Prolog
Programming in Logic

Paper 7 Computer Science

Part 1B and Part II 50%

Ian Lewis, Andrew Rice

Agenda for this lecture

1) Aims & Objectives for the course
2) What's the point?
3) View Video #1 - "Prolog Basics"
4) Recap: Programming style, program structure, terms, unification
5) Course outline
6) Success vs. Failure in Prolog - life lessons

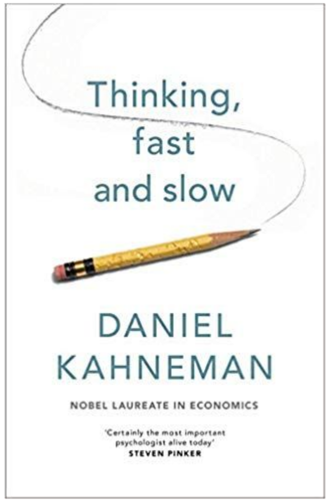
Aims

1. Introduce programming in the Prolog Language
2. A different programming style
3. Solve 'real' problems
4. Practical experimentation encouraged

Objectives

1. Understand the powerful capabilities of 'pure' Prolog: term structure, facts, rules and queries, unification.
2. Know how to model the backtracking behaviour of Prolog program execution, and recognize it as depth first, left-to-right search.
3. Appreciate the unique perspective Prolog gives to problem solving and algorithm design.
4. Understand how larger programs can be created using the basic programming techniques used in this course.

Why study Prolog?



Why study Prolog?

- In an imperative science, know and cherish the ***declarative approach***

If you have a fact: taller(andy, ian). you are DECLARING, or ASSERTING, a relationship “taller” to hold between atoms “andy” and “ian”.

You can declare taller as an infix operator: op(500, xfx, taller).

Hence: andy taller ian.

?- andy taller X.

X = ian

Why study Prolog?

<table>
<tr><td>1200</td><td>xfx</td><td>-->, :-</td></tr>
<tr><td>1200</td><td>fx</td><td>:-, ?-</td></tr>
<tr><td>1150</td><td>fx</td><td>dynamic, discontiguous, initialization, meta_predicate, module_transparent, multifile, public, thread_local, thread_initialization, volatile</td></tr>
<tr><td>1100</td><td>xfy</td><td>;, |</td></tr>
<tr><td>1050</td><td>xfy</td><td>->, *-></td></tr>
<tr><td>1000</td><td>xfy</td><td>,</td></tr>
<tr><td>990</td><td>xfx</td><td>:=</td></tr>
<tr><td>900</td><td>fy</td><td>\+</td></tr>
<tr><td>700</td><td>xfx</td><td><, =, =.., =@=, \=@=, =:=, =<, ==, =\=, >, >=, @<, @=<, @>, @>=, \=, \==, as, is, >:<, :<</td></tr>
<tr><td>600</td><td>xfy</td><td>:</td></tr>
<tr><td>500</td><td>yfx</td><td>+, -, /\, \/, xor</td></tr>
<tr><td>500</td><td>fx</td><td>?</td></tr>
<tr><td>400</td><td>yfx</td><td>*, /, //, div, rdiv, <<, >>, mod, rem</td></tr>
<tr><td>200</td><td>xfx</td><td>**</td></tr>
<tr><td>200</td><td>xfy</td><td>^</td></tr>
<tr><td>200</td><td>fy</td><td>+, -, \</td></tr>
<tr><td>100</td><td>yfx</td><td>.</td></tr>
<tr><td>1</td><td>fx</td><td>$</td></tr>
</table>

Table 5 : System operators

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

Why study Prolog?

```
fun fact(1) = 1;
    fact(N) = N * fact(N-1).

fun fact(N) = if (N = 1)
              then 1
              else N * fact(N-1).

fun append([ ],Y) = Y;
    append([X|Xs],Y) = [X|append(Xs,Y)].
```

These are all valid Prolog terms.

“Everything is a relation” (mostly, with a few hairy edges, like arithmetic)

You can write programs about programs.

Why study Prolog?

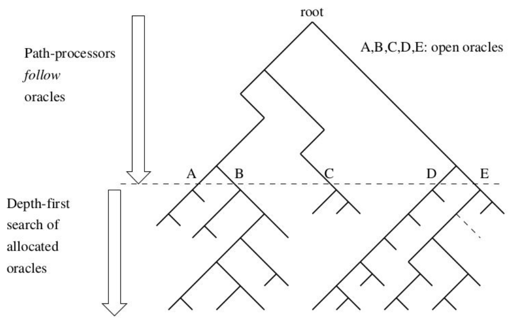


Figure 2.11: Second phase of *breadth-first partitioning.*

You will learn Prolog backtracking can be interpreted as a "search tree".

Actually, given that a Prolog program is itself a valid Prolog term, you can apply *simple* transformations to that program to manipulate the tree. E.g.

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

Goes to:

```
last([X], X, [1]).
last([_|T],X,[2|P]) :- last(T,X,P).
```

```
?- last([a,b,c,d],X,P).
X= d
P = [2,2,2,1]
```

Why study Prolog?

Don't worry about ANY of that.

Just recognize Prolog is all about DECLARING / ASSERTING RELATIONS.

"Everything" in Prolog is a 'meaningless' relation (with a few practical exceptions which are certain to torture you at some point).

Prolog programs are *facts* and *rules*, with *backtracking* providing a powerful search facility.

Unification on its own is an immensely powerful paradigm.

The combination of these 'simple' things can produce very complex behaviour.

Clauses + Unification + Backtracking = Programs.

Video #1: Prolog Basics

Programming Style

IMPERATIVE	DECLARATIVE

`l = [ 1,2,3,4,5 ];`
`sum = 0;`
`for (i=0; i<length(L); i++) {`
`    sum += 1;`
`}`
`return sum;`
	`fun sum([]) = 0`
	`  |  sum(x::xs) = x + sum(xs);`
	
	`sum([],0).`
	`sum([X|L],S) :- sum(L,N), S is N+X.`

Program Structure

Terms = atoms, variables, compound terms (can be infix)

Clauses = Facts + Rules.

Rules = Head :- Body.

Comments = % <anything>

?- = query prompt (often with side effects).

?- [<filename omitting .pl>]. = “consult” a file.

?- [user]. = “consult” user input (uses Prolog “assert”)

Terms

?- X = foo.

X = foo.

?- X = 1.

?- X = a.

?- X = 1.2.

?- X = a(1,a,Y,2).

?- X is 1+2

X = 3. (actually “?- is(X,+(1,2))”)

Compound term:

functor/arity

E.g.

foo(a,b(1),c) -> foo/3

Unification

Unification does not have a “direction”...

Atoms <-> Atoms (and constants)

Variable <-> Anything

Compound Term <-> (same functor/arity) & (arguments unify)

Occurs check e.g. X = a(X).

Unification

Term 1	Term 2	Result after unification
a	a	yes
1.2345	1.2345	yes
foo	X	X=foo
a(b,C)	a(X,p(q))	X=b,C=p(q)
a(b,c)	X(b,c)	

```
:- X = a(Y), Y = 7.
X = a(7),
Y = 7.
```

Backtracking

```
:- [user].
a(1).
a(3).
a(7).
a(9).
^D
:- X = a(Y), Y = 7.
X = a(7),
Y = 7
```

Prolog backtracking is depth-first, left-to-right

Life Lessons #1

Think DECLARATIVE.

`len([],0).` is asserting that “[]” and “0” are associated via the “len” relation.

Queries

`:- len([],X).`

`:- len(X,0).`

are equally reasonable.

Life Lessons #2

Think DEPTH-FIRST LEFT-TO-RIGHT

```
:- [user].
a(1).
a(3).
a(7).
a(9).
^D
:- X = a(Y), Y = 7.
X = a(7),
Y = 7
```

Your program might never end...

Life Lessons #2

Think DEPTH-FIRST LEFT-TO-RIGHT

```
:- [user].
len([],0).
len([_|T],N) :- len(T,M), N is M+1.
^D
:- len([a,b,c,d],N).
N = 4.
:- len(L,0).
```

Your program might never end...

Life Lessons #3

Don't inject FUNCTIONAL support that doesn't exist in Prolog

```
foo(L) :- ... X = max(L) ...
```

Life Lessons #4

Comment each relation:

```
% len(L,N) succeeds if L is a list and N is the length of that list.
len([],0)
. . .
```

Adhere to variable naming and ordering conventions:

If your relation has 'input' and 'output' arguments, say so in your comment AND put the input variables to the left of the output variables in the head of the clause.

Use variable names H and T (or L) for head and tail of a list (or H1, T1). Do not assume all variables have to be a single letter...

Summary:

Think DECLARATIVE.

Think DEPTH-FIRST LEFT-TO-RIGHT.

Comment each relation.

Adhere to variable naming and ordering conventions.

GOOD LUCK

Prolog
Programming in Logic

Lecture #2

Ian Lewis, Andrew Rice

Video/Lesson Recap

Lecture 1:
- **Video 1: Prolog Basics**
  - *Style* (Imperative, Functional, Logic)
  - *Facts*
  - *Queries*
  - *Terms* (constants/atoms, Variables, compound)
  - *Unification*

Lecture 2:
- **Video 2: Logic Puzzle** (zebra) - 5 houses, patterns
  - *Facts* + *Unification++*
- **Video 3: Rules:** Head, Body, Recursion.
- **Video 4: Lists:** [ ], [a], [a|T], [a,b|T]

Any questions from the FIRST lecture and video?

1. Interacting with the Prolog interpreter e.g. **[consult].** , ‘**,**’ and, ‘**;**’ or/next, ‘**.**’ stop.
2. The succeed/true, fail/false Closed-World of Prolog.
3. Prolog terms (atoms, variables, compound).
4. Unification.

Course Outline

1. Introduction, terms, facts, unification
2. Unification. Rules. Lists.
3. Arithmetic, Accumulators, Backtracking
4. Generate and Test
5. Extra-logical predicates (cut, negation, assert)
6. Graph Search
7. Difference Lists
8. Wrap Up.

Today's discussion

Videos:

- Solving a logic puzzle
- Prolog rules
- Lists

Where's the Zebra ?

There are five houses.
The Englishman lives in the red house.
The Spaniard owns the dog.
Coffee is drunk in the green house.
The Ukrainian drinks tea.
The green house is immediately to the right of the ivory house.
The Old Gold smoker owns snails.
Kools are smoked in the yellow house.
Milk is drunk in the middle house.
The Norwegian lives in the first house.
The man who smokes Chesterfields lives in the house next to the man with the fox.
Kools are smoked in the house next to the house where the horse is kept.
The Lucky Strike smoker drinks orange juice.
The Japanese smokes Parliaments.
The Norwegian lives next to the blue house.

Where's the Zebra ?

Represent houses as 5-tuple **(A,B,C,D,E)**.

Represent each house as **house(Nation,Pet,Smokes,Drinks,Colour)**

The Englishman lives in the red house.

can be represented with:

house(british, _, _, _, red).

Note we are structuring our COMPOUND TERMS here, not defining facts/rules. The similarity (and possible confusion) results from Prolog's symmetry between a PROGRAM and a TERM.

Zebra puzzle

```
exists(A,(A,_,_,_,_)).
exists(A,(_,A,_,_,_)).
exists(A,(_,_,A,_,_)).
exists(A,(_,_,_,A,_)).
exists(A,(_,_,_,_,A)).

rightOf(A,B,(B,A,_,_,_)).
rightOf(A,B,(_,B,A,_,_)).
rightOf(A,B,(_,_,B,A,_)).
rightOf(A,B,(_,_,_,B,A)).

middleHouse(A,(_,_,A,_,_)).

firstHouse(A,(A,_,_,_,_)).

nextTo(A,B,(A,B,_,_,_)).
nextTo(A,B,(_,A,B,_,_)).
nextTo(A,B,(_,_,A,B,_)).
nextTo(A,B,(_,_,_,A,B)).
nextTo(A,B,(B,A,_,_,_)).
nextTo(A,B,(_,B,A,_,_)).
nextTo(A,B,(_,_,B,A,_)).
nextTo(A,B,(_,_,_,B,A)).
```

```
:- exists(house(british,_,_,_,red),Houses),
   exists(house(spanish,dog,_,_,_),Houses),
   exists(house(_,_,_,coffee,green),Houses),
   exists(house(ukrainian,_,_,tea,_),Houses),
   rightOf(house(_,_,_,_,green),house(_,_,_,_,ivory),Houses),
   exists(house(_,snail,oldgold,_,_),Houses),
   exists(house(_,_,kools,_,yellow),Houses),
   middleHouse(house(_,_,_,milk,_),Houses),
   firstHouse(house(norwegian,_,_,_,_),Houses),
   nextTo(house(_,_,chesterfields,_,_),house(_,fox,_,_,_),Houses),
   nextTo(house(_,_,kools,_,_),house(_,horse,_,_,_),Houses),
   exists(house(_,_,luckystrike,orangejuice,_),Houses),
   exists(house(japanese,_,parliaments,_,_),Houses),
   nextTo(house(norwegian,_,_,_,_),house(_,_,_,_,blue),Houses),
   exists(house(WaterDrinker,_,_,water,_),Houses),
   exists(house(ZebraOwner,zebra,_,_,_),Houses),
   print(ZebraOwner),nl,
   print(WaterDrinker),nl.
```

Zebra puzzle

(If you haven't watched the video you'll be confused at this point)

1. You're not expected to be able to write that program **yet**.
2. The example uses only **facts** and **UNIFICATION**, without **lists** and **rules**.
3. Typical query term: The Spaniard owns the dog:
   exists(house(spanish,dog,Smokes,Drinks,Colour),Houses).

This 'exists' relation provides essential backtracking.

Zebra puzzle

```
exists(A,(A,_,_,_,_)).
exists(A,(_,A,_,_,_)).
exists(A,(_,_,A,_,_)).
exists(A,(_,_,_,A,_)).
exists(A,(_,_,_,_,A)).

rightOf(A,B,(B,A,_,_,_)).
rightOf(A,B,(_,B,A,_,_)).
rightOf(A,B,(_,_,B,A,_)).
rightOf(A,B,(_,_,_,B,A)).

middleHouse(A,(_,_,A,_,_)).

firstHouse(A,(A,_,_,_,_)).

nextTo(A,B,(A,B,_,_,_)).
nextTo(A,B,(_,A,B,_,_)).
nextTo(A,B,(_,_,A,B,_)).
nextTo(A,B,(_,_,_,A,B)).
nextTo(A,B,(B,A,_,_,_)).
nextTo(A,B,(_,B,A,_,_)).
nextTo(A,B,(_,_,B,A,_)).
nextTo(A,B,(_,_,_,B,A)).
```

```
:-  exists(house(british,_,_,_,red),Houses),
    exists(house(spanish,dog,_,_,_),Houses),
    exists(house(_,_,_,coffee,green),Houses),
    exists(house(ukranian,_,_,tea,_),Houses),
    rightOf(house(_,_,_,_,green),house(_,_,_,_,ivory),Houses),
    exists(house(_,snail,oldgold,_,_),Houses),
    exists(house(_,_,kools,_,yellow),Houses),
    middleHouse(house(_,_,_,milk,_),Houses),
    firstHouse(house(norwegian,_,_,_,_),Houses),
    nextTo(house(_,_,chesterfields,_,_),house(_,fox,_,_,_),Houses),
    nextTo(house(_,_,kools,_,_),house(_,horse,_,_,_),Houses),
    exists(house(_,_,luckystrike,orangejuice,_),Houses),
    exists(house(japanese,_,parliaments,_,_),Houses),
    nextTo(house(norwegian,_,_,_,_),house(_,_,_,_,blue),Houses),
    exists(house(WaterDrinker,_,_,water,_),Houses),
    exists(house(ZebraOwner,zebra,_,_,_),Houses),
    print(ZebraOwner),nl,
    print(WaterDrinker),nl.
```

Zebra puzzle

```
exists(A, (A,_,_,_,_)).
exists(A, (_,A,_,_,_)).
exists(A, (_,_,A,_,_)).
exists(A, (_,_,_,A,_)).
exists(A, (_,_,_,_,A)).

:-  exists(house(british,_,_,_,red),Houses),
    exists(house(spanish,dog,_,_,_),Houses),
     ...
```

Zebra puzzle

```
exists(A, (A,_,_,_,_)).
exists(A, (_,A,_,_,_)).
exists(A, (_,_,A,_,_)).
exists(A, (_,_,_,A,_)).
exists(A, (_,_,_,_,A)).

?-  exists(house(british,_,_,_,red), Houses).
Houses = (house(british,_,_,_,red),_,_,_,_)
```

Zebra puzzle

```
exists(A, (A,_,_,_,_)).
exists(A, (_,A,_,_,_)).
exists(A, (_,_,A,_,_)).
exists(A, (_,_,_,A,_)).
exists(A, (_,_,_,_,A)).

:-  exists(house(british,_,_,_,red), Houses),
A = house(british,_,_,_,red),
Houses = (house(british,_,_,_,red),_,_,_,_)
```

SUCCESS !!

Zebra puzzle

```
exists(A,(A,_,_,_,_)).
exists(A,(_,A,_,_,_)).
exists(A,(_,_,A,_,_)).
exists(A,(_,_,_,A,_)).
exists(A,(_,_,_,_,A)).

:-  exists(house(british,_,_,_,red),Houses),
A = house(british,_,_,_,red),
Houses = (house(british,_,_,_,red),_,_,_,_)
    exists(house(spanish,dog,_,_,_),Houses),
```

Zebra puzzle

```
exists(A,(A,_,_,_,_)).
exists(A,(_,A,_,_,_)).
exists(A,(_,_,A,_,_)).
exists(A,(_,_,_,A,_)).
exists(A,(_,_,_,_,A)).

:-  exists(house(british,_,_,_,red),Houses),
A = house(british,_,_,_,red),
Houses = (house(british,_,_,_,red),_,_,_,_)
    exists(house(spanish,dog,_,_,_),(house(british,_,_,_,red),_,_,_,_)),
```

Zebra puzzle

```
exists(A,(A,_,_,_,_)).
exists(A,(_,A,_,_,_)).
exists(A,(_,_,A,_,_)).
exists(A,(_,_,_,A,_)).
exists(A,(_,_,_,_,A)).

:-  exists(house(british,_,_,_,red),Houses),
A = house(british,_,_,_,red),
Houses = (house(british,_,_,_,red),_,_,_,_)
    exists(house(spanish,dog,_,_,_),(house(british,_,_,_,red),_,_,_,_)),
```

FAIL !!

