

Meta-interpretation

Artificial Intelligence Programming in Prolog

Lecturer: Tim Smith

Lecture 17

25/11/04

Contents

- Controlling the flow of computation
 - Representing logical relationships
 - conjunctions ($P \wedge Q$)
 - disjunctions ($P \vee Q$)
 - conjunctive not $\neg (P \wedge Q)$.
 - if.....then....else.....
- Meta-Interpreters
 - clause/2
 - left-to-right interpreter
 - right-to-left interpreter
 - breadth-first
 - best-first
 - others

Controlling the flow of computation

- Prolog has many built-in predicates and operators that can be used to control how queries are proved.
- First, I will introduce a set of functions that can be used within normal Prolog programs then I will show how these ideas can be used to create Meta-Interpreters.

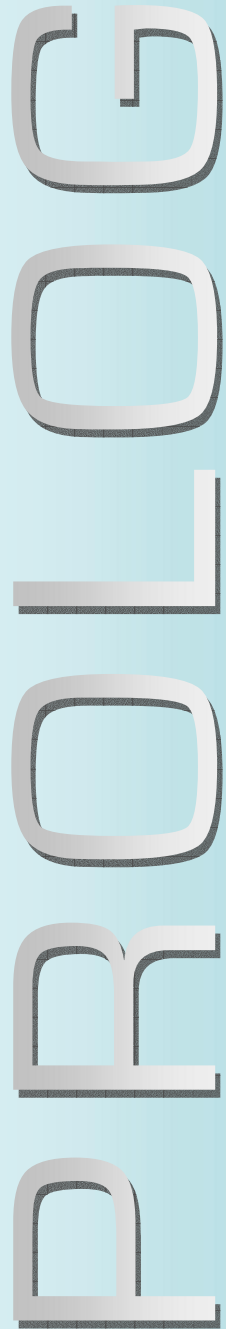
- The main predicate of this type is `call/1`.
- This takes one argument in the form of a goal (i.e. a single term) and checks whether the goal succeeds.

```
|?- call(write('Hello')).
```

```
Hello?
```

```
yes
```

- Mostly used to call goals constructed using `=..`, `functor/3` and `arg/3`.



A Conjunction of Goals

- A conjunction of goals ($P \wedge Q$) can be called by collecting the goals together in round brackets.

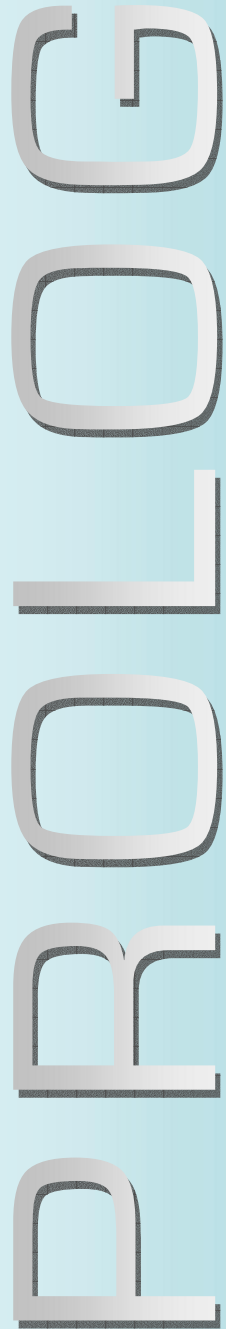
```
| ?- X = ( Y=[a,b,f,g] , member(f,Y) ) , call(X) .
```

```
X = [a,b,f,g]=[a,b,f,g] , member(f,[a,b,f,g]) ,
```

```
Y = [a,b,f,g] ?
```

yes

- The two goals $Y=[a,b,f,g]$ and $\text{member}(f,Y)$ are conjoined as one term and instantiated with X .
- $\text{call}(X)$ then calls them in order and will only succeed if all the goals contained within X succeed (hence, it is checking if the conjunction of the two goals is true).



