Prolog Lecture 1

David Eyers
Michaelmas 2008

Notes derived from those created
by Andy Rice and Kate Taylor
used with permission

Course aims

Introduce a declarative style of programming

- Explain fundamental elements of Prolog: terms, clauses, lists, arithmetic, cuts, backtracking, negation

Demonstrate Prolog problem-solving techniques

By the end of the course you should be able to:

- Write and understand Prolog programs
- Use difference structures
- Understand basic constraint programming principles

Assessment

One exam question in Paper 3

Assessed Exercise (a tick)

- you must get a tick for either Prolog or C & C++
- tick exercises and submission done in Lent term
- more information to follow closer to the time

Supervision work

Some example questions are provided at the end of the lecture handout

- Note: Prolog examples are often easy to follow ...
- Make sure you can write your own programs too!

I will give some pointers and outline solutions during the lectures

- Make sure you understand the fundamentals well

Recommended text

"PROLOG Programming for Artificial Intelligence", Ivan Bratko, Addison Wesley (3rd edition, 2000)

- Provides an alternative angle on the basics
- Examines problem solving with Prolog in detail
- Not all of the textbook is directly relevant

Lecture 1

Logic programming and declarative programs

Introduction to Prolog

Basic operation of the Prolog interpreter

Prolog syntax: Terms

Unification of terms

Solving a logic puzzle

Imperative programming

Formulate a "how to compute it" recipe, e.g.:

- to compute the sum of the list, iterate through the list adding each value to an accumulator variable

```
int sum(int[] list ) {
   int result = 0;
   for(int i=0; i<list.length; ++i) {
     result += list[i];
   }
   return result;
}
```

Functional programming

Again formulate a "how to compute it" recipe

- Probably will need to do recursive decomposition

```
(* The sum of the empty list is zero and
 the sum of the list with head h and tail
 t is h plus the sum of the tail. *)

fun sum([])    = 0
  | sum(h::t) = h + sum(t);
```

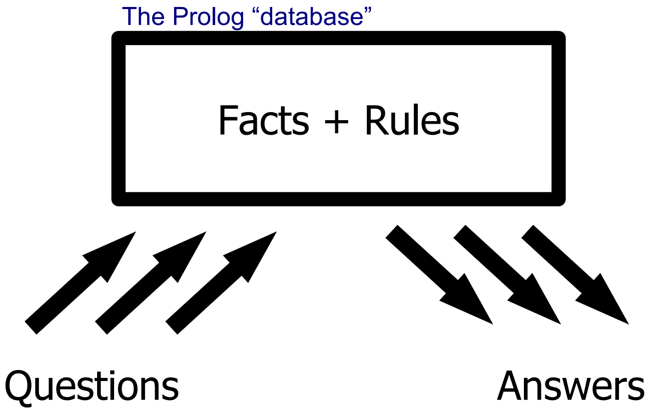
Logic programming

```
% the sum of the empty list is zero
sum([],0).

% the sum of the list with head H and
% tail T is N if the sum of the list T
% is M and N is M + H
sum([H|T],N) :- sum(T,M), N is M+H.
```

This is a declarative reading of a program (p23)

- Not "how to compute" the result
- Instead "this is true about the result"

Prolog programs answer questions



The Prolog "database"

Facts + Rules

Questions

Answers

Prolog came from the field of Natural Language Processing

PROgramming en LOGique

Colmerauer, A., Kanoui, H., Roussel, P . and Pasero, R. "Un systeme de communication homme-machine en français", Groupe de Recherche en Intelligence Artificielle, Université d'Aix-Marseille, 1973.

Modern Prolog interpreters use the Warren Abstract Machine

WAM is like a logic programming virtual machine

- David H. D. Warren. "An abstract Prolog instruction set." Technical Note 309, SRI International, Menlo Park, CA, October 1983.

Can significantly improve Prolog memory use

- Modern Prolog implementations will is the WAM
- ... or something similar to it

You are expected to use SWI-Prolog

Open-source (GPL) Prolog environment

- http://www.swi-prolog.org/
- Development began in 1987
- Available for Linux, MacOS X and Windows
- Fully featured, with many libraries

We will use SWI-Prolog throughout this course

- Get yourself a copy! (or at least access to one)
- Experiment with it!

SWI-Prolog is available to you

SWI-Prolog is installed on the PWF (5.6.38)

- Reboot a PWF terminal into Linux
- Log into linux.pwf.cam.ac.uk

You can easily install it on your own computer

- Linux users: use your package manager
  - (e.g. pl on Fedora Core, swi-prolog on Ubuntu)
- Otherwise consult the SWI-Prolog download page
  - Windows and MacOS binary installers are available
  - Build it from source

Prolog can answer simple questions directly from its database

```
> prolog                         ..................or maybe pl on your system
?- [user].                       ..........get ready to enter a new program
|: milestone(rousell,1972).
|: milestone(warren,1983).
|: milestone(swiprolog,1987).                              type [CTRL]-D
|: milestone(yourcourse,2008).                   ...............when done
|: % user://1 compiled 0.01 sec, 764 bytes
Yes
?- milestone(warren,1983).                ................ask it a question
Yes                                       ...........the answer is “yes”
?- milestone(swiprolog,X).                ..........let it find an answer
X=1987                                    ............the answer is 1987
Yes
?- milestone(yourcourse,2007).                   ........ask it a question
No                                        ...................the answer is “no”
?- halt.                                  ..................exit the interpreter
```

We will usually load Prolog programs from source files on disk

```
> cat milestone.pl              .........enter your program in a text file
milestone(rousell,1972).                 (it should have a .pl extension)
milestone(warren,1983).
milestone(swiprolog,1987).
milestone(yourcourse,2008).
> prolog
?- [milestone].           ...........instruct Prolog to load the program
?- milestone(warren,1983).
Yes
?- milestone(X,Y).                          ...........................find answers
X = rousell
Y = 1972 ;             ......you type a semi-colon (;) for more answers

X= warren
Y = 1983               .........you press enter when you've had enough
Yes
?- halt.
```

Our program is composed of facts and queries

```
> cat milestone.pl
milestone(rousell,1972).
milestone(warren,1983).
milestone(swiprolog,1987).
milestone(yourcourse,2008).
> prolog
?- [milestone].
?- milestone(warren,1983).
Yes
?- milestone(X,Y).
X = rousell
Y = 1972 ;

X= warren
Y = 1983
Yes
?- halt.
```

These are facts
(a particular type of clause)

These are queries
(and replies to the queries)

Using the Prolog shell

The Prolog shell at top-level only accepts queries

When a query result is being displayed:

- Press enter to accept a query answer and return to the top level shell
- Type a semi-colon (;) to request the next answer
- Type w to display fully a long result that Prolog has abbreviated

Terms are the building blocks with which Prolog represents data

Constants

Numbers:
1 -2 3.14

Atoms:
tigger
'100 Acre Wood'

Variables
X A_variable _

Compound terms

likes(pooh_bear,honey)
plus(4,mult(3,plus(1,9)))

Each term is either a constant, a variable, or a compound term. (p29)

Prolog can build compound terms from infix operators

Placing compound terms within compound terms builds tree structures:

- E.g. `html(head(title(blah)),body(...))`
- This is a prefix notation
  - i.e. the name of a tree node comes before its children

Prolog also supports infix expressions: (p31)

- e.g. `X-Y`, 2+3, etc.
- Any infix expression has an equivalent prefix form
- Ensure that you are comfortable with this equivalence

You can ask Prolog to display any term using prefix notation alone

Infix notation is just for human convenience

- Does not affect Prolog's internal data structures
- Requires operator precedence to be defined
  - Otherwise terms such as `x-y-z` are ambiguous

The query write_cannonical(*Term*) will display *Term* without using any infix operators

- Potentially useful for your Prolog experimentation
- We will gloss over how this actually works for now

Unification is Prolog's fundamental operation

Do these terms unify with each other?

1	a	a	**2**	a	b
3	a	A	**4**	a	B
5	tree(l,r)	A	**6**	tree(l,r)	tree(B,C)
7	tree(A,r)	tree(l,C)	**8**	tree(A,r)	tree(A,B)
9	A	a(A)	**10**	a	a(A)

Note: _ is a special variable that unifies with anything.

- Each _ in an expression unifies independently
  - (as if they were all unique, one-use named variables)

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?

(clearly some of the assumptions aren't explicitly stated above!)

Form a model of the situation

Represent each house with the term:

house(Nationality,Pet,Smokes,Drinks,Colour)

Represent the ordered row of houses as follows:

(H1,H2,H3,H4,H5)

We will show that the Zebra Puzzle can be solved using facts, unification and a single (large) query.

- More conventional Prolog programs will define predicates to help solve problems
- (we will talk about these soon)

Question

What sort of a term is:

house(Nationality,Pet,Smokes,Drinks,Colour)

a) number

b) atom

c) compound

d) variable

Question

What sort of a term is:

Nationality

a) number

b) atom

c) compound

d) variable

Question

What sort of a term is:

(H1,H2,H3,H4,H5)

a) number

b) atom

c) compound

d) variable

Define relevant facts

Let's consider one of the puzzle statements:

- The Englishman lives in the red house.
- That is: `house(british,_,_,_,red)`

This term must unify with one of the houses

- Simplify: let's say it unifies with the first house.
- The "houses" 5-tuple would then unify with term:
  - `(house(british,_,_,_,red),_,_,_,_)`

Generalise into a fact:

- firstHouse(HouseTerm,(HouseTerm,_,_,_,_)).
- Really we want "atLeastOneHouse" though...

Define relevant facts

To query two properties about the first house:

```
?- firstHouse(SomeCondition,Houses),
   firstHouse(OtherCondition,Houses).
```

- The comma requires both parts of the query to hold
- Prolog will progressively bind variables via unification
  - Including binding variables within compound terms

Prolog attempts to prove the query

- Variable bindings are a side-effect of the proof

However, we're usually specifically interested in what the variables actually get bound to!

Define relevant facts

Call our atLeastOneHouse fact “exists”

- i.e. there exists a house that has a certain property
- We discussed firstHouse`(A,(A,_,_,_,_))`.
- The generalisation to “at least one house” is:

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

Feeling lost? Ask Prolog!

Test the `exists` predicate on simpler data:

```
?- exists(1,(1,2,3,4,5)).
```

Other things to try:

```
?- exists(2,(1,2,3,4,5)).
?- exists(A,(1,2,3,4,5)).
?- exists(4,(1,2,3,A,5)).
?- exists(1,apple).
?- exists(1,(1,2,3,4)).
```

More constraint-building facts

The facts we are defining allow us to encode the explicit constraints in the problem statement

- We are encoding just the red highlighted part:

  – The green house is immediately to the right of the ivory house.

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

More constraint-building facts

9. Milk is drunk in the middle house.

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

10. The Norwegian lives in the first house.

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

More constraint-building facts

11. The man who smokes Chesterfields lives in the house next to the man with the fox.

```
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)).
```

Express the puzzle as one big query

2. The Englishman lives in the red house.

```
?- 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).
```

Express the puzzle as one big query

3. The Spaniard owns the dog.

```
?- 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).
```

Express the puzzle as one big query

6. The green house is immediately to the right of the ivory house.

```
?- 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).
```

Including queries in your source file

Normal lines in the source file define new clauses

- We've used this to defining fact clauses so far...