Zebra puzzle

```
exists(A,(A,_,_,_,_)).
exists(A,(_,A,_,_,_)).
exists(A,(_,_,A,_,_)).
exists(A,(_,_,_,A,_)).
exists(A,(_,_,_,_,A)).
```

BACKTRACK / RETRY

```
:-  exists(house(british,_,_,_,red),Houses),
A = house(british,_,_,_,red),
Houses = (house(british,_,_,_,red),_,_,_,_)
    exists(house(spanish,dog,_,_,_),Houses),
    exists(house(spanish,dog,_,_,_), (house(british,_,_,_,red),_,_,_,_))
    exists(house(spanish,dog,_,_,_), (house(british,_,_,_,red), house(spanish,dog,_,_,_),_,_,_))
```

SUCCESS !!

Backtracking

Note that Prolog backtracked and retried the ‘Spanish’ house assignment, not the ‘British’.

Zebra puzzle

```
exists(A,(A,_,_,_,_)).
exists(A,(_,A,_,_,_)).
exists(A,(_,_,A,_,_)).
exists(A,(_,_,_,A,_)).
exists(A,(_,_,_,_,A)).

rightOf(A,B,(B,A,_,_,_)).
rightOf(A,B,(_,B,A,_,_)).
rightOf(A,B,(_,_,B,A,_)).
rightOf(A,B,(_,_,_,B,A)).

middleHouse(A,(_,_,A,_,_)).

firstHouse(A,(A,_,_,_,_)).

nextTo(A,B,(A,B,_,_,_)).
nextTo(A,B,(_,A,B,_,_)).
nextTo(A,B,(_,_,A,B,_)).
nextTo(A,B,(_,_,_,A,B)).
nextTo(A,B,(B,A,_,_,_)).
nextTo(A,B,(_,B,A,_,_)).
nextTo(A,B,(_,_,B,A,_)).
nextTo(A,B,(_,_,_,B,A)).
```

```
:-  exists(house(british,_,_,_,red),Houses),
    exists(house(spanish,dog,_,_,_),Houses),
    exists(house(_,_,_,coffee,green),Houses),
    exists(house(ukranian,_,_,tea,_),Houses),
    rightOf(house(_,_,_,_,green),house(_,_,_,_,ivory),Houses),
    exists(house(_,snail,oldgold,_,_),Houses),
    exists(house(_,_,kools,_,yellow),Houses),
    middleHouse(house(_,_,_,milk,_),Houses),
    firstHouse(house(norwegian,_,_,_,_),Houses),
    nextTo(house(_,_,chesterfields,_,_),house(_,fox,_,_,_),Houses),
    nextTo(house(_,_,kools,_,_),house(_,horse,_,_,_),Houses),
    exists(house(_,_,luckystrike,orangejuice,_),Houses),
    exists(house(japanese,_,parliaments,_,_),Houses),
    nextTo(house(norwegian,_,_,_,_),house(_,_,_,_,blue),Houses),
    exists(house(WaterDrinker,_,_,water,_),Houses),
    exists(house(ZebraOwner,zebra,_,_,_),Houses),
    print(ZebraOwner),nl,
    print(WaterDrinker),nl.
```

Zebra puzzle

```
exists(A,(A,_,_,_,_)).
exists(A,(_,A,_,_,_)).
exists(A,(_,_,A,_,_)).
exists(A,(_,_,_,A,_)).
exists(A,(_,_,_,_,A)).

rightOf(A,B,(B,A,_,_,_)).
rightOf(A,B,(_,B,A,_,_)).
rightOf(A,B,(_,_,B,A,_)).
rightOf(A,B,(_,_,_,B,A)).

middleHouse(A,(_,_,A,_,_)).

firstHouse(A,(A,_,_,_,_)).

nextTo(A,B,(A,B,_,_,_)).
nextTo(A,B,(_,A,B,_,_)).
nextTo(A,B,(_,_,A,B,_)).
nextTo(A,B,(_,_,_,A,B)).
nextTo(A,B,(B,A,_,_,_)).
nextTo(A,B,(_,B,A,_,_)).
nextTo(A,B,(_,_,B,A,_)).
nextTo(A,B,(_,_,_,B,A)).
```

GENERATE

```
:-  exists(house(british,_,_,_,red),Houses),
    exists(house(spanish,dog,_,_,_),Houses),
    exists(house(_,_,_,coffee,green),Houses),
    exists(house(ukranian,_,_,tea,_),Houses),
    exists(house(_,snail,oldgold,_,_),Houses),
    exists(house(_,_,kools,_,yellow),Houses),
    exists(house(_,_,luckystrike,orangejuice,_),Houses),
    exists(house(japanese,_,parliaments,_,_),Houses),
    exists(house(WaterDrinker,_,_,water,_),Houses),
    exists(house(ZebraOwner,zebra,_,_,_),Houses),
```

TEST

```
    rightOf(house(_,_,_,_,green),house(_,_,_,_,ivory),Houses),
    middleHouse(house(_,_,_,milk,_),Houses),
    firstHouse(house(norwegian,_,_,_,_),Houses),
    nextTo(house(_,_,chesterfields,_,_),house(_,fox,_,_,_),Houses),
    nextTo(house(_,_,kools,_,_),house(_,horse,_,_,_),Houses),
    nextTo(house(norwegian,_,_,_,_),house(_,_,_,_,blue),Houses),
```

Course Outline

1. Introduction, terms, facts, unification
2. Unification. Rules. Lists.
3. Backtracking
4. Generate and Test
5. Extra-logical predicates (cut, negation, assert)
6. Graph Search
7. Difference Lists
8. Wrap Up.

Rules

Q: In the Zebra puzzle, why isn't the `rightOf` fact used help define the `nextTo` fact?

Improving on nextTo

```
nextTo(A,B,(A,B,_,_,_)).
nextTo(A,B,(_,A,B,_,_)).
nextTo(A,B,(_,_,A,B,_)).
nextTo(A,B,(_,_,_,A,B)).
nextTo(A,B,(B,A,_,_,_)).
nextTo(A,B,(_,B,A,_,_)).
nextTo(A,B,(_,_,B,A,_)).
nextTo(A,B,(_,_,_,B,A)).
```

```
nextTo(A,B,Houses) :- rightOf(A,B,Houses).

nextTo(A,B,Houses) :- rightOf(B,A,Houses).
```

Unification recap

Which of these are true statements

1. _ unifies with anything
2. 1+1 unifies with 2
3. prolog unifies with prolog
4. prolog unifies with java

Unification recap

Which of these are true statements

1. _ unifies with anything
2. 1+1 unifies with 2
3. prolog unifies with prolog
4. prolog unifies with java

What's the result of unifying:
cons(1,cons(X)) with
cons(1,cons(2,cons(Y)))

1. False: they don't unify
2. True: they unify
3. True: X is now cons(2,cons(Y))
4. True: X is now cons(1,cons(2,cons(Y)))

What's the result of unifying:
cons(1,cons(X)) with
cons(1,cons(2,cons(Y)))

1. False: they don't unify
2. True: they unify
3. True: X is now cons(2,cons(Y))
4. True: X is now cons(1,cons(2,cons(Y)))

cons(X) cannot unify with cons(2,cons(Y))

for the same reason, cons(X) cannot unify with cons(2,3)

Which of these is a list containing the numbers 1,2,3

1. [1 , 2 , 3]
2. [1 | [2 , 3]]
3. [1 | 2 , 3]
4. [1 , 2 | 3]
5. [1 , 2 | [3]]
6. [1 , 2 , 3 | []]

Which of these is a list containing the numbers 1,2,3

1. [1 , 2 , 3]
2. [1 | [2 , 3]]
3. [1 | 2 , 3]
4. [1 , 2 | 3]
5. [1 , 2 | [3]]
6. [1 , 2 , 3 | []]

Lists, Unification, and program termination

Q: I often write logically-correct code which doesn't terminate. What heuristics can I apply to see if this will happen without running the code?

Q: I often write logically-correct code which doesn't terminate. What heuristics can I apply to see if this will happen without running the code?

A: Its quite hard to do this without using things like arithmetic, but let's look at some examples now and then some more next time.

Does this program terminate?

```
a(X) :- a(X).
```

Does this program terminate?

```
a(X) :- a(X).
```

Yes! Trick question. This program doesn't have any queries in it...

Does this program terminate?

```
a(X) :- a(X).

:- a(1).
```

Does this program terminate?

```
a(X) :- a(X).

:- a(1).
```

NO.
In trying to ‘solve’ or ‘prove’ a(1), Prolog will unify X=1 in the single rule, and then try and prove a(1)...

Does this program terminate?

```
a([]).

a([_|T]) :- a(T).

:- X = <any_finite_list>, a(X).
```

Does this program terminate?

```
a([]).

a([_|T]) :- a(T).

:- X = <any_finite_list>, a(X).
```

Does this program terminate?

```
a([]).
a([_|T]) :- a(T).
:- X = <any_finite_list>, a(X).
```

YES. Recursive call is with shorter list.

More interesting query: `:- a(X).`

What does this print?

```
a([],R) :- print(R), a(R,[]).
a([H|T],R) :- a(T,[H|R]).
:- a([1,2,3],[]).
```

Does this terminate?

```
a([]) :- a([1|X]).
:- a([]).
```

Does this terminate?

```
a([]) :- a([1|X]).
:- a([]).
```

ABSOLUTELY! With fail/false.

In trying to prove `a([])`, Prolog tries to prove `a([1|X])`, and that fails to unify with any fact or rule.

Super-Heuristic - Determinism

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

(1) Call with ?- last([a,b,c],H).
H = c.
(2) Call with ?- last(L, a).

Super-Heuristic - Determinism

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

(1) Call with ?- last([a,b,c],H).
H = c.
(2) Call with ?- last(L, a).
L = [a] ;
L = [_222, a] ;
L = [_333, _222, a] ...

Super-Heuristic - Determinism

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

(1) Call with ?- len([a,b,c],N).
N = 3.
(2) Call with ?- len(L, 0).
L = [ ] ;
?

...

Today's programming challenge - Map colouring

Colour the regions shown below using four different colours so that no touching regions have the same colour.

```
+----+--------+-------+
|    |   C2   |   C3  |
|    +----+---+--+----+
| C1 | C4 | C7   |    |
|    +----+------+    |
|    |           | C8 |
+----+    C6     |    |
| C5 |           |    |
+----+-----------+----+
```

Hint 1: Write down what is true...

You have 4 colours and they are all different…

```
+----+--------+-------+
|    |   C2   |   C3  |
|    +----+---+--+----+
| C1 | C4 | C7   |    |
|    +----+------+    |
|    |           | C8 |
+----+    C6     |    |
| C5 |           |    |
+----+-----------+----+
```

Hint 1: Write down what is true...

You have 4 colours and they are all different…

```
+----+--------+-------+
|    |   C2   |   C3  |
|    +----+---+--+----+
| C1 | C4 | C7   |    |
|    +----+------+    |
|    |           | C8 |
+----+    C6     |    |
| C5 |           |    |
+----+-----------+----+
```

```
diff(red,green).
diff(red,blue).
diff(red,yellow).
diff(green,red).
diff(green,blue).
diff(green,yellow).
```

...etc...

Hint 2: Ask for the answer

What colour does each region need to be so its different to its neighbours

```
+----+--------+-------+
|    |   C2   |   C3  |
|    +----+---+--+----+
| C1 | C4 | C7   |    |
|    +----+------+    |
|    |           | C8 |
+----+    C6     |    |
| C5 |           |    |
+----+-----------+----+
```

```
:
```

Hint 2: Ask for the answer

What colour does each region need to be so its different to its neighbours

```
+----+--------+-------+      :- diff(C1,C5),
|    |   C2   |   C3  |         diff(C1,C2),
|    +----+---+--+----+         diff(C1,C4),
| C1 | C4 | C7   |    |         diff(C1,C6),
|    +----+------+    |         diff(C2,C4),
|    |           | C8 |         diff(C2,C7),
+----+    C6     |    |         ...etc...
| C5 |           |    |
+----+-----------+----+
```

Coloured map

```
+----+--------+-------+
|    |   C2   |   C3  |
|    +----+---+--+----+
| C1 | C4 | C7   |    |
|    +----+------+    |
|    |           | C8 |
+----+    C6     |    |
| C5 |           |    |
+----+-----------+----+
```

Map colours

```
diff(X,Y) :- X \= Y.
color(red).
color(blue).
color(green).
color(yellow).
```

```
ans :- color(C1), color(C2), color(C3), color(C4), color(C5), color(C6), color(C7), color(C8),
  diff(C1,C5),
  diff(C1,C2),
  diff(C1,C4),
  diff(C1,C6),
  diff(C2,C3),
  diff(C2,C4),
  diff(C2,C7),
  diff(C3,C7),
  diff(C3,C8),
  diff(C4,C6),
  diff(C4,C7),
  diff(C5,C6),
  diff(C6,C7),
  diff(C6,C8),
  diff(C7,C8),
  print([C1,C2,C3,C4,C5,C6,C7,C8]).
```

Next time

Videos

- Arithmetic
- Backtracking

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

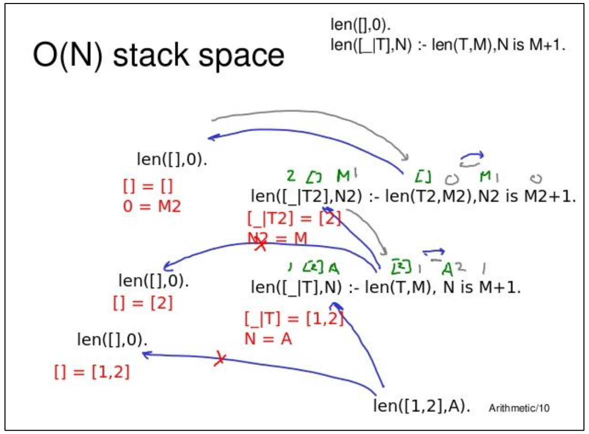
A space-efficient way of passing a partial result through a Prolog computation.

```
% 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.
```

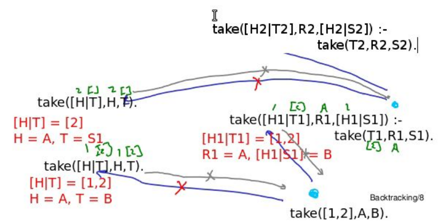
Backtracking

```
% 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).
```

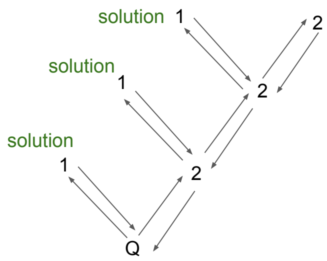
```
[trace] ?- take([a,b,c],X,L).
   Call: (8) take([a, b, c], _4088, _4090) ? creep
   Exit: (8) take([a, b, c], a, [b, c]) ? creep
X = a,
L = [b, c] ;
   Redo: (8) take([a, b, c], _4088, _4090) ? creep
   Call: (9) take([b, c], _4088, _4356) ? creep
   Exit: (9) take([b, c], b, [c]) ? creep
   Exit: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



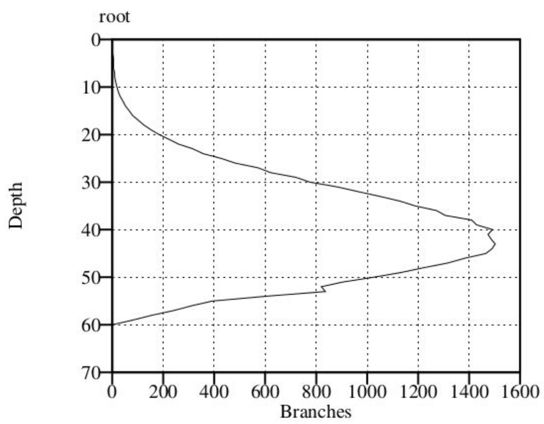


Figure 3.1: Search tree for 8 queens problem.

Next time

Videos

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

Prolog
Programming in Logic

Lecture #4

Ian Lewis, Andrew Rice

Q. What format are the comments?

There's a Prolog convention for comments

% take(+L,-H,-T)
% take succeeds if list T is list L with H removed.

```
take([H|T],H,T).                        % if you take H from a list containing
                                        [H|T] you are left with T

take([H|T],E,[H|R]) :- take(T,E,R).     % you can take E from some list with
                                        head H and tail T leaving a list with
                                        head H and tail R if you can take E
                                        from T leaving R.
```

This are called 'modes'

++	The argument is ground (no variables anywhere).
+	Instantiated but not necessarily ground.
-	The argument is an 'output' argument.
?	Anything.

I've missed some of the classes out in the interests of time. See the full list here: 'Type, mode and determinism declaration headers'
http://www.swi-prolog.org/pldoc/man?section=modes

Q. Why use an accumulator? LCO?

Q. Why use an accumulator? LCO?

A. Space Optimisation.

The 'accumulator version' of a relation may permit LCO (len).

Or, working with lists, the accumulator may allow building a list by adding a 'head' rather than appending.

Q. Is Prolog logic 'incomplete' because it lacks functions?

Q. Is Prolog logic 'incomplete' because it lacks functions?

A. Prolog doesn't lack functions. These are deterministic relations implemented by flattening:

```
fun fact(1) = 1;
    fact(N) = N * fact(N-1).
```

```
fact(1,1).
fact(N,FN) :- N > 1, M is N - 1, fact(M,FM), FN is N * FM.
```

Q: Regarding cut, If we have rule answer :- generate, !, test. would this evaluate to false (& thus miss solutions) if test is not true for the first generated solution?

A: Yes. But we're not talking about cut today...

Today's discussion

Videos:

Generate and Test (Dutch Flag, Sudoku)

Symbolic (Eval)

Course Outline

1. Introduction, terms, facts, unification
2. Unification. Rules. Lists.
3. Arithmetic, Accumulators, Backtracking
4. Generate and Test (Dutch Flag, Sudoku), eval.
5. Extra-logical predicates (cut, negation, assert)
6. Graph Search
7. Difference Lists
8. Wrap Up (..Sudoku..)

Today's discussion

Videos:

Generate and Test (Dutch Flag, Sudoku)

Symbolic (Eval)

Dutch Flag

```
?- dutch_national_flag([blue,red,white,blue],L)

L = [red,white,blue,blue]
```

```
dutch_national_flag(In,Out) :-
                    perm(In,Out),          GENERATE
                    checkColours(Out).     TEST

checkColours(List) :- checkRed(List).

checkRed([red|T]) :- checkRed(T).
checkRed([white|T]) :- checkWhite(T).

checkWhite([white|T]) :- checkWhite(T).
checkWhite([blue|T]) :- checkBlue(T).

checkBlue([blue|T]) :- checkBlue(T).
checkBlue([]).
```

brevity: Flag

LIST CONTAINS AT LEAST ONE OF EACH COLOUR

```
?-
flag([b,r,w,b],L)

L = [r,w,b,b]

...
```

Flag

LIST CONTAINS AT LEAST ONE OF EACH COLOUR

```
?-
flag([b,r,w,b],L)

L = [r,w,b,b]

...
```

```
% checkB(L) succeeds if all list L are blue or L empty.
checkB([b|T]) :- checkB(T).
checkB([]).
```

Flag

LIST CONTAINS AT LEAST ONE OF EACH COLOUR

```
?-
flag([b,r,w,b],L)

L = [r,w,b,b]

...
```

```
% checkB(L) succeeds if all list L are blue or L empty.
checkB([b|T]) :- checkB(T).
checkB([]).