A Conjunction of Goals (2)

- The actual job of conjoining goals is performed by the ‘,’ operator. (‘,’ = the logical \wedge)
`?- (3,4) = ', '(3,4) .`
yes
- This is a right-associative operator:
 - You can see this using `?- current_op(1000, xfy, ', ')`.
= When used in a series of operators with the same precedence the comma associates with a single term to the left and groups the rest of the operators and arguments to the right.
 - *(works in a similar way to Head a Tail list notation).
`| ?- (3,4,5,6,7,8) = (3, (4, (5, (6, (7,8)))) .`
yes
`| ?- (3,4,5,6,7,8) = (((((3,4),5),6),7),8) .`
no

A Conjunction of Goals (2)

- Because of this associativity, groups of conjoined goals can be stripped apart by making them equal to (FirstGoal,OtherGoals).
 - `FirstGoal` is a single Prolog goal
 - `OtherGoals` may be a single goal or another pair consisting of another goal and remaining goals (grouped around ',').

```
| ?- (3,4,5,6,7,8) = (H,T) .
```

```
H = 3,
```

```
T = 4,5,6,7,8 ? ;
```

```
no
```

- This allows us to recursively manipulate sequences of goals just as we previously manipulated lists.

```
| ?- (3,4,5,6,7,8) = (A,B), B = (C,D), D = (E,F), .....
```

```
A = 3,          B = 4,5,6,7,8,
```

```
C = 4,          D = 5,6,7,8,
```

```
E = 5,          F = 6,7,8, .....
```

Repeated use of same test = recursion

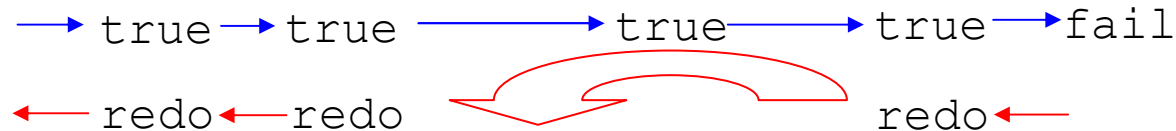
Why use call?

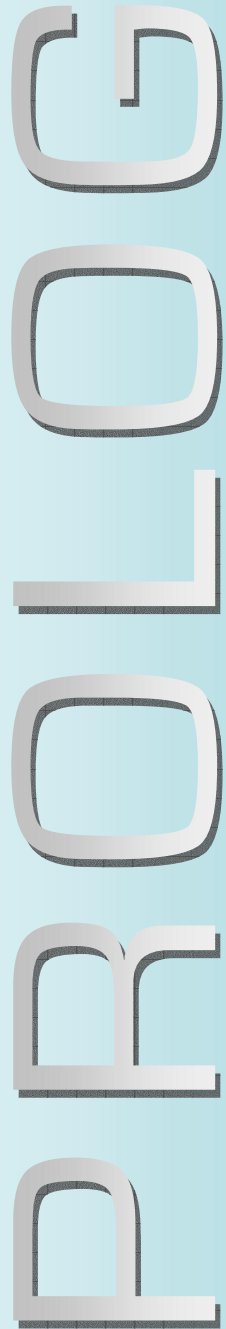
- But, why would we use call(X) as it seems to have the same function as just placing the variable X as a goal in your code:

e.g. `X = (Y=[a,b,f,g], member(f,Y)), call(X).`
`X = (Y=[a,b,f,g], member(f,Y)), X.`

- The main reason is because it keeps the solution of X isolated from the rest of the program within which call(X) resides.
 - Specifically, any cuts (!) within the conjoined set of goals X only stop backtracking within X.
 - It does not stop backtracking outside of call(X).

