,
[a] & [c] => [] & [a,c]
Tautology.
This branch has been proved.
Second branch :  prove,
[a] & [b,b => c] => [] & [c]
[b,a] & [-b V c] => [] & [c]                                   by rule_5
First branch :  prove,
[b,a] & [-b] => [] & [c]
[b,a] & [] => [] & [b,c]                                       by rule_2
Tautology.
This branch has been proved.
Second branch :  prove,
[b,a] & [c] => [] & [c]
Tautology.
This branch has been proved.
Formula is valid.

Formula:
(a => b) <=> (-a V b)
[] & []=>[] & [((a=>b)=>-a V b) & (-a V b=>a=>b)]              by rule_8
First branch :  prove,
[] & [] => [] & [(a => b) => -a V b]
[] & [] => [] & [-(a => b) V -a V b]                           by rule_6
[] & [] => [] & [-(a => b),-a V b]                             by rule_3
[] & [a => b] => [] & [-a V b]                                 by rule_2
[] & [-a V b] => [] & [-a V b]                                 by rule_3
[] & [-a V b] => [] & [-a,b]                                   by rule_3
```

```
[] & [a,-a V b] => [] & [b]                                    by rule_2
First branch :  prove,
[a] & [-a] => [] & [b]
[a] & [] => [] & [a,b]                                         by rule_2
Tautology.
This branch has been proved.
Second branch :  prove,
[a] & [b] => [] & [b]
Tautology.
This branch has been proved.
Second branch :  prove,
[] & [] => [] & [-a V b => a => b]
[] & [] => [] & [-(-a V b) V (a => b)]                         by rule_6
[] & [] => [] & [-(-a V b),a => b]                             by rule_3
[] & [-a V b] => [] & [a => b]                                 by rule_2
[] & [-a V b] => [] & [-a V b]                                 by rule_6
[] & [-a V b] => [] & [-a,b]                                   by rule_3
[] & [a,-a V b] => [] & [b]                                    by rule_2
First branch :  prove,
[a] & [-a] => [] & [b]
[a] & [] => [] & [a,b]                                         by rule_2
Tautology.
This branch has been proved.
Second branch :  prove,
[a] & [b] => [] & [b]
Tautology.
This branch has been proved.
Formula is valid.
```

PROBLEM 68

Verbal statement:

The facts:

The maid said that she saw the butler in the living room.
The living room adjoins the kitchen.
The shot was fired in the kitchen, and could be heard in all nearby rooms.
The butler, who had good hearing, said he did not hear the shot.

To prove:

If the maid told the truth, the butler lied.

Logic program:

Use the previous program for Wang's algorithm.

Execution:

Use the following representation:

p = The maid told the truth
q = The butler was in the living room
r = The butler was near the kitchen
s = The butler heard the shot
u = The butler told the truth

The premises are written:

-p V q (If the maid told the truth, the butler was in the living room.)

-q V r (If the butler was in the living room, he was near the kitchen.)

-r V s (If he was near the kitchen, he heard the shot.)

-u V -s (If he told the truth, he did not hear the shot.)

The theorem is written:

-p V -u (If the maid told the truth, the butler did not.)

The formula input to the program is:

(-P V Q) & (-Q V R) & (-R V S) & (-U V S) => (-P V -U)

PROBLEM 69

Verbal statement:

The facts:

The following chemical equations:

MgO+H2 --> Mg + H2O
C+O2 --> CO2
CO2+H2O --> H2CO3

The existing components:

MgO,H2,O2 and C

To prove:

It is possible to compose H2CO3.

Logic program:

Use any of the previous Propositional Calculus theorem provers.

Execution:

The premises are written:

(MgO & H2) => (Mg & H2O)
(C & O2) => CO2
(CO2 & H2O) => H2CO3

The theorem is written:

(((MgO & H2) => (Mg & H2O))
& ((C & O2) => CO2)
& ((CO2 & H2O) => H2CO3) & MgO & H2 & O2 &C) => H2CO3

PROBLEM 70 [EMDEN,1976a]

Verbal statement:

Consider a machine defined as a set of commands, each of which is a binary relation over some supposedly given set of states.

In the set notation for the commands of this example, the variables u,v,w,u',v',w' are considered to range over the rational numbers.

The set of states = {(u,v,w)} U {(u,v)} U {(w)}

The commands : (Figure 5)

set notation for command	"Algol" notation for command
{((u,v,w),(u,v',w')): v'=v-1 & w'=u × w}	v,w:=v-1,u × w
{((u,v,w),(u',v',w)): u'=u × u & v'=v/2}	u,v:=u × u,v/2
(E.W. Dijkstra's "parallel assignment")	
{((u,v),(u,v,1))}	real w:=1
("initializing declaration")	
{((u,v,w),(w))}	und u,v
("undeclaration")	
{((u,0,w),(u,0,w))}	v=0
{((u,v,w),(u,v,w)): ¬v=0}	¬v=0
("guard")	
{((u,v,w),(u',v',w)):even(v) & u'=u × u & v'=v/2}	even(v);u,v:=u × u,v/2
{((u,0,w),(w))}	v=0;und u,v
("guarded command")	

Figure 5

The machine : some set closed under product containing the above commands

The program schema =

(nonterminals: { S,P,Q }

,terminals: { W1,V0,UV,VW }

```
,productions:  { S --> W1 P
               ,P --> V0
               ,P --> UV P
               ,P --> VW P
               }
,start symbol:  S
)
```

The program is obtained by the following identifications:

W1 = (real w:=1)

V0 = (v=0; nod u,v)

UV = (even(v); u,v:=u × u,v/2)

VW = (v,w:=v-1,u × w)

Some "computations" : (Figure 6)

	computation 1		computation 2		computation 3	
i	t_i	x_i	t_i	x_i	t_i	x_i
0	--	(2,10)	--	(2,10)	--	(2,10)
1	W1	(2,10,1)	W1	(2,10,1)	W1	(2,10,1)
2	UV	(4,5,1)	UV	(4,5,1)	UV	(4,5,1)
3	VW	(4,4,4)	VW	(4,4,4)	VW	(4,4,4)
4	UV	(16,2,4)	UV	(16,2,4)	UV	(16,2,4)
5	UV	(256,1,4)	VW	(16,1,64)	VW	(16,1,64)
6	VW	(256,0,1024)	VW	(16,0,1024)	VW	(16,0,1024)
7	V0	(1024)	V0	(1024)	VW	$(16,-1,2^{-14})$
8					VW	$(16,-1,2^{-18})$
						⋮
						ad inf

Figure 6

Two representations of this machine, the flowgraph (Figure 7) and the flowdiagram (Figure 8), are given.

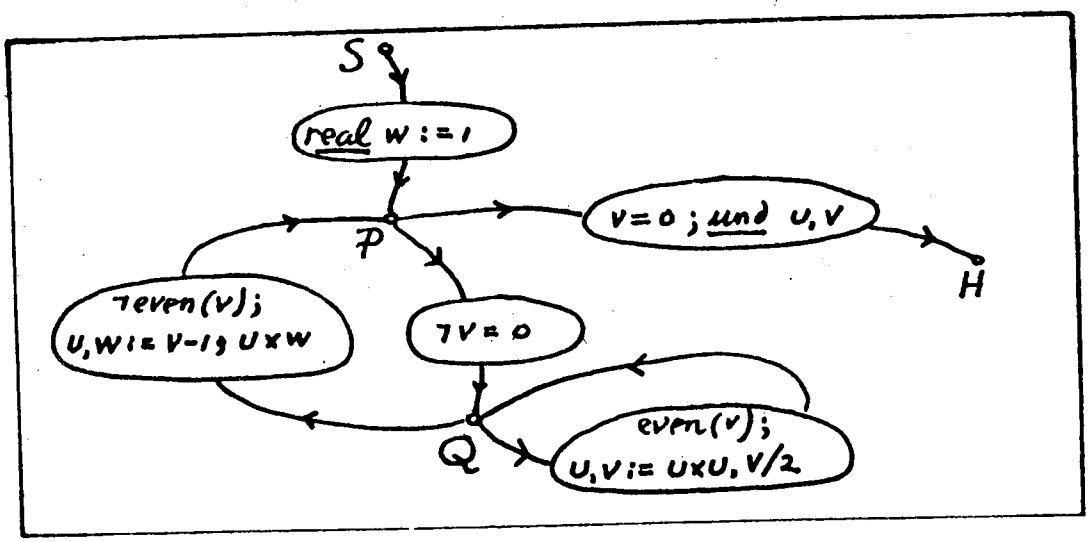


Figure 7

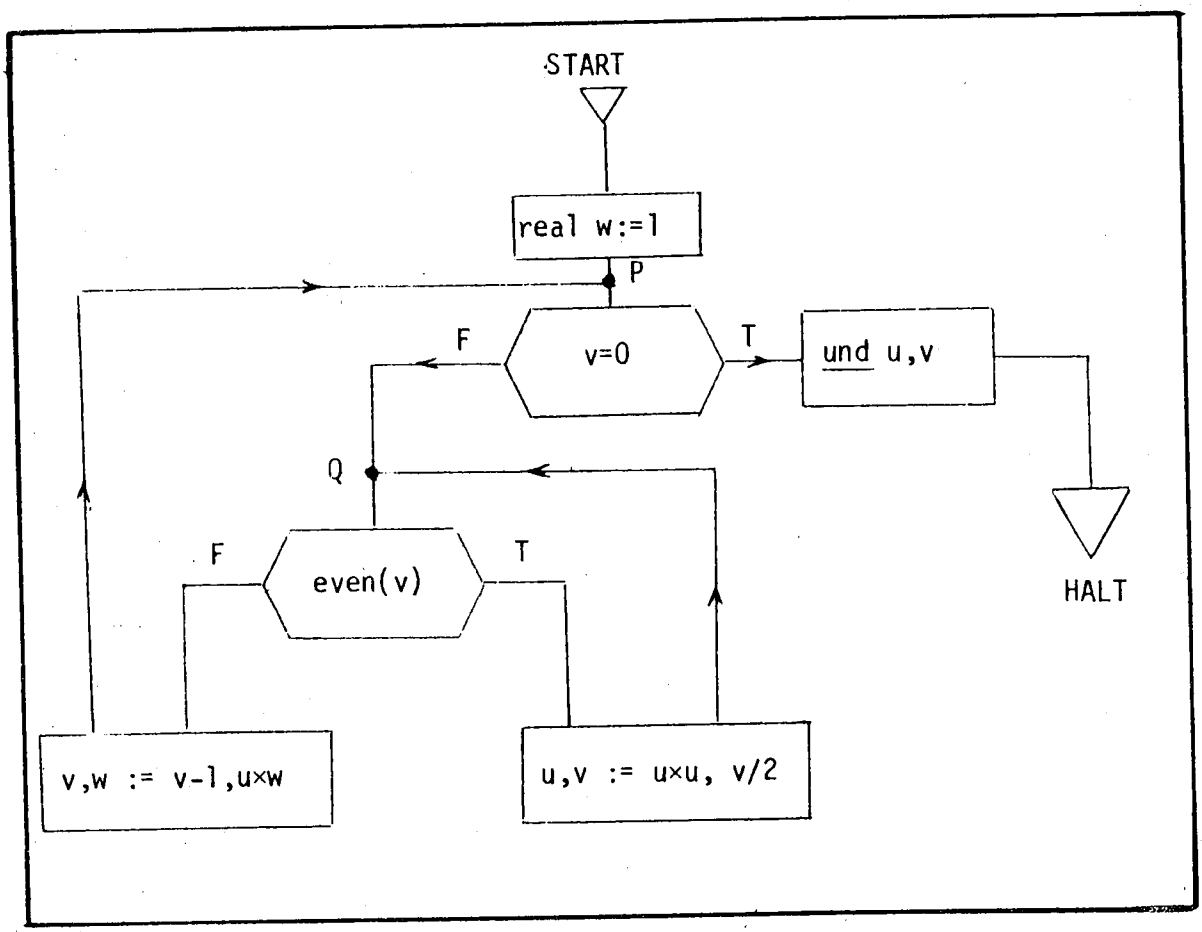


Figure 8

Write a program implementing this machine.

Logic program:

```
:-op(600,yfx,'.').

s(X):- w1(X,Y),p(Y,Z),write(Z),nl.

p(X,Y):- v0(X,Y).
p(X,Z):- uv(X,Y),p(Y,Z).
p(X,Z):- vw(X,Y),p(Y,Z).

w1(U.V,U.V.1).

vo(U.0.W,W).

even(V):- V1 is V mod 2, V1==0.

uv(U.V.W,U1.V1.W):- even(V),
                    U1 is U*U,
                    V1 is V/2.

vw(U.V.W,U.V1.W1):- V1 is V-1,
                    W1 is U*W.
```

Execution:

```
?-s(2.10).

1024
```

3.5 PLANNING (nos. 71 to 79)

PROBLEM 71 [WARREN,1974a]

Verbal statement:

Many problem domains can naturally be formalised as a world with a set of actions which transform that world from one state to another.

A particular problem is then specified by describing an initial state and a desired goal state, the problem solver is required to generate a plan, a simple sequence of actions which transforms the world from the initial state to the goal state.

Write a problem solver (WARPLAN), independent from any particular problem and able to generate a plan of actions (in order to transform an initial state into a goal state) for each data base (world description) you give it.

Logic program:

```
:-op(700,xfy,&).
:-op(650,yfx,=>).


/* Problem solver entry :  Generation and output of a plan */

plans(C,T):- not(consistent(C,true)),!,nl,
             write('impossible'),nl,nl.
plans(C,T):- plan(C,true,T,T1),nl,output(T1),nl,nl.

output(Xs=>X):- !,output1(Xs),write(X),write('.'),nl.

output1(Xs=>X):- !,output1(Xs),write(X),write(' ;'),nl.
output1(X):- write(X),write(' ;'),nl.


/* Entry point to the main recursive loop */

plan(X&C,P,T,T2):- !,solve(X,P,T,P1,T1)plan(C,P1,T1,T2).
plan(X,P,T,T1):- solve(X,P,T,P1,T1).


/* Ways of solving a goal */

solve(X,P,T,P,T):- always(X) ;  X.
solve(X,P,T,P1,T):- holds(X,T),and(X,P,P1).
solve(X,P,T,X&P,T1):- add(X,U),achieve(X,U,P,T,T1).
```

```
/* Methods of achieving an action */

/* By extension */

achieve(X,U,P,T,T1=>U):- preserves(U,P),
                         can(U,C),
                         consistent(C,P),
                         plan(C,P,T,T1),
                         preserves(U,P).

/* By insertion */

achieve(X,U,P,T=>V,T1=>V):- preserved(X,V),
                            retrace(P,V,P1),
                            achieve(X,U,P1,T,T1),
                            preserved(X,V).

/* Check if a fact holds in a certain state */

holds(X,T=>V):- add(X,V).
holds(X,T=>V):- !,preserved(X,V),
                holds(X,T),
                preserved(X,V).
holds(X,T):-given(T,X).


/* Prove that an action preserves a fact */

preserves(U,X&C):- preserved(X,U),preserves(U,C).
preserves(_,true).

preserved(X,V):- numbervars(X&V,0,N),
                 del(X,V),!,fail.
preserved(_,true).


/* Retracing a goal already achieved */

retrace(P,V,P2):- can(V,C),
                  retrace1(P,V,C,P1),
                  append(C,P1,P2).

retrace1(X&P,V,C,P1):- add(Y,V),equiv(X,Y),!,
                       retrace1(P,V,C,P1).
retrace1(X&P,V,C,P1):- elem(Y,C),equiv(X,Y),!,
                       retrace1(P,V,C,P1).
retrace1(true,V,C,true).

/* Consistency with a goal already achieved */

consistent(C,P):- numbervars(C&P,0,N),
                  imposs(S),
                  not(not(intersect(C,S))),
                  implied(S,C&P),!,fail.
consistent(_,_).
```

```
/* Utility routines */

and(X,P,P):- elem(Y,P),equiv(X,Y),!.
and(X,P,X&P).

append(X&C,P,X&P1):- !,append(C,P,P1).
append(X,P,X&P).

elem(X,Y&C):- elem(X,Y).
elem(X,Y&C):-!,elem(X,C).
elem(X,X).

implied(S1&S2,C):- !,implied(S1,C),implied(S2,C).
implied(X,C):- elem(X,C).
implied(X,C):- X.

intersect(S1,S2):- elem(X,S1),elem(X,S2).

not_equal(X,Y):- not(X=Y),
                 not(X='$VAR'(_)),
                 not(Y='$VAR'(_)).

equiv(X,Y):- not(nonequiv(X,Y)).

nonequiv(X,Y):- numbervars(X&Y,0,N),X=Y,!,fail.
nonequiv(_,_).
```

Remarks on the logical program :

0) Operators interpretation:

   'X&Y' is 'X and Y'
   'X=>Y' is 'the state after doing Y in X'

1) The central predicate is "plan(C,P,T,T1)", it has 4 arguments:

   C is a conjunction of goals to be solved;

   T is a (already generated) partial plan;

   P is a conjunction of goals already solved by T which must be protected;

   T1 is a new plan, which contain T as a subplan and preserves the already solved goals P, and which also solves the new goals C.

   Essencialy this predicate states that a plan can be produced by 'solve'-ing each goal in the given order (C,P,T behave as input variables and T1 as output variable).

2) The predicate 'solve(X,P,T,P1,T1)', has the following arguments:

X is an atomic goal;

T is a partial plan;

P is a conjunction of goals achieved by T;

T1 is a plan, containing T as a subplan, which solves P1;

P1 is a conjunction comprising P and X, where X is not repeated.

There are three ways in which a goal may be 'solve'-d : It may be 'always' true in the world. It may be that it already 'holds' in the state produced by the current partial plan. Finally we may look in the database for action U which 'add'-s the goal X and then 'solve' X by 'achieve'-ing U (X,P,T behave as input variables and P,T1 as output variables).

3) The predicate 'achieve(X,U,P,T,T1)' has two clauses corresponding to two methods of 'achieve'-ing an action : extension and insertion.

The clause for the extension method (the first one) checks that the action U 'preserves' (ie, does not delete) the protected facts P. Then we lookup the preconditions C in the database and check that C is consistent with the protected facts. All being well, we call 'plan' recursivelly to modify the current plan T to a new T1 which produces a state in which C and P are attained. U can then be applied in T1, corresponding to the plan resulting from this call of 'achieve'.

Finally, the check that U preserves P is repeated, for the reason that U and P may not have been instantiated to ground terms at the time of the original check.

The clause for the insertion method (the second one) follows the axiom: If the last action V in the current partial plan does not delete the current goal X, we can try to insert the action U somewhere before V, provided we retrace the set of protected facts to the point before V.

4) 'holds(X,Y)' follows the method: everything that 'holds' in a state of the world can be determined from the plan which produces that state of the world. The system chains backwards through the sequence of actions, so long as none of these actions deletes the sought-for facts, until the fact is found in the 'add'-set of an action or was given in the initial state.

PROBLEM 72 [WARREN,1974a]

Verbal statement:

Using the problem solver WARPLAN, introduced in the previous problem, write the world description and the initial state for the 3 blocks shown in Figure 9.

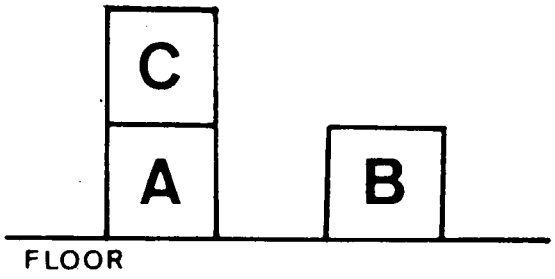


Figure 9

Achieve the goal state on(a,b)&on(b,c).

Sugestion:

Use the predicates

add(X,U)	with the meaning	fact X is added by action U; ie. X is true in any state resulting from U (and U is a possible action in some state in which X is not true)
del(X,U)	"	fact X is deleted by action U; ie. it is not the case that X is preserved by U. (X is preserved by U if and only if X is not added by U and X is true in a state resulting from U whenever X is true in the preceding state)
can(U,C)	"	the conjunction of facts C is the precondition of action U; ie. U is possible in any state in which C is true.
always(X)	"	fact X is true in any state
imposs(X)	"	the conjunction of facts X is impossible in any state

given(T,X) with the meaning fact X is true in the initial state T (but it is not the case that X is true in all states).

Logic program:

In order to use WARPLAN we have just to write the data base according to the block situation above.