% checkWB(L) succeeds if L is white followed by blue.
checkWB([w|T]) :- checkWB(T).
checkWB([b|T]) :- checkB(T).
```

Flag

LIST CONTAINS AT LEAST ONE OF EACH COLOUR

```
?-
flag([b,r,w,b],L)

L = [r,w,b,b]

...
```

```
% checkB(L) succeeds if all list L are blue or L empty.
checkB([b|T]) :- checkB(T).
checkB([]).

% checkWB(L) succeeds if L is white followed by blue.
checkWB([w|T]) :- checkWB(T).
checkWB([b|T]) :- checkB(T).

% checkRWB succeeds if L is reds.. whites.. blues.
checkRWB([r|T]) :- checkRWB(T).
checkRWB([w|T]) :- checkWB(T).
```

Flag

LIST CONTAINS AT LEAST ONE OF EACH COLOUR

```
?-
flag([b,r,w,b],L)

L = [r,w,b,b]

...
```

```
% checkB(L) succeeds if all list L are blue or L empty.
checkB([b|T]) :- checkB(T).
checkB([]).

% checkWB(L) succeeds if L is white followed by blue.
checkWB([w|T]) :- checkWB(T).
checkWB([b|T]) :- checkB(T).

% checkRWB succeeds if L is reds.. whites.. blues.
checkRWB([r|T]) :- checkRWB(T).
checkRWB([w|T]) :- checkWB(T).

% flag(L1,L2) succeeds if L2 is red-white-blue sorted L1
flag(L1,L2) :-
   perm(L1,L2),        GENERATE
   checkRWB(L2).       TEST
```

Dutch Flag

```
?- dutch_national_flag([blue,red,white,blue],L)

L = [red,white,blue,blue]
```

```
dutch_national_flag(In,Out) :-
   perm(In,Out),          GENERATE
   checkColours(Out).     TEST

checkColours(List) :- checkRed(List).

checkRed([red|T]) :- checkRed(T).
checkRed([white|T]) :- checkWhite(T).

checkWhite([white|T]) :- checkWhite(T).
checkWhite([blue|T]) :- checkBlue(T).

checkBlue([blue|T]) :- checkBlue(T).
checkBlue([]).
```

SUDOKU 8x8

https://en.wikipedia.org/wiki/Sudoku

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

```
puzzle1 :- solve([5,3,_,_,7,_,_,_,_],
                 [6,_,_,1,9,5,_,_,_],
                 [_,9,8,_,_,_,_,6,_],
                 [8,_,_,_,6,_,_,_,3],
                 [4,_,_,8,_,3,_,_,1],
                 [7,_,_,_,2,_,_,_,6],
                 [_,6,_,_,_,_,2,8,_],
                 [_,_,_,4,1,9,_,_,5],
                 [_,_,_,_,8,_,_,7,9]
                ).
```

perm

```
% take(L1,X,L2) succeeds if output list L2 is input list L1 minus element X
take([H|T],H,T).
take([H|T],R,[H|S]) :- take(T,R,S).

% perm(L1,L2) succeeds if output list L2 is a permutation of input list L1
perm([],[]).
perm(A,[R|S]) :- take(A,R,P), perm(P,S).

% gen_digits(L) succeeds if L is a permutation of [1,2,3,4,5,6,7,8,9]
gen_digits(L) :- perm([1,2,3,4,5,6,7,8,9],L).
```

SUDOKU 9x9

```
puzzle1 :- A = [5,3,_,_,7,_,_,_,_],        gen_digits(A),
           B = [6,_,_,1,9,5,_,_,_],        gen_digits(B),
           C = [_,9,8,_,_,_,_,6,_],        gen_digits(C),
           D = [8,_,_,_,6,_,_,_,3],        gen_digits(D),
           E = [4,_,_,8,_,3,_,_,1],        gen_digits(E),
           F = [7,_,_,_,2,_,_,_,6],        gen_digits(F),
           G = [_,6,_,_,_,_,2,8,_],        gen_digits(G),
           H = [_,_,_,4,1,9,_,_,5],        gen_digits(H),
           I = [_,_,_,_,8,_,_,7,9],        gen_digits(I),
                                           Test fixed numbers,
                                           Test the 'boxes',
                                           Test the 'columns',
                                           Print Solution.
```

GENERATE: gen_digits(A), with and without 'seed'

```
puzzle1 :- A = [5,3,1,2,7,4,6,8,9],        gen_digits(A),
           B = [6,_,_,1,9,5,_,_,_],        gen_digits(B),
           C = [_,9,8,_,_,_,_,6,_],        gen_digits(C),
           D = [8,_,_,_,6,_,_,_,3],        gen_digits(D),
           E = [4,_,_,8,_,3,_,_,1],        gen_digits(E),
           F = [7,_,_,_,2,_,_,_,6],        gen_digits(F),
           G = [_,6,_,_,_,_,2,8,_],        gen_digits(G),
           H = [_,_,_,4,1,9,_,_,5],        gen_digits(H),
           I = [_,_,_,_,8,_,_,7,9],        gen_digits(I),
                                           Test fixed numbers,
                                           Test the 'boxes',
                                           Test the 'columns',
                                           Print Solution.
```

```
puzzle1 :- A = [5,3,1,2,7,4,6,8,9],        gen_digits(A),
           B = [6,2,3,1,9,5,4,7,8],        gen_digits(B),
           C = [1,9,8,2,3,4,5,6,7],        gen_digits(C),
           D = [8,1,2,4,6,5,7,9,3],        gen_digits(D),
           E = [4,2,5,8,6,3,7,9,1],        gen_digits(E),
           F = [7,1,3,4,2,5,8,9,6],        gen_digits(F),
           G = [1,6,3,4,5,7,2,8,9],        gen_digits(G),
           H = [2,3,6,4,1,9,7,8,5],        gen_digits(H),
           I = [1,2,3,4,8,5,6,7,9],        gen_digits(I),
                                           Test fixed numbers,
                                           Test the 'boxes',
                                           Test the 'columns',
                                           Print Solution.
```

SUDOKU 9x9

```
puzzle1 :- A = [5,3,1,2,7,4,6,8,9],        gen_digits(A),
           B = [6,2,3,1,9,5,4,7,8],        gen_digits(B),
           C = [1,9,8,2,3,4,5,6,7],        gen_digits(C),
           D = [8,1,2,4,6,5,7,9,3],        gen_digits(D),
           E = [4,2,5,8,6,3,7,9,1],        gen_digits(E),
           F = [7,1,3,4,2,5,8,9,6],        gen_digits(F),
           G = [1,6,3,4,5,7,2,8,9],        gen_digits(G),
           H = [2,3,6,4,1,9,7,8,5],        gen_digits(H),
           I = [1,2,3,4,8,5,6,7,9],        gen_digits(I),
                                           Test fixed numbers,
                                           Test the 'boxes',
                                           Test the 'columns',
                                           Print Solution.
```

SUDOKU 9x9

```
puzzle1 :- A = [5,3,1,2,7,4,6,8,9],        gen_digits(A),
           B = [6,2,3,1,9,5,4,7,8],        gen_digits(B),
           C = [1,9,8,2,3,4,5,6,7],        gen_digits(C),
           D = [8,1,2,4,6,5,7,9,3],        gen_digits(D),
           E = [4,2,5,8,6,3,7,9,1],        gen_digits(E),
           F = [7,1,3,4,2,5,8,9,6],        gen_digits(F),
           G = [1,6,3,4,5,7,2,8,9],        gen_digits(G),
           H = [2,3,6,4,1,9,7,8,5],        gen_digits(H),
           I = [1,2,3,4,8,5,6,7,9],        gen_digits(I),
                                           Test the 'boxes',
                                           Test the 'columns',
                                           Print Solution.
```

SUDOKU 9x9

```
puzzle1 :- A = [5,3,1,2,7,4,6,8,9],        gen_digits(A),
           B = [6,2,3,1,9,5,4,7,8],        gen_digits(B),
           C = [1,9,8,2,3,4,5,6,7],        gen_digits(C),
           D = [8,1,2,4,6,5,7,9,3],        gen_digits(D),
           E = [4,2,5,8,6,3,7,9,1],        gen_digits(E),
           F = [7,1,3,4,2,5,8,9,6],        gen_digits(F),
           G = [1,6,3,4,5,7,2,8,9],        gen_digits(G),
           H = [2,3,6,4,1,9,7,8,5],        gen_digits(H),
           I = [1,2,3,4,8,5,6,7,9],        gen_digits(I),
                                           Test the 'boxes',
                                           Test the 'columns',
                                           Print Solution.
```

SUDOKU 9x9

```
puzzle1 :- A = [5,3,1,2,7,4,6,8,9],        gen_digits(A),
           B = [6,2,3,1,9,5,4,7,8],        gen_digits(B),
           C = [1,9,8,2,3,4,5,6,7],        gen_digits(C),
           D = [8,1,2,4,6,5,7,9,3],        gen_digits(D),
           E = [4,2,5,8,6,3,7,9,1],        gen_digits(E),
           F = [7,1,3,4,2,5,8,9,6],        gen_digits(F),
           G = [1,6,3,4,5,7,2,8,9],        gen_digits(G),
           H = [2,3,6,4,1,9,7,8,5],        gen_digits(H),
           I = [1,2,3,4,8,5,6,7,9],        gen_digits(I),
                                           Test the 'boxes',
                                           Test the 'columns',
                                           Print Solution.
```

SUDOKU 9x9

```
puzzle1 :- A = [5,3,1,2,7,4,6,8,9],        gen_digits(A),
           B = [6,2,3,1,9,5,4,7,8],        gen_digits(B),
           C = [1,9,8,2,3,4,5,6,7],        gen_digits(C),
           D = [8,1,2,4,6,5,7,9,3],        gen_digits(D),
           E = [4,2,5,8,6,3,7,9,1],        gen_digits(E),
           F = [7,1,3,4,2,5,8,9,6],        gen_digits(F),
           G = [1,6,3,4,5,7,2,8,9],        gen_digits(G),
           H = [2,3,6,4,1,9,7,8,5],        gen_digits(H),
           I = [1,2,3,4,8,5,6,7,9],        gen_digits(I),
                                           Test the 'boxes',
                                           Test the 'columns',
                                           Print Solution.
```

SUDOKU 9x9

```
puzzle1 :- A = [5,3,1,2,7,4,6,8,9],
           B = [6,2,3,1,9,5,4,7,8],
           C = [1,9,8,2,3,4,5,6,7],
           D = [8,1,2,4,6,5,7,9,3],
           E = [4,2,5,8,6,3,7,9,1],
           F = [7,1,3,4,2,5,8,9,6],
           G = [1,6,3,4,5,7,2,8,9],
           H = [2,3,6,4,1,9,7,8,5],
           I = [1,2,3,4,8,5,6,7,9],
gen_digits(A),
gen_digits(B),
gen_digits(C),
gen_digits(D),
gen_digits(E),
gen_digits(F),
gen_digits(G),
gen_digits(H),
gen_digits(I),
Test the 'boxes',
Test the 'columns',
Print Solution.
```

```
puzzle1 :-A = [5,3,_,_,7,_,_,_,_],
          B = [6,_,_,1,9,5,_,_,_],
          C = [_,9,8,_,_,_,_,6,_],
          D = [8,_,_,_,6,_,_,_,3],
          E = [4,_,_,8,_,3,_,_,1],
          F = [7,_,_,_,2,_,_,_,6],
          G = [_,6,_,_,_,_,2,8,_],
          H = [_,_,_,4,1,9,_,_,5],
          I = [_,_,_,_,8,_,_,7,9],
          gen_digits(A), A = [5,3,1,2,7,4,6,8,9]
          gen_digits(B),
          gen_digits(C),
          ...
          Test the 'boxes'
          Test the 'columns'
```

```
[5,3,1,2,7,4,6,8,9]
[6,_,_,1,9,5,_,_,_]
[_,9,8,_,_,_,_,6,_]
[8,_,_,_,6,_,_,_,3]
[4,_,_,8,_,3,_,_,1]
[7,_,_,_,2,_,_,_,6]
[_,6,_,_,_,_,2,8,_]
[_,_,_,4,1,9,_,_,5]
[_,_,_,_,8,_,_,7,9]
```

```
puzzle1 :-A = [5,3,_,_,7,_,_,_,_],
          B = [6,_,_,1,9,5,_,_,_],
          C = [_,9,8,_,_,_,_,6,_],
          D = [8,_,_,_,6,_,_,_,3],
          E = [4,_,_,8,_,3,_,_,1],
          F = [7,_,_,_,2,_,_,_,6],
          G = [_,6,_,_,_,_,2,8,_],
          H = [_,_,_,4,1,9,_,_,5],
          I = [_,_,_,_,8,_,_,7,9],
          gen_digits(A), A = [5,3,1,2,7,4,6,8,9]
          Test the 'columns'
          gen_digits(B),
          gen_digits(C),
          ...
          Test the 'boxes'
```

```
[5,3,1,2,7,4,6,8,9]
[6,_,_,1,9,5,_,_,_]
[_,9,8,_,_,_,_,6,_]
[8,_,_,_,6,_,_,_,3]
[4,_,_,8,_,3,_,_,1]
[7,_,_,_,2,_,_,_,6]
[_,6,_,_,_,_,2,8,_]
[_,_,_,4,1,9,_,_,5]
[_,_,_,_,8,_,_,7,9]
```

```
test_digits([5,6,_,8,4,7,_,_,_])

Input argument of VARS and DIGITS

Succeed if DIGITS unique in [1..9]

heads([[a,b,c],
       [d,e,f],
       [g,h,i]], X)
X = [a,d,g]
```

```
[5,3,1,2,7,4,6,8,9]
[6,_,_,1,9,5,_,_,_]
[_,9,8,_,_,_,_,6,_]
[8,_,_,_,6,_,_,_,3]
[4,_,_,8,_,3,_,_,1]
[7,_,_,_,2,_,_,_,6]
[_,6,_,_,_,_,2,8,_]
[_,_,_,4,1,9,_,_,5]
[_,_,_,_,8,_,_,7,9]
```

```
test_digits([5,6,_,8,4,7,_,_,_])

Input argument of VARS and DIGITS

Succeed if DIGITS unique in [1..9]

heads([[a,b,c],
       [d,e,f],
       [g,h,i]], X, Tails)
X = [a,d,g]
Tails = [[b,c],
         [e,f],
         [h,i]]
```

test_digits(L)

```
% test_digits(L) succeeds if:
%    input list L contains digits and/or variables
%    the digits in L are unique from [1..9]
test_digits(L) :- test_set(L,[1,2,3,4,5,6,7,8,9]).

% test_set(L1,L2) succeeds if:
%    input list L1 contains digits and/or variables
%    the digits in L1 are unique in input list L2
test_set([],_).
test_set([H|T], Digits) :- var(H),
                           test_set(T,Digits).
test_set([H|T], Digits) :- ground(H),
                           take(Digits,H,RemainingDigits),
                           test_set(T,RemainingDigits).
```

heads(LL, LHeads, LTails)

```
% heads(LL, LHeads, LTails) succeeds if:
%   LL is an input list of lists
%   LHeads is a list of the heads of each list in LL
%   LTails is LL with the head of each list removed.
heads([],[],[]).
heads([[H|T1]|T],[H|Hs], [T1|Tails]) :- heads(T,Hs,Tails).

