Prolog
Programming in Logic

Paper 4 Computer Science

Part 1B

Lecture #1

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

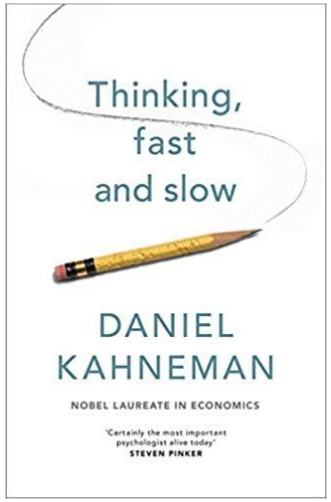
Aims

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

Steps

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 declarative 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.

Query: ?- andy taller X.

A solution: X = ian

What about functional programming?

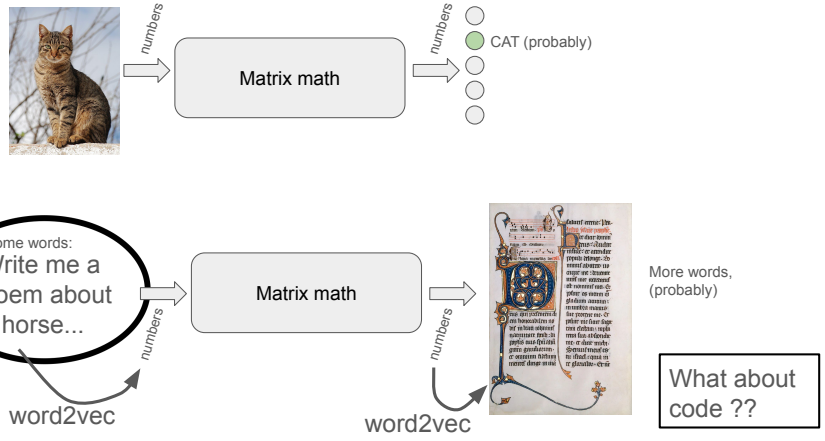
```
fun fact(1) = 1
  | fact(N) = N * fact(N-1).
```

Recursion ?

Declarative ?

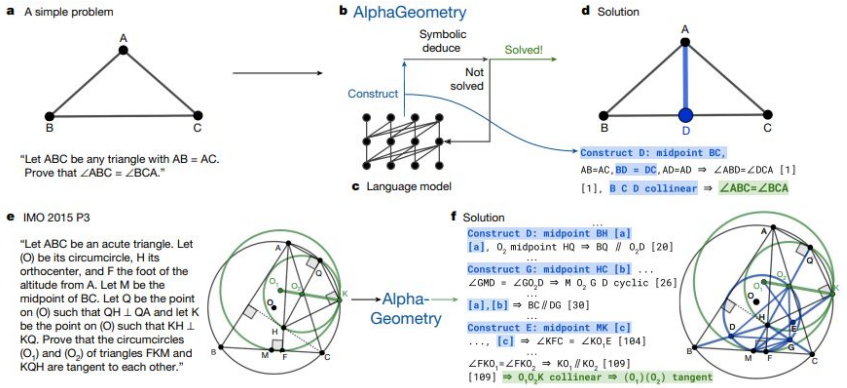
Execution strategy ?

AI/Machine Learning

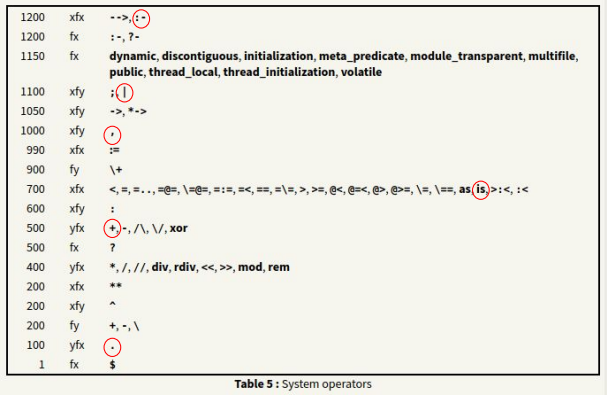


Why study Prolog?

AlphaGeometry: Google DeepMind 17-Jan-2024



Why study Prolog?



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

Table 5 : System operators

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

?- len([a,b,c],L).
L = 3

?-
```

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?

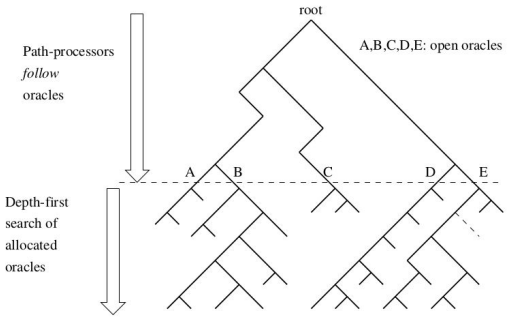


Figure 2.11: Second phase of *breadth-first partitioning*.

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

```
l = [ 1,2,3,4,5 ];

sum = 0;

for (i=0; i<length(L); i++) {

    sum += 1;

}

return sum;
```

DECLARATIVE

```
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.
```

a(Y) here is a COMPOUND TERM

Backtracking

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

a(Y) here is a RELATION

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

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

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]

Q? "Why isn't Prolog more popular?"

≡ "Why study Prolog?"

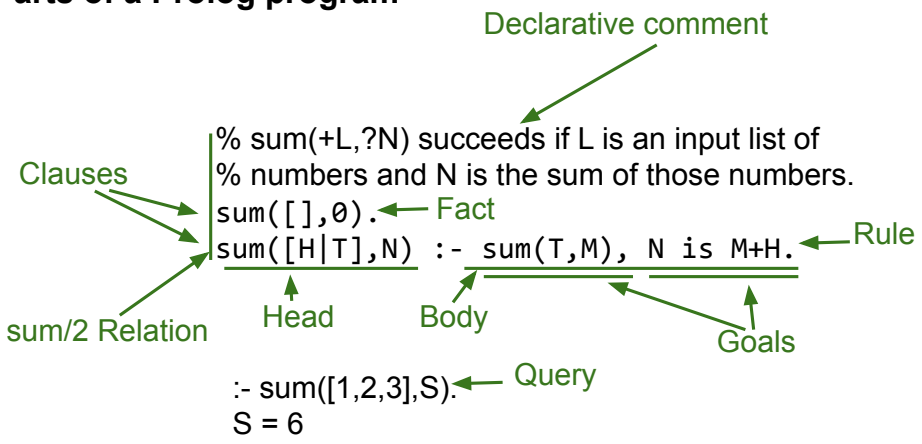
It's not for a website, GUI application, or most systems code. For those, whoever has the most useful libraries wins.

Prolog has niche usage, e.g. verifying hardware.

Any other 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.

Parts of a Prolog program



Programming in Prolog - syntax / vocabulary.

```
1.  tall(andy).
2.  tall(chris).

3.  linked(a,b).
4.  linked(b,c).

5.  path(X,X).
6.  path(X,Z) :- linked(X,Y), path(Y,Z).

7.  ?- path(a,X).
```

terms
- variables (can be anonymous)
- constants / atoms
- compound terms

facts.

rules.

clauses.

head :- body.

body = conjunction of goals.

relation aka predicate.

query = conjunction of goals.

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.

*Note: **data representation** - the problem has 5 houses, each house has 5 properties, no zebra.*

Where's the Zebra ?

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

Note: more mainstream would be e.g. **houses(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.

Where's the Zebra ?

Zebra Puzzle

1. There are five houses.
2. The Englishman lives in the red house.
3. The Spaniard owns the dog.
4. Coffee is drunk in the green house.
5. The Ukrainian drinks tea.
6. The green house is immediately to the right of the ivory house.
7. The Old Gold smoker owns snails.
8. Kools are smoked in the yellow house.
9. Milk is drunk in the middle house.
10. The Norwegian lives in the first house.
11. The man who smokes Chesterfields lives in the house next to the man with the fox.
12. Kools are smoked in the house next to the house where the horse is kept.
13. The Lucky Strike smoker drinks orange juice.
14. The Japanese smokes Parliaments.
15. The Norwegian lives next to the blue house.

Who drinks water? Who owns the zebra? Zebra/2

Zebra puzzle

1. There are five houses.

```
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)).
```

FACTS

Zebra puzzle

QUERY

```
?- 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.
```

Who drinks water? Who owns the zebra?

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).

Query term also-known-as **Goal**

This 'exists' relation provides essential **backtracking**.

Zebra puzzle

This 'exists' relation provides essential **backtracking**.

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

Example query in video:

```
exists(h2,(h1,h2,h3,h4,h5)). % note A in the facts is instantiated to atom h2.
```

Puzzle actually uses a variant of:

```
exists(A, (h1,h2,h3,h4,h5)). % will succeed with A=h1, A=h2, ...
```

In this simple h1,h2... example those terms are ground where in the Zebra puzzle those terms (which unify with A) are partially instantiated.

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),
    ...
```

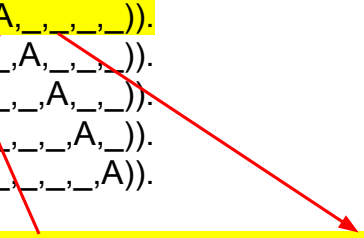
Zebra puzzle

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

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

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),_,_,_,_)),
```

:- **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),
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)).

:-  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,_,_,_),_,_,_))
```

BACKTRACK / RETRY

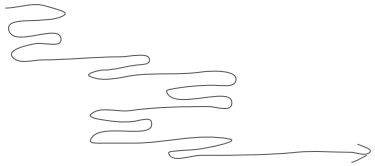
SUCCESS !!

Backtracking

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

This is what DEPTH-FIRST means...

?- goal_1, goal_2, goal_3, goal_4 ...



```
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)).
```

Zebra puzzle

```
:- 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.
```

Note: the order of these goals simply came from the puzzle.

Where’s the Zebra ?

Zebra Puzzle

1. There are five houses.
2. The Englishman lives in the red house.
3. The Spaniard owns the dog.
4. Coffee is drunk in the green house.
5. The Ukrainian drinks tea.
6. The green house is immediately to the right of the ivory house.
7. The Old Gold smoker owns snails.
8. Kools are smoked in the yellow house.
9. Milk is drunk in the middle house.
10. The Norwegian lives in the first house.
11. The man who smokes Chesterfields lives in the house next to the man with the fox.
12. Kools are smoked in the house next to the house where the horse is kept.
13. The Lucky Strike smoker drinks orange juice.
14. The Japanese smokes Parliaments.
15. The Norwegian lives next to the blue house.

Who drinks water? Who owns the zebra? Zebra/2


```
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)).
```


Zebra puzzle

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 - simple extension to the Zebra Puzzle

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).
```

Mainly because at this point we're trying to focus on terms and unification rather than rules.

Unification recap

- Variables will unify with anything
- Atoms will unify with themselves
- Compound terms will unify with other compound terms with the same functor & arity if each of the arguments also unify.

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)

Lists

Spider diagrams

and last/2:

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

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 | []]

Some questions regarding:
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).
```

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],[]).
```

What does this print?

The only 'print' statement requires 1st arg = [ ]

```
a([],R) :- print(R), a(R,[]).
a([H|T],R) :- a(T,[H|R]).
:- a([1,2,3],[]). 1st call
      a([2,3], [1]) 2nd call
         a([3], [2,1]) 3rd call
            a([],[3,2,1]) 4th call
```

Then calls a/2 with args reversed ...

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(T,H).
```

(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(T,H).
```

(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.

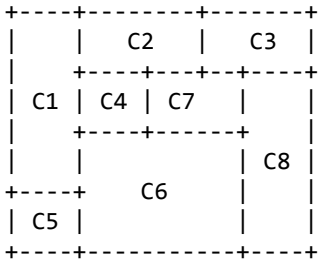
(2) Call with ?- len(L, 0).
L = [ ] ;
?

```
?- len(L,0).

len([], 0).

    L=[],0=0    (succeeds with the 1st clause)

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

    L=[_|T] N=0

                len([],0), 0 is 0+1(fails)

                T=[], M=0

then from 2nd clause: (tries the 2nd clause)

   T=[_|T1], M=N1, len(T1,M1) .. N1 is M1 + 1
```

```
?- len(L,0).

len([], 0).

    L=[],0=0    (succeeds)

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

    L=[_|T] N=0

                len([],0), 0 is 0+1(fails)

                T=[], M=0

then from 2nd clause:

   T=[_|T1], M=N1, len(T1,M1) .. N1 is M1 + 1

                    (will repeat failing after 1st clause & trying 2nd)
```

Today's programming challenge - Map colouring

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




```
color(red).
color(blue).
color(green).
color(yellow).
```

Hint 1: Write down what is true...

You have 4 colours and they are all different…

Hint 1: Write down what is true...

You have 4 colours and they are all different…


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

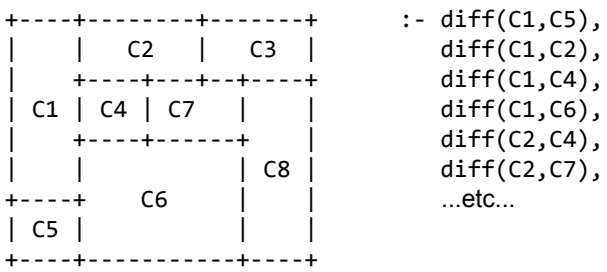
Hint 2: Ask for the answer

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

:

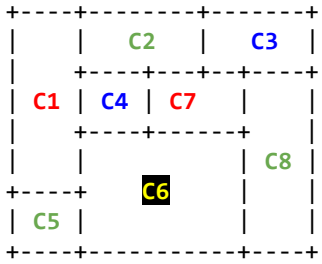
Hint 2: Ask for the answer

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




```
:- diff(C1,C5),
   diff(C1,C2),
   diff(C1,C4),
   diff(C1,C6),
   diff(C2,C4),
   diff(C2,C7),
   ...etc...
```

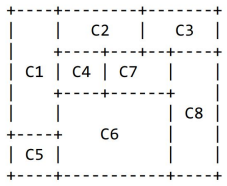
Coloured map



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

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:

- Arithmetic - 'is', space efficiency, Last Call Optimisation, ACCUMULATORS
  - len, len2_acc - spider diagrams
- Backtracking
  - take - spider diagram
  - len backwards - spider diagram

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.

Genius question: Is a query the same as the body of a rule ?

YES - both are a conjunction of goal terms

?- pencil(A), color(A,C), size(A,S).

a :- pencil(A), color(A,C), size(A,S).
?- a.
true

The difference is Prolog throws away the unifications in the body.

pencil_info(A,C,S) :- pencil(A), color(A,C), size(A,S).
?- pencil_info(A,C,S).
A = mypencil
C = green ...

Unification recap

Vars
Atoms/Constants
Compound Terms

```
?- foo(a,X) = foo(Y,b).
X = b,
Y = a.

?- A = foo(a,X), X = b.
A = foo(a,b),
X = b.

?- [user].
q(A) :- B = A, length([1,2,3], B).
^D
?- q(A).
A = 3.
```

Style / Form / Function

```
foo(X,Ans) :- X = 1, symbol(X,S), Ans = S.
```

Style / Form / Function

```
foo(X,Ans) :- X = 1, symbol(X,S), Ans = S.

foo(1,S) :- symbol(1,S).
```

Style / Form / Function

```
foo(X,Ans) :- X = 1, Ans = a ; X = 2, Ans = b.
```

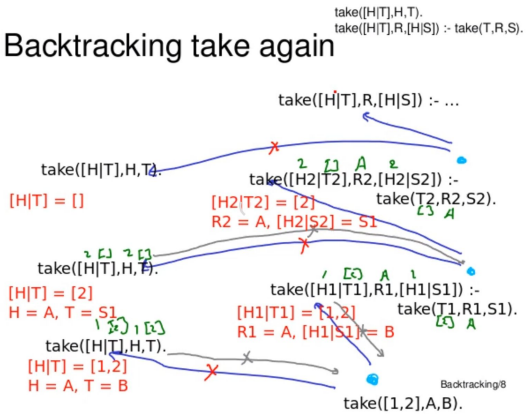
Style / Form / Function

OR

```
foo(X,Ans) :- X = 1, Ans = a ; X = 2, Ans = b.

Trying to recreate the functional style:
    foo(X) { if (X==1) then a elif X = 2 then b }

Instead:

foo(1,a).
foo(2,b).
```

This will always be a mistake: `q([a|T],[H|T2]) :- H = b, ...`

Depth-first, left-to-right?



SEARCH: Depth-first, left-to-right?

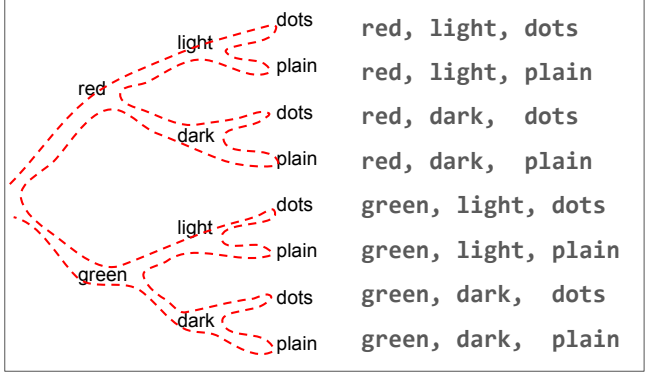
```
color(red).
color(green).

tone(light).
tone(dark).

pattern(dots).
pattern(plain).
```

```
?- color(C),    tone(T),    pattern(P).
   C = red, T = light, P = dots ;
   C = red, T = light, P = plain ;
   C = red, T = dark,  P = dots ;
   ...
```

SEARCH: Depth-first, left-to-right?

