Prolog
Programming in Logic

Lecture #3

Ian Lewis, Andrew Rice

Today's discussion

Videos:

- Arithmetic - ‘is’, space efficiency, Last Call Optimisation, ACCUMULATORS
- Backtracking

From last time...

?- [zebra].

true ◄what's this ?

?-

Because:

(1) '?-' is the QUERY prompt

(2) [zebra]. is syntactic sugar for consult(zebra).

(3) The query consult(zebra) ***succeeds*** (aka returns true) (?- 1 = 1. succeeds)

(4) With a normal query, that would be the end, but consult is an extra-logical predicate with a side-effect of updating the internal database of clauses.

From last time...

...

Arithmetic: Which of these are true statements?

1. 2 is 1+1
2. 2 is +(1,1)
3. 1+1 is 1+1
4. A is 1+1, A = 2
5. 1+1 is A, A = 2

RHS ground numeric expression which is ‘reduced’ to a constant.

LHS constant or variable

Arithmetic: Which of these are true statements?

1. 2 is 1+1
2. 2 is +(1,1)
3. 1+1 is 1+1
4. A is 1+1, A = 2
5. 1+1 is A, A = 2

RHS ground numeric expression which is ‘reduced’ to a constant.

LHS constant or variable

A brief aside: Last Call Optimisation

...a space optimization technique, which applies when a predicate is determinate at the point where it is about to call the last goal in the body of a clause.[1]

[1] Sicstus Prolog Manual

Could you apply LCO to this?

```
last([L],L).

last([_|T],L) :- last(T,L).
```

What about:

```
foo(_,hello).

foo(I, W) :- I > 10, J is I - 1, foo(J, W).
```

DETERMINISTIC vs. NON-DETERMINISTIC

When does LCO get applied?

Interpreted Prolog

Easy - it's applied during execution. The interpreter basically avoids allocating a new stack frame when the predicate is determinate at the point that the last clause needs to be checked

Compiled Prolog

Depends how you compiled it. But you can tell statically that LCO is applicable

Does that make it partly determined[1] by the arguments?

It's not determined by the type of the arguments: there's only one type! (everything is a term)

It's not determined by the value of the arguments:

> think about how the search happens
>
> Prolog would need to try the unification to know if it needs to come back

[1] haha!

Wrap up: Last Call Optimisation

applies when a predicate is determinate at the point where it is about to call the last goal in the body of a clause.