```
/* Data Base for the three blocks problem */


add(on(U,W),move(U,V,W)).
add(clear(V),move(U,V,W)).

del(on(U,Z),move(U,V,W)).
del(clear(W),move(U,V,W)).

can(move(U,V,floor),on(U,V)&not_equal(V,floor)&clear(U)).
can(move(U,V,W),clear(W)&on(U,V)&not_equal(U,W)&clear(U)).

impos(on(X,Y)&clear(Y)).
impos(on(X,Y)&on(X,Z)&not_equal(Y,Z)).
impos(on(X,X)).

given(start,on(a,floor)).
given(start,on(b,floor)).
given(start,on(c,a)).
given(start,clear(b)).
given(start,clear(c)).
```

Execution:

After adding this database to WARPLAN you just have to give the command:

```
:-plans(on(a,b)&on(b,c),start).
```

the solution is:

```
    start ;
    move(c,a,floor) ;
    move(b,floor,c) ;
    move(a,floor,b) .
```

PROBLEM 73

Verbal statement:

Consider the initial state of 5 blocks as shown in Figure 10 below, apply the program WARPLAN, presented in PROBLEM 63, to achieve the goal state :

on(a,b)&on(b,c)&on(c,d)&on(d,e),

as it was done in the preceding problem.

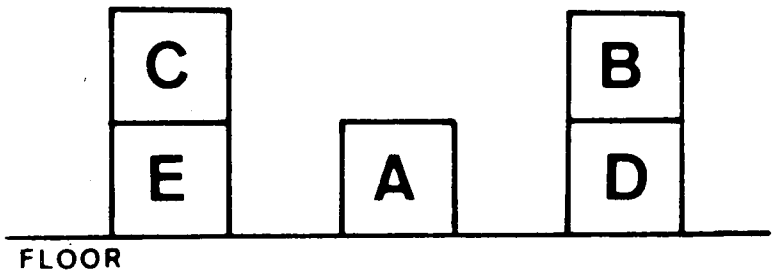


Figure 10

Logic program:

Same facts 'add' , 'del' , 'can' , and 'imposs' as those in the preceding problem.

The description of the initial state is:

```
/* Data Base for the five blocks problem */

given(start,on(e,floor)).
given(start,on(c,e)).
given(start,clear(c)).
given(start,on(a,floor)).
given(start,clear(a)).
given(start,on(d,floor)).
given(start,on(b,d)).
given(start,clear(b)).
```

Execution:

The command ":- plans(on(a,b)&on(b,c)&on(c,d)&on(d,e),start)."

runs the WARPLAN program and produces:

```
    start ;
    move(b,d,floor) ;
    move(c,e,floor) ;
    move(d,floor,e) ;
    move(c,floor,d) ;
    move(b,floor,c) ;
    move(a,floor,b) .
```

PROBLEM 74 [WARREN,1974a]

Verbal statement:

Consider a world comprising two areas 'inside' and 'outside'. In the area 'inside' there are four distinct locations, namely 'table', 'box1' , 'box2' , 'door' . There is a robot which is able to move about and transport objects. If the robot attempts to pickup an object at a location, all that can be ascertained is that it will be holding one of the objects, if any, at that location. The robot is only allowed to pickup an object if it is the only object at a location. Consider 3 objects 'key1' , 'key2' and 'red1' and two actions 'go' and 'take'. The robot is only allowed to 'go' or 'take' something 'outside' if both the objects 'key1' and 'key2' are at the 'door'.

Consider the initial state, given in Figure 11 :

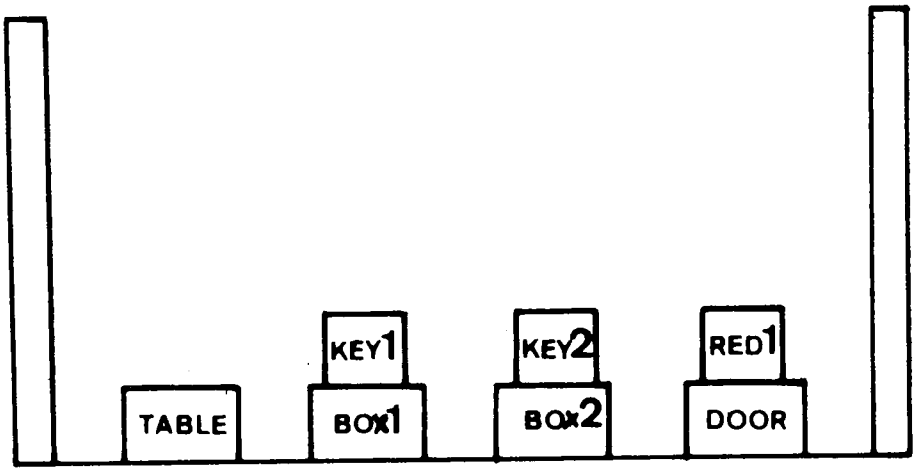


Figure 11

Use the program WARPLAN to achieve the goal : to have 'red1' outside.

Logic program:

```
:-op(300,yfx,'IS').
:-op(200,fx,'set ').
:-op(200,fx,'placed ').
:-op(200,fx,'only ').
:-op(100,yfx,'at').

add(position 'IS' P,go(P)).
add(X 'IS' placed Q,take(X,P,Q).
add(set placed P 'IS' nothing,take(X,P,Q)).
add(set placed Q 'IS' only X,take1(X,P,Q)).
```

```
add(Z,take1(X,P,Q)):- add(Z,take(X,P,Q)).

del(position 'IS' Z,go(P)).
del(X 'IS' placed Z,take(X,P,Q)).
del(position 'IS' Z,take(X,P,Q,)).
del(set placed Q 'IS' Z,take(X,P,Q)).
del(set placed P 'IS' Z,take(X,P,Q)).

del(Z,take1(X,P,Q)):- del(Z,take(X,P,Q)).

can(go(inside at L),true).
can(take(X,inside at L1,inside at L2),
          set placed inside at L1 'IS' only X &
          position 'IS' inside at L1).
can(take1(X,inside at L1, inside at L2),
          set placed inside at L2 'IS' nothing &
          set placed inside at L1 'IS' only X &
          position 'IS' inside at L1).
can(take(X,inside at L,outside),
          set placed inside at L 'IS' only X &
          position 'IS' inside at L &
          key1 'IS' placed inside at door &
          key2 'IS' placed inside at door).

given(start,set placed inside at table 'IS' nothing).
given(start,set placed inside at box1 'IS' only key1).
given(start, set placed inside at box2 'IS' only key2).
given(start,set placed inside at door 'IS' only red1).

imposs(position 'IS' P & position 'IS' Q & not_equal(P,Q)).
```

Execution:

The command ":- plans(red1 'IS' placed outside,start)." runs WARPLAN after having added this data base.

The answer is:

```
    start ;
    go(inside at door) ;
    take1(red1, inside at door, inside at table) ;
    go(inside at box1) ;
    take(key1, inside at box1, inside at door) ;
    go(inside at box2) ;
    take(key2, inside at box2, inside at door) ;
    go(inside at table) ;
    take(red1, inside at table, outside) .
```

Remark:

The reason for the two versions of 'take' is that an action must have a unique set of preconditions, and the effects of 'take' depend on the set of objects at the destination. (See clauses 'can' in the logic program).

PROBLEM 75 [WARREN,1974a]

Verbal statement:

Consider a very simple computer, comprising an accumulator and an unspecified number of general purpose registers. There are just four instructions 'load' , 'store' , 'add' and 'subtract'. To axiomatise the domain in order to use the program WARPLAN, presented in PROBLEM 63, it is necessary to follow each instruction in an assembly language program by a comment. The comment is introduced by '#' and states the value which will be in the accumulator after the instruction has been executed.

Describe such a machine and make it solve:

(1) acc is (c1-c2) + (c3-c4)

(2) acc is (c1-c2) + (c1-c2)

(3) acc is c1 + (c2-c3) &
reg2 is c2-c3 &
reg3 is c4 + c4

(4) reg1 is c1 + (c2-c3) &
reg2 is c2-c3 &
acc is c1

from the initial state : reg1 is c1
reg2 is c2
reg3 is c3
reg4 is c4

Logic program:

```
:-op(250,yfx,'#').
:-op(250,yfx,'IS').
:-op(150,xfy,'+').
:-op(150,xfy,'-').
:-op(150,fx,'load').
:-op(250,fx,'add').
:-op(150,fx,'subtract').
:-op(150,fx,'store').
:-op(150,fx,'reg').

/* Machine code generation */

add(acc 'IS' V1+V2, add R # V1+V2).
add(acc 'IS' V1-V2, subtract R # V1-V2).
add(acc 'IS' V, load R # V).
add(reg R 'IS' V, store R #V).

del(acc 'IS' Z,U:- add(acc 'IS' V,U).
del(reg R 'IS' Z,U):- add(reg R 'IS' V,U).
```

```
can(load R # V, reg R 'IS' V).
can(store R # V, acc 'IS' V).
can(add R # V1+V2, reg R 'IS' V2 & acc 'IS' V1).
can(subtract R # V1-V2,reg R 'IS' V2 & acc 'IS' V1).

given(start,reg1 'IS' c1).
given(start,reg2 'IS' c2).
given(start,reg3 'IS' c3).
given(start,reg4 'IS' c4).
```

Execution:

```
(1) :- plans(acc 'IS' (c1-c2)+(c3-c4),start).
```

Answer:

```
        start ;
        load 3 # c3 ;
        subtract 4 # c3-c4 ;
        store X1 # c3-c4 ;
        load 1 # c1;
        subtract 2 # c1-c2 ;
        add X1 # (c1-c2)+(c3-c4) .
```

```
(2) :- plans(acc 'IS' (c1-c2)+(c1-c2),start).
```

Answer:

```
        start ;
        load1 # c1
        subtract2 # c1-c2
        store X1 # c1-c2
        add X1 # (c1-c2)+(c1-c2) .
```

```
(3) :- plans(reg1 'IS' c1+(c2-c3) &
             reg2 'IS' c2-c3 &
             reg3 'IS' c4+c4,start).
```

Answer:

```
        start ;
        load2 # c2;
        subtract 3 # c2-c3 ;
        store 2 # c2-c3 ;
        load 1 # c1 ;
        add 2 # c1+(c2-c3) ;
        store 1 # c1+(c2-c3) ;
        load 4 # c4;
        add 4 # c4+c4 ;
        store 3 # c4+c4 .
```

```
(4) :- plans(res1 'IS' c1+(c2-c3) &
             res2 'IS' c2-c3 &
             acc 'IS' c1,start).
```

Answer:

```
start ;
load 2 # c2;
subtract 3 # c2-c3 ;
store 2 # c2-c3 ;
load 1 # c1 ;
store X1 # c1 ;
add 2 # c1+(c2-c3) ;
store 1 # c1+(c2-c3) ;
load X1 # c1 .
```

Remark: The first branch in WARPLAN'S search space for question (4) is infinite. Interactive intervention to block this branch resulted in the above solution being found without any further assistance.

PROBLEM 76 [WARREN,1974a]

Verbal statement:

Consider the world presented in Figure 12, belonging to STRIPS1.

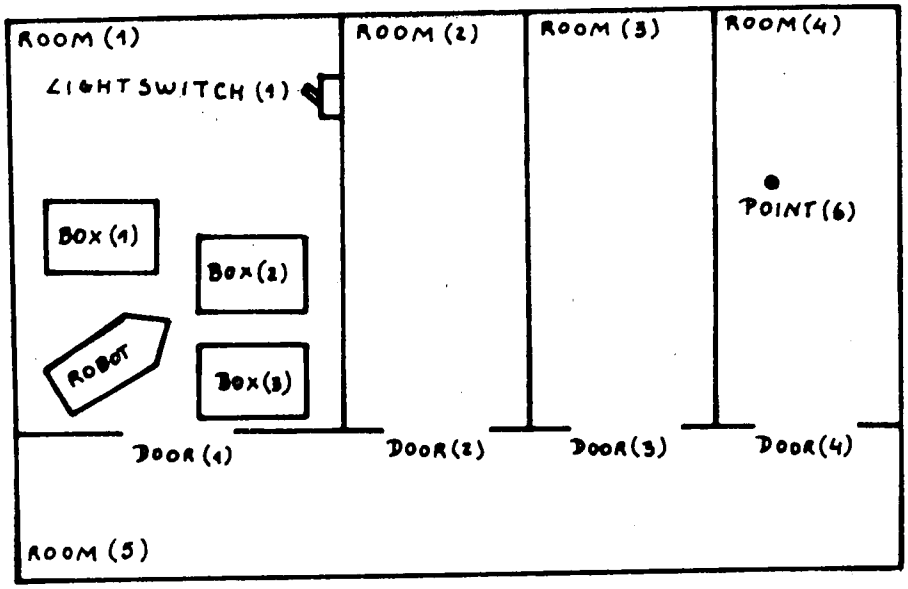


Figure12

Describe it and use WARPLAN to achieve the goals:

(1) status(lightswitch(1),on)

(2) nextto(box(1),box(2))&nextto(box(2),box(3))

```
(3) at(robot,point(6))

(4) nextto(box(2),box(3))&nextto(box(3),door(1))&
            status(lightswitch(1),on)&nextto(box(1),box(2))&
            inroom(robot,room(2))
```

Logic program:

```
/* Data Base for the STRIPS1 problem */


add(at(robot,P),goto1(P,R)).
add(nextto(robot,X),goto2(X,R)).
add(nextto(X,Y),pushto(X,Y,R)).
add(nextto(Y,X),pushto(X,Y,R)).
add(status(S,on),turnon(S)).
add(on(robot,B),climbon(B)).
add(onfloor,climboff(B)).
add(inroom(robot,R2),gothru(D,R1,R2)).

del(at(X,Z),U):- moved(X,U).
del(nextto(Z,robot),U):- !,del(nextto(robot,Z),U).
del(nextto(robot,X),pushto(X,Y,R)):- !, fail.
del(nextto(robot,B),climbon(B)):- !,fail.
del(nextto(robot,B),climboff(B)):- !,fail.
del(nextto(X,Z),U):- moved(X,U).
del(nextto(Z,X),U):- moved(X,U).
del(on(X,Z),U):- moved(X,U).
del(onfloor,climbon(B)).
del(inroom(robot,Z),gothru(D,R1,R2)).
del(status(S,Z),turnon(S)).

moved(robot,goto1(P,R)).
moved(robot,goto2(X,R)).
moved(robot,pushto(X,Y,R)).
moved(X,pushto(X,Y,R)).
moved(robot,climbon(B)).
moved(robot,climboff(B)).
moved(robot,gothru(D,R1,R2)).

can(goto1(P,R),locinroom(P,R)&inroom(robot,R)&onfloor).
can(goto2(X,R),inroom(X,R)&inroom(robot,R)&onfloor).
can(pushto(X,Y,R),pushable(X)&inroom(Y,R)&inroom(X,R)&
                    nextto(robot,X)&onfloor).
can(turnon(lightswitch(S)),on(robot,box(1))&
                           nextto(box(1),lightswitch(S))).
can(climbon(box(B)),nexto(robot,box(B))&onfloor).
can(climboff(box(B)),on(robot,box(B))).
can(gothru(D,R1,R2),connects(D,R1,R2)&inroom(robot,R1)&
                     nextto(robot,D)&onfloor).

always(connects(D,R1,R2)):- connects1(D,R1,R2).
always(connects(D,R2,R1)):- connects1(D,R1,R2).
always(inroom(D,R1)):- always(connects(D,R0,R1)).
```

```
always(pushable(box(N))).
always(locinroom(point(6),room(4))).
always(inroom(lightswitch(1),room(1))).
always(at(lightswitch(1),point(4))).

connects1(door(N),room(N),room(5)):- range(N,1,4).

range(M,M,N).
range(M,L,N):- not_equal(L,N), L1 is L+1, range(M,L1,N).

given(strips1,at(box(N),point(N))):- range(N,1,3).
given(strips1,at(robot,point(5))).
given(strips1,inroom(box(N),room(1))):- range(N,1,3).
given(strips1,inroom(robot,room(1))).
given(strips1,onfloor).
given(strips1,status(lightswitch(1),off)).


Execution:


(1) :- plans(status(lightswitch(1),on),start).

Answer:

        start ;
        goto2(box(1),room(1)) ;
        pushto(box(1),lightswitch(1),room(1)) ;
        climbon(box(1)) ;
        turnon(lightswitch(1)) .


(2) :- plans(nextto(box(1),box(2) &
              nexto(box(2),box(3)),start).

Answer:

        start ;
        goto2(box(2),room(1)) ;
        pushto(box(2),box(3),room(1)) ;
        goto2(box(1),room(1)) ;
        pushto(box(1),box(2),room(1)) .


(3) :- plans(at(robot,point(6)),start).

Answer:

        start ;
        goto2(door(1),room(1)) ;
        gothru(door(1),room(1),room(5)) ;
        goto2(door(4),room(5)) ;
        gothru(door(4),room(5),room(4)) ;
        goto1(point(6),room(4)) .
```

```
(4) :- plans(nextto(box(2),box(3)) &
             nextto(box(3),door(1)) &
             status(lightswitch(1),on) &
             nextto(box(1),box(2)) &
             inroom(robot,room(2)),start).

Answer:

        start ;
        goto2(box(3),room(1)) ;
        pushto(box(3),door(1),room(1)) ;
        goto2(box(2),room(1)) ;
        pushto(box(2),box(3),room(1)) ;
        goto2(box(1),room(1)) ;
        pushto(box(1),lightswitch(1),room(1)) ;
        climbon(box(1)) ;
        turnon(lightswitch(1)) ;
        climboff(box(1)) ;
        goto2(box(1),room(1)) ;
        pushto(box(1),box(2),room(1)) ;
        goto2(door(1),room(1)) ;
        gothru(door(1),room(1),room(5)) ;
        goto2(door(2),room(5)) ;
        gothru(door(2),room(5),room(2)) .
```

PROBLEM 77 [WARREN,1975b]

Verbal statement:

Consider a world consisting of the following objects, facts and actions:

Objects:

```
numbers,     N = <1,2,...etc>
directions,  D = <left,right>
wheels,      W = <wheel1, wheel2,...,wheelN>
axles,       A = <axle1,...,axle N>
endofaxles,  E = <D1endof A1> where D1 D and A1 A
holes,       H = <hole1,...,hole N>
vice,        V = <vice>
```

Facts:

```
 Static, D1 is opposite of D2,          where D1,D2∈D

Dynamic, W1 is attached to E1,          where W1 ∈ W and E1 ∈ E
Dynamic, A1 is thru H1,                 where A1 ∈ A and H1 ∈ H
Dynamic, E1 points D1,                  where E1 ∈ E and D1 ∈ D
Dynamic, carbody is blocked to D1,      where D1 ∈ D
Dynamic, carbody is unblocked to D1,    where D1 ∈ D
Dynamic, W1 + A1 is clamped,            where W1 ∈ W and A1 ∈ A
Dynamic, W1+A1+H1+V1 is free,           where W1 ∈ W,A1 ∈ A,H1 ∈ H
                                        and V1 ∈ V
```

Actions:

insert E1 into W1,	where E1 ∈ E and W1 ∈ W
push W1 from D1 to D2 onto E1 in H1,	where W1 ∈ W, H1 ∈ H, E1 ∈ E
slide E1 into H1 from D1,	where E1 ∈ E, H1 ∈ H and D1 ∈ D
block carbody to D1,	where D1 ∈ D
unblock carbody to D1,	where D1 ∈ D
clamp W1,	where W1 ∈ W
unclamp W1+A1,	where W1 ∈ W and A1 ∈ A

the actions may be pictured as follows (Figures 13 and 14) :

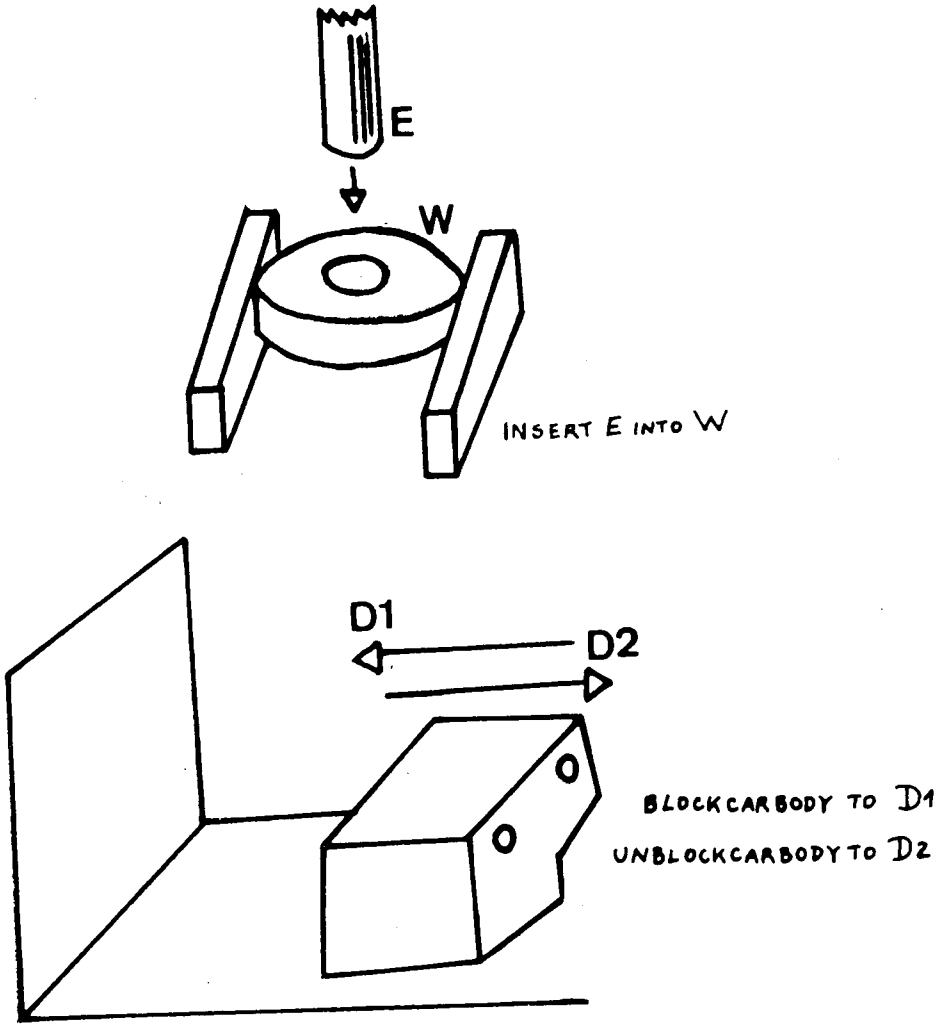
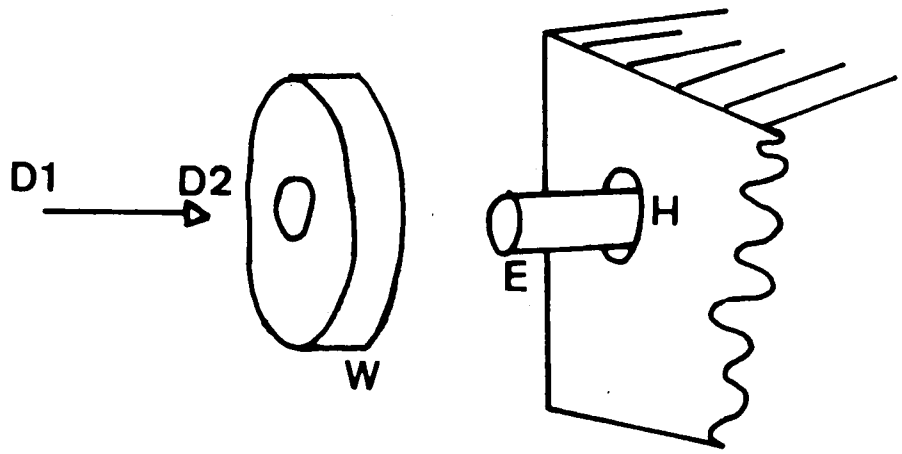
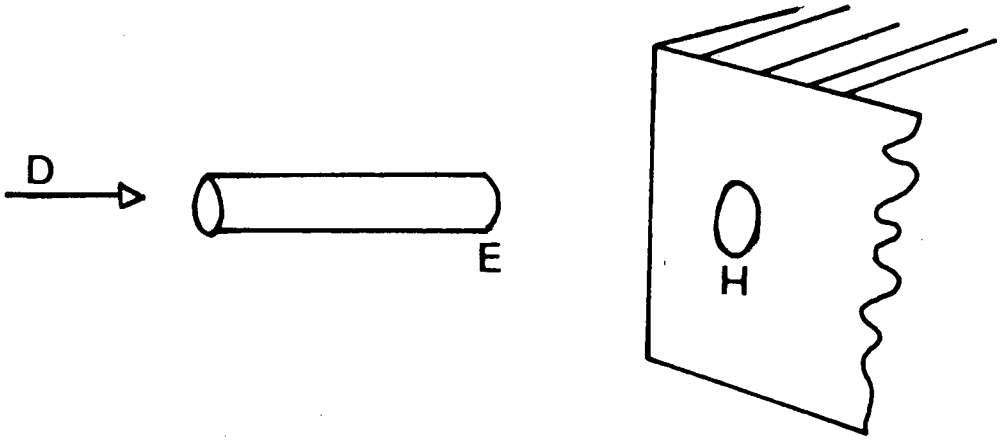


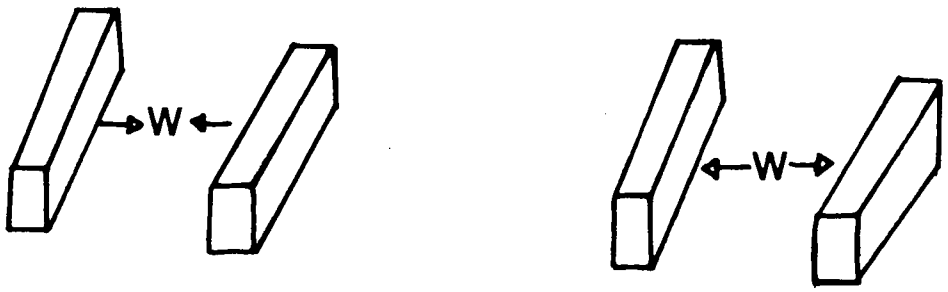
Figure 13



PUSH W FROM D1 TO D2 ONTO E IN H



SLIDE E INTO H FROM D



CLAMP W

UNCLAMP W

Fisure 14

Describe this world as a data base in order to use WARPLAN to solve car assembly problems.

Logic program:

```
/* Operators definition */

:-op(600,fx,'insert').
:-op(600,fx,'push').
:-op(600,fx,'slide').
:-op(600,fx,'clamp').
:-op(600,fx,'unclamp').
:-op(600,fx,'block_car_body_to').
:-op(600,fx,'unblock_car_body_to').
:-op(600,fx,'car_body_is_unblocked_to').
:-op(600,fx,'car_body_is_blocked_to').

:-op(600,xf,'is_free').
:-op(600,xf,'is_clamped').

:-op(500,xfy,'is_thru').
:-op(500,xfy,'points').
:-op(500,xfy,'is_opposite_of').
:-op(500,xfy,'is_attached_to').

:-op(400,yfx,'into').
:-op(400,yfx,'from').
:-op(400,yfx,'to').
:-op(400,yfx,'onto').
:-op(400,yfx,'in').

:-op(300,xfy,'end_of').

:-op(200,fx,'wheel').
:-op(200,fx,'axle').
:-op(200,fx,'hole').

/* Definition of the goal state */

goal(T):- plans(axle A1 is_thru hole1 &
                axle A2 is_thru hole2 &
                wheel W1 is_attached_to left endof axle A1 &
                wheel W2 is_attached_to left end_of axle A2 &
                wheel W3 is_attached_to eisht end_of axle A1 &
                wheel W4 is_attached_to risht end_of axle A2,T).


/* Data base */

add(W is_attached_to E, insert E into W).
add(W is_attached_to E, push W from D1 to D2 onto E in H).
add(A is_thru H, slide D1 end_of A into H from D2).
add(wheel W is_clamped, clamp wheel W).
add(axle A is_clamped, insert D end_of axle A into W).
add(wheel W is_free, unclamp wheel W).
add(axle A is_free, unclamp axle A).
add(vice is_free, unclamp X).
add(car_body_is_blocked_to D, block_car_body_to D).
```

```
add(car_body_is_unblocked_to D, unblock_car_body_to D).
add(D1 end_of A points D, slide D1 end_of A into H from D2):-
                                        always(D is_opposite_of D2).
add(d end_of A points D2, slide D1 end_of A into H from D2):-
                                        always(D is_opposire_of D1).


can(insert D end_of axle A into wheel W, axle A is_free &
                                          D end_of axle A is_free &
                                          wheel W is_clamped).
can(push wheel W from D1 to D2 onto D end_of axle A in hole H,
                        wheel W is _free &
                        D end_of axle A is_free &
                        axle A is_thru hole H &
                        D end_of axle A points D1 &
                        car_body_is_unblocked to D1 &
                        D2 is_opposite_of D1 &
                        car_body_is_blocked_to D2).
can(slide D1 end_of axle A into hole H from D2,
                        axle A is_free &
                        D1 end_of axle A is_free &
                        hole H is_free &
                        car_body_is_unblocked_to D2).
can(clamp wheel W, wheel W is_free & vice is_free).
can(unclamp X, X is_clamped).
can(block_car_body_to D, true).
can(unblock_car_body_to D,true).


del(X is_free,U):- add(X is_clamped,U).
del(X is_free,U):- add(X is_attached_to Z,U).
del(X is_free,U):- add(Z is_attached_to X,U).
del(A is_free,slide D1 end_of A into H from D2).
del(H is_free,slide E into H from D2).
del(vice is_free, clamp W).
del(car_body_is_unblocked_to D, block_car_body_to D).
del(car_body_is_blocked_to D, unblock_car_body_to D).
del(Z is_clamped,unclamp Z).
del(W is_clamped, insert E into W).


imposs(axle A is_free & axle A is_thru hole H).
imposs(D end_of A is_free & W is_attached_to D end_of A).
imposs(wheel W is_free & wheel W is_attached_to E).
imposs(wheel W is_free & wheel W is_clamped).
imposs(hole H is_free & A is_thru hole H).
imposs(axle A is_free & axle A is clamped).
imposs(vice is_free & X is_clamped).
imposs(car_body_is_blocked_to D & car_body_is_unblocked_to D).

always(true).
always(left is_opposite_of right).
always(right is_opposite_of left).
```

PROBLEM 78 [WARREN,1975b]

Verbal statement:

Consider the initial state of the car assembly process as it is pictured in Figure 15.

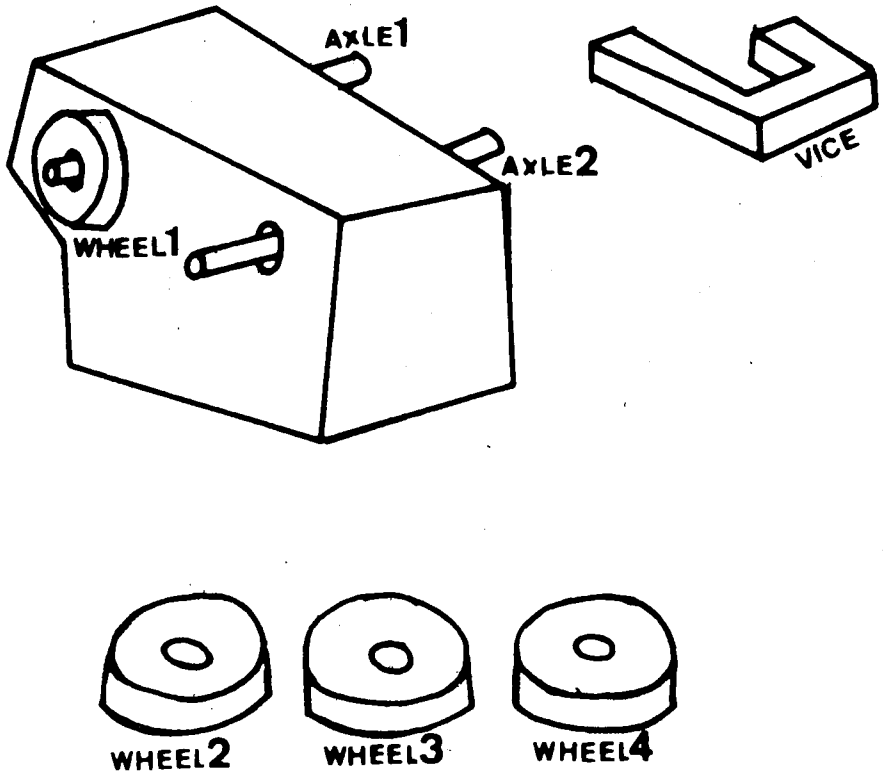


Figure 15

Describe this initial state in order to use the preceding problem and the program WARPLAN to solve the rest of the assembly process.

Logic program:

This state is called the "middle state" just because a small part of the car is already assembled when the process starts.

```
/* Description of the middle state */


given(middle,wheel W is_free):- member(W,[2,3,4]).
given(middle,wheel 1 is_attached_to left end_of axle 1).
given(middle,axle N is_thru hole N):- member(N,[1,2]).
given(middle, D end_of axle A points D):-
                          member(D,[left,right]),
                          member(A,[1,2]).
given(middle,right end_of axle A is_free):- member(A,[1,2]).
given(middle,left end_of axle 2 is_free).
given(middle, vice is_free).
given(middle,car_body_is_unblocked_to D):-
                          member(D,[left,right]).


member(A,[A|_]).
member(A,[_|B]):- member(A,B).
```

Execution:

To run the program WARPLAN with such an initial state and such a data base it is just needed to give the command: ":- goal(middle).".

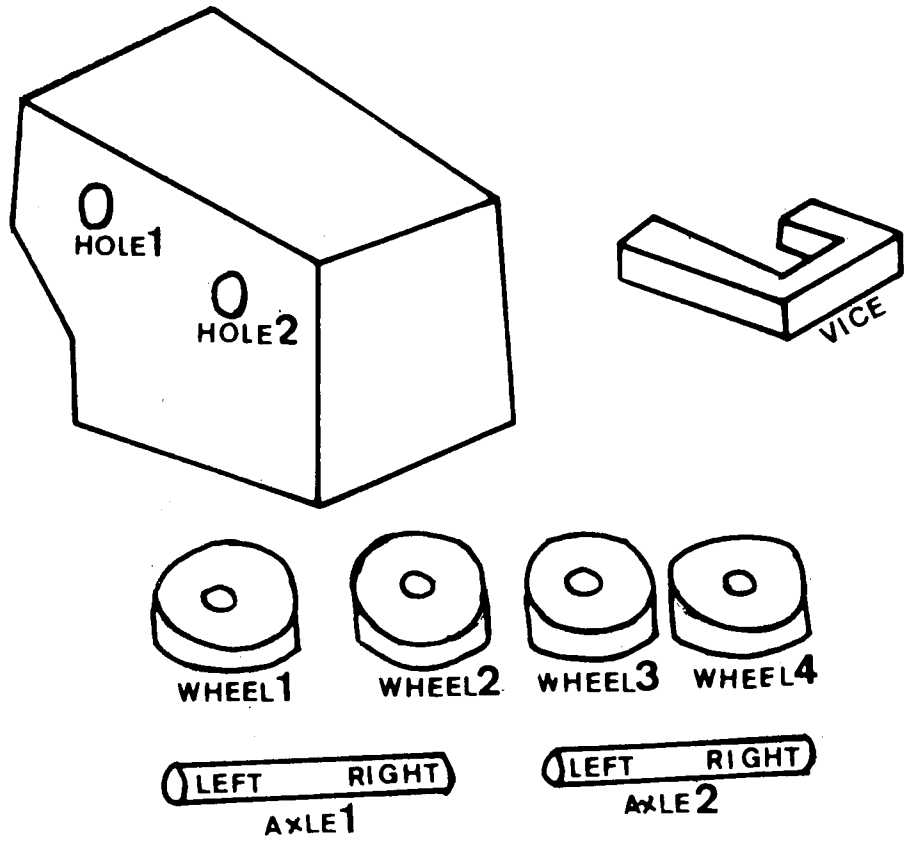
The answer is:

```
    block_car_body_to right ;
    Push wheel2 from left to right onto left end_of  axle  2
    in hole 2 ;
    unblock_car_body to right ;
    block_car_body to left ;
    Push wheel 3 from right to left onto right end_of axle 1
    in hole 1 ;
    Push wheel 4 from right to left onto right end_of axle 2
    in hole 2 .
```

EBOBLEM 79 [WARREN,1975b]

Verbal statement:

Now consider the initial state of the car assembly process as in Figure 16.



Fisure 16

Solve the preceding problem considering this initial state and taking into account the program WARPLAN.

Logic program:

```
/* Description of the start state */

given(start,wheel W is_free):- member(W,[1,2,3,4]).
given(start,axle A is_free):- member(A,[1,2]).
given(start,D end_of axle A is_free):- member(D,[left,right]),
                                       member(A,[1,2]).
given(start,hole H is_free):- member(H,[1,2]).
given(start,vice is_free).
given(start,car_body_is_unblocked_to D):-
                                       member(D,[left,right]).


member(A[A|_]).
member(A,[_|B]):- member(A,B).
```

Execution:

Just give the command: ":- goal(start)." and the answer will be:

```
slide left end_of axle 1 into hole 1 from left ;
slide left end_of axle 2 into hole 2 from left ;
block_car_body_to' left ;
Push wheel 1 from right to left outo left end_of axle  1
in hole 1 ;
Push wheel 2 from right to left outo left end_of axle  2
in hole 2 ;
unblock_car_body_to left ;
block_car_body_to right ;
Push wheel 3 from left to right outo right end_of axle 1
in hole 1 ;
Push wheel 4 from left to right outo right end_of axle 2
in hole 2 .
```

3.6 MACHINE TRANSLATION (nos. 80 to 86)

PROBLEM 80

Verbal statement:

Transform a natural language sentence, taken as a string of characteres with some spaces between them, into a list whose elements are its words.

Susestion: 1) Consider the followins cut marks : 'space' , '?', '.' , '!' , ',' , 'line_feed' and 'CR'.

2) Suppose the sentences may finish by '?', '.' , or '!' .

3) Knowins that PROLOG cannot read word by word, you have to imasine a character by character transformation.

Losic program:

```
sentence(F):- set(C),words(C,F).

words(C,[P|Ps]):- letter(C),word(C,C1,L),
                  name(P,L),words(C1,Ps).
words(44,[',']):- set(C1),words(C1,Ps).
words(63,[?]).
words(46,[.]).
words(33,[!]).
words(_,P):- set(C),words(C,P).

word(C,C1,[C|Cs]):- set0(C2),(letter(C2),word(C2,C1,Cs);
                    C1=C2,Cs=[]).


letter(32):- !, fail.                  /*   space   */
letter(63):- !, fail.                  /*     ?     */
letter(46):- !, fail.                  /*     .     */
letter(33):- !, fail.                  /*     !     */
letter(44):- !, fail.                  /*     ,     */
letter(10):- !, fail.                  /* line_feed */
letter(13):- !, fail.                  /*     CR     */
letter(_).
```

PROBLEM 81

Verbal statement:

In the preceding problem, make the suitable modification in order that capital letters might be transformed into small letters, knowing that the internal codes of capital letter go from 64 to 90 and that the codes of the corresponding small letter can be obtained adding 32.

Logic programs:

```
/* Program 1

Define two new predicates 'set' an 'set0' as follows:  */

set(C,A):- set(C),(C=<90,C>=64,A is C + 32 ;  A=C).
set0(C,A):- set0(C),(C=<90,C>=64,A is C+32;  A=C).

/* Replace the usual calls to 'set' and 'set0' by new calls of
the above predicates, the resulting program will be (we will
write only the new clauses):  */

sentence(F):- set(C,A),words(A,F).

words(44,[',',!Ps]):- set(C1,A),words(A,Ps).
words(_,P):- set(C,A),words(A,P).
word(C,C1,[C|Cs]):- set0(C2,A,(letter(A),word(A,C1,Cs);
                    C1=A, Cs=[]).


/* Program 2

Redefine the predicate 'letter' as follows:  */

letter(32,_):- !,fail.                  /*   space    */
letter(63,_):- !,fail.                  /*     ?      */
letter(46,_):- !,fail.                  /*     .      */
letter(33,_):- !,fail.                  /*     !      */
letter(44,_):- !,fail.                  /*     ,      */
letter(10,_):- !,fail.                  /* line_feed  */
letter(13,_):- !,fail.                  /*     CR     */
letter(L,NL):- (L=<90,L>=64,NL is L+32;  L=NL).

/* Then we replace the calls of the old 'letter' by calls to the
new 'letter':  */

words(C,[P|Ps]):- letter(C,NC),word(NC,C1,L),
                  name(P,L),words(C1,Ps).

word(C,C1,[C|Cs]):- set0(C2),(letter(C2,NC2),word(NC2,C1,Cs);
                    C1 = C2, Cs = []).
```

PROBLEM 82

Verbal statement:

Suppose you have a list L of atoms, for instance proper nouns. Output that list word by word, each of them with the first letter as a capital, and separate from the previous by a comma, the last one separate by "and".

Logic program:

```
output([N]):-capitals(N,N1),write(N1),write('.').
output([N,N1]):- capitals(N,N2),write(N2),write(' and '),
                 output([N1]).
output([N|Ns]):- capitals(N,N1),write(N1),write(', '),
                   output(Ns).

capitals(X,X1):- name(X,[A|L]),A>=97,
                 B is A-32,name(X1,[B|L]).
capitals(X,X).
```

Remarks: 1) the second clause of "capitals" is only needed when the word has already capital letter.

2) the output of the list [John,mary,James] will be:

"John, Mary and James.".

PROBLEM 83 [WARREN,1977b]

Verbal statement:

Consider the following BNF grammar and write it as a PROLOG program.