```
color(red).
color(green).

tone(light).
tone(dark).

pattern(dots).
pattern(plain).
```

```
?- color(C), tone(T), pattern(P).
```




```
red, light, dots
red, light, plain
red, dark,  dots
red, dark,  plain
green, light, dots
green, light, plain
green, dark,  dots
green, dark,  plain
```

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 number 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 number or variable

Arithmetic: What happens here?

?- A = 1+1, S is A.

Arithmetic: What happens here?

?- A = 1+1, S is A.

A = 1+1

S = 2

?-

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

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).
```

```
p(1).
p(2).

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

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. Prolog has additional optimisations e.g. for orthogonal heads:

```
foo([H|T],Acc, N) :- NewAcc is Acc+1, foo(T,NewAcc,N).
foo([], Acc, Acc).
```

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] pun intended!

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.

Three clauses defining our relation f/1:

```
f(A) :- a(A), (1) f(A).
f(A) :- b(A), (2) f(A).
f(A) :- c(A), (3) f(A).
```

Is f/1 determinate at points (1), (2) or (3)?

Accumulators

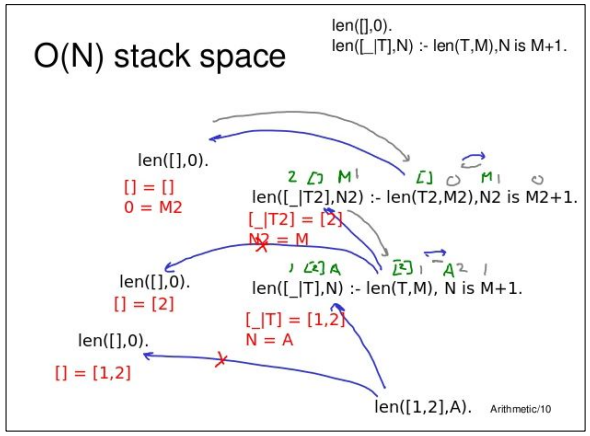
A space-efficient way of passing a partial result through a Prolog computation.

```
% 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).

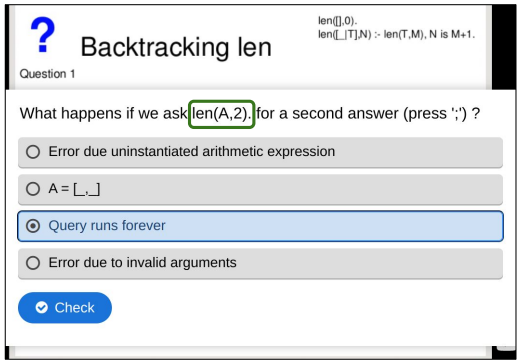
Accumulators

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



Accumulators

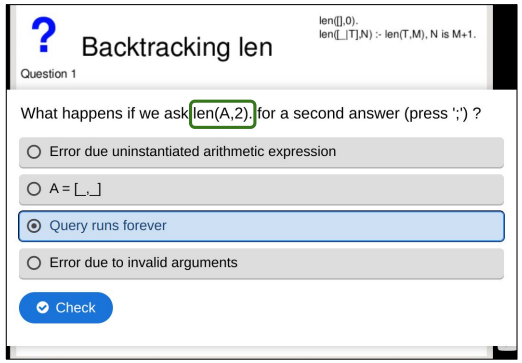
```
len([],   0).
len([_|T],N) :- len(T,M), N is M + 1.
```



Accumulators

```
   A    2
len([],   0).
len([_|T],N) :- len(T,M), N is M + 1.
```

```
2 is 0 + 1 FAIL
2 is 1 + 1 SUCCEED
2 is 2 + 1 FAIL
      ...  FAILS
```



Accumulators

Video example: same functor, different arities...

Use an accumulator for O(1) space

```
len2([],Acc,Acc).
len2([_|T],Acc,R) :-  B is Acc + 1,
                      len2(T,B,R).

len2(List,R) :- len2(List,0,R).
```

len2/3

len2/2

Arithmetic/11

Styles notes:

Scope of variables is only within clause.

* If clause has only one list call it L.

* If clause has multiple lists use L1, L2 ..

* If you're unifying with head and tail, [H|T]

* If you're expecting a number, call the var N.

* Refer to len2/2 and len2/3 (or don't...)

* Comment

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

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).
```

ARGUMENT MODES IN YOUR COMMENT:

+ => INSTANTIATED
? => INSTANTIATED OR A VARIABLE

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

Accumulators - Spiders vs. Traces

(1) len_acc([],Acc,Acc).
(2) len_acc([_|T],Acc, N) :- A1 is Acc + 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([],Acc,Acc).
(2) len_acc([_|T],Acc, N) :- A1 is Acc + 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], Acc = 0, N=N … A1 is 0+1, len_acc([b,c],1,N).

Accumulators

(1) len_acc([],Acc,Acc).
(2) len_acc([_|T],Acc, N) :- A1 is Acc + 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], Acc = 0, N=N … A1 is 0+1, len_acc([b,c],1,N).
Try: (1) - fail
Try: (2) T = [c], Acc = 1, N=N … A1 is 1+1, len_acc([c],2,N).

Accumulators

(1) len_acc([],Acc,Acc).
(2) len_acc([_|T],Acc, N) :- A1 is Acc + 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], Acc = 0, N=N … A1 is 0+1, len_acc([b,c],1,N).
Try: (1) - fail
Try: (2) T = [c], Acc = 1, N=N … A1 is 1+1, len_acc([c],2,N).
Try: (1) - fail
Try: (2) T = [], Acc = 2, N=N … A1 is 2+1, len_acc([],3,N).

Accumulators

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

Note: in the exam don't use built-in relations (such as member, append) unless the question explicitly permits it. The good news is if you know how to write those relations (it would be worrying if you didn't) you'll pick up easy marks.

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).

(1) ?- app([a,b,c],[1,2,3],L).
L = [a,b,c,1,2,3].
(2) ?- app(X,[1,2,3],[a,b,c,1,2,3])
???
(3) ?- app(X,Y,[a,b,c]).
???
(4) ?- 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).
```

(1) Base Case
(2) Recursive Case
Can use inductive reasoning, LCO appies.

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).
```

note base case 'copies' accumulator to 'output'

Can use inductive reasoning, LCO appies.

Accumulators: List reverse - another (classic) example

Can step through as seen with len_acc previously (we won't do it for rev_acc here):

Accumulators

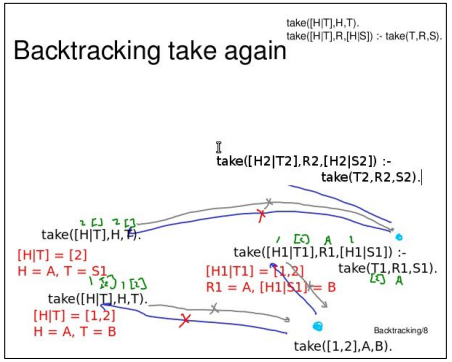
```
(1)   len_acc([],A,A).
(2)   len_acc([_|T],A, N) :- A1 is A + 1, len_acc(T,A1,N).

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/3 recap. (another classic)

```
% 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/2 = A non-deterministic relation

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



Here's Andy's Spider Diagram

Backtracking

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

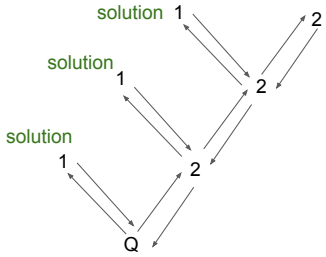
```
[trace] ?- take([a,b,c],X,L).
   Call: (8) take([a, b, c], _4088, _4090) ? creep
   Exit: (8) take([a, b, c], a, [b, c]) ? creep
X = a,
L = [b, c] ;
   Redo: (8) take([a, b, c], _4088, _4090) ? creep
   Call: (9) take([b, c], _4088, _4356) ? creep
   Exit: (9) take([b, c], b, [c]) ? creep
   Exit: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



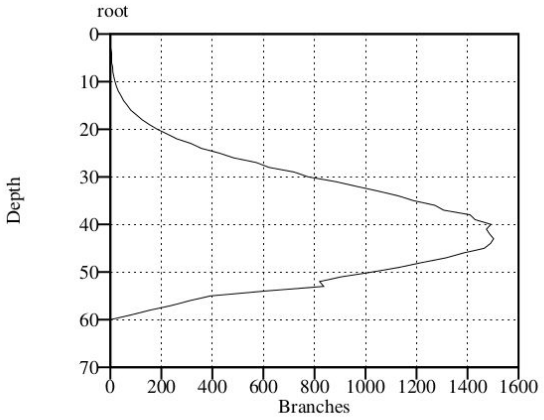


Figure 3.1: Search tree for 8 queens problem.

COMPUTER SCIENCE TRIPOS Part IB 75%, Part II 50% – 2022

10 Prolog (ijl20)

When answering this question ... You should avoid unnecessary use of cut (`!`) and not use extra-logical relations such as `findall`, `assertz` and `not` (`\+`). Built-in library relations should not be assumed.

Avoid unnecessary use of cut (`!`) and not use extra-logical relations such as `findall`, `assertz` and `not` (`\+`). Built-in library relations should not be assumed.

```
fib(0, 0).

fib(1, 1).

fib(N, F) :-
    N1 is N - 1,
    N2 is N - 2,
    fib(N1, F1),
    fib(N2, F2),
    F is F1 + F2.
```

Avoid unnecessary use of cut (`!`) and not use extra-logical relations such as `findall`, `assertz` and `not` (`\+`). Built-in library relations should not be assumed.

```
fib(0, 0) :- !.        DO NOT DO THIS!
fib(1, 1) :- !.

fib(N, F) :-
    N1 is N - 1,
    N2 is N - 2,
    fib(N1, F1),
    fib(N2, F2),
    F is F1 + F2.
```

```
fib(0, 0).             DO THIS!
fib(1, 1).

fib(N, F) :-
    N > 1,             GUARD
    N1 is N - 1,
    N2 is N - 2,
    fib(N1, F1),
    fib(N2, F2),
    F is F1 + F2.
```

Next time

Videos

- Generate and Test - ESSENTIAL PROLOG
- Symbolic evaluation of arithmetic
  - flexibility of Prolog
  - another spider diagram

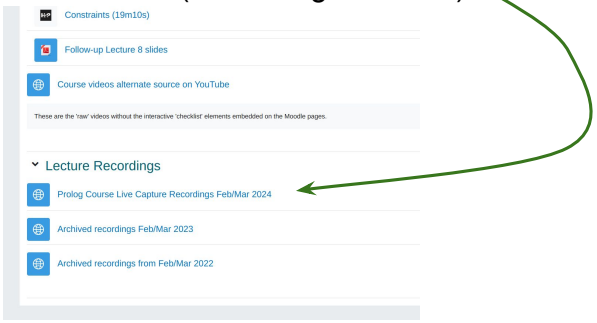
Prolog
Programming in Logic

Lecture #4

Ian Lewis, Andrew Rice

Q. Is this like a normal course with videos?

A. Content in two parts:
* the Prolog instructional videos
* the LT1 lectures (also being recorded)



Q. Argument 'modes' - we use these in *comments*

++	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?

A1. 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.

A2. You're interested in the thing you're accumulating.

Without accumulator vars, the Prolog 'solution' does not include the cumulative evidence of how it derived that solution (which is contained on the execution stack). E.g. (theory) steps in a proof, (practical) sequence of nodes in a graph forming a path.

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

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

A. No. Prolog doesn't lack functions. These are deterministic relations which can be implemented by adding a deterministic 'evaluation' relation to your code or *flattening* the expression and using standard Prolog.

```
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...

Course Outline

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

Today's discussion

Videos:

Generate and Test (Dutch Flag, 8queens, anagram)

Symbolic (Eval)

Today's discussion

Videos:

Generate and Test (Dutch Flag, 8queens, anagram)

[w,b,r,w,w,b,r,r] ⟶ [r,r,r,w,w,w,b,b]

Symbolic (Eval)

plus(1,mult(4,5)) ⟶ 21

Plus we'll look at Sodoku as an example of data structure + generate & test.

Symbolic Evaluation: eval, reduce, flatten

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

Function support - using compound terms (:- op(1200,fx,fun).):
    fun fact(1) = 1;
        fact(N) = N * fact(N-1).

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

Eval relation (run time) vs. Flattening (compile time):
    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
```

Dutch Flag - Generate & Test

Pause to re-visit `take/3` and `perm/2`...

```
% take(+L1,?X,?L2) succeeds if output list L2 is input list L1 less element X
take([H|T],H,T).                    % BASE CASE      (correct?)
take([H|T],X,[H|R]) :- take(T,X,R).   % RECURSIVE CASE (correct?)

% perm(+L1,?L2) succeeds if output list L2 is a permutation of input list L1
perm([],[]).                              % BASE CASE
perm(L,[H|T]) :- take(L,H,R), perm(R,T).   % RECURSIVE CASE
```

Dutch Flag

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

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

```
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([]).
```

Flag

LIST CONTAINS AT LEAST ONE OF EACH COLOUR

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

L = [r,w,b,b]

...
```

Flag - checkBlue

LIST CONTAINS AT LEAST ONE OF EACH COLOUR

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

L = [r,w,b,b]

...
```

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

Note implicitly in this example work through, we are changing the order of the definitions of the relations. This has NO impact on the meaning of declarative Prolog programs.

Changing the order of the clauses in a relation will have a BIG impact on the meaning of the Prolog relation, unless the facts and rule heads are orthogonal with regard to the intended argument modes.

Flag - checkWhite

LIST CONTAINS AT LEAST ONE OF EACH COLOUR

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

L = [r,w,b,b]

...
```

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

% checkWhite(L) succeeds if L is whites followed by blues,
% or the head of L is blue, followed by blues or [].
checkWhite([w|T]) :- checkWhite(T).
checkWhite([b|T]) :- checkBlue(T).
```

Flag - checkRed

LIST CONTAINS AT LEAST ONE OF EACH COLOUR

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

L = [r,w,b,b]

...
```

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

% checkWhite(L) succeeds if L is whites followed by blues,
% or the head of L is blue, followed by blues or [].
checkWhite([w|T]) :- checkWhite(T).
checkWhite([b|T]) :- checkBlue(T).

% checkRed succeeds if L is reds followed by whites and then
% blues, or L is whites followed by blues.
checkRed([r|T]) :- checkRed(T).
checkRed([w|T]) :- checkWhite(T).
```

Flag - solution

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

L = [r,w,b,b]

...
```

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

% checkWhite(L) succeeds if L is whites followed by blues,
% or the head of L is blue, followed by blues or [].
checkWhite([w|T]) :- checkWhite(T).
checkWhite([b|T]) :- checkBlue(T).

% checkRed succeeds if L is reds followed by whites and then
% blues, or L is whites followed by blues.
checkRed([r|T]) :- checkRed(T).
checkRed([w|T]) :- checkWhite(T).

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

Dutch Flag - as in video

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

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

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

checkColours(List) :- checkRed(List). % can replace call to checkColors with checkRed

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([]).
```

Dutch Flag - is this a bug?

```
?- flag([white,blue],L)

L = [white,blue]
```

```
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([]).
```

Dutch Flag - is this a bug?

```
?- flag([white,blue],L)

L = [white,blue]
```

```
flag(In,Out) :-
   perm(In,Out),
   checkColours(Out).

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([]).
```

Individually, each of these relations succeeds even if the input List does NOT start with their intended colour.

This is not an issue with `checkBlue` or `checkWhite` because they are only ever called with the intended colour ALREADY identified prior to the call.

But it is an issue for `checkRed`.

Dutch Flag - re-factored code.

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

% red_white_blue(+L) succeeds if input L is reds.. whites.. blues.
red_white_blue([r|T]) :- red_white_blue(T).
red_white_blue([r|T]) :- white_blue(T).

% white_blue(+L) succeeds if input L is whites.. blues.
white_blue([w|T]) :- white_blue(T).
white_blue([w|T]) :- blue(T).

