KNOWLEDGE

REPRESENTATION

P. Gribomont

Introduction to artificial intelligence

Introduction to logic programming

PROLOG

Problem solving in PROLOG

P. Gochet et P. Gribomont, *LOGIQUE, méthodes pour l'intelligence artificielle*, Hermès, Paris, 2000 (Chapitres 10, 11 et 12)

L. Sterling and E. Shapiro, *The Art of Prolog*, MIT Press, 1994 (2nd ed).

I. Bratko, *Prolog Programming for Artificial Intelligence*, Prentice Hall, 2000 (3rd ed).

Terms, atomic terms

Objects are (represented by) terms

Atoms (atomic terms) are constants and variables

For prolog, data terms have no specific meaning

`0`, `-3` et `2.567`: numbers (constants)
`abc`, `+` et `rs232`: symbols (constants)
`X`, `Ys`, `Truc`, `_ght54d`: variables

Compound terms, ground terms

functor: allows to build compound terms
arity: any natural number

`m(a,b,c)`,
`m(f(a,b),a(c,g(2,d)))`
`+(3,*(4,5))`
ground terms do not contain variables

constant: atomic ground term
A 0-ary functor is a constant

Compound terms are trees

The tree `+(3,*(4,5))` and the number 23 are not equal !

Terms, lists, pairs

Lists are "anonymous" terms:

`p(gates,bill,45)` is a term,

`[gates,bill,45]` is a list,

standing for the term

`.(gates,.(bill,.(45,[])))`

also written

`[gates | [bill | [45 | []]]]`

`.` is a binary functor

Empty list: `[]`

Terms and objects I

ground term (hexapawn position)

```
hp(c(1,1,e),c(1,2,b),c(1,3,b),
   c(2,1,b),c(2,2,w),c(2,3,e),
   c(3,1,w),c(3,2,e),c(3,3,w))
```

`w` for white pawn,

`b` for black pawn,

`e` for empty cell

non-ground term:

`m(f(a,X),f(X,b),Y)`

ground instances:

`m(f(a,a),f(a,b),c)`

`m(f(a,g(c)),f(g(c),b),g(c))`

Terms and objects II

- Prolog objects are the ground terms.
- A term can be viewed as the set of its ground instances.
- A term can represent any of its ground instances.
- A ground term is its own unique instance.
- A variable is the most general term,
  associated with the set of all ground terms.

Facts and relations

The fact

```
r(a,b,c).
```

means that the n-uple

```
(a,b,c)
```

belongs to the ternary relation

```
r
```

$n = 0$: proposition:

```
z.
```

Facts and descriptions

A graph description:

```
node(g,n1).        arc(g,n1,n3).
node(g,n2).        arc(g,n2,n3).
node(g,n3).        arc(g,n1,n5).
node(g,n4).        arc(g,n4,n5).
node(g,n5).        arc(g,n5,n6).
node(g,n6).        arc(g,n2,n4).
                   arc(g,n4,n6).
```

Derived facts, rules

```
path(g,[n1,n5,n6]).
```

if `X` is a node,
then `[X]` is a path
("basis clause")

if `(X,Y)` is an arc and if `[Y|Ys]` is a path,
then `[X,Y|Ys]` is a path
("induction clause")

Rules and clauses

```
path(G,[X]) :- node(G,X).

path(G,[X,Y|Ys]) :- node(G,X), arc(G,X,Y),
                     path(G,[Y|Ys]).
```

```
A :- B,C,D.
```

if `B`, `C` and `D` are true, then `A` is true

Fact = absolute, non-conditional rule

We write "`A.`" instead of "`A :- .`"

Queries and answers

- What are the nodes of graph `g`?
- Is `[n1,n5,n6]` a path in `g`?
- Which paths in `g` start from `n1` or `n2` and end in `n6`?

Rule instance:

```
path(g,[n1,n5|[n6]]) :-
  node(g,n1), arc(g,n1,n5), path(g,[n5|[n6]]).
```

also written as:

```
path(g,[n1,n5,n6]) :-
  node(g,n1), arc(g,n1,n5), path(g,[n5,n6]).
```

Reduction, subqueries

The main query reduces to three subqueries:

```
node(g,n1)
arc(g,n1,n5)
path(g,[n5,n6])
```

Two subqueries vanish (they are facts).

The main query reduces to the subquery

```
path(g,[n5,n6]).
```

Prolog program, search tree and deduction

`append([],Bs`^{3}`,Bs`^{3}`)` (basis clause – `append`)

`append([],[b],[b])` (instantiation, line 1)

`append([A`^{2}`|As`^{2}`],Bs`^{2}`,[A`^{2}`|Cs`^{2}`])` if `append(As`^{2}`,Bs`^{2}`,Cs`^{2}`)`
(induction clause – `append`)

`append([a],[b],[a,b])` if `append([],[b],[b])` (instantiation, line 3)

`append([a],[b],[a,b])` (deduction, lines 2 and 4)

`prefix(Xs`^{1}`,Ys`^{1}`)` if `append(Xs`^{1}`,Zs`^{1}`,Ys`^{1}`)` (clause – `prefix`)

`prefix([a],[a,b])` if `append([a],[b],[a,b])` (instantiation, line 6)

`prefix([a],[a,b])` (deduction, lines 5 and 7)

Unit resolution

S_0: set of Horn clauses (Horn set)
clause $\simeq$ disjunctive set of literals
$\Box$: empty clause, *false*
Non-deterministic algorithm

$\{S = S_0\}$
while $\Box \notin S$ do
    select p and c such that
        $p \in S$ positive unit clause,
        $c \in S$ such that $\neg p \in c$;
    $r := c \setminus \{\neg p\}$;
    $S := (S \setminus \{c\}) \cup \{r\}$.
$\{S = S_f\}$

Properties of unit resolution

Invariant: $\mathcal{M}(S) = \mathcal{M}(S_0)$

if U is a set of formulas, then sets $S_n = U \cup \{p\,, \neg p \vee X\}$ and $S_{n+1} = U \cup \{p\,, X\}$ are logically equivalent.

Termination: strictly decreasing set of negative literals.
Computation time: quadratic (number of negative literals).
(Why is general resolution exponential in worst case?)

Normal termination: $\square \in S$.
Inconsistency: $S_0 \simeq S_f$ et $\square \in S_f$.

Abortion: cannot select p and c.
Consistency: S_0 and S_f allow the canonical model.

Canonical model

In case of abortion, a model C can be defined:
$C(p) = \mathbf{T}$ iff p is a unit clause of S_f.

C is the canonical, minimal model of S_f and S_0.

For each clause $c \in S_f$ (unit or not), $C(c) = \mathbf{T}$.

Minimality: if D is another model of S_f,
then if $C(p) = \mathbf{T}$, then $D(p) = \mathbf{T}$.

$C(p) = \mathbf{T}$ iff $S_0 \models p$.

Three basic programs I

```
append([],Bs,Bs).
append([A|As],Bs,[A|Cs]) :- append(As,Bs,Cs).

select(A,[A|As],As).
select(B,[A|As],[A|Bs]) :- select(B,As,Bs).

permutation([],[]).
permutation([A|As],Bs) :- permutation(As,Cs), select(A,Bs,Cs).
```

Three basic programs II

```
?- append([a,b],[c],L).
L = [a, b, c].

?- append(Xs,Ys,[a, b, c]).
Xs = [],
Ys = [a, b, c] ;
Xs = [a],
Ys = [b, c] ;
Xs = [a, b],
Ys = [c] ;
Xs = [a, b, c],
Ys = [] ;
false.
```

Three basic programs III

```
?- select(x,Xs,[a, b, c]).
Xs = [x, a, b, c] ;
Xs = [a, x, b, c] ;
Xs = [a, b, x, c] ;
Xs = [a, b, c, x] ;
false.

?- select(X,[a,b,c],Xs).
X = a,
Xs = [b, c] ;
X = b,
Xs = [a, c] ;
X = c,
Xs = [a, b] ;
false.
```

Three basic programs IV

```
?- permutation([a,b,c],L).
L = [a, b, c] ;
L = [b, a, c] ;
L = [b, c, a] ;
L = [a, c, b] ;
L = [c, a, b] ;
L = [c, b, a] ;
false.

?- permutation(L,[a,b,c]).
L = [a, b, c] ;
L = [b, a, c] ;
L = [c, a, b] ;
L = [a, c, b] ;
L = [b, c, a] ;
L = [c, b, a] ;
< no termination !!! >
```

Elementary programs

```
suffix(Xs,Xs).
suffix(Xs,[Y|Ys]) :- suffix(Xs,Ys).

prefix([],Xs).
prefix([X|Xs],[X|Ys]) :- prefix(Xs,Ys).

sublist1(Xs,Ys) :- suffix(Xs,Zs), prefix(Zs,Ys).

sublist2(Xs,Ys) :- prefix(Zs,Ys), suffix(Xs,Zs).

subset1([],[]).
subset1([X|Xs],[X|Zs]) :- subset1(Xs,Zs).
subset1(Xs,[X|Zs]) :- subset1(Xs,Zs).

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

subset2([],Xs).
subset2([Y|Ys],Xs):- member(Y,Xs), subset2(Ys,Xs).
```

Computation I

```
?- subset1(Xs,[a,b,c]).
Xs = [a, b, c] ;
Xs = [a, b] ;
Xs = [a, c] ;
Xs = [a] ;
Xs = [b, c] ;
Xs = [b] ;
Xs = [c] ;
Xs = [] ;
No

?- subset1([1,3],[1,2,3,4]).
Yes

?- subset1([1,1,3],[1,2,3,4]).
No

?- subset1([3,1],[1,2,3,4]).
No
```

Computation II

```
?- subset2([1,1,3],[1,2,3,4]).
Yes

?- subset2([3,1],[1,2,3,4]).
Yes

?- subset2(Xs,[1,2,3,4]).
Xs = [] ;
Xs = [1] ;
Xs = [1, 1] ;
Xs = [1, 1, 1] ;
...
```

Computation III

```
subset_list([],[[]]).
subset_list([X|Xs],Zss) :-
      subset_list(Xs,Uss),
      put_in_all(X,Uss,Vss),
      append(Vss,Uss,Zss).

put_in_all(A,[],[]).
put_in_all(A,[Bs|Bss],[[A|Bs]|Css]) :- put_in_all(A,Bss,Css).
```

How does that work?

```
?- put_in_all(x,[[a,b],[],[c,d,e]],Xss).
Xss = [[x, a, b], [x], [x, c, d, e]]

?- subset_list([a,b,c],Xss).
Xss = [[a, b, c], [a, b], [a, c], [a],
       [b, c], [b], [c], []]
```

The French cathedrals I

Five beautiful cathedrals: Amiens, Beauvais, Bourges, Chartres and Laon.
They were built in (start): 1155, 1191, 1196, 1220 and 1225.
Heights (under the nave): 25, 37, 38, 42 and 48 m.
Lengths: 70, 110, 118, 130 and 145 m.

Clue 1: Amiens's cathedral is higher than the 118m-cathedral
but lower than the 110m-cathedral.
Clue 2: Bourges's cathedral was built before Chartres's cathedral.
Clue 3: Bourges's cathedral is higher than the 130m-cathedral
but lower than the 1220 cathedral.
Clue 4: Laon's cathedral is longer than the 1225-cathedral
and older than Bourges's cathedral.
Clue 5: Chartres's cathedral is less than 38m high.

The French cathedrals II

```
go(St) :-  % natural ordering
St = [[amiens,  Yam,Ham,Lam],
      [beauvais,Ybe,Hbe,Lbe],
      [bourges, Ybo,Hbo,Lbo],
      [chartres,Ych,Hch,Lch],
      [laon,    Yla,Hla,Lla]],
    permutation([Yam,Ybe,Ybo,Ych,Yla],[1155,1191,1196,1220,1225]),
    permutation([Ham,Hbe,Hbo,Hch,Hla],[25,37,38,42,48]),
    permutation([Lam,Lbe,Lbo,Lch,Lla],[70,110,118,130,145]),
    member([_,_,Hx,118],St), Ham > Hx,
    member([_,1225,Hy,_],St), Ham < Hy,
    member([_,_,Hz,110],St), Hch > Hz,
    Ybo < Ych,
    member([_,_,Hw,130],St), Hbo > Hw,
    member([_,1220,Hv,_],St), Hbo < Hv,
    member([_,1225,_,Lu],St), Lla > Lu,
    Yla < Ybo,
    Hch < 38.
```

The French cathedrals III

```
go(St) :-  % optimal ordering
St = [[amiens,  Yam,Ham,Lam],
      [beauvais,Ybe,Hbe,Lbe],
      [bourges, Ybo,Hbo,Lbo],
      [chartres,Ych,Hch,Lch],
      [laon,    Yla,Hla,Lla]],
    permutation([Yam,Ybe,Ybo,Ych,Yla],[1155,1191,1196,1220,1225]),
    Ybo < Ych,
    Yla < Ybo,
    permutation([Ham,Hbe,Hbo,Hch,Hla],[25,37,38,42,48]),
    Hch < 38.
    member([_,_,Hx,118],St), Ham > Hx,
    member([_,1225,Hy,_],St), Ham < Hy,
    member([_,_,Hz,110],St), Hch > Hz,
    member([_,_,Hw,130],St), Hbo > Hw,
    member([_,1220,Hv,_],St), Hbo < Hv,
    permutation([Lam,Lbe,Lbo,Lch,Lla],[70,110,118,130,145]),
    member([_,1225,_,Lu],St), Lla > Lu.
```

The French cathedrals IV

```
/*
?- time(go(St)).
% 16,990,074 inferences, 6.200 CPU in 6.206 seconds (100% CPU, 2740335 Lips)
St = [[amiens, 1220, 42, 145],
      [beauvais, 1225, 48, 70],
      [bourges, 1191, 38, 118],
      [chartres, 1196, 37, 130],
      [laon, 1155, 25, 110]].

?- time(go2(St)).
% 32,525 inferences, 0.020 CPU in 0.021 seconds (93% CPU, 1626250 Lips)
St = [[amiens, 1220, 42, 145],
      [beauvais, 1225, 48, 70],
      [bourges, 1191, 38, 118],
      [chartres, 1196, 37, 130],
      [laon, 1155, 25, 110]].
*/
```

Extra-galactic tribes I

Kolokolos always tell the truth; Tarzaneaters always lie.

All answer questions by "whamm" or "galagala".

Some days, "whamm" means "yes" and "galagala" means "no", the other days . . . it is quite the reverse!.

You get lost in the forest and meet a group of natives.

A question-answer session takes place:

Extra-galactic tribes II

1. You ask Grocongo :
   “Is ‘whamm’ affirmative today?”
   Answer: whamm !

2. You ask Minibola :
   “Are you all from the same tribe?”
   Answer: whamm !

3. You ask Biribiri :
   “Dodo, Enigmo, Sitopri and you, are you from the same tribe?”
   Answer: galagala !

4. You ask Limacela :
   “Crocro et Minibola, are they from the same tribe?”
   Answer: galagala !

Extra-galactic tribes III

5. You ask Lasercho :
   "Grocongo et Crocro and you, are you from the same tribe?"
   Answer: whamm !

6. You ask Albarama :
   "Crocro, Limacela and you, are you from the same tribe?"
   Answer: galagala !

7. You ask Dada :
   "Pikaglass and Bonifacio, are they from the same tribe?"
   Answer: galagala !

8. You ask Crocro :
   "You and Minibola, are you from the same tribe?"
   Answer: galagala !

Extra-galactic tribes IV

9. You ask Dodo :
   "You and Biribiri, are you from the same tribe?"
   Answer: whamm !

10. You ask Enigmo :
    "Sitopri, Dada and you, are you from the same tribe?"
    Answer: whamm !

11. You ask Sitopri :
    "Dodo and Enigmo, are they from the same tribe?"
    Answer: whamm !

12. You ask Gagagogo :
    "Enigmo, Dodo and you, are you from the same tribe?"
    Answer: whamm !

Extra-galactic tribes V

The questions :

Could you tell me :

Is Gagagogo a Kolokolo or a Tarzaneater?

Which question do I ask Bonifacio,
in order to know whether I must turn left or right?

Extra-galactic tribes VI

Quite obviously, Prolog will not solve everything.
Prolog will attempt to know whether "Yes" is "whamm" or "galagala",
and to determine the tribe of all natives.

Besides, only clues from 3 to 12 are prone to easy Prolog coding, through predicates **`are_you`** and **`are_they`**.

Clue 1 does not tell whether "whamm" is affirmative or not;
it tell us Grocongo is a (truthful) Kolokolo.

Answer to clue 2 is obviously negative, so either Minibola is a Kolokolo and "whamm" means "no", or Minibola is a Tarzaneater and "whamm" means "yes".