`|?- goal1, goal2, call((goal3, !, goal4, goal5)).`





A Disjunction of Goals (;)

- As well as ',' = the logical AND (\wedge)
- We also have an operator that represents the logical OR (\vee).
 - **Goal1 ; Goal2** = A disjunction of Goal1 and Goal2.
 - This will succeed if **either** Goal1 or Goal2 are true.
- | ?- 5<4;3<4.
yes
- Semicolon is an operator (`current_op(1100, xfy, ;)`) so it can be used in prefix position as well:
 - | ?- ;(5<4, 3<4).
yes
- This operator is right associative like ',':
 - | ?- (3;4;5;6;7;8) = (A;B), B = (C;D), D = (E;F),
 - A = 3, B = 4;5;6;7;8,
 - C = 4, D = 5;6;7;8,
 - E = 5, F = 6;7;8,

Conjoining Disjunctions

- However, OR has a higher precedence value than AND, so AND always groups first:
 - `current_op(1100, xfy, ;)`.
 - `current_op(1000, xfy, ',')`.
 - A sequence such as `(b,c,d,e ; f)`.
 - Is equivalent to `(b,(c,(d,e))) ; f`.
 - not `((b,c),d), (e;f)`.
- This is important when you are using `;` in rules :
`a:- b,c,d,e ; f.`
- Says that *“a is true if b, c, d, AND e are true OR f is true”*.
- In other words it can be written as:
`a:- b,c,d,e.`
`a:- f.`

Conjoining Disjunctions (2)

- A predicate definition with multiple clauses is preferred over the use of ; as it makes the definition easier to read.
- However, ; can be useful when the two definitions share a large number of preconditions but differ by a small number of final goals:
 - e.g. $a :- b, c, d, e.$
 $a :- b, c, d, f.$
- It is inefficient to test b, c, and d again so instead you could write one rule that just tested e OR f:
 - e.g. $a :- b, c, d, (e;f) .$
- The brackets impose your grouping preference on the structure.
- This can be read as: *“a is true is b, c, d, AND e OR f are true.”*

Conjoining Disjunctions (3)

- But, remember all OR constructions can always be replaced by using an auxiliary predicate with multiple clauses.
 - `a :- b, c, d, (e; f) .`
- Can be re-defined as:
 - `a :- b, c, d, aux .`
 - `aux :- e .`
 - `aux :- f .`
- Please be aware that whenever you are writing Prolog you are already representing logical relationships:
 - A Body of a clause full of goals separated by ‘,’ is a *conjunction*.
 - Defining a predicate with multiple clauses represents a *disjunction* of the conditions by which that predicate can be proven true.
- You should always use these innate logical structures before using extra operators (such as ;).

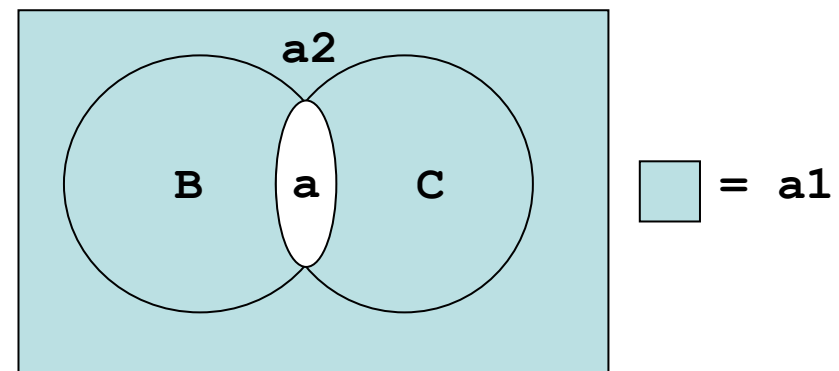
Creating a Conjoined 'not'