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

```
?- flag([b,w,b,r,w],F).
F = [r, w, w, b, b] ...
```

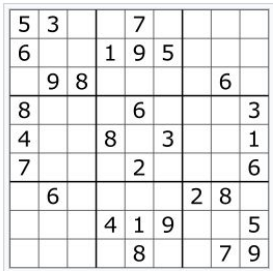
SUDOKU 9x9

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

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

SUDOKU 9x9

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



```
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]
                ).
```

Immediately see choice of data structure with solve/9, and each row as a 9-element list.

gen_digits using perm (& perm using take)

```
% take(+L1,?X,?L2) succeeds if output list L2 is input list L1 less element X
take([H|T],H,T).                          % BASE CASE
take([H|T],X,[H|R]) :- take(T,X,R).     % RECURSIVE CASE

% perm(+L1,?L2) succeeds if output list L2 is a permutation of input list L1
perm([],[]).                                  % BASE CASE
perm(L,[H|T]) :- take(L,H,R), perm(R,T).     % RECURSIVE CASE

% 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).

?- gen_digits(L).
L = [1, 2, 3, 4, 5, 6, 7, 8, 9] ;
L = [1, 2, 3, 4, 5, 6, 7, 9, 8] ;
L = [1, 2, 3, 4, 5, 6, 8, 7, 9] ;
...
```

gen_digits using perm (& perm using take)

```
% take(+L1,?X,?L2) succeeds if output list L2 is input list L1 less element X
take([H|T],H,T).                          % BASE CASE
take([H|T],X,[H|R]) :- take(T,X,R).     % RECURSIVE CASE

% perm(+L1,?L2) succeeds if output list L2 is a permutation of input list L1
perm([],[]).                                  % BASE CASE
perm(L,[H|T]) :- take(L,H,R), perm(R,T).     % RECURSIVE CASE

% 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).

?- gen_digits([9,8,7,A,5,4,B,2,1]).
            [9,8,7,3,5,4,6,2,1] ;   A = 3, B = 6 ...
            [9,8,7,6,5,4,3,2,1] ;   A = 6, B = 3 ...
?
```

gen_digits using perm (& perm using take)

```
% take(+L1,?X,?L2) succeeds if output list L2 is input list L1 less element X
take([H|T],H,T).                          % BASE CASE
take([H|T],X,[H|R]) :- take(T,X,R).     % RECURSIVE CASE

% perm(+L1,?L2) succeeds if output list L2 is a permutation of input list L1
perm([],[]).                                  % BASE CASE
perm(L,[H|T]) :- take(L,H,R), perm(R,T).     % RECURSIVE CASE

% 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).

?- gen_digits([9,8,7,A,5,4,B,2,1]). → perm([1,2,3,4,5,6,7,8,9],[9,8,7,A,5,4,B,2,1])
                                          L                 [H|T]
                                    take([1,2,3,4,5,6,7,8,9],9,R),perm(R,[8,7,A,5,4,B,2,1])
                                    will not unify with first clause of take
...
```

gen_digits using perm (& perm using take)

```
% take(+L1,?X,?L2) succeeds if output list L2 is input list L1 less element X
take([H|T],H,T).                        % BASE CASE
take([H|T],X,[H|R]) :- take(T,X,R).     % RECURSIVE CASE

% perm(+L1,?L2) succeeds if output list L2 is a permutation of input list L1
perm([],[]).                                % BASE CASE
perm(L,[H|T]) :- take(L,H,R), perm(R,T).    % RECURSIVE CASE

% 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).

?- gen_digits([9,8,7,A,5,4,B,2,1]).
perm generates [9,8,7,6,5,4,3,2,1] i.e.
A = 6
B = 3 ;
then perm generates [9,8,7,3,5,4,6,2,1] i.e.
A = 3, B = 6
```

gen_digits using perm (& perm using take)

```
% take(+L1,?X,?L2) succeeds if output list L2 is input list L1 less element X
take([H|T],H,T).                        % BASE CASE
take([H|T],X,[H|R]) :- take(T,X,R).     % RECURSIVE CASE

% perm(+L1,?L2) succeeds if output list L2 is a permutation of input list L1
perm([],[]).                                % BASE CASE
perm(L,[H|T]) :- take(L,H,R), perm(R,T).    % RECURSIVE CASE

% 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).

?- gen_digits([9,8,7,A,5,4,B,2,1]).
A = 6, B = 3 ;
A = 3, B = 6 ;
false
```

SUDOKU 9x9

```
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],
```

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

```
           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 fixed/given numbers,
           Test the 'boxes',
           Test the 'columns',
           Test the 'rows',
           Print Solution.
```

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

```
puzzle1 :- A = [5,3,1,2,7,4,6,8,9],
           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),
           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 fixed/given numbers,
           Test the 'boxes',
           Test the 'columns',
           Test the 'rows',
           Print Solution.
```

```
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 fixed/given numbers,
Test the ‘boxes’,
Test the ‘columns’,
Test the 'rows',
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 FIXED/GIVEN numbers,
Test the ‘boxes’,
Test the ‘columns’,
Test the 'rows',
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’,
Test the 'rows',
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’,
Test the 'rows',
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',
Test the 'rows',
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',
Test the 'rows',
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.
```

Let's look at the 'test the columns'

Remember the Zebra puzzle?

We took the relation calls in the query and re-ordered them to make the generate & test nature of the solution method more explicit.

We'll do something similar in reverse to make our search process less naive.

```
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'
```

Note can test columns after row A...

```
[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]
```

This is already going to FAIL *if* we can write a **test_digits/1** that is happy to ignore free vars.

```
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'
```

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 or is [] and
%    all the digits in L1 are unique from input list L2
test_set([],_).                                          % BASE CASE
test_set([H|T], Digits) :- var(H),                       % VARS OK
                           test_set(T,Digits).
test_set([H|T], Digits) :- ground(H),                    % TAKE 1..9
                           take(Digits,H,RemainingDigits),
                           test_set(T,RemainingDigits).
```

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) :- perm([1,2,3,4,5,6,7,8,9],L).
```

Did you think of that definition?

Note that perm will deliver multiple results if there are variables in L.

The other thing we're going to need is a relation that gives us the vertical stripes through a set of equal length lists.

```
[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]
```

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

The other thing we're going to need is a relation that gives us the vertical stripes through a set of equal length lists.

```
[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]
```

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

How would you 'write a program' | 'declare some facts/rules' giving you heads(+LL,Heads,Tails) ?

How much 'code' is this going to be?

heads(+LL, ?LHeads, ?LTails)

```
[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]
```

```
% heads(+LL, ?LHeads, ?LTails) succeeds if:
%   LL is an input list of equal-length 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([[H1|T1]|T],[H1|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]]
```

Testing the columns

OK, so now we have :

test_digits(+L)
where L is any list of 9 digits & vars.

and

heads(+LL, ?Heads, ?Tails)
where LL is a list of equal-length lists
and Heads will be a list containing all the heads
and tails is a list of the remaining lists.

So unsurprisingly we are going to apply `test_digits` to the Heads.

test_cols(Rows)

```
% test_cols(Rows) succeeds if:
%    input Rows is a list of equal-length lists of digits and variables
%    each 'column' of digits and variables contain unique digits from [1..9]
test_cols([[]|_]).                          % BASE CASE
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]
```

We don't need a `test_col` (singular) because that is simply `test_digits`.

Note slight subtlety in base case - we end up with list of empty lists, not [].

```
[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]
```

Note here after `gen_digits(B)`, `test_cols` will FAIL and backtrack to another solution from `gen_digits(B)` until it succeeds.

```
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_cols([A,B,C,D,E,F,G,H,I]),
          gen_digits(B), B = [6,2,3,1,9,5,4,8,7]
          test_cols([A,B,C,D,E,F,G,H,I]),
          ...
          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]
```

Now we have 3 rows, we can pro-actively test the 'boxes'...

```
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_cols([A,B,C,D,E,F,G,H,I]),
          gen_digits(B), B = [6,2,3,1,9,5,4,8,7]
          test_cols([A,B,C,D,E,F,G,H,I]),
          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]
```

Note at this point we have 3 rows, so we have already generated 3 BOXES that can be tested.

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

We'll call test_boxes each time we have 3 rows, i.e. we'll test 3 boxes each time...

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]).
```

Unsurprisingly, we are going to apply `test_digits` to the boxes.

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]
     ).
```

So here's the complete program ...

SUDOKU 9x9

```
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]
                ).
```

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

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

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

Course Outline

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

Next time

Videos

- Cut
- Negation
- Databases

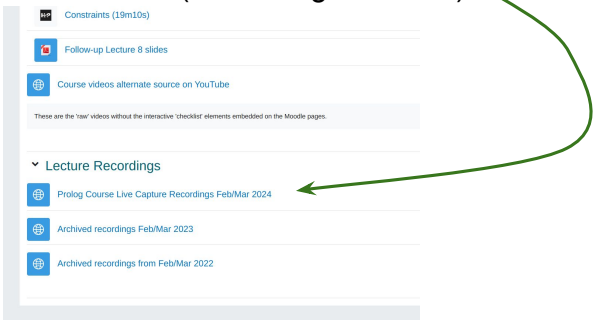
Prolog
Programming in Logic

Lecture #4

Ian Lewis, Andrew Rice

Q. Is this like a normal course with videos?

A. Content in two parts:
* the Prolog instructional videos
* the LT1 lectures (also being recorded)



Q. Argument 'modes' - we use these in *comments*

++	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?

A1. 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.

A2. You're interested in the thing you're accumulating.

Without accumulator vars, the Prolog 'solution' does not include the cumulative evidence of how it derived that solution (which is contained on the execution stack). E.g. (theory) steps in a proof, (practical) sequence of nodes in a graph forming a path.

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

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

A. No. Prolog doesn't lack functions. These are deterministic relations which can be implemented by adding a deterministic 'evaluation' relation to your code or *flattening* the expression and using standard Prolog.

```
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...

Course Outline

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

Today's discussion

Videos:

Generate and Test (Dutch Flag, 8queens, anagram)

Symbolic (Eval)

Today's discussion

Videos:

Generate and Test (Dutch Flag, 8queens, anagram)

[w,b,r,w,w,b,r,r] ⟶ [r,r,r,w,w,w,b,b]

Symbolic (Eval)

plus(1,mult(4,5)) ⟶ 21

Plus we'll look at Sodoku as an example of data structure + generate & test.

Symbolic Evaluation: eval, reduce, flatten

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

Function support - using compound terms (:- op(1200,fx,fun).):
    fun fact(1) = 1;
        fact(N) = N * fact(N-1).

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

Eval relation (run time) vs. Flattening (compile time):
    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
```

Dutch Flag - Generate & Test

Pause to re-visit `take/3` and `perm/2`...

```
% take(+L1,?X,?L2) succeeds if output list L2 is input list L1 less element X
take([H|T],H,T).                    % BASE CASE      (correct?)
take([H|T],X,[H|R]) :- take(T,X,R).   % RECURSIVE CASE (correct?)

% perm(+L1,?L2) succeeds if output list L2 is a permutation of input list L1
perm([],[]).                              % BASE CASE
perm(L,[H|T]) :- take(L,H,R), perm(R,T).   % RECURSIVE CASE
```

Dutch Flag

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

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

```
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([]).
```

Flag

LIST CONTAINS AT LEAST ONE OF EACH COLOUR

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

L = [r,w,b,b]

...
```

Flag - checkBlue

LIST CONTAINS AT LEAST ONE OF EACH COLOUR

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

L = [r,w,b,b]

...
```

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

Note implicitly in this example work through, we are changing the order of the definitions of the relations. This has NO impact on the meaning of declarative Prolog programs.

Changing the order of the clauses in a relation will have a BIG impact on the meaning of the Prolog relation, unless the facts and rule heads are orthogonal with regard to the intended argument modes.

Flag - checkWhite

LIST CONTAINS AT LEAST ONE OF EACH COLOUR

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

L = [r,w,b,b]

...
```

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

% checkWhite(L) succeeds if L is whites followed by blues,
% or the head of L is blue, followed by blues or [].
checkWhite([w|T]) :- checkWhite(T).
checkWhite([b|T]) :- checkBlue(T).
```

Flag - checkRed

LIST CONTAINS AT LEAST ONE OF EACH COLOUR

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

L = [r,w,b,b]

...
```

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

% checkWhite(L) succeeds if L is whites followed by blues,
% or the head of L is blue, followed by blues or [].
checkWhite([w|T]) :- checkWhite(T).
checkWhite([b|T]) :- checkBlue(T).

% checkRed succeeds if L is reds followed by whites and then
% blues, or L is whites followed by blues.
checkRed([r|T]) :- checkRed(T).
checkRed([w|T]) :- checkWhite(T).
```

Flag - solution

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

L = [r,w,b,b]

...
```

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

% checkWhite(L) succeeds if L is whites followed by blues,
% or the head of L is blue, followed by blues or [].
checkWhite([w|T]) :- checkWhite(T).
checkWhite([b|T]) :- checkBlue(T).

% checkRed succeeds if L is reds followed by whites and then
% blues, or L is whites followed by blues.
checkRed([r|T]) :- checkRed(T).
checkRed([w|T]) :- checkWhite(T).

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

Dutch Flag - as in video

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

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

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

checkColours(List) :- checkRed(List). % can replace call to checkColors with checkRed

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([]).
```

Dutch Flag - is this a bug?

```
?- flag([white,blue],L)

L = [white,blue]
```

```
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([]).
```

Dutch Flag - is this a bug?

```
?- flag([white,blue],L)

L = [white,blue]
```

```
flag(In,Out) :-
   perm(In,Out),
   checkColours(Out).

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([]).
```

Individually, each of these relations succeeds even if the input List does NOT start with their intended colour.

This is not an issue with `checkBlue` or `checkWhite` because they are only ever called with the intended colour ALREADY identified prior to the call.

But it is an issue for `checkRed`.

Dutch Flag - re-factored code.

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

% red_white_blue(+L) succeeds if input L is reds.. whites.. blues.
red_white_blue([r|T]) :- red_white_blue(T).
red_white_blue([r|T]) :- white_blue(T).

% white_blue(+L) succeeds if input L is whites.. blues.
white_blue([w|T]) :- white_blue(T).
white_blue([w|T]) :- blue(T).

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

```
?- flag([b,w,b,r,w],F).
F = [r, w, w, b, b] ...
```

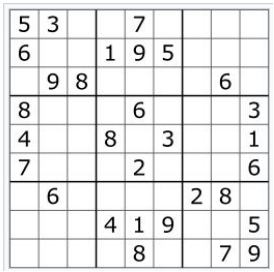
SUDOKU 9x9

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

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

SUDOKU 9x9

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



```
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]
                ).
```

Immediately see choice of data structure with solve/9, and each row as a 9-element list.

gen_digits using perm (& perm using take)

```
% take(+L1,?X,?L2) succeeds if output list L2 is input list L1 less element X
take([H|T],H,T).                          % BASE CASE
take([H|T],X,[H|R]) :- take(T,X,R).       % RECURSIVE CASE

% perm(+L1,?L2) succeeds if output list L2 is a permutation of input list L1
perm([],[]).                                  % BASE CASE
perm(L,[H|T]) :- take(L,H,R), perm(R,T).      % RECURSIVE CASE

% 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).

?- gen_digits(L).
L = [1, 2, 3, 4, 5, 6, 7, 8, 9] ;
L = [1, 2, 3, 4, 5, 6, 7, 9, 8] ;
L = [1, 2, 3, 4, 5, 6, 8, 7, 9] ;
...
```

gen_digits using perm (& perm using take)

```
% take(+L1,?X,?L2) succeeds if output list L2 is input list L1 less element X
take([H|T],H,T).                          % BASE CASE
take([H|T],X,[H|R]) :- take(T,X,R).       % RECURSIVE CASE

% perm(+L1,?L2) succeeds if output list L2 is a permutation of input list L1
perm([],[]).                                  % BASE CASE
perm(L,[H|T]) :- take(L,H,R), perm(R,T).      % RECURSIVE CASE

% 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).

?- gen_digits([9,8,7,A,5,4,B,2,1]).
L = [1, 2, 3, 4, 5, 6, 7, 8, 9] ;
L = [1, 2, 3, 4, 5, 6, 7, 9, 8] ;
L = [1, 2, 3, 4, 5, 6, 8, 7, 9] ;
...
```

gen_digits using perm (& perm using take)

```
% take(+L1,?X,?L2) succeeds if output list L2 is input list L1 less element X
take([H|T],H,T).                          % BASE CASE
take([H|T],X,[H|R]) :- take(T,X,R).       % RECURSIVE CASE

% perm(+L1,?L2) succeeds if output list L2 is a permutation of input list L1
perm([],[]).                                  % BASE CASE
perm(L,[H|T]) :- take(L,H,R), perm(R,T).      % RECURSIVE CASE

% 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).

?- gen_digits([9,8,7,A,5,4,B,2,1]).   ──► perm([1,2,3,4,5,6,7,8,9],[9,8,7,A,5,4,B,2,1])
L = [1, 2, 3, 4, 5, 6, 7, 8, 9] ;                L          [H|T]
L = [1, 2, 3, 4, 5, 6, 7, 9, 8] ;        take([1,2,3,4,5,6,7,8,9],9,R),perm(R,[8,7,A,5,4,B,2,1])
L = [1, 2, 3, 4, 5, 6, 8, 7, 9] ;    will not unify with first clause of take
...
```

gen_digits using perm (& perm using take)

```
% take(+L1,?X,?L2) succeeds if output list L2 is input list L1 less element X
take([H|T],H,T).                        % BASE CASE
take([H|T],X,[H|R]) :- take(T,X,R).     % RECURSIVE CASE

% perm(+L1,?L2) succeeds if output list L2 is a permutation of input list L1
perm([],[]).                                % BASE CASE
perm(L,[H|T]) :- take(L,H,R), perm(R,T).    % RECURSIVE CASE

% 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).

?- gen_digits([9,8,7,A,5,4,B,2,1]).
perm generates [9,8,7,6,5,4,3,2,1] i.e.
A = 6
B = 3 ;
then perm generates [9,8,7,3,5,4,6,2,1] i.e.
A = 3, B = 6
```

gen_digits using perm (& perm using take)

```
% take(+L1,?X,?L2) succeeds if output list L2 is input list L1 less element X
take([H|T],H,T).                        % BASE CASE
take([H|T],X,[H|R]) :- take(T,X,R).     % RECURSIVE CASE

% perm(+L1,?L2) succeeds if output list L2 is a permutation of input list L1
perm([],[]).                                % BASE CASE
perm(L,[H|T]) :- take(L,H,R), perm(R,T).    % RECURSIVE CASE

% 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).

?- gen_digits([9,8,7,A,5,4,B,2,1]).
A = 6, B = 3 ;
A = 3, B = 6 ;
false
```

SUDOKU 9x9

```
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],
```

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

```
           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 fixed/given numbers,
           Test the 'boxes',
           Test the 'columns',
           Test the 'rows',
           Print Solution.
```

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

```
puzzle1 :- A = [5,3,1,2,7,4,6,8,9],
           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),
           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 fixed/given numbers,
           Test the 'boxes',
           Test the 'columns',
           Test the 'rows',
           Print Solution.
```

```
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 fixed/given numbers,
Test the ‘boxes’,
Test the ‘columns’,
Test the 'rows',
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 FIXED/GIVEN numbers,
Test the ‘boxes’,
Test the ‘columns’,
Test the 'rows',
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’,
Test the 'rows',
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’,
Test the 'rows',
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',
Test the 'rows',
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',
Test the 'rows',
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.
```

Let's look at the 'test the columns'

Remember the Zebra puzzle?

