PROLOG – Facts and Rules

Two components:

- Facts;
- Rules.

Generally intended for solving symbolic rather than mathematical problems.

A Prolog program is a database of facts and rules.
Atoms

- The basic element of Prolog.
- Atoms represent things.
- Atoms start with a lower-case letter.
- E.g.
  - atom
  - another_atom
  - molecule
  - atom1
Facts

- Facts are represented as relationships between atoms.
- A relationship is written `relationname(atom,...)`.
- E.g.
  - `teaches(csci337,piper)`.
  - `teaches(csci322,piper)`.
  - `teaches(csci204,castle)`.
  - `teaches(csci336,castle)`.

- Represents the relationship subject ... is taught by ...
- Note that a relationship can have one or more atoms.
Unary, Binary, N-ary relationships.

- Relationships involving a single atom are called unary relationships.
  - E.g.
    - subject(csci337).

- Relationships between pairs of atoms are called binary relationships.
  - E.g.
    - capital(australia,sydney).

- In general, a relationship with n atoms is called an n-ary relationship.
Care With Relationships

Prolog does not know what a relationship means; that is the programmers problem.

E.g.
- `parent(mary,john).`

could mean:
- mary is the parent of john
  or
- the parent of mary is john

The programmer must decide (and be consistent).
Comments

We can always use comments to provide a clue to the interpretation of a relationship.

The Prolog comment mechanism is similar to that of C, using /* comment text */

E.g.

```prolog
/*
   parent( , ) the parent relation
   parent(child,adult).
   */
parent(john,fred).
parent(bill,john).
```
Running Prolog

- We first create a Prolog program in a text file with an extension of .P.
- We then start Prolog.
- Next we load the program.
- Now we are ready to run the program.
A Simple Example

Create a file test.P with the following contents:

- teaches(csci337,piper).
- teaches(csci322,piper).
- teaches(csci204,castle).
- teaches(csci336,castle).

Now start Prolog, at the Unix prompt type:

- xsb

Load the program:

- [test].
A Simple Example

$ xsb
[xsb_configuration loaded]
[sysinitrc loaded]
[packaging loaded]

XSB Version 2.6 (Duff) of June 24, 2003
[i686-pc-cygwin; mode: optimal; engine: slg-wam; gc: indirection; scheduling: local]

| ?- [test].
[Compiling ./test]
[test compiled, cpu time used: 0.0320 seconds]
[test loaded]

yes
| ?-
Asking Prolog Questions

- We can now ask simple true/false questions:
  - Is csci337 taught by piper?
  - Is csci322 taught by castle?

with
- | ?- teaches(csci337,piper).
  yes
  | ?- teaches(csci322,castle).
  no
Not the Right Answer?

What is going on with the following question and answer?

| ?- teaches(csci203,piper).
| no
| ?-

Does Prolog know I teach CSCI 203?

Always remember, Prolog only “knows” what it has been told.
Open-ended Questions and Variables.

If we want Prolog to answer questions like:

- Who teaches CSCI 337

we need to introduce variables:

| ?- teaches(csci337,Who).

Who = piper <\n>

yes

| ?- 

A variable name starts with a capital letter.
Multiple Answers

Sometimes, Prolog can find multiple answers to a question.

E.g. what subjects are taught by piper?

```prolog
| ?- teaches(What, piper).
```

```
What = csci337;
What = csci322;
no
| ?-
```
Multiple Answers

Note that in the preceding example, Prolog pauses at the end of each answer in turn.

If I type enter Prolog will halt the search.

E.g.

\[
\begin{align*}
&| \quad ?- \text{ teaches(What,piper)}. \\
&\quad \text{What} = \text{csci337} <\backslash n>
\end{align*}
\]

yes

| ?-

The semi-colon, signifying “or”, is typed to signify continuation.
More Complex Questions

Let us now ask the following question:

- What subjects are taught by the person who teaches csci337?