Extra-galactic tribes VII

```
kol([]).
kol([k|Xs]) :- kol(Xs).

tar([]).
tar([t|Xs]) :- tar(Xs).

some_kol([k|_]).
some_kol([t|Xs]) :- some_kol(Xs).

some_tar([t|_]).
some_tar([k|Xs]) :- some_tar(Xs).
```

Extra-galactic tribes VIII

```
% arg 1-: interlocutor is either a Kolokolo or a Tarzaneater (k or t)
% arg 2+: list of named natives
% arg 3-: the word meaning "yes" (w or g)
% arg 4+: the answer (w or g)

are_you(k,L,g,g) :- kol(L).
are_you(k,L,w,g) :- some_tar(L).
are_you(t,L,g,g) :- some_kol(L).
are_you(t,L,w,g) :- tar(L).
are_you(k,L,w,w) :- kol(L).
are_you(k,L,g,w) :- some_tar(L).
are_you(t,L,w,w) :- some_kol(L).
are_you(t,L,g,w) :- tar(L).

are_they(k,L,g,g) :- kol(L); tar(L).
are_they(k,L,w,g) :- some_kol(L), some_tar(L).
are_they(t,L,g,g) :- some_kol(L), some_tar(L).
are_they(t,L,w,g) :- kol(L); tar(L).
are_they(k,L,w,w) :- kol(L); tar(L).
are_they(k,L,g,w) :- some_kol(L), some_tar(L).
are_they(t,L,w,w) :- some_kol(L), some_tar(L).
are_they(t,L,g,w) :- kol(L); tar(L).
```

Extra-galactic tribes IX

```
go(St) :-
  St = [Yes,Gro,Min,Bir,Lim,Las,Dad,Cro,Dod,Eni,Sit,Gag,Alb,Pik,Bon],
  Gro = k,
  (Min=k,Yes=g; Min=t,Yes=w),
  are_you(Bir,[Dod,Eni,Sit],Yes,g),
  are_they(Lim,[Cro,Min],Yes,g),
  are_you(Las,[Gro,Cro],Yes,w),
  are_you(Alb,[Cro,Lim],Yes,g),
  are_they(Dad,[Pik,Bon],Yes,g),
  are_you(Cro,[Min],Yes,g),
  are_you(Dod,[Bir],Yes,w),
  are_you(Eni,[Sit,Dad],Yes,w),
  are_they(Sit,[Dod,Eni],Yes,w),
  are_you(Gag,[Eni,Dod],Yes,w),
  nl, write(St), fail.
```

Extra-galactic tribes X

```
?- [tribus].
% tribus compiled 0.01 sec, 1,212 bytes
true.

?- go(St).

[w, k, t, k, t, t, k, t, k, t, t, t, k, k, t]
[w, k, t, k, t, t, k, t, k, t, t, t, k, t, k]
[w, k, t, k, t, t, k, t, k, t, t, t, t, k, t]
[w, k, t, k, t, t, k, t, k, t, t, t, t, t, k]
false.
```

... and now we can devise the question!
Hint: we do know Bonifacio and Pikaglass belong to distinct tribes.

Input resolution

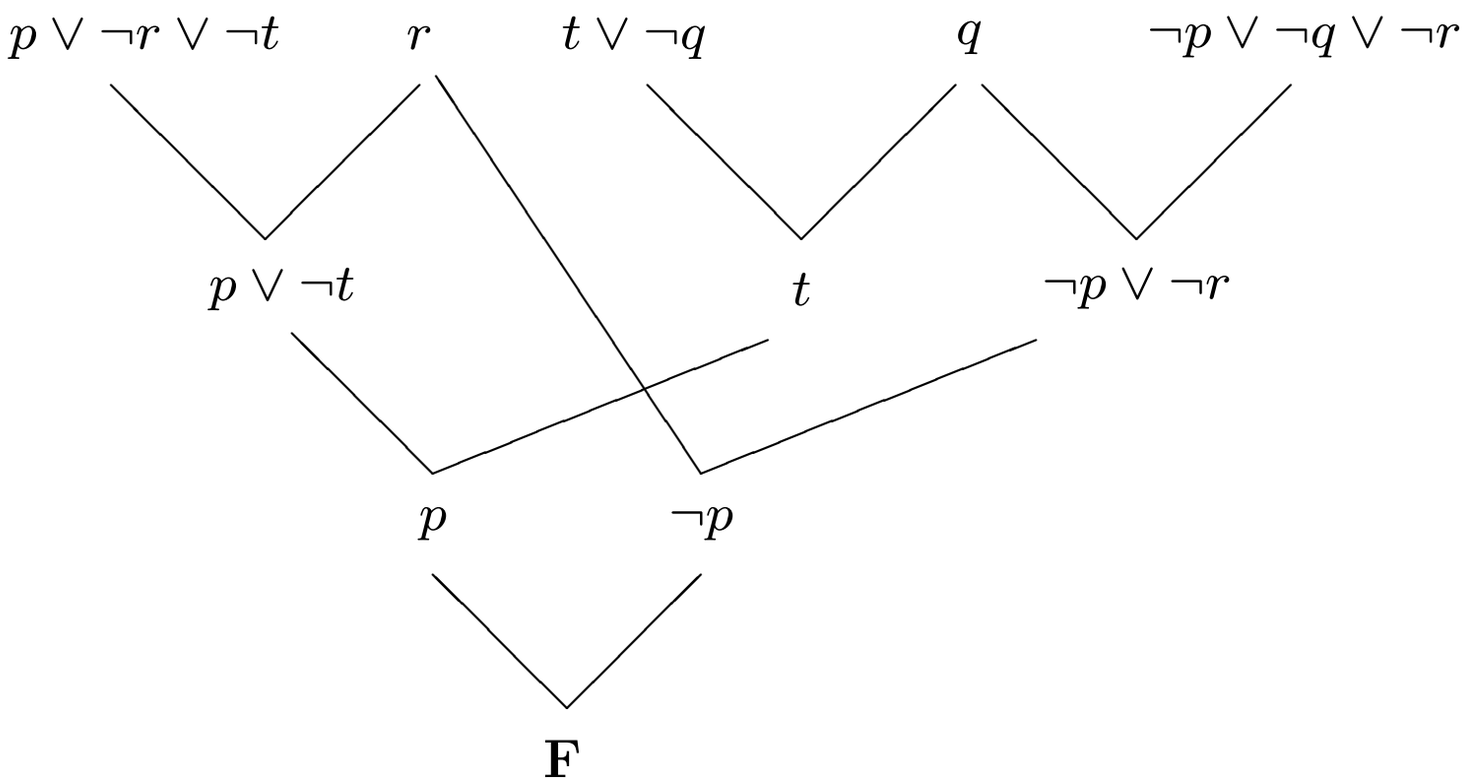
In unit resolution there is no special status for the goal, the negative clause (negation of the query).

Input resolution is a suitable improvement.

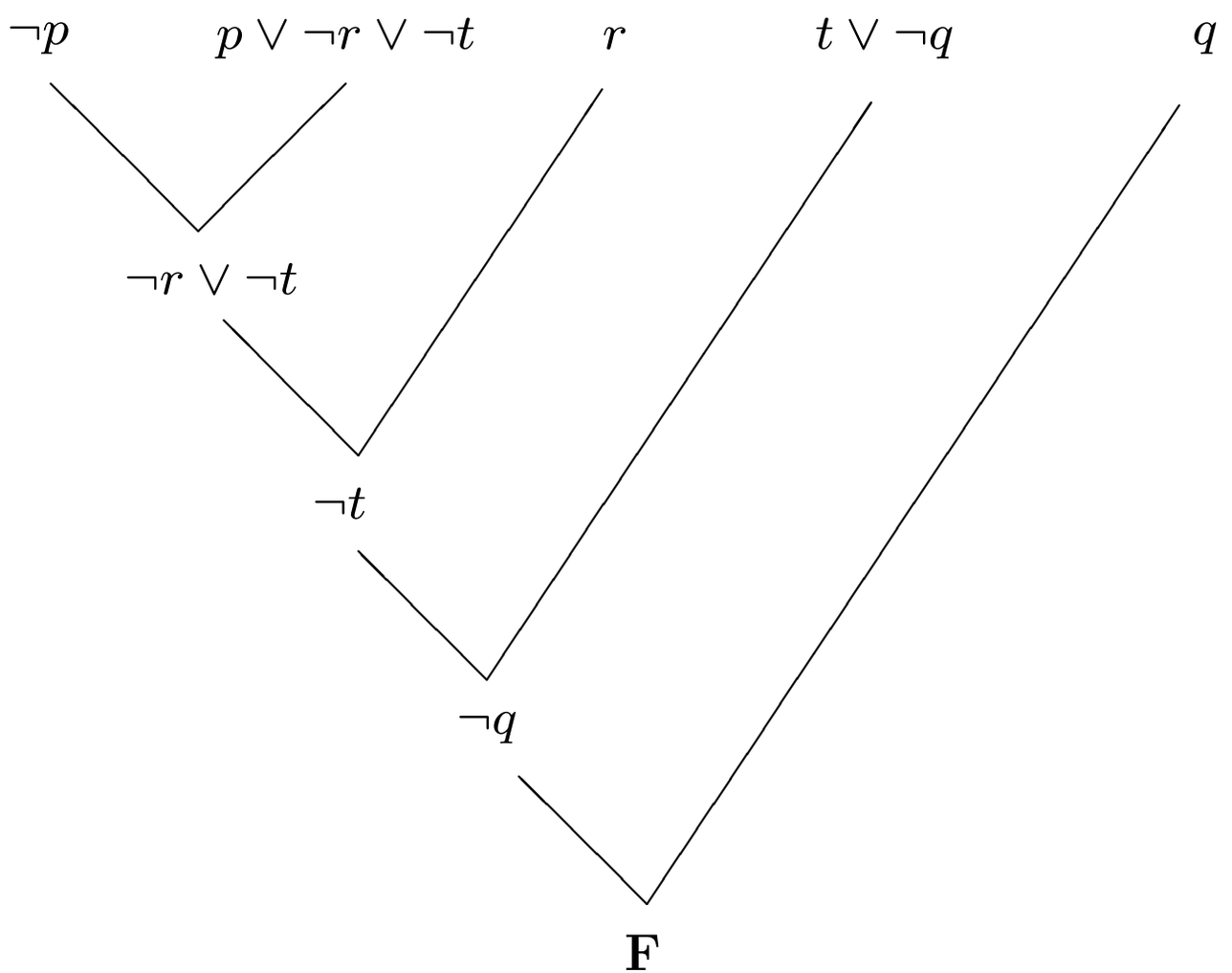
It is based on a variable G, the "current goal".

$\{G = G_0\}$
while $G \neq \mathbf{F}$ do
select p and c such that
$\neg p \in G$,
$c \in L$ and $p \in c$;
$G := (G \setminus \{\neg p\}) \cup (c \setminus \{p\})$.

Example of unit resolution



Example of input resolution



UR and IR are "equivalent"

Theorem. Minimal Horn H can be UR-refuted if and only if it can be IR-refuted; the basis of the IR-refutation is any negative clause in H.

Constructive proof.

There are two (recursive) algorithms to convert an UR-refutation of H into an IR-refutation, and conversely.
Induction on the atom set.

Base case: $\Pi = \{p\}$.
Induction case: reduction of Π to $\Pi \setminus \{q\}$, with $q \in \Pi$.

Base case is obvious:
$\{p, \neg p\}$ allows a single refutation,
which is both UR and IR (based on $\neg p$).

From UR-refutation to IR-refutation

Let $\mathcal{R}$ be a UR-refutation and Π the atom set.

An arbitrary unit clause $q \in H$ is selected.

The first step consists in removing clause q, its offspring and all occurrences of literal $\neg q$ from $\mathcal{R}$. The resulting tree $\mathcal{R}'$ is an UR-refutation of $H' = \{c \backslash \{\neg q\} : c \in H \backslash \{q\}\}$.

The second step is to obtain recursively an IR-refutation $\mathcal{S}$, based on some negative clause of H' and equivalent to the UR-refutation $\mathcal{R}'$.

The third step is to expand the IR-refutation $\mathcal{S}$ by reintroducing clause q as an additional leaf, and also the literal $\neg q$ in some other leaves, in order to get back the leaf set of $\mathcal{R}$; this results in a tree $\mathcal{S}'$.

The fourth step consists in propagating downward the literal $\neg q$ where it is needed in order to get back a correct IR-deduction $\mathcal{S}''$. Most likely it is no longer an IR-refutation since it will produce the clause $\neg q$ instead of the empty clause. In this case, a single branch from leaf $\neg q$ is used to complete the deduction and obtain the empty clause.

From IR-refutation to UR-refutation

The same technique applies,

try it!

Beware: the devil is in the detail

A naive sort

```
naive_sort(Xs,Ys) :- permut(Xs,Ys), ordered(Ys).

permut(Xs,[Z|Zs]) :- select(Z,Xs,Ys), permut(Ys,Zs).
permut([],[]).

select(X,[X|Xs],Xs).
select(X,[Y|Ys],[Y|Zs]) :- select(X,Ys,Zs).

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

slow, not reversible

Insert sort

```
insert_sort([],[]).
insert_sort([X|Xs],Ys) :- insert_sort(Xs,Zs), insert(X,Zs,Ys).

insert(X,[],[X]).
insert(X,[Y|Ys],[Y|Zs]) :- X>Y, insert(X,Ys,Zs).
insert(X,[Y|Ys],[X|[Y|Ys]]) :- X=<Y.
```

Quadratic, not reversible

Quick sort

```
quick_sort([],[]).
quick_sort([X|Xs],Ys) :- part(Xs,X,Littles,Bigs),
                         quick_sort(Littles,Ls),
                         quick_sort(Bigs,Bs),
                         append(Ls,[X|Bs],Ys).

part([],Y,[],[]).
part([X|Xs],Y,[X|Ls],Bs) :- X=<Y, part(Xs,Y,Ls,Bs).
part([X|Xs],Y,Ls,[X|Bs]) :- X>Y, part(Xs,Y,Ls,Bs).
```

Fast, nearly reversible

```
quick_sort_bis([],[]).
quick_sort_bis([X|Xs],Ys) :- append(Ls,[X|Bs],Ys),
                             quick_sort_bis(Littles,Ls),
                             quick_sort_bis(Bigs,Bs),
                             part(Xs,X,Littles,Bigs).
```

Merge sort

```
merge_([],Xs,Xs).
merge_([X|Xs],[],[X|Xs]).
merge_([X|Xs],[Y|Ys],[X|Zs]) :- X<Y, merge_(Xs,[Y|Ys],Zs).
merge_([X|Xs],[Y|Ys],[X|[Y|Zs]]) :- X=Y, merge_(Xs,Ys,Zs).
merge_([X|Xs],[Y|Ys],[Y|Zs]) :- X>Y, merge_([X|Xs],Ys,Zs).

merge_sort([],[]).
merge_sort([X],[X]).
merge_sort([X|[X1|Xs]],Ys) :- division(Xs,Fs,Ss),
                              merge_sort([X|Fs],FFs),
                              merge_sort([X1|Ss],SSs),
                              merge_(FFs,SSs,Ys).

division([],[],[]).
division([X],[X],[]).
division([X|[X1|Xs]],[X|Ys],[X1|Zs]) :- division(Xs,Ys,Zs).
```

Functional programming vs. logic programming I

Usually the functional approach is the best choice, but . . .

A partition of a set S is a family of S-subsets ("parts") such that
1. Parts are not empty;
2. The intersection of two (distinct) parts is empty;
3. The union of all parts is S.

Partitions can be generated in a recursive way.

The only partition of the empty set is the empty set. (Why?)

If $x \notin S$ any partition P of $\{x\} \cup S$ is obtained from some partition Q of S;
P can be $\{\{x\}\} \cup Q$ (a new part has been created for x) or
P can be $(Q \backslash \{R\}) \cup \{\{x\} \cup R\}$ (x has been inserted in some existing part R).

Functional programming vs. logic programming II

Scheme

```
(define partitions
  (lambda (e)
    (if (null? e)
        '(())
        (let ((rec (partitions (cdr e))))
          (append (way_1 (car e) rec) (way_2 (car e) rec))))))

(define way_1 (lambda (x lp) (insert_in_all (list x) lp)))

(define insert_in_all (lambda (x le) (map (lambda (e) (cons x e)) le)))

(define way_2 (lambda (x lp) (map_append (lambda (p) (split x p)) lp)))

(define map_append
  (lambda (f l)
    (if (null? l) '() (append (f (car l)) (map_append f (cdr l))))))

(define split
  (lambda (x ll)
    (if (null? ll)
        '()
        (cons (cons (cons x (car ll)) (cdr ll))
              (map (lambda (ss) (cons (car ll) ss)) (split x (cdr ll)))))))
```

Functional programming vs. logic programming III

Prolog (naive program translation)

```
c_partitions([],[[]]).
c_partitions([X|Xs],Ps) :- c_partitions(Xs,Qs),
                           way_1(X,Qs,P1s), way_2(X,Qs,P2s),
                           append(P1s,P2s,Ps).

way_1(X,Qs,Rs) :- insert_in_all([X],Qs,Rs).

insert_in_all(U,[],[]).
insert_in_all(U,[Us|Uss],[[U|Us]|Vss]) :- insert_in_all(U,Uss,Vss).

way_2(X,[],[]).
way_2(X,[Q|Qs],Ps) :- split(X,Q,R1s), way_2(X,Qs,R2s), append(R1s,R2s,Ps).

split(X,[],[]).
split(X,[Xs|Xss],[[[X|Xs]|Xss]|Ysss]) :- split(X,Xss,Xsss),
                                          insert_in_all(Xs,Xsss,Ysss).
```

Functional programming vs. logic programming IV

```
?- insert_in_all(x,[[1],[2,3],[4]],LL).
LL = [[x, 1], [x, 2, 3], [x, 4]] ;
false.

?- way_1(x,[[[1],[2,3]],[[2],[1,3]]],LP).
LP = [[[x], [1], [2, 3]], [[x], [2], [1, 3]]] ;
false.