:- heads([[1,2,3],
          [4,5,6],
          [7,8,9]],Heads, Tails).
Heads = [1,4,7]
Tails = [[2,3],[5,6],[8,9]]
```

test_cols(Rows)

```
% test_cols(Rows) succeeds if:
%    input Rows is a list of lists of digits and variables
%    the 'columns' of digits and variables contain unique digits from [1..9]
test_cols([[]|_]).
test_cols(Rows) :- heads(Rows,Heads,Tails),
                   test_digits(Heads),
                   test_cols(Tails).
```

```
[5,3,1,2,7,4,6,8,9]
[6,_,_,1,9,5,_,_,_]
[_,9,8,_,_,_,_,6,_]
[8,_,_,_,6,_,_,_,3]
[4,_,_,8,_,3,_,_,1]
[7,_,_,_,2,_,_,_,6]
[_,6,_,_,_,_,2,8,_]
[_,_,_,4,1,9,_,_,5]
[_,_,_,_,8,_,_,7,9]
```

```
[5,3,1,2,7,4,6,9,8]
[6,2,3,1,9,5,4,8,7]
[_,9,8,_,_,_,_,6,_]
[8,_,_,_,6,_,_,_,3]
[4,_,_,8,_,3,_,_,1]
[7,_,_,_,2,_,_,_,6]
[_,6,_,_,_,_,2,8,_]
[_,_,_,4,1,9,_,_,5]
[_,_,_,_,8,_,_,7,9]
```

```
puzzle1 :-A = [5,3,_,_,7,_,_,_,_],
          B = [6,_,_,1,9,5,_,_,_],
          C = [_,9,8,_,_,_,_,6,_],
          D = [8,_,_,_,6,_,_,_,3],
          E = [4,_,_,8,_,3,_,_,1],
          F = [7,_,_,_,2,_,_,_,6],
          G = [_,6,_,_,_,_,2,8,_],
          H = [_,_,_,4,1,9,_,_,5],
          I = [_,_,_,_,8,_,_,7,9],
          gen_digits(A), A = [5,3,1,2,7,4,6,9,8]
          Test the 'columns'
          gen_digits(B), B = [6,2,3,1,9,5,4,8,7]
          Test the 'columns'
          gen_digits(C),
          ...
          Test the 'boxes'
```

```
[5,3,1,2,7,4,6,9,8]
[6,2,3,1,9,5,4,8,7]
[1,9,8,2,3,4,5,6,7]
[8,_,_,_,6,_,_,_,3]
[4,_,_,8,_,3,_,_,1]
[7,_,_,_,2,_,_,_,6]
[_,6,_,_,_,_,2,8,_]
[_,_,_,4,1,9,_,_,5]
[_,_,_,_,8,_,_,7,9]
```

```
puzzle1 :-A = [5,3,_,_,7,_,_,_,_],
          B = [6,_,_,1,9,5,_,_,_],
          C = [_,9,8,_,_,_,_,6,_],
          D = [8,_,_,_,6,_,_,_,3],
          E = [4,_,_,8,_,3,_,_,1],
          F = [7,_,_,_,2,_,_,_,6],
          G = [_,6,_,_,_,_,2,8,_],
          H = [_,_,_,4,1,9,_,_,5],
          I = [_,_,_,_,8,_,_,7,9],
          gen_digits(A), A = [5,3,1,2,7,4,6,9,8]
          Test the 'columns'
          gen_digits(B), B = [6,2,3,1,9,5,4,8,7]
          Test the 'columns'
          gen_digits(C),
          ...
          Test the 'boxes'
```

```
[5,3,1,2,7,4,6,9,8]
[6,2,3,1,9,5,4,8,7]
[1,9,8,2,3,4,5,6,7]
[8,_,_,_,6,_,_,_,3]
[4,_,_,8,_,3,_,_,1]
[7,_,_,_,2,_,_,_,6]
[_,6,_,_,_,_,2,8,_]
[_,_,_,4,1,9,_,_,5]
[_,_,_,_,8,_,_,7,9]
```

```
puzzle1 :-A = [5,3,_,_,7,_,_,_,_],
          B = [6,_,_,1,9,5,_,_,_],
          C = [_,9,8,_,_,_,_,6,_],
          D = [8,_,_,_,6,_,_,_,3],
          E = [4,_,_,8,_,3,_,_,1],
          F = [7,_,_,_,2,_,_,_,6],
          G = [_,6,_,_,_,_,2,8,_],
          H = [_,_,_,4,1,9,_,_,5],
          I = [_,_,_,_,8,_,_,7,9],
          gen_digits(A), A = [5,3,1,2,7,4,6,9,8]
          Test the 'columns'
          gen_digits(B), B = [6,2,3,1,9,5,4,8,7]
          Test the 'columns'
          gen_digits(C),
          Test the 'boxes'
```

test_boxes(RowA,RowB,RowC)

```
% test_boxes(RowA,RowB,RowC) succeeds if:
%    RowA, RowB, RowC are input lists of 9 unique digits [1..9]
%    each aligned 3x3 'box' contains 9 unique digits 1..9
test_boxes([A1,A2,A3,A4,A5,A6,A7,A8,A9],
           [B1,B2,B3,B4,B5,B6,B7,B8,B9],
           [C1,C2,C3,C4,C5,C6,C7,C8,C9]) :- test_digits([A1,A2,A3,B1,B2,B3,C1,C2,C3]),
                                            test_digits([A4,A5,A6,B4,B5,B6,C4,C5,C6]),
                                            test_digits([A7,A8,A9,B7,B8,B9,C7,C8,C9]).
```

Note by time of test, boxes are fully instantiated, i.e. ground.

```
solve(A,B,C,D,E,F,G,H,I) :- gen_digits(A),
                            test_cols([A,B,C,D,E,F,G,H,I]),
                            gen_digits(B),
                            test_cols([A,B,C,D,E,F,G,H,I]),
                            gen_digits(C),
                            test_boxes(A,B,C),
                            gen_digits(D),
                            test_cols([A,B,C,D,E,F,G,H,I]),
                            gen_digits(E),
                            test_cols([A,B,C,D,E,F,G,H,I]),
                            gen_digits(F),
                            test_cols([A,B,C,D,E,F,G,H,I]),
                            test_boxes(D,E,F),
                            gen_digits(G),
                            test_cols([A,B,C,D,E,F,G,H,I]),
                            gen_digits(H),
                            test_cols([A,B,C,D,E,F,G,H,I]),
                            gen_digits(I),
                            test_boxes(G,H,I),
                            test_cols([A,B,C,D,E,F,G,H,I]).
```

```
solve([5,3,_,_,7,_,_,_,_],
      [6,_,_,1,9,5,_,_,_],
      [_,9,8,_,_,_,_,6,_],
      [8,_,_,_,6,_,_,_,3],
      [4,_,_,8,_,3,_,_,1],
      [7,_,_,_,2,_,_,_,6],
      [_,6,_,_,_,_,2,8,_],
      [_,_,_,4,1,9,_,_,5],
      [_,_,_,_,8,_,_,7,9]
     ).
```

SUDOKU 8x8

https://en.wikipedia.org/wiki/Sudoku

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

```
puzzle1 :- solve([5,3,_,_,7,_,_,_,_],
                 [6,_,_,1,9,5,_,_,_],
                 [_,9,8,_,_,_,_,6,_],
                 [8,_,_,_,6,_,_,_,3],
                 [4,_,_,8,_,3,_,_,1],
                 [7,_,_,_,2,_,_,_,6],
                 [_,6,_,_,_,_,2,8,_],
                 [_,_,_,4,1,9,_,_,5],
                 [_,_,_,_,8,_,_,7,9]
                ).
```

SUDOKU 8x8

https://en.wikipedia.org/wiki/Sudoku

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Symbolic Evaluation: eval, reduce, flatten

```
eval(add(A,B),C) :- eval(A,A1), eval(B,B1), C is A1+B1.
reduce(+(A,B),C) :- reduce(A,A1), reduce(B,B1), add(A1,B1,C).

Function support:
    fun fact(1) = 1;
        fact(N) = N * fact(N-1).

    fun(;(=(fact(1),1), =(fact(N),*(N,fact(-(N,1)))))).

Flattening:
    foo(X,Y) :- moo(fact(X+3),Y).
becomes:
    foo(X,Y) :- add(X,3,A1), fact(A1,A2), moo(A2,Y).
       NOTE THAT FUNCTIONAL REDUCTION IS DETERMINISTIC
```

Next time

Videos

- Cut
- Negation
- Databases

Course Outline

1. Introduction, terms, facts, unification
2. Unification. Rules. Lists.
3. Arithmetic, Accumulators, Backtracking
4. Generate and Test (Dutch Flag, Sudoku), eval.
5. Extra-logical predicates (cut, negation, assert)
6. Graph Search
7. Difference Lists
8. Wrap Up (..Sudoku..)

Prolog
Programming in Logic

Lecture #5

Ian Lewis, Andrew Rice

Q: is gnd() special in Prolog or is it just a frequently used naming convention?

Q: is gnd() special in Prolog or is it just a frequently used naming convention?

A: swipl has ground(X) which is true if X is a ground term. gnd() is just a compound term. Don't use ground(X) in the exam... also atom(X), var(X) ...

Q: All the methods taught so far (like generate and test) don't seem too efficient computationally. In the exam should we think of more complex logic to do so?

Q: All the methods taught so far (like generate and test) don't seem too efficient computationally. In the exam should we think of more complex logic to do so?

A: If the exam question is interested in efficiency it will say so...past questions have not asked this. You make generate and test more efficient by generating better!

Q: With drawing out the execution traces - it's pretty difficult to understand them if you look back at them. How can we convey it in an exam?

Q: With drawing out the execution traces - it's pretty difficult to understand them if you look back at them. How can we convey it in an exam?

A: I can't remember an exam question where I asked for a search tree to be drawn out. Instead a question might ask for what happens: e.g. what results do you get. We'll see more of the 'search tree' in this lecture.

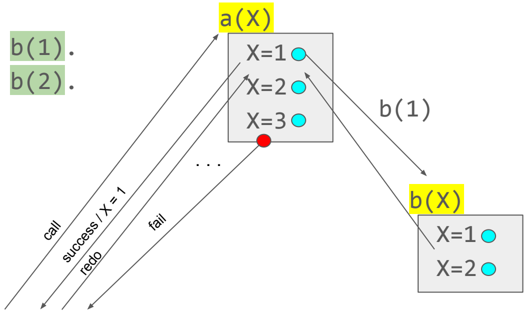
Today's discussion

Videos

- Cut
- Negation
- Databases (using the relational calculus for a relational database)

A procedural interpretation of cut(!)

```
a(1) :- b(1).
a(2) :- b(2).
a(3).

b(1).
b(2).

?- a(X).
```

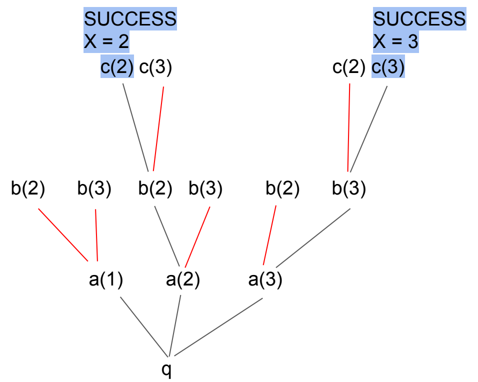


A procedural interpretation of cut(!)

```
a(1).
a(2).
a(3).

b(2).
b(3).

c(2).
c(3).

q :- a(X), b(X), !, c(X).

:- q.
```

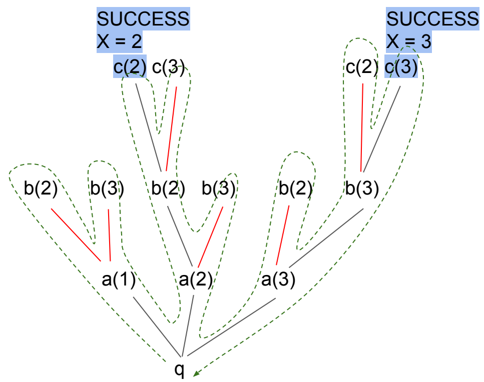
A procedural interpretation of cut(!) - without the cut:

```
a(1).
a(2).
a(3).

b(2).
b(3).

c(2).
c(3).

q :- a(X), b(X), c(X).

:- q.
```



A procedural interpretation of cut(!) - without the cut:

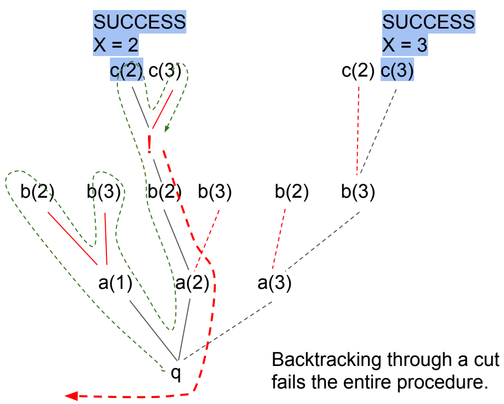
```
a(1).
a(2).
a(3).

b(2).
b(3).

c(2).
c(3).

q :- a(X), b(X), c(X).

:- q.
```



DEPTH-FIRST LEFT-TO-RIGHT SEARCH

A procedural interpretation of cut(!)

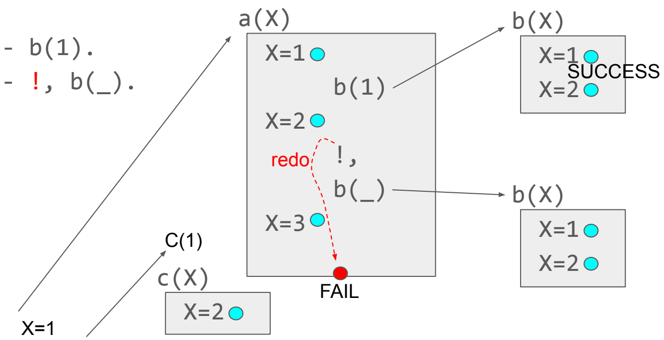
```
a(1).
a(2).
a(3).

b(2).
b(3).

c(2).
c(3).

q :- a(X), b(X), !, c(X).

:- q.
```



Backtracking through a cut fails the entire procedure.

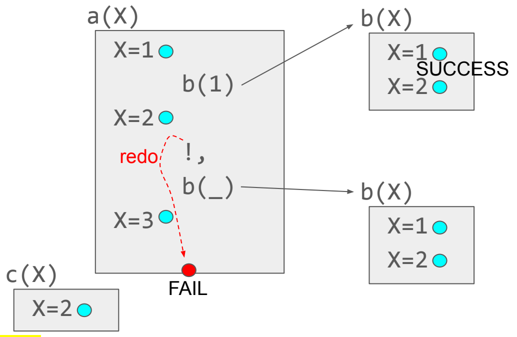
A procedural interpretation of cut(!)

```
a(1) :- b(1).
a(2) :- !, b(_).
a(3).

b(1).
b(2).

c(2).

?- a(X), c(X).
```



A procedural interpretation of cut(!)

```
a(1) :- b(1).
a(2) :- !, b(_).
a(3).

b(1).
b(2).

c(2).
```



?- b(X), a(X), c(X).

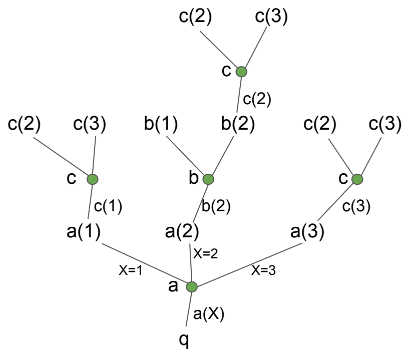
another (very similar) example - without cut

a(1).
a(2) :- b(2).
a(3).

b(1).
b(2).

c(2).
c(3).

q :- a(X), c(X).



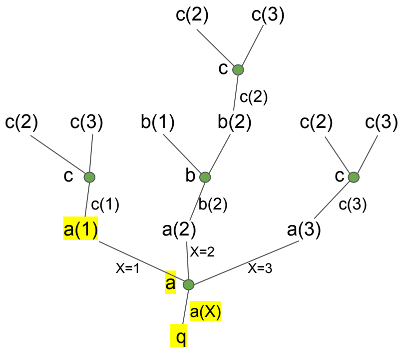
another (very similar) example - without cut

a(1).
a(2) :- b(2).
a(3).

b(1).
b(2).

c(2).
c(3).

q :- a(X), c(X).



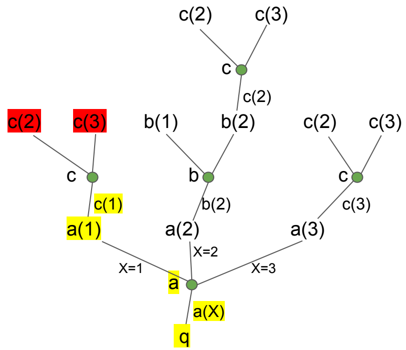
another (very similar) example - without cut

a(1).
a(2) :- b(2).
a(3).

b(1).
b(2).

c(2).
c(3).

q :- a(X), c(X).



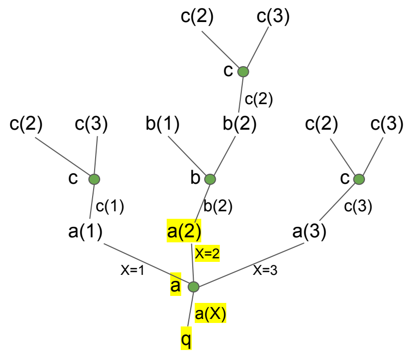
another (very similar) example - without cut

```
a(1).
a(2) :- b(2).
a(3).
```

```
b(1).
b(2).
```

```
c(2).
c(3).
```

```
q :- a(X), c(X).
```



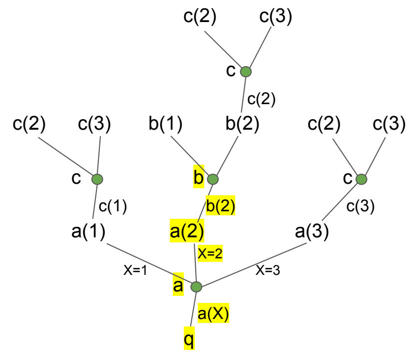
another (very similar) example - without cut

```
a(1).
a(2) :- b(2).
a(3).
```

```
b(1).
b(2).
```

```
c(2).
c(3).
```

```
q :- a(X), c(X).
```



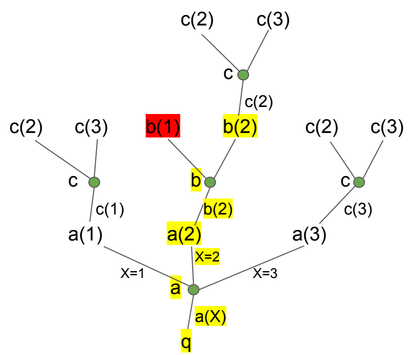
another (very similar) example - without cut

```
a(1).
a(2) :- b(2).
a(3).
```

```
b(1).
b(2).
```

```
c(2).
c(3).
```

```
q :- a(X), c(X).
```



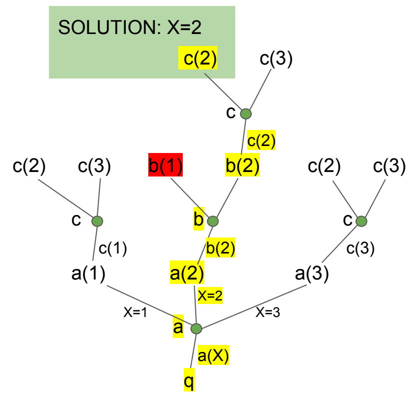
another (very similar) example - without cut

```
a(1).
a(2) :- b(2).
a(3).
```

```
b(1).
b(2).
```

```
c(2).
c(3).
```

```
q :- a(X), c(X).
```



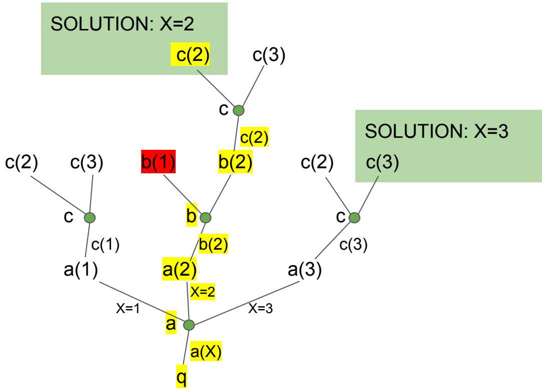
another (very similar) example - without cut

```
a(1).
a(2) :- b(2).
a(3).
```

```
b(1).
b(2).
```

```
c(2).
c(3).
```

```
q :- a(X), c(X).
```



another (very similar) example - cut

```
a(1).
a(2) :- b(2).
a(3).
```

```
b(1).
b(2).
```

```
c(2).
c(3).
```

```
q :- a(X), c(X).
```

another (very similar) example - cut

```
a(1).
a(2) :- !, b(2).
a(3).
```

```
b(1).
b(2).
```

```
c(2).
c(3).
```

```
q :- a(X), c(X).
```

another (very similar) example - cut

```
a(1).
a(2) :- b(2).
a(3).
```

```
b(1).
b(2).
```

```
c(2).
c(3).
```

```
q :- a(X), !, c(X).
```

another (very similar) example - cut

```
a(1).
a(2) :- !, b(2).
a(3).
```

```
b(1).
b(2).
```

```
c(2).
c(3).
```

```
q :- a(X), c(X).
```

the last word on what cut does...