```
<program>          ::=  <statements>
<statements>       ::=  <statement>|
                        <statement>;<statements>
<statement>        ::=  <name>:=<expr>|
                        IF <test> THEN <statement> ELSE <statement>|
                        WHILE <test> DO <statement>|
                        READ <name>|
                        WRITE <expr>|
                        (<statements>)
<test>             ::=  <expr><comparison op><expr>
<expr>             ::=  <expr><op 2><expr 1>|
                        <expr 1>
<expr 1>           ::=  <expr 1><op 1><expr 0>|
                        <expr 0>
<expr 0>           ::=  <name>|
                        <integer>|
                        (<expr>)
<comparison op>::=  = | < | > | =< |>= |=\=
```

```
<op 2>          ::=  * | /
<op 1>          ::=  + | -
```

Logic program:

```
program(Z0,Z,X) :- statements(Z0,Z,X).

statements(Z0,Z,X):- statement(Z0,Z1,X0),
                     restatements(Z1,Z,X0,X).

restatements((';',Z0),Z,X0,(X0;X)) :- statements(Z0,Z,X).
restatements(Z,Z,X,X).

statement((V,':=',Z0),Z,assign(name(V),Expr)) :-
   atom(V), expr(Z0,Z,Expr).
statement((if,Z0),Z,if(Test,Then,Else)) :-
   test(Z0,(then,Z1),Test),
   statement(Z1,(else,Z2),Then),
   statement(Z2,Z,Else).
statement((while,Z0),Z,while(Test,Do)) :-
   test(Z0,(do,Z1),Test),
   statement(Z1,Z,Do).
statement((read,V,Z),Z,read(name(V))) :- atom(V).
statement((write,Z0),Z,write(expr)) :- expr(Z0,Z,Expr).
statement(('(',Z0),Z,S) :- statements(Z0,(')',Z),S).

test(Z0,Z,test(Op,X1,X2)) :-
   expr(Z0,(Op,Z1),X1), comparisonop(Op),
   expr(Z1,Z,X2).

expr(Z0,Z,X) :- subexpr(2,Z0,Z,X).

subexpr(N,Z0,Z,X) :- N>0, N1 is N-1,
   subexpr(N1,Z0,Z1,X0),
   restexpr(N,Z1,Z,X0,X).
subexpr(0,(X,Z),Z,name(X)) :- atom(X).
subexpr(0,(X,Z),Z,const(X)) :- integer(X).
subexpr(0,('(',Z0),Z,X) :- subexp(2,Z0,(')',Z),X).

restexpr(N,(Op,Z0),Z,X1,X) :- op(N,Op), N1 is N-1,
   subexpr(N1,Z0,Z1,X2),
   restexp(N,Z1,Z,expr(Op,X1,X2),X).
restexpr(N,Z,Z,X,X).

comparisonop(=).
comparisonop(<).
comparisonop(>).
comparisonop(=<).
comparisonop(>=).
comparisonop(\=).

op(2,*).              op(1,+).
op(2,/).              op(1,-).
```

PROBLEM 84 [PEREIRA et al,1978]

Verbal statement:

Construct a simple grammar which parses an arithmetic expression (made up of digits and operators) and computes its value. Consider "-2+3*5+1" as an example of an arithmetic expression.

Logic program:

```
expr(Z) --> term(X),"+",expr(Y),{Z is X+Y}.
expr(Z) --> term(X),"-",expr(Y),{Z is X-Y}.
expr(Z) --> term(Z).

term(Z) --> number(X),"*",term(Y),{Z is X*Y}.
term(Z) --> number(X),"/",term(Y),{Z is X/Y}.
term(Z) --> number(Z).

number(C) --> "+",number(C).
number(C) --> "-",number(X),{C is -X}.
number(X) --> [C],{"0"=<C, C<"9", X is C-"0"}.
```

Execution:

The question: ?-expr(Z,"-2+3*5+1",[]).

will compute Z=14

PROBLEM 85 [WARREN,1974b]

Verbal statement:

Write a program able to translate the functional LISPish notation into the usual PROLOG notation, for example translate

```
:rev([X:L]) = :app(:rev(L),[X]) &
:rev([]) = [] &

:app([X:L1],L2) = [X::app(L1,L2)] &
:app([],L) = L
```

into

```
rev([X:L1],L) :- rev(L1,L2), app(L2,[X],L).
rev([],[]).

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

Logic program:

```
:-op(300,yfx,&).
:-op(200,yfx,=).
:-op(100,fx,:).

/* Translation from functional notation into
   PROLOG notation */

let(P):- let1(P).
let(_).

let1(P&Q):- let1(P).
let1(P&Q):- !,let1(Q).
let1(:T0=V0):- T0=..[F|A0],trans(V0,V,[],C),
               app(A0,[V],A1), T=..[F|A1],
               write(T),output(C),nl,fail.

trans(T0,T0,C0,C0):- var(T0),!.
trans(:T0,V,C0,C):- !, T0=..[F|A0],translist(A0,A,[T|C0],C),
                    app(A,[V],A1), T=..[F|A1].
trans(T0,T,C0,C):- T0=..[F|A0],translist(A0,A,C0,C),
                   T=..[F|A].

translist([T0|A0],[T|A],C0,C):- translist(A0,A,C0,C1),
                                trans(T0,T,C1,C).

translist([],[],C0,C0).

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

output([]):- write('.'),nl,!.
output(X):- write(':- '),output1(X).

output1([A]):- write(A),write('.'),nl,!.
output1([A|B]):- write(A),write(', '),output1(B).
```

Execution:

To perform the program give the commend ":-let(P)." where P must be what you want to translate into PROLOG, for example:

```
:-let(  : rev([X|L]) = :app( :rev(L),[X]) &
        : rev([]) = [] &
        : app([X|L1],L2) = [X|:app(L1,L2)] &
        : app([],L) = L     ).

rev([X|L],L1):- rev(L,L2),app(L2,[X],L1).
rev([],[]).

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

PROBLEM 86

Verbal statement:

Write a program able to translate the PROLOG notation into pseudo-English.

Consider the following definitions as a start point:

```
    P :- Q    means    P if Q
    P , Q     means    P and Q
    P ; Q     means    P or Q
```

the program must also be able to achieve new definitions from the user and to retrovert English into PROLOG notation.

Logic program:

```
:-op(300,xf,?).

begin:- asserta(definition(X,X)),
        asserta(definition((P;Q),[P,or,Q])),
        asserta(definition((P:-Q),[P,if,Q])),
        asserta(definition((P,Q),[P,and,Q])),
        repeat,nl,nl,read(T),(T=stop;process(T),fail).

process(T-D):- !,asserta(definition(T,D)).
process([T1|T2]?):- retrovert(T,[T1|T2]),!,nl,write(T).
process(T):- translate(T,D),nl,output(D),!.
process(_):- nl,write(untranslatable),nl.

/* Retrovertion process */

retrovert(T,D):- definition(T1,D),!,T1=..[N|L],
                 retrovert_args(args,L), T=..[N|Args].

retrovert_args([A|Args],[L|Ls]):- retrovert(A,L),
                                  retrovert_args(Args,Ls).
retrovert_args([],[]).

/* Translation process */

translate(X,X):- var(X),!.
translate(T,D):- T=..[N|Args],args(Args,L),
                 T1=..[N|L],definition(T1,D).

args([A|Args],[L|Ls]):- translate(A,L),args(Args,Ls).
args([],[]).

/* Output process */

output([X|S]):- var(X),write(X),write(' '),output(S).
output([[X]|S]):- output([X|S]).
output([[X|L]|S]):- write('('),output([X|L]),
                    write(')'),output(S).
```

```
output([X|S]):- write(X),write(' '),output(S).
output([]).
```

Execution:

As an example of this program we are going to see the translation of some of its clauses.

```
(:-LC)

(:-op(300,xf,?))

process(_73-_96) if (!  and asserta(definition(_73,_96)) )

process([_78|_101]?) if (retrovert(_133,[_78|_101) and  (!   and
(nl and (write(_133) and put([46]) ))))

process(_73) if (translate(_73,_118) and  (nl  and  (ouput(_118)
and !  )))

process(_73) if (nl and (write(untranslatable) and nl ))

retrovert(_73,_73) if (var(_73) and !  )

retrovert(_73,_94)  if   (definition(_124,_94)   and   (!    and
(_124=..[_177|_200]     and    (retrovert_args(_228,_200)    and
_73=..[_177|_228] ))))

translate(_73,_73) if (var(_73) and !)

translate(_73,_94) if  (_73=..[_134|_157]  and  (args(_157,_201)
and (_222=..[_134|_201] and definition(_222,_94) )))
```

3.7 NATURAL LANGUAGE (nos. 87 to 95)

PROBLEM 87

Verbal statement:

Write a simple grammar G1 for analysing the sentences:

"the giraffe dreams"
"the giraffe eats apples"

Logic program:

```
sent(X,Y):- np(X,U),vp(U,Y).

np(X,Y):- det(X,U),noun(U,Y).

vp(X,Y):- iverb(X,Y).
vp(X,Y):- tverb(X,U),np(U,Y).

det([the|Y],Y).

noun([giraffe|Y],Y).
noun([apples|Y],Y).

iverb([dreams|Y],Y).

tverb([dreams|Y],Y).
tverb([eats|Y],Y).
```

PROBLEM 88

Verbal statement:

Reconsider the previous grammar G1 in order to deal with the following context-sensitive aspects:

1) the number of a noun phrase agrees with that of the corresponding verb phrase;

2) the number of a noun phrase need not be determined by the noun only (this fish--these fish) and not by the 'th-word' only (the giraffe-the giraffes), but number is a feature of the entire noun phrase;

3) a noun phrase need not have a determiner (giraffes dream) presumably provided that the noun phrase is plural.

Logic program:

```
sent(X,Y):- np(N,X,U),vp(N,U,Y).

np(N,X,Y):- th(N,X,U),noun(N,U,Y).
np(plu,X,Y):- noun(plu,X,Y).

vp(N,X,Y):- verb(i,N,X,Y).
vp(N,X,Y):- verb(t,N,X,U),np(M,U,Y).

noun(sin,[U|Y],Y):- npr(U,V).
noun(plu,[V|Y],Y):- npr(U,V).

verb(T,sin,[U|Y],Y):- conj(T,U,V).
verb(T,plu,[V|Y],Y):- conj(T,U,V).

npr(giraffe,giraffes).
npr(fish,fish).
npr(dream,dreams).

conj(T,dreams,dream).
conj(T,eats,eat).

th(N,[the|Y],Y).
th(sin,[this|Y],Y).
th(plu,[these|Y],Y).
```

PROBLEM 89

Verbal statement:

Write a simple grammar able to parse sentences, such as

"John ate the cake"

and to construct their corresponding deep tree structures.

Suggestion: use the definite clause grammars formalism.

Logic program:

```
sentence(s(T1,T2)) --> noun_phrase(T1),
                       verb_phrase(T2).

noun_phrase(np(T1,T2)) --> determiner(T1),
                           noun(T2).

verb_phrase(vp(T1,T2)) --> verb(T1),noun_phrase(T2).

determiner(det(the)) --> [the].
determiner(det([])) --> [].
```

```
noun(n(John)) --> [John].
noun(n(cake)) --> [cake].

verb(v(ate)) --> [ate].
```

PROBLEM 90 [PEREIRA&WARREN,1978]

Verbal statement:

Write a context-free grammar that covers the following sentences,

"John loves Mary",

"every man that lives loves a woman"

and that formalises the mapping between English and formulae of classical logic.

Suggestion: use the definite clause grammars formalism

Logic program:

```
:-op(900,xfx,=>).
:-op(800,xfy,&).
:-op(300,xfx,:).


sentence(P) --> noun_phrase(X,P1,P), verb_phrase(X,P1).

noun_phrase(X,P1,P) -->
   determiner(X,P2,P1,P), noun(X,P3),rel_clause(X,P3,P2).
noun_phrase(X,P,P) --> name(X).

verb_phrase(X,P) --> trans_verb(X,Y,P1), noun_phrase(Y,P1,P).
verb_phrase(X,P) --> intrans_verb(X,P).

rel_clause(X,P1,P1&P2) --> [that], verb_phrase(X,P2).
rel_clause(_,P,P) --> [].

determiner(X,P1,P2, all(X):(P1=>P2) )--> [every].
determiner(X,P1,P2, exists(X):(P1&P2) ) --> [a].

noun(X,man(X)) --> [man].
noun(X,woman(X)) --> [woman].

name(John) --> [John].

trans_verb(X,Y,loves(X,Y)) --> [loves].
intrans_verb(X,lives(X)) --> [lives].
```

PROBLEM 91 [PEREIRA;WARREN,1978]

Verbal statement:

Write a grammar that covers the following sentences,

'fred shot John'

'mary was liked by John'

'fred told mary to shot John'

'John was believed to have been shot by fred'

'was John believed to have told mary to tell fred'

Suggestion: use the definite clause grammars formalism.

Logic program:

```
sentence(S) -->
   [W], {aux_verb(W,Verb,Tense)},
   noun_phrase(G_Subj),
   rest_sentence(q,G_Subj,Verb,Tense,S).
sentence(S) -->
   noun_phrase(G_Subj),
   [W], {verb(W,Verb,Tense)},
   rest_sentence(del,G_Subj,Verb,Tense,S).

rest_sentence(Type,G_Subj,Verb,Tense,
                  s(Type,L_Subj,tns(Tense1),VP) ) -->
   rest_verb(Verb,Tense,Verb1,Tense1),
   {verbtype(Verb1,VType)},
   complement(VType,Verb1,G_Subj,L_Subj,VP).

rest_verb(have,Tense,Verb,(Tense,perfect)) -->
   [W], {past_participle(W,Verb)}.
rest_verb(Verb,Tense,Verb,Tense) --> [].

complement(copula,be,Obj,Subj, vp(v(Verb),Obj1) ) -->
   [W], {past_participle(W,Verb), transitive(Verb)},
   rest_object(Obj,Verb,Obj1),
   agent(Subj).
complement(transitive,Verb,Subj,Subj, vp(v(Verb),Obj1) ) -->
   noun_phrase(Obj),
   rest_object(Obj,Verb,Obj1).
complement(intransitive,Verb,Subj,Subj, vp(v(Verb)) ) --> [].

rest_object(Obj,Verb,S) -->
   {s_transitive(Verb)},
   [to,Verb1], {infinitive(Verb1)},
   rest_sentence(del,Obj,Verb1,present,S).
rest_object(Obj,_,Obj) --> [].
```

```
agent(Subj) --> [by], noun_phrase(Subj).
agent(np(pro(someone))) --> [].

noun_phrase(np(Det,adj(Adjs),n(Noun))) -->
   [Det], {determiner(Det)},
   adjectives(Adjs),
   [Noun], {noun(Noun)}.
noun_phrase(np(npr(PN))) --> [PN], {proper_noun(PN)}.

adjectives([Adj|Adjs]) -->
   [Adj], {adjective(Adj)},
   adjectives(Adjs).
adjectives([]) --> [].


aux_verb(W,V,T):- verb(W,V,T),auxiliary(V).

auxiliary(be).
auxiliary(have).

verb(is,be,present).
verb(was,be,perfect).
verb(tell,tell,present).
verb(told,tell,perfect).
verb(shoot,shoot,present).
verb(shot,shoot,perfect).
verb(like,like,present).
verb(liked,like,perfect).
verb(believe,believe,present).
verb(believed,believe,perfect).


proper_noun(john).
proper_noun(fred).
proper_noun(mary).


determiner(the).

adjective(nice).

noun(book).

verbtype(be,copula).
verbtype(V,transitive) :- transitive(V).
verbtype(V,intransitive) :- intransitive(V).


transitive(V) :- s_transitive(V).
transitive(shoot).

s_transitive(tell).
s_transitive(believe).
s_transitive(like).
```

```
infinitive(be).
infinitive(shoo).
infinitive(tell).
infinitive(have).

past_participle(been,be).
past_participle(shot,shoot).
past_participle(told,tell).
past_participle(liked,like).
past_participle(believed,believe).
```

PROBLEM 92 [COLMERAUER,1977]

Verbal statement:

Write a simple Portuguese grammar for analysing sentences, such as:

"helder habita em caparica" (helder lives at caparica)

"helder e' mais velho que luis" (helder is older than luis)

Consider the input sentence as a list.

Logic program:

```
:-op(500,xfy,'-').


/* Dialogue control */

ola:- nl,read(X),phrase(R,X,[]),answer(R),nl.

answer(X):- X,!, display('Sim.'),nl,nl.
answer(X):- display('Nao'),nl,nl.

/* Deductive block */

and(X,Y):- X,Y.

local(X):- file(_,_,X).

type(X):- file(X,_,_).

live_at(X,Y):- file(X,_,Y).

older_than(X,Y):- file(X,I1,L1),file(Y,I2,L2),
                  I2 < I1.

/* Grammar */

phrase(O3) --> np(X,O2,O3),vp([suj-X|L],O1),
               compls(L,O1,O2).
```

```
compls([],0,0) --> !.
compls([K-X|L],01,03) --> compls(L,01,02),case(K),
                          np(X,02,03).

np(X,02,04) --> art(X,01,02,03),name([suj-X|L],01),
                compls(L,03,04).
np(X,0,0) --> [X],{ind(X)}.

ind(X):- type(X).
ind(X):- local(X).

case(em) --> [em].
case(que) --> [que].
case(dir) --> [].

art(X,01,02,and(01,02)) --> [um].
art(X,01,02,not(and(01,02))) --> [nenhum].
art(X,01,02,not(and(01,not(02)))) --> [cada].

name([suj-X],type(X)) --> [tipo];[pessoa].
name([suj-X],local(X)) --> [local];[morada].

vp([suj-X,em-Y],live_at(X,Y)) --> [habita].
vp([suj-X,que-Y],older_than(X,Y)) --> ['e'''',mais velho].


/* Data base */


file(luis,29,caparica).
file(helder,33,restelo).
file(fernando,26,alvalade).
file(carlos,36,olivais).
```

PROBLEM 93

Verbal statement:

Write a simple program able to derive Portuguese relative constructions, given their sentential components.

Hint:

Consider the following two sentences,

"eu vi o rapaz" (I saw the boy)

"o rapaz comprou o livro" (The boy bought the book)

build up its deep structure and analyse how it may be transformed to represent the corresponding relative sentence.

Logic program:

```
go:- frase(F),write(F),nl
     sent(T,F,[]),nl,write(T)
     transrel(T,V),nl,write(V),nl,
     sent(V,TR,[]),nl,write(TR),nl.


/* Grammar */


sent([fr,X,Y,Z]) --> su(X),aux(Y),sv(Z).

sn([sn,X]) --> n(X).
sn([sn,X,Y]) --> det(X),n(Y).
sn([sn,X]) --> pro(X).
sn([sn,X,Y]) --> sn(X),sent(Y).

sv([sv,X]) --> v(X).
sv([sv,X,Y]) --> cop(X),adj(Y).
sv([sv,X,Y]) --> v(X),sn(Y).
sv([sv,X]) --> cop(X),sn(Y).

aux([aux,X]) --> tpo(X).

/* lexicon */

tpo([tpo,pass]) --> [pass].

cop([cop,estar]) --> [estar].
cop([cop,ser]) --> [ser].

adj([adj,contente]) --> [contente].

pro([pro,que]) --> [que].

n([n,eu]) --> [eu].
n([n,rapaz]) --> [rapaz].
n([n,livro]) --> [livro].

det([det,o]) --> [o].

v([v,ver]) --> [ver]
v([v,comprar]) --> [comprar].

/* transformations for the grammar */

transrel(T,V):- change([sn,[sn,ST1],R],
                       [sn,[sn,ST1],[fr,ST2]],
                       T,
                       V),
                change([sn,[pro,que]],
                       [sn,ST1]
                       [fr,ST2],
                       R).
```

```
/* useful predicates */

change(CH,T,T,CH):- !.
change(CH,ST,[OP,T|TS],[OP,R|TS]):- change(CH,ST,T,R).
change(CH,ST,[OP,T|TS],[OP,T|R]):- change1(CH,ST,TS,R).

change1(CH,ST,[T|TS],[R|TS]): change(CH,ST,T,R).
change1(CH,ST,[T|TS],[T|R]):- change1(CH,ST,TS,R).


/* Sentences */

frase([eu,pass,ver,o,rapaz,o,rapaz,pass,comprar,o,livro]).
```

PROBLEM 94

Verbal statement:

Write a program able to interpret document titles and to be included in a question-answering system, due to implement an automatic library service.

The program goal consists of proposing categories to the user by reading the document titles, and interpreting the role played by prepositions.

Consider the following title :

"Logic for problem solving".

Logic program:

```
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*                        TITLE INTERPRETATION                          */
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/



:-op(400,xfy,'->').


/* Using the title definition of a text for classification
   purposes */


/* Grasping meaning for the title entities of a document */


parser(T,C):-
          title(Out,T,[]),semant(Out,C1,C2),
          output(C1,C2),append(C1,C2,C).
```

```
output([],[]):-!.
output([],C2):-display('I recommend as categories:  '),
               print_ph(C2),nl,!.
output(C1,C2):-display('I recommend as categories:  '),
               print_ph(C1),display('and '),print_ph(C2),nl.


/* Basic grammar rules governing the title construction */


title([np(X),Y,Z]) --> bp(X,Y),!,title(Z).
title([np(X)]) --> np(X).


bp([],P) --> prep(P).
bp([Y|L],P) --> det(Y),!,bp(L,P).
bp([n(X)|Y],P) --> [X],bp(Y,P).

np([Y|L]) --> det(Y),!,np1(L).
np(X) --> np1(X).

np1([n(X)|Y]) --> [X],np1(Y).
np1([n(X)]) --> [X].