We took the relation calls in the query and re-ordered them to make the generate & test nature of the solution method more explicit.

We'll do something similar in reverse to make our search process less naive.

```
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'
```

Note can test columns after row A...

```
[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]
```

This is already going to FAIL *if* we can write a **test_digits/1** that is happy to ignore free vars.

```
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'
```

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 or is [] and
%    all the digits in L1 are unique from input list L2
test_set([],_).                                          % BASE CASE
test_set([H|T], Digits) :- var(H),                       % VARS OK
                           test_set(T,Digits).
test_set([H|T], Digits) :- ground(H),                    % TAKE 1..9
                           take(Digits,H,RemainingDigits),
                           test_set(T,RemainingDigits).
```

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) :- perm([1,2,3,4,5,6,7,8,9],L).
```

Did you think of that definition?

Note that perm will deliver multiple results if there are variables in L.

The other thing we're going to need is a relation that gives us the vertical stripes through a set of equal length lists.

```
[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]
```

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

The other thing we're going to need is a relation that gives us the vertical stripes through a set of equal length lists.

```
[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]
```

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

How would you 'write a program' | 'declare some facts/rules' giving you heads(+LL,Heads,Tails) ?

How much 'code' is this going to be?

heads(+LL, ?LHeads, ?LTails)

```
[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]
```

```
% heads(+LL, ?LHeads, ?LTails) succeeds if:
%   LL is an input list of equal-length 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([[H1|T1]|T],[H1|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]]
```

Testing the columns

OK, so now we have :

test_digits(+L)
where L is any list of 9 digits & vars.

and

heads(+LL, ?Heads, ?Tails)
where LL is a list of equal-length lists
and Heads will be a list containing all the heads
and tails is a list of the remaining lists.

So unsurprisingly we are going to apply `test_digits` to the Heads.

test_cols(Rows)

```
% test_cols(Rows) succeeds if:
%    input Rows is a list of equal-length lists of digits and variables
%    each 'column' of digits and variables contain unique digits from [1..9]
test_cols([[]|_]).                          % BASE CASE
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]
```

We don't need a `test_col` (singular) because that is simply `test_digits`.

Note slight subtlety in base case - we end up with list of empty lists, not [].

```
[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]
```

Note here after `gen_digits(B)`, `test_cols` will FAIL and backtrack to another solution from `gen_digits(B)` until it succeeds.

```
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_cols([A,B,C,D,E,F,G,H,I]),
          gen_digits(B), B = [6,2,3,1,9,5,4,8,7]
          test_cols([A,B,C,D,E,F,G,H,I]),
          ...
          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]
```

Now we have 3 rows, we can pro-actively test the 'boxes'...

```
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_cols([A,B,C,D,E,F,G,H,I]),
          gen_digits(B), B = [6,2,3,1,9,5,4,8,7]
          test_cols([A,B,C,D,E,F,G,H,I]),
          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]
```

Note at this point we have 3 rows, so we have already generated 3 BOXES that can be tested.

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

We'll call test_boxes each time we have 3 rows, i.e. we'll test 3 boxes each time...

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]).
```

Unsurprisingly, we are going to apply `test_digits` to the boxes.

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]
     ).
```

So here's the complete program ...

SUDOKU 9x9

```
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]
                ).
```

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

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

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

Course Outline

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

Next time

Videos

- Cut
- Negation
- Databases

Prolog
Programming in Logic

Lecture #5

Ian Lewis, Andrew Rice

VIDEOS:

- CUT (17m)
  - more spider diagrams
  - Call .. Exit
  - Redo Fail
- NEGATION (11m)
- DATABASES (6m) (1 slide)

We'll review these, and do a little bit of a 'list relations' recap.

Q: I get a loop in my Prolog sometimes ....
... e.g. defining 'spouse'

Q: I get a loop in my Prolog sometimes ....
... e.g. defining 'spouse'

Inevitably because some clause will be:

```
spouse(A,B) :- spouse(B,A).
```

Q: I get a loop in my Prolog sometimes ....
... e.g. defining 'spouse'

Inevitably will be because some clause will be:
spouse(A,B) :- spouse(B,A).

E.g. (1) spouse(mickey, minnie).
(2) spouse(wilma, fred).
(3) spouse(A,B) :- spouse(B,A).

?- spouse(wilma, fred).
true; from clause (2), will now try clause (3) spouse(wilma, fred) :- spouse(fred, wilma)
true;
… repeats ad infinitum

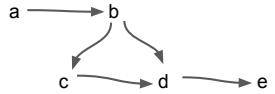
Q: I get a loop in my Prolog sometimes ....

This crops up more often with e.g. arcs defining graphs:

arc(a,b).
arc(b,c).
arc(b,d).
arc(c,d).
arc(d,e).

% maybe this for a cyclic graph:
arc(X,Y) :- arc(Y,X). ←**WRONG**

The acyclic graph will be ok:



path(A,B) :- arc(A,B).
path(A,C) :- arc(A,B), path(B,C).

linked(X,Y) :- arc(X,Y).
linked(X,Y) :- arc(Y,X).

?- linked(d,b).

Still need to be careful e.g. with path(X,Y) using linked(X,Y) if GRAPH is cyclic.

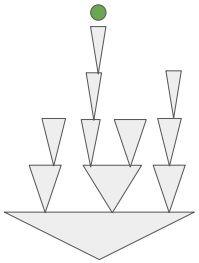
Q: All the methods taught so far (like generate and test) don't seem too efficient computationally. In the exam should we think of more complex logic to do so?

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!

As with Sudoku - tests are interleaved with generate.

```
[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_cols(..) with the 'permute' version (== gen_digits !) will actually **populate** the column with a viable set of numbers.

So gen_digits(B) will always have a fully populated row.

```
gen_digits(A),
test_cols([A,B,C,D,E,F,G,H,I]),
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 fixed/given numbers,
Test the 'boxes',
Test the 'columns',
Test the 'rows',
Print Solution.
```

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 which required 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)

We need to talk about CUT

or.. we need to talk about the execution model of Prolog.

From the Cut video, split/3

?

Question 1

What does split/3 do?

```
split([],[],[]).
split([H|T],[H|L],R) :- H < 5, split(T,L,R).
split([H|T],L,[H|R]) :- H >= 5, split(T,L,R).
```

From the Cut video, split/3

?

Question 1

What does split(+L1,?L2,?L3) do ?

(It's got arithmetic in it, so it's not going to work with L1 as a free variable)

USE CUT ????

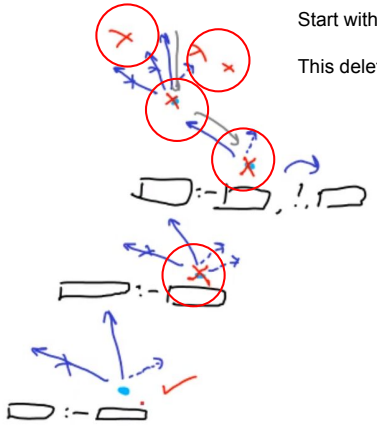
```
split([],[],[]).
split([H|T],[H|L],R) :- H < 5, split(T,L,R).
split([H|T],L,[H|R]) :- H >= 5, split(T,L,R).
```

Prolog if a Computer Lab graduate designed it:

```
split([],[],[]).
split([H|T],[H|L],R), H < 5 :- split(T,L,R).
split([H|T],L,[H|R]), H >= 5 :- split(T,L,R).
```

OR MOVE THE GUARD INTO THE HEAD?

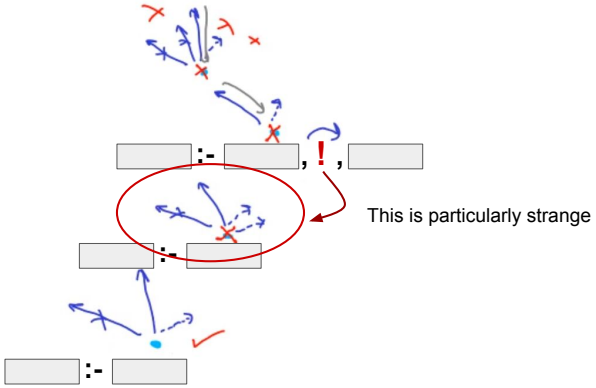
Explaining cuts .... messy Spider:



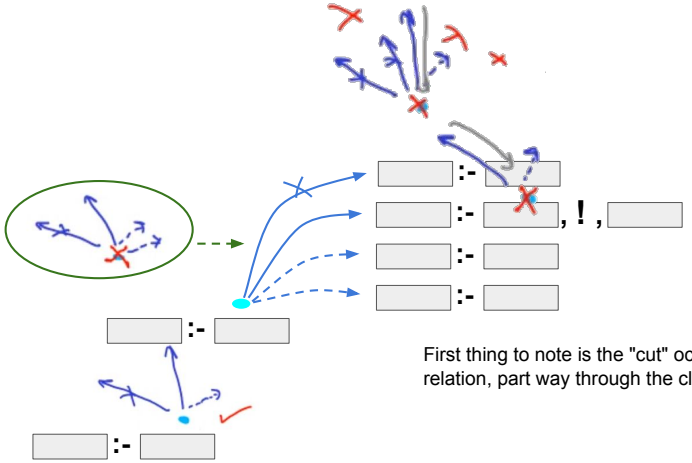
Start with a Spider Diagram ...

This deletion of choice points is hard to understand.

From the Cut video ... slightly tidier Spider



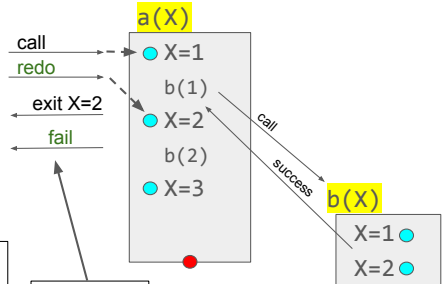
Example 1. From the Cut video ... Spider redraw clauses



Example 2. Procedural box model

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

b(1).
b(2).

?- a(X).
```

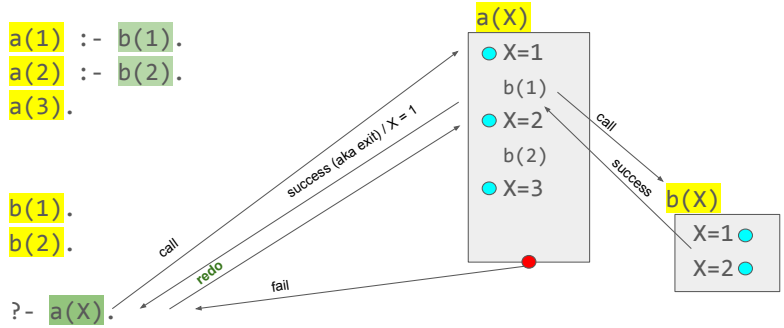
This is compiled into a procedure.



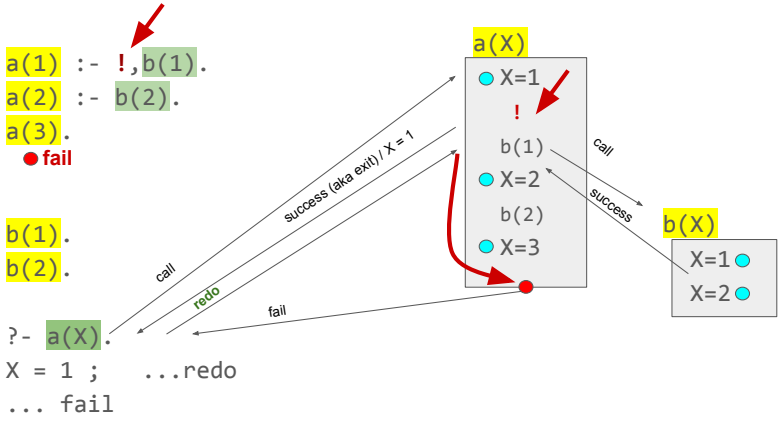
These are the Prolog extended call/return features.

Example 2. Procedural box model: without cut

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

b(1).
b(2).

?- a(X).
X = 1 ;    ...redo
X = 2 ;    ...redo
X = 3      ...redo ... fail
```



Example 2. Procedural box model: with cut (!)



At this point we have THREE interpretations of the Prolog behaviour:

1. The 'hand-drawn' sub-query/unification Spider diagram as in the video's.
2. The depth-first left-to-right tree representing the stack diagram.
3. The 'procedural box' model representing the actual compiled implementation.

The notable difference between 1. and 3. is the latter explains the implementation of cut(!) *within* the compiled relation/procedure containing the cut(!).

Example 3. A procedural interpretation of cut(!)

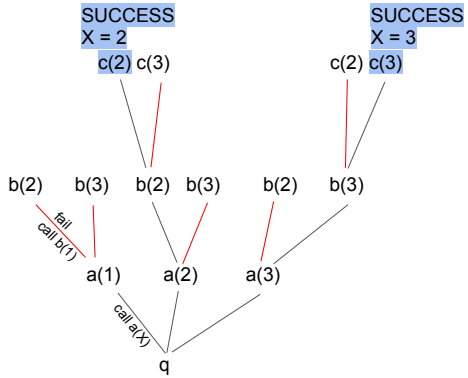
```
a(1).
a(2).
a(3).

b(2).
b(3).

c(2).
c(3).

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

:- q.
```

We'll look at these new simple facts, with a simple query.

This is the cut we'll look into.

Example 3. A procedural interpretation of cut(!) - without the cut(!)

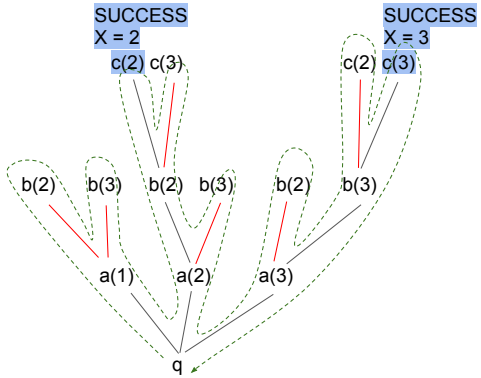
```
a(1).
a(2).
a(3).

b(2).
b(3).

c(2).
c(3).

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

:- q.
```



This is our 'depth-first left-to-right' view (red lines are fails).

Example 3. 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.
```

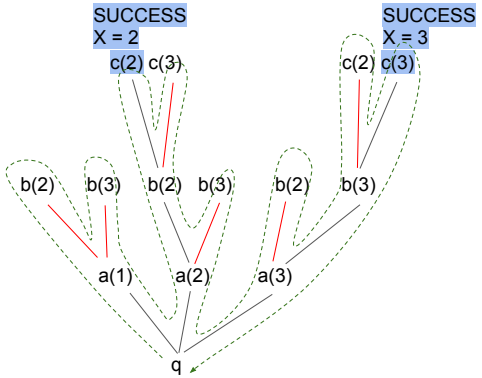


DEPTH-FIRST LEFT-TO-RIGHT SEARCH

Example 4. A procedural interpretation of cut(!) - without the cut(!)

Note you get the same search tree extending a/1 and removing b/1 from the query

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

b(2).
b(3).

c(2).
c(3).

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

:- q.
```

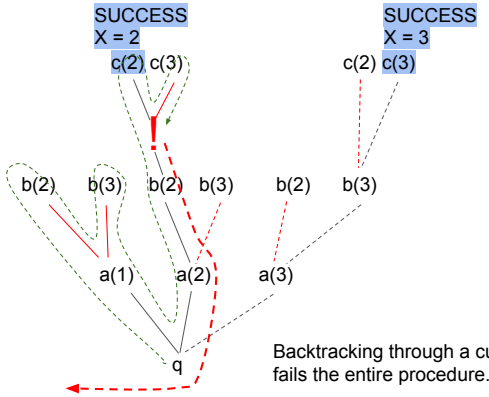


DEPTH-FIRST LEFT-TO-RIGHT SEARCH

Example 5. Adding the cut(!) into the q relation.

Note the cut(!) is now in the search tree and harder to interpret.

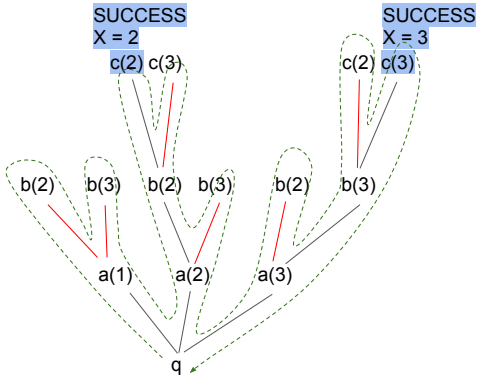
```
a(1).
a(2).
a(3).

b(2).
b(3).

c(2).
c(3).