We formulate this in Prolog as follows:

```prolog
| ?- teaches(csci337,WHO),teaches(WHAT,WHO).
```

WHO = piper
WHAT = csci337 ;

WHO = piper
WHAT = csci322 ;

no
| ?-

Note the comma separating the sub-questions.
Comma is interpreted as “and”.
A More Complex Problem

Consider the following Prolog database:

```prolog
/* married(<husband>,<wife>) */
married(fred,mary).
moved(sid,doris).
moved(gary,wendy).
moved(ian,sandra).
moved(clark,julie).
moved(gary,sue).
```

Who’s a bigamist?
Find the Bigamist

Let’s try the query:

?- married(Husband, Wife1), married(Husband, Wife2).

Husband = fred
Wife1 = mary
Wife2 = mary ;

Husband = sid
Wife1 = doris
Wife2 = doris ;
...

This did not work – why?
Find the Bigamist

We need to ensure that Wife1 and Wife2 are different.

\[
\text{?- married(Husband, Wife1), married(Husband, Wife2), not Wife1 = Wife2.}
\]

Husband = gary
Wife1 = wendy
Wife2 = sue;

Husband = gary
Wife1 = sue
Wife2 = wendy;

no

Nearly right...
Rules

A rule in Prolog allows the expression of relationships between entities.

E.g.

- If an animal is a dog it is a carnivore.
- If an animal is a cow it is a herbivore.

These rules are written thus:

- carnivore(Animal) :- dog(Animal).
- herbivore(Animal) :- cow(Animal).
We can also express these rules in a more general way:

- \( \text{eats(Animal,meat)} \) :- \( \text{dog(Animal)} \).
- \( \text{eats(Animal,plants)} \) :- \( \text{cow(Animal)} \).

This is probably a better approach to this particular case.

As a general description so far:
- Rules express common regularity in data.

This is not all that rules can do.
Rules

Rules can also relate specific properties to abstract ones.

E.g.

- \texttt{disease(Kid,measles) :- feverish(Kid), spots(Kid)}.

I.e. if a kid has a fever and spots we can conclude that they have measles.
Related Rules

Sometimes, more than one set of facts will lead to the same conclusion:

E.g.

- An animal is a carnivore if it is a cat or a dog.

We represent this with a rule set:

- carnivore(Animal) :- dog(Animal).
- carnivore(Animal) :- cat(Animal).
Testing Rules

We can test a rule in Prolog in exactly the same way we test a fact:

Given the Prolog program:

- `dog(fido).`
- `dog(rex).`
- `cat(fluff).`
- `cow(clara).`
- `sheep(dolly).`

- `eats(Animal,meat) :- dog(Animal).`
- `eats(Animal,meat) :- cat(Animal).`
- `eats(Animal,grass) :- cow(Animal).`
- `eats(Animal,grass) :- sheep(Animal).`
Testing Rules

We can evaluate propositions:

\begin{itemize}
  \item | ?- \texttt{eats(fido,meat)}.
     \begin{itemize}
       \item yes
       \item | ?- \texttt{eats(fido,grass)}.
         \begin{itemize}
           \item no
           \item | ?- \texttt{eats(X,grass)}.
             \begin{itemize}
               \item X = clara;
               \item X = dolly;
               \item no
               \item | ?- \texttt{eats(fido,What)}.
                 \begin{itemize}
                   \item What = meat
                   \item yes
                 \end{itemize}
               \end{itemize}
             \end{itemize}
           \end{itemize}
         \end{itemize}
     \end{itemize}
\end{itemize}
Rule Chains

Given the relationship:

- parent(<adult>,<child>)

we can define the relationship grandparent as follows:

- grandparent(A,C) :-
  parent(A,B), parent(B,C).

This says a grandparent is the parent of a parent.

This rule looks for chains of relationships.
Rule Chains

We can also define the relationship ancestor as follows:

- \text{ancestor}(A,B) \ :- \ \text{parent}(A,B).
- \text{ancestor}(A,C) \ :- \ \text{parent}(A,B), \text{ancestor}(B,C).