/* Dictionary:  lexical units */


prep(prep(for)) --> [for].

det([a]) --> [a].
det([an]) --> [an].
det([the]) --> [the].


/* Prepositional cases:  ordering plans for conversation */


prep_case(for,X,Y):-
          assert(case(tool->X,application->Y)),nl,nl,
          write('your document deals with "'),
          print_ph(X),write('"'),
          write(' as a tool,'),nl,
          write('with the objective to contribute to "'),
          print_ph(Y),write('".'),nl,nl,hypothesis1(X,Y).


hypothesis1(X,Y):-
          write('My first hypothesis for classifying it '),
          write('is to take its nouns as hints,'),
          nl,write('So " '),print_ph(X),write('"'),
          write(' will drive us, in order '),
          write('to attain the main category, and'),
          nl,write('"'),print_ph(Y),
          write('"'),write(' towards other categories.'),nl,nl.
```

```
/* Gathering of clues,given by user,for classifying a
   document */


semant([np(X)],X,Y).

semant([np(X),prep(Y),[np(Z)]],W,T):-
            list_nouns(X,Z,U,V),
            pre_case(Y,U,V),
            drive_in(W,T).


/* Driving the conversation about title content */


drive_in(W,V):-case(X,Y),
                  drive(X,U),drive(Y,V),test(X,U,W).
drive(X->Y,C):-move(X,Y),plan(X,L),state(X,L,C),!.
drive(X->Y,[]).


move(X,Y):-write('Do you want to follow the idea that '),
            print_ph(Y),write('is an AI '),
            write(X),write(?),nl,agreement(yes),
            write('OK.Let us talk about this hypothesis'),
            write(' and fix the kind of '),
            write(X),write(!),nl,!.


agreement(Input):-read(Input).


state(_,[],[]):- !.
state(A,X,Y):-
        write('During our dialogue I grasp the followings:'),nl,
        print_ph(X),write('as core topics of your text.'),nl,
        under(A,X,Y),state1(Y),nl,!.


state1([]):- !.
state1(Y):-write('So I conclude it may be under'),
            write(' the categories:  '),
            print_ph(Y).


under(A,[X|L],[N|NL]):-choose(A,X,N),under(_,L,NL).
under(_,[],[]).


choose(tool,X,Y):-s_tool(X,Y).
choose(application,X,Y):-s_application(X,Y).
choose(_,_,[]).
```

```
/* Plan 1:  through the tree of tools */


plan(tool,L):-
     try(t([techniques,programming_languages],
[t([reasoning,representational,searching],
[t([puzzle_solving,question_answering,common_sense],
[[],[],t([deduction,planning],[[],[]])]),
t([predicate_calculus,semantic_nets],[[],[]]),
t([state_space,problem_reduction],[[],[]])]),
t([numerical_languages,ai_languages],[[],
t([list_type,predicate_logic_type],[[],[]])])]),L,tool),!.

/* Plan 2:  through the tree of applications */

plan(application,L):-
try(t([cognition,other_fields],[t([activities,abilities],
[t([intellectual_skills,motor_skills],
[t([playing,proving],[[],
t([theorems,programs],[[],[]])],[]]),
t([language,vision],[[],[]])]),[]]),L,application).

try(t([X|_],[T|_]),[X|L],C):-
            request(X,C),!,try(T,L,C).

try(t([_|XN],[_|TN]),L,C):-try(t(XN,TN),L,C).

try(_,[],_).

request(X,C):-write('Is '),write(X),
               write(' an adequate '),write(C),
               write(' worked out in your text?'),nl,
               agreement(yes).

/* testing return to a previous conversation */

test(X->Y,[],Z):-
   write('As no conclusion was taken about core topics of '),
   write(X),nl,
   write('I suggest we restart our conversation!'),
   nl,write('Do you agree?'),nl,
   agreement(yes).
test(_,Y,Y).



Execution:


:-parser.
Document title?
[logic,for,problem,solving].
```

```
Your document deals with 'logic ' as a tool,
with the objective to contribute to 'problem solving '.

My first hypothesis for classifying it is to take its nouns as
hints.
So ' logic ' will drive us, in order to attain the main
category, and
'problem solving ' towards other categories.

Do you want to follow the idea that logic is an AI tool?
yes.
Ok.Let us talk about this hypothesis and fix the kind of tool!
Is techniques an adequate tool worked out in your text?
yes.
Is reasoning an adequate tool worked out in your text?
yes.
Is puzzle_solving an adequate tool worked out in your text?
no.
Is question_answering an adequate tool worked out in your text?
no.
Is common_sense an adequate tool worked out in your text?
no.
During our dialogue I grasp the following:
techniques reasoning as core topics of your text.
So I conclude it may be under the categories:  [] 364
Do you want to follow the idea that problem solving is an AI
application?
Ok.Let us talk about this hypothesis and fix the kind of
application!
Is cognition an adequate application worked out in your text?
yes.
Is activities an adequate application worked out in your text?
no.
Is abilities an adequate application worked out in your text?
yes.
Is language an adequate application worked out in your text?
yes.
During our dialogue I grasp the following:
cognition abilities language as core topics of your text.
So I conclude it may be under the categories:  [] 336 365
I recommend as categories:  [] 364 and [] 336 365
```

PROBLEM 95

Verbal statement:

Write a program to generate the deep structure (tree) of a natural language sentence.

Logic program:

```
sentence(s(T1,T2)) --> noun_phrase(T1),
                       verb_phrase(T2).
```

```
noun_phrase(np(T1,T2)) --> determiner(T1),
                           noun(T2).

verb_phrase(vp(T1,T2)) --> verb(T1), noun_phrase(T2).

determiner(det(the)) --> [the].
determiner(det([])) --> [].

noun(n(john)) --> [john].
noun(n(cake)) --> [cake].

verb(v(ate)) --> [ate].
```

3.8 GRAPH THEORY (nos. 96 to 100)

PROBLEM 96

Verbal statement:

Find a path from A to Z in the following directed graph (Figure 17).

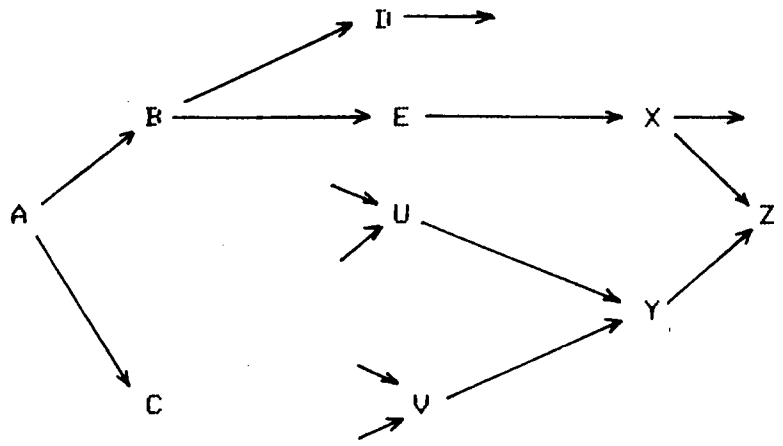


Figure 17

Logic program:

```
so(X,Y):- so(X,Z),write('Path:'),nl,write((X,Z)),
          so(Z,Y),write((Z,Y)),nl

so(a,b).                  so(x,z).
so(a,c).                  so(y,z).
so(b,d).                  so(u,y).
so(b,e).                  so(v,y).
so(e,x).
```

Execution:

```
:-so(a,z).
```

PROBLEM 97 [EMDEN,1976e]

Verbal statement:

Let the interpretation I be

{G(A,B),G(A,C),G(A,D),G(B,F),G(C,F),G(E,F),G(F,C)}

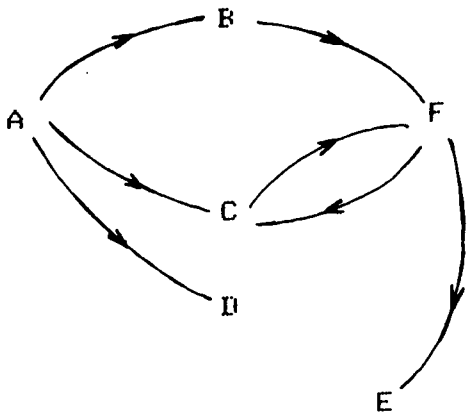


Figure 18

Calculate:

- Q1=G(x1,x2) ∧ G(y1,y2)
- Q2=G(x,y) ∧ G(y,z)
- Q3=G(x,x)
- Q4= ∃y.G(x,y)
- Q5= ∃y.G(x,y) ∧ G(y,z)

Logic program:

```
q1(X1,X2,Y1,Y2):- s(X1,X2),s(Y1,Y2).
q2(X,Y,Z):- s(X,Y),s(Y,Z).
q3(X):- s(X,X).
q4(X):- s(X,_).
q5(X,Z):- s(X,Y),s(Y,Z).


s(a,b).   s(a,c).   s(a,d).
s(b,f).   s(c,f).   s(e,f).   s(f,c).


answer1:- q1(X1,X2,X3,X4),
          write((X1,X2,X3,X4)),nl.
answer1.
```

PROBLEM 98 [SZEREDI,1977]

Verbal statement:

Construct a program for drawing a picture (a graph), such as Figure 19, using a continuous line.

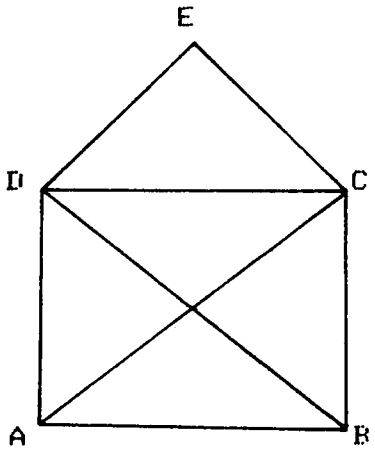


Figure 19

Logic program:

```
:-op(600,xfy,'-').


draw(G,[P,Q|L]):- choose(G,P-Q,G1),
                  draw(G1,[Q|L]).
draw([],[Q]).


choose([P-Q|G],P-Q,G).
choose([P-Q|G],Q-P,G).
choose([E|G],F,[E|G1]):- choose(G,F,G1).
```

Execution:

```
:-draw([a-b,a-c,a-d,b-c,b-d,c-d,c-e,d-e],L),
  output(L).
```

The result is output on line-printer by execution of procedure output to be specified.

PROBLEM 99

Verbal statement:

It is required a graph generator able to:

1) generate all possible graphs with one-way edges from an initial given graph;

2) fix some nodes for each generated graph, i.e. erase the edges connecting to those nodes;

3) determine the end nodes;

4) determine which nodes (and their number) may attain the previous end nodes.

Logic program:

```
begin:- display('list of nodes ?'),nl,read(L),record(nds(L)),
        display('list of edges ?'),nl,read(X),record(ed(X)),
        display('list of relevant origin ?'),nl,read(Y)),
        record(or(Y)),matrizes.

matrix:- recorded(ed(L),Pt),erase(Pt),compact(L,L1),
         calcul(L1),fail.
matrix:- display('Done.'),nl,nl,nl.

compact([],[]).
compact([m(X,Y)|B],R):- in(m(X,Y),B,D),!,
         (R=[m(X,Y)|C] ;  R=[m(Y,X)|C]),
         compact(D,C).
compact([A|B],[A|C]):- compact(B,C).

in(m(A,B),[m(B,A)|X],X).
in(m(A,B),[C|X],[C|R]):- in(m(A,B),X,R).

calcul(L1):- ?or(Y),in(A,Y),name(A,A1),
       X=[78,79,46|A1],name(X1,X),tell(X1),
       slash(out_of_list(A,L1,L2)), ?nds(L),
       slash(look(L,L2,L3)),calcul1(L2,L3).

calcul1(L2,L3):- in(B,L3),
        test(B),complete(L2,B).

complete(L2,B):- ?nds(L),oris(L2,B,L,O),length(O,No),put(' '),
         test(No),output(O),nl,!.

oris(L2,B,[N|Ns],[N|O]):- path(N,B,L2),
     oris(L2,B,[N|Ns],[N|O]):- path(N,B,L2),
          oris(L2,B,Ns,O).
oris(L2,B,[_|Ns],O):-oris(L2,B,Ns,O).
oris(_,_,[],[]).
```

```
slash(P):- call(P),!.

path(A,B,L):- in(m(A,B),L).
path(A,B,L):- in(m(A,C),L),slash(out_of_list(C,L,Li)),
    path(C,B,Li).

out_of_list(A,[m(_,A)|X],Y):- out_of_list(A,X,Y).
out_of_list(A,[B|X],[B|Y]):- out_of_list(A,X,Y).
out_of_list(_,[],[]).

look([N|Ns],L2,Y):- in(m(N,_),L2),
        look(Ns,L2,Y).
look([N|Ns],L2,[N|Y]):- in(m(_,N),L2),
        look(Ns,L2,Y).
look([N|Ns],L2,Y):- look(Ns,L2,Y).
look([],_,[]).

in(A,[A|_]).
in(A,[_|B]):- in(A,B).

output([A]):- put(" "),test(A).
output([A|B]):- put(" "),test(A),output(B).

test(A):- A<10,put(" "),write(A),!.
test(A):- write(A),!.
```

PROBLEM 100

Verbal statement:

Write a program to test connections between nodes in a given graph. Apply your program to the graph in Figure 20 in order to test connections between A and D, and A and E.

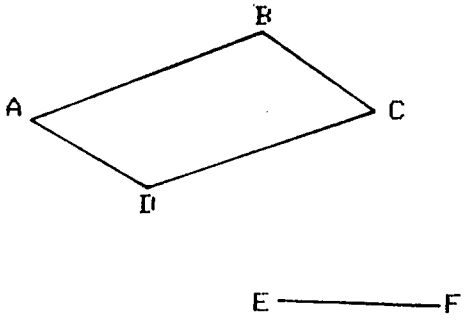


Figure 20

Remark: This problem examplifies the use of 'ancestors'.Observe that your program must prevent loops caused by the circuits of the graph. Use the built-in procedure "ancestors".

Logic program:

Program 1 :

```
/* connections test */

connections(X,Y):- (connected(X,Y);connected(Y,X)),
                   nl,write('yes.'),nl.
connections(_,_):- nl,write('no.'),nl.

connected(X,Y):- ancestors(L),
                 in2(connected(X,Y),L),!,fail.
connected(X,Y):- edge(X,Y).
connected(X,Y):- edge(X,Z),connected(Z,Y).

in2(A,[A|B]):- in(A,B).
in2(A,[_|B]):- in2(A,B).

in(A,[A|_]).
in(A,[_|B]):- in(A,B).

/* Graph description */

edge(a,b).        edge(b,c).        edge(e,d).
edge(d,a).        edge(e,f).
```

Program 2 :

```
/* connection test */

connections(X,Y):-(connected(X,Y);connected(Y,X)),
                   nl,write('yes.'),nl.
connections(_,_):- nl,write('no.'),nl.

connected(X,Y):- edge(X,Y).
connected(X,Y):- edge(X,Z),
                 (subgoal_of(connected(Z,Y)),!,fail;
                  connected(Z,Y)).

/* Graph description */

edge(a,b).        edge(b,c).        edge(e,d).
edge(d,a).        edge(e,b).
```

Execution:

```
:-connections(a,d).

yes.

:-connections(a,e).

no.
```

3.9 ALGEBRA (nos. 101 to 103)

PROBLEM 101

Verbal statement:

Specify the differentiation function.

Logic program:

```
:-op(700,xfx,:).
:-op(500,xfy,[+,-]).
:-op(400,xfy,[*,/]).
:-op(300,xfy,@).
:-op(300,fy,[-,exp,log,sin,cos,tg]).

dv:-repeat,read(T),( T=stop ;  derive(T),fail).

derive(T:X):- T :  X, !.

X :  [V] :- X :  V.

X :  [U:V] :- d(X,U,W), simplify(W,Z), Z :  V.

X :  Y :-d(X,Y,W), simplify(W,Z),
         ttynl,output(Z),ttynl,ttynl.

d(X,X,1).

d(T,X,0):- atom(T) ;  number T.

d(U+V,X,A+B):- d(U,X,A), d(V,X,B).

d(U-V,X,A+(-B)):- d(U,X,A), d(V,X,B).

d(-T,X,-R):- d(T,X,R).

d(K*U,X,K*W):- number K, d(U,X,W).

d(U*V,X,B*U+A*V):- d(U,X,A), d(V,X,B).

d(U/V,X,W):- d(U*V@(-1),X,W).

d(U@V,X,V*W*U@(V+(-1))):- number V, d(U,X,W).

d(U@V,X,Z*log(U)*U@V+V*W*U@(V+(-1))):-
                                        d(U,X,W), d(V,X,Z).

d(log T,X,R*T@(-1)):- d(T,X,R).
```

```
d(exp T,X,R*exp T):- d(T,X,R).

d(sin T,X,R*cos T):- d(T,X,R).

d(cos T,X,-R*sin T):- d(T,X,R).

d(tg T,X,W):- d(sin T/cos T,X,W).

simplify(X,X):- atomic(X).
simplify(X,Y):- X =..[Op,Z],simplify(Z,Z1), rewrite(Op,Z1,Y).
simplify(X,Y):- X=..[Op,Z,W],simplify(Z,Z1),
                simplify(W,W1),
                rewrite(Op,Z1,W1,Y).

rewrite(exp,K*log X,W):- binary(@,X,K,W).
rewrite(-,A+B,C+D):- unary(-,A,C), unary(-,B,D).
rewrite(X,Y,Z):- unary(X,Y,Z).

rewrite(*,X,X,W):-binary(@,X,2,W).
rewrite(*,-X,X,W):- binary(@,X,2,Z),unary(-,Z,W).
rewrite(*,X,-X,W):-binary(@,X,2,Z),unary(-,Z,W).
rewrite(*,-Y,Z):- binary(*,X,Y,Z).
rewrite(*,X,-Y,W):- binary(*,X,Y,Z), unary(-,Z,W).
rewrite(*,-X,Y,W):- binary(*,X,Y,Z), unary(-,Z,W).
rewrite(*,A@X,A@Y,W):- binary(+,X,Y,Z), binary(@,A,Z,W).
rewrite(*,X@A,Y@A,W):- binary(*,X,Y,Z), binary(@,Z,A,W).
rewrite(@,X*Y@(-1),-Z,W):- binary(@,Y*X@(-1),Z,W).
rewrite(X,Y,Z,W):- binary(X,Y,Z,W).

unary(-,-X,X).
unary(-,0,0).
unary(-,X,Y):- Y isr -X.
unary(log,1,0).                          unary(log,exp X,X).
unary(exp,0,1).                          unary(exp,log X,X).
unary(sin,0,0).                          unary(sin,-X,- sin X).
unary(cos,0,1).                          unary(cos,-X,cos X).
unary(Op,X,Y):- Y =..[Op,X].

binary(+,X,0,X).                         binary(+,0,X,X).
binary(+,X,-X,0).                        binary(+,-X,X,0).
binary(+,A,B+C,R+C):- R isr A+B.
binary(+,A+B,C,R+B):- R isr A+C.
binary(+,(sin X)@2,(cos X)@2,1).
binary(+,(cos X)@2,(sin X)@2,1).
binary(+,X,Y,Z):- Z isr X+Y.
binary(+,A,B,B+A):- number B.

binary(*,X,1,X).                         binary(*,1,X,X).
binary(*,0,_,0).                         binary(*,_,0,0).
binary(*,-1,X,-X).                       binary(*,X,-1,-X).
binary(*,X,X@(-1),1).                    binary(*,X@(-1),X,1).
binary(*,A,B*C,R*C):- R isr A*B.
binary(*,A*B,C,R*B):- R isr A*C.
binary(*,X,Y,Z):- Z isr X*Y.
binary(*,A,B,B*A):- number B.
```

```
binary(@,1,_,1).
binary(@,X,1,X).
binary(@,_,0,1).

binary(Op,X,Y,Z):- Z =..[Op,X,Y].

output(X):- clean(X,Y), write(Y).

clean(X,X):- atom(X) ; number X.
clean(X,Y):- X =..[Op,Z], clean(Z,Z1), Y =..[Op,Z1].
clean(X,Y):- X =..[Op,Z,W], clean(Z,Z1), clean(W,W1),
             U =..[Op,Z1,W1], change(U,Y).

change(-1+X,X-1).
change(X+(-Y),X-Y).
change(X*Y@(-1),X/Y).
change(X@(-1)*Y,Y/X).
change(X,X).
```

PROBLEM 102 [EMDEN,1976c]

Verbal statement:

Define the exponentiation function:

$$exp(x,y,z) \quad \text{iff} \quad x^y = z$$

The following properties of integer exponentiation are taken as the defining ones:

$$x^0 = 1 \quad \text{i.e.} \quad x.exp(x,0,1)$$

and

$$x^y = x.x^{y-1} \quad \text{i.e.} \quad x,y,z.exp(x,y-1,z) \Rightarrow exp(x,y,x.z)$$

It is taken for granted that the following defining property can be added without altering the relation defined and yet making the definition more 'efficient':

$$x^{2y} = (x^2)^y \quad \text{i.e.} \quad x^y = (x^2)^{y/2} \quad \text{i.e.}$$

$$x,y,z.(exp(x.x,y/2,z) \ \ even(y)) \Rightarrow exp(x,y,z)$$

Logic program:

```
exp(X,0,1).
exp(X,Y,Z):- even(Y), R is Y/2, P is X*X,
             exp(P,R,Z).
exp(X,Y,Z):- T is Y-1, exp(X,T,Z1), Z is Z1*X.

even(Y):- R is Y mod 2, R==0.
```

Execution:

```
:- exp(2,10,Z),write(Z),nl.