```
a(1).
a(2) :- !, b(2).
a(3).
```

```
b(1).
b(2).
```

```
c(2).
c(3).
```

```
q :- c(X), a(X), c(X).
```

Cut(!) toxicology

```
a(1).
a(2) :- !.
a(3).

b(X) :- a(X).
b(4).
```

```
:- a(X).
X = 1 ;
X = 2
```

```
:- b(X).
X = 1 ;
X = 2 ;
X = 4
```

```
:- a(3).
true
```

Cut(!) toxicology

```
a(1).
a(2) :- !.
a(3).

b(X) :- a(X).
b(4).
```

```
:- a(X).
X = 1 ;
X = 2
```

```
:- b(X).
X = 1 ;
X = 2 ;
X = 4
```

```
:- a(3).
true
```

negation / not

q :- ... ,\+ foo(X), ...

relational database

```
%    name      age                         name  floor
age(andy,       35).        location(andy,       2).
age(alastair, 45).        location(alastair, 2).
age(ian,          65).        location(ian,          1).
age(jon,          60).

SELECT name, floor FROM age,location
 WHERE age.name=location.name AND age > 40.

:- age(Name,Age), location(Name,Floor), Age > 40.
```

List relations - len/2, mem/2

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

% mem(?X,?L)
% succeeds if X is in list L.
mem(X,[X|_]).
mem(X,[_|T]) :- mem(X,T).
```

List relations - app/3, reverse/2

```
% app(?L1,?L2,?L3) = APPEND
% succeeds if L1 appended to L2 is L3.
app([],L2,L2).
app([X|T],L2,[X|L3]) :- app(T,L2,L3).

% reverse(+L1,-L2)
% succeeds if list L2 is the reverse of list L1
reverse([],[]).
reverse([X|T], L) :- reverse(T, L1), append(L1,[X],L).
```

List relations - take/3, perm/2

```
% take(+L1,-X,-L2)
% succeeds if list L2 is list L1 minus element X
take([H|T],H,T).
take([H|T],R,[H|S]) :- take(T,R,S).

% perm(+L1,-L2)
% succeeds if list L2 is a permutation of list L1
perm([],[]).
perm(List,[H|T]) :- take(List,H,R), perm(R,T).
```


List relations - take/3, perm/2

```
% take(+L1,-X,-L2)
% succeeds if list L2 is list L1 minus element X
take([H|T],H,T).
take([H|T],R,[H|S]) :- take(T,R,S).

% perm(+L1,-L2)
% succeeds if list L2 is a permutation of list L1
perm([],[]).
perm(List,[H|T]) :- take(List,H,R), perm(R,T).
```


List relations - len/2, rev/2 with accumulators

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

% len_acc(+L,+A,-N) succeeds if A is an accumulated length so far,
% L is the remaining list to be counted,  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).

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

% rev_acc(+L1, +ListAcc, -L2) succeeds if ListAcc is the accumulated reversed list so far,
% L1 is the remaining list to be reversed, L2 is the reverse of the original list.
rev_acc([], ListAcc, ListAcc).
rev_acc([H|T], ListAcc, L2) :-  rev_acc(T, [H|ListAcc], L2).
```

List relations - len/2, rev/2 with accumulators

Simple:

```
len(+L,-N)
mem(?X,?L)
app(?L1,?L2,?L3)
reverse(+L1,-L2)
take(+L1,-X,-L2)
perm(+L1,-L2)
```

Accumulator:

```
len(+L,-N)
rev(+L1,-L2)
```

ASSUME member(X,L), append(L1,L2,L3).

Next time

Lecture #6: Videos

- Countdown (more generate-and-test)
- Graph search

Lecture #7: Difference lists

Lecture #8: (Sudoku), Wrap Up.

Prolog
Programming in Logic

Lecture #6

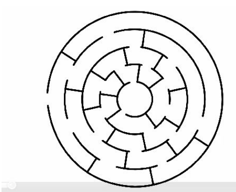
Ian Lewis, Andrew Rice

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!

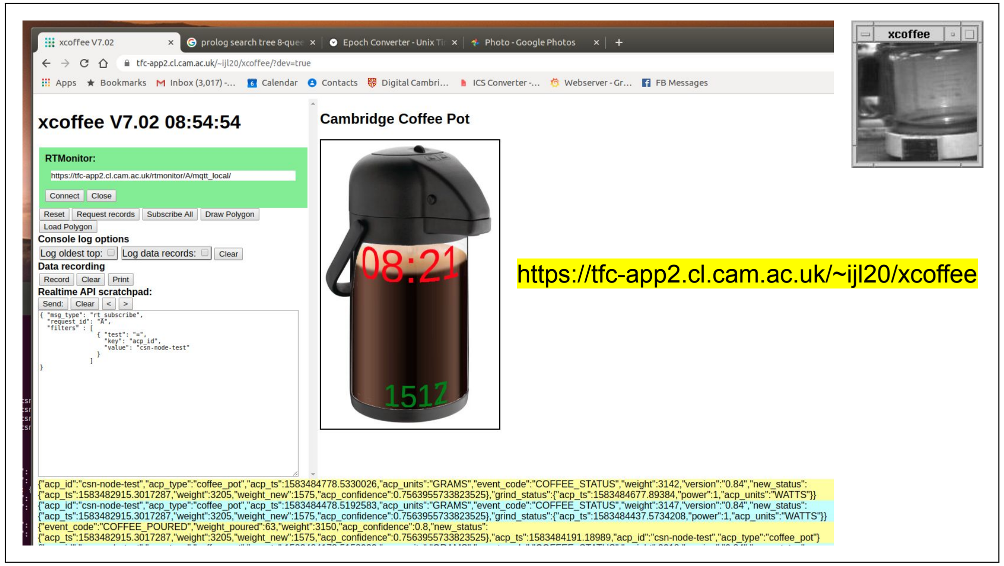
Q: What does Prolog allow us to do (other than coding in a different way) that other languages can't? Not meaning to sound dismissive just curious of applications!

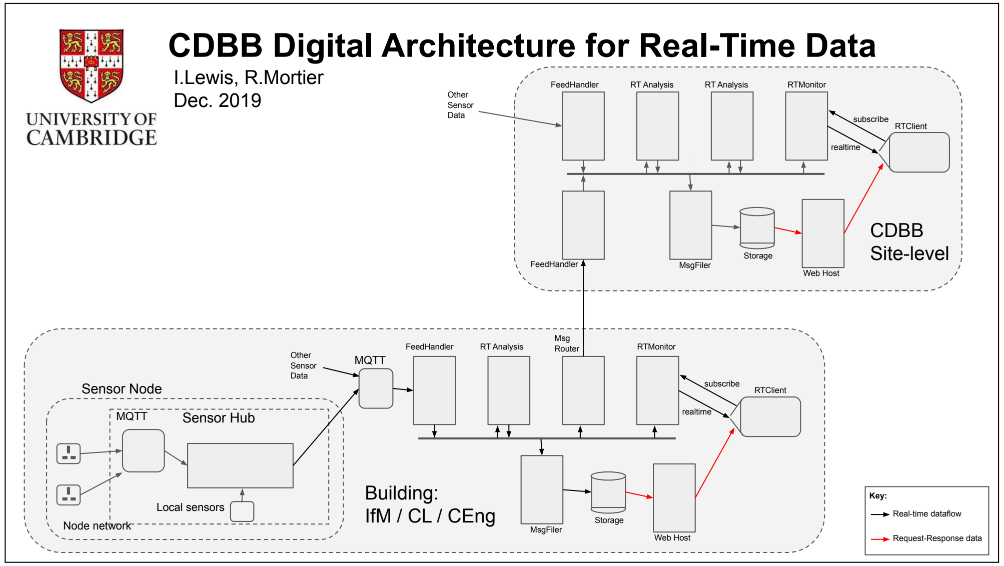
A:

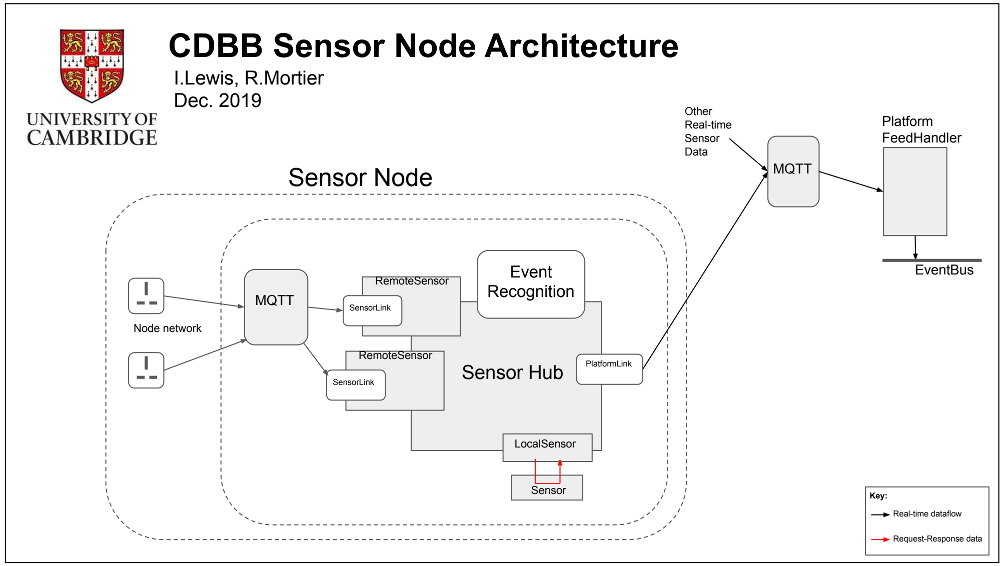
* Pure Prolog subset 'Datalog' used for network verification.
* Prolog used for Java Virtual Machine verification
* Prolog quite good at digital logic simulation

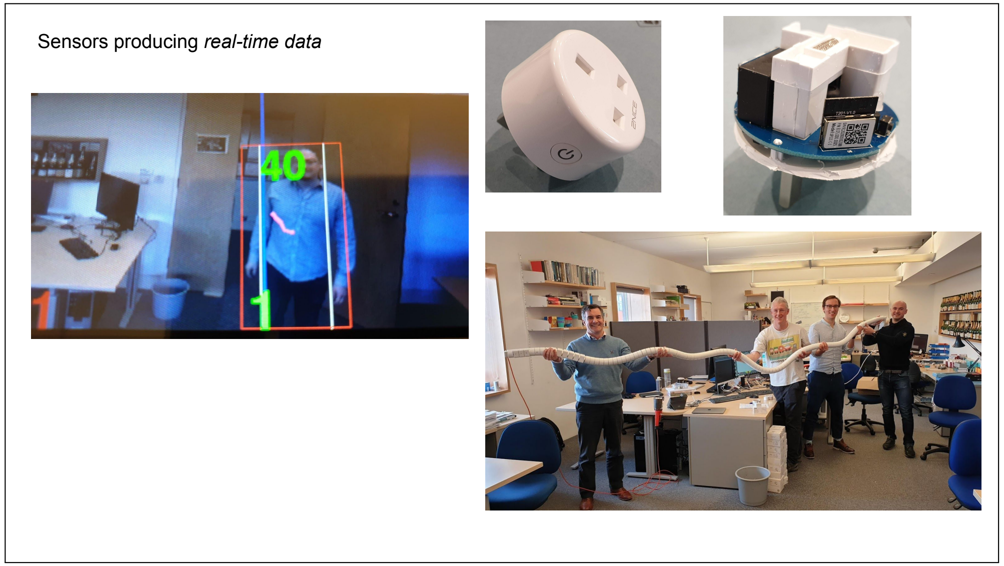
... theorem provers written in Prolog.

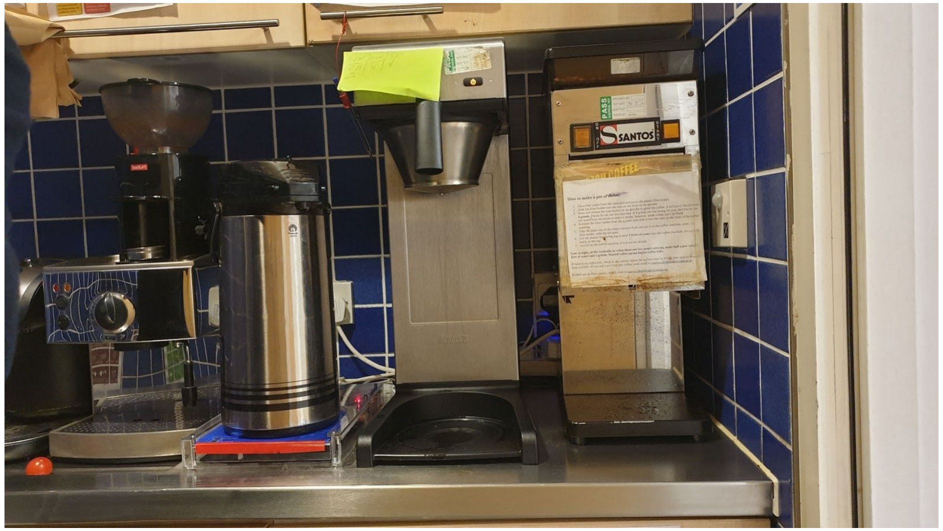
* Sooner or later, some method of reasoning with NN data will emerge.

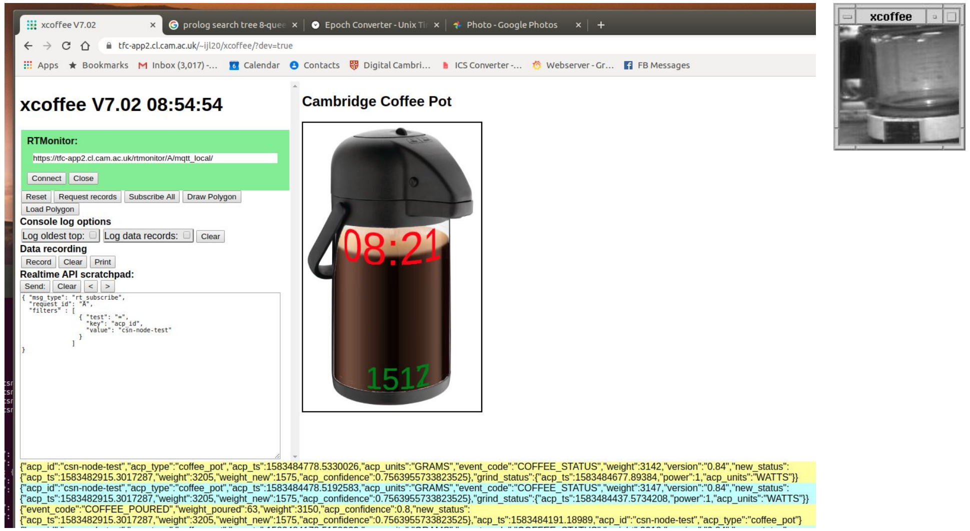

xcoffee V7.02 08:54:54
RTMonitor:
Connect
Close
Reset
Request records
Subscribe All
Draw Polygon
Load Polygon
Console log options
Log oldest top:
Log data records:
Clear
Data recording
Record
Clear
Print
Realtime API scratchpad:
Send
Clear
Cambridge Coffee Pot
08:21
1512
https://tfc-app2.cl.cam.ac.uk/~ijl20/xcoffee
xcoffee


CDBB Digital Architecture for Real-Time Data
I.Lewis, R.Mortier
Dec. 2019
UNIVERSITY OF CAMBRIDGE
Other Sensor Data
FeedHandler
RT Analysis
RT Analysis
RTMonitor
subscribe
realtime
RTClient
FeedHandler
MsgFiler
Storage
Web Host
CDBB Site-level
Other Sensor Data
MQTT
FeedHandler
RT Analysis
Msg Router
RTMonitor
subscribe
realtime
RTClient
Sensor Node
MQTT
Sensor Hub
Local sensors
Node network
Building: IfM / CL / CEng
MsgFiler
Storage
Web Host
Key:
Real-time dataflow
Request-Response data


CDBB Sensor Node Architecture
I.Lewis, R.Mortier
Dec. 2019
UNIVERSITY OF CAMBRIDGE
Sensor Node
Other Real-time Sensor Data
MQTT
Platform FeedHandler
EventBus
Node network
MQTT
RemoteSensor
SensorLink
RemoteSensor
SensorLink
Event Recognition
Sensor Hub
PlatformLink
LocalSensor
Sensor
Key:
Real-time dataflow
Request-Response data


Sensors producing real-time data
40
1

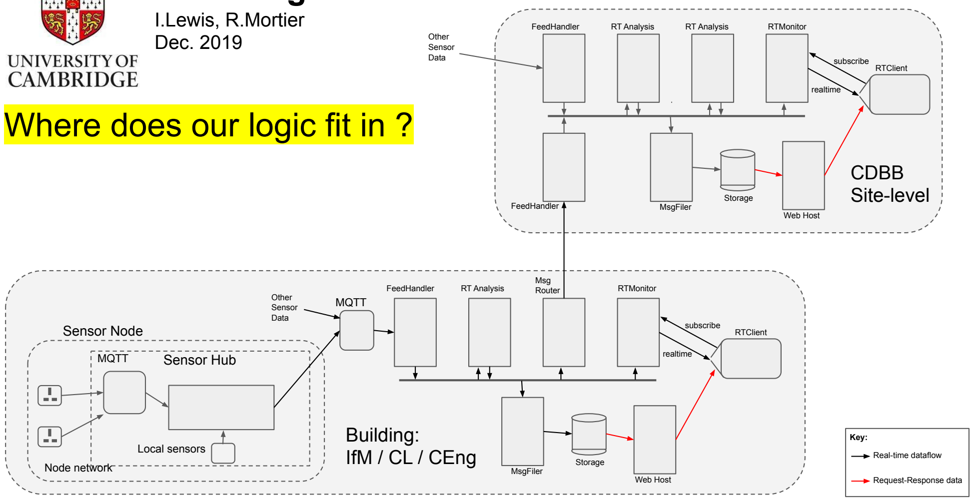




CDBB Digital Architecture for Real-Time Data

I.Lewis, R.Mortier
Dec. 2019
UNIVERSITY OF CAMBRIDGE

Where does our logic fit in ?



Q: Operators & precedence? :- op(700, xfx, arc).

300	xfx	mod	Arithmetic function
400	yfx	*	Arithmetic function
400	yfx	/	Arithmetic function
400	yfx	//	Arithmetic function
500	fx	+	Arithmetic function
500	fx	-	Arithmetic function
500	yfx	+	Arithmetic function
500	yfx	-	Arithmetic function
700	xfx	<	Predicate
700	xfx	=	Predicate
700	xfx	=..	Predicate
700	xfx	<	Predicate
700	xfx	>	Predicate
700	xfx	>=	Predicate
700	xfx	is	Predicate
900	fy	\+	Predicate
1000	xfy	,	Predicate
1100	xfy	;	Predicate
1200	fx	:-	Introduces a directive
1200	xfx	:-	head :- body. separator

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?

arc(X,Y)
op(700, xfx, arc).

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

```
a arc b.
b arc c.
c arc d.
c arc e.

path(A,B) :- A arc B.
path(A,B) :- A arc X, path(X,B).
```

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

```
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?
            ...