This says an ancestor is a parent or the parent of an ancestor.
Rule Chains

The rule:

- \texttt{ancestor(A,C) :- parent(B,C), ancestor(A,B).}

Defines ancestor recursively.
Lists

Rather than write:

- `dog(fido).`
- `dog(rex).`
- `dog(butch).`
- `dog(rover).`

we can write:

- `dogs([fido, rex, butch, rover]).`

This notation creates a list of names.
Operations on Lists

E.g.

<table>
<thead>
<tr>
<th></th>
<th>?- dogs([fido, rex, bruce, rover]).</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>?- dogs([fido, rex, rover, bruce]).</td>
</tr>
<tr>
<td></td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>?- dogs(X).</td>
</tr>
<tr>
<td></td>
<td>X = [fido, rex, bruce, rover]</td>
</tr>
<tr>
<td></td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>?- dogs([X]).</td>
</tr>
<tr>
<td></td>
<td>no</td>
</tr>
</tbody>
</table>
List Membership

A list can be regarded as the composition of an atom (the head) and a list (the tail):

We use the character “|” to denote the composition operator.

|  ?- dogs([Head|Tail]).

Head = fido
Tail = [rex, bruce, rover]

yes

Can we tell if a name is in the list?
List Membership

Consider the following program:

```prolog
dogs([fido, rex, bruce, rover]).

member(X, [X|Y]).
member(X, [H|T]) :- member(X, T).
```

We notice that we get warnings about the variables Y and H because each only appears once within a rule.

What can we do about this?
Anonymous Variables

The special variable "_" can be used in any rule where we want a variable to appear only once.

We can re-write the previous program as:

```
dogs([fido,rex,bruce,rover]).

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

which eliminates the warnings.
List Membership

We now compile the amended program and test it:

- | ?- member(a, [a, b, c]).
  yes
- | ?- member(b, [a, b, c]).
  yes
- | ?- member(d, [a, b, c]).
  no

So far, so good.
List Membership

Now to test dogs:
- | ?- dogs(X),member(fido,X).
  X = [fido, rex, bruce, rover]
  yes
- | ?- dogs(X),member(rex,X).
  X = [fido, rex, bruce, rover]
  yes
- | ?- dogs(X),member(goofy,X).
  no
Building Lists

Consider the following Prolog program:

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

Which we can use:

- `| ?- append([a,b,c], [d], X).
  X = [a,b,c,d]
  yes`

- `| ?- append([a,b], [c,d], X).
  X = [a,b,c,d]
  yes`
Building Lists

We can even use it backwards:

\[
\begin{align*}
&\text{?- append(X,Y,[a,b,c,d])}.

&X = []
Y = [a,b,c,d];

&X = [a]
Y = [b,c,d];

&X = [a,b]
Y = [c,d];

&X = [a,b,c]
Y = [d];

&X = [a,b,c,d]
Y = [];\]

&\text{no}
\end{align*}
\]
Reversing a List

Consider the following Prolog program:

- `append([], X, X).`
- `append([A|B], Y, [A|Z]) :- append(B, Y, Z).`
- `reverse([], []).`
- `reverse([H|T], L) :-`
  - `reverse(T, L1), append(L1, [H], L).`

Which we can use:

- `| ?- reverse([a,b,c,d,e], X).`

  `X = [e,d,c,b,a]`

  `yes`
Prolog supports the function \texttt{write()} for output.

\begin{itemize}
  \item \texttt{| ?- write(a).}
  \begin{verbatim}
  a
  yes
  \end{verbatim}
  \item \texttt{| ?- write([a,b,c]).}
  \begin{verbatim}
  [a,b,c]
  yes
  \end{verbatim}
  \item \texttt{| ?- X=a,write(X).}
  \begin{verbatim}
  a
  X = a
  yes
  \end{verbatim}
\end{itemize}
Input/Output

Prolog supports the function `read()` for input.

E.g.

- `?- read(X),write(X).
  aaa.
  aaa
  X = aaa
  yes`

- `?- read(X),write(X).
  [a,b,c].
  [a,b,c]
  X = [a,b,c]
  yes`
Input/Output

Or, a more complex example:

```prolog
| ?- read(X), X=[H|T], write(H).
[\text{a,b,c}].
\text{a}
X = [\text{a,b,c}]
H = \text{a}
T = [\text{b,c}]

yes
```
Arithmetic

- Prolog has built-in arithmetic operations and an arithmetic evaluator.
- We can use either infix $2 + 3$ or prefix $+(2,3)$ notation.

Let’s try this:

```
| ?- write(2+3).
2 + 3
yes
```

What happened?
Arithmetic

Prolog cannot tell whether you want to consider a term as a term or to evaluate it. You have to tell Prolog to evaluate a term explicitly using “is”.

\[-\text{What is } 2+3.\]

\[\text{What } = 5\]

\[\text{yes}\]
Arithmetic

This explains the following:

- | ?- 1 + 2 = 2 + 1.
  no

- | ?- X = 1 + 2, Y = 2 + 1, X=Y.
  no

- | ?- X is 1 + 2, Y is 2+1, X = Y.
  X = 3
  Y = 3
  yes
An Arithmetic Program

Consider the following Prolog program:

\begin{itemize}
  \item \texttt{gcd(U,0,U)}.
  \item \texttt{gcd}(U,V,W) :- not(V=0), R is U mod V, \texttt{gcd}(V,R,W).
\end{itemize}

which calculates the greatest common divisor of \(U\) and \(V\).

E.g.

\begin{itemize}
  \item \texttt{?- gcd(6125,6450,X)}.
\end{itemize}

\begin{align*}
\text{x} & = 25 \\
\text{yes}
\end{align*}
We saw that we could use append to list all possible pairs that could be joined to form a given list.

Consider the function:

\[
\text{listpairs}(L) :- \\
\text{append}(X,Y,L), \text{write}(X), \text{write}(Y), \text{nl}.
\]

where \text{nl} writes a new line.
Looping

Running this gives:

| ?- listpairs([1,2,3]).
  
  [[]][1,2,3]

  yes

which is only the first pair.

We must force Prolog to backtrack even if a match is found.

We do this with the predicate “fail”.
Looping

Changing our function:

- `listpairs(L) :-
  append(X,Y,L),write(X),write(Y),nl,
  fail.`

We now get:

- `| ?- listpairs([1,2,3]).
  [] [1,2,3]
  [1] [2,3]
  [1,2] [3]
  [1,2,3] []

  no`
Infinite Loops

The function:
- \texttt{num(0)}.
- \texttt{num(X) :- num(Y), X is Y + 1.}

generates all integers \( \geq 0 \) as the goal of \texttt{num(A)}.

Thus:
- \texttt{| ?- num(A).}

\hspace{1cm} A = 0;
\hspace{1cm} A = 1;
\hspace{1cm} A = 2;
\hspace{1cm} ...

Infinite Loops

We try the program:

```prolog
listnum(I,J) :-
    num(X), I =< X, X =< J,
    write(X), nl, fail.
```

to list the numbers from I to J.

When run:

```
| ?- listnum(1,3).
1
2
3
```

but Prolog then goes into a hung state.
Infinite Loops

- It is testing every integer to see if it is $\leq 3!$
- This time our loop is too resilient.
- Prolog has a mechanism to stop processing, the cut, written "!".
- We can modify the listnum function as follows:

```prolog
listnum(I,J) :-
    num(X), I =\leq X, X =\leq J,
    write(X), nl, X = J, !, fail.
```

Infinite Loops

When we run this version we get:

```
| ?- listnum(1,3).
  1
  2
  3
no
```

which is what we want.

Strictly, the cut and fail are redundant in this example, the $X = J$ assertion is enough for termination.
If-Then-Else

We can use the cut to emulate if-then-else constructs we have seen in imperative languages:

- \( D :- A, !, B. \)
- \( D :- C. \)
Primes

Given the program:

- `primes(Limit,Ps) :-
  integers(2,Limit,Is),sieve(Is,Ps).
  integers(Low,High,[Low|Rest]) :-
    Low =< High, !, M is Low + 1,
    integers(M,High,Rest).
  integers(_,_,[]).
  sieve([],[]).
  sieve([I|Is],[I|Ps]) :- remove(I,Is,New),
    sieve(New,Ps).
  remove(_,[],[]).
  remove(P,[I|Is],[I|Nis]) :-
    not(0 is I mod P), !,
    remove(P, Is, Nis).
  remove(P,[I|Is],Nis) :- 0 is I mod P, !,
    remove(P, Is, Nis).`
We get the following output when it is run:

\[- \texttt{primes(40,P).} \]

\[
P = [2,3,5,7,11,13,17,19,23,29,31,37]
\]

\text{yes}

which are the primes < 40.
Let us consider this program in detail:

```
primes(Limit,Ps) :-
    integers(2,Limit,Is), sieve(Is,Ps).
```

This asserts that if Is, is a list of integers from 2 to limit and Ps is the list derived from Is by sieve then Ps is a list of the primes between 2 and Limit.

Seems reasonable.
Primes

Then integers:

\[
\text{integers}(\text{Low}, \text{High}, [\text{Low}|\text{Rest}]) :- \\
\text{Low} \leq \text{High}, !, \\
\text{M is Low + 1,} \\
\text{integers(M, High, Rest).} \\
\text{integers(_, _, []).}
\]

Here, we assert that the list of integers ranging from Low to High is the low value, followed by the list of integers from Low + 1 to High, provided that Low \( \leq \) High.
Primes

Now for sieve:

- $\text{sieve}([],[])$.
- $\text{sieve}([I|Is],[I|Ps]) :\text{-}
  \text{remove}(I,Is,New), \text{sieve}(New,Ps)$.

This says that if we have a list starting with $I$ and followed by $Is$ we can get a list $New$ by removing multiples of $I$ from $Is$ and then recursively sieve $New$ to remove multiples of its head.
Finally, remove:

- `remove(_, [], []).`
- `remove(P, [I|Ps], [I|NPs]) :-
  not(0 is I mod P), !,
  remove(P, Ps, NPs).`
- `remove(P, [I|Ps], NPs) :-
  0 is I mod P, !,
  remove(P, Ps, NPs).`

This removes the head of the list if it is a multiple of \( P \) and recursively processes the tail.
Adding Facts

Often, we need the result of a function to be saved as a fact in the Prolog database.
We can do this with the “assert” predicate.
The expression:
\[ | ?- \text{assert}(	ext{dog(fido)}). \]
in the runtime environment is almost the same as:
\[ \text{dog(fido)}. \]
read in from a file.
Removing Facts

- In a similar way, we can also remove facts from the database.
- This is done with the “retract” predicate.
- Thus, following sequence:
  - | ?- assert(dog(fido)).
    yes
  - | ?- retract(dog(fido)).
    yes
  - | ?- dog(X).
    no
- Prolog no longer “knows” fido is a dog.
Adding to lists

Consider the following program:

```
newdogs :- assert(dogs([])).