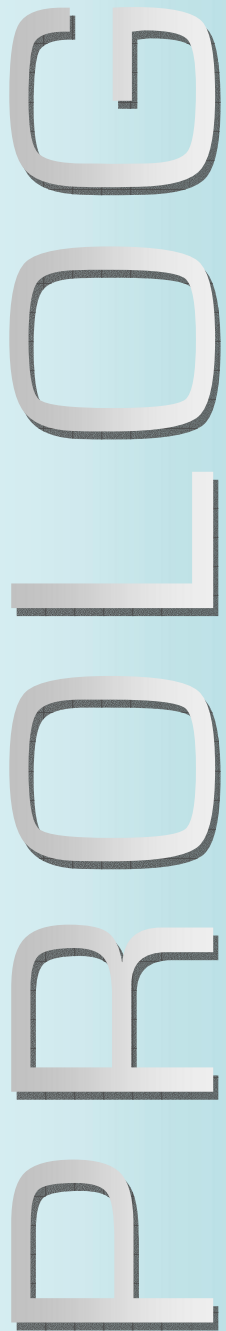
- Now that we can conjoin goals we can also check for their negation i.e. $\neg (P \wedge Q)$.
- Usually we are checking if a conjunction of terms in the body or a clause is true e.g. $a(x) :- b(x), c(x)$.
- But sometimes we want a predicate to succeed only if a conjunction of terms is false

e.g. $a1(x) :- \text{\+}(b(x), c(x))$.

- * The space before the prefix operator \+ and the brackets is important. If there was no space the interpreter would look for $\text{\+}/2$.

- This is distinct from:
 $a2(x) :- \text{\+}b(x), \text{\+}c(x)$.
 Which is equal to the space outside of both b and c.





Creating a Conjoined 'not' (2)

- But when would you use a conjoined not?
 - “*X is true if it is less than 4 or greater than 8.*”
 - For example, we want X to be true if it is 3 or 9.
- We could represent this using a disjunction:
 - $(X < 4 ; 8 < X)$.
- Or we could represent it as a conjoined not:
 - $\neg (4 \leq X, X \leq 8)$.
- This is possible as logic permits this transformation:
 - $\neg P \vee \neg Q = \neg (P \wedge Q)$
- Sometimes it might be easier to prove a goal $(4 \leq X)$ rather than its opposite $(X < 4)$ so we would need to use a conjoined not: $\neg (P \wedge Q)$

Using *if* statements

- In a lot of other programming languages *if.. then... else...* constructions are very common.
- In Prolog there is a built-in operator (**->/2**) that allows you to make similar constructions:
 - “if X then Y” = `X -> Y.`
 - “if X then Y else Z” = `X -> Y; Z.`
 - n.b. the `;` is part of the “if..then...else...” construction so its scope is limited to the *if..* construction.
- These can be used at the command line or within your predicate definitions.
- However, whenever we are writing Prolog rules we are already representing an “if....then....” relationship.
 - This rule `a:- b, c, d, e -> f; g.`
 - Is equal to `a:- b, c, d, aux(X). aux(f):- e. aux(g).`

Meta-interpretation

- You've seen by now that Prolog has its own *proof strategy*, which is the way it goes about trying to solve a goal you give it.
- Goals and sub-goals are taken in a left-to-right, and *depth-first* manner.
- However, since we are able to access the Prolog database, we can manipulate the contents of the Prolog database as if it were any other sort of data (which is what we mean by *meta-programming*--programming where the data is bits of program, rather than information about entities in the world).
- You've seen how we can modify the database, using *assert/1* and *retract/1*.
- But we can also create a *meta-interpreter*, which allows us to create a whole new proof strategy. We don't have to rely on the basic built-in proof strategy which Prolog provides.

A Prolog Meta-interpreter

- A Prolog meta-interpreter takes a Prolog program and a goal and attempts to prove that the goal logically follows from the program.
- The most basic meta-interpreter takes a consulted Prolog program and tries to prove a Goal by calling it:

```
prove (Goal) :-  
    call (Goal) .
```

- `call/1` uses the original Prolog interpreter to prove the Goal so it performs the default proof strategy.
- To begin controlling the proof strategy we need to reduce the 'grain size' of the interpreter (the size of the elements it manipulates).
 - We can do this by using `clause/2`

clause/2

- You might remember `clause/2` from the Database Manipulation lecture (lecture 14).
- `clause(Head, Body)` succeeds if there is a clause in the current Prolog database which is unifiable with:

– `Head :- Body.`

E.g. `:- dynamic a/2. % all predicates must be first declared as dynamic before they can be seen with clause/2.`
`a(1,2).`
`a(X,_):- c(X).`
`a(X,Y):- b(X), b(Y).`

```
|?- clause(a(Arg1,Arg2), Body).
Arg1 = 1, Arg2 = 2, Body = true?;
Body = c(Arg1)?;
Body = b(Arg1), b(Arg2)?;
no
```

- Note that if the clause is a fact, and has no body, then the second argument of `clause/2` is instantiated to *true*.

clause/2 (2)

- If the Body of the clause contains one goal then Body is equal to this:
 - `a(X,_) :- c(X). Body = c(Arg1)`
- If the Body contains several goals then they are instantiated with Body as a pair grouped together by brackets:
 - `Body = (FirstGoal, OtherGoals).`
 - * When this instantiation is printed to the screen the brackets will not be shown.
- `OtherGoals` may again be a pair consisting of another goal and remaining goals
 - `Program: a(X,Y) :- b(X), c(Y), d(Y).`
 - `?- clause(a(Arg1,Arg2),Body), Body = (H, T).`
 - `H = b(Arg1),`
 - `T = c(Arg2), d(Arg2),`
 - `Body = b(Arg1), c(Arg2), d(Arg2) ?`

A simple meta-interpreter

- We can use `clause/2` to match Goal to the Head of a clause and then recursively test the goals in the clause Body.

```

solve(true).           (4) If the clause is a fact then
                        Body = true. Current Goal is proven.

solve(Goal) :-
    \+ Goal = (_, _),  (1) If Goal is a single term
    clause(Goal, Body), (2) find a head that matches Goal
    solve(Body).       (3) and recurse on the clause Body

solve((Goal1, Goal2)) :- (5) If Body contains >1 Goal.
    solve(Goal1),        (6) Try to prove Goal1
    solve(Goal2).       (7) Try to prove rest of goals
                        (remember 2nd argument is a
                        compound structure)

```

- This replicates Prolog's normal proof strategy: attempting to solve each goal in a left-to-right, *depth-first* manner.

A simple meta-interpreter (2)

- For example, if we have consulted this program:

```
:-dynamic a/1,b/1,c/1,d/1.
```

```
a(1).
```

```
a(X):- b(X).
```

```
a(X):- c(X),d(X).
```

```
b(2).
```

```
c(3).
```

```
d(3).
```

- And used solve/1 to prove a goal.

```
solve(true).
```

```
solve(Goal) :-
```

```
    \+ Goal = (_, _),
    clause(Goal, Body),
    solve(Body).
```

```
solve((Goal1, Goal2)) :-
```

```
    solve(Goal1),
    solve(Goal2).
```

```
| ?- solve(a(1)).
```

```
Call: solve(a(1)) ?
```

```
Call: a(1)=(_1032,_1033) ?
```

```
Fail: a(1)=(_1032,_1033) ?
```

```
Call: clause(user:a(1),_1027)?
```

```
Exit: clause(user:a(1),true)?
```

```
Call: solve(true) ? ?
```

```
Exit: solve(true) ? ?
```

```
Exit: solve(a(1)) ?
```

```
yes
```

A simple meta-interpreter (3)

```

:-dynamic a/1,b/1,c/1,d/1.
a(1).
a(X):- b(X).
a(X):- c(X),d(X).

b(2).
c(3).
d(3).

solve(true).

solve(Goal) :-
    \+ Goal = (_, _),
    clause(Goal, Body),
    solve(Body).

solve((Goal1, Goal2)) :-
    solve(Goal1),
    solve(Goal2).