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

:- q.
```



Backtracking through a cut fails the entire procedure.

IMPORTANT DIFFERENCE WITH FUNCTIONAL PROGRAMMING: The actual execution sequence is to carry forwards the results and pause with each solution. Note a functional language will POP the stack on returning a result, Prolog does not.

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

b(2).
b(3).

c(2).
c(3).

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

:- q.
```



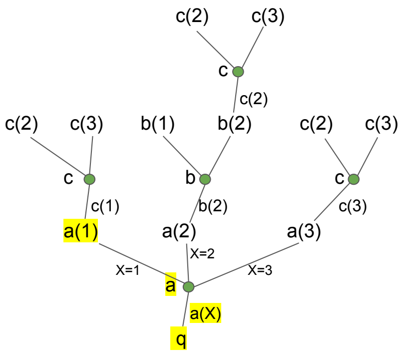
Example 6. Without cut(!).

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

b(1).
b(2).

c(2).
c(3).

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



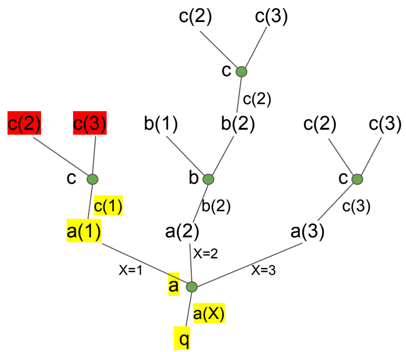
Example 6. Without cut(!).

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

b(1).
b(2).

c(2).
c(3).

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



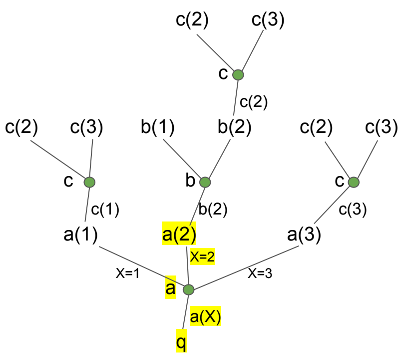
Example 6. Without cut(!).

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

b(1).
b(2).

c(2).
c(3).

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



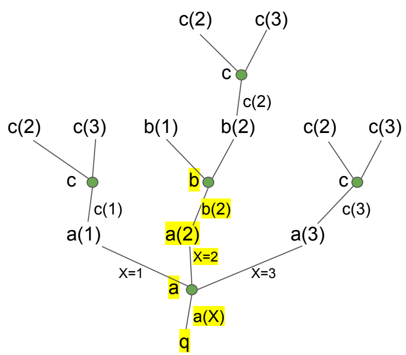
Example 6. Without cut(!).

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

b(1).
b(2).

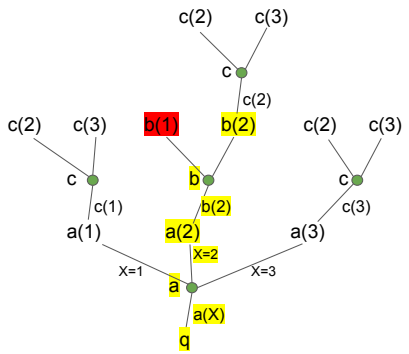
c(2).
c(3).

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



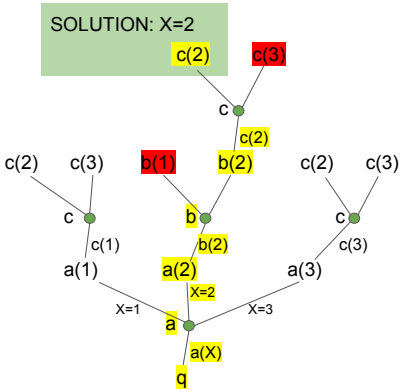
Example 6. Without cut(!).

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

b(1).
b(2).

c(2).
c(3).

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



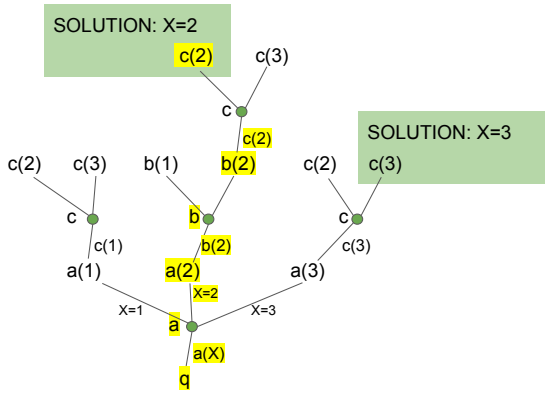
Example 6. Without cut(!).

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

b(1).
b(2).

c(2).
c(3).

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



Example 6. Without cut(!).

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

b(1).
b(2).

c(2).
c(3).

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



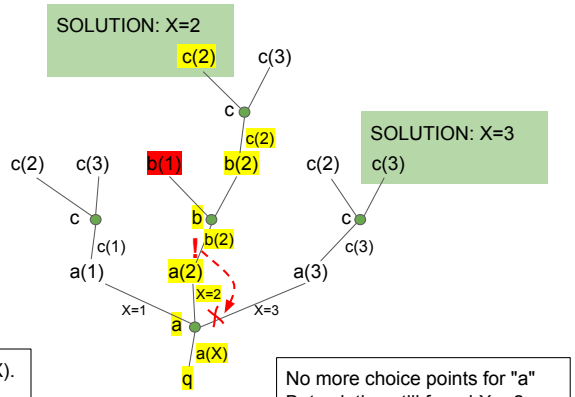
Example 6.1. With cut(!).

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

b(1).
b(2).

c(2).
c(3).

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

```
?- a(X),c(X).
X = 2.

?-
```

Example 6.1. With cut(!) in a(2).

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

b(1).
b(2).

c(2).
c(3).

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

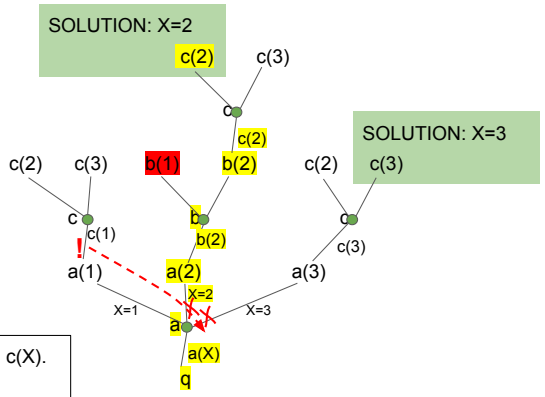



```
?- a(X),c(X).
X = 2.

?-
```

No more choice points for "a"
But solution still found X = 2

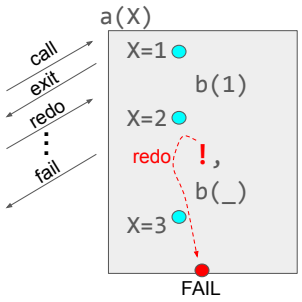
Example 6.2. With cut(!) in q.

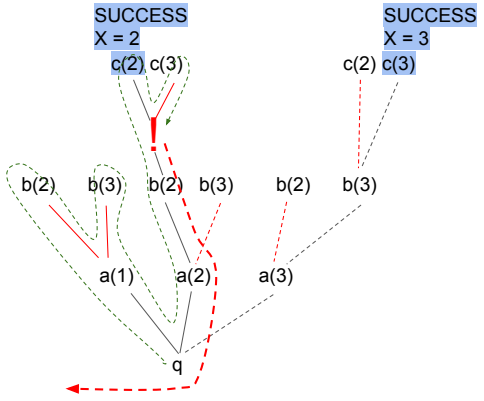
```
a(1).
a(2) :- b(2).
a(3).

b(1).
b(2).

c(2).
c(3).

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




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

?-
```

Cut





Cut is much easier to understand in the procedural box model. The extra-logical side effect of cut(!) is to **prune** the search tree - that is harder to understand but it's why cut is called cut.

Cut(!) toxicology

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

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

Cut(!) toxicology

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

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

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

?-
```

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 ;

?-
```

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).
```

Searching the tree...

Whether you find the solution depends where you start the search from !!!!!!

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

?-
```

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

?-
```

```
?- a(3).
true

?-
```

negation: "not" aka "\+"

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

This is an extra-logical RELATION e.g. "not(A)", as is "call(A)".
Negation by failure, i.e. `\+(A)` succeeds if `call(A)` fails. *(Higher order logic)*
In general this is potentially dangerous, as illustrated in the video.

Given that foo(X) must have failed for us to continue with the query, any unification with X will have been undone.

\+ can use 'one solution' semantics.

A legitimate, and common, use case is deterministic **notmember**(+X,+L).
e.g. rather than `foo(X), \+ member(X,[1,2,3])`. However, if possible make your life simpler by declaring a **mode** for the arguments of such a relation e.g. notmember(+X,+L).

When these requirements crop up, the first option to consider is the goal term A \= B
which is A will not unify with B. unification is a deterministic algorithm, so this is deterministic.

negation / not, other operators

\+ A : negation, i.e. A will fail. A is a RELATION (HOL)!!
Some equalities:
A = B : A will unify with B
A =:= B : A1 is A, B1 is B, A1 = B1. (numerical)
Some inequalities:
A \= B : A will NOT unify with B (succeed/fail)
A =\= B : A1 is A, B1 is B, A1 \= B1. (numerical)
A \== B : not IDENTICAL, i.e. A \== B succeeds. (???)

Summary: negation / not, other operators

Note that the deterministic infix "A = B" aka *unify*(A,B) also has a negated counterpart, "A \= B". This is possibly the most legitimate of the negation extra-logical operators and is useful for example where you might want to know something is *not* a member of a list.

But even A \= B has some "undefined" behavior e.g. with f(1,X) \= f(1,a(X)). i.e. \= has similar issues with occurs check.

relational database

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

```
                name   floor
location(andy,      2).
location(alastair, 2).
location(ian,       1).
```

location

name	*floor*
andy	2
alastair	2
ian	1

```
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 input 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.
% will produce an infinite number of solutions if L1 is a var.
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 - 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)
```

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

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

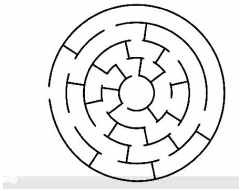
Ian Lewis, Andrew Rice

Today's discussion

Videos

Countdown

Graph search

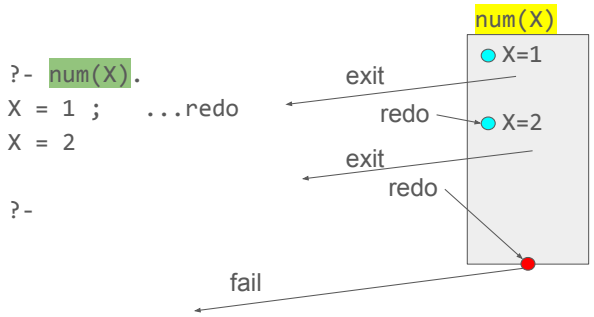


Q: Why does Prolog respond differently?

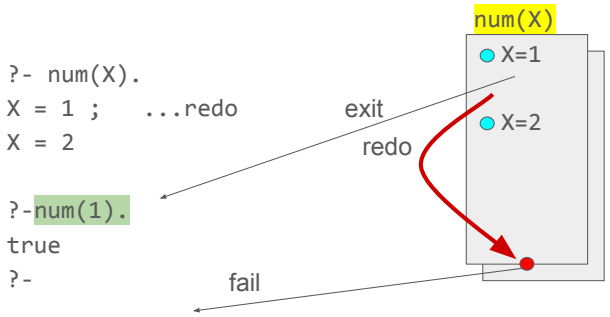
Considering orthogonal clause heads (list12.pl):

```
num(1).
num(2).

list01([]).
list01([_]).

list12([_]).
list12([_,_]).

listN([1]).
listN([2]).
```

Procedural box model



Procedural box model



Q: Why supervisor say wrong???

```
max(X,Y,X) :- X > Y; !.
max(X,Y,Y).
```

Q: Why supervisor say wrong???

```
max(X,Y,X) :- X > Y; !.
max(X,Y,Y).

% max(+X,+Y,?Z) succeeds if Z is bigger of numbers X and Y.
max(X,Y,X) :- X > Y.
max(X,Y,Y) :- X =< Y.
```

Q: Why supervisor say wrong???

```
max(X,Y,X) :- X > Y; !.
max(X,Y,Y).
```

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

```
% max(+X,+Y,?Z) succeeds if Z is bigger of numbers X and Y.
max(X,Y,X) :- X > Y.
max(X,Y,Y) :- X =< Y.
```

For the record, this is "wrong" as well.

```
len(L,N) :- L = [], !, N = 0 ; L = [H|T], N is 1 + len(T).
```

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?

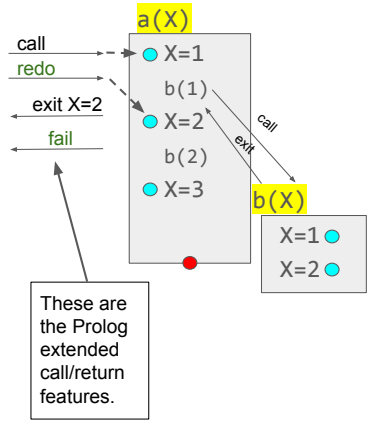
```
% foo(+X,?Y) succeeds if output number Y
% is double input number X
foo(X,Y) :- Y is 2 * X.
```

Small detail, previous lecture

(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.
```



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!

1. Pure Prolog subset 'Datalog' used for network verification.
2. Prolog used for Java Virtual Machine verification
3. Prolog quite good at digital logic simulation
4. In building real-time privacy protection frameworks
5. Prolog search has interesting parallelism

Actually you can embed Prolog in Python, so you can think of Prolog as an 'engine' rather than a language.
... theorem provers written in Prolog.

$$\sin(-\theta)=-\sin\theta$$
$$\cos(-\theta)=\cos\theta$$
$$\sin(\theta+\phi)=\sin\theta\cos\phi+\cos\theta\sin\phi$$
$$\cos(\theta-\phi)=\cos\theta\cos\phi+\sin\theta\sin\phi$$

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

Countdown

... may use only the four basic operations of addition, subtraction, multiplication and division,[45] and do not have to use all six numbers. A number may not be used more times than it appears on the board. Division can only be performed if the result has no remainder (i.e., the divisor is a factor of the dividend). Fractions are not allowed, and only positive integers may be obtained as a result at any stage of the calculation.

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)

Note we could be reducing these arithmetic pairs as we go along, but then we wouldn't be keeping the formula (i.e. path) for the solution.

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

```
% countdown(+L, +Target, ?Soln) succeeds if the Soln evaluates to the Target and is
% derivable with simple arithmetic operations using the numbers provided in L.
countdown([Soln|_],Target, Soln) :-  eval(Soln,Target).

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

will step through...

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],Rem),
     arithop(A,B,Exp),
     countdown([Exp|Rem],Target,Soln).
```

```
?- countdown([25,50,75,100,3,6], 952, Soln).
   ... 2nd: choose(2,[25,50,75,100,3,6],[A,B],Rem)  ... A=25, B = 50, Rem = [75,100,3,6]
   ...      arithop(25, 50, Exp)                    ... Exp = 25 + 50
   ...      countdown([25+50,75,100,3,6], 952, Soln)
            ... 1st clause ... eval(25+50, 952)     ... Fail
            ... 2nd clause ... choose(2,[25+50,75,100,3,6],[A,B], Rem) ... A=25+50, B=75, Rem = [100,3,6]
                               arithop(25+50,75,Exp)                   ... Exp = (25+50)+75
                               countdown([(25+50)+75,100,3,6], 952, Soln).
                                    ... first clause ...
                                    ...
```

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],Rem),
     arithop(A,B,Exp),
     countdown([Exp|Rem],Target,Soln).
```

```
?- countdown([25,50,75,100,3,6], 952, Soln).
   ... 2nd: choose(2,[25,50,75,100,3,6],[A,B],Rem)  ... A=25, B = 50, Rem = [75,100,3,6]
   ...      arithop(25, 50, Exp)                    ... Exp = 25 + 50
   ...      countdown([25+50,75,100,3,6], 952, Soln)
            ... 1st clause ... eval(25+50, 952)     ... Fail
            ... 2nd clause ... choose(2,[25+50,75,100,3,6],[A,B], Rem) ... A=25+50, B=75, Rem = [100,3,6]
                               arithop(25+50,75,Exp)                   ... Exp = (25+50)+75
                               countdown([(25+50)+75,100,3,6], 952, Soln).
                                    ... first clause ...
                                    ...
```

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],Rem),
     arithop(A,B,Exp),
     countdown([Exp|Rem],Target,Soln).

?- countdown([25,50,75,100,3,6], 952, Soln).
   ... 2nd: choose(2,[25,50,75,100,3,6],[A,B],Rem)  ... A=25, B = 50, Rem = [75,100,3,6]
   ...      arithop(25, 50, Exp)                    ... Exp = 25 + 50
   ...      countdown([25+50,75,100,3,6], 952, Soln)
            ... 1st clause ... eval(25+50, 952)      ... Fail
            ... 2nd clause ... choose(2,[25+50,75,100,3,6],[A,B], Rem) ... A=25+50, B=75, Rem = [100,3,6]
                               arithop(25+50,75,Exp)                ... Exp = (25+50)+75
                               countdown([(25+50)+75,100,3,6], 952, Soln).
                                    ... first clause ...
                                    ...
```

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],Rem),
     arithop(A,B,Exp),
     countdown([Exp|Rem],Target,Soln).

