Control in Prolog

• , represents **and**

• ; represents **or**

• -> represents **if ... then**

• -> ; represents **if .. then ... else ...**
Disjunction “;”

By itself ; is just really necessary for one can replace rules such as

\[ a:~ b,c,d, (e\mid f). \]

by either

\[ a:~ b,c,d,e. \]
\[ a:~ b,c,d,f. \]

or (better)

\[ a:~ b,c,d,h. \]

\[ h:~ e. \]
\[ h:~ f. \]

It is however often useful because it may be easier to read.
if ... then ... else

Suppose that we have the following database:

a(1,a). a(1,b). a(2,b). a(3,c).
b(b, 4). b(b,5).
c(10). c(20).
g(X,Z) :- a(X,Y) -> b(Y,Z) ; c(Z).

Consider the goals.

• g(1,Z).

• g(2,Z).

• g(3,Z).

• g(4,Z).
?- g(1, Z).

no

?- g(2, Z).

Z = 4 ?

Z = 5

yes

?- g(3, Z).

no

?- g(4, Z).

Z = 10 ?

Z = 20

yes

Why?
The cut (!)

Purposes of the cut:

1. To turn a non-determinate predicate into a determinate one. Eg.

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

2. To exclude some cases by committing to the current choice:

   
   - max(X, Y, X) :- X >= Y, !.
   - max(X, Y, Y).

There is a real problem with this second example. For max(10,0,0) will succeed. So this apparently more verbose version is better.

   
   - max(X, Y, X) :- X >= Y, !.
   - max(X, Y, Y) :- X < Y.
Example: Ordered Search trees

We represent a node by n(Item, Left, Right) where [] is used to denote the empty tree.

Then we have insert(Item, OldTree, Newtree):

\[
\text{insert}(I, [], n(I, [], [])). \\
\text{insert}(I, n(N,L,R), n(N,L1,R)) :- I < N, !, \\
\hspace{1cm} \text{\% really to left} \\
\hspace{2cm} \text{insert}(I,L,L1). \\
\text{insert}(I, n(N,L,R), n(N,L,R1)) :- I > N, !, \\
\hspace{1cm} \text{\% really to right} \\
\hspace{2cm} \text{insert}(I, R, R1). \\
\text{insert}(I,n(I,L,R),n(I,L,R)). \hspace{1cm} \text{\% already there.}
\]
Once

It is useful that there is a required builtin once/1, which behaves as though defined by.

\[
\text{once}(G) :- \text{call}(G), !.
\]

Note the (meta) predicate \text{call}/1 which treats its argument as a goal.
Negation as failure

What does it mean when the interpreter responds \texttt{no}?

There is a \textit{closed world assumption} that what cannot be proved is false. So that \texttt{no} really means \texttt{not provable}.

Traditionally \texttt{not/1} is defined as if by

\begin{verbatim}
not(G) :- call(G), !, fail.
not(_).
\end{verbatim}

The notation \texttt{\+/1} is also used.
Some versions of Prolog require that the term $G$ in $\text{not}(G)$ be ground when the predicate is called. When this is not the case one can define the useful.

\[
\text{satisfiable}(G) :- \ \text{\texttt{\textbackslash + \textbackslash + G}}.
\]
Repeat

repeat is as though defined by

repeat.
repeat :- repeat.

You can use it to drive loops.

loop :-
    repeat,
    read(X),
    write(X),nl,
    (X = end -> !; fail).
Exception Handling

Exception handling is done using a pair of control constructs.

- catch(G,C,R)

- throw(B)

The following example is quite artificial, but it is required to make a point. We will follow it with a much more sensible use of the constructs.
Catch and throw example 1

Consider the following database:

\[
g :- \text{catch}(p, B, \text{write}(h2)), \\
    \text{coo}(c).
\]

\[
p.
\]
\[
p :- \text{throw}(b).
\]

\[
\text{coo}(X) :- \text{throw}(X).
\]

and the goal \text{catch}(g, C, \text{write}(h1)).
A more sensible example

safe_open(File) :-
catch(open(File, read, S), B, (write(B), nl)).
Difference Structures

The idea is to represent a list as a difference between tow lists so that \([a,b,c] = [a,b,c,d,e,f] - [d,e,f]\). This gives a slick version of append and of flatten.

\[
\text{concat}(A-B, B-C, A-C).
\]

\[
\text{flatten}(X,Y) :- \text{flatpair}(X, Y - []).\]

\[
\text{flatpair}([], L-L).
\]

\[
\text{flatpair}([H|T], L1 - L3) :-
\quad \text{flatpair}(H, L1-L2), \text{flatpair}(T, L2-L3).
\]

\[
\text{flatpair}(X, [X|Z] -Z).
\]
Definite Clause Grammars DCGs

The operator --> has a special meaning in Prolog files. It corresponds to the usage in production rules.