1024
```

PROBLEM 103

Verbal statement:

Write a program describing the Fibbonaci function.

Logic program:

```
fib(0,1).
fib(1,1).
fib(X,F):- Z is X-2, Y is Z+1, fib(Y,Y1),fib(Z,Z1),
           F is Y1+Z1.
```

3.10 ARITHMETIC (nos. 104 to 116)

PROBLEM 104

Verbal statement:

Find the absolute value of an integer.

Logic program:

```
abs(X,X):- X >= 0,!.
abs(X,Y):- Y is -X.
```

PROBLEM 105

Verbal statement:

Define factorial.

Logic program:

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

Execution:

```
:-factorial(10,Y),write(Y),nl.
3628800
```

PROBLEM 106

Verbal statement:

Define factorial with recursion on the right.

Logic program:

```
factorial2(N,F):- fact(N,1,F).

fact(0,F,F).
fact(N,X,F):- M is N-1, Y is X*N,
              fact(M,Y,F).
```

PROBLEM 107 [CLARK&KOWALSKI 1977]

Verbal statement:

Define factorial in a bottom-up way.

Logic program:

```
fact(U,FU):- bufact(0,1,U,FU).

bufact(X,FX,X,FX).
bufact(X,FX,U,FU):- NX is X+1, FNX is NX*FX,
                    bufact(NX,FNX,U,FU).
```

PROBLEM 108 [adapted from BRUYNOOGHE 1978]

Verbal statement:

Generate primes in first N integers and call them primes(N,CP).

Logic program:

```
primes(N,[1|LP]):- M is N-1,
                   integers(2,M,LI),
                   sift(LI,LP).

integers(N,0,[N]).
integers(N,M,[N|LI]):- R is M-1,
                       Q is N+1,
                       integers(Q,R,LI).

sift([],[]).
sift([P|LI],[P|LP]):- filter(P,LI,NLI),
                      sift(NLI,LP).


filter(P,[],[]).
filter(P,[N|LI],[N|NLI]):- divide(P,N,false),
                           filter(P,LI,NLI).

filter(P,[N|LI],NLI):- divide(P,N,true),
                       filter(P,LI,NLI).

divide(P,N,true):- 0 is N-P*(N/P).
divide(P,N,false).
```

The method is the "sieve of Erastosthenes". It consists in generating the list of integers from 2 to M, and in deleting from that list all the integers multiple of some element in the list.

PROBLEM 109

Verbal statement:

Define Euclid's algorithm: maximum common divisor and minimum common multiple.

Logic program:

```
mcd(N,0,N).
mcd(N,M,D):- R is N mod M, mcd(M,R,D).

mcm(N,M,E):- mcd(N,M,D),E is (N*M)/D.
```

PROBLEM 110

Verbal statement:

Define the roman to arabic number conversion.

Logic program:

```
convert:- repeat,read(R),(R==stop;roman(R),fail).

roman(R):- roman(D,R,[]),!,nl,
           write(D),nl,nl.

roman(R) --> value(X),value(Y),{X<Y},
             roman(R1),{R is Y-X+R1}.
roman(R) --> value(X),roman(R1),{R is X+R1}.

roman(0)    --> [].

value(1)    --> "I".
value(5)    --> "V".
value(10)   --> "X".
value(50)   --> "L".
value(100)  --> "C".
value(500)  --> "D".
value(1000) --> "M".
```

PROBLEM 111

Verbal statement:

Compute the maximum of two numbers.

Logic program:

```
max(X,X,X).
max(X,Y,Y):- X<Y.
max(X,Y,X):- Y<X.
```

PROBLEM 112

Verbal statement:

Define an arithmetic for rational expressions.

Logic program:

```
/* Package 1 ;  A and B non negative integers */

rinf(A,B):- somreel(B,-A,C),signal(C,1).

rdiff(A,B):- equal(A,B),!,fail.
rdiff(A,B).

absdiff(X,Y,Z):- rinf(X,Y),!,somreel(Y,-X,Z).
absdiff(X,Y,Z):- somreel(X,-Y,Z).

somreel(A,B,C):- decomp(A,S1,A1,A2),
                 decomp(B,S2,B1,B2),
                 ppcm(A2,B2,P,A12,B12),
                 N1 is A1*A12,N2 is B1*B12,
                 som(S1,N1,S2,N2,S,N),
                 pgcd(N,P,D,H,K),
                 decomp(C,S,H,K).

som(S,A,S1,A1,1,0):- S\==S1,!.
som(S,A,S,B,S,C):- !, C is A+B.
som(S,A,S1,B,S,C):- B<A,!,C is A-B.
som(S,A,S1,B,S1,C):- C is B-A.

subreel(-A,-B,C):- !,somreel(-A,B,C).
subreel(-A,B,C):- !,somreel(-A,-B,C).
subreel(A,-B,C):- !,somreel(A,B,C).
subreel(A,B,C):- !,somreel(A,-B,C).

prodreel(A,B,C):- decomp(A,S1,A1,A2),
                  decomp(B,S2,B1,B2),
                  sigprod(S,S2,S),
                  C1 is A1*B1,C2 is A2*B2,
                  pgcd(C1,C2,D,C11,C12),
                  decomp(C,S,C11,C12).

divreel(A,B,C):- decomp(A,S1,A1,A2),
                 decomp(B,S2,B1,B2),
                 sigprod(S1,S2,S),
                 N1 is A1*B2, D1 is A2*B1,
                 pgcd(N1,D1,G,N,D),
                 decomp(C,S,N,D).

sigprod(S,S,1):- !.
sigprod(S,S1,-(1)).

pgcd(A,B,C,D,E):- B<A,!,gcd(A,B,C),
                  D is A/C, E is B/C.
```

```
pgcd(A,B,C,D,E):- gcd(B,A,C),
                  D is A/C, E is B/C.

gcd(A,0,A):- !.
gcd(A,B,D):- C is A mod B,gcd(B,C,D).

ppcm(A,B,P,C,E):- pgcd(A,B,D,E,C),
                  Q is A*B, P is Q/D.

signal(-A,-(1)):- !,signal(A,1).
signal(-A,1):- !,fail.
signal(A,-(1)):- !,fail.
signal(0,0):- !.
signal(0,1):- !,fail.
signal(A,1).

decomp(-A,-S,A1,A2):- !,decomp(A,S,A1,A2).
decomp(A\B,1,A,B):- B\==1,!.
decomp(A,1,A,1).

equal(A,A):- !.
equal(A,B):- somreel(A,-B,C),
             somreel(C,-(1\1000),R1),
             somreel(C,1\1000,R2),
             signal(R1,-(1)),
             signal(R2,1).



/* Package 2 */

:-op(700,xfx,[isr,norms_to,denorms_to]).
:-op(300,xfy,^).

X isr Y :- rat(Y,X),!.

rat(U+V,W):- integer(U),integer(V),W is U+V;
   rat(U,X),rat(V,Y),
   X norms_to X1/X2,Y norms_to Y1/Y2,
   lcm(X2,Y2,P,X12,Y12),
   Z is X1*X12+Y1*Y12,
   gcd(Z,P,_,W1,W2),
   W1/W2 denorms_to W.
rat(U*V,W):- integer(U),integer(V),W is U*V;
   rat(U,X),rat(V,Y),
   X norms_to X1/X2,Y norms_to Y1/Y2,
   Z1 is X1*Y1,Z2 is X2*Y2,
   gcd(Z1,Z2,_,W1,W2),
   W1/W2 denorms_to W.
rat(-U,V):- integer(U),V is -U ;
   rat(U,V),
   U norms_to U1/U2,V1 is -U1,
   V1/U2 denorms_to V.
rat(U-V,W):- rat(-V,V1),rat(U+V1,W).
rat(U/1,V):- rat(U,V).
```

```
rat(U/V,W):- rat(U,X),rat(V,Y),
   X norms_to X1/X2,Y norms_to Y1/Y2,
   sign(Y1,S,A),
   Z1 is X1*Y2*S,Z2 is X2*A,
   gcd(Z1,Z2,_,W1,W2),
   W1/W2 denorms_to W.
rat(U^V,W):- rat(V,Y),integer(Y),
   rat(U,X),X norms_to X1/X2,
   sign(Y,_,E),expt(X1,E,Z1),expt(X2,E,Z2),
   (Y<0,sign(Z1,S,W1),W2 is S*Z2,W2/W1 denorms_to W;
    Z1/Z2 denorms_to W).
rat(U,U):- integer(U).

X norms_to X/1:- integer(X),!.
X/Y norms_to X/Y:- integer(X),integer(Y),Y>0.

X/1 denorms_to X:- !.
X denorms_to X.

number(X):- X norms_to _ .

sign(X,S,A):- X<0,!,S is -1,A is -X.
sign(X,1,X).

lcm(A,B,P,C,E):- gcd(A,B,D,E,C),P is A*B/D.

gcd(A,B,D,E):- sign(A,_,A1),sign(B,_,B1),
   (B1<A1,!,gcd1(A1,B1,C);
    gcd1(B1,A1,C) ),
   D is A/C,E is B/C.

gcd1(A,0,A):- !.
gcd1(A,B,D):- C is A mod B,gcd1(B,C,D).

expt(X,1,X):- !.
expt(_,0,1):- !.
expt(1,_,1).
expt(X,E,Y):- even(E),E1 is E>>1,expt(X,E1,Y1),Y is Y1*Y1;
   E1 is E-1,expt(X,E1,Y1),Y is X*Y1.
```

PROBLEM 113

Verbal statement:

Consider Spencer Brown's 'The laws of form' (Bantam Books,1973) and write a program to implement his primary arithmetic (ps.12-24).

Logic program:

```
:-op(500,xfy,':').
```

```
read:- repeat,read(X),
       (X = end ;  value(X,Y,Why),
       ttynl,ttynl,display('one has proved '),
       value_is(X,Y,Why),
       ttynl,ttynl,ttynl,fail).

value([[]],0,cancel):- !.
value([]:[],[],condense):- !.

value(0:0,0,_):- !.
value(0:[],[],_):- !.
value([]:0,[],_):- !.
value([],[],_:- !.
value([0],[],_):- !.
value(0,0,_):- !.

value(X:Y,Z,_):- value(X,XX,Why),
                 value(Y,YY,Why2),
                 value(XX:YY,Z,_),
                 since,value_is(X,XX,Why),
                 and,value_is(Y,YY,Why2),
                 value_is(X:Y,Z,_).

value([X],Y,_):- value(X,Z,Why),value([Z],Y,_),
     since,value_is(X,Z,Why),value_is([X],Y,_).

since:- ttynl,ttynl,display('since ').

and:- ttynl,display('and '),

value_is(X,Y,Why):- (Why = not_given;
                     Why = cancel,display('by cancellation');
                     Why = condense,display('by condensation') ),
                     !,ttynl,
                     show(X),display('  '),show(Y).


show(0):- !.
show(X:Y):- !,show(X),display(':'),show(Y).
show([X]):- !,display('['),show(X),display(']').
show(X):- display(X).
```

PROBLEM 114

Verbal statement:

Write an adder for numbers with twenty digits.

Consider: 34832965 + 12469328.

(Suggestion: Consider the number 34832965 as n(0,0,3483,2965))

Logic program:

```
adder(n(A1,B1,C1,D1),n(A,B,C,D),n(A2,B2,C2,D2)):-
        X is (D1 + D) / 100000,D2 is (D1+D)mod 100000,
        Y is (C1 + C + X) / 100000, C2 is (C1+C+X)mod 100000,
        Z is (B1+B+Y) / 100000, B2 is (B1+B+Y) mod 100000,
        A2 is A1+A+Z.
```

Execution:

The result is 47302293 (that is n(0,0,4730,2293))

PROBLEM 115

Verbal statement:

Write a multiplier for numbers with different number of digits.

Logic program:

```
product(n(R1,R2,R3,R4),Pt,t(A,B),Rt,n(Nr1,Nr2,Nr3,Nr4)):-
    N1 is (Pt*Rt) mod 100000, N2 is (Pt*Rt) / 100000,
    P1 is (B*N1) mod 100000, P2 is (B*N1) / 100000,
    P3 is (A*N1) mod 100000, P4 is (A*N1) / 100000,
    P5 is (B*N2) mod 100000, P6 is (B*N2) / 100000,
    P7 is (A*N2) mod 100000, P8 is (A*N2) / 100000,
    Nr4 is (P1*R4) mod 100000, T3 is (P1+R4) / 100000,
    Nr3 is (T3+P2+P3+P5+R3) mod 100000,
    T2 is (T3+P2+P5+R3) / 100000,
    Nr2 is (T2+P4+P6+P7+R2) mod 100000,
    T1 is (T2+P4+P6+P7+R2) / 100000,
    Nr1 is T1+P8+R1.
```

PROBLEM 116

Verbal statement:

Specify the distribution of "*" by "+".

Logic program:

```
multiply(X+Y,Z,X1+Y1):- multiply(X,Z,X1),
                        multiply(Y,Z,Y1).

multiply(Z,X+Y,X1+Y1):- multiply(Z,X,X1),
                        multiply(Z,Y,Y1).
multiply(X,Y,X*Y).
```

3.11 MISCELLANEOUS (nos. 117 to 120)

PROBLEM 117 [PEREIRA et al,1978]

Verbal statement:

Write a PROLOG interpreter in PROLOG, and illustrate the use of a variable goal.

In this mini-interpreter, goals and clauses may be represented as ordinary PROLOG data structures (ie. terms). Terms representing clauses may be specified using the unary predicate 'clause', eg.

```
clause( (grandparent(X,Z):- parent(X,Y),parent(Y,Z)) ).
```

A unit clause may be represented by a term such as:-

```
( parent(John,mary):- true)
```

Logic program:

```
execute(true):- !.
execute((P,Q)):- !,execute(P),execute(Q).
execute(P):- clause((P:-Q)),execute(Q).
execute(P):- P.
```

PROBLEM 118

Verbal statement:

Extend the mini-interpreter considered in the previous problem, in order to deal with the "cut" symbol.

Logic program:

```
execute(true,_).
execute(!,_).
execute(!,cut).
execute((P,Q),V):- !,execute(P,C),
                   (C==cut,V=cut;
                    execute(Q,V)).
execute(P,V):- clause((P:-Q)),
               execute(Q,V),
               (V==cut,!,fail;true).
execute(P,_):- P.
```

PROBLEM 119 [PEREIRA et al,1978]

Verbal statement:

Illustrate the use of the meta-predicates 'var' and '=..'. The procedure call 'variables(Term,L,[])' instantiates variable L to a list of all the variable occurrences in the term Term.
eg. variables(d(U*V,X,DU*V+U*DV),[U,V,X,DU,V,U,DV], []).

Logic program:

```
variables(X,[X|L],L):- var(X),!.
variables(T,L0,L):- T =..[F|A],variables1(A,L0,L).

variables1([T|A],L0,L):- variables(T,L0,L1),variables1(A,L1,L).
variables1([],L,L).
```

PROBLEM 120

Verbal statement:

Define conditional.

Logic program:

```
:-op(1050,xfx,'->').

(P -> Q;R):- P,!,Q.
(P -> Q;R):- !,R.
(P;Q):- P.
(P;Q):- Q.
(P,Q):- P,Q.

true.

:-repeat,read(X),(X=end -> true;
                   X -> write((Yes;)),fail.
                        write((no;)),fail).
```

Execution:

```
24 > 23.
yes

23 > 23.
no

23 >= 23.
yes

end.
```

ACKNOWLEDGEMENTS

We would like to thank the authors from whom we borrow the examples included in this report.

We will be grateful for any comments and corrections regarding these examples, and for any additional programs to be included in the next edition.

BIBLIOGRAPHY

ANDREKA, H. ; NEMETI, I. [1976]
The Generalized Completeness of Horn Predicate Logic as a Programming Language
DAI report no. 21
Univ. of Edinburgh

BALOGH, K. [1978]
On an Interactive Program Verifier for PROLOG Programs
Draft for the Colloquium on Mathematical Logic in Programming, Salgotarjan, Hungary
North-Holland 1979.

BATTANI, G. ; MELONI, H. [1973]
Interpreteur du Language de Programmation PROLOG
Groupe de IA, UER Luminy
Univ. d'Aix-Marseille

BATTANI, G. ; MELONI, H. [1974]
Un Bel Example de PROLOG en Analyse et Synthése
Groupe de IA, UER Luminy
Univ. d'Aix-Marseille

BATTANI, G. ; MELONI, H. [1975]
Mise en Oeuvre des Contraintes Phonologiques Syntaxiques et Semantiques dans un Systeme de Comprehension Automatique de la Parole
Groupe de IA, UER Luminy
Univ. d'Aix-Marseille

BERGER, G. [1979]
Logique conceptuelle
Groupe de IA, UER Luminy, DEA
Univ. d'Aix-Marseille

BERGMAN, M. [1973]
Resolution par la Demonstration Automatique de Quelques Problémes en Integration Symbolique sur Calculateur.
Thése de 3éme cycle
Groupe de IA, UER Luminy
Univ. d'Aix-Marseille

BERGMAN, M. ; KANOUI, H. [1973]
Application of Mechanical Theorem Proving to Symbolic Calculus
Groupe de IA, UER Luminy
Univ. d'Aix - Marseille

BERGMAN, M. ; KANOUI, H. [1975]
SYCOPHANTE: Systeme de Calcul Formel et d'Integration Symbolique sur Ordinateur
Groupe de IA, UER Luminy
Univ. d'Aix-Marseille

BYRD, L. [1979]
The new prolog interpreter
DAI, Working report
Univ. of Edinburgh

BOSSU, G. ; TEMINE, G. [1978]
Un systeme capable de poser des questions a un utilisateur en vue de la construction et de l' interrogation d'une base de données -- CURIEUX
Groupe de IA, UER Luminy
Univ. d'Aix-Marseille

BOYER, R. S. ; MORE, J. S. [1972]
The Sharing of Structure in Theorem-Proving Programs
DAI memo no. 47
Univ. of Edinburgh
and Machine Intelligence 7.

BRAZDIL, P. [1978]
Experimental learning model
Proceed. of AISB78

BRUYNOOGHE, M. [1975]
The Inheritance of Links in a Connection Graph
Report CW2
Applied Mathematics and Programming Dept.
Katholieke Univ. Leuven (Belgium)

BRUYNOOGHE, M. [1976]
An Interpreter for Predicate Logic Programs
Part I : Basic Principles
Report CW 10
Applied Mathematics and Programming Dept.
Katholieke Univ. Leuven (Belgium)

BRUYNOOGHE, M. [1977]
An Interface Between Prolog and Cyber-EDMS
in Logic and Data Bases
Workshop Toulouse

BRUYNOOGHE, M. [1978]
Intelligent Bachtraching for Horn Clause Logic Programs
Applied Mathematics and Programming Dept.
Katholieke Univ. Leuven (Belgium)

BRUYNOOGHE, .M. [1979]
Analysis of dependencies to improve the behaviour of logic programs
Report CW19
Univ. Catholique de Louvaine

BRUYNOOGHE, M. [1979]
Solving combinatorial search problems by intelligent backtracking
Report CW18
Univ. Catholique de Louvaine

BUNDY, A. [1979]
Mathematical reasoning course notes
DAI Working paper
Univ. of Edinburgh

BUNDY, A. ; WELHAM, B. [1977]
Utility Procedures in PROLOG
DAI Occasional paper no. 9
Univ. of Edinburgh

BUNDY, A. ; WELHAM,B. [1979]
Using meta-level descriptions for selective application of multiple rewrite rules in algebraic manipulation
DAI Research paper no. 121
Univ. of Edinburgh

BUNDY, A. et al [1979]
Solving mechanics problems using meta-level inference
DAI Research paper no. 112
Univ. of Edinburgh

CANEGHEM, M. VAN [1977]
Systeme d'Analyse et de Synthese Morphologique en Francais pour l'Exploitation de Banques de Donnees
(Vol. 1,2)
Groupe de IA, UER Luminy
Univ. d'Aix-Marseille

CLARK, K. L. [1977a]
Negation as Failure
Department of Computing and Control
Imperial College

CLARK, K. L. [1977b]
The Synthesis and Verification of Logic Programs
Department of Computing and Control
Imperial College

CLARK, K. L. ; TARNLUND, S. A. [1977]
A First-Order Theory of Data and Programs
Proc. IFIP 77

CLARK, K. ; KOWALSKI, R. [1977]
Predicate Logic as a Programming Language
Department of Computing and Control
Imperial College

CLARK, K. ; DARLINGTON, J. [1978]
Algorithm classification through synthesis
Department od Computing and Control Report no. 78/43
Imperial College

CLARK, K. L. [forthcoming]
Predicate Logic as a Computational Formalism
Ph. D. Thesis, Imperial College

COELHO, H. [1974]
An Inquiry on the Geometry Machine and its Extension with a Review of Previous Work
Working report no. 1 for INVOTAN
LNEC, 1976

COELHO, H. [1976]
On a conversational interface between users and a data base
LNEC

COELHO, H. ; PEREIRA, L. M. [1976]
GEOM: A PROLOG Geometry Theorem Prover
LNEC

COELHO, H. [1977]
Natural Language and Data Bases
LNEC

COELHO, H. [1979a]
Aspectos do Trabalho Desenvolvido em 1978 pelo 'Groupe de IA' da 'Universite' d'Aix-Marseille'. Relatório de uma visita de estudo realizada em Dezembro de 1978
LNEC

COELHO, H. [1979b]
TUGA's Users Guide
LNEC

COELHO, H. [1979c]
Notes on Natural Language Conversations between a program and its users
LNEC

COELHO, H. [1979d]
Context and Conversation: a discussion from a semantic point of view
LNEC

COELHO, H. [1979e]
A program conversing in Protuguese providing a library service
Ph. D. Thesis
Univ. of Edinburgh

COELHO, H. [1980a]
Interacção coloquial com o computador -- para a definição de uma Engenharia da Linguagem
Comunicação ao CPI 80

COELHO, H. [1980b]
O recurso da Lógica
Comunicação ao 1o. Encontro Nacional da Sociedade Portuguesa de Matemática

COELHO, H. [1980c]
Elementos para uma Ensenharia da Linguagem
Tese para Especialista
LNEC

COLMERAUER, A. [1970a]
Total Procedure Relations
JACM vol 17 no. 1, Jan. 1970

COLMERAUER, A. [1970b]
Les systemes-Q ou un formalisme pour analyser et synthetiser des phrases sur ordinateur
Department d'Informatique, Pub. int. no. 43
Univ. de Montreal

COLMERAUER, A. ; DANSEREAU, J. ; HARRIS, B. [1971]
Rapport annuel du project de traduction automatique de l'Université de Montreal
TAUM 71

COLMERAUER, A. et al [1973]
Un Systéme de Communication Homme-Machine en Français
Groupe de IA, UER Luminy
Univ. d'Aix-Marseille

COLMERAUER, A. [1974]
Programmation en Langue Naturelle
Groupe de IA, UER Luminy
Univ. d'Aix-Marseille

COLMERAUER, A. [1975]
Les Grammaires de Metamorphose
Groupe de IA, UER Luminy
Univ. d'Aix-Marseille

COLMERAUER, A. [1977a]
Un Sous-Ensemble Interessant du Français
Groupe de IA, UER Luminy
Univ. d'Aix - Marseille

COLMERAUER, A. [1977b]
A Useful Subset of French
in Logic and Data Bases, Workshop Toulouse

COLMERAUER, A. [1979]
Sur les bases theoriques de Prolog
Groupe de IA, UER Luminy
Univ. d'Aix-Marseille

COLMERAUER, A. ;PIQUE, J. [1979]
About natural logic
Groupe de IA, UER Luminy
Univ. d'Aix-Marseille

COLMERAUER, A. ; KANOUI, H. ; CANEGHEM, M. [1979]
Etude et Realisation d'un Systeme Prolog
Groupe de IA, UER de Luminy
Univ. d'Aix-Marseille

COTTA, J. C. ; SILVA, A. P. [1978]
Interacção com Bases de Dados
LNEC

COTTA, J. C. [1980]
Experiência de utilização da língua natural no acesso a bases de dados
Comunicação ao 1o. CPI80

DAHL, V. ; SAMBUC, R. [1976]
Un Systéme de Banque de Données en Logique du Premier Ordre, en Vue de sa Consultation en Langue Naturelle
Groupe de IA, UER Luminy
Univ. d'Aix-Marseille

DAHL, V. [1977a]
Some Experiences on Natural Language Question-Answering Systems
in Logic and Data Bases, Workshop Toulouse

DAHL, V. [1977b]
Un Systeme Deductif d'Interrogation de Banques de Données en Espagnol
Groupe de IA, UER Luminy
Univ. d'Aix-Marseille

DARVAS, F. ; FUTO, I. ; SZEREDI, P. [1976]
Some applications of teorem proving based machine intelligence in QASR: automatic calculation of molecular properties and automatic interpretation of quantitative structure activity relationships
International Conference on QASR, Suhl, GDR

DARVAS, F. ; FUTO, I. ; SZEREDI, P. [1978]
Logic Based Program System for Predicting Drug Interactions
Int. J. Bio-Medical Computing(9)

DELIYANNI, A. ; KOWALSKI, R. A. [1977]
Logic and Semantic Networks
Department of Computing and Control
Imperial College
in Logic and Data Bases, Workshop Toulouse

DONZ, P. [1979]
Une méthode de transformation et d'optimisation de programmes Prolog: definition et implementation
Groupe de IA, UER Luminy
Univ. d'Aix-Marseille

DWIGGINS, D. [1979]
A knowledge-based automated message understanding methodology for an advanced indications system
Operating Systems

EDER, G. [1976a]
A PROLOG-Like Interpreter for non-Horn Clauses
DAI report no. 26
Univ. of Edinburgh

EDER, G. [1976b]
A System for Cautious Planning
DAI report no. 27
Univ. of Edinburgh

ELBAAMRANI ; MOULAY [1979]
Sur les equations algebriques
Groupe de IA, UER Luminy, DEA
Univ. d'Aix-Marseille

EMDEN, M. van [1974a]
First-Order Predicate Logic as a High-Level Program Language
DAI, MIP-R-106
Univ. of Edinburgh

EMDEN, M. van [1974b]
The Semantics of Predicate Logic as a Programming Language
DAI, Memo 73
Univ. of Edinburgh

EMDEN, M. van [1975]
Programming with Resolution Logic
Research report CS-75-30
DCS, Univ. of Waterloo, Canada

EMDEN, M. van ; KOWALSKI, R. [1976a]
Verification Conditions as Representations for Programs
Research report CS-76-03
DCS, Univ. of Waterloo, Canada

EMDEN, M. van [1976b]
Unstructured Systematic Programming
Research report CS-76-09
DCS, Univ. of Waterloo, Canada

EMDEN, M. van [1976c]
A Proposal for an Imperative Complement to PROLOG
Research report CS-76-39
DCS, Univ. of Waterloo, Canada

EMDEN, M. van [1976d]
Deductive Information Retrieval on Virtual Relational Databases
Research report CS-76-42
DCS, Univ. of Waterloo, Canada

EMDEN, M. van [1976e]
Logic Programs for Querying Relational Databases
Working paper
DCS, Univ. of Waterloo, Canada

EMDEN, M. van [1977]
Computation and Deductive Information Retrieval
Research report CS-77-16
DCS, Univ. of Waterloo, Canada

EMDEN, M. van [1978]
Relational Programming
Research report CS-78-48
DCS, Univ. of Waterloo, Canada

EMDEN, M. van [1980]
Chess-endgame advice: a case study in computer utilization of knowledge
Research report CS-80-05
DCS, Univ. of Waterloo

FUTÓ, I. ; SZEREDI, P. ; DARVAS, F. [1977]
Some Implemented and Planned PROLOG Applications
in Logic and Databases
Workshop Toulouse

FUTÓ, I. ; DARVAS, F. ; CHOLNOKY, E. [1977]
Pratical Applications of an Artificial Intelligence Language
Preprints of the Secound Hungarian Computer
Science Conference, Budapest

GIANNESINI, F. [1978]
Representation du Monde et Linearisation d'un Problème en
Robotique
Groupe de IA, UER Luminy
Univ. d'Aix-Marseille

GIANNESINI, J. [1978]
Generateur de Plans Utilisant une Hierarchie sur un Ensemble
de Buts
Groupe de IA, UER Luminy
Univ. d'Aix-Marseille

GUIZOL, J. [1975]
Synthése du Français a partir d'une Representation en
Logique du Premier Ordre
Groupe de IA, UER Luminy
Univ. d'Aix-Marseille

GUIZOL, J. [1976]
Remarques a Propos de la Synthése du Français
Groupe de IA, UER Luminy
Univ. d'Aix- Marseille

GUIZOL, J. ; MELONI, H. [1976]
PROLOG Modulaire
Groupe de IA, UER Luminy
Univ. d'Aix-Marseille

GUIZOL, J. ; MELONI, H. [1979]
Identification d'evenements pseudo-phonetiques et des
trajectoires fonnantiques pour la reconaissance automatique
de la parole continue
Groupe de IA, UER Luminy
Univ. d'Aix-Marseille

HILL, R. [1974]
LUSH - Resolution and its Completeness
DAI, Memo 78
Univ. of Edinburgh

HOGGER, C. J. [1978a]
Program Synthesis in Predicate Logic
Imperial College

HOGGER, C. J. [1978b]
Goal-oriented Derivation of Logic Programs
Imperial College

HOGGER, C. J. [forthcoming]
Derivation of Logic Programs
Ph. D. Thesis, Imperial College

HORVATH, K.; LABADI, K. ; LAUFER, T. [1977]
QAL1 a logic for question-answering systems
2nd. Hungarian Conference on Computing

JOUBERT, M. [1974]
Un Systeme de Resolution de Problemes a Tendance Naturelle
Groupe de IA, UER Luminy
Univ. d'Aix - Marseille

KANOUI, H. [1973a]
Some aspects of Symbolic Integration via Predicate Logic Programming
3IJCAI

KANOUI, H. [1973b]
Application de la Demonstration Automatique aux Manipulations Algébriques et a l'Integration Formelle sur Ordinateur
Groupe de IA, UER Luminy
Univ. d'Aix - Marseille

KANOUI, H. [1975]
Rapport d'activite du G.I.A. (Integration Symbolique)
Groupe de IA, UER Luminy
Univ. d'Aix - Marseille

KANOUI, H. ; BERGMAN, M. [1977]
Generalized Substitutions
Groupe de IA, UER Luminy
Univ. d'Aix - Marseille

KANOUI, H. ; CANEGHEM, M. van [1979]
Implementing a very high level language on a very low cost computer
Groupe de IA, UER Luminy
Univ. d'Aix-Marseille

KOMOROWSKI, H. Jan [1979]
QLOG interactive environment -- the experience from embedding a generalized Prolog in INTERLISP
Informatics Laboratory Report LITH-MAR-R-79-19
Linkoping Univ.

KOWALSKI, R. ; KUEHNER, D. [1971]
Linear Resolution with Selection Function
AI Journal 2

KOWALSKI, R. [1973a]
An improved Theorem-Proving System for First-Order Logic
DAI, Memo 65
Univ. of Edinburgh

KOWALSKI, R. [1973b]
Predicate Logic as Programming Language
DAI, Memo 70
Univ. of Edinburgh
and Proc. IFIP 74 North-Holland

KOWALSKI, R. [1974a]
A proof Procedure using Connection Graphs
DAI, Memo 74
Univ. of Edinburgh
and JACM Vol. 22 no. 4 - Oct/1975

KOWALSKI, R. [1974b]
Logic for Problem Solving
DAI, Memo 75
Univ. of Edinburgh

KOWALSKI, R. [1976a]
Algorithm = Logic + Control
Department of Computing and Control
Imperial College

KOWALSKI, R. [1976b]
Logic and Data Bases
Department of Computing and Control
Imperial College

KOWALSKI, R. [1977a]
Logic as Programming Language
Visit to N. America, 25 March - 29 April 1977
Department of Computing and Control
Imperial College

KOWALSKI, R. [1977b]
General laws in data description
in Logic and Databases
Workshop Toulouse

KOWALSKI, R. [1979]
Logic for Problem Solving
North-Holland

LICHTMAN, B. M. [1975]
Features of Very High Level Programming with PROLOG
Department of Computing and Control
Imperial College

MARKUSZ, Z. [1977]
How to Design Variants of Flats using Programming Language
PROLOG based on Mathematical Logic
Information Processing, B. Gilchrist (ed.)
IFIP - 1977, North-Holland

MELLISH, C. S. [1977]
An Approach to the GUS Travel Agent Problem using PROLOG
DAI
Univ. of Edinburgh

MELLISH, C. S. [1978a]
Syntax - Semantics Interaction in Naturel Language Parsing
DAI, Working Paper no. 31
Univ. of Edinburgh

MELLISH, C. [1978b]
Preliminary syntactic analysis and interpretation of
Mechanics problems stated in English
DAI Working Paper no. 48
Univ. of Edinburgh

MELONI, H. [1976]
PROLOG - mise en route de l'interpreteur et exercices
Groupe de IA, UER Luminy
Univ. d'Aix - Marseille

MILNE, R. [1979]
A case for deterministic parsing using syntax
DAI Working Paper no. 53
Univ. of Edinburgh

MILNE, R. [1980]
A framework for deterministic parsing using syntax and semantics
DAI Working Paper no. 64
Univ. of Edinburgh

MOORE, J. S. [1973]
Computational Logic: Structure Sharing and Proof of Program Properties - Parts I & II
DAI memo 67
Univ. of Edinburgh

PASERO, R. [1972]
Representation du Français en Logique du Premier Ordre en Vue de Dialoguer avec un Ordinateur
Groupe de IA, UER Luminy
Univ. d'Aix - Marseille

PASERO, R. [1976]
Un essai de Communication Sensée en Langue Naturelle
Groupe de IA, UER Luminy
Univ. d'Aix - Marseille

PEREIRA, F. ; WARREN, D. H. [1978]
Definite Clause Grammars Compared with Augmented Transition Network
DAI
Univ. of Edinburgh

PEREIRA, F. [1979]
Extraposition Grammars
DAI, Working paper n.59
Univ. of Edinburgh

PEREIRA, L. M. [1977]
PROLOG, uma Linguagem de Programação em Lógica
LNEC

PEREIRA, L. M. [1978]
PROLOG, Linguagem de Resolução de Problemas em Lógica (Part 1 & 2)
in Informática, Volume 2, no. 4 1978, Volume 2, no. 6, 1979

PEREIRA, L. M. [1979a]
Cálculo da distribuição das famílias portuguesas por escalões de rendimento
LNEC

PEREIRA, L. M. [1979b]
Backtracking Intellisently in AND/OR trees
Universidade Nova deLisboa

PEREIRA, L. M. ; MONTEIRO, L. F. [1978]
The Semantics of Parallelism and Co-Routining in Logic Programs
LNEC
and Colloquium on Mathematical Logic in Programming, Salgotarjan, Hungary
North-Holland (to be published)

PEREIRA, L. M. ; PEREIRA, F. ; WARREN, D. [1978]
User's Guide to DECsystem 10 PROLOG
LNEC

PEREIRA, L. M. ; COELHO, H. [1979]
A Lógica, Instrumento de Comunicação em Português com o Computador
LNEC

PEREIRA, L. M. ; PORTO, A. [1979]
Intellisent Backtracking and sidetracking in Horn clause programs - the theory
Universidade Nova de Lisboa, Lisbon

PEREIRA, L.M.; PORTO, A. [1979]
Intellisent backtracking and sidetracking in Horn clause programs - the implementation
Universidade Nova de Lisboa, Lisbon

PEREIRA, L. M. ; PORTO, A. [1980]
Selective bactracking for logic programs
Departamento de Informatica, CIUNL no. 1/80
Univ. Nova de Lisboa

PIQUE, J. [1978]
Interrogation en Français d' une base de données relationelle
Groupe de IS, UER Luminy, DEA
Univ. d'Aix-Marseille

ROBERTS, G.M. [1977]
An Implementation of PROLOG
DCS, Univ. of Waterloo, Canada

RODRIGUEZ, M. [1978]
Un Systeme Patient d'Aide a la Conception: JOB
these de 3eme cycle
Groupe de IA, UER Luminy
Univ. d'Aix - Marseille

ROUSSEL, P. [1972]
Definition et Traitement de l'Esalité Formelle en Demonstration Automatique
Groupe de IA, UER Luminy
Univ. d'Aix - Marseille

ROUSSEL, P. [1975]
PROLOG - Manuel de Réference et d'Utilisation
Groupe de IA, UER Luminy
Univ. d'Aix - Marseille

SABATIER, P. [1978]
Ellipsis, grammaire interpretative des constructions elliptiques du Français
Groupe de IA, UER Luminy, DEA
Univ. d'Aix-Marseille

SANTOS, M.J. [1979]
Interface gráfica em Prolog
LNEC

SILVA, A.P. ; COTTA, J. C. [1978]
The Travel Problem Revisited
LNEC

SIMOES, C. M. D. [1980]
CARSIM: um programa para listar encomendas
CTT/TLP

SIMONET, M. [1978]
Transcription de la W-Grammaire d'un Language Simple
Lab. d'Informatique et Mathematique Appliques
Grenoble

SZEREDI, P. [1977]
PROLOG - A Very High Level Language Based on Predicate Logic
Preprints of Second Hungarian Computer Science
Conference, Budapest

TÄRNLUND, S. [1975]
Logic Information Processing
Report TRITA-IBADB - 1034
Dept. of Inf. Processing
Univ. of Stockholm

TÄRNLUND, S. [1976a]
A Logical Basis for Data Bases
Report TRITA - IBADB no. 1029
DCS - Royal Institute of Technology Stockholm
in Logic and Data Bases - Workshop Toulouse

TÄRNLUND, S. [1976b]
Programming as a Deductive Method
Report TRITA - IBADB - 1031
Dept. of Inf. Processing
Univ. of Stockholm

TÄRNLUND, S. [1977]
Horn Clause Computability
BIT 17 (1977) p.ps 215-226

TÄRNLUND, S. [1978]
An Axiomatic Data Base Theory
Dept of Inf. Processing
Univ. of Stockholm

TOWNSHEND, H. [1979]
PRAMS scheme and Prolog computer program
DAI draft
Univ. of Edinburgh

WARREN, D. H. D. [1974a]
WARPLAN - a System for Generating Plans
DAI, memo 76
Univ. of Edinburgh

WARREN, D. H. D. [1974b]
PROLOG notes
notes from lectures in November 74
(unpublished)

WARREN, D. H. D. [1975a]
A User's Guide to PROLOG Supervisor SVW
(Marseille Version)
DAI (unpublished)
Univ. of Edinburgh

WARREN, D. H. D. [1975b]
Examples for WARPLAN and PROLOG: the car assembly problem
and two scene analysis problem
(unpublished)

WARREN, D. H. D. [1976a]
Generating Conditional Plans and Programs
AISB

WARREN, D. H. D. [1976b]
Machine Code Synthesis (SRC proposal)
DAI (unpublished)
Univ. of Edinburgh

WARREN, D. H. D. [1977a]
Implementing PROLOG - Compiling Predicate Logic Programs
Vol 1 & 2 - DAI, Research report no. 39
Univ. of Edinburgh

WARREN, D. H. D. [1977b]
Logic Programming and Compiler writing
DAI, report no. 44
Univ. of Edinburgh

WARREN, D. H. D. [1979]
Prolog on the DEC-system-10
DAI Research Paper no. 127
Univ. of Edinburgh

WARREN, D. ; PEREIRA, L. M. ; PEREIRA, F. [1977]
PROLOG - The Language and its Implementation Compared with
LISP
LNEC
and SIGART/SIGPLAN Symposium, August 1977
Rochester

WELHAM, R. [1976]
Geometry Problem Solving
DAI, research report no. 14
Univ. of Edinburgh

APPENDIX 1

PROLOG IMPLEMENTATIONS

Computer System	Programming Language	Implementation	Year	Authors	Site or University
IBM 360	ALGOL - W	interpreter	1970	P.Roussel	Marseille
IBM 360	FORTRAN	interpreter	1972	Battani, Melloni	Marseille
ICL-1903/A ICL-105/E	CDL	interpreter	1975	Szeredi et al.	Budapest
IBM 370/158- VM/CMS	ASSEMBLER	interpreter	1976	Roberts	Waterloo
Solar Exorciser	ASSEMBLER	interpreter	1976	P.Roussel	Marseille
	PASCAL	interpreter	1976	Bruynooghe	Lovaina
IBM 370	CDL	interpreter	1976	Szeredi et al.	Budapest
ICL 4/70 ICL 1903A	"	interpreter	1976	Szeredi et al.	Budapest
Honeywell Bull 66/20	" "	interpreter	1976	Szeredi et al.	Budapest
EMG 840	"	interpreter	1976	Szeredi et al.	Budapest
ODRA 1304	"	interpreter	1976	Szeredi et al.	Budapest
RIAD R22	"	interpreter	1977	Szeredi et al.	Budapest
DEC-10/20	MACRO-10 and PROLOG	compiler and interpreter	1977	D.Warren F.Pereira L.Pereira	Edinburgh and Lisbon
PDP11/60 UNIX	ASSEMBLER	interpreter	1978	C.Mellish	Edinburgh
EXORCISER	Candide	interpreter	1979	Colmerauer H.Kanoui M.Caneghan	Marseille

ICL 4/75 EMAS	IMP	interpreter	1979	Damas	Edinburgh
PDP-11	FORTH+ SNOBOL	compiler	1979	Dwiggins	Los Angel.
CDC6600					
IBM/370	PASCAL(1)	interpreter	1979	McCabe	London (IC)

---

(1) This version differs significantly from other systems in the control primitives.

APPENDIX 2

LIST OF PROLOG APPLICATIONS

PROBLEM DOMAIN	AUTHORS	YEAR
I) PURE RESEARCH		
Plane geometry	Welham	1976
	Coelho & Pereira	1976
Mechanics	Bundy et al	1979
Symbolic calculus	Kanoui	1973
	Bergman & Kanoui	1975
	Kanoui	1975
Natural language understanding	Colmerauer	1971
	Pasero	1972
	Colmerauer & Kanoui	1973
	Roussel & Pasero	1973
	Colmerauer	1974
	Colmerauer	1975
	Guizol	1975
	Pasero	1976
	Dahl	1977
	Pique	1978
	Mellish	1978
	Pereira & Warren	1979
	Milne	1979
	Coelho	1979
Speech understanding	Batani & Meloni	1975
Learning	Brazdil	1978
Knowledge engineering: Chess	Emden	1980
Robotics	Warren	1974
	Giannesini	1978

II) PRACTICAL PROGRAMS		
Compiler writing	Colmerauer Warren	1975 1977
Interpreter writing	Pereira & Porto Byrd	1979 1979
Computer utilities	Battani & Meloni	1975
Travel agent problem	Mellish Silva & Cotta	1976 1978
Computer catalogue	Dahl	1976
Plotter programs generation	Darvas	1976
Pollution control	Darvas	1976
Pesticide information	Darvas	1976
Statistics	Darvas	1976
Flat design	Markusz	1977
Architecture	Rodriguez	1978
Civil engineering legislation	Cotta & Silva	1978
Drug interaction prediction	Darvas & Futo' & Szeredi	1978
Carcinogenic activity	Darvas	1978
GT-42 Picture book	Santos	1979
Distribution of portuguese families through a scale of income	Pereira	1979
Pre-registration appointments	Townshend	1979
List of equipment	Simoes	1980

APPENDIX 3

PROLOG RELEVANT ADDRESSES

BUDAPEST

NIM IGUSZI
1363 Budapest
P.U.B. 33
Hungary

EDINBURGH

Department of Artificial Intelligence University of Edinburgh
8. Hope Park Sq., Meadow Lane
Edinburgh EH8
Scotland

LEUVEN

Applied Mathematics and Programming Dept.
Katholieke Univ.
Leuven
Belgium

LISBON

Centro de Informática
Laboratório Nacional de Engenharia Civil
101 Av. do Brasil
1799 Lisboa Codex
Portugal

LISBON

Departamento de Engenharia Informática
Universidade Nova de Lisboa
Seminário dos Olivais
Quinta do Cabeço
1800 Lisboa
Portugal

LONDON

Department of Computing and Control Imperial College
University of London
180 Queens Gate
London SW7
England

MARSEILLE

Groupe d'Intellisence Artificielle UER Scientifique
de Luminy
70 route léon-lachaup
13009 Marseille
France

STOCKHOLM

Department of Computer Science University of Stockholm
106 91 Stockholm
Sweden

WATERLOO

Computer Science Department University of Waterloo
Waterloo, Ontario
Canada

APPENDIX 4

PROLOG BUILT-IN PROCEDURES

Built-in procedures are also referred to as evaluable predicates.

4.1 INPUT / OUTPUT

A total of fourteen I/O streams may be open at any one time for input and output. An extra stream is available, for input and output to the user's terminal. A stream to a file F is opened for input by the first "see(F)" executed. F then becomes the current input stream. Similarly, a stream to file H is opened for output by the first "tell(H)" executed. H then becomes the current output stream. Subsequent calls to "see(F)" or to "tell(H)" make F or H the current input or output stream, respectively. Any input or output is always to the current stream.

When no input or output stream has been specified, the standard ersatz file 'user', denoting the user's terminal, is utilized for input and/or output. Terminal output is only displayed after a newline is written or 'ttyflush' is called. When the terminal is waiting for input on a new line, the prompt '|' is displayed.

When the current input and/or output stream is closed, the user's terminal becomes the current input and/or output stream.

No file except the ersatz file 'user' can be simultaneously open for input and output.

A file is referred to by its name, written as an atom, eg.