?- split(x,[[1],[2,3],[4]],LL).
LL = [[[x, 1], [2, 3], [4]], [[1], [x, 2, 3], [4]], [[1], [2, 3], [x, 4]]] ;
false.

?- way_2(x,[[[1],[2,3]],[[2],[1,3]]],LP).
LP = [[[x, 1], [2, 3]], [[1], [x, 2, 3]], [[x, 2], [1, 3]], [[2], [x, 1, 3]]] ;
false.
```

Functional programming vs. logic programming V

Prolog (direct translation of the basic idea)

```
partition([],[]).
partition([X|Xs],[[X]|P]) :- partition(Xs,P).
partition([X|Xs],Q) :- partition(Xs,P),
                       append(R1,[R|R2],P), append(R1,[[X|R]|R2],Q).

partitions(Xs,Ps) :- findall(P,partition(Xs,P),Ps).

partitions_bis(Xs) :- partition(Xs,P), write(P), nl, fail.
```

Functional programming vs. logic programming VI

```
?- partitions([1,2,3],Ps).

Ps = [[[1],[2],[3]],[[1],[2,3]],
      [[1,2],[3]],[[2],[1,3]],[[1,2,3]]]

partitions_bis([1,2,3],Ps).
[[1],[2],[3]]
[[1],[2,3]]
[[1,2],[3]]
[[2],[1,3]]
[[1,2,3]]

No
```

Functional programming vs. logic programming VII

```
bipart([X],[Y|Xs],[X,Y|Xs]) :- X<Y.
bipart([X|Xs],[Y|Ys],[X|Zs]) :- bipart(Xs,[Y|Ys],Zs),X<Y.
bipart([X|Xs],[Y|Ys],[Y|Zs]) :- bipart([X|Xs],Ys,Zs),X>Y.

part2([X|Xs],[Y|Ys],Zs) :- bipart([X|Xs],[Y|Ys],Zs), X<Y.
part2([X|Xs],[Y|Ys],Zs) :- bipart([Y|Ys],[X|Xs],Zs), X<Y.

part(Xs,P) :- part1(Xs,P).
part(Xs,P) :- part+(Xs,P).

part1([X|Xs],[[X|Xs]]).

part+(Xs,[Ys,Zs]) :- part2(Ys,Zs,Xs).
part+([X,Y|Zs],[[A|As]|Ass]) :-
   part2([A|As],[U|Us],[X,Y|Zs]), part+([U|Us],Ass).

partitions_ter(Xs,Ps) :- findall(P,part(Xs,P),Ps).
```

Functional programming vs. logic programming VIII

```
?- bipart(U,V,[1,2,3]).
U = [1], V = [2, 3] ;
U = [1, 2], V = [3] ;
U = [2], V = [1, 3] ;
false.

?- bipart(U,V,[1,2,3,4]).
U = [1], V = [2, 3, 4] ;
U = [1, 2], V = [3, 4] ;
U = [1, 2, 3], V = [4] ;
U = [1, 3], V = [2, 4] ;
U = [2], V = [1, 3, 4] ;
U = [2, 3], V = [1, 4] ;
U = [3], V = [1, 2, 4] ;
false.
```

Functional programming vs. logic programming IX

```
?- part2(U,V,[1,2,3]).
U = [1], V = [2, 3] ;
U = [1, 2], V = [3] ;
U = [1, 3], V = [2] ;
false.

?- part2(U,V,[1,2,3,4]).
U = [1], V = [2, 3, 4] ;
U = [1, 2], V = [3, 4] ;
U = [1, 2, 3], V = [4] ;
U = [1, 3], V = [2, 4] ;
U = [1, 3, 4], V = [2] ;
U = [1, 4], V = [2, 3] ;
U = [1, 2, 4], V = [3] ;
false.
```

Functional programming vs. logic programming X

```
?- partitions_ter([1,2,3],Ps).

Ps = [[[1,2,3]],[[1],[2,3]],[[1,2],[3]],[[1,3],[2]],[[1],[2],[3]]]

?- partitions_ter([1,2,3,4],Ps).

Ps = [[[1,2,3,4]],[[1],[2,3,4]],[[1,2],[3,4]],
      [[1,2,3],[4]],[[1,3],[2,4]],[[1,3,4],[2]],
      [[1,4],[2,3]],[[1,2,4],[3]],[[1],[2],[3,4]],
      [[1],[2,3],[4]],[[1],[2,4],[3]],
      [[1],[2],[3],[4]],[[1,2],[3],[4]],
      [[1,3],[2],[4]],[[1,4],[2],[3]]]
```

Homework

Bob and Ann invited their friends Diana and Cindy over for a cookout. On the menu were grilled marinated beef cubes and four kinds of vegetables -- mushrooms, onions, peppers, and tomatoes -- which were put onto skewers. The first skewer that each person made had three beef cubes and one piece of three kinds of vegetables -- each person disliked a different vegetable and omitted it from his or her skewer. The six pieces can be numbered 1 to 6 from the handle to the point of the skewer. Can you tell what item each person had in each position?

1. No kebab had two beef cubes right next to each other.
2. No one's beef cubes were in the same three positions as anyone else's.
3. One shish kebab's first three items (numbers 1, 2, and 3 respectively) were beef, pepper, and mushroom; this wasn't Diana's.
4. One skewer had beef cubes in positions 1, 3, and 5, and a tomato wedge in position 6.
5. Bob, who loves onions and included a chunk on his skewer, had vegetables in both positions 4 and 5.
6. On the four kebabs, the items in position 5 were beef, mushroom, onion and tomato.
7. Each onion chunk was immediately between two beef cubes.
8. No pepper was immediately between two beef cubes.
9. Cindy can't stand mushrooms and left them off her skewer.
10 At least two kebabs had the same vegetable in the same position at least once.

Determine: Person - 1 - 2 - 3 - 4 - 5 - 6 - Veggie skipped

Two (correct) versions are requested.
First, an elementary program, easy to write and to read;
second, an efficient program, inducing a fast computation.

A finite automaton

```
/* Automaton, language (ab)* */
init(q0).
fin(q0).
delta(q0,a,q1).
delta(q1,b,q0).

/* Automaton, acceptance algorithm */
acc(Xs) :- init(Q), acc(Q,Xs).
acc(Q,[]) :- fin(Q).
acc(Q,[X|Xs]) :- delta(Q,X,Q1), acc(Q1,Xs).
```

A pushdown automaton

```
/* Pushdown automaton
   for the {a,b}-palindrome language */
init(q0).
fin(q1).
delta(q0,X,S,q0,[X|S]).
delta(q0,X,S,q1,[X|S]).
delta(q0,X,S,q1,S).
delta(q1,X,[X|S],q1,S).

/* Acceptance algorithm */
acc(Xs) :- init(Q), acc(Q,Xs,[]).
acc(Q,[],[]) :- fin(Q).
acc(Q,[X|Xs],S) :- delta(Q,X,S,Q1,S1),
                   acc(Q1,Xs,S1).
```

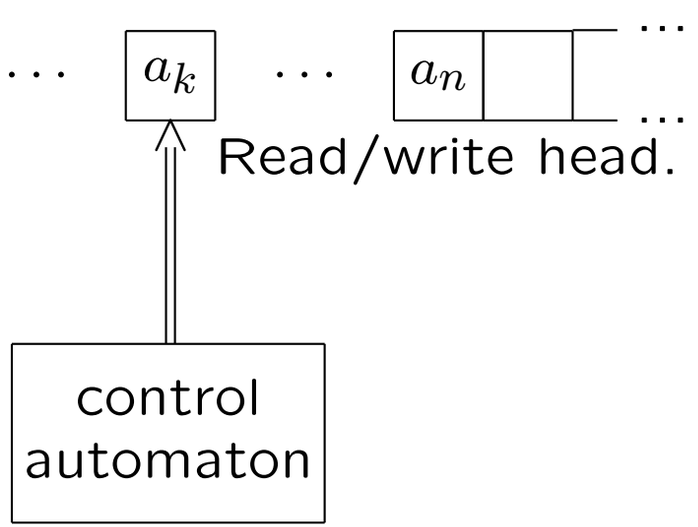
The Turing machine I

For language $\{a^n b^n : n \in \mathbf{N}\}$, only one stack is needed.

For language $\{a^n b^n c^n : n \in \mathbf{N}\}$, two stacks are needed.

Other possibility: Turing machine.

Tape : $a_i \in$ *alphabet*.



The Turing machine II

```
/* Turing machine, Prolog code */

init(q0).                  final(q4).

delta(q0,a,q1,x,right).  delta(q0,y,q3,y,right).
delta(q1,a,q1,a,right).  delta(q1,b,q2,y,left).
delta(q1,y,q1,y,right).  delta(q2,a,q2,a,left).
delta(q2,x,q0,x,right).  delta(q2,y,q2,y,left).
delta(q3,y,q3,y,right).  delta(q3,w,q4,w,right).
```

Definition.

`cfg(Q,Ds,Head,Fs)` corresponds to some configuration: control state is `Q`; tape word is the concatenation of the reverse of `Ds`, the symbol `Head` and the word `Fs`; the symbol `Head` is under the head.

The Turing machine III

```
/* Prolog code, Turing machine */

acc_MT([]) :- init(Q), acc(cfg(Q,[],w,[])).
acc_MT([A|As]) :- init(Q), acc(cfg(Q,[],A,As)).

acc(cfg(Q,Ds,Head,Fs)) :- final(Q).
acc(cfg(Q,Ds,Head,Fs)) :- next_cfg(cfg(Q,Ds,Head,Fs), cfg(Qn,Dsn,Headn,Fsn)),
                          acc(cfg(Qn,Dsn,Headn,Fsn)).

next_cfg(cfg(Q,Ds,Head,Fs), cfg(Qn,Dsn,Headn,Fsn)) :-
   delta(Q,Head,Qn,Symb,Act),
   modif(cfg(Q,Ds,Head,Fs), Symb, Act, cfg(Qn,Dsn,Headn,Fsn)).

modif(cfg(Q,Ds,Head,[]), Symb, right, cfg(Qn,[Symb|Ds],w,[])).

modif(cfg(Q,Ds,Head,[F|Fs]), Symb, right, cfg(Qn,[Symb|Ds],F,Fs)).

modif(cfg(Q,[D|Ds],Head,Fs), Symb, left, cfg(Qn,Ds,D,[Symb|Fs])).
```

Recursive functions I

Primitives

$$\begin{array}{ll} (x_1, \ldots, x_n) \mapsto 0 & : n\text{-ary zero function;} \\ x \mapsto x + 1 & : \text{successor function;} \\ (x_1, \ldots, x_n) \mapsto x_i & : n\text{-ary } i\text{th projection.} \end{array}$$

Composition

$$f(x_1, \ldots, x_n) =_{def} h(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n)) .$$

Primitive recursion

$$f(x_1, \ldots, x_n, 0) = h(x_1, \ldots, x_n) ,$$

$$f(x_1, \ldots, x_n, y + 1) = g(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, y)) .$$

Minimisation

$$f(x_1, \ldots, x_n) = \mu y.[g(x_1, \ldots, x_n, y) = 0] .$$

Recursive functions II

Primitive functions

```
zero_(X1,...,Xn,o).
pr_j(X1,...,Xn,Xj).
succ_(X,s(X)).
```

Composition

```
f_(X1,...,Xn,Y) :- g_1(X1,...,Xn,Y1), ..., g_m(X1,...,Xn,Ym),
                   h_(Y1,...,Ym,Y).
```

Primitive recursion

```
f_(X1,...,Xn,o,Z) :- h_(X1,...,Xn,Z).
f_(X1,...,Xn,s(Y),Z) :- f_(X1,...,Xn,Y,U), g_(X1,...,Xn,Y,U,Z).
```

Recursive functions III

Minimisation

```
f_(X1,...,Xn,Y) :- g_(X1,...,Xn,o,U), r_(X1,...,Xn,o,U,Y).
r_(X1,...,Xn,Y,o,Y).
r_(X1,...,Xn,Y,s(V),Z) :- g_(X1,...,Xn,s(Y),U), r_(X1,...,Xn,s(Y),U,Z).
```

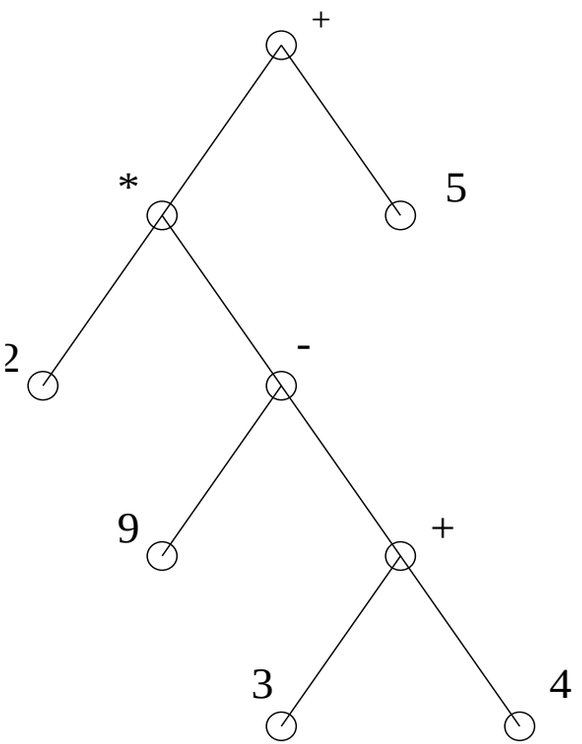
(n+3)-ary predicate `r_` is associated with a (n+2)-ary auxiliary function r; this function is specified:

If $g(x_1,\ldots,x_n,\xi) > 0$ *for all* $\xi = 0,1,\ldots,y-1$
and if $g(x_1,\ldots,x_n,y) = u$,
then $r(x_1,\ldots,x_n,y,u) = \inf\{\zeta : \zeta \geq y \wedge g(x_1,\ldots,x_n,\zeta) = 0\}$.

$r(x_1,\ldots,x_n,y,u)$ is undefined if $u > 0$ and $g(x_1,\ldots,x_n,\zeta) > 0$ for each $\zeta > y$. In these case the query `r_(X1,...,Xn,Y,U,Z)` does not terminate.

Symbolic computation I

$$((2 * (9 - (3 + 4))) + 5)$$



Symbolic computation II

The term-handling predicates `functor`, `args` and `=..` can be used, but the usual list notation is used here.

An atomic **N**-tree is a number.

A compound **N**-tree is a three element-list `[X,Y,Z]` where `X` is an arithmetic operator and where `Y` and `Z` are the **N**-trees corresponding to the operands.

The **N**-tree

```
[+,[*,2,[-,9,[+,3,4]]],5]
```

corresponds to an expression; its list-like representation is:

```
[(,(,2,*,(,9,-,(,3,+,4,),),),+,5,)]
```

Symbolic computation III

With prefix and postfix notation, there is no need for parentheses.

```
[(,(,2,*,(,9,-,(,3,+,4,),),),+,5,)]
```

is translated into

```
[+,*,2,-,9,+,3,4,5]
```

or into

```
[2,9,3,4,+,-,*,5,+]
```

Basic problem: how to convert an **N**-tree into its list-like prefix, postfix, infix representation, and conversely?

Symbolic computation IV

N-tree analysis, naïve solution

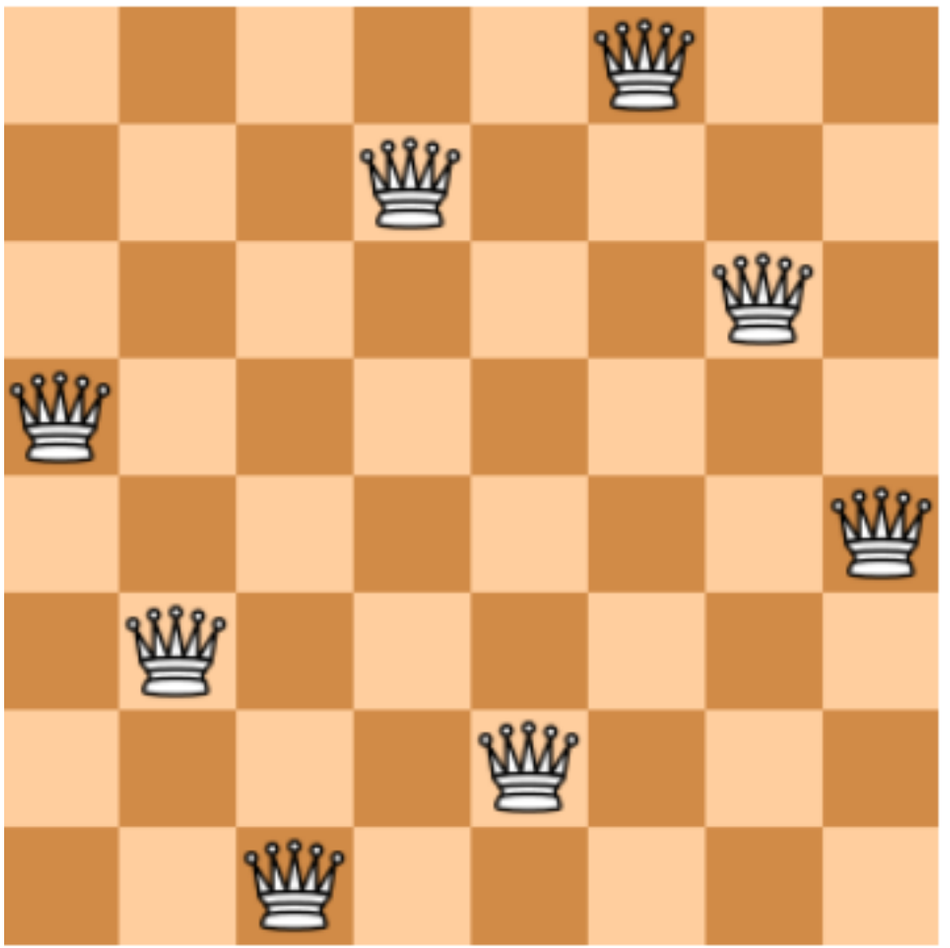
```
tree_to_pref(T,[T]) :- number(T).
tree_to_pref([X,Y,Z],[X|Xs]) :-
    tree_to_pref(Y,Ys),
    tree_to_pref(Z,Zs),
    append(Ys,Zs,Xs).