```

```

| ?- solve(a(2)).

Call: solve(a(2)) ?
Call: a(2)=(_1032,_1033) ?
Fail: a(2)=(_1032,_1033) ?
Call: clause(user:a(2),_1027)?
Exit: clause(user:a(2),b(2)) ?
Call: solve(b(2)) ?
Call: b(2)=(_2831,_2832) ?
Fail: b(2)=(_2831,_2832) ?
Call: clause(user:b(2),_2826)?
Exit: clause(user:b(2),true) ?
Call: solve(true) ?
Exit: solve(true) ?
Exit: solve(b(2)) ?
Exit: solve(a(2)) ?

yes

```

A simple meta-interpreter (4)

```
| ?- solve(a(3)).
```

```
Call: solve(a(3)) ?
Call: a(3)=(_1032,_1033) ?
Fail: a(3)=(_1032,_1033) ?
Call: clause(user:a(3),_1027) ?
Exit: clause(user:a(3),b(3)) ?
Call: solve(b(3)) ?
Call: b(3)=(_2831,_2832) ?
Fail: b(3)=(_2831,_2832) ?
Call: clause(user:b(3),_2826) ?
Fail: clause(user:b(3),_2826) ?
Fail: solve(b(3)) ?
Redo: clause(user:a(3),b(3)) ?
Exit: clause(user:a(3),(c(3),d(3)))
Call: solve((c(3),d(3))) ?
Call: (c(3),d(3))=(_2836,_2837) ?
Exit: (c(3),d(3))=(c(3),d(3)) ?
Call: solve(c(3)) ?
Call: c(3)=(_3414,_3415) ?
```

```
Fail: c(3)=(_3414,_3415) ?
Call: clause(user:c(3),_3409) ?
Exit: clause(user:c(3),true) ?
Call: solve(true) ?
Exit: solve(true) ?
Exit: solve(c(3)) ?
Call: solve(d(3)) ?
Call: d(3)=(_6895,_6896) ?
Fail: d(3)=(_6895,_6896) ?
Call: clause(user:d(3),_6890) ?
Exit: clause(user:d(3),true) ?
Call: solve(true) ?
Exit: solve(true) ?
Exit: solve(d(3)) ?
Exit: solve((c(3),d(3))) ?
Exit: solve(a(3)) ?
```

yes

A right-to-left meta-interpreter

- Now that we have this basic meta-interpreter we can begin modifying the proof strategy.
- We could rewrite the interpreter very easily to make it attempt to solve the goal in a right-to-left (backwards) manner.

```

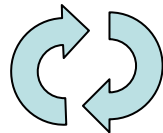
solve(true) .
solve(Goal) :-
    \+ Goal = (_, _),
    clause(Goal, Body),
    solve(Body) .

```

```

solve((Goal1, Goal2)) :-
    solve(Goal2),
    solve(Goal1) .

```



Attempt to solve the rest of the body before solving the first goal.

A right-to-left meta-interpreter (2)

- When trying to solve a goal with more than one subgoal it will try to prove the bottom- (right-) most goal first.

`a(X) :- c(X), d(X).`

1. Try to prove `d(X)`.
2. Then try to prove `c(X)`.

- For clauses in which the order of the goals isn't important, this works fine.
- But, most Prolog clauses represent a development of computation through the clause so ordering is important.
 - e.g. `a(X):- b(Y), X is Y + 1, c(X)`.
 - would fail as `solve/1` *cannot use any built-in predicates or tests* (such as `is/2`).
 - e.g. `a(X):- b(List), member(X,List)`.
 - Fails because the value of `List` must be known before `member/2` can be performed on it.