```
myfile
'123'
'DATA.LST'
'DTA1:ABC.PL'
```

Note that reference to directories other than the user's is not possible at present.

consult(F)

Instructs the interpreter to read file F. When a directive is read it is immediately executed. When a clause is read it is put after any clauses already read by the interpreter for that procedure. The consulted file may define its own character convention ('LC' or 'NOLC') without affecting the convention prevailing outside.

reconsult(F)

Like 'consult' except that any procedure defined in the

"reconsulted" file erases any clauses for that procedure already present in the interpreter. 'reconsult', used in conjunction with 'save' and 'restore', facilitates the amendment of a program without having to consult again from scratch all the files which make up the program. The file "reconsulted" is normally a temporary "patch" file containing only the amended procedure(s). Note that it is possible to call 'reconsult(user)' and then enter a patch directly on the terminal (ending with ":-end." or ^Z). This is only recommended for small, tentative patches.

see(F)

File F becomes the current input stream.

seeing(F)

F is unified with the name of the current input file.

seen

Closes current input stream.

tell(F)

File F becomes the current output stream.

telling(F)

F is unified with the name of the current output file.

told

Closes the current output stream.

close(F)

File F, currently open for input or output, is closed.

read(X)

The next term, delimited by a "fullstop" (ie. a '.' followed by <cr> or a space), is read from the current input stream and unified with X. The syntax of the term must accord with current operator declarations. If a call 'read(X)' causes the end of the current input stream to be reached, X is unified with the term ':-end.'. Further calls to 'read' for the same stream will then cause an error failure.

write(X)

The term X is written to the current output stream according to current operator declarations.

nl

A new line is started on the current output stream.

display(X)

The term X is displayed on the terminal in standard parenthesised prefix notation.

ttynl

A new line is started on the terminal and the buffer is flushed.

ttyflush

Flushes the terminal output buffer.

ttyget0(N)

N is the ASCII code of the next character input from the terminal.

ttyget(N)

N is the ASCII code of the next non-blank printable character from the terminal.

ttyskip(N)

Skips to just past the next ASCII character code N from the terminal. N may be an integer expression.

ttyput(N)

The ASCII character code N is output to the terminal. N may be an integer expression.

get0(N)

N is the ASCII code of the next character from the current input stream.

get(N)

N is the ASCII code of the next non-blank printable character from the current input stream.

skip(N)

Skips to just past the next ASCII character code N from the current input stream. N may be an integer expression.

put(N)

ASCII character code N is output to the current output stream. N may be an integer expression.

tab(N)

N spaces are output to the current output stream. N may be an integer expression.

putatom(X)

The name of atom X is output to the current output stream.

fileerrors

Undoes the effect of 'nofileerrors'.

nofileerrors

After a call to this predicate, the I/O error conditions "incorrect file name...", "can't see file...", "can't tell file..." and "end of file..." cause a call to 'fail' instead of the default action, which is to type an error message and then call 'abort'.

rename(F,N)

If file F is currently open, closes it and renames it to N. If N is '[]', deletes the file.

log

Enables the logging of terminal interaction to file PROLOG.LOG. It is the default.

nolog

Disables the logging of terminal interaction.

4.2 ARITHMETIC

Arithmetic is performed by built-in procedures which take as arguments integer expressions and evaluate them. An integer expression is a term built from evaluable functors, integers and variables. At the time of evaluation, each variable in an integer expression must be bound to an integer, or, for the interpreter ONLY, to an integer expression. Although Prolog integers must be in the range -2^17 to 2^17-1, the integers in arguments to arithmetic procedures and the intermediate results of the evaluation may range from -2^35 to 2^35-1.

Each evaluable functor stands for an arithmetic operation. The evaluable functors are as follows, where X and Y are integer expressions:

X+Y	integer addition
X-Y	integer subtraction
X*Y	integer multiplication
X/Y	integer division
X mod Y	X modulo Y
-X	unary minus
X/\Y	bitwise conjunction
X\/Y	bitwise disjunction

\X bitwise negation

X<<Y bitwise left shift of X by Y places

X>>Y bitwise right shift of X by Y places

!(X) the number in the range 0 to 2^18-1 which is equal to X modulo 2^18

$(X) the number in the range -2^17 to 2^17-1 which is equal to X modulo 2^18

[X] evaluates to X if X is an integer (therefore eg. "A" behaves within arithmetic expressions as an integer constant which is the ASCII code for letter A)

The arithmetic built-in procedures are as follows, where X and Y stand for arithmetic expressions, and Z for some term:

Z is X

Integer expression X is evaluated and the result, reduced modulo 2^18 to a number in the range -2^17 to 2^17-1, is unified with Z. Fails if X is not an integer expression.

X=:=Y

The values of X and Y are equal.

X=\=Y

The values of X and Y are not equal.

X < Y

The value of X is less than the value of Y.

X > Y

The value of X is greater than the value of Y.

X =< Y

The value of X is less than or equal to the value of Y.

X >= Y

The value of X is greater than or equal to the value of Y.

4.3 CONVENIENCE

P,Q

P and Q

P;Q

P or Q

true

Always succeeds.

X=Y

Defined as if by clause ' Z=Z. '.

length(L,N)

L must be instantiated to a list of determinate length. This length is unified with N.

4.4 EXTRA CONTROL

See Section 2.3.

not(P)

If the goal P has a solution, fail, otherwise succed. It is defined as if by:

```
not(P):- P, !, fail.
not(_).
```

Not yet available for compiled code.

P -> Q;R

Analogous to

"if P then Q else R"

ie. defined as if by:

```
P -> Q;R :- P, !, Q.
P -> Q;R :- R.
```

Not yet available for compiled code.

P -> Q

When occuring other than as one of the alternatives of a disjunction, is equivalent to:

```
P -> Q;fail.
```

Not yet available for compiled code.

repeat

Generates an infinite sequence of bactracking choices. It behaves (but doesn't use store!) as if defined by the clauses:

```
repeat.
repeat :- repeat.
```

fail

Always fails.

abort

Aborts the current execution.

4.5 META-LOGICAL

var(X)

Tests whether X is currently instantiated to a variable.

nonvar(X)

Tests whether X is currently instantiated to a non-variable term.

atom(X)

Checks that X is currently instantiated to an atom (ie. a non-variable term of arity 0, other than an integer).

integer(X)

Checks that X is currently instantiated to an integer.

atomic(X)

Checks that X is currently instantiated to an atom or integer.

X == Y

Tests if the terms currently instantiating X and Y are literally identical (in particular, variables in equivalent positions in the two terms must be identical).

X \== Y

Tests if the terms currently instantiating X and Y are not literally identical.

functor(T,F,N)

The principal functor of term T has name F and arity N, where F is either an atom or, provided N is 0, an integer. Initially, either T must be instantiated to, respectively, either an atom and a non-negative integer or an integer and 0. If these conditions are not satisfied, an error message is given. In the case where T is initially not instantiated to a variable, the result of the call is to instantiate T to the most general term having the principal functor indicated.

arg(I,T,X)

Initially, I must be instantiated to a positive integer and T to a compound term. The result of the call is to unify X with the Ith argument of term T. (The arguments are numbered from 1 upwards.) If the initial conditions are not satisfied or I is out of range, the call merely fails.

X=..Y

Y is a list whose head is the atom corresponding to the principal functor of X and whose tail is the argument list of that functor in X. eg. :

```
product(0,N,N-1) =.. [product,0,N,N-1]
N-1 =.. [-,N,1]
product =.. [product]
```

If X is instantiated to a variable, then Y must be instantiated to a list of determinate length whose head is atomic (ie. an atom or integer).

name(X,L)

If X is an atom or integer then L is a list of the ASCII codes of the characters comprising the name of X. eg.:

```
name(product,[112,114,111,100,117,99,116])
```

ie. name(product,"product")

```
name(1976,[49,57,55,54])

name(:-,[58,45])

name([],"[]")
```

If X is instantiated to a variable, L must be instantiated to a list of ASCII character codes. eg.:

```
?-name(X,[58,45]).

X = :-

?-name(X,":-").

X = :-
```

call(X)

If X is instantiated to a term which would be acceptable as body of a clause, the goal 'call(X)' is executed exactly as if that term appeared textually in place of 'call(X)'. In particular, any cut ('!') occurring in X is interpreted as if it occured in the body of the clause containing 'call(X)', unless that clause is a compiled clause, in which case only the alternatives in the execution of X are cut. If X is not instantiated as described above, an error message is printed and 'call' fails.

X

(where X is a variable) Exactly the same as 'call(X)'.

assert(C)

The current instance of C is interpreted as a clause and is added to the current interpreted program (with new private variables replacing any uninstantiated variables). The position of the new clause within the procedure concerned is implementation-defined. C must be instantiated to a non-variable.

asserta(C)

Like 'assert(C)', except that the new clause becomes the first clause for the procedure concerned.

clause(P,Q)

P must be bound to a non-variable term, and the current interpreted program is searched for clauses whose head matches P. The head and body of those clauses are unified with P and Q respectively. If one of the clauses is a unit clause, Q will be unified with 'true'.

assertz(C)

Like 'assert(C)', except that the new clause becomes the last clause for the procedure concerned.

retract(C)

The first clause in the current interpreted program that

matches C is erased. C must be initially instantiated to a non-variable, and becomes unified with the value of the erased clause. The space occupied by the erased clause will be recovered when instances of the clause are no longer in use. On backtracking it will erase the next clause that matches C.

retractall(P)

All clauses in the current interpreted program whose head matches P are 'retract'ed. P must be bound to a non-variable term.

listing(A)

Lists in the current output stream all the interpreted clauses for predicates with naome A, where A is bound to an atom.

listing

Lists in the current output stream all the clauses in the current interpreted program.

NOTE: If a clause contains any atom or functor whose name has to be written in quotes, the listing of that clause will be still readable, but syntatically incorrect. Otherwise, clauses listed to a file by 'listing(A)' or 'listing' can be consulted back.

numbervars(X,N,M)

Unifies each of the variables in term X with a special term, so that write(X) prints each of these variables as 'I', where the Is are consecutive integers from N to M-1. N must be instantiated to an integer.

ancestors(L)

Unifies L with a list of ancestor goals for the current clause. The list starts with the parent goal and ends with the most recent ancestor coming from a 'call' in a compiled clause. Not available for compiled code.

subgoal_of(S)

The goal 'subgoal_of(S) is equivalent to the sequence of goals:

```
ancestors(L),in(S,L)
```

where the predicate 'in' succesively matches its first argument with each of the elements of its second argument. Not available for compiled code.

4.6 INTERNAL DATABASE

These predicates remain in the system purely for compatibility reasons, and will be removed at some future date.

record(X)

The current instance of X is 'recorded' in the internal database at some implementation-defined position in the sequence of terms which constitutes the internal database (with new private variables replacing any uninstantiated variables). X must be instantiated to a non-variable.

recorda(X)

Like 'record(X)', except that the new term is 'recorded' at the 'top' of the internal database.

recordz(X)

Like 'record(X)', except that the new term is 'recorded' at the 'bottom' of the internal database.

?(X)

The internal database is searched for previously 'recorded' terms which match the current instance of X (which must not be a variable). These terms are successively unified with

X in the order in which they are recorded in the internal database.

recorded(X,P)

The database is searched for previously 'recorded' terms that match the current instance of X (which must not be a variable). These terms are successively unified with X in the order in which they are recorded in the internal database. P is unified with a 'pointer' which identifies the 'recorded' term matching X. (A 'pointer' is a term whose internal structure is implementation-defined).

instance(P,X)

X is unified with the database term identified by 'pointer' P.

erase(P)

The database term identified by 'pointer' P is erased from the internal database. The space occupied by the erased term will be recovered when instances of the term are no longer in use.

eraseall(X)

All the database terms matching the current instance of X are 'erased', in the sense of 'erase(_)'.

4.7 ENVIRONMENTAL

'NOLC'

Establishes the "no lower-case" convention.

'LC'

Establishes the "full character set" convention. It is the default setting.

trace

Enable trace.

notrace

Disable trace. It is the default setting.

leash

Enable 'leashed' mode for tracing. It is the default setting.

unleash

Disable 'leashed' mode for tracing.

op(priority,type,name)

Treat name name as an operator of the stated type and priority. name may also be a list of names in which case all are to be treated as operators of the stated type and priority.

break

Causes the current execution to be interrupted at the next interpreted procedure call. Then the message " % break: " is displayed. The interpreter is then ready to accept input as though it was at top level. To close the break and resume the execution which was suspended, the command " :-end. " or Z must be typed. Execution will be resumed at the procedure call where it had been suspended. Alternatively, the suspended execution can be aborted by giving the command " :-abort. ".

save(F)

The system saves the current state of the system into file F.

restore(F)

The superviser is returned to the system state previously saved to file F.

maxdepth(D)

Positive integer D specifies the maximum depth, ie. invocation level, beyond which the system will induce an automatic failure. Top level has zero depth. This is useful for guarding against loops in an untested program, or for curtailing infinite execution branches.

depth(D)

Integer D will give indication of current level of invocation.

gcguide(N)

N must be instantiated to an integer from 0 to 512, indicating the desirable threshold of global stack pages below which garbage collection should be avoided if possible. The default is N=6.

gc

Enables garbage collection of the global stack (the default).

nogc

Disables garbage collection of the global stack.

trimcore

Reduces free space on the stacks and trail as much as

possible and, in virtual memory Monitors only, releases core no longer needed, thereby reducing the size of the low segment. The interpreter automatically calls 'trimcore' after each directive at top-level, after an 'abort' and after a (re)consult.

statistics

Display on the terminal statistics relating to core usage, run time, garbage collection of the global stack and stack shifts.

statistics(K,V)

This allows a program to gather various execution statistics. For each of the possible keys K, V is unified with a list of values, as follows:

Key	Values		
core	low_segment	high_segment	
heap	size	free	
global_stack	size	free	
local_stack	size	free	
trail	size	free	
runtime	since start of PROLOG	since previous 'statistics'	
garbage_collection	no. of GCs	words freed	time spent
stack_shifts	no. of local shifts	no. of trail shifts	time spent

Times are in miliseconds, sizes of areas in words. If a time exceeds 129.071 sec., it will be returned as a term:

xwd(T1,T2)

representing:

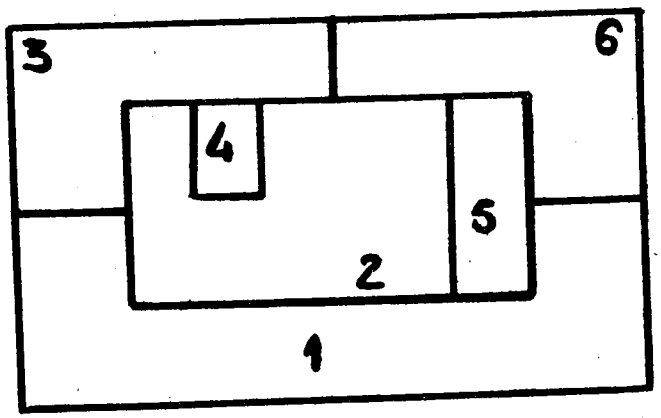
T1*2^18 + T2 mod 2^18

Note that such a term occurs in an interpreted arithmetic expression, it will be evaluated correctly.

INDEX

ERRATA

Page	Line	Where is	Must be
18	8	tail [H1:T1]	tail LT1
	27,28	:-sort ([c....	:-sort ([c,q,w,e,r,t,y,u,i,o,p, a,s,d,f,g,h,j,k,l,z,x,c,v,b,n,m] ,S), write (S).
24	21	compact (L1,	compact ([L1,
24	43		linked (X,Y,):-cat (X,_,Y,_). linked (X,Z,):-cat (X,_,Y,_), linked (Y,Z).
28	20	U2),Z K	U2),Z,K
	20	U2),Z):-	U2),Z)):-
	24	K2),void).	K2),void)).
34	21/22	greater or equal ones	greater ones
35	19	2,[],3	2,[]),3
35	21	18,[],19	18,[]),19
41	5	takes (m:adiri	takes (m.:adiri
55	22	fact (ok (F1,N)yes)	fact (ok(F1,N),yes)
	33	discover (lit,Val)	discover (Lit,Val)
	33	fact (lit,Val)	fact (Lit,Val)
	34	discover (lit,Val)	discover (Lit,Val)
	34	askclient (lit,	askclient (Lit,
56	9	interpret (A,B,Ø)	interpret (A.B,Ø)
	10	dealwith (A,B,1)	dealwith (A.B,1)
	12	dealwith (A,B,_)	dealwith (A.B,[])
	20		output ( ).
63	9/10		



	10	below	above
93	24	preserved (_,true).	preserved (_,_).
	32	retrace 1 (true,V,C,true).	retrace 1 (true,V,C,true). retrace 1 (X&P,V,C,X&P1):- retrace 1 (P,V,C,P1).
125	10		np (X,Y): - noun (X,Y).