?- countdown([25,50,75,100,3,6], 952, Soln).
   ... 2nd: choose(2,[25,50,75,100,3,6],[A,B],Rem)  ... A=25, B = 50, Rem = [75,100,3,6]
   ...      arithop(25, 50, Exp)                    ... Exp = 25 + 50
   ...      countdown([25+50,75,100,3,6], 952, Soln)
            ... 1st clause ... eval(25+50, 952)      ... Fail
            ... 2nd clause ... choose(2,[25+50,75,100,3,6],[A,B], Rem) ... A=25+50, B=75, Rem = [100,3,6]
                               arithop(25+50,75,Exp)                ... Exp = (25+50)+75
                               countdown([(25+50)+75,100,3,6], 952, Soln).
                                    ... first clause ...
                                    ...
```

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],Rem),
     arithop(A,B,Exp),
     countdown([Exp|Rem],Target,Soln).

?- countdown([25,50,75,100,3,6], 952, Soln).
   ... 2nd: choose(2,[25,50,75,100,3,6],[A,B],Rem)  ... A=25, B = 50, Rem = [75,100,3,6]
   ...      arithop(25, 50, Exp)                    ... Exp = 25 + 50
   ...      countdown([25+50,75,100,3,6], 952, Soln)
            ... 1st clause ... eval(25+50, 952)      ... Fail
            ... 2nd clause ... choose(2,[25+50,75,100,3,6],[A,B], Rem) ... A=25+50, B=75, Rem = [100,3,6]
                               arithop(25+50,75,Exp)                ... Exp = (25+50)+75
                               countdown([(25+50)+75,100,3,6], 952, Soln).
                                    ... first clause ...
                                    ...
```

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],Rem),
     arithop(A,B,Exp),
     countdown([Exp|Rem],Target,Soln).

?- countdown([25,50,75,100,3,6], 952, Soln).
   ... 2nd: choose(2,[25,50,75,100,3,6],[A,B],Rem)  ... A=25, B = 50, Rem = [75,100,3,6]
   ...      arithop(25, 50, Exp)                    ... Exp = 25 + 50
   ...      countdown([25+50,75,100,3,6], 952, Soln)
            ... 1st clause ... eval(25+50, 952)      ... Fail
            ... 2nd clause ... choose(2,[25+50,75,100,3,6],[A,B], Rem) ... A=25+50, B=75, Rem = [100,3,6]
                               arithop(25+50,75,Exp)                ... Exp = (25+50)+75
                               countdown([(25+50)+75,100,3,6], 952, Soln).
                                    ... first clause ...
                                    ...
```

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],Rem),
     arithop(A,B,Exp),
     countdown([Exp|Rem],Target,Soln).
```

```
?- countdown([25,50,75,100,3,6], 952, Soln).
   ... 2nd: choose(2,[25,50,75,100,3,6],[A,B],Rem)  ... A=25, B = 50, Rem = [75,100,3,6]
   ...      arithop(25, 50, Exp)                    ... Exp = 25 + 50
   ...      countdown([25+50,75,100,3,6], 952, Soln)
            ... 1st clause ... eval(25+50, 952)     ... Fail
            ... 2nd clause ... choose(2,[25+50,75,100,3,6],[A,B], Rem) ... A=25+50, B=75, Rem = [100,3,6]
                               arithop(25+50,75,Exp)              ... Exp = (25+50)+75
                               countdown([(25+50)+75,100,3,6], 952, Soln).
                                    ... first clause ...
                                    ...
```

arithOp(+A,+B,?Exp).

Non-deterministically choosing an arithmetic expression given a pair of arguments.

```
arithop(A,B,A+B).
arithop(A,B,A-B) :- eval(A,A1), eval(B,B1), A1 > B1.
arithop(B,A,A-B) :- eval(A,A1), eval(B,B1), A1 > B1.
arithop(A,B,A*B).
arithop(A,B,A/B) :- eval(B,B1), B1 > 0, eval(A,A1), 0 is A1 rem B1.
arithop(B,A,A/B) :- eval(B,B1), B1 > 0, eval(A,A1), 0 is A1 rem B1.
```

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.

```
arithop(A,B,plus(A,B)).
arithop(A,B,minus(A,B)) :- eval(A,D), eval(B,E), D>E.
arithop(B,A,minus(A,B)) :- eval(A,D), eval(B,E), D>E.
arithop(A,B,mult(A,B)) :- eval(A,D), D \== 1, eval(B,E), E \== 1.
arithop(A,B,div(A,B)) :- eval(B,E), E \== 1, E \== 0, eval(A,D), 0 is D rem E.
arithop(B,A,div(A,B)) :- eval(B,E), E \== 1, E \== 0, eval(A,D), 0 is D rem E.
```

choose(+N,+L,?Chosen,?Remaining)

Non-deterministically choosing multiple elements from a list with choose(+N, +List, ?Chosen, ?Remaining) seems similar to 'take' of one element.

```
% take(+List,?Element,?Remaining)
take([H|T],H,T).
take([H|T],X,[H|L]) :- take(T,X,L).

% choose(+N, +List, ?Chosen, ?Remaining)
choose(0,L,[],L).
choose(N,L,[H|T],Remaining) :- N > 0,
    take(L,H,Rem1),
    M is N - 1,
    choose(M,Rem1,T,Remaining).
```

choose

Here's another version, this time combining the 'take' into the 'choose' relation:

```
% choose(+N, +List, ?Chosen, ?Remaining)
choose(0,L,[],L).

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

choose(N,[H|T],Chosen,[H|Remaining]) :- N > 0,
     choose(N,T,Chosen,Remaining).
```

will step through...

choose

```
% choose(+N, +List, ?Chosen, ?Remaining)
choose(0,L,[],L).                              % BASE CASE

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

choose(N,[H|T],Chosen,[H|Remaining]) :- N > 0,
     choose(N,T,Chosen,Remaining).
```

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

choose

```
% choose(+N, +List, ?Chosen, ?Remaining)
choose(0,L,[],L).                              % BASE CASE

choose(N,[H|T],[H|Chosen], Remaining) :- N > 0,     % CHOOSE HEAD
     M is N-1,
     choose(M,T,Chosen,Remaining).

choose(N,[H|T],Chosen,[H|Remaining]) :- N > 0,
     choose(N,T,Chosen,Remaining).
```

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).                              % BASE CASE

choose(N,[H|T],[H|C], Remaining) :- N > 0,     % CHOOSE HEAD
     M is N-1,
     choose(M,T,C,Remaining).

choose(N,[H|T],Chosen,[H|Remaining]) :- N > 0,        % SKIP HEAD
     choose(N,T,Chosen,Remaining).
```

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(L, max(L), Remaining) , ...

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

F L A T T E N I N G

E.g. also:

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

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

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

F L A T T E N I N G

E.g. also:

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

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

Another look at choose ?

```
max_acc([],Acc,Acc).
max_acc([H|T],Acc,Max) :- H > Acc, max_acc(T,H,Max).
max_acc([H|T],Acc,Max) :- H =< Acc, max_acc(T,Acc,Max).

max(L,Max) :- max_acc(L,0,Max).
```

```
max([A],A).
max([H|T],H) :- max(T,TMax), H > TMax.
max([H|T],TMax) :- max(T,TMax), H =< M.
```

choose

% choose(N, List, Chosen, Remaining) - earlier version

```
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|Remaining]) :- N > 0, choose(N,T,Chosen,Remaining).
```

For our purposes choose/4 could be choose2/3...

choose2 is basically take,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).
```

```
% choose2(+L,?A,?B,?Rem)
choose2(L,A,B,R2)  :-
     take(A,L,R1),
     take(B,R1,R2).
```

Note:
choose/4 passes through the list ONCE, picking a first and second number (expression) as it goes.

choose2 generates any 1st number followed by any 2nd number. E.g. take2 will generate 1,2 & 2,1.

Yet Another Alternative Version of choose

% choose(N, List, Chosen, Remaining) - earlier version

```
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|Remaining]) :- N > 0, choose(N,T,Chosen,Remaining).
```

choose is still 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]

Choices

```
% choose(+N, +List, ?Chosen, ?Remaining)
choose(0,L,[],L).
choose(N,L,[H|T],Remaining) :- N > 0,
    take(L,H,Rem1),
    M is N - 1,
    choose(M,Rem1,T,Remaining).
```

```
% choose2(+L,?Chosen,?Rem)
choose2(L,[A,B],R2)  :-
      take(A,L,R1),
      take(B,R1,R2).
```

```
% choose(+N, +List, ?Chosen, ?Remaining)
choose(0,L,[],L).

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

choose(N,[H|T],Chosen,[H|Remaining]) :- N > 0,
     choose(N,T,Chosen,Remaining).
```

```
% take_list(+Choices,+L2,?RemL)
take_list([], L, L).
take_list([H|T],L,R) :-
      take(H,L,LR),
      take_list(T, LR, R).
```

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.

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).
```

The Final Countdown - complete solution

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

choose2(L,[A,B],Rem2) :- take(A,L,Rem1), take(B, Rem1, Rem2).

arithop(A,B,A+B).
arithop(A,B,A-B) :- A1 is A, B1 is B, A1 > B1.
arithop(A,B,A*B).
arithop(A,B,A/B) :- B1 is B, B1 > 0, A1 is A, 0 is A1 rem B1.

countdown([Soln|_],Target, Soln) :- Target is Soln.
countdown(L,Target,Soln) :- choose2(L,[A,B],R), arithop(A,B,C), countdown([C|R],Target,Soln).

?- countdown([25,50,75,100,3,6],952,Soln).
Soln =  ((100+6)*(75*3)-50)/25
...
```

This is using 'choose2' to provide both perms of 2 numbers, so 'arithop' is simpler.

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.

```
countdown([Soln|_],Target, Soln) :-  Target is Soln.
countdown(L,Target,Soln) :-  take2(L,[A,B],R), arithop(A,B,C), 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.

```
countdown([Soln|_],Target, Soln) :-  Target is Soln.
countdown(L,Target,Soln) :-  take2(L,[A,B],R), arithop(A,B,C), countdown([C|R],Target,Soln).

countdown([Soln|_],Target, Soln, Delta) :-  Error is abs(Target - Soln), Error =< Delta.
countdown(L,Target,Soln,Delta) :-  take2(L,[A,B],R), arithop(A,B,C), countdown([C|R],Target,Soln,Delta).

?- countdown([25,50,75,100,3,6],952,Soln,1).
Soln =  (3*6+25*50)*75/100
```

951

Countdown Iterative Deepening

Note: reduction occurs here.

```
countdown([Soln|_],Target, Soln, Delta) :-  Error is abs(Target - Soln), Error =< Delta.
countdown(L,Target,Soln,Delta) :-  take2(L,[A,B],R), arithop(A,B,C), countdown([C|R],Target,Soln,Delta).

?- countdown([25,50,75,100,3,6],952,Soln,1).
Soln =  (3*6+25*50)*75/100
```

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

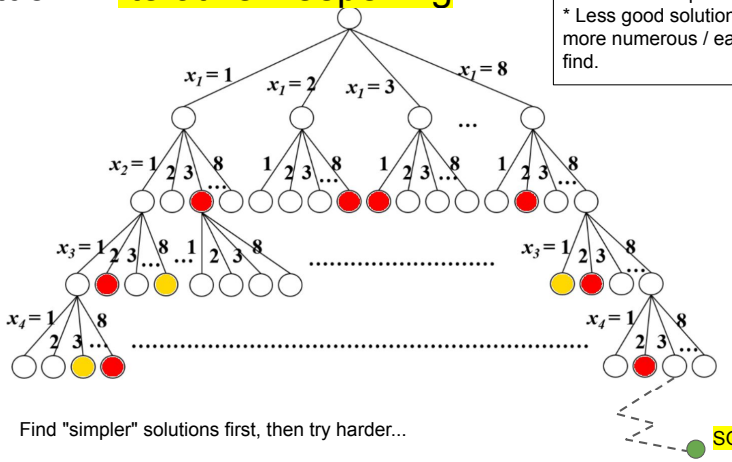
Diff =< Threshold: the approach is slightly different here than in the video (both are valid) - we are asking for solutions *within* a 'distance' from the exact answer (not *at* an exact distance).

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).

Countdown Iterative Deepening

2 concepts:
* Avoid infinite paths
* Less good solutions are more numerous / easier to find.

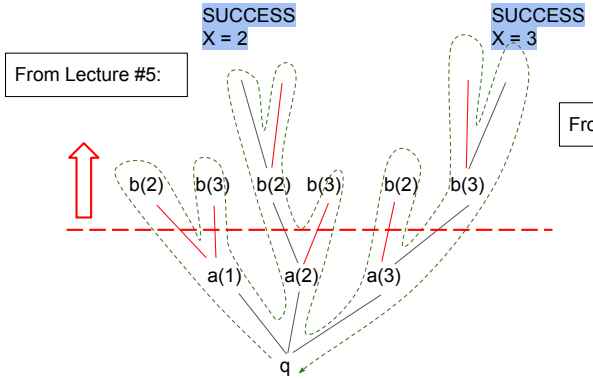


Find "simpler" solutions first, then try harder...

Iterative Deepening

2 concepts:
* Avoid infinite paths
* Less good solutions are more numerous / easier to find.

From Lecture #5:

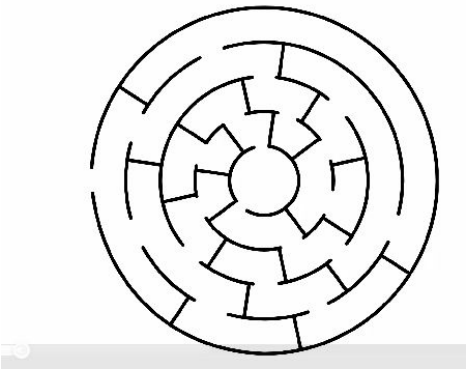


From Lecture #1:

```
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]
```

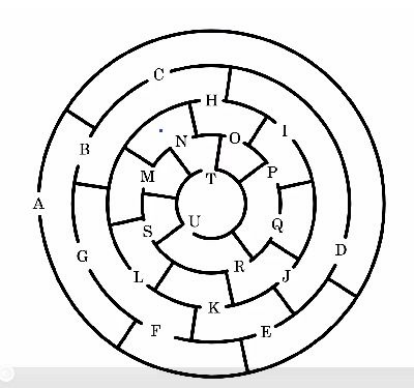
Find "simpler" solutions first, then try harder...

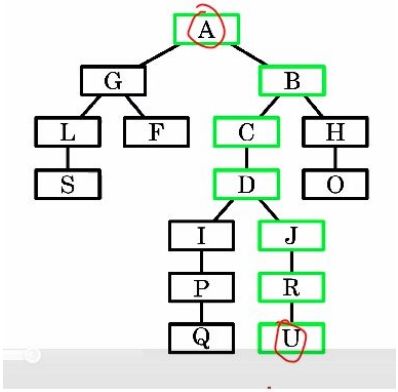
Graph Search



Problem statement

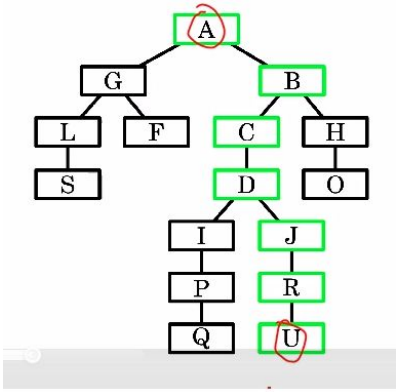
Graph Search





Convert to graph...

Graph Search

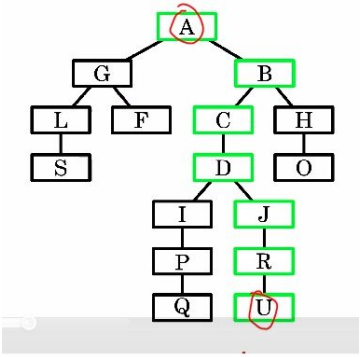



```
route(a,g).                start(a).
route(g,l).                finish(u).
route(l,s).
...
travel(A,A).
travel(A,C) :- route(A,B),travel(B,C).

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

TBH solve(A,B) would be more mainstream.