tree_to_postf(T,[T]) :- number(T).
tree_to_postf([X,Y,Z],Us) :-
    tree_to_postf(Y,Ys),
    tree_to_postf(Z,Zs),
    append(Ys,Zs,Xs),
    append(Xs,[X],Us).
```

Symbolic computation V

```
tree_to_inf(T,[T]) :- number(T).
tree_to_inf([X,Y,Z],[l|Xs]) :-
    tree_to_inf(Y,Ys),
    tree_to_inf(Z,Zs),
    append([X|Zs],[r],Us),
    append(Ys,Us,Xs).
```

`l`: left parenthesis; `r`: right parenthesis.

Eight queens puzzle I



Eight queens puzzle II

```
(build-solution '(6 1 4 8))
            (6 1 4 8)
          (2 6 1 4 8)
        (7 2 6 1 4 8)
        (5 2 6 1 4 8)
      (7 5 2 6 1 4 8)
      (7 5 2 6 1 4 8)
        (5 2 6 1 4 8)
          (2 6 1 4 8)
            (6 1 4 8)
            (3 1 4 8)
          (6 3 1 4 8)
        (2 6 3 1 4 8)
      (7 2 6 3 1 4 8)
    (5 7 2 6 3 1 4 8)
```

Eight queens puzzle III

```
(define legal?
  (lambda (try conf)
    (letrec
      ((good?
         (lambda (new-pl up down)
           (or (null? new-pl)
               (let ((next-pos (car new-pl)))
                 (and (not (= next-pos try))
                      (not (= next-pos up))
                      (not (= next-pos down))
                      (good? (cdr new-pl) (add1 up) (sub1 down))))))))
      (good? conf (add1 try) (sub1 try)))))
```

Eight queens puzzle IV

```
(define build-solution
  (lambda (conf)
    (if (solution? conf)
        conf
        (forward fresh-try conf))))

(define forward
  (lambda (try conf)
    (cond ((zero? try) (backtrack conf))
          ((legal? try conf) (build-solution (cons try conf)))
          (else (forward (sub1 try) conf)))))

(define backtrack
  (lambda (conf)
    (if (null? conf)
        '()
        (forward (sub1 (car conf)) (cdr conf)))))
```

Eight queens puzzle V

```
(build-solution '()) :-) (5 7 2 6 3 1 4 8) ;; solution 01

(backtrack '(5 7 2 6 3 1 4 8)) :-) (4 7 5 2 6 1 3 8) ;; solution 02

(backtrack '(4 7 5 2 6 1 3 8)) :-) (6 4 7 1 3 5 2 8) ;; solution 03

(backtrack '(6 4 7 1 3 5 2 8)) :-) (6 3 5 7 1 4 2 8) ;; solution 04

... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ...

(backtrack '(3 5 2 8 6 4 7 1)) :-) (5 2 4 7 3 8 6 1) ;; solution 91

(backtrack '(5 2 4 7 3 8 6 1)) :-) (4 2 7 3 6 8 5 1) ;; solution 92

(backtrack '(4 2 7 3 6 8 5 1)) :-) ()              ;; plus de solution
```

Eight queens puzzle VI

```
permut([],[]).
permut(Xs,[Z|Zs]) :- select(Z,Xs,Ys), permut(Ys,Zs).

range(N,N,[N]).
range(M,N,[M|Ns]) :- M<N, M1 is M+1, range(M1,N,Ns).

attack(X,Xs) :- attack(X,1,Xs).

attack(X,N,[Y|Ys]) :- X is Y+N.
attack(X,N,[Y|Ys]) :- X is Y-N.
attack(X,N,[Y|Ys]) :- N1 is N+1, attack(X,N1,Ys).

safe([]).
safe([Q|Qs]) :- safe(Qs), not(attack(Q,Qs)).

queens(N,Qs) :- range(1,N,Ns), permut(Ns,Qs), safe(Qs).
```

Eight queens puzzle VII

```
range(N,N,[N]).
range(M,N,[M|Ns]) :- M<N, M1 is M+1, range(M1,N,Ns).

attack(X,Xs) :- attack(X,1,Xs).

attack(X,N,[Y|Ys]) :- X is Y+N.
attack(X,N,[Y|Ys]) :- X is Y-N.
attack(X,N,[Y|Ys]) :- N1 is N+1, attack(X,N1,Ys).

queens(N,Qs) :- range(1,N,Ns), queens(Ns,[],Qs).

queens([],Qs,Qs).
queens(UQs,SQs,Qs) :- select(Q,UQs,UQs1),
                      not(attack(Q,SQs)),
                      queens(UQs1,[Q|SQs],Qs).
```

Negation in logic programming I

In order to define negation in logic programming,
we should investigate truth in logic programming.

Without negation, there is no "excluded middle":
$L \models p$ is true or false,
$L \models \neg p$ is always false!

$\overline{L}$: if $L = \{\ldots, (q \wedge r) \Rightarrow p,\ s \Rightarrow p, \ldots\}$
then $\overline{L} = \{\ldots, [(q \wedge r) \vee s] \equiv p, \ldots\}$

Negation in logic programming II

Four possibilities!

1. $L \vdash \neg p$ iff $L \models \neg p$.

2. $L \vdash \neg p$ iff "?- p" fails.

3. $L \vdash \neg p$ iff $\overline{L} \models \neg p$.

4. $L \vdash \neg p$ iff $L \not\models p$.

Negation in logic programming III

1. useless

2. easy but restrictive

3. possible but too slow

4. CWA: possible only in Datalog

Remember: implicit quantification

Negation in logic programming IV

What about operational semantics?

`not(p)` succeeds if `p` root of a finite failed tree.

`not(p)` fails if `p` succeeds.

When `not(p)` is "evaluated", it must be fully instantiated!

Cut I: Principle

- A logic program is highly non-deterministic, but its prolog counterpart is not: the depthfirst, left-to-right strategy is fixed for the exploration of the search tree. Sometimes we would like to prune the search tree, to cut unwanted or useless branches of the tree. This means, some clauses are to be "triggered" in a restrictive way only.

- The problem is, how to specify in the *program* (which is independent on any specific goal) which branches of the *search tree* (which depends on a specific goal) should be pruned? The link between the program and the search tree is not trivial.

- As far as the search tree is concerned, any branch which does not contain any success-leaf can be pruned, but how can we identify them? Besides, sometimes branches containing "bad" success-leaves could or should be pruned too.

- As far as the program is concerned, it must be determined whether some clause can be left untriggered when some condition is satisfied.

- If we do this, the logic, declarative semantics of programs is no longer possible, but the operational semantics can still be used.

Cut II: General rule

Let P be a program, G the goal (labelling the root of the search tree) and $G_1, \ldots, G_r$ a subgoal (labelling an internal node of the search tree). The branches at this node correspond to the clauses satisfying the unification condition: variable instances exist such that the head of a clause matches the first literal $G1$ of the subgoal.

Cut III: How to prevent clause use?

One could specify that, if c_1 is selected to reduce G, then subsequent clauses $c_2, c_3, \ldots$ cannot be selected as alternate ways to solve G_1.

That is a rather strong restriction: if the body of c_1 cannot be reduced, G_1 will fail, even if c_2 for instance would have succeeded in this task.

Besides, clauses $c_2, c_3, \ldots$ become useless except if some goals match their heads and do not match the head of c_1.

A more prudent, moderate restriction, would be to specify that, as soon as some specified part of the body of c_1 has been reduced, then $c_2, c_3, \ldots$ are no longer possible alternates to solve G_1.

Cut IV: Definition

Let c be the clause A :- $B_1, \ldots, B_k, !, B_{k+1}, \ldots, B_n$.
Special atom noted "!" is a *cut*.
Let $G = G_1, \ldots, G_r$ the current subgoal, just before c is used.
The next subgoal is $B_1\theta, \ldots, B_k\theta, !, B_{k+1}\theta, \ldots, B_n\theta, G_2\theta, \ldots, G_r\theta$ where θ is the "most general unifier" matching G_1 and A.
If the reduction of $B_1, \ldots, B_k$ fails, the cut is not triggered and has no effect. Otherwise, the new current subgoal is
$!, B_{k+1}\lambda, \ldots, B_n\lambda, G_2\lambda, \ldots, G_r\lambda$ where λ is less general than θ.
The rôle of the cut is to prevent alternate reductions of G_1.
Otherwise stated, we are committed to the choices (substitutions) made during the reduction of $B_1, \ldots, B_k$. The reductions of $B_{k+1}\lambda, \ldots, B_n\lambda$ will determine the reductions of G_1.

Cut V: Effect on the search tree

The search tree is incrementally built as usual until the subgoal $B_{k+1}\lambda, \ldots, B_n\lambda, G_2\lambda, \ldots, G_r\lambda$ fails.
Then, any branch which could be grown on the right of the branch starting from the father node $G = G_1, \ldots, G_r$ of the goal $B_1\theta, \ldots, B_k\theta, !, B_{k+1}\theta, \ldots, B_n\theta, G_2\theta, \ldots, G_r\theta$ to the goal $!, B_{k+1}\lambda, \ldots, B_n\lambda, G_2\lambda, \ldots, G_r\lambda$ are (virtually) pruned (they would correspond to alternate reductions of the subgoal $B_1\theta, \ldots, B_k\theta$ or of G itself.

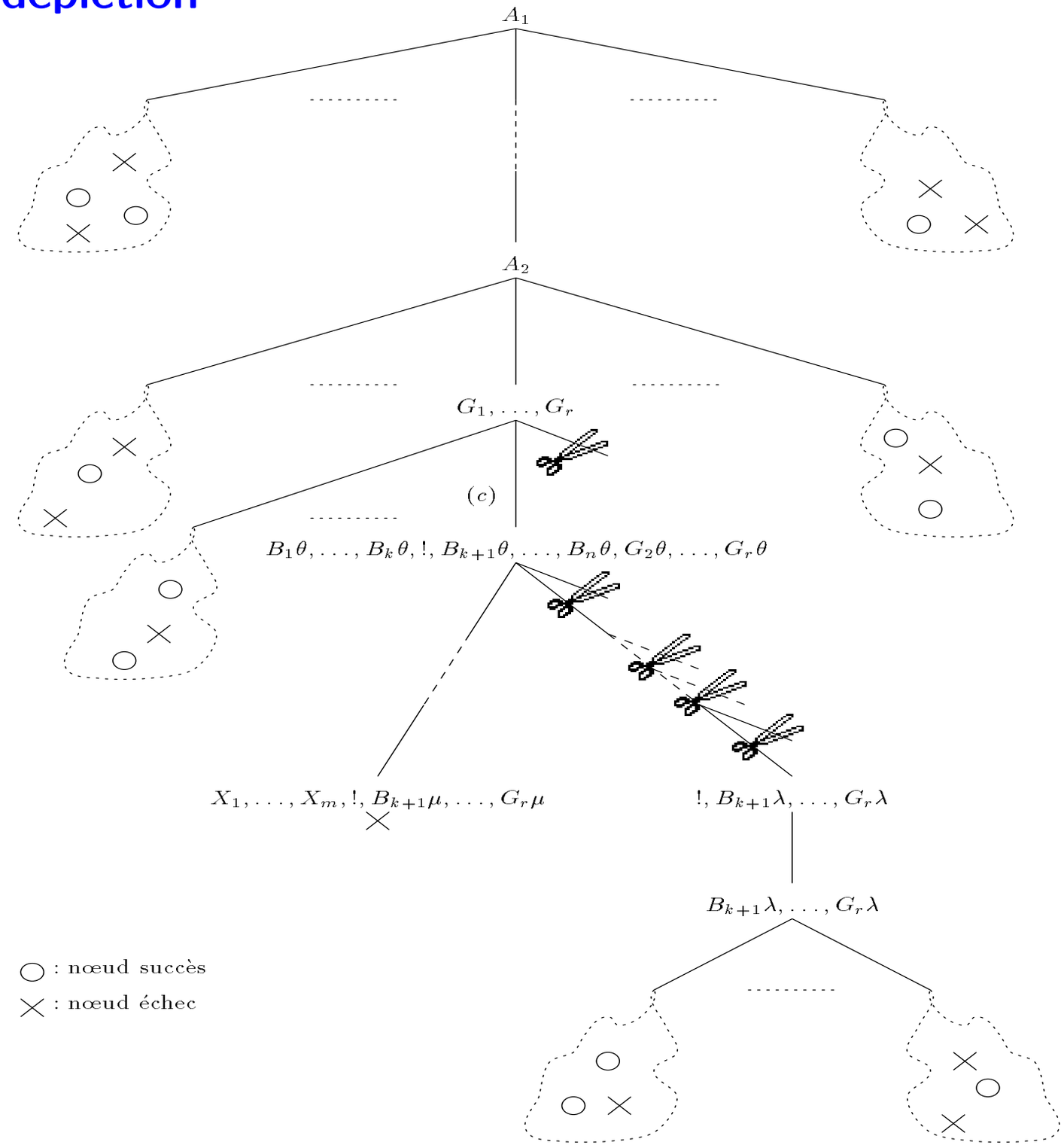
The search tree exploration can go on starting from the father of G.

Cut VI: Dynamic definition

Cuts take effect during backtraking, when going up from some node to its father; if this father's label starts with a cut, then backtraking goes up until the father of the ancestor responsible for the selection of the cut-containing clause.

In the special case where the cut was in the main goal, such a backtracking stops the tree exploration.

Graphical depiction



Cut VIII: An example

```
/* cut */
p1(X,Y) :- p2(X), !, p3(Y).
p1(X,Y) :- p4(Y).
/* if p2 then p3 else p4 */
p2(a).
p2(b).
p3(b).
p4(a).
p5(a).
p5(b).


?- p1(a,a).
    --> no

?- p5(X), p1(X,X).
    --> X = b ;
    --> no
```

Cut IX: Negation as cut

```
not(G) :- G,!,fail.
not(G).
```

Funny, isn't it?

Recognizers I

```
?- atom(paulette).
    --> yes
?- atom(friend(sandra)).
    --> no
?- atom(X).
    --> no
?- atom(123).
    --> no

?- integer(123).
    --> yes
?- integer(1.23).
    --> no

?- number(123).
    --> yes
?- number(1.23).
    --> yes
```

Recognizers II

```
?- atomic(paulette).
    --> yes
?- atomic(friend(sandra)).
    --> no
?- atomic(X).
    --> no
?- atomic(123).
    --> yes
```

A useful primitive predicate

```
?- var(f(X)).
    --> no
?- var(X).
    --> X = _0

?- X=a, var(X).
    --> no
?- var(X), X=a, atom(X).
    --> X = a
?- X=Y, var(X), var(Y).
    --> X = _0 , Y = _0
?- X=f(Y).
    --> X = f(_1) , Y = _1
?- X=f(Y), var(X).
    --> no
```

Arithmetics I

```
?- 3 == 3.
    --> yes
?- 1+2 == +(1,2).
    --> yes
?- 1+2 == 3.
    --> no
?- 1+2 = 3.
    --> no
?- X is 12+44.
    --> X = 56
?- X is 1+2*3.
    --> X = 7
?- 7 is 1+2*3.
    --> yes
?- 9 is 1+2*3.
    --> no
```

Arithmetics II

```
?- X = 22, Y is 2*X.
    --> X = 22 , Y = 44
?- X is 2+3, Y is X+X.
    --> X = 5 , Y = 10
?- X = 2+3, Y is X+X.
    --> X = 2+3 , Y = 10
```

Arithmetics III

```
?- X is log(2).
    --> X = 0.693147
?- X is log10(2).
    --> X = 0.30103
?- 3 is 2+1.
    --> yes
?- x is 4+3.
    --> no
?- X is 4+P.
    --> ! Error: not a number
?- f(X) is 3+8.
    --> no
?- X = log10(2), Y is X.
    --> X = log10(2) , Y = 0.30103
?- X = Y + log10(Z), Y = 1, Z = 6-4, R is X.
    --> X=1+log10(6-4), Y=1, Z=6-4, R=1.30103
```

Some examples

```
/* factorial */
fact(1,1) :- !.
fact(N,R) :- integer(N), N > 1, N1 is N-1,
             fact(N1,R1), R is N*R1.
```

```
?- [H|T] = [a].
    --> H = a , T = []
?- [A1,A2,_,A4 |T] = [what,we,call,a,rose].
    --> A1 = what, A2 = we, A4 = a, T = [rose]
```

Term analysis I

Lists are just anonymous terms?

What about usual terms?