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

```
% 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) , ...
```

choose

An aside/caution regarding functional support...

FLATTENING

```
% 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

```
% 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+B,C) :- eval(A,A1), eval(B,B1), C is A1 + B1.
eval(A*B,C) :- eval(A,A1), eval(B,B1), C is A1 * B1.
eval(A/B,C) :- eval(A,A1), eval(B,B1), C is A1 / B1.
eval(A-B,C) :- eval(A,A1), eval(B,B1), C is A1 - B1.
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+B,C) :- eval(A,A1), eval(B,B1), C is A1 + B1.
eval(A*B,C) :- eval(A,A1), eval(B,B1), C is A1 * B1.
eval(A/B,C) :- eval(A,A1), eval(B,B1), C is A1 / B1.
eval(A-B,C) :- eval(A,A1), eval(B,B1), C is A1 - B1.
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(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 - alternative version of countdown/3

```
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 - alternative version of countdown/3

```
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 - alternative version of countdown/3

```
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 Iterative Deepening

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) ).
```

Countdown Iterative Deepening

```
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).

Countdown Iterative Deepening

```
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)
```

Countdown Iterative Deepening

```
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 Iterative Deepening

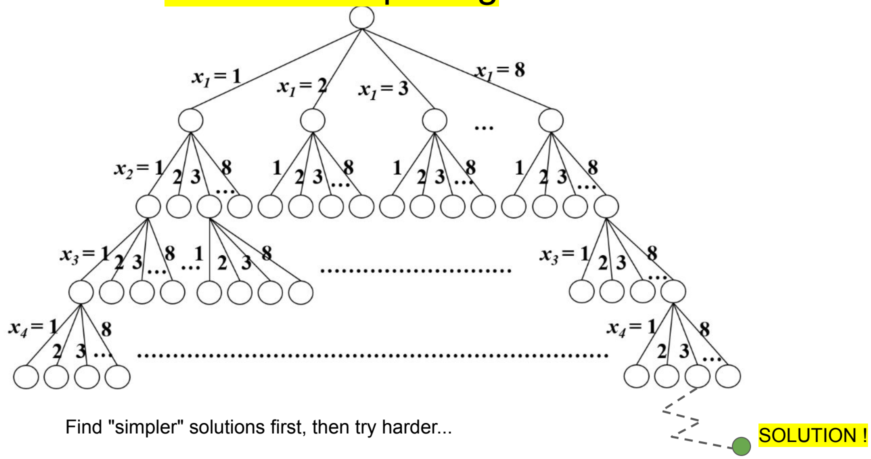
```
diff(A,B,Diff) :- Delta is A - B, (Delta < 0 , Diff is -Delta ; Delta >= 0, Diff is Delta).

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)
```

..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) ).
```

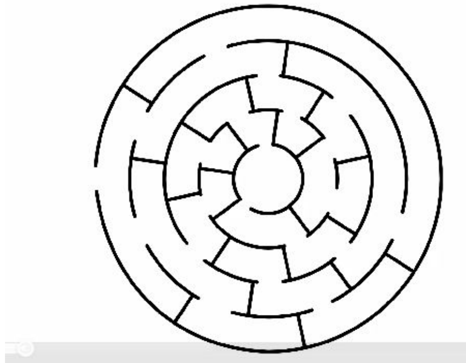
Required Threshold   Actual Difference

```
:- 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 ..."

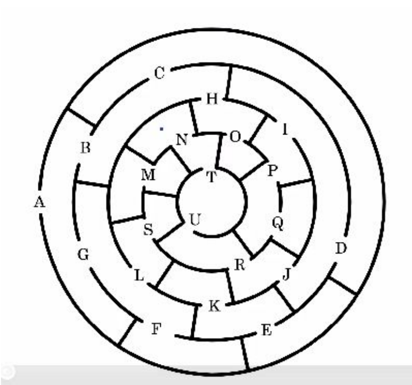
Also as video: closest(L, Target, Soln, Threshold) :- range(0,100,Threshold), solve2(L,Target,Soln,Threshold).

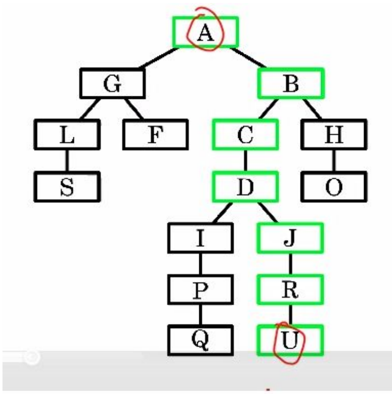
Graph Search



Problem statement

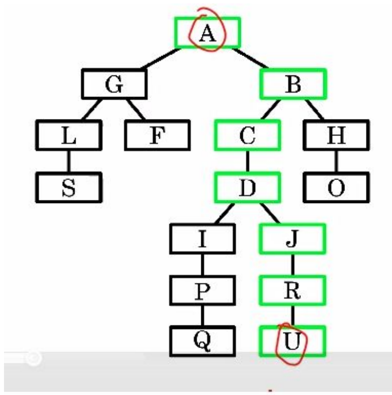
Graph Search





Convert to graph...

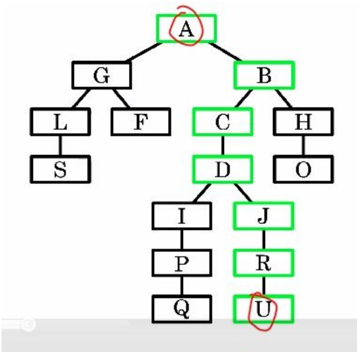
Graph Search




```
route(a,g).          start(a).
route(g,l).          finish(u).
route(l,s).
...
travel(A,A).
travel(A,C) :- route(A,B),travel(B,C).

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

Sample implementation (simple, given graph)

Graph Search



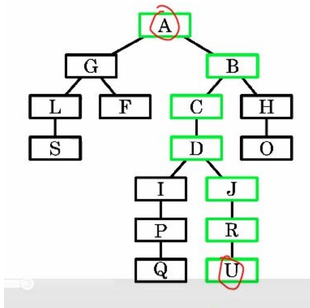

```
route(a,g).          start(a).
route(g,l).          finish(u).
route(l,s).
...
travel(A,A).
travel(A,C) :- route(A,B),travel(B,C).

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

```
:- 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.
```

```
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: `path_acc(X,Finish,Acc,Path) :- X arc Finish, reverse([Finish,X|Acc],Path).`

(2) Recursive case: `path_acc(X,Finish,Acc,Path) :- X arc Z, path_acc(Z,Finish,[X|Acc],Path).`

Accumulating here

Copying to solution here (via reverse/2)

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?

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

This pointlessly backtracks after finding the answer.

So change axiom to: last(H,[H]) :- !.

Q: Is single cut rule bad practice?

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

This pointlessly backtracks after finding the answer.
So change axiom to: last(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(Before,Player,After)`
Represent moves as move(Position,Player)
Represent the game state as a list of moves that have been made

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

```
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

Prolog
Programming in Logic

Lecture #7

Ian Lewis, Andrew Rice

Today's discussion

Videos

Difference

Empty difference lists

Difference list example

Q: Attempting the towers of hanoi problem and run into stack space issues which makes me think my state representation is bad, any hints (specific and general)?

Q: Attempting the towers of hanoi problem and run into stack space issues which makes me think my state representation is bad, any hints (specific and general)?

A: if you are running out of stack space you probably have a searching-forever problem are more rules matching than you thought? General state representation hint: as little redundancy as possible.

```
% Towers of Hanoi

% Represent the current state as rings(A,B,C) where
%  A is the peg that the smallest ring is on
%  B is the peg that the middle ring is on
%  C is the peg that the largest ring is on
% Start rings(1,1,1). Finish rings(3,3,3).

range(Min,_,Min).
range(Min,Max,Next) :- N2 is Min+1, N2 < Max, range(N2,Max,Next).

move(rings(Src,A,B),rings(Dest,A,B)) :-
        range(1,4,Src),
        range(1,4,Dest),
        Src \= Dest.

move(rings(A,Src,B),rings(A,Dest,B)) :-
        range(1,4,Src),
        range(1,4,Dest),
        A \= Src, A \= Dest.

move(rings(A,B,Src),rings(A,B,Dest)) :-
        range(1,4,Src),
        range(1,4,Dest),
        A \= Src, A \= Dest,
        B \= Src, B \= Dest.

search(Dest,Dest,_,[]).
search(Src,Dest,Closed,[Mid|Path]) :-
        move(Src,Mid),
        \+member(Mid,Closed),
        search(Mid,Dest,[Mid|Closed],Path).

solve :- search(rings(1,1,1),rings(3,3,3),[rings(1,1,1)],Path),
         print([rings(1,1,1)|Path]).
```

```
% Hanoi puzzle
% Rings number 1,2,3.
% Each tower A,B,C a list of rings (Head = top).
% State stored as state(A, B, C).
% Start state([1,2,3],[],[]). Finish state([],[],[1,2,3]).

% make_move(+Tower1,+Tower2, -Tower1_after, -Tower2_after).
% Will only make valid moves, i.e. onto empty or bigger tower.
make_move([A1|A],[],A,[A1]).
make_move([A1|A],[B1|B],A,[A1,B1|B]) :- A1 < B1.

% move(+State_before, -State_after)
% Generate valid moves
% move A->B or A->C or B->A or B->C or C->A or C->B.
move(state(A,B,C), state(AN,BN,C)) :- make_move(A,B,AN,BN).
move(state(A,B,C), state(AN,B,CN)) :- make_move(A,C,AN,CN).
move(state(A,B,C), state(AN,BN,C)) :- make_move(B,A,BN,AN).
move(state(A,B,C), state(A,BN,CN)) :- make_move(B,C,BN,CN).
move(state(A,B,C), state(AN,B,CN)) :- make_move(C,A,CN,AN).
move(state(A,B,C), state(A,BN,CN)) :- make_move(C,B,CN,BN).

search(State_from, State_to, Path) :- move(State_from,State_to),
                                      print(Path).

search(State_from, State_to, Path) :-
        move(State_from, Next_state),
        \+ member(Next_state, Path),
        search(Next_state,State_to,[Next_state|Path]).

solve :- search(state([1,2,3],[],[]),state([],[],[1,2,3]), []).
```

Q. N > 0 ? extra-logical ?

```
choose(0, L, [], L).
choose(N, [H|T], [H|R], S) :- N > 0, N2 is N-1, choose(N2, T, R, S).
choose(N, [H|T], R, [H|S]) :- N > 0, choose(N, T, R, S).


        :- choose(2, [a,b,c,d], Chosen, Remaining).

        :- choose(0, [a,b,c,d], Chosen, Remaining).
```

Q. N > 0 ? extra-logical ?

```
choose(0, L, [], L).
choose(N, [H|T], [H|R], S) :- N > 0, N2 is N-1, choose(N2, T, R, S).
choose(N, [H|T], R, [H|S]) :- N > 0, choose(N, T, R, S).

% Trace version
choose2(0, L, [], L) :-
    print('clause 1- success'), nl.
choose2(N, [H|T], [H|R], S) :-
    print('clause 2-  N='), print(N), print(' from '), print([H|T]), nl,
    N2 is N-1, choose2(N2, T, R, S).
choose2(N, [H|T], R, [H|S]) :-
    print('clause 3-  N='), print(N), print(' from '), print([H|T]), nl,
    choose2(N, T, R, S).
```

Difference lists

Difference lists

Which of these is a difference list:

1. diff(A,B)
2. A-B
3. [1,2,3|A]-A
4. [1,2,3|A]-B
5. []-[]
6. A-A

Difference lists

Which of these is a difference list:

1. diff(A,B)
2. A-B
3. [1,2,3|A]-A
4. [1,2,3|A]-B
5. []-[]
6. A-A

A gentle introduction to difference lists.

```
?- X = [1,2,A,4,5].
X = [1,2,_4096,4,5].
```

A gentle introduction to difference lists.

```
?- X = [1,2,A,4,5].
X = [1,2,A,4,5].
```

A gentle introduction to difference lists.

```
?- X = [1,2,A,4,5], A = woohoo.  -- retrospectively fill in the hole.
X = [1,2,woohoo,4,5],
A = woohoo.
```

A gentle introduction to difference lists.

```
?- X = [1,2,A,4,5], A = woohoo.
X = [1,2,woohoo,4,5],
A = woohoo.
```

That's great! But what's the point ????

You can pass around as-yet-incomplete data structures.

e.g. you can add an element to the Tail of a list (the canonical example).

You get to hone your unification comprehension.

A gentle introduction to difference lists.

```
?- X = [1,2,3|A].
X = [1,2,3|A].        --- A list with a hole in it...

                       -- Retrospectively fill in the tail...
```

A gentle introduction to difference lists.

```
?- X = [1,2,3|A].
X = [1,2,3|A].        --- A list with a hole in it...

The tail of a list is always a list. So what about:
?- X = [1,2,3|7].
X = [1,2,3|7].

? X = [1,2,3|7], X = [A,B,C].
false

? X = [1,2,3|7], X = [A,B,C,D].
false
```

In fact you just have a compound term |(3,7) stuck on the end of the list and all the relations expecting |(last_element,[]) simply fail. Depends on implementation.

So [1,2,3|7] IS NOT A LIST.

A gentle introduction to difference lists.

Very few of you will *write* a list [1,2,3|7]... it arises from:

```
?- X = [1,2,3|A], A=7.        Correct would be A = [7].
```

A gentle introduction to difference lists.

A more significant / common / relevant example:

Set the difflist var as the empty list.

```
?- X = [1,2,3|A], A=[].
X = [1,2,3],
A = [].                   -- we are simply terminating the list.
```

A gentle introduction to difference lists.

A more significant / common / relevant example:

```
?- X = [1,2,3|A], A=[].
X = [1,2,3],
A = [].                   -- we are simply terminating the list.
```

For the avoidance of doubt, the 'X' list is equally:

```
?- X = [ 1 | [ 2 | [ 3 | [] ] ] ].
X = [1,2,3].
```

A gentle introduction to difference lists.

With two lists:

```
?- X = [1,2,3|A], Y = [4,5,6|B].
X = [1,2,3|A],
Y = [4,5,6|B].
```

A gentle introduction to difference lists.

With two lists:

```
?- X = [1,2,3|A], Y = [4,5,6|B].
X = [1,2,3|A],
Y = [4,5,6|B].
```

Linking the lists...

```
?- X = [1,2,3|A], Y = [4,5,6|B], A = Y.
X = [1,2,3,4,5,6|B],  -- so we have managed to append, via unification
A = Y,
Y=[4,5,6|B].
```

A gentle introduction to difference lists.

This is great! How to write an append ?

```
?- X = [1,2,3|A], Y = [4,5,6|B], append(X,Y,Z).

append([],Y,Y).
append(?,?,?) :- ...
```

A gentle introduction to difference lists.

This is great! How to write an append ?

```
?- X = [1,2,3|A], Y = [4,5,6|B], append(X,Y,Z).

append([],Y,Y).
append(?,?,?) :- ...
```

You can't, is the short answer... you need to propagate a reference to A and B.