Lines beginning with `:-` are "immediate" queries

- (that's a colon followed directly by a hyphen)
- Prolog executes those queries when the file is loaded
- We'll have more to say on this later...

The query `print(T)` prints out term T (in SWI)

- e.g. `print('Hello World').`

Zebra Puzzle

```
> prolog
?- [zebra].
norwegian
japanese
% zebra compiled 0.00 sec, 6,384
bytes
Yes
?- halt.
```

We use `print(WaterDrinker)`, `print(ZebraOwner)` in our query to produce this output

End

- Next lecture:
  - recursive reasoning,
  - lists,
  - arithmetic
  - and more puzzles...

Prolog Lecture 2

- Rules
- Lists
- Arithmetic
- Last-call optimisation
- Backtracking
- Generate and Test

Rules have a head which is true if the body is true

Our Prolog databases have contained only facts

- e.g. `lecturer(prolog,dave).`

Most programs require more complex rules (p8)

- Not just "this is true", but "this is true if that is true"

$$\underbrace{\texttt{rule(X,Y)}}_{\text{head}}\ \texttt{:-}\ \underbrace{\texttt{part1(X), part2(X,Y)}}_{\text{body}}\texttt{.}$$

You can read this as: "rule(X,Y) is true if part1(X) is true and part2(X,Y) is true"

- Note: X and Y also need to be unified appropriately

Variables can be internal to a rule

The variable Z is not present in the clause head:

```
rule2(X) :- thing(X,Z), thang(Z).
```

Read this as "rule2(X) is true if there is a Z such that thing(X,Z) is true and thang(Z) is true"

Prolog and first order logic

The :- symbol is an ASCII-art arrow pointing left

- The "neck" (it's between the clause head and body!)

The arrow represents logical implication

- Mathematically we'd usually write `clause→head`
- It's not as clean as a graphical arrow ...
- In practice Prolog is not as clean as logic either!

Note that quantifiers (∀ and ∃) are not explicitly expressed in Prolog

Rules can be recursive

```
rule3(ground).
rule3(In) :- anotherRule(In,Out),
             rule3(Out).
```

In a recursive reading `rule3(ground)` is a base case, and the other clause is the recursive case.

In a declarative reading both clauses simply represent a situation in which the rule is true.

Prolog identifies clauses by name and arity

We refer to a rule using its clause's head term

The clause

- rule.

is referred to as rule/0 and is different to:

- rule(A).

which is referred to as rule/1 (i.e. it has arity 1)

- rule(_,Y).

would be referred to as rule/2, etc.

Prolog has built-in support for lists

Items are put within square brackets, separated by commas, e.g.[1,2,3,4] (p61)

- The empty list is denoted []

A single list may contain terms of any kind:

- [1,2,an_atom,5,Variable,compound(a,b,c)]

Use a pipe symbol to refer to the tail of a list

- e.g. [Head|Tail] and [1|T] and [1,2,3|T]
- Try unifying [H|T] and [H1,H2|T] with [1,2,3,4]
    - i.e. ?- [H|T] = [1,2,3,4].

We can write rules to find the first and last element of a list

Like functional languages, Prolog uses linked lists

```
first([H|_],H).
```

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

Make sure that you (eventually) understand what this shows you about Prolog's list representation:

write_canonical([1,2,3]).

Question

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

What happens if we ask: last([],X). ?

a) pattern-match exception
b) Prolog says no
c) Prolog says yes, X = []
d) Prolog says yes, X = ???

You should include tests for your clauses in your source code

Example last.pl:

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

% this is a test assertion
:- last([1,2,3],A), A=3.
```

What happens if the test assertion fails?

What happens if we ask:

?- last(List,3).

Prolog provides a way to trace through the execution path

Query trace/0, evaluation then goes step by step

- Press enter to “creep” through the trace
- Pressing s will “skip” over a call

```
?- [last].
% last compiled 0.01 sec, 604 bytes

Yes
?- trace,last([1,2],A).
   Call: (8) last([1, 2], _G187) ? creep
   Call: (9) last([2], _G187) ? creep
   Exit: (9) last([2], 2) ? creep
   Exit: (8) last([1, 2], 2) ? creep

A = 2
Yes
```

Arithmetic Expressions

(AKA “Why Prolog is a bit special/different/surprising”)

What happens if you ask Prolog:

?- A = 1+2.

Arithmetic equality is not the same as Unification

```
?- A = 1+2.
A = 1+2
Yes

?- 1+2 = 3.
No
```

This should raise anyone's procedural eyebrows...

Arithmetical operators get no special treatment!

Unification, unification, unification

In Prolog "=" is not assignment!

"=" does not evaluate expressions!

"=" means "try to unify two terms"

Arithmetic equality is not the same as Unification

```
?- A = money+power.
A = money+power
Yes

?- money+power = A,
    A = +(money,power).
A = money+power
Yes
```

Plus (+) is just forming compound terms

We discussed this in lecture 1

Use the "is" operator to evaluate arithmetic

The "is" operator tells Prolog: (p81)

(1) evaluate the right-hand expression numerically

(2) then unify the expression result with the left

```
?- A is 1+2.
A = 3
Yes

?- A is money+power.
ERROR: is/2: Arithmetic: `money/0' is not a function
```

Ensure that you can explain what will happen here:

?- 3 is 1+2   ?- 1+2 is 3

The right hand side must be a ground term (no variables)

```
?- A is B+2.
ERROR: is: Arguments are not sufficiently
 instantiated

?- 3 is B+2.
ERROR: is: Arguments are not sufficiently
 instantiated
```

It seems that “is” is some sort of magic predicate

- Our predicates do not force instantiation of variables

In fact it can be implemented in logic

- See the supervision worksheet

We can now write a rule about the length of a list

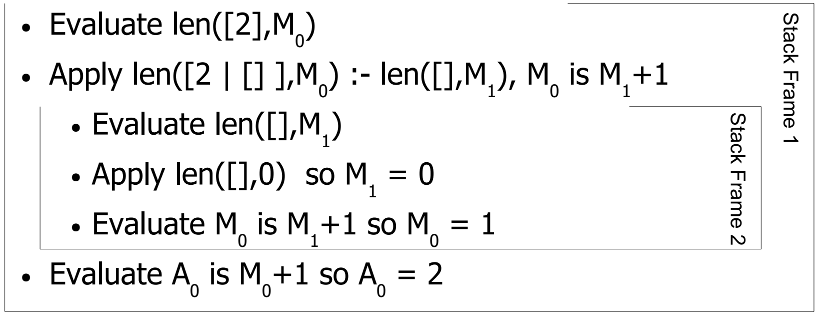
List length:

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

This uses O(N) stack space for a list of length N

List length using O(N) stack space

- Evaluate len([1,2],A).
- Apply len([1| [2]],A_0) :- len([2],M_0), A_0 is M_0+1



- Evaluate len([2],M_0)
- Apply len([2 | []],M_0) :- len([],M_1), M_0 is M_1+1
  - Evaluate len([],M_1)
  - Apply len([],0) so M_1 = 0
  - Evaluate M_0 is M_1+1 so M_0 = 1
- Evaluate A_0 is M_0+1 so A_0 = 2

- Result len([1,2],2)
- This takes O(N) space because of the variables in each frame

List length using O(1) stack space

List length using an accumulator:

```
len2([],Acc,Acc).
len2([_|Tail],Acc,Result) :-
     AccNext is Acc + 1,
     len2(Tail,AccNext,Result).

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

We are passing variables to the recursive len2 call that we do not need to use in future evaluations

- Make sure that you understand an example trace

List length using O(1) stack space

- Evaluate len2([1,2],0,R)
- Apply len2([1| [2]],0,R) :- AccNext is 0+1,
  len2([2],AccNext,R).
  - Evaluate AccNext is 0+1 so AccNext = 1
  - Evaluate len2([2],1,R)

Stack Frame 1

- Apply len2([2| []],1,R) :- AccNext is 1+1,
  len2([],AccNext,R).
  - Evaluate AccNext is 1+1 so AccNext = 2
  - Evaluate len2([],2,R).

Stack Frame 2

- Apply len2([],2,2) so R = 2
- I didn't need to use any subscripts on variable instances!

Last Call Optimisation turns recursion into iteration

Any decent Prolog implementation will apply "Last Call Optimisation" to tail recursion (p186)

- The last query in a clause body can re-use the stack frame of its caller
- This "tail" recursion can be implemented as iteration, drastically reducing the stack space required

Can only apply LCO to rules that are determinate

- The rule must have exhausted all of its options for change: no further computation or backtracking

We can demonstrate that Prolog is applying last call optimisation

Trace will not help

- The debugger will likely interfere with LCO!

How about a "test to destruction"?

```
biglist(0,[]).
biglist(N,[N|T]) :-
   M is N-1,
   biglist(M,T),
   M=M.
```

Prolog uses depth-first search to find answers

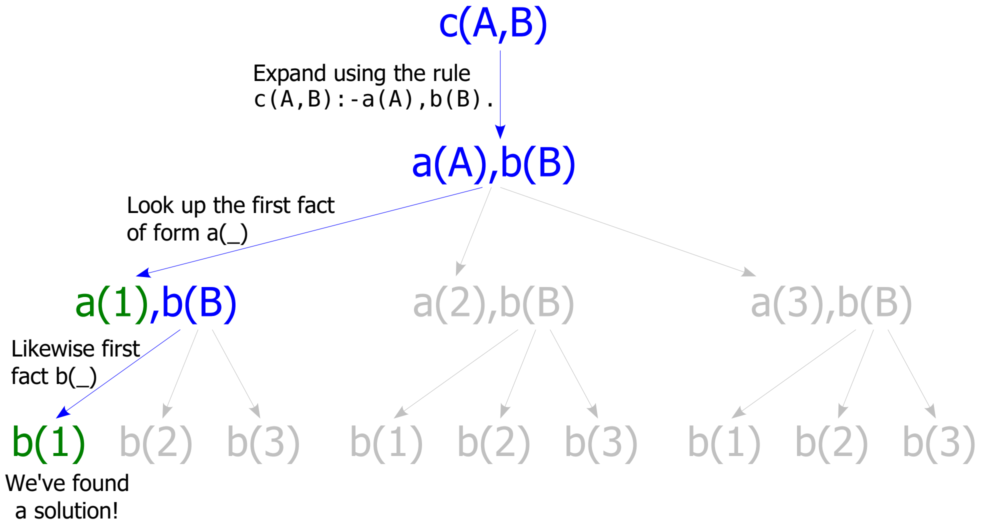
Here is a (boring) program:

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

What does Prolog do when given this query?

c(A,B).