```
?- f(a,b,c) =.. L.
    --> L = [f,a,b,c]
?- f(a,b,c) =.. [F|Args].
    --> F = f , Args = [a,b,c]
?- (a*b+c) =.. [Op|Args].
    --> Op = + , Args = [a*b,c]
```

Term analysis II

```
?- functor(owns(john,umbrella), F, N).
    --> F = owns , N = 2
?- functor(f(a,b), f, 2).
    --> yes
?- arg(2, owns(john, umbrella), A).
    --> A = umbrella
```

```
?- name(blabla,L).
    --> L = [98,108,97,98,108,97]
?- name(bac, [A|_]) , name(X,[A]).
    --> A = 98 , X = b
```

A recapitulation predicate I

```
mother(vera,julie).     father(joseluis,julie).
mother(vera,alice).     father(joseluis,alice).
mother(mary,joseluis).  father(alfonso,joseluis).
mother(anne,vera).      father(andrew,vera).
grandfather(X,Y) :- father(X,Z),mother(Z,Y).
grandfather(X,Y) :- father(X,Z),father(Z,Y).
grandmother(X,Y) :- mother(X,Z),mother(Z,Y).
grandmother(X,Y) :- mother(X,Z),father(Z,Y).

all(F) :- functor(T,F,2), T, write('the '),
          write(F), arg(2,T,A2), write(' of '),
          write(A2), arg(1,T,A1),
          write(' is '), write(A1),nl,fail.
```

A recapitulation predicate II

```
?- all(grandmother).
    the grandmother of julie is anne
    the grandmother of alice is anne
    the grandmother of julie is mary
    the grandmother of alice is mary
    --> no
```

Native sorting predicate

```
?- sort([2,2,3,1],L).
 --> L = [1,2,3]

?- sort([[a],[a,b],f(x),f(a,b),f(a,X),X],L).
 --> X = _31 , L = [_31,f(x),[a],[a,b],f(a,_31),f(a,b)]

?- sort([[a],[a,b],f(x),f(a,b),f(a,X),X],L).
 --> L = [X,f(x),f(a,X),f(a,b),[a],[a,b]]
```

variable `@<` number `@<` symbol `@<` coumpound term

variables and coumpound term: system-dependent ordering

Operators, precedence I

```
?- a+b*c = X + Y.
    --> X = a , Y = b*c
?- a+b+c = X + Y.
    --> X = a+b , Y = c
?- -a+b+c = X + Y.
    --> X = -a+b , Y = c

?- X = union(a,b), write(X), nl,
   op(500, yfx, union), write(X).
    union(a,b)
    a union b
    --> X = a union b

?- Y = fact(a), write(Y), nl,
    op(700, xf, fact), write(Y).
    fact(a)
    a fact
    --> Y = a fact
```

Operators, precedence II

Operator definitions in Prolog:

```
:-  op(Precedence,Type,Name).
```

Precedence is a number between 0 and 1200. The higher the number, the greater the precedence. Type is an atom specifying the type and associativity of the operator. In the case of + this atom is yfx , which says that + is an infix operator; the f represents the operator, and the x and y the arguments. Furthermore, x stands for an argument which has a precedence which is lower than the precedence of + and y stands for an argument which has a precedence which lower or equal to the precedence of + . There are the following possibilities for type:

```
infix  xfx , xfy , yfx
prefix  fx , fy
suffix  xf , yf
```

Operators, precedence III

```
:-  op(  1200,  xfx,  [  :-,  -->  ]).
:-  op(  1200,   fx,  [  :-,  ?-  ]).
:-  op(  1100,  xfy,  [  ;  ]).
:-  op(  1000,  xfy,  [  ’,’  ]).
:-  op(   700,  xfx,  [  =,  is,  =..,  ==,  \==,
                        =:=,  =\=,  <,  >,  =<,  >=  ]).
:-  op(   500,  yfx,  [  +,  -]).
:-  op(   500,   fx,  [  +,  -  ]).
:-  op(   300,  xfx,  [  mod  ]).
:-  op(   200,  xfy,  [  ^  ]).
```

Operator definitions do not specify the meanings of operators!

Difference lists I

Definition. `L1-L2` stands for `L3` iff `append(L3,L2,L1)`.

Example. `[a,b,c,d]-[c,d]` stands for `[a,b]`.

This "virtual concatenation" is in constant time;
Real concatenation is in linear time.

Most important application:

```
dappend(Xs-Ys,Ys-Zs,Xs-Zs).
```

Difference lists II

```
image2(Xs,Ys) :- dimage(Xs,Ys-[]).
dimage([],Xs-Xs).
dimage([X|Xs],Ys-Zs) :- dimage(Xs,Ys-[X|Zs]).

?- dimage([a,b],L-[]).  --> L = [b,a]
?- image2([a,b,c],L).   --> L = [c,b,a]
?- image2(L,[b,a]).     --> L = [a,b]
```

```
?- dimage([a,b], L-[]).
%    dimage([b],L-[a])
%    dimage([],L-[b,a])
%    dimage([],[b,a]-[b,a])
      --> L = [b,a]
```

Difference lists III

```
flatten([],[]).
flatten(X,[X]) :- atomic(X), X \= [].
flatten([X|Xs],Ys) :-
  flatten(X,Us), flatten(Xs,Vs), append(Us,Vs,Ys).

dflatten([],Xs-Xs).
dflatten(X,[X|Xs]-Xs) :- atomic(X), X \= [].
dflatten([X|Xs],Ys-Zs) :-
  dflatten(X,Us-As), dflatten(Xs,Vs-Bs), dappend(Us-As,Vs-Bs,Ys-Zs).

dflatten([],Xs-Xs).
dflatten(X,[X|Xs]-Xs) :- atomic(X), X \= [].
dflatten([X|Xs],Ys-Zs) :-
  dflatten(X,Ys-Rs), dflatten(Xs,Rs-Zs)/*, dappend(Ys-Rs,Rs-Zs,Ys-Zs)*/.
```

Blocks world I

```
trans(S1,S2,P) :- t0(S1,S2,[S1],P).

t0(S,S,V,[]).
t0(S1,S2,V,[A|As]) :- legal(A,S1), res(A,S1,S),
                      not(member(S,V)), t0(S,S2,[S|V],As).

legal(m(B,Y,P),S) :- on(B,Y,S), cl(B,S), p(P), cl(P,S).
legal(m(B1,Y,B2),S) :- on(B1,Y,S), cl(B1,S), b(B2), B1\=B2, cl(B2,S).

on(X,Y,S) :- member([X|Y],S).

cl(X,S) :- member([Y|X],S),!,fail.
cl(X,S).

res(m(X,Y,Z),S,S1) :- sub([X|Y],[X|Z],S,S1).

sub(X,Y,[X|Xs],[Y|Xs]).
sub(X,Y,[Z|Xs],[Z|Ys]) :- sub(X,Y,Xs,Ys).
```

Blocks world II

```
p(p).  p(q).  p(r).
b(a).  b(b).  b(c).
```

```
?- trans([[a|b],[b|p],[c|r]],[[a|b],[b|c],[c|r]],Z).
   Z = [m(a,b,q),m(a,q,c),m(b,p,q),m(a,c,p),
m(a,p,b),m(c,r,p),m(a,b,r),m(a,r,c),m(b,q,r),
m(a,c,q),m(a,q,b),m(c,p,q),m(a,b,p),m(a,p,c),
m(b,r,p),m(a,c,r),m(b,p,a),m(c,q,p),m(b,a,c),
m(a,r,q),m(b,c,a),m(c,p,r),m(b,a,c),m(a,q,p),
m(a,p,b)]
Yes ,
...
```

Blocks world III

```
trans(S1,S2,P) :- t0(S1,S2,[S1],P).

t0(S,S,V,[]).
t0(S1,S2,V,[A|As]) :- next(A,S1,S2), res(A,S1,S),
                      not(member(S,V)), t0(S,S2,[S|V],As).

next(A,S1,S2) :- hint(A,S2), legal(A,S1).
next(A,S1,S2) :- legal(A,S1).

hint(m(X,Y,Z),S) :- on(X,Z,S).

legal(m(B,Y,P),S) :- on(B,Y,S), cl(B,S), p(P), cl(P,S).
legal(m(B1,Y,B2),S) :- on(B1,Y,S), cl(B1,S), b(B2), B1\=B2, cl(B2,S).

on(X,Y,S) :- member([X|Y],S).

cl(X,S) :- member([Y|X],S),!,fail.
cl(X,S).
```

Blocks world IV

```
p(p).  p(q).  p(r).
b(a).  b(b).  b(c).


?- trans([[a|b],[b|p],[c|r]],[[a|b],[b|c],[c|r]],Z).
   Z = [m(a,b,q),m(b,p,c),m(a,q,b)]
Yes ,
...
```

Term analysis III

Terms and subterms.

```
subterm(Term,Term).
subterm(Sub,Term) :- compound(Term),
                     functor(Term,F,N),
                     subterm(N,Sub,Term).

subterm(N,Sub,Term) :- arg(N,Term,Arg),
                       subterm(Sub,Arg).
subterm(N,Sub,Term) :- N > 1, N1 is N-1,
                       subterm(N1,Sub,Term).
```

Term analysis IV: term substitution

```
substitute(Old,New,Old,New).

substitute(Old,New,Term,Term)
 :- atomic(Term) , Term \= Old.

substitute(Old,New,Term,Term1)
 :- compound(Term), functor(Term,F,N), functor(Term1,F,N),
    substitute(N,Old,New,Term,Term1).

substitute(N,Old,New,Term,Term1)
 :- N > 0, arg(N,Term,Arg),
    substitute(Old,New,Arg,Arg1), arg(N,Term1,Arg1),
    N1 is N-1, substitute(N1,Old,New,Term,Term1).

substitute(0,Old,New,Term,Term1).
```

Term analysis V: term substitution, trace

```
substitute(cat,dog,owns(jane,cat),X)        X=owns(jane,cat)
    atomic(owns(jane,cat))                  f
substitute(cat,dog,owns(jane,cat),X)
    compound(owns(jane,cat))
    functor(owns(jane,cat),F,N)             F=owns, N=2
    functor(X,owns,2)                       X=owns(X1,X2)
    substitute(2,cat,dog,owns(jane,cate),owns(X1,X2))
        2 > 0
        arg(2,owns(jane,cat),Arg)           Arg=cat
        substitute(cat,dog,cat,Arg1)        Arg1=dog
        arg(2,owns(X1,X2),dog)              X2=dog
        N1 is 2-1                           N1=1
        substitute(1,cat,dog,owns(jane,cat),owns(X1,dog))
            1 > 0
            arg(1,owns(jane,cat),Arg2)      Arg2=jane
            substitute(cat,dog,jane,Arg3)   Arg3=jane
                atomic(jane)
                jane \= cat
            arg(1,owns(X1,dog),jane)        X1=jane
            N2 is 1-1                       N2=0
            substitute(0,cat,dog,owns(jane,cat),owns(jane,dog))
                0 > 0                       f
            substitute(0,cat,dog,owns(jane,cat),owns(jane,dog))
                true
                    Output: (X=owns(jane,dog))
```

Metalogic predicates I

```
plus(X,Y,Z) :- nonvar(X), nonvar(Y), Z is X+Y.

plus(X,Y,Z) :- nonvar(X), nonvar(Z), Y is Z-X.

plus(X,Y,Z) :- nonvar(Y), nonvar(Z), X is Z-Y.
```

Clause ordering optimisation.

```
grandparent(X,Z) :- nonvar(X),
                    parent(X,Y), parent(Y,Z).

grandparent(X,Z) :- nonvar(Z),
                    parent(Y,Z), parent(X,Y).
```

Metalogic predicates II

Ground terms.

```
ground(Term) :- nonvar(Term), atomic(Term).

ground(Term) :- nonvar(Term), compound(Term),
                functor(Term,F,N), ground(N,Term).

ground(0,Term).

ground(N,Term) :- N > 0, arg(N,Term,Arg), ground(Arg),
                  N1 is N-1, ground(N1,Term).
```

Metalogic predicates III: unification

```
unify(X,X).   % naive version


unify(X,Y) :- var(X), var(Y), X=Y.

unify(X,Y) :- var(X), nonvar(Y), X=Y.

unify(X,Y) :- var(Y), nonvar(X), Y=X.

unify(X,Y) :- nonvar(X), nonvar(Y),
              atomic(X), atomic(Y), X=Y.

unify(X,Y) :- nonvar(X), nonvar(Y),
              compound(X), compound(Y),
              term_unify(X,Y).
```

Metalogic predicates IV: unification

```
term_unify(X,Y) :- functor(X,F,N),
                   functor(Y,F,N),
                   unify_args(N,X,Y).

unify_args(N,X,Y) :- N > 0,
                     unify_arg(N,X,Y),
                     N1 is N-1,
                     unify_args(N1,X,Y).

unify_args(0,X,Y).

unify_arg(N,X,Y) :- arg(N,X,ArgX),
                    arg(N,Y,ArgY),
                    unify(ArgX,ArgY).
```

Metalogic predicates V: unification

```
unify(X,Y) :- var(X), var(Y), X=Y.
unify(X,Y) :- var(X), nonvar(Y), no_oc(X,Y), X=Y.
unify(X,Y) :- var(Y), nonvar(X), no_oc(Y,X), Y=X.
unify(X,Y) :- nonvar(X), nonvar(Y), atomic(X), atomic(Y), X=Y.
unify(X,Y) :- nonvar(X), nonvar(Y), compound(X), compound(Y),
              term_unify(X,Y).
```

```
term_unify(X,Y) :- functor(X,F,N), functor(Y,F,N), unify_args(N,X,Y).
unify_args(N,X,Y) :- N > 0, unify_arg(N,X,Y),
                     N1 is N-1, unify_args(N1,X,Y).
unify_args(0,X,Y).
unify_arg(N,X,Y) :- arg(N,X,ArgX), arg(N,Y,ArgY), unify(ArgX,ArgY).
```

Metalogic predicates VI: unification

```
no_oc(X,Y) :- var(Y), X \== Y.
no_oc(X,Y) :- nonvar(Y), atomic(Y).
no_oc(X,Y) :- nonvar(Y), compound(Y), functor(Y,F,N), no_oc(N,X,Y).


no_oc(N,X,Y) :- N > 0, arg(N,Y,Arg), no_oc(X,Arg),
                N1 is N-1, no_oc(N1,X,Y).

no_oc(0,X,Y).
```

Grammars I

```
sentence(Ph) :-
  subject(S), verb(V), complement(C), append(S,V,Z), append(Z,C,Ph).
subject(S) :- article(A), noun(N), append(A,N,S).
complement(C) :- adject(C).

 article([the]).
 noun([lady]).
 noun([bike]).
 noun([whale]).
 adject([blue]).
 adject([charming]).
 adject([nice]).
 verb([becomes]).
 verb([is]).
```

Grammars II

```
?- sentence([the,lady,is,white]).
    --> no
?- sentence(L).
    --> L = [the,lady,becomes,blue] ;
    --> L = [the,lady,becomes,charming] ;
    --> L = [the,lady,becomes,nice] ;
    --> L = [the,lady,is,blue] ;
    --> L = [the,lady,is,charming] ;
    --> L = [the,lady,is,nice] ;
    --> L = [the,bike,becomes,blue] ;
    --> L = [the,bike,becomes,charming] ;
    --> L = [the,bike,becomes,nice] ;
    --> L = [the,bike,is,blue] ;
    --> L = [the,bike,is,charming] ;
    --> L = [the,bike,is,nice] ;
    --> L = [the,whale,becomes,blue] ;
    --> L = [the,whale,becomes,charming] ;
    --> L = [the,whale,becomes,nice] ;
    --> L = [the,whale,is,blue] ;
    --> L = [the,whale,is,charming] ;
    --> L = [the,whale,is,nice] ;
    --> no
```

Grammars III

```
sentence(L,X) :- subject(L,L1), verb(L1,L2), complement(L2,X).
subject(L,X) :- article(L,L1), noun(L1,X).
complement(L,X) :- adjective(L,X).
article(L,X) :- c(L,the,X).
noun(L,X) :- c(L,lady,X).
noun(L,X) :- c(L,bike,X).
noun(L,X) :- c(L,whale,X).
adjective(L,X) :- c(L,blue,X).
adjective(L,X) :- c(L,charming,X).
adjective(L,X) :- c(L,nice,X).
verb(L,X) :- c(L,becomes,X).
verb(L,X) :- c(L,is,X).
c([A|X],A,X).

?- subject([the,whale,goes,by,train], X). --> X = [goes,by,train]
?- sentence(L,[]). --> L = [the, lady, becomes, blue]
```

Grammars IV

```
sentence --> subject, verb, complement.
subject --> article, noun.
complement --> adjective.
article --> [the].
noun --> [lady].
noun --> [bike].
noun --> [whale].
adjective --> [blue].
adjective --> [charming].
adjective --> [nice].
verb --> [becomes].
verb --> [is].