Sample implementation (simple, given graph)

Graph Search



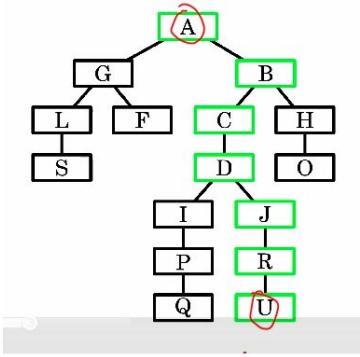

```
route(a,g).                start(a).
route(g,l).                finish(u).
route(l,s).
...
travel(A,A).
travel(A,C) :- route(A,B),travel(B,C).

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

Note alternative data representations, both with relations:

```
:- 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).
```

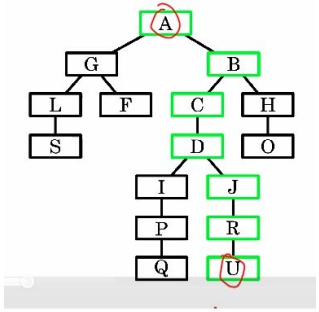
Graph Search



Alternative data representation node(name, left, right) using a single compound term.

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

Graph Search



(1) Base case:

(2) Recursive case:

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).
```

Accumulating here

Copying to solution here (via reverse/2)

Next time

Videos

- Difference
- Empty difference lists
- Difference list example

Prolog
Programming in Logic

Lecture #7

Ian Lewis, Andrew Rice

Today's discussion

Videos

- Difference
  - append(A-B,B-C,A-C).
- Empty difference lists
  - A-A vs. []-[]
- Difference list example
  - rotate(L1,L2).

Q: how to implement mid(+L,+X)?

[1,2,3,4,5]

```
% mid(+L,+X) succeeds if input X is the center element of
% input list L
mid(L,X) :- mid_acc(L,X,0).

% mid_acc(+L,+X,+N) succeeds if input X would be the center
% element of input list L if L were prepended by an additional
% N elements.
mid_acc([H|L],H,N) :- len(L,N).                          % BASE CASE - SUCCESS
mid_acc([_|L],H,N) :- len(L,M), M > N, N1 is N + 1, mid_acc(L,H,N1). % RECURSIVE
```

Q: Can our countdown relation find single-number solutions?

E.g. Countdown [25, 50, 75, 100, 3, 6], Target 3 ?

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

countdown(L,Target,Soln) :-
     choose(2,L,[A,B],Rem),
     arithop(A,B,Exp),
     countdown([Exp|Rem],Target,Soln).
```

Q: Can our countdown relation find single-number solutions?

E.g. Countdown [25, 50, 75, 100, 3, 6], Target 3 ?

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

countdown(L,Target,Soln) :-
     choose(2,L,[A,B],Rem),
     arithop(A,B,Exp),
     countdown([Exp|Rem],Target,Soln).
```

So for this to succeed, either Soln needs to evaluate to the Target, or Exp needs to evaluate to the Target.

Q: Can our countdown relation find single-number solutions?

E.g. Countdown [25, 50, 75, 100, 3, 6], Target 3 ?

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

countdown(L,Target,Soln) :-
     choose(2,L,[A,B],Rem),
     arithop(A,B,Exp),
     countdown([Exp|Rem],Target,Soln).
```

BASE case: Countdown [25, 50, 75, 100, 3, 6], Target 25 is OK

Q: Can our countdown relation find single-number solutions?

E.g. Countdown [25, 50, 75, 100, 3, 6], Target 3 ?

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

countdown(L,Target,Soln) :-
     choose(2,L,[A,B],Rem),
     arithop(A,B,Exp),
     countdown([Exp|Rem],Target,Soln).
```

BASE case: Countdown [25, 50, 75, 100, 3, 6], Target 25 is OK

RECURSIVE case: Countdown [25+50, 75, 100, 3, 6], eval HEAD

RECURSIVE case: Countdown [(25+50)+75, 100, 3, 6], eval HEAD

Q: Can our countdown relation find single-number solutions?

E.g. Countdown [25, 50, 75, 100, 3, 6], Target 3 ?

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

countdown(L,Target,Soln) :-
     choose(2,L,[A,B],Rem),
     arithop(A,B,Exp),
     countdown([Exp|Rem],Target,Soln).
```

The point is countdown([3|...],3,Soln) is never called, and a single-number solution is never suggested (apart from 25)

Q: Can our countdown relation find single-number solutions?

E.g. Countdown [25, 50, 75, 100, 3, 6], Target 3 ?

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

countdown(L,Target,Soln) :-
     choose(2,L,[A,B],Rem),
     arithop(A,B,Exp),
     countdown([Exp|Rem],Target,Soln).
```

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

countdown(L,Target,Soln) :-
     choose(2,L,[A,B],Rem),
     arithop(A,B,Exp),
     append(Rem,[Exp],ExprList),
     countdown(ExprList,Target,Soln).
```

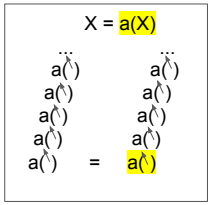
A 'fix' would be to build the Expression list in reverse, i.e. [75, 100, 3, 6, 25+50]

Quick aside on `unify_with_occurs_check(?X,?Y)`

Unify in Prolog:
X = Y, ...
That's how important it is.

unify_with_occurs_check/2 is an example of something retrospectively added to Prolog, which is important, but support for infinite terms simply doesn't exist within Prolog except in some special cases. Functional programming often has weak support for infinite terms, but similarly you can program for them e.g. using functions to represent the terms remaining in a sequence.

The humorous part is the programmer of the unify_with_occurs_check/2 relation presumably looked up the possible symbols he could use as an operator for the new relation (== ?, =:= ?, ...) and discovered they had already been used by previous programmers for lesser functions.



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

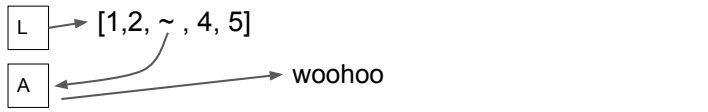
A gentle introduction to difference lists.

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

As you know, Prolog has no problem creating/unifying terms with Vars inside.

A gentle introduction to difference lists.

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

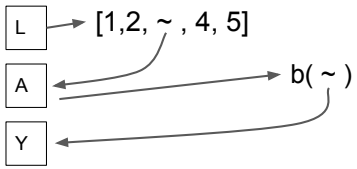


Prolog does NOT proactively copy "woohoo" into the list.

A gentle introduction to difference lists.

```
?- L = [1,2,A,4,5], A = b(Y).
L = [1,2,b(Y),4,5],
A = b(Y).
```

Changing atom "woohoo" to compound term b(Y)

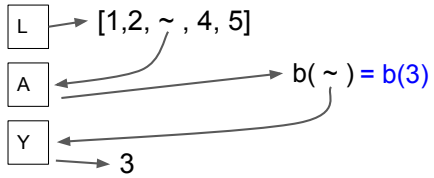


Similarly, Prolog does not copy 'b(Y)' into the list L.

A gentle introduction to difference lists.

```
?- L = [1,2,A,4,5], A = b(Y), A = b(3).
L = [1,2,b(3),4,5],
A = b(3).
Y = 3.
```

Adding an additional A = b(3) term to the query.



Prolog will do this additional unification

and Y will be instantiated to '3'.

Think of the 'procedural box' model, all this is going on inside the procedural box.

A gentle introduction to difference lists.

```
?- L = [1,2,A,4,5], A = woohoo.
L = [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 as we discuss this.

Before we look at 'append', let's look at 'rotate'.

```
It's actually pretty simple.
To 'rotate' A:
?- A = [1,2,3|B],

A = [H|T],
    H = 1          % useful
    T = [2,3|B]    % useful

B = [H|C].
    T = [2,3,1|C]
```

```
?- A = [1,2,3|B],

A = [_ _ _,Z],
    imaginary syntax where
    "_ _ _" unifies with any
    sequence of list elements.
```

Rotate with difference lists.

```
Making the 'B' available, we
can pick off the 'H' and unify
it into the end of the tail in
a single unification step :

?- A = [1,2,3|B] - B,

A = [H|T] - [H|C].

Rotated list is T - C, where
T = [2,3,1|C].
```

Rotate with difference lists.

```
Making the 'B' available, we
can pick off the 'H' and unify
it into the end of the tail in
a single unification step :

?- A = [1,2,3|B] - B,

A = [H|T] - [H|C].

Rotated list is T - C, where
T = [2,3,1|C].
```

```
Hence :

rotate([H|T]-[H|C], T-C).

?- L1=[1,2,3|B]-B,rotate(L1,L2).
L1 = [1, 2, 3, 1|C]-[1|C],
B = [1|C],
L2 = [2, 3, 1|C]-C
```

More on difference lists.

```
?- L1 = [1,2,3|A].
L1 = [1,2,3|A].       --- A list with a hole in it...

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

A gentle introduction to difference lists.

```
?- L1 = [1,2,3|A].
L1 = [1,2,3|A].       --- A list with a hole in it...

The tail of a list is always a list. So what about:
?- L1 = [1,2,3|7].
L1 = [1,2,3|7].

? L1 = [1,2,3|7], L1 = [A,B,C].
false

? L1 = [1,2,3|7], L1 = [A,B,C,D].
false
```

[1,2,3|7] IS NOT A PROPERLY FORMED LIST, so don't expect unification with Prolog lists to work.

A gentle introduction to difference lists.

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

```
?- L1 = [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.

```
?- L1 = [1,2,3|L2], L2=[].
L1 = [1,2,3],
L2 = [].                % we are simply terminating the list.
```

A gentle introduction to difference lists.

The most relevant example:

```
?- L1 = [1,2,3|L2], L2 = [4,5,6,7].
L1 = [1,2,3,4,5,6,7],
```

That feels a bit like append.

A gentle introduction to difference lists.

With two lists:

```
?- L1 = [1,2,3|A], L2 = [4,5,6|B].
L1 = [1,2,3|A],
L2 = [4,5,6|B].
```

A gentle introduction to difference lists.

With two lists:

```
?- L1 = [1,2,3|A], L2 = [4,5,6|B].
L1 = [1,2,3|A],
L2 = [4,5,6|B].
```

Linking the lists...

```
?- L1 = [1,2,3|A], L2 = [4,5,6|B], A = L2.
L1 = [1,2,3,4,5,6|B], % so we have managed to append, via unification
A = L2,
L2 = [4,5,6|B].
```

A gentle introduction to difference lists.

This is great! How to write an append ?

```
?- L1 = [1,2,3|A], L2 = [4,5,6|B], append(L1,L2,L3).

append([],L2,L2).        % BASE CASE
append(?,?,?) :- ...     % ??
```

A gentle introduction to difference lists.

This is great! How to write an append ?

```
?- L1 = [1,2,3|A], L2 = [4,5,6|B], append(L1,L2,L3).

append([],L2,L2).
append(?,?,?) :- ...
```

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

```
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 Z list with the X list
                 ZVar = YVar. -- make the var of Z as for Y
?- append([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].
```

IRL don't redefine built-in 'append' ...

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 Z list with the X list
                 ZVar = YVar. -- make the var of Z as for Y
?- append([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

?- append([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

?- append([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

?- append([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.

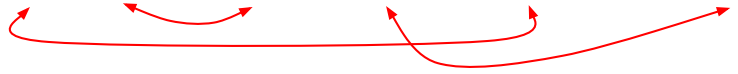
```
?- X = [1,2,3|XVar]-XVar, Y = [4,5,6|YVar]-YVar, append(X,Y,Z).



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

?- append([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,XL-YVar).

?- append([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,XL-YVar).
i.e. rename to:
append(A-B,B-C,A-C). % SO YOU DON'T NEED THIS RELATION
                     % JUST DO THE UNIFICATIONS EARLIER IN THE CODE.
If you have e.g. [1,2,3|B] and [4,5,6|C], just unify the B with
[4,5,6|C] and keep the C as the list end var.

So [1,2,3|B]-B and [4,5,6|C]-C goes to [1,2,3,4,5,6|C]-C
```



A gentle introduction to difference lists.

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

So that lengthy build-up leads to this simple and easily-memorable result, which can typically be combined with the fact you can do the unifications in the calling code, without needing a difference list 'append' relation.

A gentle introduction to difference lists.

```
% Final empty diff list list thoughts.

?- L1 = [a,b,c|A]-A, A = [], L1 = MyList-_ .
MyList = [a,b,c].

? L1 = A-A, A=[], L1 = MyList-_.
MyList = [].

* Empty diff list is ALWAYS A-A.
* To TEST for it you attempt a unification with something that only matches
<freevar>-<freevar>, typically []-[].
* To TERMINATE a diff list, unify with X-[] and use the X.

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

We'll do this using regular Prolog lists, then difference lists.

Implement Quicksort without/with difference lists.

```
% 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

% append(+L1,+L2,?L3) succeeds if L3 is elements of L1 then L2.
```

This is going to be the same algorithm, with these same declarative definitions for the relations, whether we use regular lists or difference lists.

Implement `partition(+Pivot,+List,?Left,?Right)`

```
% 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(_,[],[],[]).                  % BASE CASE
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).

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

Implement `quicksort(+L1,?L2)`

```
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([],[]).                   % BASE CASE
quicksort([P|T],Sorted) :-          % RECURSE. Head = Partition
    partition(P,T,L,R),             % Split into lower/higher lists
    quicksort(L,L1),                % Sort the lower list into L1
    quicksort(R,R1),                % Sort the higher list into R1
    append(L1,[P|R1],Sorted).       % Append L1,R1, re-inserting P
```

Is it useful to turn this into difference lists?

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

Is it useful to turn this into difference lists?

```
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([],[]).
quicksort([P|T],Sorted) :-
    partition(P,T,L,R),
    quicksort(L,L1),
    quicksort(R,R1),
    append(L1,[P|R1],Sorted).    % HERE IS WHERE WE BENEFIT
```

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).
```

So it is these lists in the append that we want to be difference lists.

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).
```

Taking the vars in the append, and highlighting them throughout the rule, what do you notice?

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).
```

The input list remains a 'normal' list.

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).
```

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).
```

```
quicksort([],A-A).

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

Step 2: Worry about empty difference lists

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

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

Step 2: Worry about empty difference lists

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

Should this be []-[] or A-A ?
A-A because we are GENERATING 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

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

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

Replace L2 with [P|R1]

By "irrelevant" we mean unnecessary, i.e. we'll have done the required unifications already.

Step 3: Substitutions to make the append irrelevant

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

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

Replace L2 with [P|R1]

Step 3: Substitutions to make the append irrelevant

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

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

Replace L1 with Sorted

Step 3: Substitutions to make the append irrelevant

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

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

Replace L1 with Sorted

Step 3: Substitutions to make the append irrelevant

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

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

Replace R2 with S

Step 3: Substitutions to make the append irrelevant

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

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

Replace R2 with S

Step 4: Remove the append because that unification has been propagated back into the code.

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

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

Can use '[]' in the query to convert Sorted to a regular Prolog list.

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

Next time

Videos

- Sudoku (if you really want to, or watch video for Lecture #4)
- Constraints (don't worry about the details, just the concept)

```
% 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]), []).
```

Prolog
Programming in Logic

Lecture #8

Ian Lewis, Andrew Rice

Today's discussion

Videos - optional for additional background / understanding:

- Sudoku
- Constraints

This lecture is for checking your understanding of key topics, we'll review the Towers of Hanoi problem, and look a little bit more at the 'search' behaviour of Prolog.

Great question re: 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).
```

```
SQL:      SELECT name, floor FROM age,location
          WHERE age.name=location.name AND age > 40.
PROLOG: ?- age(Age_Name,Age), location(Location_Name,Floor),
          Age_Name = Location_name, Age > 40.

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

What about "SELECT count(*) FROM age;" ?

```
age(andy,     35).
age(alastair, 45).
age(ian,      65).
age(jon,      60).

What about "SELECT count(*) from age." ?
```

```
?- count_proc(age,N).
```

relation 'age/2' as an argument

```
count_age(N) :- age(_,_), add_1_to_count_then_fail.
count_age(N) :- N = count_so_far.
```

'count' as global variable

What about "SELECT count(*) FROM age;" ?

```
age(andy,     35).
age(alastair, 45).
age(ian,      65).
age(jon,      60).

What about "SELECT count(*) from age." ?

Using an accumulator for the count:
```

```
1. count_age(N) :- count_age_acc(0,N).

2. count_age_acc(Count,N) :-
3.     age(_,_),
4.     Count1 is Count + 1,
5.     count_age_acc(Count1,N).

6. count_age_acc(Count,Count).
```

What about "SELECT count(*) FROM age;" ?

```
age(andy,     35).
age(alastair, 45).
age(ian,      65).
age(jon,      60).

What about "SELECT count(*) from age." ?

count_age/2 with an accumulator, this time that
accumulator is a list of prior states.
```

```
count_age(CountedList,N) :-
     age(X,Y),
     notmember([X,Y],CountedList),
     count_age([[X,Y]|CountedList],N).

count_age(CountedList,N) :- length(CountedList,
N).

?- count_age([],N).
```

Alternative: data table as a compound term:

```
age(andy,     35).
age(alastair, 45).
age(ian,      65).
age(jon,      60).

What about "SELECT count(*) from age." ?
```

```
count_age(CountedList,N) :-
     age(X,Y),
     notmember([X,Y],CountedList),
     count_age([[X,Y]|CountedList],N).

count_age(CountedList,N) :- length(CountedList, N).

?- count_age([],N).
```

```
table_age([
     (andy,     35),
     (alastair, 45),
     (ian,      65),
     (jon,      60)
]).

?- table_age(T), length(T,N).
```

Instead of a fact-per-row, we have a fact-per-table.