Depth-first solution of query c(A,B)



c(A,B)

Expand using the rule
`c(A,B):-a(A),b(B).`

a(A),b(B)

Look up the first fact
of form a(_)

a(1),b(B) a(2),b(B) a(3),b(B)

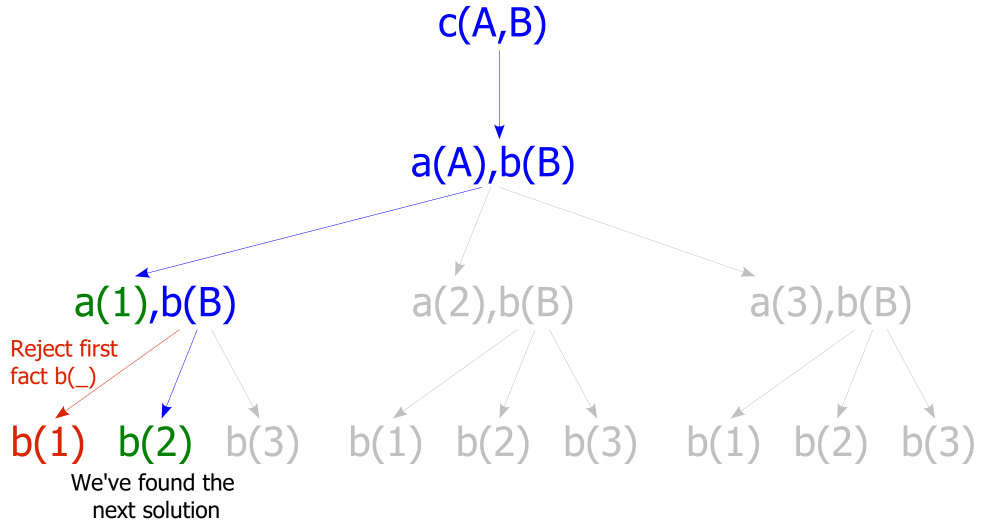
Likewise first
fact b(_)

b(1) b(2) b(3) b(1) b(2) b(3) b(1) b(2) b(3)

We've found
a solution!

Variable bindings: A=1, B=1

Backtrack to find the next solution



c(A,B)

a(A),b(B)

a(1),b(B) a(2),b(B) a(3),b(B)

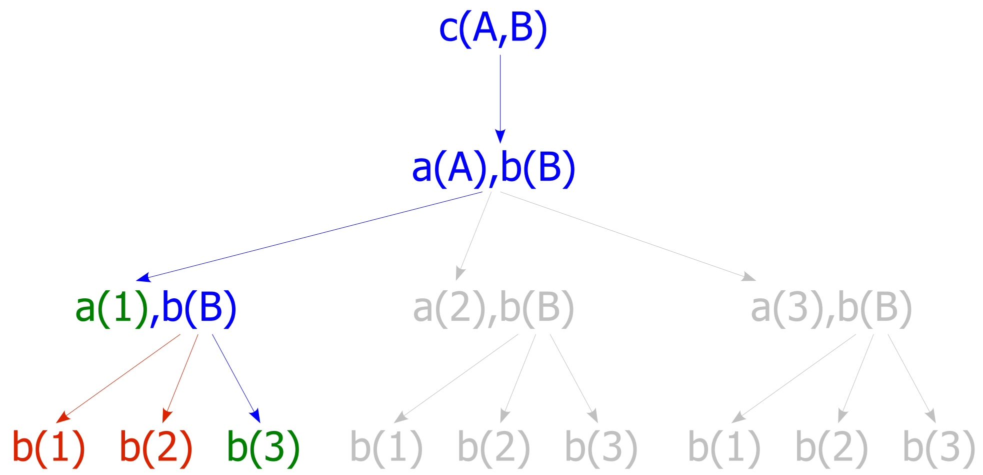
Reject first
fact b(_)

b(1) b(2) b(3) b(1) b(2) b(3) b(1) b(2) b(3)

We've found the
next solution

Variable bindings: A=1, B=2

Backtrack to find another solution



c(A,B)

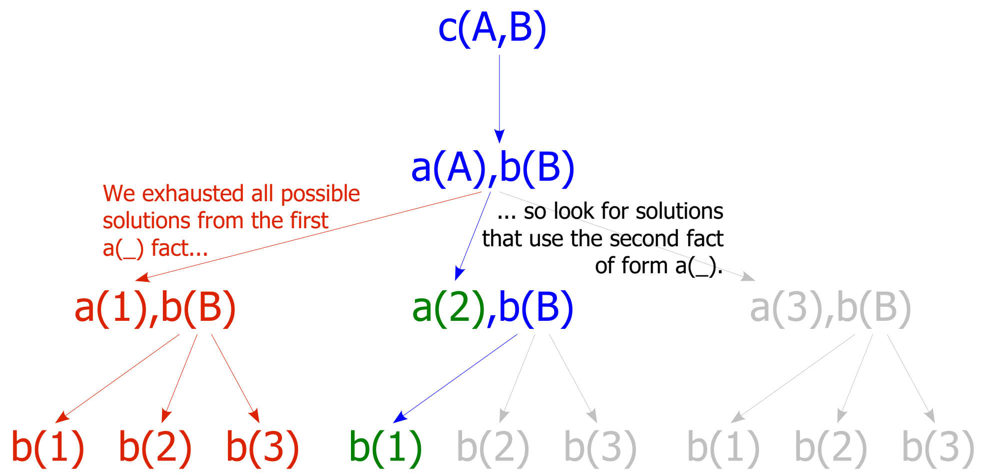
a(A),b(B)

a(1),b(B) a(2),b(B) a(3),b(B)

b(1) b(2) b(3) b(1) b(2) b(3) b(1) b(2) b(3)

Variable bindings: A=1, B=3

Backtrack to find another solution



c(A,B)

a(A),b(B)

We exhausted all possible
solutions from the first
a(_) fact...

... so look for solutions
that use the second fact
of form a(_).

a(1),b(B) a(2),b(B) a(3),b(B)

b(1) b(2) b(3) b(1) b(2) b(3) b(1) b(2) b(3)

Variable bindings: A=2, B=1

Take from a list

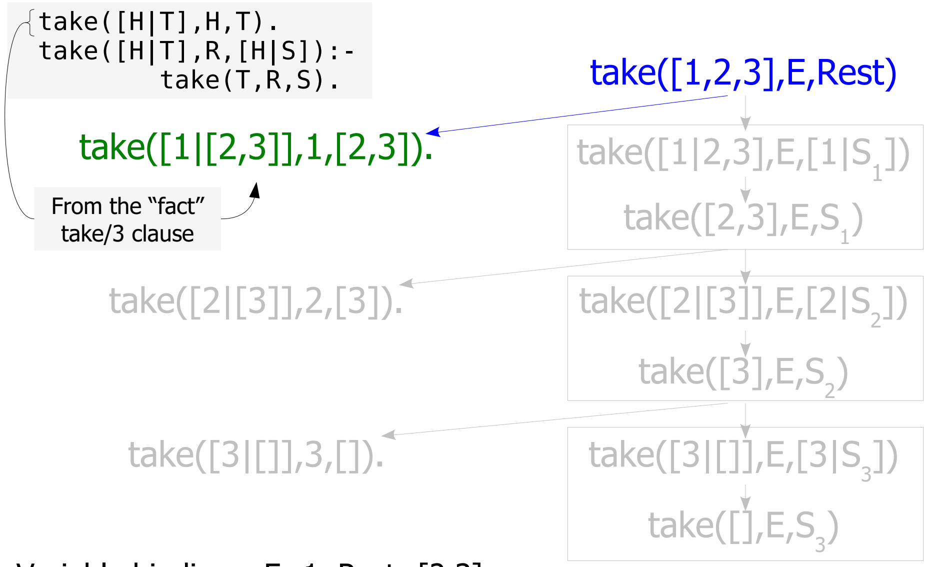
Here is a program that takes an element from a list:

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

What does Prolog do when given the query:

take([1,2,3],E,Rest).

All solutions for take([1,2,3],E,Rest)



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

take([1,2,3],E,Rest)

take([1|[2,3]],1,[2,3]).

From the "fact" take/3 clause

take([1|2,3],E,[1|S_1])

take([2,3],E,S_1)

take([2|[3]],2,[3]).

take([2|[3]],E,[2|S_2])

take([3],E,S_2)

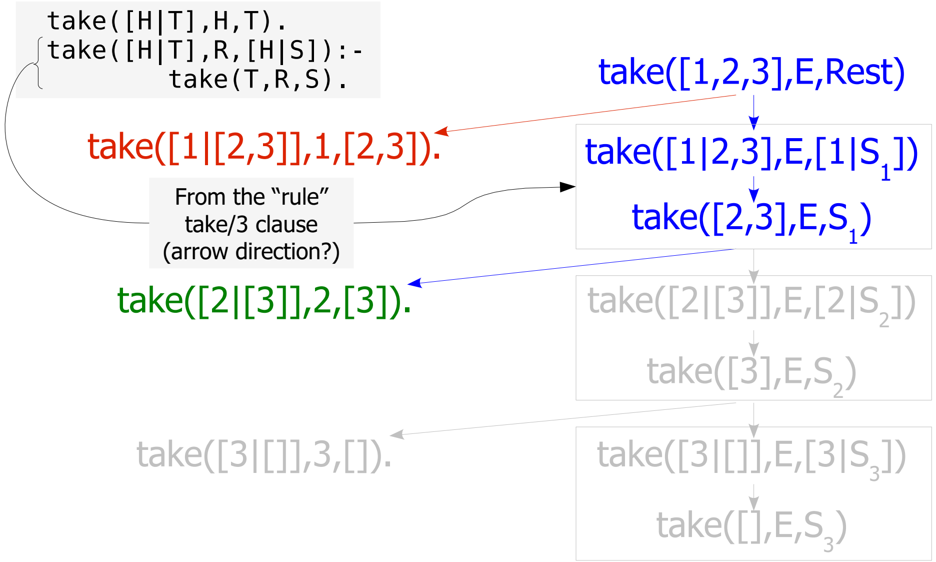
take([3|[]],3,[]).

take([3|[]],E,[3|S_3])

take([],E,S_3)

Variable bindings: E=1, Rest=[2,3]

Backtrack for next solution



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

take([1,2,3],E,Rest)

take([1|[2,3]],1,[2,3]).

take([1|2,3],E,[1|S_1])

From the "rule" take/3 clause (arrow direction?)

take([2,3],E,S_1)

take([2|[3]],2,[3]).

take([2|[3]],E,[2|S_2])

take([3],E,S_2)

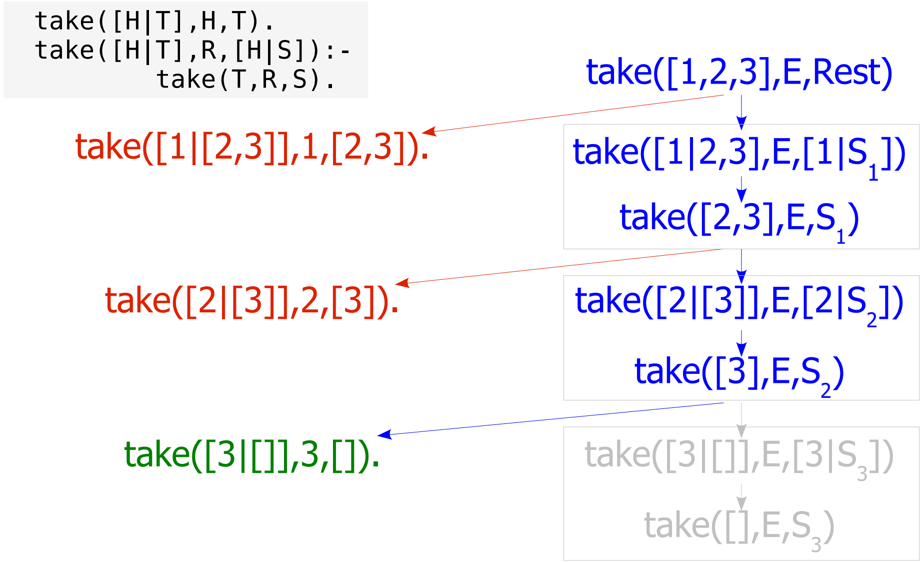
take([3|[]],3,[]).

take([3|[]],E,[3|S_3])

take([],E,S_3)

Variable bindings: E=2, Rest=[1,3], S_1=[3]

Backtrack for another solution



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

take([1,2,3],E,Rest)

take([1|[2,3]],1,[2,3]).

take([1|2,3],E,[1|S_1])

take([2,3],E,S_1)

take([2|[3]],2,[3]).

take([2|[3]],E,[2|S_2])

take([3],E,S_2)

take([3|[]],3,[]).