Accumulators

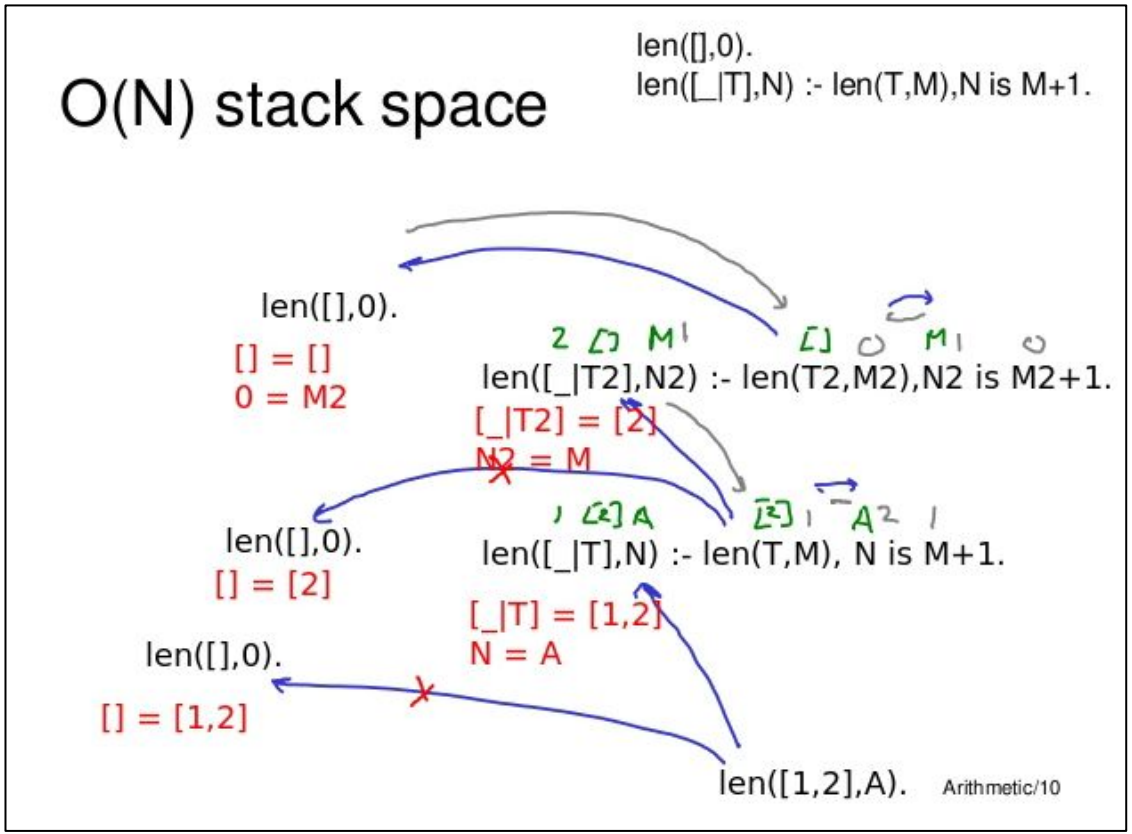
A space-efficient way of passing a partial result through a Prolog computation.

```
% len(L,N) succeeds if N is the length of input list L.
len([],0).
len([_|T],N) :- len(T,M), N is M + 1.
```

The execution stack grows O(n).

Accumulators

```
len([],0).
len([_|T],N) :- len(T,M), N is M + 1.
```



Accumulators - length of a list: len/2 + len_acc/3:

```
% len(L,N) succeeds if N is the length of input list L.
len(L,N) :- len_acc(L,0,N).

% len_acc(L,A,N) succeeds if
%    input L is the remaining list to be counted
%    input A is an accumulated length so far
%    output N is the total length of the original list.
len_acc([],A,A).
len_acc([_|T],A, N) :- A1 is A + 1, len_acc(T,A1,N).
```

Accumulators

```
(1)   len_acc([],A,A).
(2)   len_acc([_|T],A, N) :- A1 is A + 1, len_acc(T,A1,N).

len(L,N) :- len_acc(L,0,N).

?- len([a,b,c],N).
      Call: len_acc([a,b,c],0,N).
```

Accumulators

```
(1)   len_acc([],A,A).
(2)   len_acc([_|T],A, N) :- A1 is A + 1, len_acc(T,A1,N).

len(L,N) :- len_acc(L,0,N).
```

?- len([a,b,c],N).

Call: len_acc([a,b,c],0,N).

Try: (1) - fail

Try: (2) T = [b,c], A = 0, N=N … A1 is 0+1, len_acc([b,c],1,N).

Accumulators

```
(1)   len_acc([],A,A).
(2)   len_acc([_|T],A, N) :- A1 is A + 1, len_acc(T,A1,N).

len(L,N) :- len_acc(L,0,N).

?- len([a,b,c],N).
      Call: len_acc([a,b,c],0,N).
            Try: (1) - fail
            Try: (2) T = [b,c], A = 0, N=N … A1 is 0+1, len_acc([b,c],1,N).
                  Try: (1) - fail
                  Try: (2) T = [c], A = 1, N=N … A1 is 1+1, len_acc([c],2,N).
```

Accumulators

```
(1)   len_acc([],A,A).
(2)   len_acc([_|T],A, N) :- A1 is A + 1, len_acc(T,A1,N).

len(L,N) :- len_acc(L,0,N).

?- len([a,b,c],N).
      Call: len_acc([a,b,c],0,N).
            Try: (1) - fail
            Try: (2) T = [b,c], A = 0, N=N … A1 is 0+1, len_acc([b,c],1,N).
                  Try: (1) - fail
                  Try: (2) T = [c], A = 1, N=N … A1 is 1+1, len_acc([c],2,N).
                        Try: (1) - fail
                        Try: (2) T = [], A = 2, N=N … A1 is 2+1, len_acc([],3,N).
```

Accumulators

```
(1)   len_acc([],A,A).
(2)   len_acc([_|T],A, N) :- A1 is A + 1, len_acc(T,A1,N).

len(L,N) :- len_acc(L,0,N).
```

?- len([a,b,c],N).

