Today's discussion

Videos

Countdown

Graph search
Q: You mentioned that we can use cuts and negation in the exam. Can we also use implication (->)?
Q: You mentioned that we can use cuts and negation in the exam. Can we also use implication (->)?

A: **No.** You also can’t use ‘;’, assume any library predicates, or use any extra-logical stuff (except cut) like `findAll`, `call` etc.
Q: When figuring out what a Prolog program does, how can we work out which of the arguments are intended to be supplied with constants, and which with variables.

A: Did I manage to answer this last time?

% foo(+X,-Y) succeeds if output number Y
% is double input number X
foo(X,Y) :- Y is 2 * X.
Q: What does Prolog allow us to do (other than coding in a different way) that other languages can't? Not meaning to sound dismissive just curious of applications!
Q: What does Prolog allow us to do (other than coding in a different way) that other languages can't? Not meaning to sound dismissive just curious of applications!

A:
* Pure Prolog subset 'Datalog' used for network verification.
* Prolog used for Java Virtual Machine verification
* Prolog quite good at digital logic simulation
... theorem provers written in Prolog.

* Sooner or later, some method of reasoning with NN data will emerge.
CDBB Digital Architecture for Real-Time Data

I. Lewis, R. Mortier
Dec. 2019

Building: IfM / CL / CEng

Key:
- Real-time dataflow
- Request-Response data
Sensors producing real-time data
Where does our logic fit in?
Q: Operators & precedence? :- op(700, xfx, arc).

1. Precedence 0..1200 (0 highest)
2. (...) has precedence 0
3. -: and , both have low precedence so you can have
   (complicated stuff) ::= (more complicated stuff), ..., ...
4. . is an "end delimiter"

fx Prefix (non-associative).
fy Prefix (right-associative) e.g. fact fact 3.
xfx Infix (non-associative)
xfy Infix (right-associative)
yfx Infix (left-associative)
Q: Operators & precedence?

\[ \text{arc}(X, Y) \]
\[ \text{op}(700, \text{xfx}, \text{arc}). \]

Used 700 because that's typical for a relation (aka Predicate).
Used xfx because we won't have \( A \text{ arc } B \text{ arc } C \).

\begin{align*}
\text{a arc b.} \\
\text{b arc c.} \\
\text{c arc d.} \\
\text{c arc e.}
\end{align*}

\begin{align*}
\text{path}(A,B) & : - A \text{ arc B.} \\
\text{path}(A,B) & : - A \text{ arc X, path}(X,B).
\end{align*}
Countdown
Countdown \([25, 50, 75, 100, 3, 6]\), Target 952

Start with 6 values: \([25, 50, 75, 100, 3, 6]\)

Remove any 2 values (e.g. \([75, 3]\)) and generate symbolic formula for this pair, add to head of remaining list, e.g.

\([75 + 3], 25, 50, 100, 6\] (note list now of length 5)

If head of list evaluates to 952: SUCCESS

else repeat, e.g. new pair \([75 + 3], 100\], (leaves \([25, 50, 6]\))

generate new operator for pair (e.g. +): \([75 + 3] + 100, 25, 50, 6\] (length 4)
countdown([Soln|_], Target, Soln) :- eval(Soln, Target).

countdown(L, Target, Soln) :-
    choose(2, L, [A, B], R), arithop(A, B, C),
    countdown([C|R], Target, Soln).

generate pair,
generate arithmetic op on pair
Solution?
generate pair
generate arithmetic op on pair
Solution?
generate pair
generate arithmetic op on pair
Solution?
...

Countdown [ 25, 50, 75, 100, 3, 6 ], Target 952
choose

% choose(N, List, Chosen, Remaining)
choose(0,L,[],L).
choose(N,[H|T],[H|C], Remaining) :- N > 0, M is N-1, choose(M,T,C,Remaining).
choose(N,[H|T],Chosen, [H|S]) :- N > 0, choose(N,T,Chosen,S).
Base case - choose zero from list L, Chosen = [ ], Remaining = L.
choose

% choose(N, List, Chosen, Remaining)
choose(0,L,[],L).
choose(N,[H|T],[H|C], Remaining) :- N > 0, M is N-1, choose(M,T,C,Remaining).
choose(N,[H|T],Chosen, [H|S]) :- N > 0, choose(N,T,Chosen,S).