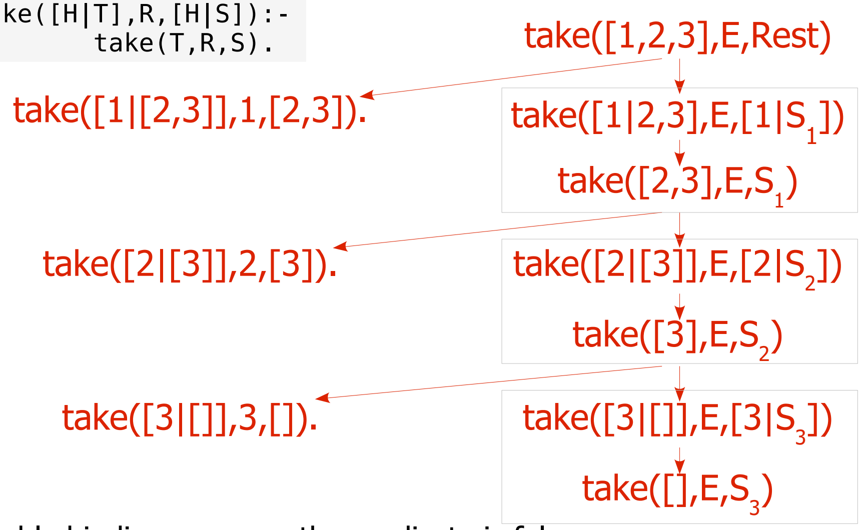
take([3|[]],E,[3|S_3])

take([],E,S_3)

Variable bindings: E=3, Rest=[1,2], S_1=[2], S_2=[]

Prolog says "no"

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



take([1,2,3],E,Rest)

take([1|[2,3]],1,[2,3]).

take([1|2,3],E,[1|S_1])

take([2,3],E,S_1)

take([2|[3]],2,[3]).

take([2|[3]],E,[2|S_2])

take([3],E,S_2)

take([3|[]],3,[]).

take([3|[]],E,[3|S_3])

take([],E,S_3)

Variable bindings: none – the predicate is false

"Find list permutation" predicate is very elegant

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

What is the declarative reading of this predicate?

Dutch national flag

The problem was used by Dijkstra as an exercise in program design and proof.

Take a list and re-order such that red precedes white precedes blue

[red,white,blue,white,red]

↓

[red,red,white,white,blue]

"Generate and Test" is a technique for solving problems like this

(1) Generate a solution

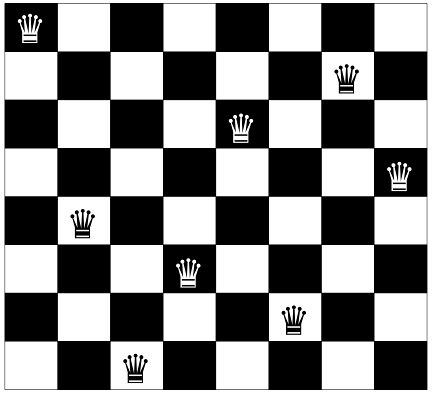
(1) Test if it is valid

(2) If not valid then backtrack to the next generated solution

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

How can we implement checkColours/1?

Place 8 queens so that none can take any other



[1 , 5 , 8 , 6 , 3 , 7 , 2 , 4]

Generate and Test works for 8 Queens too

```
8queens(R) :- perm([1,2,3,4,5,6,7,8],R),
              checkDiagonals(R).
```

Why do I only need to check the diagonals?

Anagrams

Load the dictionary into the Prolog database e.g.:

- word([a,a,r,d,v,a,r,k]).

Generate permutations of the input word and **test** if they are words from the dictionary

or

Generate words from the dictionary and **test** if they are a permutation!

http://www.cl.cam.ac.uk/~dme26/pl/anagram.pl

End

Next lecture:
controlling backtracking with cut, and negation

Prolog Lecture 3

- Symbolic evaluation of arithmetic
- Controlling backtracking: cut
- Negation

Symbolic Evaluation

Let's write some Prolog rules to evaluate symbolic arithmetic expressions such as plus(1,mult(4,5))

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

eval(mult(A,B),C) :- eval(A,A1),
                     eval(B,B1),
                     C is A1 * B1.

eval(A,A).
```

Evaluation starts with the first matching clause

Q: How does Prolog evaluate:

```
eval(plus(1,mult(4,5)),Ans)
```

A: Step 1, see if the first matching clause is true

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

In this case the variable bindings are:

- A = 1, B = mult(4,5) and C = Ans

Next it looks at the body of the rule

The body of the clause with head `eval(plus(A,B),C)` and variable bindings

A = 1, B = mult(4,5) and C = Ans is:

```
eval(1,A1),
eval(mult(4,5),B1),
Ans is A1 + B1.
```

This is a conjunction: all parts must be true for the clause to be true

The body is checked term by term from left to right

First part of the body: `eval(1,A1)`

Try: eval(plus(A,B),C) :- eval(A,A1), eval(B,B1), C is A1 + B1.
Fail because 1 does not unify with plus(A,B)

Try: eval(mult(A,B),C) :- eval(A,A1), eval(B,B1), C is A1 * B1.
Fail because 1 does not unify with mult(A,B)

Try: eval(A,A).
Succeed: eval(1,A1) is true if A1 = 1

The body is checked term by term from left to right

From previous slide, `eval(1,A1)` was provable, with the effect of binding: `A1=1`.

So continuing through the body (note A1 is now bound):

```
eval(1,1),
eval(mult(4,5),B1),
Ans is 1 + B1.
```

The body is checked term by term from left to right

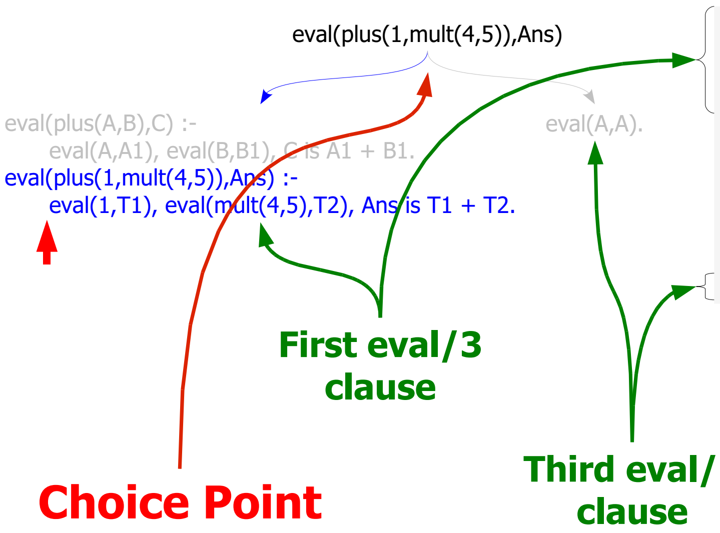
So `eval(mult(4,5),B1)` will bind `B1=20`:

```
eval(1,1),
eval(mult(4,5),20),
Ans is 1 + 20.
```

The body is checked term by term from left to right

Ans will be bound to 21, after “is” does its job.

```
eval(1,1),
eval(mult(4,5),20),
21 is 1 + 20.
```



eval(plus(1,mult(4,5)),Ans)

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

eval(plus(1,mult(4,5)),Ans) :-
eval(1,T1), eval(mult(4,5),T2), Ans is T1 + T2.

eval(A,A).

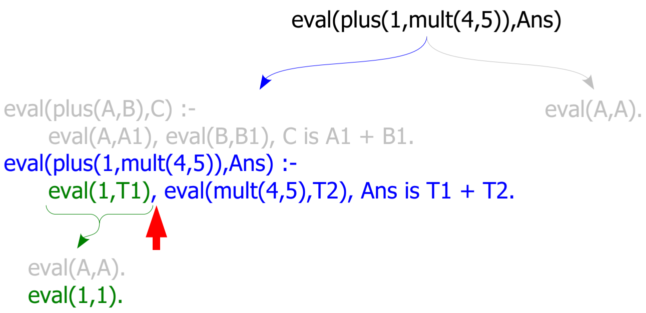
First eval/3 clause

Third eval/3 clause

Choice Point

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

eval(mult(A,B),C):-
    eval(A,A1),
    eval(B,B1),
    C is A1 * B1.

eval(A,A).
```

Be sure that you understand why the second eval/3 clause does not appear in this choice point



eval(plus(1,mult(4,5)),Ans)

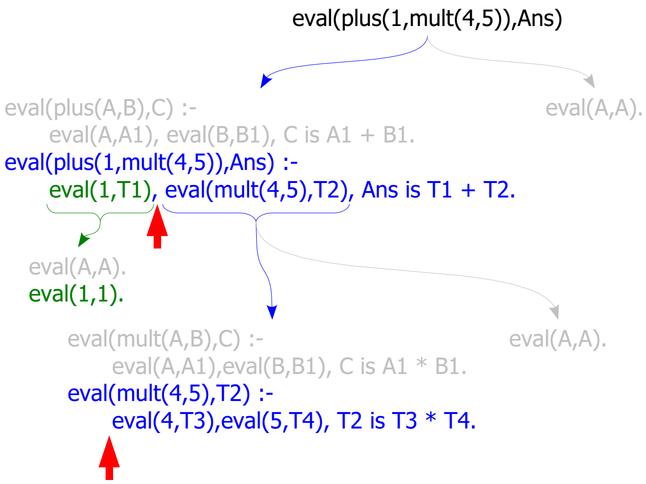
eval(plus(A,B),C) :-
eval(A,A1), eval(B,B1), C is A1 + B1.

eval(A,A).

eval(plus(1,mult(4,5)),Ans) :-
eval(1,T1), eval(mult(4,5),T2), Ans is T1 + T2.

eval(A,A).
eval(1,1).