Call: len_acc([a,b,c],0,N).

Try: (1) - fail

Try: (2) T = [b,c], A = 0, N=N … A1 is 0+1, len_acc([b,c],1,N).

Try: (1) - fail

Try: (2) T = [c], A = 1, N=N … A1 is 1+1, len_acc([c],2,N).

Try: (1) - fail

Try: (2) T = [], A = 2, N=N … A1 is 2+1, len_acc([],3,N).

Try: (1) []=[], A = 3, 3=N.

+ with LCO...

Accumulators: List reverse - another (classic) example

```
% rev(L1,L2) succeeds if list L2 is the reverse of input list L1
rev([], []).
rev([H|T], L2) :-   rev(T, Trev), append(Trev, [H], L2).


% built-in append:
?- append([a,b,c],[1,2,3],L)
L = [a,b,c,1,2,3]
```

Lecture backtracking… let’s write an ‘append’:

```
% Call it app/3 to avoid built-in append:
?- app([a,b,c],[1,2,3],L)
L = [a,b,c,1,2,3]
```

Pause .. think .. think … think

Append: (1) Comment

```
% app(L1,L2,L3) succeeds if
%     list L3 is the concatenation of lists L1 and L2
```

Append: (2) base case

```
% app(L1,L2,L3) succeeds if
%     list L3 is the concatenation of lists L1 and L2
app([],L,L).
```

Append: (3) Recursive case

```
% app(L1,L2,L3) succeeds if
%     list L3 is the concatenation of lists L1 and L2
app([],L,L).
app([H|T],L1,[H|L2]) :- app(T,L1,L2).
```

?- app([a,b,c],[1,2,3],L).

L = [a,b,c,1,2,3].

?- app(X,[1,2,3],[a,b,c,1,2,3]) or app(X,Y,[a,b,c]).

???

?- app([a,b,c],[1,X,3],L).

???

Accumulators: List reverse - another (classic) example

```
% rev(L1,L2) succeeds if list L2 is the reverse of input list L1
rev(L1, L2) :- rev_acc(L1, [], L2).

% rev_acc(L1, ListAcc, L2) succeeds if L2 is the reverse of
% input list L1 pre-pended onto ListAcc.
% For empty list, ListAcc holds reverse of original list.
rev_acc([], ListAcc, ListAcc).
rev_acc([H|T], ListAcc, L2) :-  rev_acc(T, [H|ListAcc], L2).
```

Can use inductive reasoning, LCO appies.

Accumulators: List reverse - another (classic) example

Can step through as with len_acc before:

Accumulators

```
(1)   len_acc([],A,A).
(2)   len_acc([_|T],A, N) :- A1 is A + 1, len_acc(T,A1,N).

len(L,N) :- len_acc(L,0,N).


?- len([a,b,c],N).
      Call: len_acc([a,b,c],0,N).
            Try: (1) - fail
            Try: (2) T = [b,c], A = 0, N=N … A1 is 0+1, len_acc([b,c],1,N).
                  Try: (1) - fail
                  Try: (2) T = [c], A = 1, N=N … A1 is 1+1, len_acc([c],2,N).
                        Try: (1) - fail
                        Try: (2) T = [], A = 2, N=N … A1 is 2+1, len_acc([],3,N).
                              Try: (1) []=[], A = 3, 3=N.
```

