**Converting from logic to prolog**

"and" transforms into separate statements

```
c(Y) :- h(Y), w. becomes c(Y) --> h(Y) && w
If statement prolog: (condition -> true ; false)
```

```
Ax.(r(x) && c(x) -> s(x)) && !Ax.!s(x)
= Ax.(r(x) && c(x) -> s(x)) && Ex.s(x)
= Ax.(r(x) && c(x) -> s(x)) && Ey.s(y)
= Ax.Ey.(!r(x) || !c(x) || s(x)) && s(y)

!r(x) || !c(x)  || s(x)
s(y)

!(r(x) && c(x)) || s(x)
s(y)

r(x) && c(x) -> s(x)
s(y)

s(x) :- r(x), c(x).
s(y).

Ax. (a && b && c -> r(x))
= r(x) :- a, b, c.

a || b || c || !d || !e || !f
= a || b || c || !(d && e && f)
= (a || b || c) || !(d && e && f)
= d && e && f -> a || b || c
= a :- d, e ,f.
  b :- d, e, f.
  c :- d, e, f.
```

If there isn't exactly one positive term in a clause, it can not be translated into prolog

$$P_1 \Rightarrow P_2 \equiv \neg P_1 \vee P_2$$

$$\neg \exists X\ [P(X)] \equiv \forall X\ [\neg P(X)]$$
$$\neg \forall X\ [P(X)] \equiv \exists X\ [\neg P(X)]$$

$$\neg(P_1 \wedge P_2) \equiv \neg P_1 \vee \neg P_2$$
$$\neg(P_1 \vee P_2) \equiv \neg P_1 \wedge \neg P_2$$
$$\neg\neg P \equiv P$$

$$(P_1 \Leftrightarrow P_2) \equiv (P_1 \Rightarrow P_2) \wedge (P_2 \Rightarrow P_1)$$

$$P_1 \vee (P_2 \wedge P_3) \equiv (P_1 \vee P_2) \wedge (P_1 \vee P_3)$$
$$P_1 \wedge (P_2 \vee P_3) \equiv (P_1 \wedge P_2) \vee (P_1 \wedge P_3)$$

$$P_1 \vee P_2 \equiv P_2 \vee P_1$$

## Clocksin and Melish Procedure

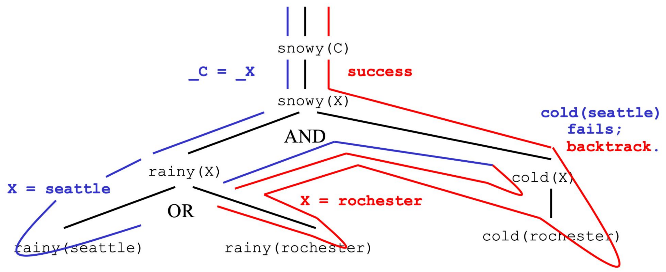
1. Eliminate implication (⇒) and equivalence (⇔).
2. Move negation (¬) inwards to individual terms.
3. *Skolemization*: eliminate existential quantifiers (∃).
4. Move universal quantifiers (∀) to top-level and make implicit , i.e., all variables are universally quantified.
5. Use distributive, associative and commutative rules of ∨, ∧, and ¬, to move into *conjuctive normal form*, i.e., a conjuction of disjunctions (or *clauses*.)

$$C \Leftarrow A,B$$
$$E \Leftarrow C,D$$
$$\overline{\phantom{xxxxxxxxxx}}$$
$$E \Leftarrow A,B,D$$

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

```
male(carlos).
parent(carlos, tatiana).
parent(carlos, catalina).
father(X,Y) :- parent(X,Y), male(X).
-----------------------------------------------
father(carlos, tatiana).
father(carlos, catalina).
```

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X).
```




| | |
|---|---|
| call(P) | Invoke predicate as a goal. |
| assert(P) | Adds predicate to database. |
| retract(P) | Removes predicate from database. |
| functor(T,F,A) | Succeeds if T is a *term* with *functor* F and *arity* A. |
| findall(F,P,L) | Returns a list L with elements F satisfying predicate P |

The meaning of `not(snowy(X))` is:

$$\neg\exists X\ [snowy(X)]$$

rather than:

$$\exists X\ [\neg snowy(X)]$$

```
% General Prolog: Define sequence(X, Y),
% which is true if X is a subsequence of Y.
sequence([], Y).
sequence([X | Xs], [X | Ys]) :- sequence(Xs, Ys).
sequence(Xs, [_ | Ys]) :- sequence(Xs, Ys).
```

---

AX.((p(X) || q(X)) -> m(X)) && AX.EY.(!r(X,Y))
AX.(!(p(X) || q(X)) || m(X)) && AZ.EY.(!r(Z,Y))
AX.AZ.EY.(!(p(X) || q(X)) || m(X)) && !r(Z,Y)
(!(p(X) || q(X)) || m(X)) && !r(Z,Y)
((!p(X) && !q(X)) || m(X)) && !r(Z,Y)
(m(X) || !p(X)) && (m(X) || !q(X)) && !r(Z,Y)

this is not doable in prolog as the !r(Z, Y) clause does not have a single positive and can not be changed

---

% Givens:
p(a).
p(b).
p(c).
p(d).
p(e).
q(f).
q(b).
q(d).
r(b).
r(a).

% Functions:
u(X) :- p(X), q(X), r(X).   % b
v(X) :- p(X), !, q(X), r(X).  % false because the first thing that satisfies p (AKA a) does not satisfy q && r
w(X) :- p(X), q(X), !, r(X).  % b

---

% General Prolog 2: Define intermediate(B), which is true if there is a block which is directly above and below it. Locations are statements in the form location(id, [x, y]).