Base case - choose zero from a list L, Chosen = [], Remaining = L.

First recursive case: choose Head, choose N-1 from Tail
choose

% choose(N, List, Chosen, Remaining)
choose(0,L,[],L).
choose(N,[H|T],[H|C], Remaining) :- N > 0, M is N-1, choose(M,T,C,Remaining).
choose(N,[H|T],Chosen, [H|S]) :- N > 0, choose(N,T,Chosen,S).

Base case - choose zero from a list L, Chosen = [], Remaining = L.

First recursive case: choose Head, choose N-1 from Tail

Seconds recursive case: ignore Head, choose N from Tail, Remaining = H + remaining from tail.
choose

An aside/caution regarding functional support...

% choose(N, List, Chosen, Remaining)
choose(0,L,[],L).
choose(N,[H|T],[H|C], Remaining) :- N > 0, choose(N-1,T,C,Remaining).
choose(N,[H|T],Chosen, [H|S]) :- N > 0, choose(N,T,Chosen,S).

E.g. also:

... , take(max(L), L, Remaining) , ...
An aside/caution regarding functional support...

% choose(N, List, Chosen, Remaining)
choose(0,L,[],L).
choose(N,[H|T],[H|C], Remaining) :- N > 0, M is N-1, choose(M,T,C,Remaining).
choose(N,[H|T],Chosen, [H|S]) :- N > 0, choose(N,T,Chosen,S).

E.g. also:

... , max(L,M), take(M, L, Remaining) , ...

Does choose look familiar to you?
% choose(N, List, Chosen, Remaining)
choose(0,L,[],L).
choose(N,[H|T],[H|C], Remaining) :- N > 0, M is N-1, choose(M,T,C,Remaining).
choose(N,[H|T],Chosen,[H|S]) :- N > 0, choose(N,T,Chosen,S).

For our purposes choose/4 could be choose/3...

choose is basically take:  -- I've swapped arguments around, keeping you on your toes...

take(H,[H|T],T).
take(X,[H|T],[H|R]) :- take(X,T,R).
Alternative version of `choose`

% `choose(N, List, Chosen, Remaining)`
choose(0, L, [], L).
choose(N,[H|T], [H|C], Remaining) :- N > 0, M is N-1, choose(M,T,C,Remaining).
choose(N,[H|T], Chosen, [H|S]) :- N > 0, choose(N,T,Chosen,S).

`choose` is basically `take`:

take(H, [H|T], T).
take(X, [H|T], [H|R]) :- take(X,T,R).

E.g. we can write a `take_list(A,B,C)`:

% `take_list(+A,+B,-C)` succeeds if list C is the remaining elements from B after removing list A.
% call with A instantiated to a list of variables, and B ground.
take_list([], L, L).
take_list([H|T],L,R) :- take(H,L,LR), take_list(T, LR, R).

?- take_list([A,B], [a,b,c], L).
A=a, B=b, L = [c]
eval : reducing arithmetic terms to a number.

countdown([Soln|_],Target, Soln) :- eval(Soln,Target).

countdown(L,Target,Soln) :- choose(2,L,[A,B],R), arithop(A,B,C), countdown([C|R],Target,Soln).

generate pair,
generate arithmetic op on pair
Solution?
generate pair
generate arithmetic op on pair
Solution?
generate pair
generate arithmetic op on pair
Solution?
...

eval : reducing arithmetic terms

% eval(+ArithTerm, -N)
eval(A,A) :- number(A).

I'm showing an alternative to Andy's plus(A,B) etc. terms, simply to show infix operators +, -, *, / which already conveniently have the required precedence.

Can you spot anything here?
eval : reducing arithmetic terms

% eval(+ArithTerm, -N)
 eval(A,A) :- number(A).