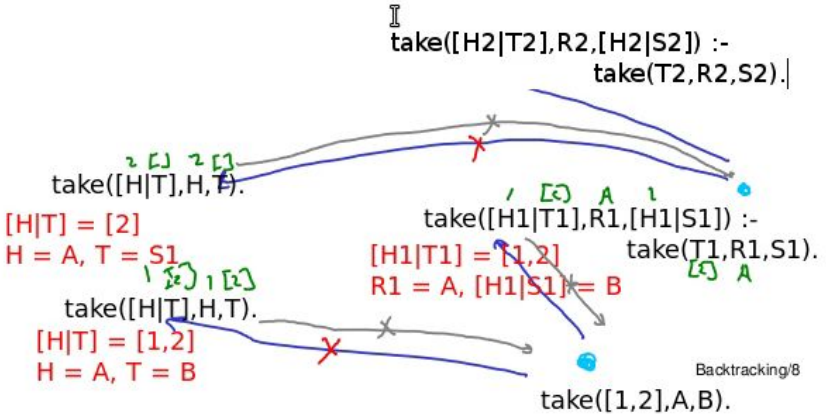
Backtracking

```
% take(L1,X,L2) succeeds if
%     list L2 is the input list L1 omitting element X.
take([H|T],H,T).
take([H|T],X,[H|L]) :- take(T,X,L).
```

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

Backtracking take again

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



Backtracking

(1) take([H|T],H,T).
(2) take([H|T],X,[H|L]) :- take(T,X,L).

```
[trace] ?- take([a,b,c],X,L).
   Call: (8) take([a, b, c], _4088, _4090) ? creep
   Exit: (8) take([a, b, c], a, [b, c]) ? creep
X = a,
L = [b, c] ;
   Redo: (8) take([a, b, c], _4088, _4090) ? creep
   Call: (9) take([b, c], _4088, _4356) ? creep
   Exit: (9) take([b, c], b, [c]) ? creep
   Exit: (8) take([a, b, c], b, [a, c]) ? creep
X = b,
L = [a, c] ;
   Redo: (9) take([b, c], _4088, _4356) ? creep
   Call: (10) take([c], _4088, _4362) ? creep
   Exit: (10) take([c], c, []) ? creep
   Exit: (9) take([b, c], c, [b]) ? creep
   Exit: (8) take([a, b, c], c, [a, b]) ? creep
X = c,
L = [a, b] ;
   Redo: (10) take([c], _4088, _4362) ? creep
   Call: (11) take([], _4088, _4368) ? creep
   Fail: (11) take([], _4088, _4368) ? creep
   Fail: (10) take([c], _4088, _4362) ? creep
   Fail: (9) take([b, c], _4088, _4356) ? creep
   Fail: (8) take([a, b, c], _4088, _4090) ? creep
false.
```

```
?- take([a,b,c],X,L).
      Call: take([a, b, c], X, L)
      Exit: take([a, b, c], a, [b, c])
X = a,
L = [b, c] ;
      Redo: take([a, b, c], X, L)
            Call: take([b, c], X, L1)
            Exit: take([b, c], b, [c])
      Exit: take([a, b, c], b, [a, c])
X = b,
L = [a, c] ;
            Redo: take([b, c], X, L1)
                  Call: take([c], X, L2)
                  Exit: take([c], c, [])
            Exit: take([b, c], c, [b])
      Exit: take([a, b, c], c, [a, b])
X = c,
L = [a, b] ;
                  Redo: take([c], X, L2)
                        Call: take([], X, L3)
                        Fail: take([], X, L3)
                  Fail: take([c], X, L2)
            Fail: take([b, c], X, L1)
      Fail: take([a, b, c], X, L)
false.
```

Backtracking

```
% take/3
take([H|T],H,T).
take([H|T],X,[H|L]) :- take(T,X,L).

% take_path/4
(1)    take_path([H|T],H,T,[1]).
(2)    take_path([H|T],X,[H|L],[2|Path]) :- take_path(T,X,L,Path).
```