```
intermediate(B) :-
        location(B, [X, Y]),
        location(_, [X, Y + 1]),
        location(_, [X, Y - 1]).
```

---

% Constraint Search and Propagate: Define atm(M, D20, D10, D5, D1). The variables D20, D10, ... are denominations of money, respective to their variable name. M is total money. atm is true if the denominations add up to M, and there is no more than 10 of each denomination.

```
:- use_module(library(clpfd)).
atm_(M, D20, D10, D5, D1) :-
  D20 in 0..10,
  D10 in 0..10,
  D5 in 0..10,
  D1 in 0..10,
  M #= D20 * 20 + D10 * 10 + D5 * 5 + D1 * 1.

atm(M, D20, D10, D5, D1) :-
       atm_(M, D20, D10, D5, D1),
       label([M, D20, D10, D5, D1]).
```

```
append3([], [], [], M, L) :- L = M.

append3([], [], [X | Rest], M, L) :-
       append(M, [X], M2),
       append3([], [], Rest, M2, L).

append3([], [X | Rest], L3, M, L) :-
       append(M, [X], M2),
       append3([], Rest, L3, M2, L).

append3([X | Rest], L2, L3, M, L) :-
       append(M, [X], M2),
       append3(Rest, L2, L3, M2, L).

append3(L1, L2, L3, L) :-
       append3(L1, L2, L3, M, L).
```

---

Write a minimum(L,M) predicate in Prolog that finds the minimum value M in a list of numbers L only using arithmetic operators (e.g., <, >,...) and the minimum predicate itself recursively

```
% Part a
minimum([], _) :-
write('nyuuuuuuuuuuuuuuuuuuuuuuuuuu'), false.
minimum([X], X).

minimum([Head | Tail], Min) :-
       minimum(Tail, TailMin),
       (Head < TailMin -> Min = Head ; Min =
                                           TailMin).
```

---

Write a Prolog program to find the highest block on a table, where block locations are represented using the location(L,[X,Y])

```
highest(HighestBlocks) :-
       findall(Block, (location(Block, [_, Y]), \+
(location(OtherBlock, [_, OtherY]), OtherY > Y)),
HighestBlocks).
```

---

(5 points) Which of the following is a Horn clause?

- [x] $p \vee \neg q$
- [ ] $p \vee q \vee \neg r$
- [ ] $p \vee q \vee r \vee \neg s$
- [ ] All of the above.
- [ ] None of the above.

Justify your answer.

Exactly one non-negated term (positive)

f(X, y) and f(Y, X) unify with each other because X binds to Y and Y bounds to X

Prolog's "closed-world" assumptions means its knowledge base is assumed to contain **everything that is true**

maplist(writeln, [1,2,3,4]

sort(List, SortedList)

**Part A:**
Write a constraint propagation program in Oz or Prolog to solve the following equations assuming that X and Y variables can only take integer values in the range of [0..100]:

```
A1 * X + B1 * Y = C1
A2 * X + B2 * Y = C2
```

A sample Prolog interaction using a predicate solve(A1,B1,C1,A2,B2,C2,X,Y):

```
solve(A1, B1, C1, A2, B2, C2, X, Y) :-
       between(0, 100, X),
       between(0, 100, Y),
       Eq1 is A1 * X + B1 * Y,
       Eq2 is A2 * X + B2 * Y,
       Eq1 =:= C1,
       Eq2 =:= C2.
```

**Part B:**
Generalize your program to a system of N linear equations, specified as $M * V = C$ where M is an N × N matrix of the coefficients, V is the vector of variables to solve for, and C is the vector of constants resulting from the matrix-vector multiplication. For example, the system of two equations in (a) would be represented

$$\begin{bmatrix}1 & 1\\2 & -1\end{bmatrix}\begin{bmatrix}X\\Y\end{bmatrix}=\begin{bmatrix}10\\11\end{bmatrix}$$

Create a constraint program to solve the system of N equations, solveN(M,V,C). Assume that all variables in V can only take integer values in the range of [0..100]. A sample Prolog interaction with the matrix M represented as a list of rows follows:

```
:- use_module(library(clpfd)).

% Matrix-vector multiplication
mat_vec_mult([], _, []).
mat_vec_mult([Row|Matrix], V, [Result|Results]) :-
       dot_product(Row, V, Result),
       mat_vec_mult(Matrix, V, Results).

dot_product([], [], 0).
dot_product([X|Xs], [Y|Ys], DotProduct) :-
       dot_product(Xs, Ys, Rest),
       DotProduct #= X * Y + Rest.

% Solve the system of linear equations
solveN(Matrix, V, Constants) :-
       V ins 0..100,
       mat_vec_mult(Matrix, V, Constants),
        % TransposedMatrix * V #= Constants,
       label(V).2

% Example usage:
% solveN([[1,1],[2,-1]],[X,Y],[10,11]).
% solveN([[2]],[X],[10]).
%
solveN([[1,1,1],[2,3,-1],[1,-1,2]],[X,Y,Z],[9,6,8]).
```

```
include(inHouse(House), Items, HouseItems),
findall([SubPrice, SubVolume, Subset], (
        subsetz(HouseItems, Subset),
        calculate_totals(Subset, SubPrice,
                                 SubVolume),
        forall(member(Constraint,
             HouseConstraints), (
                    testConstraint(SubPrice,
                          SubVolume, Constraint)
        ))
), ValidSubsets),
```

---

```
append([a,b], [c], X).
X = [a,b,c].
```

---

```
% For some `s` and `t`, checks if `s` is a
subset of `t`.
subsetz([], []).
subsetz([H | T], [H | T1]) :- subsetz(T, T1).
subsetz([_ | T], T1) :- subsetz(T, T1).
```

---

```
atom_number(CubicFeet, NCubicFeet),
```