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

eval(mult(A,B),C):-
    eval(A,A1),
    eval(B,B1),
    C is A1 * B1.

eval(A,A).
```



eval(plus(1,mult(4,5)),Ans)

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

eval(A,A).

eval(plus(1,mult(4,5)),Ans) :-
eval(1,T1), eval(mult(4,5),T2), Ans is T1 + T2.

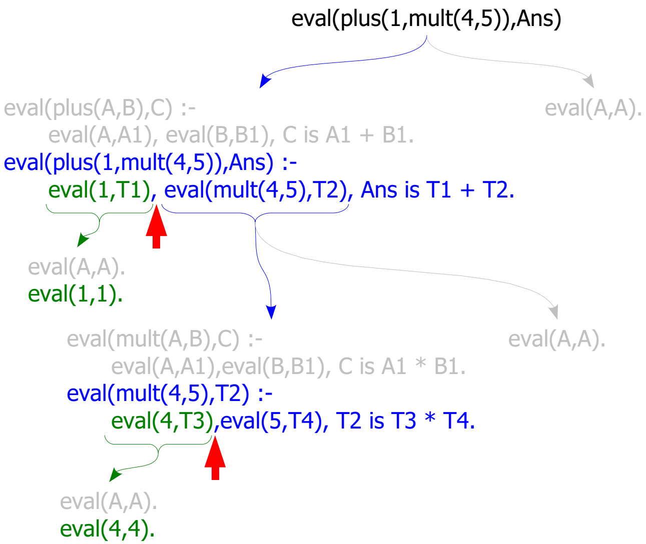
eval(A,A).
eval(1,1).

eval(mult(A,B),C) :-
eval(A,A1),eval(B,B1), C is A1 * B1.

eval(A,A).

eval(mult(4,5),T2) :-
eval(4,T3),eval(5,T4), T2 is T3 * T4.

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

eval(mult(A,B),C):-
    eval(A,A1),
    eval(B,B1),
    C is A1 * B1.

eval(A,A).
```



eval(plus(1,mult(4,5)),Ans)

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

eval(A,A).

eval(plus(1,mult(4,5)),Ans) :-
eval(1,T1), eval(mult(4,5),T2), Ans is T1 + T2.

eval(A,A).
eval(1,1).

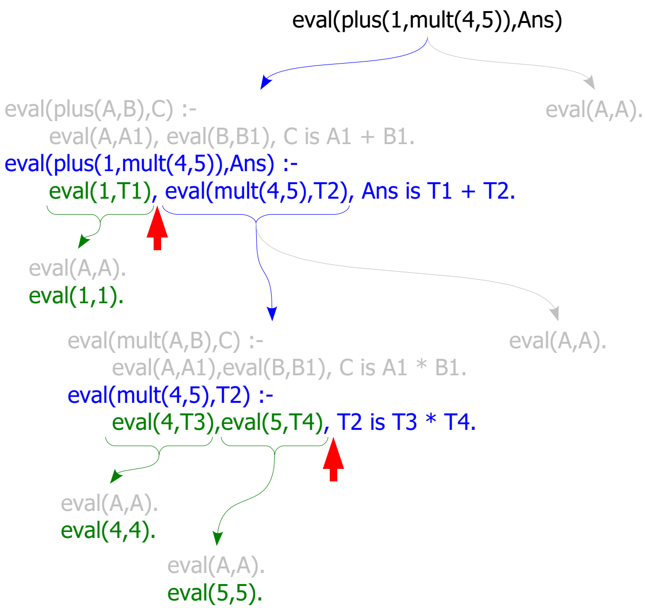
eval(mult(A,B),C) :-
eval(A,A1),eval(B,B1), C is A1 * B1.

eval(A,A).

eval(mult(4,5),T2) :-
eval(4,T3),eval(5,T4), T2 is T3 * T4.

eval(A,A).
eval(4,4).

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

eval(mult(A,B),C):-
    eval(A,A1),
    eval(B,B1),
    C is A1 * B1.

eval(A,A).
```



eval(plus(1,mult(4,5)),Ans)

eval(plus(A,B),C) :-
eval(A,A1), eval(B,B1), C is A1 + B1.
eval(plus(1,mult(4,5)),Ans) :-
eval(1,T1), eval(mult(4,5),T2), Ans is T1 + T2.

eval(A,A).

eval(A,A).
eval(1,1).

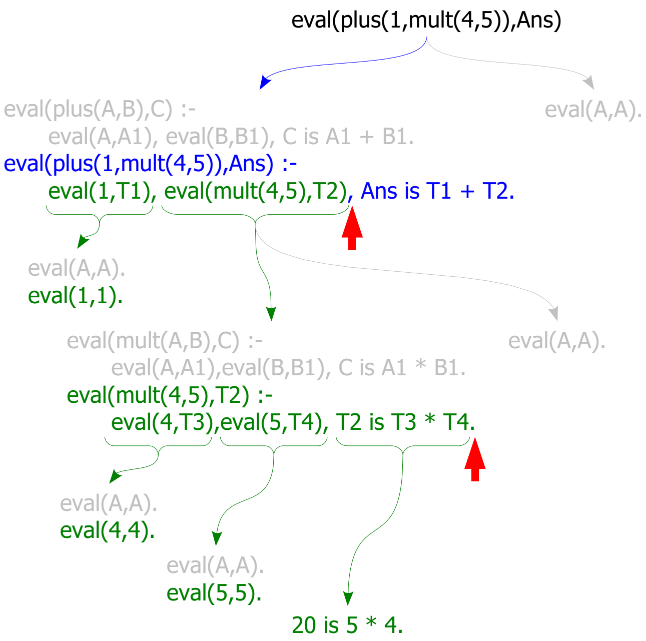
eval(mult(A,B),C) :-
eval(A,A1),eval(B,B1), C is A1 * B1.
eval(mult(4,5),T2) :-
eval(4,T3),eval(5,T4), T2 is T3 * T4.

eval(A,A).

eval(A,A).
eval(4,4).

eval(A,A).
eval(5,5).

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

eval(mult(A,B),C):-
    eval(A,A1),
    eval(B,B1),
    C is A1 * B1.

eval(A,A).
```



eval(plus(1,mult(4,5)),Ans)

eval(plus(A,B),C) :-
eval(A,A1), eval(B,B1), C is A1 + B1.
eval(plus(1,mult(4,5)),Ans) :-
eval(1,T1), eval(mult(4,5),T2), Ans is T1 + T2.

eval(A,A).

eval(A,A).
eval(1,1).

eval(mult(A,B),C) :-
eval(A,A1),eval(B,B1), C is A1 * B1.
eval(mult(4,5),T2) :-
eval(4,T3),eval(5,T4), T2 is T3 * T4.

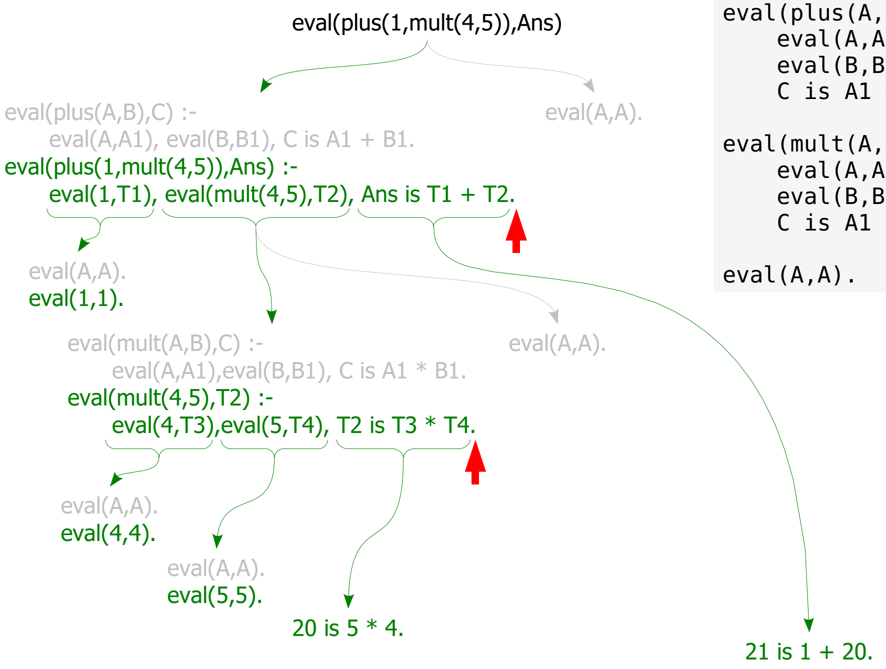
eval(A,A).

eval(A,A).
eval(4,4).

eval(A,A).
eval(5,5).

20 is 5 * 4.

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

eval(mult(A,B),C):-
    eval(A,A1),
    eval(B,B1),
    C is A1 * B1.

eval(A,A).
```



eval(plus(1,mult(4,5)),Ans)

eval(plus(A,B),C) :-
eval(A,A1), eval(B,B1), C is A1 + B1.
eval(plus(1,mult(4,5)),Ans) :-
eval(1,T1), eval(mult(4,5),T2), Ans is T1 + T2.

eval(A,A).

eval(A,A).
eval(1,1).

eval(mult(A,B),C) :-
eval(A,A1),eval(B,B1), C is A1 * B1.
eval(mult(4,5),T2) :-
eval(4,T3),eval(5,T4), T2 is T3 * T4.

eval(A,A).

eval(A,A).
eval(4,4).

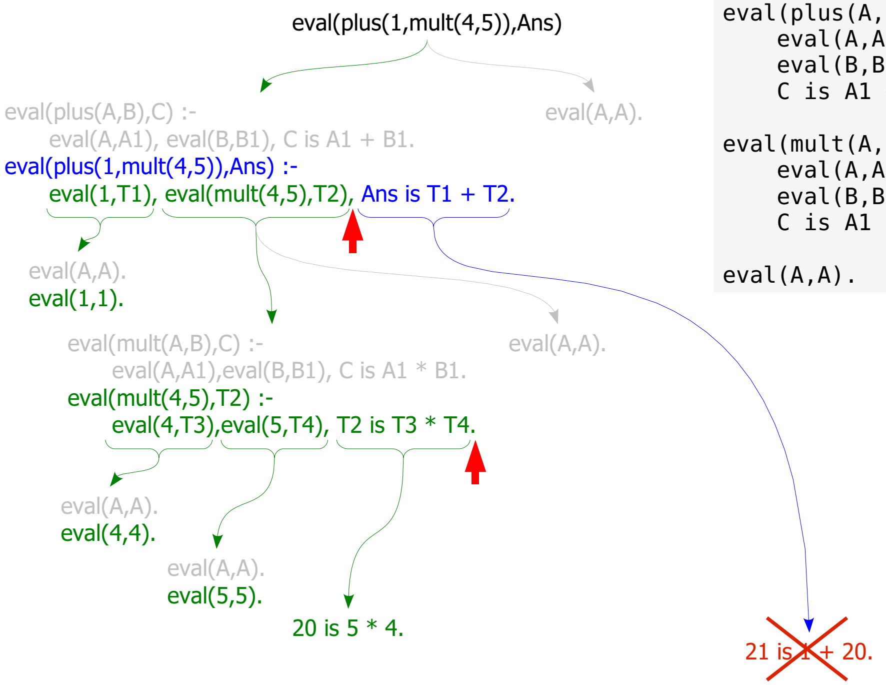
eval(A,A).
eval(5,5).

20 is 5 * 4.

21 is 1 + 20.

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

eval(mult(A,B),C):-
    eval(A,A1),
    eval(B,B1),
    C is A1 * B1.

eval(A,A).
```

What happens if we use backtracking and ask Prolog for the next solution?



eval(plus(1,mult(4,5)),Ans)

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

eval(A,A).

eval(plus(1,mult(4,5)),Ans) :-
eval(1,T1), eval(mult(4,5),T2), Ans is T1 + T2.

eval(A,A).
eval(1,1).

eval(mult(A,B),C) :-
eval(A,A1),eval(B,B1), C is A1 * B1.

eval(A,A).

eval(mult(4,5),T2) :-
eval(4,T3),eval(5,T4), T2 is T3 * T4.