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

adddog(X) :- dogs(L), retract(dogs(L)),
append(L,[X],M), assert(dogs(M)).
```
Adding to lists

Running this program:

- | ?- newdogs.
  yes
- | ?- adddog(fido).
  yes
- | ?- adddog(rex).
  yes
- | ?- adddog(prince).
  yes
- | ?- dogs(X).
  X = [fido, rex, prince]
  yes
Searching Lists

If we add the following:

- `isdog(X):-dogs(L),nl,match(X,L).`
- `match(X,[]):-
  write('No, '),write(X),
  write(' is not a dog.'),nl.
- `match(X,[Y|_]):-X=Y,
  write('Yes, '),write(X),
  write(' is a dog.'),nl.
- `match(X,[Y|T]):-not X=Y,match(X,T).`

to our dogs program.
Searching Lists

We can now search the list:

| ?- isdog(rex).

Yes, rex is a dog.

yes

| ?- isdog(lassie).

No, lassie is not a dog.

yes
Looped I/O

Consider the following code:

```
go:- read(X), not X=stop, write(X), nl, go.
```

which reads a sequence of lines terminated by stop.

Thus:

```
|  ?- go.
  aaa.
  aaa
  ccc.
  ccc
  stop.

  no
```

This approach has the disadvantage that we must type “.” at the end of each input.
Formatted I/O

The version of Prolog we are using supports functions “fmt_read” and “fmt_write” for C-style formatted I/O.

E.g.