Count(*) query becomes simple.

A typical "select" with join with tables as compound terms.

```
table_age([
     (andy,     35),
     (alastair, 45),
     (ian,      65),
     (jon,      60)
]).
```

```
table_location([
     (andy,     second),
     (alastair, ground),
     (ian,      second),
     (jon,      first)
]).
```

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

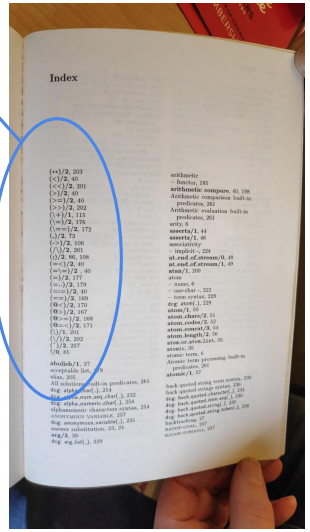
```
?- table_age(AgeTable), table_location(LocTable),
member((Name,Age),AgeTable), member((Name,Location),LocTable), Age > 40.
```

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

A data representation choice: compound terms vs. facts.

Q: My supervisor says?:

what are these:



Cambridge equalities/inequalities:

=	:	Unify
is	:	Arithmetic evaluation
>		
>=		
<	:	Numerical compare
=<		
[\=	:	Will not unify]*

*if expressly suggested in exam.

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

Q: When is Prolog not ML-like?

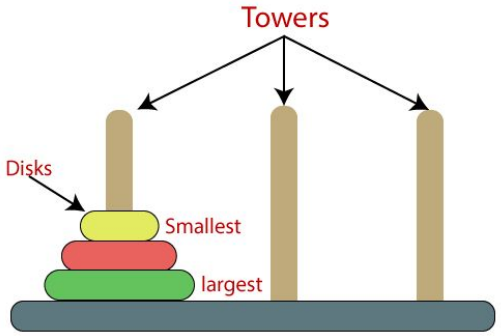
Ultimately you can write a unification function in any language, define a data structure representing relations, and create a backtracking algorithm.

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

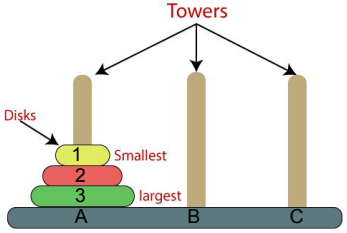
Or you can modify Prolog to support higher-order functions...

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

Towers of Hanoi



Towers of Hanoi - data representation

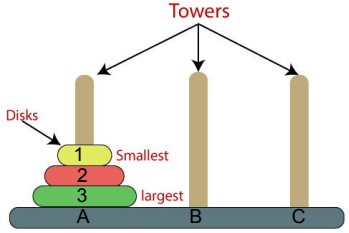


State:[A, B, C] where each tower A,B,C is a list of 0..3 numbers

```
State:  [ [1,2,3], [],   []  ]
```

I.e. State is a fixed-length list with an element per tower, where that element is a list of disk numbers.
It is equally possible to solve this puzzle with State being the position of each disk, e.g. in this case State could be [a,a,a] indicating all the disks are on the A tower (and assuming the program can never place the disks in the wrong order).

Towers of Hanoi - data representation



State:[A, B, C] where each tower A,B,C is a list of 0..3 numbers

```
State:  [ [1,2,3], [],   []  ]
Target: [ [],      [],[1,2,3]]
```

```
TOWER STATE: List of 0..3 numbers,
PUZZLE STATE: List of 3 tower states
```

Towers of Hanoi

We'll ACCUMULATE a list of 'State's as we search for the solution, and check that list each time we add a new state, so we avoid cycles in the search. Clearly there is a finite number of possible states so the program is guaranteed to terminate.

States:
[[[1,2,3],[],[]],[[2,3],[1],[]],[[2,3],[],[1]],[[3],[2],[1]],[[3],[1,2],[]],[[],[1,2],[3]],[[1],[2],[3]],[[1],[],[2,3]],[[],[],[1,2,3]]]

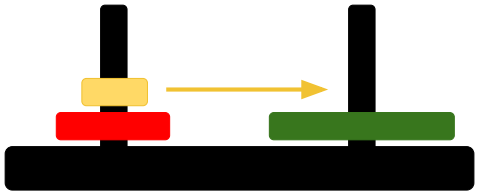
Housekeeping: some relations with side effects (printing...):

```
% printing the full list of states.
print_states([]).
print_states([H|T]) :- print_state(H), print_states(T).

% printing one state of all three towers.
print_state([A,B,C]) :-
     print_tower(A), print_tower(B), print_tower(C), nl.

% printing one tower with fixed width for formatting.
print_tower([]) :-      write('     '), write([]).
print_tower([A]) :-     write('    '), write([A]).
print_tower([A,B]) :-   write('  '), write([A,B]).
print_tower([A,B,C]) :- write([A,B,C]).
```

```
[1,2,3]     []     []
  [2,3]    [1]     []
  [2,3]     []    [1]
    [3]    [2]    [1]
    [3]  [1,2]     []
     []  [1,2]    [3]
    [1]    [2]    [3]
    [1]     []  [2,3]
     []     [][1,2,3]
```

Towers of Hanoi - TEST ok_move from T1 to T2

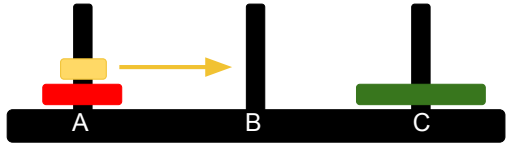
Can we take top disk from T1 & move it to T2 ?



T1 = [1,2] T2 = [3]
NT1 = [2] NT2 =[1,3]

```
% ok_move(+T1, +T2, ?NT1, ?NT2)
% (Deterministic) succeeds if T1,T2,NT1 and NT2 are TOWER STATES (i.e.
% lists of up to 3 numbers) and NT1, NT2 represents the
% state of T1,T2 after moving the top disk from T1 to T2 within rules of hanoi.
ok_move([A|As],[],    As,[A]).               % MOVE TOP OF T1 ONTO EMPTY T2
ok_move([A|As],[B|Bs],As,[A,B|Bs]) :- A < B.  % MOVE TOP OF T1 ONTO OCCUPIED T2
```

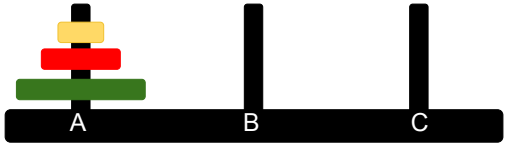
Towers of Hanoi - GENERATE move State1 to State2



State1: [[1,2], [], [3]]
State2: [[2], [1], [3]]

```
% move(+State1, ?State2)
% succeeds if State2 is the state for all 3 towers after a valid
% move from State1.
move([A,B,C], [AN,BN,C]) :- ok_move(A,B,AN,BN).   % MOVE A -> B
move([A,B,C], [AN,B,CN]) :- ok_move(A,C,AN,CN).   % MOVE A -> C
move([A,B,C], [AN,BN,C]) :- ok_move(B,A,BN,AN).   % MOVE B -> A
move([A,B,C], [A,BN,CN]) :- ok_move(B,C,BN,CN).   % MOVE B -> C
move([A,B,C], [AN,B,CN]) :- ok_move(C,A,CN,AN).   % MOVE C -> A
move([A,B,C], [A,BN,CN]) :- ok_move(C,B,CN,BN).   % MOVE C -> B
```

Towers of Hanoi - solve

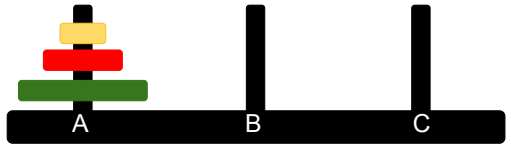


From: [[1,2,3], [], []]
To: [[], [], [1,2,3]]

```
solve :-
    From = [[1,2,3],[],[]],
    To = [[],[],[1,2,3]],
    hanoi([From], From, To).
```

Now we require `hanoi( +Visited_States_list, +From, +To).`

Towers of Hanoi - hanoi




```
From: [[1,2,3],        [],        []]
To:   [     [],        [],     [1,2,3]]
```

```
% Hanoi(+States, +From, +To) will succeed and print the full state list if
% an acceptable sequence of disk moves is found from state From to state To, without
% traversing states already in +States.
hanoi(States, From, To) :-                          % BASE CASE, DIRECT MOVE AVAILABLE
        move(From,To),                              % FIND DIRECT MOVE
        reverse([To|States], R), print_states(R).   % (housekeeping) PRINT PATH OF STATES

hanoi(States, From, To) :-                          % RECURSIVE CASE
        move(From, Next),                           % MAKE ANY VALID MOVE
        notmember(Next, States),                    % CHECK ACCUMULATOR OF PRIOR STATES
        hanoi([Next|States],Next,To).               % CHOOSE NEXT MOVE
```

Towers of Hanoi - demo run

```
Welcome to SWI-Prolog (threaded, 64 bits, version 8.2.4)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?- [hanoi].
true.

?- solve.
[1,2,3]     []     []
  [2,3]    [1]     []
    [3]    [1]    [2]
  [1,3]     []    [2]
    [3]     []  [1,2]
     []    [3]  [1,2]
    [1]    [3]    [2]
     []  [1,3]    [2]
    [2]  [1,3]     []
  [1,2]    [3]     []
  [1,2]     []    [3]
    [2]    [1]    [3]
     []    [1]  [2,3]
     []     [][1,2,3]
true
```

Accumulating the Path to a Solution

Accumulating the Path to a Solution

```
color(red).
color(blue).

pattern(check).
pattern(plain).

style(X) :- color(X).
style(X) :- pattern(X).
style(cool) :- color(blue), pattern(plain).

?- style(X).
X = red ;
X = blue ;
X = check ;
X = plain ;
X = cool.
```

Accumulating the Path to a Solution

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. for a FACT:

```
color(red).
```

becomes:

```
color(red, Path, [1|Path]).
```

Accumulating the Path to a Solution

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. for a FACT:

```
color(red).
```

becomes:

```
color(red, Path, [1|Path]).
```

Step 3. Similar for a RULE

```
style(X) :- color(X).
```

becomes:

```
style(X,Path,PathOut) :- color(X,[1|Path], PathOut).
```

Accumulating the Path to a Solution

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. for a FACT:

```
color(red).
```

becomes:

```
color(red, Path, [1|Path]).
```

Step 3. Similar for a RULE

```
style(X) :- color(X).
```

becomes:

```
style(X,Path,PathOut) :- color(X,[1|Path], PathOut).
```

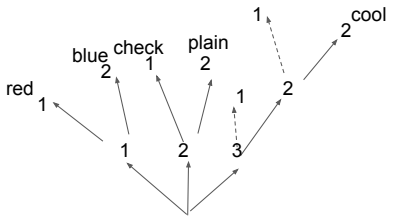
When we initially call 'style' we will set 'PathIn' to [], and expect the completed path in PathOut:

```
?- style(X,[],Path).
```

Accumulates the path 'backwards'

Accumulating the Path to a Solution - mechanical changes.

```
color(red, Path, [1|Path]).
color(blue, Path, [2|Path]).

pattern(check, Path, [1|Path]).
pattern(plain, Path, [2|Path]).

style(X,Path,PathOut) :- color(X,[1|Path], PathOut).
style(X,Path,PathOut) :- pattern(X,[2|Path], PathOut).
style(cool,Path,PathOut) :- color(blue,[3|Path],PathOut1), pattern(plain,PathOut1,PathOut).

?- style(X,[],Path).
X = red,
Path = [1, 1] ;
X = blue,
Path = [2, 1] ;
X = check,
Path = [1, 2] ;
X = plain,
Path = [2, 2] ;
X = cool,
Path = [2, 2, 3].
```



hanoi puzzle: Accumulating the Path

```
% move(+State1, ?State2)
% succeeds if State2 is a the state for all 3 towers after a valid
% move from State1.
move([A,B,C], [AN,BN,C], PathIn, [1|PathIn]) :- ok_move(A,B,AN,BN).   % MOVE A -> B
move([A,B,C], [AN,B,CN], PathIn, [2|PathIn]) :- ok_move(A,C,AN,CN).   % MOVE A -> C
move([A,B,C], [AN,BN,C], PathIn, [3|PathIn]) :- ok_move(B,A,BN,AN).   % MOVE B -> A
move([A,B,C], [A,BN,CN], PathIn, [4|PathIn]) :- ok_move(B,C,BN,CN).   % MOVE B -> C
move([A,B,C], [AN,B,CN], PathIn, [5|PathIn]) :- ok_move(C,A,CN,AN).   % MOVE C -> A
move([A,B,C], [A,BN,CN], PathIn, [6|PathIn]) :- ok_move(C,B,CN,BN).   % MOVE C -> B

% Hanoi(+States, +From, +To) will succeed and print the full state list if
% an acceptable sequence of disk moves is found from state From to state To.
hanoi(States, From, To, PathIn, PathOut) :-              % BASE CASE, DIRECT MOVE AVAILABLE
        move(From,To, [1|PathIn], PathOut),              % FIND DIRECT MOVE
        reverse([To|States], R), print_states(R).        % PRINT STATES INCLUDING FINAL

hanoi(States, From, To, PathIn, PathOut) :-              % RECURSIVE CASE
        move(From, Next, [2|PathIn], PathOut1),          % MAKE ANY VALID MOVE
        notmember(Next, States),                         % CHECK ACCUMULATOR OF PRIOR STATES
        hanoi([Next|States],Next,To, PathOut1, PathOut). % CHOOSE NEXT MOVE
```

Accumulating the Path to a Solution

```
Welcome to SWI-Prolog (threaded, 64 bits, version 8.2.4)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?- [hanoi_path].
true.

?- solve.
[1,2,3]     []     []
  [2,3]    [1]     []
    [3]    [1]    [2]
  [1,3]     []    [2]
    [3]     []  [1,2]
     []    [3]  [1,2]
    [1]    [3]    [2]
     []  [1,3]    [2]
    [2]  [1,3]     []
  [1,2]    [3]     []
  [1,2]     []    [3]
    [2]    [1]    [3]
     []    [1]  [2,3]
     []     [][1,2,3]
[2,1,2,2,2,3,2,2,2,1,2,5,2,1,2,5,2,3,2,4,2,1,2,2,1,4]:26
true
```

Why study Prolog?

[2,1,2,2,2,3,2,2,2,1,2,5,2,1,2,5,2,3,2,4,2,1,2,2,1,4]

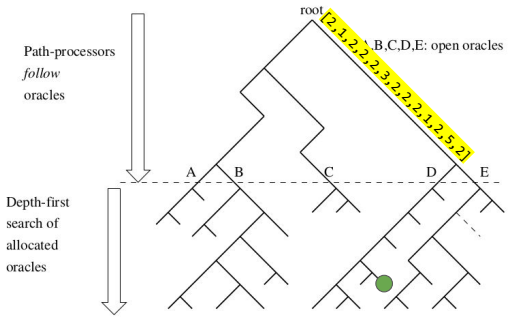


Figure 2.11: Second phase of *breadth-first partitioning*.

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

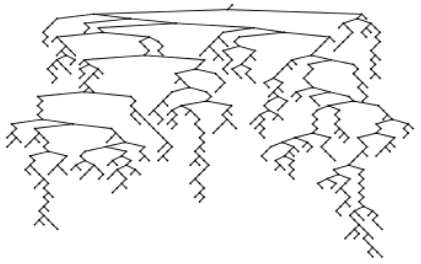
Actually, given that a Prolog program is itself a valid Prolog term, you can apply *simple* transformations to that program to manipulate the tree. E.g.

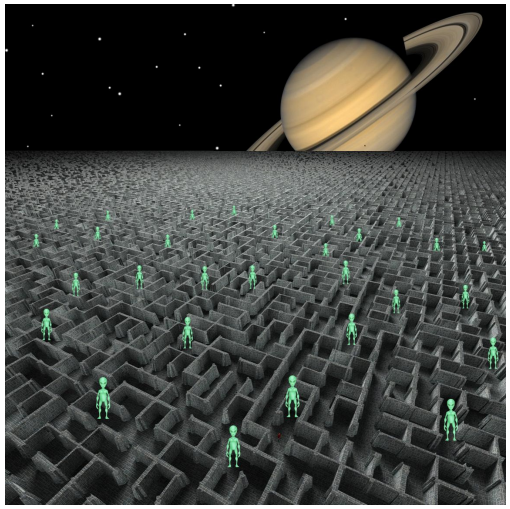
```
last([X], X).
last([_|T],X) :- last(T,X).
```

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?





Prolog: Programming in Logic

- Variables
- Atoms
- Compound Terms
- Facts
- Rules
- Unification
- Queries

- Execution Strategy
- Data Representation
- Declarative methodology
  - base case
  - recursive case(s)
- Usage mode:
  - generate (non-deterministic)
  - test (deterministic)

Lecture 5: List relations + len/2, rev/2 with accumulators

Simple:

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

Accumulator:

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

End... of... the... Prolog, Programming in Logic course...

A lot of content in the past 2.5 weeks so well done - please drop a token in the box outside!