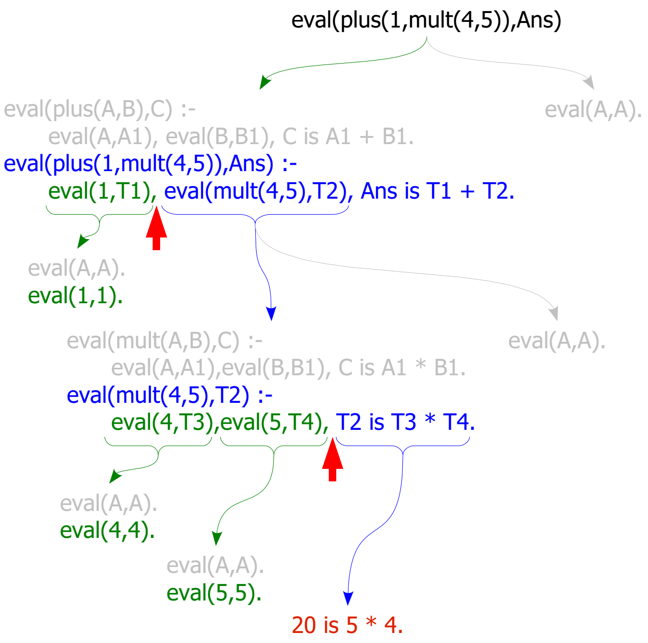
eval(A,A).
eval(4,4).

eval(A,A).
eval(5,5).

20 is 5 * 4.

~~21 is 1 + 20.~~

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

eval(mult(A,B),C):-
    eval(A,A1),
    eval(B,B1),
    C is A1 * B1.

eval(A,A).
```



eval(plus(1,mult(4,5)),Ans)

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

eval(A,A).

eval(plus(1,mult(4,5)),Ans) :-
eval(1,T1), eval(mult(4,5),T2), Ans is T1 + T2.

eval(A,A).
eval(1,1).

eval(mult(A,B),C) :-
eval(A,A1),eval(B,B1), C is A1 * B1.

eval(A,A).

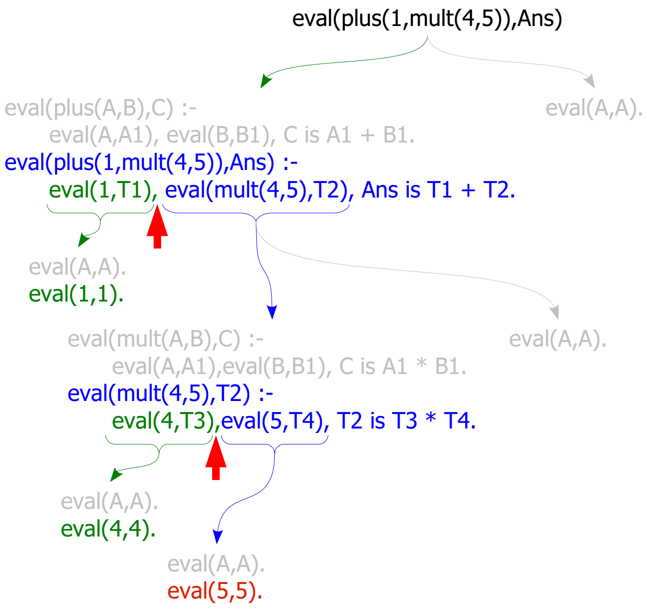
eval(mult(4,5),T2) :-
eval(4,T3),eval(5,T4), T2 is T3 * T4.

eval(A,A).
eval(4,4).

eval(A,A).
eval(5,5).

20 is 5 * 4.

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

eval(mult(A,B),C):-
    eval(A,A1),
    eval(B,B1),
    C is A1 * B1.

eval(A,A).
```



eval(plus(1,mult(4,5)),Ans)

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

eval(A,A).

eval(plus(1,mult(4,5)),Ans) :-
eval(1,T1), eval(mult(4,5),T2), Ans is T1 + T2.

eval(A,A).
eval(1,1).

eval(mult(A,B),C) :-
eval(A,A1),eval(B,B1), C is A1 * B1.

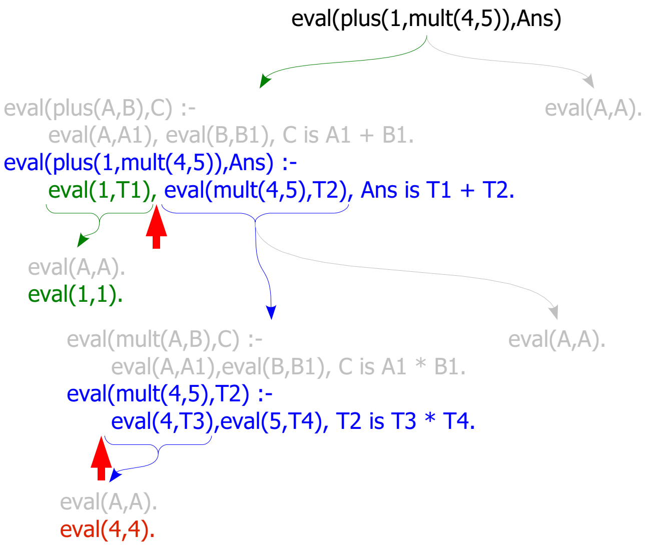
eval(A,A).

eval(mult(4,5),T2) :-
eval(4,T3),eval(5,T4), T2 is T3 * T4.

eval(A,A).
eval(4,4).

eval(A,A).
eval(5,5).

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

eval(mult(A,B),C):-
    eval(A,A1),
    eval(B,B1),
    C is A1 * B1.

eval(A,A).
```



eval(plus(1,mult(4,5)),Ans)

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

eval(A,A).

eval(plus(1,mult(4,5)),Ans) :-
eval(1,T1), eval(mult(4,5),T2), Ans is T1 + T2.

eval(A,A).
eval(1,1).

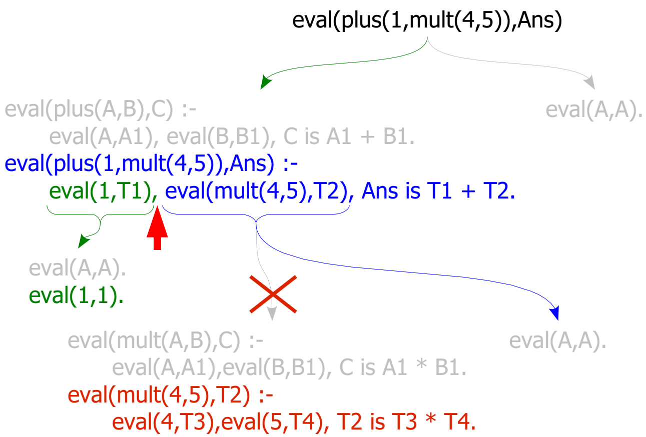
eval(mult(A,B),C) :-
eval(A,A1),eval(B,B1), C is A1 * B1.

eval(A,A).

eval(mult(4,5),T2) :-
eval(4,T3),eval(5,T4), T2 is T3 * T4.

eval(A,A).
eval(4,4).

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

eval(mult(A,B),C):-
    eval(A,A1),
    eval(B,B1),
    C is A1 * B1.

eval(A,A).
```



eval(plus(1,mult(4,5)),Ans)

eval(plus(A,B),C) :-
    eval(A,A1), eval(B,B1), C is A1 + B1.
eval(plus(1,mult(4,5)),Ans) :-
    eval(1,T1), eval(mult(4,5),T2), Ans is T1 + T2.

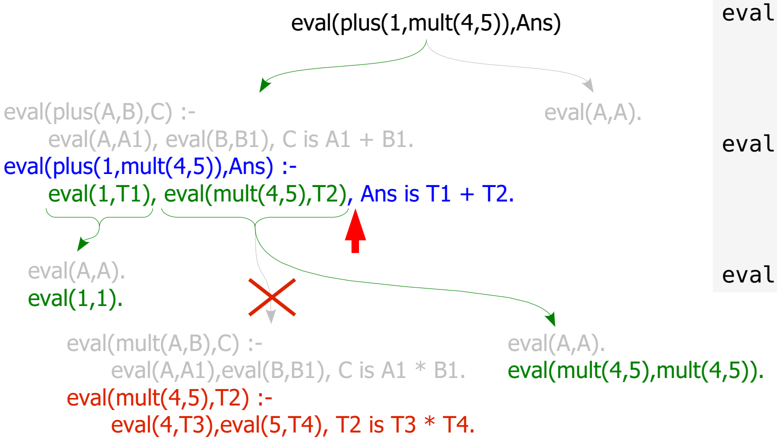
eval(A,A).

eval(A,A).
eval(1,1).

eval(mult(A,B),C) :-
    eval(A,A1),eval(B,B1), C is A1 * B1.
eval(mult(4,5),T2) :-
    eval(4,T3),eval(5,T4), T2 is T3 * T4.

eval(A,A).