```
  go:- fmt_read("%s",X,_,), not X=stop,
       fmt_write("Input is %s.\n",X), go.
  | ?- go.
    aaa
    Input is aaa.
    ccc
    Input is ccc.
    stop
    no
```

And we have no “. ”s
Let's add looped, formatted I/O to the dogs program.

1. go:- newdogs, add_loop, test_loop.
2. add_loop:-
   fmt_read("%s", X, _), add_rep(X).
3. add_rep(X):- X=stop, !.
   add_rep(X):- not X=stop,
              adddog(X), add_loop.
4. test_loop:-
   fmt_read("%s", X, _), test_rep(X).
5. test_rep(X):- X=stop, !.
   test_rep(X):- not X=stop,
              isdog(X), test_loop.
Formatted I/O

Running this program:

| ?- go. 
  
fido rex prince stop 
lassie fido goofy stop 

No, lassie is not a dog.

Yes, fido is a dog.

No, goofy is not a dog.

no

Note: we can supply multiple atoms on one line.
Sorting

Consider the following program:

- \textit{ssort}(X,Y):-\textit{nl},\textit{permutation}(X,Y),\textit{sorted}(Y).
- \textit{sorted}([]).
  \textit{sorted}([[]]).
  \textit{sorted}([[]]).
  \textit{sorted}([[X,Y|Z]]):-X=<Y,\textit{sorted}([Y|Z]).
- \textit{permutation}([],[]).
  \textit{permutation}(X,[Y|Z]):-\textit{append}(U,[Y|V],X),
  \textit{append}(U,V,W),
  \textit{permutation}(W,Z).
- \textit{append}([],X,X).
  \textit{append}([A|B],Y,[A|W]):-\textit{append}(B,Y,W).
Sorting

This provides both a specification for what a sort should do and a program to actually do the sort.

\[
\begin{align*}
& \mid \ ?-\ ssort([2,3,1],X). \\
& \quad \ x=[1,2,3] \\
& \quad \text{yes}
\end{align*}
\]

It is hardly efficient though.
Taking a longer list:

\[
\begin{align*}
| & ?- \text{ ssort([9,7,5,3,1,2,4,6,8,0],X)}. \\
\text{x} & = [0,1,2,3,4,5,6,7,8,9] \\
\text{yes}
\end{align*}
\]

Takes more than 10 seconds!

This is because we are generating all permutations of the list until we find a sorted one.
Sorting

This is one of the problems with Prolog:
- The obvious solution is rarely the best solution.

To improve on the sort performance we need to specify the steps that an algorithm must take to perform the sort.

Thus, if we take the quicksort algorithm as a basis for our program we get:
QuickSort

- `qsort([], []).`
  `qsort([H|T], S):-`
  `partition(H, T, L, R), qsort(L, L1), qsort(R, R1), append(L1, [H|R1], S).`

- `partition(P, [A|X], [A|Y], Z):-`
  `partition(P, [A|X], Y, [A|Z]):-`
  `partition(P, [], [], []).`

- `append([], X, X).`
  `append([A|B], Y, [A|W]):- append(B, Y, W).`
Quicksort

Running this program:

?- qsort([9,7,5,3,1,2,4,6,8,0],X).

X=[0,1,2,3,4,5,6,7,8,9]

yes

produces the sorted list in an instant.

Thus, the goal of automated programming with Prolog seems to be far from straightforward.
A Key Limitation to Prolog

Consider the following program:

- \texttt{fact(0,1).}
- \texttt{fact(N,R):-N>0,N1 is N-1,}
  \hspace{1cm} \texttt{fact(N1,R1),R is N*R1.}

which computes the factorial function.

We can use it in a forward direction:

- \texttt{| ?- fact(5,X).}

\begin{verbatim}
x=120
\end{verbatim}

\begin{verbatim}
yes
\end{verbatim}
A Key Limitation to Prolog

- What if we try it in reverse?
  - | ?- fact(X,120).
    + Error[XSB]: [Runtime/C] Uninstantiated argument of evaluable function -/2
      Goal: _Var - 0, probably as 2nd arg of is/2
      Aborting...

- Not exactly what we want.
- The problem is that “is” is not reversible.
- Prolog cannot do maths backwards!
Tracing Prolog

Prolog has a built-in trace function.

| ?- trace.

Once set, all calls will be traced.

E.g. with program fact loaded:

| ?- fact(2,X).
  (0) Call: fact(2, _h85) ? <\n>
  (1) Call: fact(1, _h141) ? <\n>
  (2) Call: fact(0, _h164) ? <\n>
  (2) Exit: fact(0, 1) ? <\n>
  (1) Exit: fact(1, 1) ? <\n>
  (0) Exit: fact(2, 2) ? <\n>

X = 2

yes
[trace]
Tracing Prolog

- We can turn tracing off:
  ```prolog
  | ?- notrace.
  ```

- Within a trace sequence we can skip over calls:
  ```prolog
  | ?- fact(7,X).
  (0) Call: fact(7, _h85) ? \n
  (1) Call: fact(6, _h141) ? s
  (1) Exit: fact(6, 720) ? \n
  (0) Exit: fact(7, 5040) ? \n
  X = 5040
  yes
  [trace]
  ```
Prolog and Performance

Efficiency is not usually a key issue with Prolog.

However, examining trace output can sometimes show issues that are easy to overcome.

E.g. in the dogs program:

- Using append causes a lot of unnecessary work because we don’t care about the order of the created list.
Prolog and Performance

To speed the program up we replace

- \texttt{append([],X,X)}.
  \texttt{append([A|B],Y,[A|W]) :- append(B,Y,W)}.

with

- \texttt{prepend(X,[],[X])}.
  \texttt{prepend(X,Y,[X|Y])}.

and

- \texttt{adddog(X) :- dogs(L), retract(dogs(L)),
  append(L,[X],M), assert(dogs(M)).}

with

- \texttt{adddog(X) :- dogs(L), retract(dogs(L)),
  prepend(X,L,M), assert(dogs(M)).}
Variable Initialisation

- The newdogs predicate in our dogs program was a means of avoiding having to type
  - `assert(dogs([]))`
  at the command line.

- We still have to type
  - `newdogs`
  however.

- We can avoid all of this if we can assert `dogs([])` when we load the program.
Variable Initialisation

- We do this by adding the line:
  
  ```
  :- assert(dogs([])).
  ```

  to dogs.P

- Now, when dogs is loaded, the assertion is made and we are in business.
Auto-run

In a similar manner we can start the execution of dogs by adding a line
- `:- go.`
to the program.

Now, when dogs is loaded, the “go” program is run automatically.

If we change go by adding halt to the end:
- `go:-newdogs,add_loop,test_loop,halt.
we get a program that exits xsb on completion.`