A gentle introduction to difference lists.

```
?- X = [1,2,3|A]-A, Y = [4,5,6|B]-B, append(X,Y,Z).

So you can pass the list and its tail var as a single compound term.

append(X,Y,Z) :- X = XL-XVar,  -- get the list/var components of X
                 Y = YL-YVar,  -- get the list/var components of Y
                 Z = ZL-ZVar,  -- make a new diff list for Z
                 XVar = YL,    -- unify the X var with the Y list
                 ZL = XL,      -- unify the X list with the Z list
                 ZVar = YVar.  -- make the var of Z as for Y
?- app([1,2,3|A]-A,[4,5,6|B]-B,C).
C = [1, 2, 3, 4, 5, 6|B]-B,
A = [4, 5, 6|B].
```

A gentle introduction to difference lists.

```
?- X = [1,2,3|XVar]-XVar, Y = [4,5,6|YVar]-YVar, append(X,Y,Z).

So you can pass the list and its tail var as a single compound term.
Collapse the 'parsing' unifications into the head of the clause:
append(X,Y,Z) :- X = XL-XVar,  -- get the list/var components of X
                 Y = YL-YVar,  -- get the list/var components of Y
                 Z = ZL-ZVar,  -- make a new diff list for Z
                 XVar = YL,    -- unify the X var with the Y list
                 ZL = XL,      -- unify the X list with the Z list
                 ZVar = YVar.  -- make the var of Z as for Y
?- app([1,2,3|A]-A,[4,5,6|B]-B,C).
C = [1, 2, 3, 4, 5, 6|B]-B,
A = [4, 5, 6|B].
```

A gentle introduction to difference lists.

```
?- X = [1,2,3|XVar]-XVar, Y = [4,5,6|YVar]-YVar, append(X,Y,Z).

So you can pass the list and its tail var as a single compound term.
Collapse the 'parsing' unifications into the head of the clause:
append(XL-XVar,YL-YVar,ZL-ZVar) :-
                 XVar = YL, -- unify the X var with the Y list
                 ZL = XL, -- unify the X list with the Z list
                 ZVar = YVar. -- make the var of result the same as Y

?- app([1,2,3|A]-A,[4,5,6|B]-B,C).
C = [1, 2, 3, 4, 5, 6|B]-B,
A = [4, 5, 6|B].
```

A gentle introduction to difference lists.

```
?- X = [1,2,3|XVar]-XVar, Y = [4,5,6|YVar]-YVar, append(X,Y,Z).

Now we can propagate the remaining unifications:

append(XL-XVar,YL-YVar,ZL-ZVar) :-
                 XVar = YL, -- unify the X var with the Y list
                 ZL = XL,
                 ZVar = YVar. -- make the var of result the same as Y

?- app([1,2,3|A]-A,[4,5,6|B]-B,C).
C = [1, 2, 3, 4, 5, 6|B]-B,
A = [4, 5, 6|B].
```

A gentle introduction to difference lists.

```
?- X = [1,2,3|XVar]-XVar, Y = [4,5,6|YVar]-YVar, append(X,Y,Z).



append(XL-XVar,XVar-YVar,ZL-ZVar) :-
                ZL = XL,
                ZVar = YVar. -- make the var of result the same as Y

?- app([1,2,3|A]-A,[4,5,6|B]-B,C).
C = [1, 2, 3, 4, 5, 6|B]-B,
A = [4, 5, 6|B].
```

A gentle introduction to difference lists.

```
?- X = [1,2,3|XVar]-XVar, Y = [4,5,6|YVar]-YVar, append(X,Y,Z).



append(XL-XVar,XVar-YVar,ZL-ZVar) :-
                ZL = XL,
                ZVar = YVar. -- make the var of result the same as Y

?- app([1,2,3|A]-A,[4,5,6|B]-B,C).
C = [1, 2, 3, 4, 5, 6|B]-B,
A = [4, 5, 6|B].
```

A gentle introduction to difference lists.

```
?- X = [1,2,3|XVar]-XVar, Y = [4,5,6|YVar]-YVar, append(X,Y,Z).


Rename XL to A:
append(XL-XVar,XVar-YVar,XL-YVar).

?- app([1,2,3|A]-A,[4,5,6|B]-B,C).
C = [1, 2, 3, 4, 5, 6|B]-B,
A = [4, 5, 6|B].
```

A gentle introduction to difference lists.

```
?- X = [1,2,3|XVar]-XVar, Y = [4,5,6|YVar]-YVar, append(X,Y,Z).


Rename XVar to B:
append(A-XVar,XVar-YVar,A-YVar).

?- app([1,2,3|A]-A,[4,5,6|B]-B,C).
C = [1, 2, 3, 4, 5, 6|B]-B,
A = [4, 5, 6|B].
```

A gentle introduction to difference lists.

```
?- X = [1,2,3|XVar]-XVar, Y = [4,5,6|YVar]-YVar, append(X,Y,Z).


Rename YVar to C:
append(A-B,B-YVar,A-YVar).

?- app([1,2,3|A]-A,[4,5,6|B]-B,C).
C = [1, 2, 3, 4, 5, 6|B]-B,
A = [4, 5, 6|B].
```

A gentle introduction to difference lists.

```
?- X = [1,2,3|B]-B, Y = [4,5,6|C]-C, append(X,Y,Z).


append(A-B,B-C,A-C).

?- app([1,2,3|A]-A,[4,5,6|B]-B,C).
C = [1, 2, 3, 4, 5, 6|B]-B,
A = [4, 5, 6|B].
```

A gentle introduction to difference lists.

```
% Final empty diff list list thoughts.

?- X = [a,b,c|A]-A, A = [], X = MyList-_.
MyList = [a,b,c].

? X = A-A, A=[], X = MyList-_.
MyList = [].

Empty diff list is ALWAYS A-A.  But to TEST for it you attempt a
unification with something that only matches <freevar>-<freevar>.

FWIW I think of diff lists a bit like complex numbers - with real and
the imaginary parts. Ultimately you're interested in the real part.
```

Challenge: Implement Quicksort

Partition the list into two pieces

Quicksort each half

Implement Quicksort

```
% partition(+Pivot,+List,-Left,-Right) succeeds if Left is all the
% elements in List less than or equal to the pivot and Right is
% all the elements greater than the pivot

% quicksort(+L1,-L2) succeeds if L2 contains the elements in L1 in
% ascending order
```

Implement partition

```
% partition(+Pivot,+List,-Left,-Right) succeeds if Left is all the
% elements in List less than or equal to the pivot and Right is
% all the elements greater than the pivot

partition(_,[],[],[]).
partition(P,[H|T],[H|L],R) :- P <= H, partition(P,T,L,R).
partition(P,[H|T],L,[H|R]) :- P > H, partition(P,T,L,R).

    :- partition(5, [1,3,5,7,9], Left, Right).
    Left = [1,3,5]
    Right = [7,9]
```

Implement quicksort

```
partition(_,[],[],[]).
partition(P,[H|T],[P|L],R) :- P <= H, partition(P,T,L,R).
partition(P,[H|T],L,[P|R]) :- P > H, partition(P,T,L,R).

quicksort([],[]).
quicksort([P|T],Sorted) :-
    partition(P,T,L,R),
    quicksort(L,L1), quicksort(R,R1),
    append(L1,R1,Sorted).
```

Is it useful to turn this into difference lists?

```
partition(_,[],[],[]).
partition(P,[H|T],[P|L],R) :- P <= H, partition(P,T,L,R).
partition(P,[H|T],L,[P|R]) :- P > H, partition(P,T,L,R).

quicksort([],[]).
quicksort([P|T],Sorted) :-
    partition(P,T,L,R),
    quicksort(L,L1), quicksort(R,R1),
    append(L1,[P|R1],Sorted).
```

Step 1: Replace appended lists with difference lists

```
quicksort([],[]).
quicksort([P|T],Sorted) :-
    partition(P,T,L,R),
    quicksort(L,L1), quicksort(R,R1),
    append(L1,[P|R1],Sorted).
```

What do you notice?

Step 1: Replace appended lists with difference lists

```
quicksort([],A-A).
quicksort([P|T],Sorted-S2) :-
    partition(P,T,L,R),
    quicksort(L,L1-L2), quicksort(R,R1-R2),
    append(L1-L2,[P|R1]-R2,Sorted-S2).
```

The input list remains a 'normal' list.

Step 2: Worry about empty difference lists

```
quicksort([],A-A).
quicksort([P|T],Sorted-S2) :-
    partition(P,T,L,R),
    quicksort(L,L1-L2), quicksort(R,R1-R2),
    append(L1-L2,[P|R1]-R2,Sorted-S2).
```

Should this be []-[] or A-A ?

Step 2: Worry about empty difference lists

```
quicksort([],A-A).
quicksort([P|T],Sorted-S2) :-
    partition(P,T,L,R),
    quicksort(L,L1-L2), quicksort(R,R1-R2),
    append(L1-L2,[P|R1]-R2,Sorted-S2).
```

Should this be []-[] or A-A ?
A-A because we are RETURNING an empty list, not TESTING for it.
We will call quicksort(+L,-Sorted) with the answer terminated, i.e.:

```
?- quicksort([2,5,3,9,4,6],Ans-[]).
```

Step 3: Substitutions to make the append irrelevant

Replace L2 with [P|R1]

```
append(A-B,B-C,A-C).

quicksort([],A-A).
quicksort([P|T],Sorted-S2) :-
    partition(P,T,L,R),
    quicksort(L,L1-[P|R1]), quicksort(R,R1-R2),
    append(L1-[P|R1],[P|R1]-R2,Sorted-S2).
```

Step 3: Substitutions to make the append irrelevant

Replace Sorted with L1

```
append(A-B,B-C,A-C).

quicksort([],A-A).
quicksort([P|T],L1-S2) :-
    partition(P,T,L,R),
    quicksort(L,L1-[P|R1]), quicksort(R,R1-R2),
    append(L1-[P|R1],[P|R1]-R2,L1-S2).
```

Step 3: Substitutions to make the append irrelevant

Replace S2 with R2

```
append(A-B,B-C,A-C).

quicksort([],A-A).
quicksort([P|T],L1-R2) :-
    partition(P,T,L,R),
    quicksort(L,L1-[P|R1]), quicksort(R,R1-R2),
    append(L1-[P|R1],[P|R1]-R2,L1-R2).
```

Step 3: Substitutions to make the append irrelevant

Replace S2 with R2

```
append(A-B,B-C,A-C).

quicksort([],A-A).
quicksort([P|T],L1-R2) :-
    partition(P,T,L,R),
    quicksort(L,L1-[P|R1]), quicksort(R,R1-R2),
    append(L1-[P|R1],[P|R1]-R2,L1-R2).
    append(A - B ,  B - C, A-C).
```

Step 4: Remove the append because it doesn't do anything any more.

```
% partition(+Pivot,+List,-Left,-Right).
partition(_,[],[],[]).
partition(P,[H|T],[H|L],R) :- H =< P, partition(P,T,L,R).
partition(P,[H|T],L,[H|R]) :- H > P, partition(P,T,L,R).

% quicksort(+List,-DiffList)
quicksort([],A-A).
quicksort([P|T],L1-R2) :-
    partition(P,T,L,R),
    quicksort(L,L1-[P|R1]), quicksort(R,R1-R2).
```


Step 4: Remove the append because it doesn't do anything any more.

```
% partition(+Pivot,+List,-Left,-Right).
partition(_,[],[],[]).
partition(P,[H|T],[H|L],R) :- H =< P, partition(P,T,L,R).
partition(P,[H|T],L,[H|R]) :- H > P, partition(P,T,L,R).

% quicksort(+List,-DiffList)
quicksort([],A-A).
quicksort([P|T],L1-R2) :-
    partition(P,T,L,R),
    quicksort(L,L1-[P|R1]), quicksort(R,R1-R2).
```


```
?- quicksort([2,5,3,9,4,6],Ans-[]).
```

Next time

Videos

- Sudoku
- Constraints

Prolog
Programming in Logic

Lecture #8

Ian Lewis, Andrew Rice

Today's discussion

Videos

- Sudoku
- Constraints

Q: What are the extra-logical equalities?

A: So you know to avoid them:

```
x(A) == x(A)   -- Equivalence. x(A) == x(B) fails.
  D \== 1      -- Not equivalent
Exp1 =:= Exp2  -- E1 is Exp1, E1 is Exp2
Exp1 =\= Exp2  -- \+ E1 is Exp1, E1 is Exp2
   A =@= B     -- variants (<Equivalence >Unification).
     :=        -- ? defined operator, no function
     <=        -- ? undefined (use =<)
unify_with_occurs_check(A,B) -- should be an infix op.

foo :- ..., \+ Term, ...
```

Q: Is [a,b,c|A]-B a difference list?

A: If it's a quiz: answer is NO.

A: If it's a CS conversation, you could reason if:

A = [d,e,f|B], X = [a,b,c|A]-B
means that [a,b,c|A]-B *is* a difference list.

Although, similar is X an integer? yes if X=7 ...

Q: When is Prolog not ML-like?

```
ML factorial:
fun fact 0 = 1
  | fact n = n * fact (n - 1)

f = fact 5

Prolog factorial:
fact(1,1).
fact(N,Fact) :- N > 1, M is N-1, fact(M,Mfact), Fact is N * Mfact.

-? fact(5,F).
```

For a procedure intended to be used deterministically, with ground arguments, there is very little difference apart from syntax.

Type inference is a very clever bit of ML, while Prolog is typeless.

Q: When is Prolog not ML-like?

ML reverse list:

```
fun reverse [] = []
  | reverse (x::xs) = (reverse xs) @ [x]

l = reverse([1,2,3,4])
```

Prolog reverse list:

```
reverse([],[]).
reverse([X|XS],L) :- reverse(XS,XSrev), append(XSrev,[X],L).

?- reverse([1,2,3,4],L).
?- reverse(X, [4,3,2,1]). -- almost.
?- reverse([1,2,X,4],L).
?- reverse([1,2,X,Y],[4,3,2,A]).
```

Q: When is Prolog not ML-like?

Ultimately you can write a unification function in any language, define a data structure representing relations, and create a backtracking algorithm.

At that point you have 'Prolog' embedded in your language of choice.

Or you can modify Prolog to support higher-order functions...

I.e. implement the relational calculus in ML, or the functional calculus in Prolog.

Q: 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]

```
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).
```

[1]Sicstus user manual

Sudoku 4x4 (video)

```
Example

A B 4 D
E 2 G H
I J 1 L
M 3 O P
```

```
range([]).
range([H|T]) :- range(1,5,H), range(T).

range(X,_,X).
range(X,Y,Next) :- N is X+1, N < Y, range(N,Y,Next).

diff(A,B,C,D) :- A=\=B, A=\=C, A=\=D, B=\=C, B=\=D, C=\=D.

rows([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,B,C,D), diff(E,F,G,H), diff(I,J,K,L), diff(M,N,O,P).

cols([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,E,I,M), diff(B,F,J,N), diff(C,G,K,O), diff(D,H,L,P).

boxes([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,B,E,F), diff(C,D,G,H), diff(I,J,M,N), diff(K,L,O,P).

sudoku(L) :- range(L), rows(L), cols(L), boxes(L).
```

```
[trace]  ?- sudoku([A,B,4,D,E,2,G,H,I,J,1,L,M,3,O,P]).
   Call: (8) sudoku([_1472, _1478, 4, _1490, _1496, 2, _1508, _1514|...]) ? creep
   Call: (9) range([_1472, _1478, 4, _1490, _1496, 2, _1508, _1514|...]) ? skip
   Exit: (9) range([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1]) ? creep
   Call: (9) rows([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1]) ? creep
   Call: (10) diff(1, 1, 4, 1) ? creep
   Call: (11) 1=\=1 ? creep
   Fail: (11) 1=\=1 ? creep
   Fail: (10) diff(1, 1, 4, 1) ? creep
   Fail: (9) rows([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1]) ? creep
   Redo: (9) range([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, _1562]) ? creep
   Exit: (9) range([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 2]) ?
```

Sudoku 4x4 (video)

```
range([]).
range([H|T]) :- range(1,5,H), range(T).

range(X,_,X).
range(X,Y,Next) :- N is X+1, N < Y, range(N,Y,Next).

diff(A,B,C,D) :- A=\=B, A=\=C, A=\=D, B=\=C, B=\=D, C=\=D.

rows([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,B,C,D), diff(E,F,G,H), diff(I,J,K,L), diff(M,N,O,P).

cols([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,E,I,M), diff(B,F,J,N), diff(C,G,K,O), diff(D,H,L,P).

boxes([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,B,E,F), diff(C,D,G,H), diff(I,J,M,N), diff(K,L,O,P).

sudoku(L) :- range(L), rows(L), cols(L), boxes(L).
```

```
[trace]  ?- sudoku([A,B,4,D,E,2,G,H,I,J,1,L,M,3,O,P]).
   Call: (8) sudoku([_1472, _1478, 4, _1490, _1496, 2, _1508, _1514|...]) ? creep
   Call: (9) range([_1472, _1478, 4, _1490, _1496, 2, _1508, _1514|...]) ? skip
   Exit: (9) range([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1]) ? creep
   Call: (9) rows([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1]) ? creep
   Call: (10) diff(1, 1, 4, 1) ? creep
   Call: (11) 1=\=1 ? creep
   Fail: (11) 1=\=1 ? creep
   Fail: (10) diff(1, 1, 4, 1) ? creep
   Fail: (9) rows([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1]) ? creep
   Redo: (9) range([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, _1562]) ? creep
   Exit: (9) range([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 2]) ?
```

Sudoku 4x4 (video)

```
range([]).
range([H|T]) :- range(1,5,H), range(T).

range(X,_,X).
range(X,Y,Next) :- N is X+1, N < Y, range(N,Y,Next).

diff(A,B,C,D) :- A=\=B, A=\=C, A=\=D, B=\=C, B=\=D, C=\=D.

rows([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,B,C,D), diff(E,F,G,H), diff(I,J,K,L), diff(M,N,O,P).

cols([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,E,I,M), diff(B,F,J,N), diff(C,G,K,O), diff(D,H,L,P).

boxes([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,B,E,F), diff(C,D,G,H), diff(I,J,M,N), diff(K,L,O,P).

sudoku(L) :- range(L), rows(L), cols(L), boxes(L).
```

```
[trace]  ?- sudoku([A,B,4,D,E,2,G,H,I,J,1,L,M,3,O,P]).
   Call: (8) sudoku([_1472, _1478, 4, _1490, _1496, 2, _1508, _1514|...]) ? creep
   Call: (9) range([_1472, _1478, 4, _1490, _1496, 2, _1508, _1514|...]) ? skip
   Exit: (9) range([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1]) ? creep
   Call: (9) rows([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1]) ? creep
   Call: (10) diff(1, 1, 4, 1) ? creep
   Call: (11) 1=\=1 ? creep
   Fail: (11) 1=\=1 ? creep
   Fail: (10) diff(1, 1, 4, 1) ? creep
   Fail: (9) rows([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1]) ? creep
   Redo: (9) range([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, _1562]) ? creep
   Exit: (9) range([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 2]) ?
```

Sudoku 4x4 (video)

```
range([]).
range([H|T]) :- range(1,5,H), range(T).

range(X,_,X).
range(X,Y,Next) :- N is X+1, N < Y, range(N,Y,Next).

diff(A,B,C,D) :- A=\=B, A=\=C, A=\=D, B=\=C, B=\=D, C=\=D.

rows([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,B,C,D), diff(E,F,G,H), diff(I,J,K,L), diff(M,N,O,P).

cols([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,E,I,M), diff(B,F,J,N), diff(C,G,K,O), diff(D,H,L,P).

boxes([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,B,E,F), diff(C,D,G,H), diff(I,J,M,N), diff(K,L,O,P).

sudoku(L) :- range(L), rows(L), cols(L), boxes(L).
```

```
[trace]  ?- sudoku([A,B,4,D,E,2,G,H,I,J,1,L,M,3,O,P]).
   Call: (8) sudoku([_1472, _1478, 4, _1490, _1496, 2, _1508, _1514|...]) ? creep
   Call: (9) range([_1472, _1478, 4, _1490, _1496, 2, _1508, _1514|...]) ? skip
   Exit: (9) range([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1]) ? creep
   Call: (9) rows([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1]) ? creep
   Call: (10) diff(1, 1, 4, 1) ? creep
   Call: (11) 1=\=1 ? creep
   Fail: (11) 1=\=1 ? creep
   Fail: (10) diff(1, 1, 4, 1) ? creep
   Fail: (9) rows([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1]) ? creep
   Redo: (9) range([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, _1562]) ? creep
   Exit: (9) range([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 2]) ?
```