The ‘rules’

a --> b, c.
b --> [fred].
c --> [eats].

are translated to

a(L1,L3) :- b(L1,L2), c(L2,L3).
b([fred|L],L).
c([eats|L], L).

So that the goal a([fred, eats], []). is true.
A Piece of an HTML generator  
Courtesy the folks at UPI Madrid

html_term(title(T)) --> !,  
   ['<title>',T,'</title>'],  
   newline.

html_term(ref(Addr,Text)) --> !,  
   ['<a>'],  
   html_attts([href=Addr]),  
   ['>'],  
   html_term(Text),  
   ['</a>'].

html_term(table(Attrs, Rows)) --> !,  
   ['<table>'],  
   html_attts(Attrs),  
   ['>'],  
   newline,  
   html_rows(Rows),  
   ['</table>'].
Streams

• `open(SourceSink, Mode, Stream)`.

• `open(SourceSink, Mode, _, [alias(S)])`.

• `close(StreamOrAlias)`.

• `current_input(Stream)`.

• `set_input(Stream)`.

• `set_output(Stream)`.
Character I/O

• get_char(StrOrAlias, C).

• get_char(C).

• put_char(StrOrAlias, C).

• put_char(C).
Term I/O

- read(T).
- read(S,T).
- write(T).
- write(S,T).
- writeq(T).
- write_canonical(T).
- nl.
- nl(S).
Operators

• \texttt{current\_op}(Prio, Fix, Op).

• \texttt{op}(Prio, Fix, Op).

• \(0 \leq Prio \leq 1200\)

• \(\text{Fix} = xfx, xfy, yfx, xf, yf, fx, \text{ or } fy\)

If we were to define \texttt{op}(25, xfx, @). Then we could parse jhodgson@sju as @\texttt{(jhodgson, sju)}.

You get into trouble with the ’.’
Modules

There is an ISO standard for Prolog Modules.

Unfortunately nobody implements it properly yet.

Modules have

1. An interface, specifying the predicates that are exported, plus something called metapredicates of which more later.

2. Zero or more bodies.
The Meta predicate problem

Suppose that in a module \texttt{m} we define a predicate \texttt{p} that uses \texttt{call/1}. Eg.

\begin{verbatim}
my_once(G) :- call(G), !.
\end{verbatim}

Suppose we export \texttt{my_once} and use it in a module \texttt{n}. Eg. in \texttt{n} we have \texttt{g(X) :- my_once(pred(X))}.

Which version of \texttt{pred/1} is called the one in \texttt{m} where \texttt{my_once} is defined or one in \texttt{n}.

This is the metapredicate problem.
A module example

:- module(utilities).
:- export([length/2, reverse/2]).
:- end_module(utilities).
:- body(utilities).
  length(List, Len) :- length1(List, 0, N).
  length1([], N, N).
  length1([H | T], N, L) :-
    N1 is N + 1,length1(T, N1, L).

  reverse(List, Reversed) :-
    reverse1(List, [], Reversed).
  reverse1([], R,R).
  reverse1([H | T], Acc, R) :-
    reverse1(T, [H | Acc], R).
:-end_body(utilities).

:- module(foo).
:- end_module(foo).
:- body(foo).
:-import(utilities).
  p(Y) :- q(X),length(X,Y).

  q([1,2,3,4]).
:- end_body(foo).
An example with metapredicates

:- module(trace).
   :- exports(#/1).
   :- metapredicate(#( :) ).

:- end_module(trace).
:- body(trace).
   :- op(950, fx, #).

(# Goal) :-
  Goal = Module : G,
  inform_user(’CALL’, Module, G),
  call(Goal),
  inform_user(’EXIT’, Module, G).
(# Goal) :-
  Goal = Module : G,
  inform_user(’FAIL’, Module, Goal),
  fail.
inform_user(Port, Module, Goal) :-
  write(Port), write(' '), write(Module),
  write(' calls '), writeq(Goal), nl.
:- end_body(trace).
:- module(sort_with_errors).
    :- export(sort/2).
:- end_module(sort_with_errors).
:- body(sort_with_errors).
    :- import(trace).
sort(List, SortedList) :-
    sort(List, SortedList, []).
sort([], L, L).
sort([X|L], R0, R) :-
    # split(X,L,L1,L2),
    # sort(L1, R0, R1),
    # sort(L2, [X|R1], R).
split(_, [], [], []).
split(X, [Y|L], [Y |L1], L2):-
    Y @< X, !,
    split(X,L, L2, L2).
split(X, [Y | L], [Y |L1], L2):-
    Y @< X, !,
    split(X,L, L2, L2).
split(X, [Y | L], [Y |L1], L2):-
    split(X, L, L2, L2).
:- end_body(sort_with_errors).