I'm showing an alternative to Andy's plus(A,B) etc. terms, simply to show infix operators +, -, *, / which already conveniently have the required precedence.

? Did you spot this alternative implementation:

  eval(ArithTerm, N) :- N is ArithTerm.
arithop - generating arithmetic expressions

% arithop(+A, +B, -ArithTerm)
arithop(A,B,A+B).
arithop(A,B,A-B) :- eval(A,D), eval(B,E), D>E.
arithop(B,A,A-B) :- eval(A,D), eval(B,E), D>E.
arithop(A,B,A*B) :- eval(A,D), D \== 1, eval(B,E), E \== 1.
arithop(B,A,A*B) :- eval(B,E), E \== 1, E \== 0, eval(A,D), 0 is D rem E.
arithop(A,B,A/B) :- eval(B,E), E \== 1, E \== 0, eval(A,D), 0 is D rem E.
arithop(B,A,A/B) :- eval(B,E), E \== 1, E \== 0, eval(A,D), 0 is D rem E.

We're only generating arithmetic terms relevant the the puzzle, i.e. we're using the result of the eval to check the term.

* There's a minor detail/choice here, whether the 'choose' generates both pairs (e.g. 3,4 and 4,3) or this can be provided by arithop as we are doing here.
Countdown - alternative version of countdown/3

Current version:
countdown([Soln|_], Target, Soln) :- eval(Soln, Target).

countdown(L, Target, Soln) :-
    choose(2, L, [A, B], R),
    arithop(A, B, C),
    countdown([C|R], Target, Soln).
countdown([Soln|_], Target, Soln) :- eval(Soln, Target).

countdown(L, Target, Soln) :- choose(2, L, [A, B], R),
                         arithop(A, B, C),
                         countdown([C|R], Target, Soln).

test(Soln, Target, Soln) :- eval(Soln, Target).

countdown(L, Target, Soln) :- take_list([A, B], L, R),
                              arithop(A, B, C),
                              ( test(C, Target, Soln); countdown([C|R], Target, Soln) ).
countdown([Soln|_], Target, Soln) :- eval(Soln, Target).

countdown(L, Target, Soln) :-
    choose(2, L, [A, B], R),
    arithop(A, B, C),
    countdown([C|R], Target, Soln).

test(Soln, Target, Soln) :- eval(Soln, Target).

countdown(L, Target, Soln) :-
    take_list([A, B], L, R),
    arithop(A, B, C),
    test_or_calc(C, Target, Soln, R).

test_or_calc(C, Target, Soln, _) :- test(C, Target, Soln).

test_or_calc(C, Target, Soln, R) :- countdown([C|R], Target, Soln).
countdown([Soln|_], Target, Soln) :- eval(Soln, Target).

countdown(L, Target, Soln) :- choose(2, L, [A, B], R),
  arithop(A, B, C),
  countdown([C|R], Target, Soln).

test(Soln, Target, Soln) :- eval(Soln, Target).

countdown(L, Target, Soln) :- take_list([A, B], L, R),
  arithop(A, B, C),
  ( test(C, Target, Soln) ; countdown([C|R], Target, Soln) ).
The whole point of this section is that you understand *how/why* to apply iterative deepening, rather than assume a specific implementation.

test(Soln,Target,Soln) :- eval(Soln,Target).

countdown(L,Target,Soln) :- take_list([A,B], L, R),
    arithop(A,B,C),
    ( test(C, Target, Soln) ;
        countdown([C|R],Target, Soln) ).
diff(A,B,Diff) :- Delta is A - B, (Delta < 0 , Diff is -Delta ; Delta >= 0, Diff is Delta).

test(Soln,Target,Soln, Threshold) :- eval(Soln,Result), diff(Result,Target,Diff), Diff =< Threshold.

countdown(L,Target,Soln, Threshold) :- take_list([A,B], L, R), arithop(A,B,C), ( test(C, Target, Soln, Threshold) ; countdown([C|R],Target, Soln, Threshold) ).

We add a 'Threshold' to the search clause, implement a 'diff' function, test succeeds within bounds.

