Prolog: Logic or Procedure?

Opinion:

- **anyone who tells you "Prolog is purely logic" does not really know Prolog**

Why? For example:

- **real Prolog programs have to worry about infinite loops**
- **the "cut"**

Both are significant real problems
and both depend only on "order of evaluation"
(that is, the particular proof/search strategy that Prolog uses)

Prolog as Procedure: The Cut

A way of aborting in mid-backtrack:

```
goal :- subgoal1, !, subgoal2.
```

Backtracking can go past ! in forward direction

- **but in backward direction, entire goal immediately fails (do NOT try alternative ways to satisfy this goal)**

Example : inequality:

```
equal(X, X).
not_equal(X, Y) :- equal(X, Y), !, fail.
not_equal(X, Y).
```

Prolog as Procedure: `not`

Can also use `not` as an abbreviation for the previous idiom

. . . a trap for the unwary, because `not` is different from logical negation

But recommended anyway

because `not` is often easier to understand than the general cut

```
not_equal(X, Y) :- not(equal(X, Y)).
```

Definition of built-in `not` predicate:

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

Actually:

```
not(Z) :- call(Z), !, fail.
not(Z).
```

Unfortunately `not` is not just simple logical negation.

Its behavior is only understood by appealing to the procedural view of Prolog.

Another example of cut

Restrict library facilities for overdue borrowers:

```
service(Patron, Facility) :- overdue(Patron, Book),
                    !, basic_service(Facility).
service(Patron, Facility) :- any_service(Facility).

basic_service(reference).
basic_service(enquiries).

extra_service(borrowing).
extra_service(interlibrary_loan).
extra_service(audio_visual).

any_service(F) :- basic_service(F).
any_service(F) :- extra_service(F).
```

General uses of cut

If you found the right rule, cut out later ones

Example: sum of the integers from `1` to `N`

```
sum(1,1) :- !.
sum(N, Answer) :-
   N1 is N - 1, sum(N1, A), Answer is A+N
```

Cut avoids infinite loop if `sum(1,1)` is in failing supergoal

```
ok :- sum(1, X), more(foo).
```

terminates even if `more(foo)` fails

To fail without trying alternatives:

```
   ..., !, fail.
```

Stops generating more alternatives

Extended Prolog Example: Tic-Tac-Toe

Two players, `x` and `o`:

```
otherplayer(x, o).
otherplayer(o, x).
```

Board has 9 squares:

```
square(1). square(2). square(3).
square(4). square(5). square(6).
square(7). square(8). square(9).
```

Win is three in a row across, down, or diagonally:

```
in_row(1, 2, 3). in_row(4, 5, 6). in_row(7, 8, 9).
in_row(1, 4, 7). in_row(2, 5, 8). in_row(3, 6, 9).
in_row(1, 5, 9). in_row(3, 5, 7).
```

Tic-Tac-Toe helper predicates for lists

Need access to lists by element number:

```
nth(0, [X|_], X).
nth(I, [_|Y], Z) :- I1 is I - 1, nth(I1, Y, Z).
```

Can also change lists by element number:

```
update([_|Y], 0, P, [P|Y]).
update([X|Y], I, P, [X|Z]) :-
  I1 is I - 1, update(Y, I1, P, Z).
```

Tic-Tac-Toe board

Board "configuration" is a list of length 10 that consists of:

- **which player's turn to move next, and**
- **current contents of squares**

```
to_move(B, P) :- nth(0, B, P).
```

Initial board has designated player, empty squares:

```
initial([P|S], P) :- initial_squares(S, 9).
initial_squares([ ], 0) :- !.
initial_squares([none|X], N) :-
  N1 is N - 1, initial_squares(X, N1).
```

Example:

```
?- initial(B, x).
B = [x, none, none, none, none, none, none, none, none, none]
yes
```

Tic-Tac-Toe moves

On board `B`, a move to square `S` leads to board `BB`:

```
move(B, S, BB) :-
  square(S),
  nth(S, B, none),
  to_move(B, P),
  update(B, S, P, BBB),
  otherplayer(P, Q),
  update(BBB, 0, Q, BB).
```

Board `B` is a win for player `P` if player `P` has 3 in a row

(note implicit search!)

```
won_for(B, P) :-
  in_row(X, Y, Z),
  nth(X, B, P),
  nth(Y, B, P),
  nth(Z, B, P).
```

Tic-Tac-Toe predictions

Any game with complete information has 1 of 3 outcomes for player to move:

```
forecast(B, won)  :- forced_win(B).
forecast(B, lost) :- lost_position(B).
forecast(B, tied) :- guaranteed_tie(B).
```

Player can force a win if

- **player has already won, or**
- **player hasn't lost, and can put opponent in lost position**

```
forced_win(B) :- to_move(B, P), won_for(B, P).
forced_win(B) :-
  to_move(B, P), not(lost_for(B, P)),
  move(B, _, BB), lost_position(BB).
```

Predictions (continued)

Need to know losses and ties:

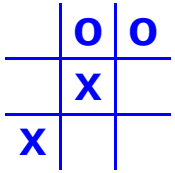
```
lost_for(B, P) :-
   otherplayer(P, Q), won_for(B, Q).

tied(B) :- won_for(B, _), !, fail.
tied(B) :- move(B, _, _), !, fail.
tied(B).
```

Predictions (continued)

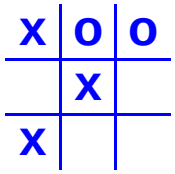
Can avoid loss if already tied, or opponent can't force win

```
saveable(B) :- tied(B).
saveable(B) :- move(B, _, BB), not(forced_win(BB)).

lost_position(B) :- to_move(B, P), lost_for(B, P).
lost_position(B) :- not(saveable(B)).

guaranteed_tie(B) :-
  not(forced_win(B)), not(lost_position(B)).
```

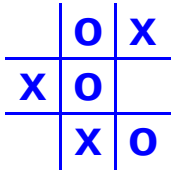
Some example predictions




```
?- forecast([x,none,o,o,none,x,none,x,none,none], Z).
Z = won
```




```
?- forecast([o,x,o,o,none,x,none,x,none,none], Z).
Z = lost
```




```
?- forecast([x,none,o,x,x,o,none,none,x,o], Z).
Z = tied
```

Power of Prolog

Pattern matching with untyped predicates, functors, and lists!

Backtracking

Unification

Moreover

- **rules, goals are also data!**

  `assert`, `retract` **modify database**

  `call(P)` **treats** `P` **as a goal, where** `P` **can be constructed at run-time**

The cut is needed but can be hard to understand

The logical model is cute but it's not the same as what Prolog actually does

- **must understand the procedural evaluation model**

Very cute language as a "thought experiment"

People do use it for real searching problems and for AI problems (especially natural language understanding)

- **also used for general-purpose programming, but only rarely**