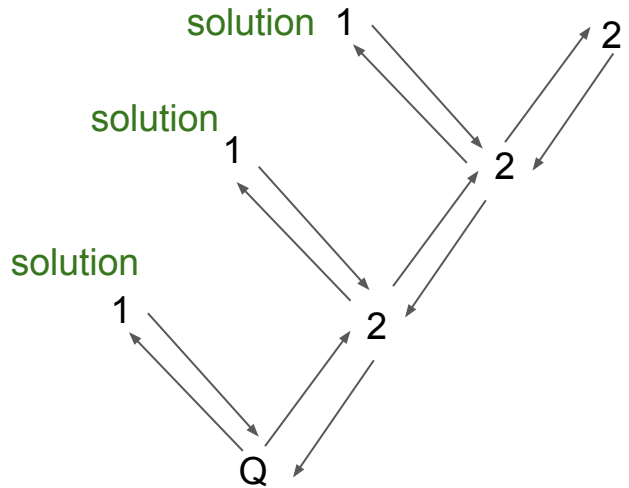
```
?- take_path([a,b,c],X,L,Path).

X = a, L = [b, c], Path = [1] ;

X = b, L = [a, c], Path = [2, 1] ;

X = c, L = [a, b], Path = [2, 2, 1] ;

false.
```



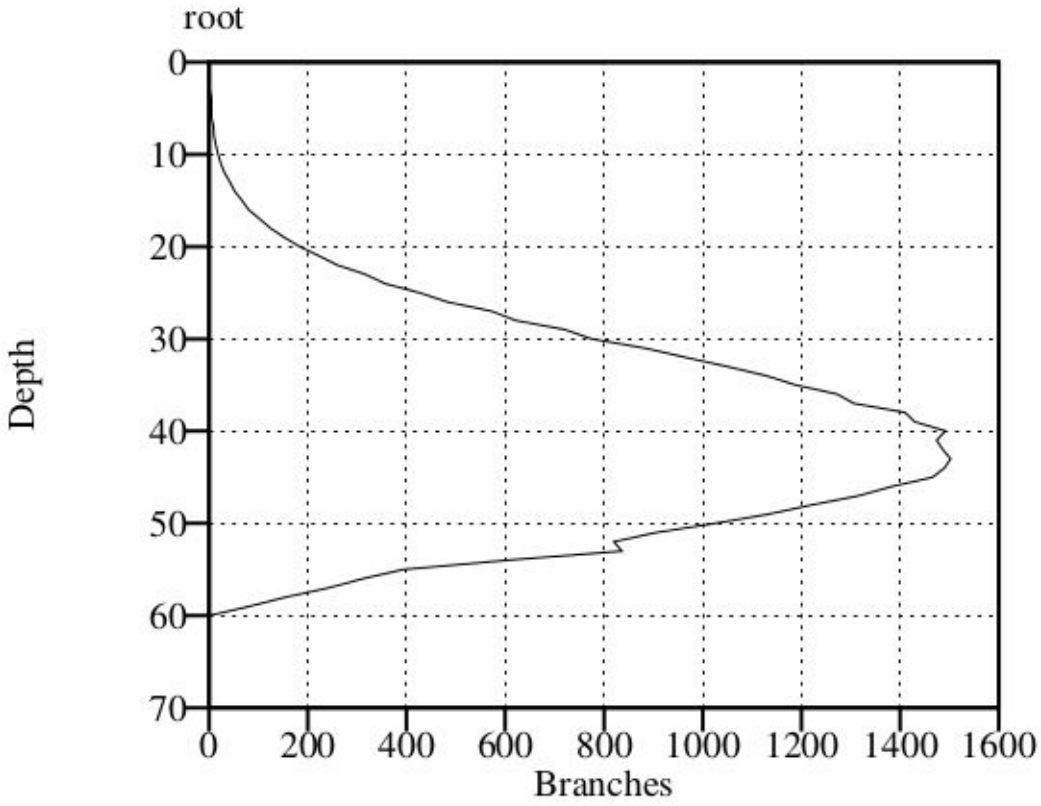


Figure 3.1: Search tree for 8 queens problem.

Next time

Videos

- Generate and Test - ESSENTIAL PROLOG
- Symbolic evaluation of arithmetic - just try and enjoy it...