Sudoku 4x4 (video)

```
range([]).
range([H|T]) :- range(1,5,H), range(T).

range(X,_,X).
range(X,Y,Next) :- N is X+1, N < Y, range(N,Y,Next).

diff(A,B,C,D) :- A=\=B, A=\=C, A=\=D, B=\=C, B=\=D, C=\=D.

rows([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,B,C,D), diff(E,F,G,H), diff(I,J,K,L), diff(M,N,O,P).

cols([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,E,I,M), diff(B,F,J,N), diff(C,G,K,O), diff(D,H,L,P).

boxes([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,B,E,F), diff(C,D,G,H), diff(I,J,M,N), diff(K,L,O,P).

sudoku(L) :- range(L), rows(L), cols(L), boxes(L).
```

```
[trace]  ?- sudoku([A,B,4,D,E,2,G,H,I,J,1,L,M,3,O,P]).
   Call: (8) sudoku([_1472, _1478, 4, _1490, _1496, 2, _1508, _1514|...]) ? creep
   Call: (9) range([_1472, _1478, 4, _1490, _1496, 2, _1508, _1514|...]) ? skip
   Exit: (9) range([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1]) ? creep
   Call: (9) rows([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1]) ? creep
   Call: (10) diff(1, 1, 4, 1) ? creep
   Call: (11) 1=\=1 ? creep
   Fail: (11) 1=\=1 ? creep
   Fail: (10) diff(1, 1, 4, 1) ? creep
   Fail: (9) rows([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1]) ? creep
   Redo: (9) range([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, _1562]) ? creep
   Exit: (9) range([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 2]) ?
```

Sudoku 4x4 (video)

```
range([]).
range([H|T]) :- range(1,5,H), range(T).

range(X,_,X).
range(X,Y,Next) :- N is X+1, N < Y, range(N,Y,Next).

diff(A,B,C,D) :- A=\=B, A=\=C, A=\=D, B=\=C, B=\=D, C=\=D. -- arithmetic not equals

rows([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,B,C,D), diff(E,F,G,H), diff(I,J,K,L), diff(M,N,O,P).

cols([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,E,I,M), diff(B,F,J,N), diff(C,G,K,O), diff(D,H,L,P).

boxes([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,B,E,F), diff(C,D,G,H), diff(I,J,M,N), diff(K,L,O,P).

sudoku(L) :- range(L), rows(L), cols(L), boxes(L).
```

```
[trace]  ?- sudoku([A,B,4,D,E,2,G,H,I,J,1,L,M,3,O,P]).
   Call: (8) sudoku([_1472, _1478, 4, _1490, _1496, 2, _1508, _1514|...]) ? creep
   Call: (9) range([_1472, _1478, 4, _1490, _1496, 2, _1508, _1514|...]) ? skip
   Exit: (9) range([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1]) ? creep
   Call: (9) rows([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1]) ? creep
   Call: (10) diff(1, 1, 4, 1) ? creep
   Call: (11) 1=\=1 ? creep
   Fail: (11) 1=\=1 ? creep
   Fail: (10) diff(1, 1, 4, 1) ? creep
   Fail: (9) rows([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1]) ? creep
   Redo: (9) range([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, _1562]) ? creep
   Exit: (9) range([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 2]) ?
```

Sudoku 4x4 (video)

```
Video efficiency improvement
diff(A,B,C,D) :- perm([1,2,3,4],[A,B,C,D]).

rows([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,B,C,D), diff(E,F,G,H), diff(I,J,K,L), diff(M,N,O,P).

cols([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,E,I,M), diff(B,F,J,N), diff(C,G,K,O), diff(D,H,L,P).

boxes([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,B,E,F), diff(C,D,G,H), diff(I,J,M,N), diff(K,L,O,P).

sudoku(L) :- rows(L), cols(L), boxes(L).
```

```
?- sudoku([A,B,4,D,
           E,2,G,H,
           I,J,1,L,
           M,3,O,P]).
```

Sudoku 4x4 (video)

```
Video efficiency improvement
diff(A,B,C,D) :- perm([1,2,3,4],[A,B,C,D]).

rows([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,B,C,D), diff(E,F,G,H), diff(I,J,K,L), diff(M,N,O,P).

cols([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,E,I,M), diff(B,F,J,N), diff(C,G,K,O), diff(D,H,L,P).

boxes([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,B,E,F), diff(C,D,G,H), diff(I,J,M,N), diff(K,L,O,P).

sudoku(L) :- rows(L), cols(L), boxes(L).
```

```
?- sudoku([A,B,4,D,
           E,2,G,H,
           I,J,1,L,
           M,3,O,P]).
```

Prolog vs ML... Imperative program ?:

```
function sudoku(board) {
      answer = make_board(board)
      if rows(answer) and cols(answer) and boxes(answer)
      then
            return answer
      else
            return fail
```

Sudoku 4x4 (video)

```
Video efficiency improvement
diff(A,B,C,D) :- perm([1,2,3,4],[A,B,C,D]).

rows([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,B,C,D), diff(E,F,G,H), diff(I,J,K,L), diff(M,N,O,P).

cols([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,E,I,M), diff(B,F,J,N), diff(C,G,K,O), diff(D,H,L,P).

boxes([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,B,E,F), diff(C,D,G,H), diff(I,J,M,N), diff(K,L,O,P).

sudoku(L) :- rows(L), cols(L), boxes(L).
```

```
?- sudoku([A,B,4,D,
           E,2,G,H,
           I,J,1,L,
           M,3,O,P]).
```

Prolog vs ML... Imperative program ?:

```
function sudoku(board) {
      answer = make_board(board)
      if rows(answer) and cols(answer) and boxes(answer)
      then
            return answer
      else
            return fail
                                    -- unification and backtracking ???
```

Sudoku 9x9 (Lecture #4)

```
% take(L1,X,L2) succeeds if output list L2 is input list L1 minus element X
take([H|T],H,T).
take([H|T],R,[H|S]) :- take(T,R,S).

% perm(L1,L2) succeeds if output list L2 is a permutation of input list L1
perm([],[]).
perm(A,[R|S]) :- take(A,R,P), perm(P,S).

% gen_digits(L) succeeds if L is a permutation of [1,2,3,4,5,6,7,8,9]
gen_digits(L) :- perm([1,2,3,4,5,6,7,8,9],L).
```

SUDOKU 9x9

```
puzzle1 :- RowA = [5,3,_,_,7,_,_,_,_],
           RowB = [6,_,_,1,9,5,_,_,_],
           RowC = [_,9,8,_,_,_,_,6,_],
           RowD = [8,_,_,_,6,_,_,_,3],
           RowE = [4,_,_,8,_,3,_,_,1],
           RowF = [7,_,_,_,2,_,_,_,6],
           RowG = [_,6,_,_,_,_,2,8,_],
           RowH = [_,_,_,4,1,9,_,_,5],
           RowI = [_,_,_,_,8,_,_,7,9],
           gen_digits(RowA),
           gen_digits(RowB),
           gen_digits(RowC),
           gen_digits(RowD),
           gen_digits(RowE),
           gen_digits(RowF),
           gen_digits(RowG),
           gen_digits(RowH),
           gen_digits(RowI),
           Test fixed numbers,
           Test the 'boxes',
           Test the 'columns',
           Print Solution.
```

```
solve(A,B,C,D,E,F,G,H,I) :- gen_digits(A),
                            test_cols([A,B,C,D,E,F,G,H,I]),
                            gen_digits(B),
                            test_cols([A,B,C,D,E,F,G,H,I]),
                            gen_digits(C),
                            test_boxes(A,B,C),
                            gen_digits(D),
                            test_cols([A,B,C,D,E,F,G,H,I]),
                            gen_digits(E),
                            test_cols([A,B,C,D,E,F,G,H,I]),
                            gen_digits(F),
                            test_cols([A,B,C,D,E,F,G,H,I]),
                            test_boxes(D,E,F),
                            gen_digits(G),
                            test_cols([A,B,C,D,E,F,G,H,I]),
                            gen_digits(H),
                            test_cols([A,B,C,D,E,F,G,H,I]),
                            gen_digits(I),
                            test_boxes(G,H,I),
                            test_cols([A,B,C,D,E,F,G,H,I]).

solve([5,3,_,_,7,_,_,_,_],
      [6,_,_,1,9,5,_,_,_],
      [_,9,8,_,_,_,_,6,_],
      [8,_,_,_,6,_,_,_,3],
      [4,_,_,8,_,3,_,_,1],
      [7,_,_,_,2,_,_,_,6],
      [_,6,_,_,_,_,2,8,_],
      [_,_,_,4,1,9,_,_,5],
      [_,_,_,_,8,_,_,7,9]
     ).
```

Q: Is our countdown mod iterative deepening?

A: ...

Q: Accumulating the Path to a Solution

```
choose(0, L, [], L).
choose(N, [H|T], [H|R], S) :- N > 0, N2 is N-1, choose(N2, T, R, S).
choose(N, [H|T], R, [H|S]) :- N > 0, choose(N, T, R, S).
```

Q: Accumulating the Path to a Solution

```
1. choose(0, L, [], L).
2. choose(N, [H|T], [H|R], S) :- N > 0, N2 is N-1, choose(N2, T, R, S).
3. choose(N, [H|T], R, [H|S]) :- N > 0, choose(N, T, R, S).
```

Step 1. Assign 'number' to each of the clauses in the procedure.

Q: Accumulating the Path to a Solution

```
1. choose(0, L, [], L).
2. choose(N, [H|T], [H|R], S) :- N > 0, N2 is N-1, choose(N2, T, R, S).
3. choose(N, [H|T], R, [H|S]) :- N > 0, choose(N, T, R, S).
```

Step 1. Assign 'number' to each of the clauses in the procedure.

Step 2. Add 2 arguments (+PathIn, -PathOut) to every (non-deterministic) relation, for the accumulated path so far and the path when this clause succeeds, e.g.:

```
choose(0,L,[],L).
```

becomes:

```
choose(0,L,[],L, PathIn, [1|PathIn]).
```

When we initially call 'choose' we will set 'PathIn' to [], and expect the completed path in PathOut:

```
?- choose(2,[a,b,c,d,e],Chosen,Remaining,[],Path).
```

Q: Accumulating the Path to a Solution

```
1. choose(0,L,[],L, PathIn, [1|PathIn]).
2. choose(N, [H|T], [H|R], S) :- N > 0, N2 is N-1, choose(N2, T, R, S).
3. choose(N, [H|T], R, [H|S]) :- N > 0, choose(N, T, R, S).
```

Step 1. Assign 'number' to each of the clauses in the procedure.

Step 2. Add 2 arguments (+PathIn, -PathOut) to every (non-deterministic) relation, for the accumulated path so far and the path when this clause succeeds.

Step 3.

```
choose(N, [H|T], [H|R], S) :-
    N > 0, N2 is N-1,
    choose(N2, T, R, S).
```

becomes:

```
choose(N, [H|T], [H|R], S, PathIn, PathOut) :-
    N > 0, N2 is N-1,
    choose(N2, T, R, S,[2|PathIn],PathOut).
```

Q: Accumulating the Path to a Solution

```
choose(0,L,[],PathIn, [1|PathIn]).

choose(N, [H|T], [H|R], S, PathIn, PathOut) :-
    N > 0, N2 is N-1,
    choose(N2, T, R, S,[2|PathIn],PathOut).

choose(N, [H|T], R, [H|S], PathIn, PathOut) :-
    N > 0,
    choose(N, T, R, S,[3|PathIn],PathOut).

?- choose(2,[a,b,c,d,e],Chosen,Remaining,[],Path).
    Chosen = [a, b],                 Chosen = [a, e],
    Remaining = [c, d, e],   . . .   Remaining = [b, c, d, f, g],
    Path = [1, 2, 2]                 Path = [1, 2, 3, 3, 3, 2]
             Path should be read 'backwards'
```

Q: Accumulating the Path to a Solution

```
choose(0,L,[],PathIn, [1|PathIn]).

choose(N, [H|T], [H|R], S, PathIn, PathOut) :-
    N > 0, N2 is N-1,
    choose(N2, T, R, S,[2|PathIn],PathOut).

choose(N, [H|T], R, [H|S], PathIn, PathOut) :-
    N > 0,
    choose(N, T, R, S,[3|PathIn],PathOut).

?- choose(2,[a,b,c,d,e],Chosen,Remaining,[],[1, 2, 3, 3, 3, 2]).
                                     Chosen = [a, e],
                                     Remaining = [b, c, d, f, g]
```

...DEMO ...

Q: Is our countdown mod iterative deepening?

```
% solve(+N, +SolnAcc, -Soln) succeeds if Soln is a solution
%  to the NxN queens problem and is an extension of
%  the accumulated solution in SolnAcc.
% e.g. :- solve(10, [], Soln), draw(Soln).
solve(N, SolnAcc, SolnAcc) :- length(SolnAcc,N).
solve(N, SolnAcc, Soln) :-
    move(N, SolnAcc, Square),
    solve(N, [Square | SolnAcc], Soln).

% move(+N, +Soln, -Square) acts as a generator for
%  'safe' squares in the next row to those already
%  allocated in the partial solution in Soln.
move(N, [], sq(1, Col)) :- drange(N, Col).
move(N, [sq(I,J) | Rest], sq(Row, Col)) :-
    I < N, Row is I + 1, drange(N, Col), safe(sq(Row, Col), [sq(I,J)|Rest]).

% drange(+N,-X) acts as generator for X = N down to 1.
drange(N, N).
drange(N, X) :- M is N - 1, M > 0, drange(M, X).

% safe_square(+QueenSquare,+Square) succeeds if QueenSquare
%  does not attack Square (or vice-versa).
safe_square(sq(I, J), sq(X, Y)) :- I \== X, J \== Y,
    U1 is I - J, V1 is X - Y, U1 \== V1,
    U2 is I + J, V2 is X + Y, U2 \== V2.

% safe(+Square,+Soln) succeeds if Square does not attack
% any queens in the partial solution Soln.
safe(_, []).
safe(Square, [QueenSquare | L]) :-
    safe_square(QueenSquare, Square), safe(Square, L).
```

N-Queens

?- solve(10,[],Soln).

Q: Is our countdown mod iterative deepening?

```
% solve(+N, +SolnAcc, -Soln) succeeds if Soln is a solution
%  to the NxN queens problem and is an extension of
%  the accumulated solution in SolnAcc.
% e.g. :- solve(10, [], Soln), draw(Soln).
solve(N, SolnAcc, SolnAcc) :- length(SolnAcc,N).
solve(N, SolnAcc, Soln) :-
    move(N, SolnAcc, Square),
    solve(N, [Square | SolnAcc], Soln).

% move(+N, +Soln, -Square) acts as a generator for
%  'safe' squares in the next row to those already
%  allocated in the partial solution in Soln.
move(N, [], sq(1, Col)) :- drange(N, Col).
move(N, [sq(I,J) | Rest], sq(Row, Col)) :-
    I < N, Row is I + 1, drange(N, Col), safe(sq(Row, Col), [sq(I,J)|Rest]).

% drange(+N,-X) acts as generator for X = N down to 1.
drange(N, N).
drange(N, X) :- M is N - 1, M > 0, drange(M, X).

% safe_square(+QueenSquare,+Square) succeeds if QueenSquare
%  does not attack Square (or vice-versa).
safe_square(sq(I, J), sq(X, Y)) :- I \== X, J \== Y,
    U1 is I - J, V1 is X - Y, U1 \== V1,
    U2 is I + J, V2 is X + Y, U2 \== V2.

% safe(+Square,+Soln) succeeds if Square does not attack
% any queens in the partial solution Soln.
safe(_, []).
safe(Square, [QueenSquare | L]) :-
    safe_square(QueenSquare, Square), safe(Square, L).
```

DETERMINISTIC

N-Queens

?- solve(10,[],Soln).

N-Queens with path accumulation

```
solve(N, SolnAcc, SolnAcc, PathIn, [1|PathIn]) :- length(SolnAcc,N).
solve(N, SolnAcc, Soln, PathIn,PathOut) :-
   move(N, SolnAcc, Square,[2|PathIn],PathOut1),
   solve(N, [Square | SolnAcc], Soln, PathOut1, PathOut).

move(N, [], sq(1, Col), PathIn, PathOut) :- drange(N, Col, [1|PathIn], PathOut).
move(N, [sq(I,J) | Rest], sq(Row, Col), PathIn, PathOut) :-
   I < N, Row is I + 1,
   drange(N, Col, [2|PathIn], PathOut), safe(sq(Row, Col), [sq(I,J)|Rest]).

% drange(N,X) acts as generator for X = N down to 1.
drange(N, N, PathIn, [1|PathIn]).
drange(N, X, PathIn, PathOut) :-
      M is N - 1, M > 0,
      drange(M, X, [2|PathIn], PathOut).
```

N-Queens

?- solve(10,[],Soln,[],Path).

ANOTHER DEMO...

Hanoi with path accumulation

```
% Hanoi puzzle
% Towers A, B, C. Each a list of rings 1,2,3 (Head = top).

ok_move([A1|A],[],A,[A1]).
ok_move([A1|A],[B1|B],A,[A1,B1|B]) :- A1 < B1.

% move A -> B
move(state(A,B,C), state(AN,BN,C), PathIn, [1|PathIn]) :- ok_move(A,B,AN,BN).
% move A -> C
move(state(A,B,C), state(AN,B,CN), PathIn, [2|PathIn]) :- ok_move(A,C,AN,CN).
% move B -> A
move(state(A,B,C), state(AN,BN,C), PathIn, [3|PathIn]) :- ok_move(B,A,BN,AN).
% move B -> C
move(state(A,B,C), state(A,BN,CN), PathIn, [4|PathIn]) :- ok_move(B,C,BN,CN).
% move C -> A
move(state(A,B,C), state(AN,B,CN), PathIn, [5|PathIn]) :- ok_move(C,A,CN,AN).


hanoi(States, State_from, State_to, PathIn, PathOut) :-
      move(State_from,State_to, [1|PathIn], PathOut),
      nl,print(States).


hanoi(States, State_from, State_to, PathIn, PathOut) :-
      move(State_from, Next_state, [2|PathIn], PathOut1),
      \+ member(Next_state, States),
      hanoi([Next_state|States],Next_state,State_to, PathOut1, PathOut).


solve(Path) :- hanoi([],state([1,2,3],[],[]),state([],[],[1,2,3]), [], Path).
```

YET ANOTHER DEMO...

Hanoi with depth limit

```
% Hanoi puzzle
% Towers A, B, C. Each a list of rings 1,2,3 (Head = top).

ok_move([A1|A],[],A,[A1]).
ok_move([A1|A],[B1|B],A,[A1,B1|B]) :- A1 < B1.

% move A -> B
move(state(A,B,C), state(AN,BN,C), PathIn, [1|PathIn]) :- length(PathIn,N), N < Limit, ok_move(A,B,AN,BN).

 .   .   .


solve(Path, Limit) :- hanoi([],state([1,2,3],[],[]),state([],[],[1,2,3]), [], Path, Limit).
```

YET ANOTHER DEMO...

End... of... the... course...

I hope you found the format helpful - please fill out the feedback forms!