Diff =< Threshold: the approach is slightly different here than in the video (both are valid) - we are asking for solutions within a 'distance' from the exact answer (not at an exact distance).
diff(A,B,Diff) :- Delta is A - B, (Delta < 0 , Diff is -Delta ; Delta >= 0, Diff is Delta).

test(Soln,Target,Soln, Threshold) :- eval(Soln,Result), diff(Result,Target,Diff), Diff =< Threshold.

countdown(L,Target,Soln, Threshold) :- take_list([A,B], L, R),
            arithop(A,B,C),
            ( test(C, Target, Soln, Threshold) ;
              countdown([C|R],Target, Soln, Threshold) ).

:- countdown([25,50,75,100,3,6],952,Soln,5)
diff(A,B,Diff) :- Delta is A - B, (Delta < 0, Diff is -Delta ; Delta >= 0, Diff is Delta).

test(Soln,Target,Soln,Threshold,Diff) :- eval(Soln,Result), diff(Result,Target,Diff), Diff =< Threshold.

countdown(L,Target,Soln,Threshold,Diff) :- take_list([A,B], L, R),
arithop(A,B,C),
( test(C, Target, Soln, Threshold, Diff) ;
countdown([C|R],Target, Soln, Threshold, Diff) ).
diff(A,B,Diff) :- Delta is A - B, (Delta < 0 , Diff is -Delta ; Delta >= 0, Diff is Delta).
test(Soln,Target,Soln,Threshold,Diff) :- eval(Soln,Result), diff(Result,Target,Diff), Diff =< Threshold.
countdown(L,Target,Soln,Threshold,Diff) :- take_list([A,B], L, R), arithop(A,B,C), ( test(C, Target, Soln, Threshold, Diff) ; countdown([C|R],Target, Soln, Threshold, Diff) ).

:. countdown([25,50,75,100,3,6],952,Soln,5,Diff)

..EXAMPLE
Countdown  Iterative Deepening

Find "simpler" solutions first, then try harder...
Countdown

Iterative Deepening - Conclusion

diff(A,B,Diff) :- Delta is A - B, (Delta < 0 , Diff is -Delta ; Delta >= 0, Diff is Delta).

test(Soln,Target,Soln, Threshold, Diff) :- eval(Soln,Result), diff(Result,Target,Diff), Diff =< Threshold.

countdown(L,Target,Soln, Threshold, Diff) :- take_list([A,B], L, R),
           arithop(A,B,C),
           ( test(C, Target, Soln, Threshold, Diff) ;
             countdown([C|R],Target, Soln, Threshold, Diff) ).

Required Threshold  Actual Difference

:- countdown([25,50,75,100,3,6],952,Soln,5, Diff)

Summary: use-case can be "find solution within threshold, check difference, find better solution ..."

Also as video: closest(L, Target, Soln, Threshold) :- range(0,100,Threshold), solve2(L,Target,Soln,Threshold).
Graph Search

Problem statement
Graph Search

Convert to graph...
Graph Search

Sample implementation (simple, given graph)

route(a,g).
route(g,l).
route(l,s).

... travel(A,A).
travel(A,C) :- route(A,B),travel(B,C).

solve :- start(A),finish(B), travel(A,B).
Graph Search

:- op(700, xfx, arc).
a arc g.
a arc b.
b arc c.
b arc h.
c arc d.
d arc i.
d arc j.
g arc f.
g arc l.
h arc o.
i arc p.
j arc r.
l arc s.
p arc q.
r arc u.

:- op(700, xfx, path).
X path Y :- X arc Y.
X path Y :- X arc W, W path Y.

route(a,g).
route(g,l).
route(l,s).

... travel(A,A).
travel(A,C) :- route(A,B), travel(B,C).
solve :- start(A), finish(B), travel(A,B).

arcs(a,[b,g]).
arcs(b,[c,h]).
arcs(c,[d]).
arcs(d,[i,j]).
arcs(g,[l,f]).
arcs(h,[o]).
arcs(i,[p]).
arcs(j,[r]).
arcs(l,[s]).
arcs(p,[q]).
arcs(r,[u]).