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

eval(mult(A,B),C):-
    eval(A,A1),
    eval(B,B1),
    C is A1 * B1.

eval(A,A).
```



eval(plus(1,mult(4,5)),Ans)

eval(plus(A,B),C) :-
    eval(A,A1), eval(B,B1), C is A1 + B1.
eval(plus(1,mult(4,5)),Ans) :-
    eval(1,T1), eval(mult(4,5),T2), Ans is T1 + T2.

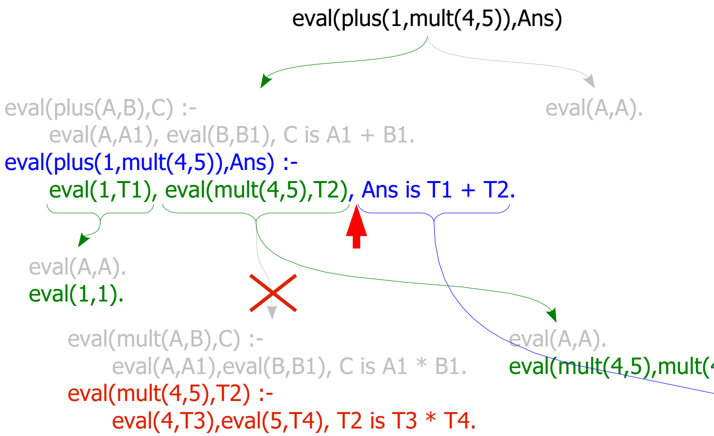
eval(A,A).

eval(A,A).
eval(1,1).

eval(mult(A,B),C) :-
    eval(A,A1),eval(B,B1), C is A1 * B1.
eval(mult(4,5),T2) :-
    eval(4,T3),eval(5,T4), T2 is T3 * T4.

eval(A,A).
eval(mult(4,5),mult(4,5)).

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

eval(mult(A,B),C):-
    eval(A,A1),
    eval(B,B1),
    C is A1 * B1.

eval(A,A).
```



eval(plus(1,mult(4,5)),Ans)

eval(plus(A,B),C) :-
    eval(A,A1), eval(B,B1), C is A1 + B1.
eval(plus(1,mult(4,5)),Ans) :-
    eval(1,T1), eval(mult(4,5),T2), Ans is T1 + T2.

eval(A,A).

eval(A,A).
eval(1,1).

eval(mult(A,B),C) :-
    eval(A,A1),eval(B,B1), C is A1 * B1.
eval(mult(4,5),T2) :-
    eval(4,T3),eval(5,T4), T2 is T3 * T4.

eval(A,A).
eval(mult(4,5),mult(4,5)).

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

eval(mult(A,B),C):-
    eval(A,A1),
    eval(B,B1),
    C is A1 * B1.

eval(A,A).
```

Ans is 1 + mult(4,5)

Ouch...

(a) Eliminate spurious solutions by making your clauses orthogonal

Need to eliminate the (unwanted) choice point

A way to do this: make sure only one clause matches: **eval(A,A)** becomes **eval(gnd(A),A)**.

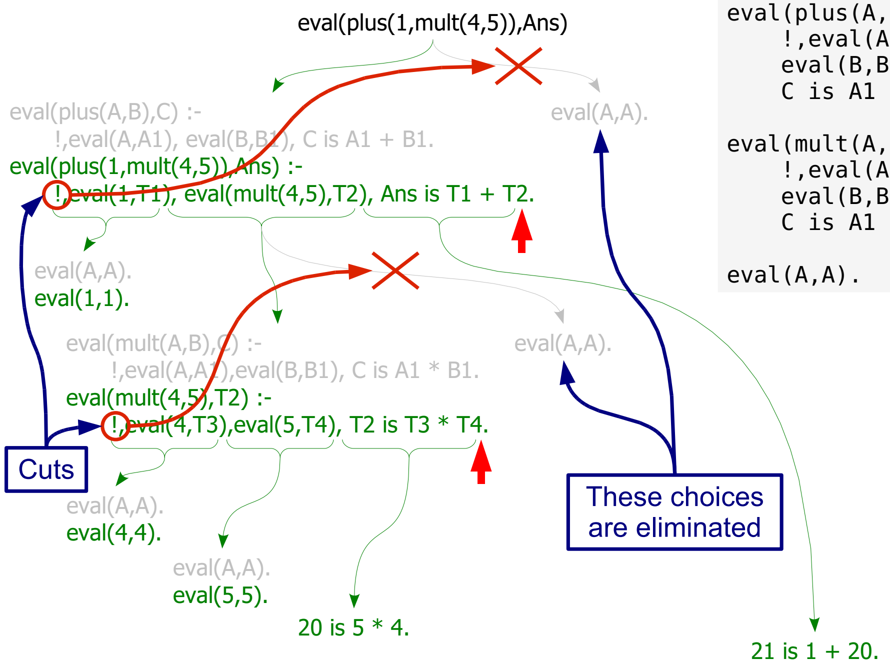
```
eval(plus(A,B),C) :- eval(A,A1),
                     eval(B,B1),
                     C is A1 + B1.
eval(mult(A,B),C) :- eval(A,A1),
                     eval(B,B1),
                     C is A1 * B1.
eval(gnd(A),A).
```

(b) Eliminate spurious solutions by explicitly discarding choice points

Alternatively we can tell Prolog to commit to its first choice and discard the choice point (p114)

We do this with the cut operator. Written: !

```
eval(plus(A,B),C) :- !,eval(A,A1),
                     eval(B,B1),
                     C is A1 + B1.
eval(mult(A,B),C) :- !,eval(A,A1),
                     eval(B,B1),
                     C is A1 * B1.
eval(A,A).
```



eval(plus(1,mult(4,5)),Ans)

eval(plus(A,B),C) :-
!,eval(A,A1), eval(B,B1), C is A1 + B1.

eval(plus(1,mult(4,5)),Ans) :-
!,eval(1,T1), eval(mult(4,5),T2), Ans is T1 + T2.

eval(A,A).

eval(A,A).
eval(1,1).

eval(mult(A,B),C) :-
!,eval(A,A1),eval(B,B1), C is A1 * B1.

eval(mult(4,5),T2) :-
!,eval(4,T3),eval(5,T4), T2 is T3 * T4.

eval(A,A).

Cuts

eval(A,A).
eval(4,4).

eval(A,A).
eval(5,5).

20 is 5 * 4.

These choices are eliminated

21 is 1 + 20.

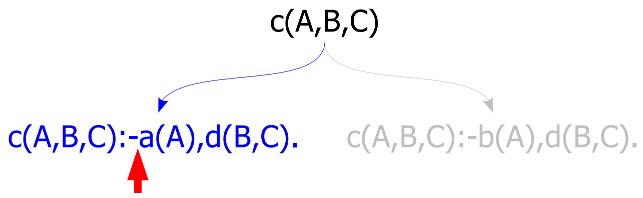
```
eval(plus(A,B),C):-
    !,eval(A,A1),
    eval(B,B1),
    C is A1 + B1.

eval(mult(A,B),C):-
    !,eval(A,A1),
    eval(B,B1),
    C is A1 * B1.

eval(A,A).
```

Cutting out choice

Whenever Prolog evaluates a cut it discards all choice points back to the parent clause

An example:

```
a(1).        c(A,B,C) :- a(A),d(B,C).
a(2).        c(A,B,C) :- b(A),d(B,C).
a(3).        d(B,C) :- a(B),!,a(C).
b(apple).    d(B,_) :- b(B).
b(orange).
```

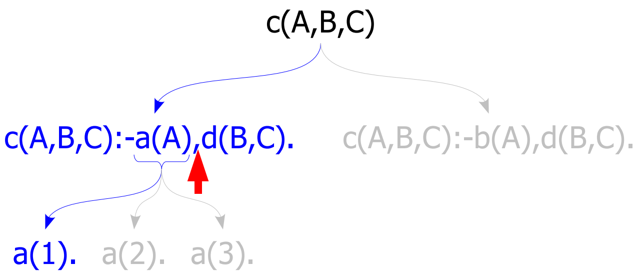


c(A,B,C)

c(A,B,C):-a(A),d(B,C).

c(A,B,C):-b(A),d(B,C).

```
a(1).
a(2).
a(3).
b(apple).
b(orange).
c(A,B,C):-a(A),d(B,C).
c(A,B,C):-b(A),d(B,C).
d(B,C):-a(B),!,a(C).
d(B,_):-b(B).
```

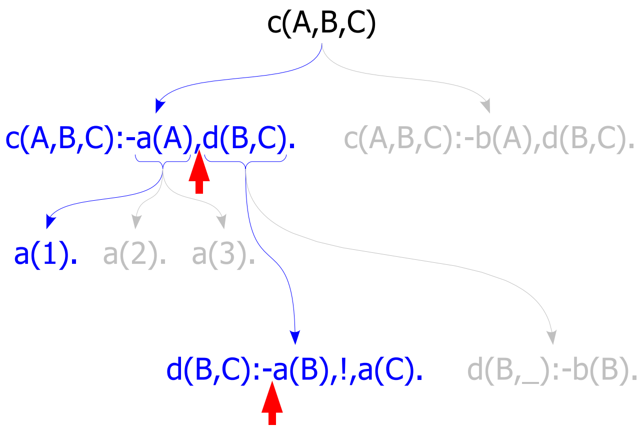


c(A,B,C)

c(A,B,C):-a(A),d(B,C). c(A,B,C):-b(A),d(B,C).

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

```
a(1).
a(2).
a(3).
b(apple).
b(orange).
c(A,B,C):-a(A),d(B,C).
c(A,B,C):-b(A),d(B,C).
d(B,C):-a(B),!,a(C).
d(B,_):-b(B).
```



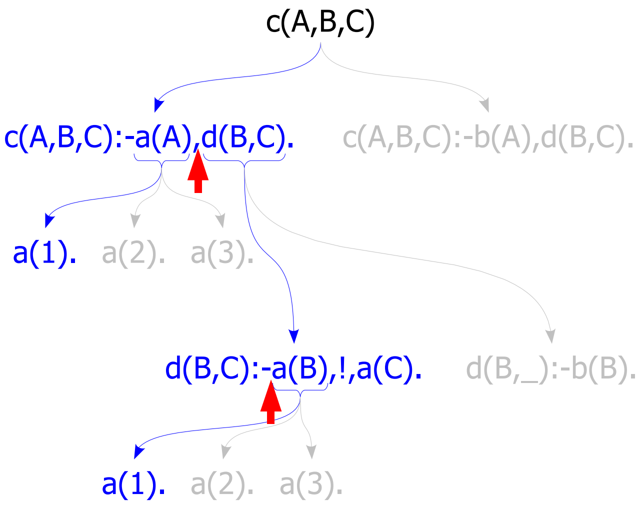
c(A,B,C)

c(A,B,C):-a(A),d(B,C). c(A,B,C):-b(A),d(B,C).

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

d(B,C):-a(B),!,a(C). d(B,_):-b(B).