?- sentence([the,lady,is,blue] , []). --> yes
?- sentence(L,[]).  --> L = [the,lady,becomes,blue] ;
                    --> L = [the,lady,becomes,charming] ;
                    etc.
```

Grammars V

```
number(Val) --> list(Val,0,_).
number(V1+V2) --> list(V1,0,_), ['.'], list(V2,-Len,Len).

list(Val,Wei,1) --> bit(Val,Wei).
list(V1+V2,Wei,Len+1) --> bit(V1,Wei+Len), list(V2,Wei,Len).

bit(0,_) --> [0].
bit(2**Wei,Wei) --> [1].

s(N,Val,X) :- number(Val,N,[]), X is Val.
```

Grammars VI

```
number_t(Val,Xs,Ys) :- list_t(Val,0,_,Xs,Ys).
number_t(V1+V2,Xs,Ys) :-
   list_t(V1,0,_,Xs,['.'|Zs]), list_t(V2,-Len,Len,Zs,Ys).

list_t(Val,Wei,1,Xs,Ys) :- bit_t(Val,Wei,Xs,Ys).
list_t(V1+V2,Wei,Len+1,Xs,Ys) :-
   bit_t(V1,Wei+Len,Xs,Zs), list_t(V2,Wei,Len,Zs,Ys).

bit_t(0,_,[0|Xs],Xs).
bit_t(2**Wei,Wei,[1|Xs],Xs).

s_t(N,Val,X) :- number_t(Val,N,[]), X is Val.
```

Grammars VII

```
 ?- s([1,0,1],V,X).
V = 2**(0+(1+1))+(0+2**0)
X = 5

 ?- s([1,0,'.',0,1,0],V,X).
V = 2**(0+1)+0+(0+(2**(-(1+1+1)+1)+0))
X = 2.25

 ?- s([1,0,'.',0,'.',1,0],V,X).
No

 ?- findall(X,s([_,_,_],_,X),L).
X = _G329
L = [0, 1, 2, 3, 4, 5, 6,
     7, 0, 0.5, 1, 1.5]

?- s(_,_,X).
X = 0 ; X = 1 ; X = 0 ; X = 1 ; X = 0 ...
```

Grammars VIII

```
list_N(L,[]).
list_N(L,[0|Xs]) :-
  L>0, L1 is L-1, list_N(L1,Xs).
list_N(L,[1|Xs]) :-
  L>0, L1 is L-1, list_N(L1,Xs).
list_N(L,['.'|Xs]) :-
  L>0, L1 is L-1, list_N(L1,Xs).

write_elem([]).
write_elem([X|Xs]) :-
  write(X), nl, write_elem(Xs).

all+(N) :-
  findall([X,Xs,Val],
          (list_N(N,Xs), s(Xs,Val,X)),
          L),
  write_elem(L),
  fail.
```

Grammars IX

```
 ?- all+(3).
[0, [0], 0]
[0, [0, 0], 0+0]
[0, [0, 0, 0], 0+(0+0)]
[1, [0, 0, 1], 0+(0+2**0)]
[1, [0, 1], 0+2**0]
[2, [0, 1, 0], 0+(2**(0+1)+0)]
[3, [0, 1, 1], 0+(2**(0+1)+2**0)]
[0, [0, ., 0], 0+0]
[0.5, [0, ., 1], 0+2**(-1)]
[1, [1], 2**0]
[2, [1, 0], 2**(0+1)+0]
[4, [1, 0, 0], 2**(0+(1+1))+(0+0)]
[5, [1, 0, 1], 2**(0+(1+1))+(0+2**0)]
[3, [1, 1], 2** (0+1)+2**0]
[6, [1, 1, 0], 2**(0+(1+1))+(2**(0+1)+0)]
[7, [1, 1, 1], 2**(0+(1+1))+(2**(0+1)+2**0)]
[1, [1, ., 0], 2**0+0]
[1.5, [1, ., 1], 2**0+2**(-1)]
No
```

Self-modifying code I

Inserting a clause `<clause>` at the beginning or at the end of the program.

```
asserta((<clause>)).
assertz((<clause>)).
```

Retracting the first clause unifying with `<C>`.

```
retract((<C>)).
```

A way to update the knowledge base – handle with care!

Self-modifying code II: memoization

```
hanoi_1(0,A,B,C,0) :- !.
hanoi_1(N,A,B,C,NMoves) :- N1 is N-1,
                           hanoi_1(N1,A,C,B,NM1),
                           hanoi_1(N1,C,B,A,NM2),
                           NMoves is NM1+1+NM2.

hanoi_2(0,A,B,C,0) :- !.
hanoi_2(N,A,B,C,NMoves) :- N1 is N-1,
                           lemma(hanoi_2(N1,A,C,B,NM1)),
                           hanoi_2(N1,C,B,A,NM2),
                           NMoves is NM1+1+NM2.

lemma(P) :- P, asserta((P :- !)).

test_hanoi_2(N,LPegs,NMoves) :- hanoi_2(N,A,B,C,NMoves),
                                LPegs = [A,B,C].
```

Self-modifying code III: memoization (listing)

```
append('[]', A, A).
append([B|C], A, [B|D]) :- append(C, A, D).

hanoi_1(0, _, _, _, 0) :- !.
hanoi_1(A, B, C, D, E) :- F is A-1,
                          hanoi_1(F, B, D, C, G),
                          hanoi_1(F, D, C, B, H),
                          E is G+1+H.

hanoi_2(0, _, _, _, 0) :- !.
hanoi_2(A, B, C, D, E) :- F is A-1,
                          lemma(hanoi_2(F,B,D,C,G)),
                          hanoi_2(F, D, C, B, H),
                          E is G+1+H.

lemma(A) :- A, asserta((A:-'!')).

test_hanoi_2(A, B, C) :- hanoi_2(A, D, E, F, C), B=[D,E,F].
```

Self-modifying code IV: memoization

```
| ?- hanoi_1(12,a,b,c,LM).

LM = 4095?
yes         /* VERY SLOW !! */

| ?- test_hanoi_2(12,[a,b,c],LM).

LM = 4095?
yes         /* VERY FAST !! */
```

Self-modifying code V: memoization

Before:

```
| ?- listing.
append('[]', A, A).
append([B|C], A, [B|D]) :-
        append(C, A, D).
hanoi_1(0, _, _, _, 0) :- !.
hanoi_1(A, B, C, D, E) :-
        F is A-1,
        hanoi_1(F, B, D, C, G),
        hanoi_1(F, D, C, B, H),
        E is G+1+H.
```

Self-modifying code VI: memoization

After (memoized results):

```
hanoi_2(11, _, _, _, 2047) :- !.
hanoi_2(10, _, _, _, 1023) :- !.
hanoi_2(9, _, _, _, 511) :- !.
hanoi_2(8, _, _, _, 255) :- !.
hanoi_2(7, _, _, _, 127) :- !.
hanoi_2(6, _, _, _, 63) :- !.
hanoi_2(5, _, _, _, 31) :- !.
hanoi_2(4, _, _, _, 15) :- !.
hanoi_2(3, _, _, _, 7) :- !.
hanoi_2(2, _, _, _, 3) :- !.
hanoi_2(1, _, _, _, 1) :- !.
hanoi_2(0, _, _, _, 0) :- !.
```

Self-modifying code VI: memoization

After (program clauses):

```
hanoi_2(A, B, C, D, E) :-
        F is A-1,
        lemma(hanoi_2(F,B,D,C,G)),
        hanoi_2(F, D, C, B, H),
        E is G+1+H.

lemma(A) :-
        A,
        asserta((A:-'!')).

test_hanoi_2(A, B, C) :-
        hanoi_2(A, D, E, F, C),
        B=[D,E,F].
```

Metainterpretation I

```
solve(A) :- A.
```

Interactive shell.

```
toy_shell :- prompt, read(G), toy_shell(G).

toy_shell(exit) :- !.
toy_shell(G) :- ground(G), !, solve_g(G), toy_shell.
toy_shell(G) :- solve_ng(G), toy_shell.

solve_ng(G) :- G, write(G), nl, fail.
solve_ng(G) :- write('No (more) solution'), nl.

solve_g(G) :- G, !, write('Yes'), nl.
solve_g(G) :- write('No'), nl.

prompt :- write('Next command ? ').
```

Metainterpretation II

```
?- toy_shell.  % NEW WAY TO WRITE RESULTS
Next command ? append(X,Y,[1,2,3,4]).
append([],[1,2,3,4],[1,2,3,4])
append([1],[2,3,4],[1,2,3,4])
append([1,2],[3,4],[1,2,3,4])
append([1,2,3],[4],[1,2,3,4])
append([1,2,3,4],[],[1,2,3,4])
No (more) solution
Next command ? X is 2+3 .
5 is 2+3
No (more) solution
Next command ? exit.
Yes
```

Metainterpretation II (bis)

```
?- toy_shell. % COLLECTING EFFECT
Next command ? append(X,[a,b],Y).
append([],[a,b],[a,b])
append([_1],[a,b],[_1,a,b])
append([_1,_2,_3],[a,b],[_1,_2,_3,a,b])
[diverge]
```

Metainterpretation III

```
solve(true).
solve((A,B)) :- solve(A),solve(B).
solve(A) :- rule(A,B),solve(B).
```

```
A.
E :- F,G,H.
```

is translated into

```
rule(A,true).
rule(E,(F,(G,H))).
```

```
rule(member(X,[X|Ys]),true).
rule(member(X,[Y|Ys]),member(X,Ys)).
```

Metainterpretation IV

```
?- solve(member(X,[a,b,c])).
Call: solve(member(_1,[a,b,c])) ?
  Call: rule(member(_1,[a,b,c]),_2) ?
  Exit: rule(member(a,[a,b,c]),true) ?
  Call: solve(true) ?
  Exit: solve(true) ?
Exit: solve(member(a,[a,b,c])) ?
X = a ;
  Redo: rule(member(_1,[a,b,c]),_2) ?
  Exit: rule(member(_1,[a,b,c]),member(_1,[b,c])) ?
  Call: solve(member(_1,[b,c])) ?
    Call: rule(member(_1,[b,c]),_4) ?
    Exit: rule(member(b,[b,c]),true) ?
    Call: solve(true) ?
    Exit: solve(true) ?
  Exit: solve(member(b,[b,c])) ?
Exit: solve(member(b,[a,b,c])) ?
X = b ;
...
```

Metainterpretation V

"(E,(F,(G,H)))" becomes "[E,F,G,H]".

Classical solution.

```
solve([]) :- !.
solve([G|Gs]) :- !, rule(G,T), append(T,Gs,Gs1), solve(Gs1).
solve(G) :- solve([G]).
```

Continuation passing style.

```
solve(G) :- solve(G,[]).

solve([],[]).
solve([],[G|Gs]) :- solve(G,Gs).
solve([A|B],Gs) :- append(B,Gs,Gs1), solve(A,Gs1).
solve(A,Gs) :- rule(A,B), solve(B,Gs).
```

Metainterpretation VI

Computation display, pretty printing.

```
dis_gl([],L) :- !.
dis_gl(G,0) :- !, dis_gl(G), display('?\n').
dis_gl(G,L) :- L1 is L-1, display('  '), dis_gl(G,L1).

dis_sl([],L) :- !.
dis_sl(G,0) :- !, dis_gl(G), display('!\n').
dis_sl(G,L) :- L1 is L-1, display('  '), dis_sl(G,L1).

dis_gl([]) :- !.
dis_gl([G]) :- !, display(G).
dis_gl([G1|[G2|Gs]]) :- !, display(G1), display(','), dis_gl([G2|Gs]).
dis_gl(G) :- dis_gl([G]).
```

Metainterpretation VI

```
solve([],L) :- !.
solve([G|Gs],L) :- !, rule(G,T), L1 is L+1,
                   append(T,Gs,Gs1), dis_gl(Gs1,L1),
                   solve(Gs1,L1), dis_sl(Gs1,L1).
solve(G,L) :- solve([G],L).

metatrace(G) :- dis_gl(G,1), solve(G,1), dis_sl(G,1).
```

Metainterpretation VII

```
rule(append([],Ys,Ys),[]).
rule(append([X|Xs],Ys,[X|Zs]), [append(Xs,Ys,Zs)]).


?- metatrace(append([a,b],[c,d],Xs)).
  append([a,b],[c,d],_10)?
    append([b],[c,d],_20)?
      append([],[c,d],_30)?
      append([],[c,d],[c,d])!
    append([b],[c,d],[b,c,d])!
  append([a,b],[c,d],[a,b,c,d])!
  Xs = [a,b,c,d]
Yes ,
No
```

Metainterpretation VIII

```
rule(member(X,[X|Ys]),[]).
rule(member(X,[Y|Ys]),[member(X,Ys)]).
rule(member2a(X,Xss),[member(X,Xs),member(Xs,Xss)]).
rule(member2b(X,Xss),[member(Xs,Xss),member(X,Xs)]).
```

```
?- metatrace(member2a(X,[[a],[b,c]])).
  member2a(_1,[[a],[b,c]])?
    member(_1,_2),member(_2,[[a],[b,c]])?
      member([_1|_3],[[a],[b,c]])?
      member([a],[[a],[b,c]])!
    member(a,[a]),member([a],[[a],[b,c]])!
  member2a(a,[[a],[b,c]])!
X = a ;
        member([_1|_3],[[b,c]])?
        member([b,c],[[b,c]])!
      member([b,c],[[a],[b,c]])!
    member(b,[b,c]),member([b,c],[[a],[b,c]])!
  member2a(b,[[a],[b,c]])!
X = b ;
... ... ... ... ... ... ... ... ...
```

Metainterpretation IX

```
... ... ... ... ... ... ... ... ...
          member([_1|_3],[])?
      member(_1,_3),member([_4|_3],[[a],[b,c]])?
        member([_4,_1|_5]),[[a],[b,c]])?
          member([_4,_1|_5]),[[b,c]])?
          member([b,c],[[b,c]])!
        member([b,c],[[a],[b,c]])!
      member(c,[c]),member([b,c],[[a],[b,c]])!
    member(c,[b,c]),member([b,c],[[a],[b,c]])!
  member2a(c,[[a],[b,c]])!
X = c ;
            member([_4,_1|_5]),[])?
        member(_1,_5),member([_4,_6|_5],[[a],[b,c]])?
          member([_4,_6,_1|_7],[[a],[b,c]])?
            member([_4,_6,_1|_7],[[b,c]])?
              member([_4,_6,_1|_7],[])?
        member(_1,_7),member([_4,_6,_8|_7],
                             [[a],[b,c]])?
[diverge]
```

Metainterpretation X

```
?- metatrace(append(Xs,Ys,[a,b,c])).
  append(_8,_9,[a,b,c])?
  append([],[a,b,c],[a,b,c])!
  Xs = []
  Ys = [a,b,c] ;
    append(_58,_9,[b,c])?
    append([],[b,c],[b,c])!
  append([a],[b,c],[a,b,c])!
  Xs = [a]
  Ys = [b,c] ;
      append(_78,_9,[c])?
      append([],[c],[c])!
    append([b],[c],[b,c])!
  append([a,b],[c],[a,b,c])!
  Xs = [a,b]
  Ys = [c] ;
```

Metainterpretation X (bis)

```
        append(_98,_9,[])?
        append([],[],[])!
      append([c],[],[c])!
    append([b,c],[],[b,c])!
  append([a,b,c],[],[a,b,c])!
  Xs = [a,b,c]
  Ys = [] ;
No
```

Metainterpretation XI

Program (for metainterpretation):

```
rule(pere(eric,jeanne), []).
rule(pere(paul,jacques), []).
rule(pere(paul,catherine), []).

rule(mere(francoise,catherine), []).
rule(mere(catherine,marie), []).
rule(mere(catherine,jeanne), []).
rule(mere(marie,nathalie), []).

rule(parent(X,Y), [pere(X,Y)]).
rule(parent(X,Y), [mere(X,Y)]).

rule(grandparent(X,Y), [parent(X,Z), parent(Z,Y)]).
```

Metainterpretation XII

```
?- metatrace(grandparent(X,marie)).
  grandparent(_8,marie)?
    parent(_8,_49),parent(_49,marie)?
      pere(_8,_49),parent(_49,marie)?
        parent(jeanne,marie)?
          pere(jeanne,marie)?
          mere(jeanne,marie)?
        parent(jacques,marie)?
          pere(jacques,marie)?
          mere(jacques,marie)?
        parent(catherine,marie)?
          pere(catherine,marie)?
          mere(catherine,marie)?
          mere(catherine,marie)!
        parent(catherine,marie)!
      pere(paul,catherine),parent(catherine,marie)!
    parent(paul,catherine),parent(catherine,marie)!
  grandparent(paul,marie)!
```

Metainterpretation XII (bis)

```
X = paul ;
      mere(_8,_49),parent(_49,marie)?
        parent(catherine,marie)?
          pere(catherine,marie)?
...   ...   ...   ...   ...
X = francoise ;
        parent(marie,marie)?
...   ...   ...   ...   ...
No
```

Metainterpretation XIII

To be improved:

- Which clauses are used?
- What happens when no usable clause is left?
- No logical proof.

Solutions:

- Just display used clauses "$\{A$:- $B, C.\}$".
- Issue a warning: "{cannot unify.}".
- Proof is reconstructed from goals and subgoals.

Metainterpretation XIV

```
dis_gl([],0) :- !,display('empty goal\n').
dis_gl(G,0) :- !, dis_gl(G), display('?\n').
dis_gl(G,L) :- L1 is L-1, display('  '), dis_gl(G,L1).

dis_gl([]) :- !.
dis_gl([G]) :- !,display(G).
dis_gl([G1|[G2|Gs]]) :- !, display(G1), display(','), dis_gl([G2|Gs]).
dis_gl(G) :- dis_gl([G]).

dis_rl(H,[],0) :- !, display(' {'), display(H), display('.}\n').
dis_rl(H,T,0) :- !, display(' {'), display(H),
                 display(' :- '), dis_gl(T), display('.}\n').
dis_rl(H,T,L) :- L1 is L-1, display('  '), dis_rl(H,T,L1).
```

Metainterpretation XIV (bis)