A breadth-first meta-interpreter

- Putting these concerns aside, we can begin to make meta-interpreters that treat the proof as a search problem.
- We can take our Goal structures, (FirstGoal, OtherGoals), and add the goals to a list (which will be used as an agenda).

```
solve([]).                % All goals proven when []

solve([true|T]) :-
    solve(T).            %once a goal is true recurse on T.

solve([Goal|Rest]) :-
    \+ Goal = (_, _),
    clause(Goal, Body),
    conj2list(Body, List), %turns conjoined goals into a list
    append(Rest, List, New),
    solve(New).

solve([(Goal1, Goal2)|Rest]) :-
    solve([Goal1, Goal2|Rest]).
```

A breadth-first meta-interpreter

- `conj2list/2` takes a structure made up of conjoined goals and adds each goal to a list in order.
- Remember: $(a,b,c,d,e,f) = (a,(b,(c,(d,(e,f))))))$

```
conj2list(Term, [Term]) :-  
    \+ Term = (_, _).
```

```
conj2list((Term1,Term2), [Term1|Terms]) :-  
    conj2list(Term2, Terms).
```

- Note that this works just like breadth-first state-space search: We maintain an *agenda* of the goals yet to be expanded, and add new goals to the end, as we expand them.

A best-first meta-interpreter

- As with state-space search, things become much more intelligent when we think about choosing where to go next based on which is *best*.
- A possible heuristic for choosing which goal to try to solve next is: take *ground* goals (i.e. those with no uninstantiated variables) first. Since they either succeed or fail, and don't depend on any other goals, choosing these first can often prune the search-space significantly.
- We use an auxiliary predicate, `split_ground/3`, to split a list of goals into the ground goals and the unground goals.

A best-first meta-interpreter (2)

- `ground/1` only succeeds if its argument contains no un-instantiated variables.

```
| ?- ground((a(2),b(4),c(x))).      no
| ?- ground((a(2),b(4),c(d))).     yes
```

```
split_ground([], [], []).
```

```
split_ground([Term|Terms], [Term|Ground], UnGround) :-
    ground(Term),
    split_ground(Terms, Ground, UnGround).
```

```
split_ground([Term|Terms], Ground, [Term|UnGround]) :-
    \+ ground(Term),
    split_ground(Terms, Ground, UnGround).
```

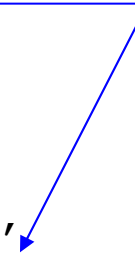
A best-first meta-interpreter (2)

```
solve(true).
```

```
solve(Goal) :-  
    \+ Goal = (_, _),  
    clause(Goal, Body),  
    solve(Body).
```

```
solve((Goal1, Goal2)) :-  
    conj2list((Goal1, Goal2), List),  
    split_ground(List, Ground, UnGround),  
    append(Ground, UnGround, [First|Rest]),  
    solve(First),  
    conj2list(Goals, Rest),  
    solve(Goals).
```

Goals with instantiated variables
(grounded) are checked first.



Other Meta-Interpreters

- The most commonly used meta-interpreter is the *tracer*.
 - This runs Prolog’s normal proof strategy but provides information on what happens with each call (EXIT, FAIL, REDO).
- *Generating Proof trees*: As the meta-interpreter proves goals it can be made to construct a representation of the proof on backtracking.
 - e.g. `gives(john,mary,chocolate) <==`
`(feels_sorry_for(john,mary) <== sad(mary)).....`
- *Object-Oriented Prolog Programs*:
 - We could write our Prolog programs in terms of *objects* and *send messages* (simulating the programming style of C++ and Java).
 - A meta-interpreter could then perform computation based on objects responding to messages.
 - Specific procedures (*methods*) can then be inherited by objects.
- See Bratko, 2001 for information on how to implement these.

Summary

- Controlling the flow of computation: `call/1`
 - Representing logical relationships
 - conjunctions $(P \wedge Q)$: `(FirstGoal, OtherGoals)`
 - disjunctions $(P \vee Q)$: `(FirstGoal; OtherGoals)`
 - conjunctive not $\neg (P \wedge Q)$: `\+ (FirstGoal, OtherGoals)`
 - if.....then....else.....
 - `X -> Y; Z`
- Meta-Interpreters
 - `clause/2`
 - left-to-right interpreter
 - right-to-left interpreter
 - breadth-first: using an agenda
 - best-first: using `ground/1`
 - others