X arc Y :- arcs(X,Nodes), member(Y,Nodes).
Graph Search

Accumulating the path (or cost...):

:- op(700, xfx, arc).
a arc g.
a arc b.
b arc c.
b arc h.
c arc d.
d arc i.
d arc j.
g arc f.
g arc l.
h arc o.
i arc p.
j arc r.
l arc s.
p arc q.
r arc u.

% path(+Start,+Finish,-Path) succeeds if...
path(Start,Finish,Path) :- path_acc(Start,Finish,[],Path).

% path_acc(+Start,+Finish,+PathSoFar,-FullPath)
path_acc(X,Finish,Acc,Path) :- X arc Finish,
reverse([Finish,X|Acc],Path).
path_acc(X,Finish,Acc,Path) :- X arc Z,
path_acc(Z,Finish,[X|Acc],Path).

(1) Base case:
(2) Recursive case:
Next time

Videos

Difference

Empty difference lists

Difference list example
Q: you generally put the base case rule first e.g. Split([], [], []) - wouldn't it be more efficient to put this last since it is less likely? (fewer unifications)
Q: you generally put the base case rule first e.g. Split([], [], []) - wouldn't it be more efficient to put this last since it is less likely? (fewer unifications)

A: you would make a small saving if you only wanted one answer but more answers were possible. But you would still have all the choice points. Remember that order often matters when you have cut.
Q: Do we need to be able to compare Prolog to ML and functional programming? As a third year 50%er that was all a while ago...
Q: Do we need to be able to compare Prolog to ML and functional programming? As a third year 50%er that was all a while ago...

A: I won’t ask you to write ML in the exam. (But I would expect you to recall the concepts of the ML course as a general principle - what’s the point of your degree otherwise?)
Q: What is the underlying difference between a rule and a compound term? Same syntax right?
Q: What is the underlying difference between a rule and a compound term? Same syntax right?

A: a compound term is a ‘term’ in first order logic, a rule is ‘formula’ in first order logic.
Q: Is single cut rule bad practice?

\begin{verbatim}
last(H,[H]).
last(X,[_|T]) :- last(X,T).
This pointlessly backtracks after finding the answer. So change axiom to: last(H,[H]) :- !.
\end{verbatim}
Q: Is single cut rule bad practice?

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

This pointlessly backtracks after finding the answer. So change axiom to: `lst(H,[H]) :- !.`

A: It's fine to put a cut on a fact. The thing to avoid is putting one everywhere!
Next time

Videos

Difference

Empty difference lists

Difference list example
Challenge: Write a tic-tac-toe (noughts and crosses) AI

What’s the first step?
Challenge: Write a tic-tac-toe (noughts and crosses) AI

What predicate will you write and when will it succeed

% nextMove(Before,Player,After) succeeds if After represents the % next state of the board after Player has made a move from state % Before

Next step?
Challenge: Write a tic-tac-toe (noughts and crosses) AI

Choose a representation for the board...
Challenge: Write a tic-tac-toe (noughts and crosses) AI

Suggestion: represent each board position as a number 1 to 9, represent the state of the board as the list of moves that have been made, e.g. [move(5,x),move(1,o)].
Challenge: Write a tic-tac-toe (noughts and crosses) AI

Now try to implement `nextMove(\text{Before}, \text{Player}, \text{After})`
Represent moves as `move(\text{Position}, \text{Player})`
Represent the game state as a list of moves that have been made
Challenge: Write a tic-tac-toe (noughts and crosses) AI

```prolog
pos(Index) :- member(Index,[1,2,3,4,5,6,7,8,9]).

used(I,[move(I,_)|_]).
used(I,[_|T]) :- used(I,T).

nextMove(Before,P,[move(Index,P)|Before]) :-
    pos(Index), \+used(Index,Before).
```
How could we make it smarter?

Teach it heuristics about good moves

- Prefer a corner at the start
- Take the middle if the corner is gone
- Win if you can
- Block the other player from winning if you can