```
dis_deduc([],H) :- !, display(H), display('\n').
dis_deduc(T,H) :- dis_gl(T), display(' => '), display(H), display('\n').

norule(0) :- !, display(' {cannot unify.}\n').
norule(L) :- L1 is L-1,display('  '),norule(L1).

find_rl(H,L) :- rule(H1,T1), not(not(H1 = H)), !.
find_rl(H,L) :- norule(L).

solve([],L) :- !, display('Proof:\n').
solve([G|Gs],L) :- !, find_rl(G,L), rule(G,T), dis_rl(G,T,L),
                   L1 is L+1, append(T,Gs,Gs1), dis_gl(Gs1,L1),
                   solve(Gs1,L1), dis_deduc(T,G).
solve(G,L) :- solve([G],L).

metatrace(G) :- dis_gl(G,1), solve(G,1).
```

Metainterpretation XV

```
?- metatrace(grandparent(X, marie)).
  grandparent(_8,marie)?
   {grandparent(_8,marie) :- parent(_8,_64), parent(_64,marie).}
    parent(_8,_64), parent(_64,marie)?
     {parent(_8,_64) :- pere(_8,_64).}
      pere(_8,_64), parent(_64,marie)?
       {pere(eric,jeanne).}
        parent(jeanne,marie)?
         {parent(jeanne,marie) :- pere(jeanne,marie).}
          pere(jeanne,marie)?
           {cannot unify.}
         {parent(jeanne,marie) :- mere(jeanne,marie).}
          mere(jeanne,marie)?
           {cannot unify.}
```

Metainterpretation XV (bis)

```
        {pere(paul,jacques).}
         parent(jacques,marie)?
          {parent(jacques,marie) :- pere(jacques,marie).}
           pere(jacques,marie)?
            {cannot unify.}
          {parent(jacques,marie) :- mere(jacques,marie).}
           mere(jacques,marie)?
            {cannot unify.}
        {pere(paul,catherine).}
         parent(catherine,marie)?
          {parent(catherine,marie) :- pere(catherine,marie).}
           pere(catherine,marie)?
            {cannot unify.}
          {parent(catherine,marie) :- mere(catherine,marie).}
           mere(catherine,marie)?
            {mere(catherine,marie).}
             empty goal
...   ...    ...   ...   ...   ...
```

Metainterpretation XVI

```
Proof:
mere(catherine,marie)
mere(catherine,marie) => parent(catherine,marie)
pere(paul,catherine)
pere(paul,catherine) => parent(paul,catherine)
parent(paul,catherine), parent(catherine,marie)
                      => grandparent(paul,marie)
  X = paul ;
```

Metainterpretation XVI (bis)

```
{parent(_8,_64) :- mere(_8,_64).}
 mere(_8,_64), parent(_64,marie)?
  {mere(francoise,catherine).}
   parent(catherine,marie)?
    {parent(catherine,marie) :- pere(catherine,marie).}
     pere(catherine,marie)?
      {cannot unify.}
    {parent(catherine,marie) :- mere(catherine,marie).}
     mere(catherine,marie)?
      {mere(catherine,marie).}
       empty goal
```

Metainterpretation XVI (ter)

```
Proof:
mere(catherine,marie)
mere(catherine,marie) => parent(catherine,marie)
mere(francoise,catherine)
mere(francoise,catherine) => parent(francoise,catherine)
parent(francoise,catherine), parent(catherine,marie)
                           => grandparent(francoise,marie)
  X = francoise ;

...   ...   ...   ...   ...   ...
```

Metainterpretation XVII (negation)

The clauses

```
solve([not G|Gs],L) :- display('First solving '),display(G),
                       display('\n'),dis_gl(G,L),solve([G],L),!,
                       display('Successfully solved '),
                       display(G), display('\n'), fail.

solve([not G|Gs],L) :- !, display('Unsuccessfully solved '),
                       display(G), display(', so going on\n'),
                       L1 is L+1, dis_gl(Gs,L1), solve(Gs,L1),
                       dis_deduc([], not G).
```

are inserted *before* the clause

```
solve([G|Gs],L) :- !, find_rl(G,L), rule(G,T), dis_rl(G,T,L),
                   L1 is L+1, append(T,Gs,Gs1), dis_gl(Gs1,L1),
                   solve(Gs1,L1), dis_deduc(T,G).
```

Metainterpretation XVIII

```
rule(element(X,[X|Xs]), []).
rule(element(X,[Y|Xs]), [element(X,Xs)]).
rule(car(X,Xs,0), [not element(X,Xs)]).
rule(car(X,Xs,1), [element(X,Xs)]).

?- metatrace(car(3,[1,2,4],X)).
  car(3,[1,2,4],_10)?
   {car(3,[1,2,4],0) :- not(element(3,[1,2,4])).}
    not(element(3,[1,2,4]))?
```

Metainterpretation XVIII (bis)

```
First solving element(3,[1,2,4])
    element(3,[1,2,4])?
     {element(3,[1,2,4]) :- element(3,[2,4]).}
      element(3,[2,4])?
       {element(3,[2,4]) :- element(3,[4]).}
        element(3,[4])?
         {element(3,[4]) :- element(3,[]).}
          element(3,[])?
           {cannot unify.}
Unsuccessfully solved element(3,[1,2,4]), so going on
      empty goal
Proof:
not(element(3,[1,2,4]))
not(element(3,[1,2,4])) => car(3,[1,2,4],0)

  X = 0;
...   ...   ...   ...
```

Metainterpretation XIX

```
?- metatrace(car(3,[1,2,3],X)).
  car(3,[1,2,3],_10)?
   {car(3,[1,2,3],0) :- not(element(3,[1,2,3])).}
    not(element(3,[1,2,3]))?
First solving element(3,[1,2,3])
    element(3,[1,2,3])?
     {element(3,[1,2,3]) :- element(3,[2,3]).}
      element(3,[2,3])?
       {element(3,[2,3]) :- element(3,[3]).}
        element(3,[3])?
         {element(3,[3]).}
          empty goal
```

Metainterpretation XIX

```
Proof:
element(3,[3])
element(3,[3]) => element(3,[2,3])
element(3,[2,3]) => element(3,[1,2,3])
Successfully solved element(3,[1,2,3])
   {car(3,[1,2,3],1) :- element(3,[1,2,3]).}
    element(3,[1,2,3])?
     {element(3,[1,2,3]) :- element(3,[2,3]).}
      element(3,[2,3])?
       {element(3,[2,3]) :- element(3,[3]).}
        element(3,[3])?
         {element(3,[3]).}
          empty goal
Proof:
element(3,[3])
element(3,[3]) => element(3,[2,3])
element(3,[2,3]) => element(3,[1,2,3])
element(3,[1,2,3]) => car(3,[1,2,3],1)

  X = 1 ;
...   ...
```

Metainterpretation XX

Pruning lengthy branches.

`solve` and `metatrace` are replaced by

```
solve(G,L,0) :- !, display('Overflow !!\n'), fail.
solve([],L,DL) :- !, display('Proof:\n').
solve([G|Gs],L,DL) :- !, find_rl(G,L), rule(G,T),
     dis_rl(G,T,L), L1 is L + 1, DL1 is DL - 1,
     append(T,Gs,Gs1), dis_gl(Gs1,L1),
     solve(Gs1,L1,DL1), dis_deduc(T,G).
solve(G,L,DL) :- solve([G],L,DL).

metatrace(G,DL) :- dis_gl(G,1), solve(G,1,DL).
metatrace(G) :- dis_gl(G,1), solve(G,1,-1).
```

Metainterpretation XXI

```
?- metatrace(member2a(u,[[u]]),4).
  member2a(u,[[u]])?
   {member2a(u,[[u]]) :-
       member(u, _5),member(_5,[[u]]).}
    member(u,_5),member(_5,[[u]])?
     {member(u,[u|_6]).}
      member([u|_6],[[u]])?
       {member([u],[[u]]).}
        empty goal
Proof:
member([u],[[u]])
member(u,[u])
member(u,[u]),member([u],[[u]]) => member2a(u,[[u]])

Yes
```

Metainterpretation XXII

```
?- metatrace(member2a(u,[[v]]),4).
  member2a(u,[[v]])?
   {member2a(u,[[v]]) :-
       member(u,_7),member(_7,[[v]]).}
    member(u,_7),member(_7,[[v]])?
     {member(u,[u|_8]).}
      member([u|_8],[[v]])?
       {member([u|_8],[[v]]) :- member([u|_8],[]).}
        member([u|_8],[])?
         {cannot unify.}
     {member(u,[_9|_8]) :- member(u,_8).}
      member(u,_8),member([_9|_8],[[v]])?
       {member(u,[u|_6]).}
        member(.(_9,[u|_6],[[v]])?
         {member([_9,u|_6],[[v]]) :-
             member([_9,u|_6],[]).}
```

Metainterpretation XXII (bis)

```
          member([_9,u|_6],[])?
Overflow !!
       {member(u,[_5|_6]) :- member(u,_6).}
        member(u,_6),member([_9,_5|_6],[[v]])?
         {member(u,[u|_4]).}
          member([_9,_5,u|_4],[[v]])?
Overflow !!
         {member(u,[_3,_4]) :- member(u,_4).}
          member(u,_4),member([_9,_5,_3|_4],[[v]])?
Overflow !!

No
```

Expert system I Asking questions, using answers

```
said_true(true).
said_false(false).

known(G) :- said_true(G).
known(G) :- said_false(G).

ask(G) :- query(G,Ans), assert(G,Ans).

query(G,Ans) :- display('Is '), display(G),
                display(' true ? (y/n)\n'), read_ans(Ans).

assert(G,'y') :- assert(said_true(G)).
assert(G,'n') :- assert(said_false(G)).

read_ans(Ans) :- read(Ans), check_ans(Ans), !.
read_ans(Ans) :- read_ans(Ans).

check_ans(y).
check_ans(n).
```

Expert system II Metainterpretation (supplementary rules)

```
solve([G|Gs],L) :- askable(G), not known(G), ask(G), fail.

solve([G|Gs],L) :- askable(G), said_true(G), !,
                   disp_us(G,yes,L), L1 is L+1, dis_gl(Gs,L1),
                   solve(Gs,L1), dis_deduc([],G).

solve([G|Gs],L) :- askable(G), said_false(G), !, disp_us(G,no,L), fail.


disp_us(G,yes,0) :- !, display(' {user says that '),
                    display(G), display(' is true.}\n').

disp_us(G,no,0) :- !, display(' {user says that '),
                   display(G), display(' is false.}\n').

disp_us(G,Ans,L) :- L1 is L-1, display('  '), disp_us(G,Ans,L1).
```

Expert system III Culinary application

```
rule(place(D,top), [pastry(D),size(D,small)]).
rule(place(D,mid), [pastry(D),size(D,big)]).
rule(place(D,mid), [main(D)]).
rule(place(D,low), [slow(D)]).
rule(pastry(D), [type(D,cake)]).
rule(pastry(D), [type(D,bread)]).
rule(main(D), [type(D,meat)]).
rule(slow(D), [type(D,pudding)]).

askable(type(D,T)) :- !.
askable(size(D,S)) :- !.
```

Expert system IV Questions and answers; deductions

```
?- metatrace(place(dish1,W)).
  place(dish1, _G244)?
   {place(dish1, top) :- pastry(dish1), size(dish1, small).}
    pastry(dish1),size(dish1, small)?
     {pastry(dish1) :- type(dish1, cake).}
      type(dish1, cake),size(dish1, small)?
Is type(dish1, cake) true ? (y/n)
|: y.
       {user says that type(dish1, cake) is true.}
        size(dish1, small)?
Is size(dish1, small) true ? (y/n)
|: n.
         {user says that size(dish1, small) is false.}
     {pastry(dish1) :- type(dish1, bread).}
      type(dish1, bread),size(dish1, small)?
Is type(dish1, bread) true ? (y/n)
|: n.
       {user says that type(dish1, bread) is false.}
```

Expert system V

```
   {place(dish1, mid) :- pastry(dish1), size(dish1, big).}
    pastry(dish1),size(dish1, big)?
     {pastry(dish1) :- type(dish1, cake).}
      type(dish1, cake),size(dish1, big)?
       {user says that type(dish1, cake) is true.}
        size(dish1, big)?
Is size(dish1, big) true ? (y/n)
|: y.
         {user says that size(dish1, big) is true.}
          empty goal
Proof:
size(dish1, big)
type(dish1, cake)
type(dish1, cake) => pastry(dish1)
pastry(dish1),size(dish1, big) => place(dish1, mid)

W = mid ;
...   ...   ...   ...
```

Expert system VI

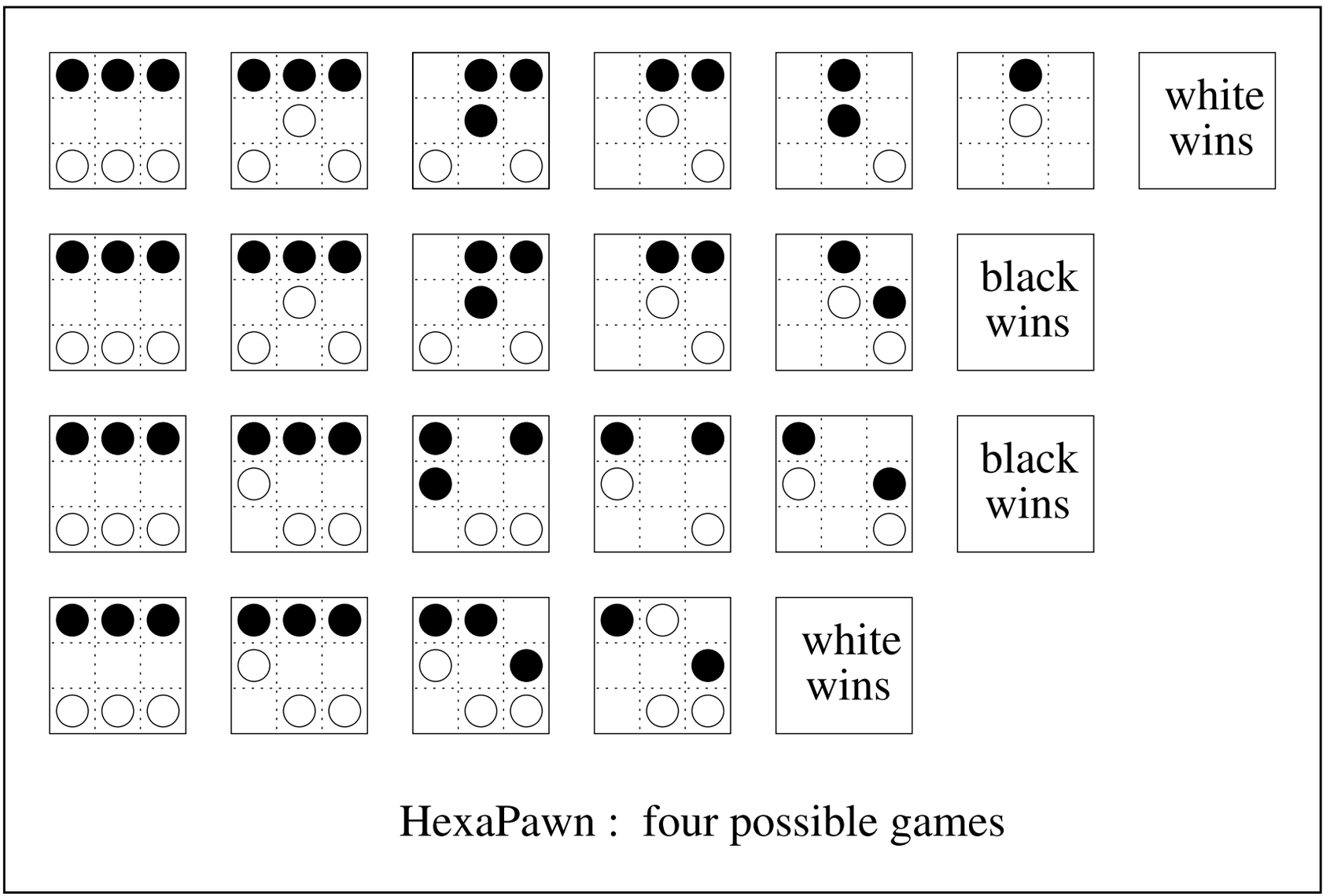
All solutions

```
...   ...   ...   ...
   {place(dish1,mid) :- pastry(dish1), size(dish1,big).}
    pastry(dish1), size(dish1,big)?
     {pastry(dish1) :- type(dish1,cake).}
      type(dish1,cake), size(dish1,big)?
       {user says that type(dish1,cake) is true.}
        size(dish1,big)?
Is size(dish1,big) true ? (y/n)
|: y.
         {user says that size(dish1,big) is true.}
          empty goal
```

Expert system VII

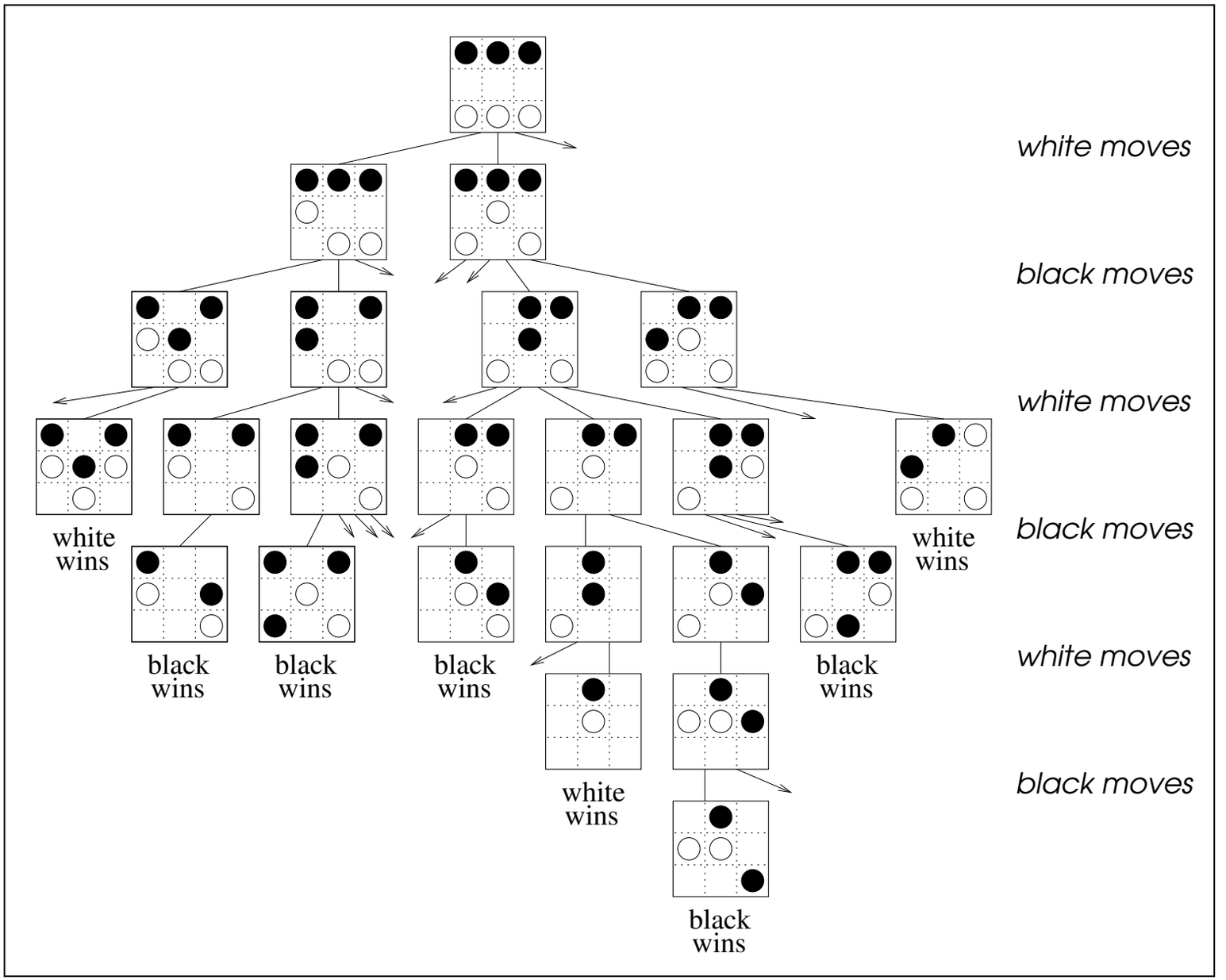
```
Proof:
size(dish1,big)
type(dish1,cake)
type(dish1,cake) => pastry(dish1)
pastry(dish1), size(dish1,big) => place(dish1,mid)
  W = mid ;
     {pastry(dish1) :- type(dish1,bread).}
      type(dish1,bread), size(dish1,big)?
       {user says that type(dish1,bread) is false.}
   {place(dish1,mid) :- main(dish1).}
    main(dish1)?
     {cannot unify.}
   {place(dish1,low) :- slow(dish1).}
    slow(dish1)?
     {cannot unify.}
No
```

Learning: hexapawn I



HexaPawn : four possible games

Learning: hexapawn II



Learning: hexapawn III

Learning means buiding/exploring/pruning the game tree.

A lost game is never played twice.

The computer "learns" only by losing.

The computer becomes "perfect" after losing a fixed number of games.

If a winning strategy exists, the computer will find it.

If no such strategy exists, the computer will at the end refuse to play!

Learning: hexapawn IV

The learning technique is theoretically perfect.

It is faster if the opponent is good.

Learning becomes too slow for realistic games (including Chess).

Learning: hexapawn V

In the specific case of Hexapawn, a winning strategy exists for the Black.

If the computer plays White, after the learning phase is completed
(a fixed number of games must be lost), the computer will stop playing.

All games lost by the computer are distinct.

Learning: hexapawn VI

```
init([p(black,1,1), p(black,2,1), p(black,3,1),
      p(white,1,3), p(white,2,3), p(white,3,3)]).

hexa :- init(S), not bad(S), learn_maybe(S), play(white, S).
hexa :- init(S), bad(S), nl, write('I cannot win.'), nl.
```

The cut can be used here:

```
% Learning games
hexa :- init(S), not bad(S), !, learn_maybe(S), play(white, S).

% Learning has ended, no winning strategy exists.
hexa :- nl, write('I cannot win.'), nl.
```

Variable `S`: configuration
Predicate `bad`: for known losing configurations
Predicate `learn_maybe`: for unknown losing configurations
(Dynamic predicates)

Learning: hexapawn VII

The predicate `play` generates moves

```
% end of the game
play(C, S) :- chessboard(S),
              ch_color(C, C1), win(C1, S),
              !, write('The winner is : '), write(C1),
              nl, winner(C1), abort.

% Black (human) moves
play(black, S) :- human(black, S, S_aft), play(white, S_aft).

% White (computer) moves
play(white, S) :- computer(white, S, S_aft), play(black, S_aft).
```

Learning: hexapawn VIII

```
win(C, S) :- ch_color(C, C1), not legal(C1, M, S).
win(black, S) :- member(p(black,X,3), S).
win(white, S) :- member(p(white,X,1), S).

human(C,S1,S2) :- learn_maybe(S1), ask_human(C,M,S1),
                  move(M,S1,S2).

computer(C,S1,S2) :- possibility(C, S1, Ms), !,
                     choose_move(S1, Ms, M), move(M, S1, S2).
computer(C, S1, S2) :- learn(C, S1), end_msg, abort.

winner(black) :- !, maybe(S), forget_maybe(S), learn_bad(S).
winner(white).

end_msg :- nl, write('I know that you are the winner').
```

Predicate `win` detect the end of the game. Predicate `winner` allows the transition of some configuration from "maybe" to "bad".

Learning: hexapawn IX

A list of possible moves is maintained; one-move wins are detected.

```
choose_move(S, [M|Ms], M1) :- choose_win(S, [M|Ms], M1), !.
choose_move(S, [M|Ms], M).

choose_win(S,[M|Ms],M) :- move(M,S,S1), win(white,S1), !.
choose_win(S,[M|Ms],M1) :- choose_win(S,Ms,M1).

possibility(C, S, [M1|M1s]) :-
        findall(M, legal(C, M, S), Ms),
        sub_bad(S, Ms, [M1|M1s]), maybe(S1), forget_maybe(S1).
```

Learning: hexapawn X

```
legal(black, m(p(black,X,Y), p(black,X,Y1), forward), S)
   :- member(p(black,X,Y), S), incr(Y, Y1), free(X, Y1, S).
legal(white, m(p(white,X,Y), p(white,X,Y1), forward), S)
   :- member(p(white,X,Y), S), decr(Y, Y1), free(X, Y1, S).
legal(C, m(p(C,X,Y), p(C,X1,Y1), eat), S)
   :- member(p(C,X,Y), S), diag(C, X, Y, X1, Y1),
      ch_color(C, C1), member(p(C1,X1,Y1), S).

move(m(P1, P2, forward), S_bef, S_aft)
   :- sub(P1, P2, S_bef, S), sort(S, S_aft).
move(m(P1, P2, eat), S_bef, S_aft)
   :- P2 = p(C2,X,Y), ch_color(C2, C),
      remove(p(C,X,Y), S_bef, S1), sub(P1, P2, S1, S), sort(S, S_aft).
```

Learning: hexapawn XI

```
is_range(X)  :- X>0, X=<3.

incr(X, X1) :- X<3, X1 is X+1.
decr(X, X1) :- X>1, X1 is X-1.

diag(black, X, Y, X1, Y1) :- incr(Y, Y1), incr(X, X1).
diag(black, X, Y, X1, Y1) :- incr(Y, Y1), decr(X, X1).
diag(white, X, Y, X1, Y1) :- decr(Y, Y1), incr(X, X1).
diag(white, X, Y, X1, Y1) :- decr(Y, Y1), decr(X, X1).

ch_color(white, black).
ch_color(black, white).

free(X, Y, S) :- not member(p(C,X,Y), S).
```

Learning: hexapawn XII

```
ask_human(C, m(P1,P2,Ty), S) :- can_move_it(C,P1,S), where(m(P1,P2,Ty),S).

read_x(X) :- write('X: '), read(X), is_range(X), !.
read_x(X) :- read_x(X).
read_y(Y) :- write('Y: '), read(Y), is_range(Y), !.
read_y(Y) :- read_y(Y).

coord(X, Y) :- write('Enter the coordinate: '), nl, read_x(X), read_y(Y).

which_pawn(C, p(C,X,Y), S) :-
   nl, write('Which '), write(C), write(' pawn do you want to move ?'),
   nl, coord(X, Y), member(p(C,X,Y), S), !.
which_pawn(C, P, S) :-
   write('There is not a '), write(C), write(' pawn there !'),
   nl, which_pawn(C, P, S).
```

Learning: hexapawn XIII

```
can_move_it(C, P, S) :-
   which_pawn(C, P, S), legal(C, m(P, P1, Type), S), !.

can_move_it(C, P, S) :-
   write('There is not a legal move for it!'),
   nl, can_move_it(C, P, S).


where(m(p(C,X1,Y1), p(C,X2,Y2), Type), S) :-
   nl, write('Where do you want to move it ?'),
   nl, coord(X2, Y2),
   legal(C, m(p(C,X1,Y1), p(C,X2,Y2), Type), S), !.

where(M, S) :- write('This is not a legal move !'), nl, where(M, S).
```

Learning: hexapawn XIV

Configuration lists management

```
show :- findall(S, bad(S), Ss), show_sit(Ss).

learn(C, S) :- findall(M, legal(C, M, S), Ms),
               forget(S, Ms), maybe(S1),
               forget_maybe(S1), learn_bad(S1).

forget(S, [M|Ms]) :- move(M, S, S1), forget_bad(S1), forget(S, Ms).
forget(S, []).


sub_bad(S1, [M|M1s], M2s) :- move(M, S1, S2), bad(S2), !,
                             sub_bad(S1, M1s, M2s).
sub_bad(S1, [M|M1s], [M|M2s]) :- sub_bad(S1, M1s, M2s).
sub_bad(S1, [], []).
```

Learning: hexapawn XV

```
show_sit([[P|Ps]|Ss]) :- chessboard([P|Ps]), show_sit(Ss).
show_sit([[]|Ss]) :- show_sit(Ss).
show_sit([]).

learn_bad([P|Ps]) :- asserta((bad([P|Ps]))).
learn_bad([]).

forget_bad([P|Ps]) :- retract((bad([P|Ps]))).
forget_bad([]).

learn_maybe([P|Ps]) :- asserta((maybe([P|Ps]))).
learn_maybe([]).

forget_maybe([P|Ps]) :- retract((maybe([P|Ps]))).
forget_maybe([]).

bad([]).

maybe([]).
```

Learning: hexapawn XVI

```
chessboard(S) :- nl, nl, draw_coord_col, draw_border,
                 draw_board(1, S), draw_border, nl.

draw_board(Y,S) :- draw_line(Y,S), incr(Y,Y1), !, draw_board(Y1,S).
draw_board(Y,S).

draw_line(Y,S) :- write(' '), write(Y), write(' |'),
                  draw_line(1,Y,S), write('|'),nl.

draw_line(X,Y,S) :- draw_cell(X,Y,S), incr(X,X1),!, draw_line(X1,Y,S).
draw_line(X,Y,S).

draw_cell(X,Y,S) :- member(p(C,X,Y), S), !, draw_pawn(C).
draw_cell(X,Y,S) :- write(' ').

draw_pawn(white) :- write('W').
draw_pawn(black) :- write('B').
```

Learning: hexapawn XVII

```
draw_coord_col :- Xd is 3//10, write('    '),
                  draw_ten(1, Xd), nl,
                  write('    '), draw_num(3), nl.

draw_ten(N,M) :- N=<M, !, write('         '), write(N),
                 N1 is N+1, draw_ten(N1,M).
draw_ten(N, M).

draw_num(N) :- N>9, number(9,Xs), draw(Xs),
               write('0'), N1 is N-10, draw_num(N1).
draw_num(N) :- N=<9, N>0, number(N, Xs), draw(Xs).
draw_num(0).

draw_border :- make_list(3,'-',Bord), write('   +'),
               draw(Bord), write('+'), nl.
```

Learning: hexapawn XVIII

```
remove(X, [X|Xs], Xs).
remove(X, [Y|Ys], [Y|Zs]) :- X\==Y, remove(X, Ys, Zs).

sub(X, Y, [X|Xs], [Y|Xs]).
sub(X, Y, [Z|Xs], [Z|Ys]) :- sub(X, Y, Xs, Ys).

make_list(N, X, [X|Xs]) :- N>0, N1 is N-1, make_list(N1, X, Xs).
make_list(0, X, []).

draw([X|Xs]) :- write(X), draw(Xs).
draw([]).

number(N, Xs) :- number(1, N, Xs).
number(N, N, [N]).
number(S, N, [S|Xs]) :- S<N, S1 is S+1, number(S1, N, Xs).

before(p(black,X1,Y1), p(white,X2,Y2)) :- !.
before(p(C,X1,Y1), p(C,X2,Y2)) :- X1 < X2, !.
before(p(C,X1,Y1), p(C,X2,Y2)) :- Y1 < Y2.
```

Learning: hexapawn XIX

```
?- hexa.

    123      123
   +---+    +---+
 1 |BBB|  1 |BBB|
 2 |   |  2 |W  |
 3 |WWW|  3 | WW|
   +---+    +---+

Which black pawn do you want to move ?
Enter the coordinate:
X: 2.
Y: 1.
```

Learning: hexapawn XX

```
Where do you want to move it ?
Enter the coordinate:
X: 1.
Y: 2.

    123     123
   +---+   +---+
 1 |B B| 1 |B B|
 2 |B  | 2 |BW |
 3 | WW| 3 |  W|
   +---+   +---+
```

Learning: hexapawn XXI

```
    123      123
   +---+    +---+
 1 |B B|  1 |B B|
 2 |BW |  2 | W |
 3 |  W|  3 |B W|
   +---+    +---+

The winner is : black

?- show.

    123
   +---+
 1 |B B|
 2 |BW |
 3 |  W|
   +---+
```

Learning: hexapawn XXII

```
   123      123      123      123      123
  +---+    +---+    +---+    +---+    +---+
1 |BBB| 1  |BBB| 1  |B B| 1  |B B| 1  |B B|
2 |   | 2  |W  | 2  |B  | 2  |B W| 2  |  W|
3 |WWW| 3  | WW| 3  | WW| 3  | W | 3  |BW |
  +---+    +---+    +---+    +---+    +---+
```

```
The winner is : black
```

Second lost game, two bad configurations

```
   123     123
  +---+    +---+
1 |B B| 1  |B B|
2 |B W| 2  |BW |
3 | W | 3  |  W|
  +---+    +---+
```

Learning: hexapawn XXIII

After third lost game

```
   123      123      123
  +---+    +---+    +---+
1 |B B| 1 |B B| 1 |B B|
2 |W  | 2 |B W| 2 |BW |
3 |  W| 3 | W | 3 |  W|
  +---+    +---+    +---+
```

Fourth game

```
   123      123      123
  +---+    +---+    +---+
1 |BBB| 1 |BBB| 1 |B B|
2 |   | 2 |W  | 2 |B  |
3 |WWW| 3 | WW| 3 | WW|
  +---+    +---+    +---+
```

```
I know that you are the winner
```

Learning: hexapawn XXIV

The computer resigns since all three possible moves lead to a forbidden configuration:

- Take advanced Black with middle White;
- Advance lateral White;
- Advance middle White.

The system now knows that lateral attack is bad.

```
?- show.

    123
   +---+
 1 |BBB|
 2 |W  |
 3 | WW|
   +---+
```

Learning: hexapawn XXV

Next game, central attack

```
?- hexa.

    123      123      123      123      123
   +---+    +---+    +---+    +---+    +---+
 1 |BBB| 1  |BBB| 1  | BB| 1  | BB| 1  | BB|
 2 |   | 2  | W | 2  | B | 2  |WB | 2  |W  |
 3 |WWW| 3  |W W| 3  |W W| 3  |  W| 3  | BW|
   +---+    +---+    +---+    +---+    +---+

The winner is : black
```

Learning: hexapawn XXVI

After losing some more games, the bad configurations are

```
   123      123      123
  +---+    +---+    +---+
1 |BBB|  1 |BBB|  1 |BBB|
2 |W  |  2 | W |  2 |  W|
3 | WW|  3 |W W|  3 |WW |
  +---+    +---+    +---+
```

so the next game will be

```
   123
  +---+
1 |BBB|
2 |   |
3 |WWW|
  +---+
```

```
I know that you are the winner
```

Learning: hexapawn XXVII

And now the computer refuses to play

```
   123
  +---+
1 |BBB|
2 |   |
3 |WWW|
  +---+

I cannot win.
```