```
a(1).
a(2).
a(3).
b(apple).
b(orange).
c(A,B,C):-a(A),d(B,C).
c(A,B,C):-b(A),d(B,C).
d(B,C):-a(B),!,a(C).
d(B,_):-b(B).
```



c(A,B,C)

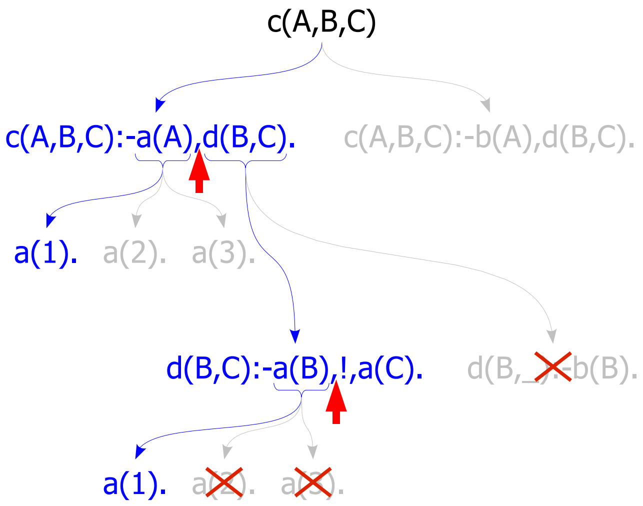
c(A,B,C):-a(A),d(B,C). c(A,B,C):-b(A),d(B,C).

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

d(B,C):-a(B),!,a(C). d(B,_):-b(B).

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

```
a(1).
a(2).
a(3).
b(apple).
b(orange).
c(A,B,C):-a(A),d(B,C).
c(A,B,C):-b(A),d(B,C).
d(B,C):-a(B),!,a(C).
d(B,_):-b(B).
```



c(A,B,C)

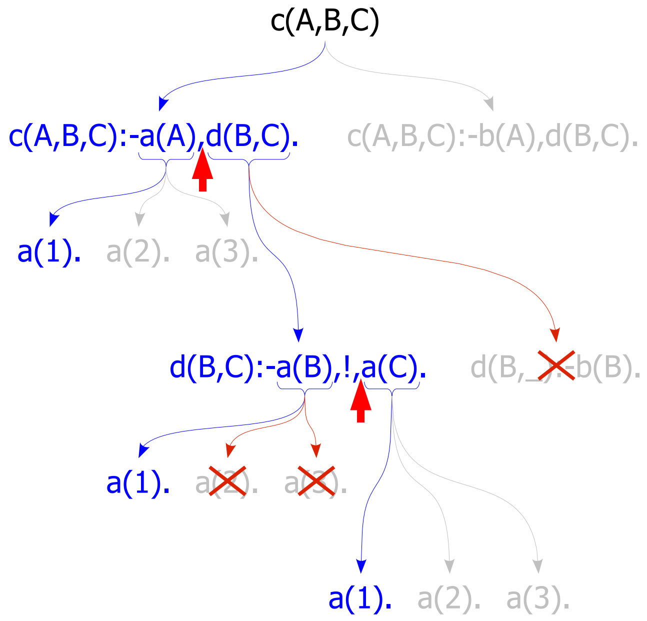
c(A,B,C):-a(A),d(B,C). c(A,B,C):-b(A),d(B,C).

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

d(B,C):-a(B),!,a(C). ~~d(B,_):-b(B).~~

a(1). ~~a(2).~~ ~~a(3).~~

```
a(1).
a(2).
a(3).
b(apple).
b(orange).
c(A,B,C):-a(A),d(B,C).
c(A,B,C):-b(A),d(B,C).
d(B,C):-a(B),!,a(C).
d(B,_):-b(B).
```



c(A,B,C)

c(A,B,C):-a(A),d(B,C). c(A,B,C):-b(A),d(B,C).

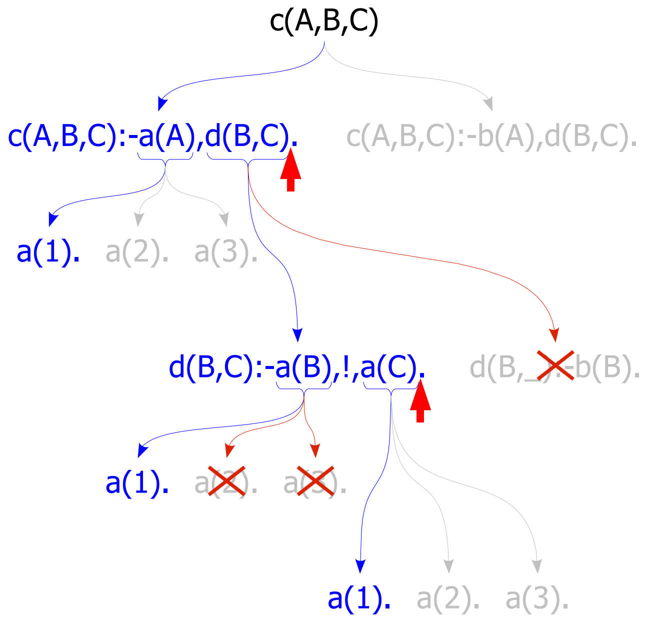
a(1). a(2). a(3).

d(B,C):-a(B),!,a(C). ~~d(B,_):-b(B).~~

a(1). ~~a(2).~~ ~~a(3).~~

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

```
a(1).
a(2).
a(3).
b(apple).
b(orange).
c(A,B,C):-a(A),d(B,C).
c(A,B,C):-b(A),d(B,C).
d(B,C):-a(B),!,a(C).
d(B,_):-b(B).
```



c(A,B,C)

c(A,B,C):-a(A),d(B,C). c(A,B,C):-b(A),d(B,C).

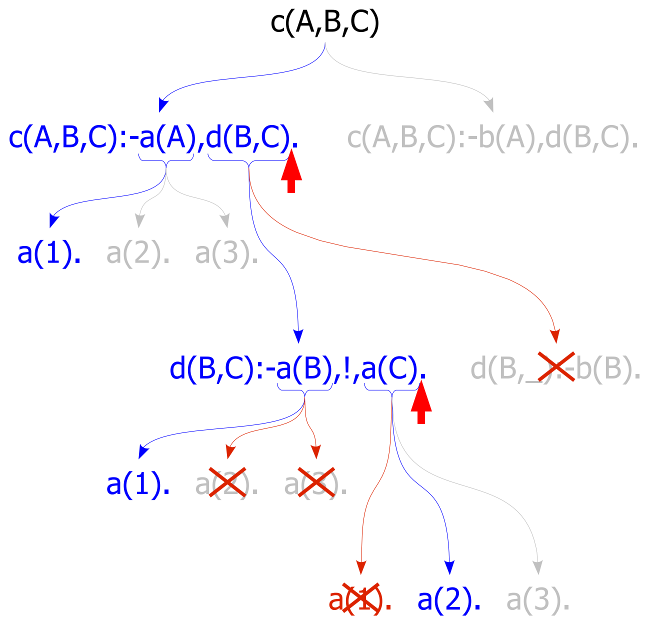
a(1). a(2). a(3).

d(B,C):-a(B),!,a(C). ~~d(B,_):-b(B).~~

a(1). ~~a(2).~~ ~~a(3).~~

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

```
a(1).
a(2).
a(3).
b(apple).
b(orange).
c(A,B,C):-a(A),d(B,C).
c(A,B,C):-b(A),d(B,C).
d(B,C):-a(B),!,a(C).
d(B,_):-b(B).
```



c(A,B,C)

c(A,B,C):-a(A),d(B,C). c(A,B,C):-b(A),d(B,C).

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

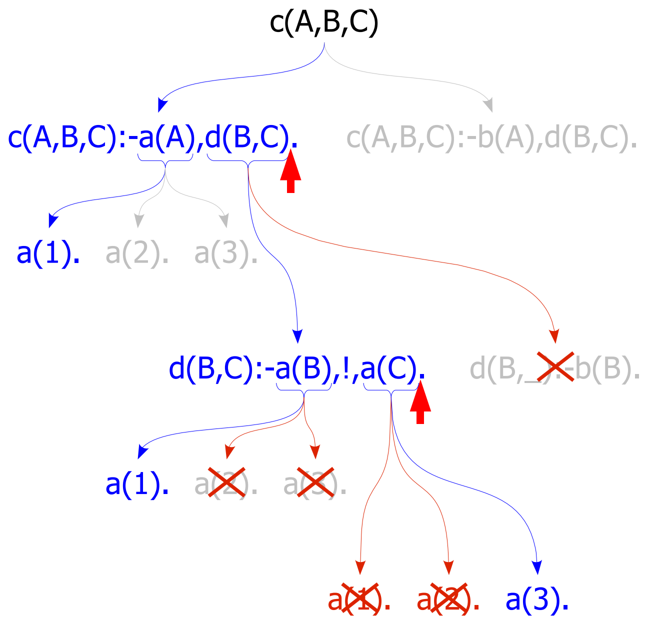
d(B,C):-a(B),!,a(C). ~~d(B,_):-b(B).~~

a(1). ~~a(2).~~ ~~a(3).~~

~~a(1).~~ a(2). a(3).

```
a(1).
a(2).
a(3).
b(apple).
b(orange).
c(A,B,C):-a(A),d(B,C).
c(A,B,C):-b(A),d(B,C).
d(B,C):-a(B),!,a(C).
d(B,_):-b(B).
```

Backtrack once



c(A,B,C)

c(A,B,C):-a(A),d(B,C). c(A,B,C):-b(A),d(B,C).

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

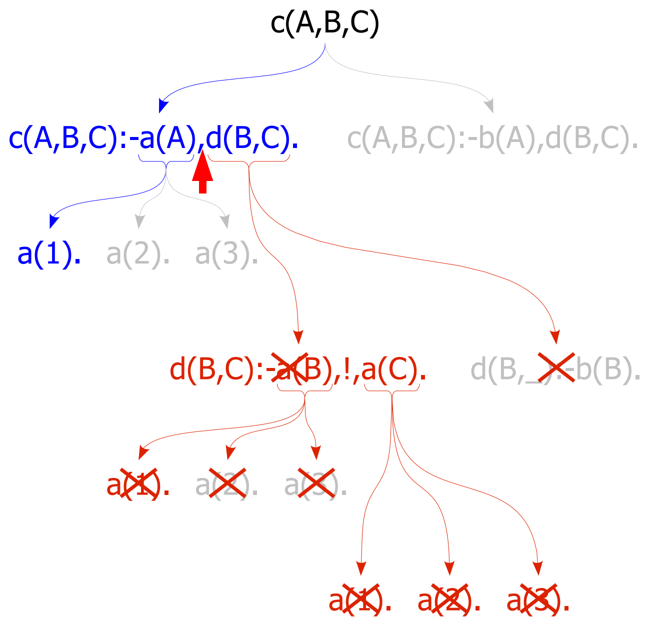
d(B,C):-a(B),!,a(C). ~~d(B,_):-b(B).~~

a(1). ~~a(2).~~ ~~a(3).~~

~~a(1).~~ ~~a(2).~~ a(3).

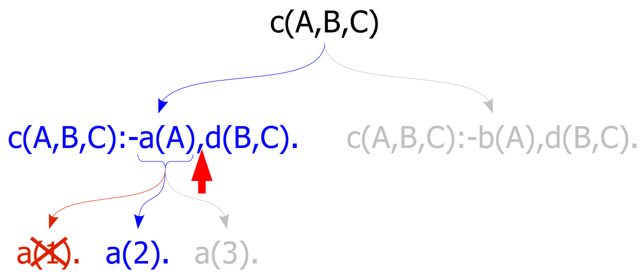
```
a(1).
a(2).
a(3).
b(apple).
b(orange).
c(A,B,C):-a(A),d(B,C).
c(A,B,C):-b(A),d(B,C).
d(B,C):-a(B),!,a(C).
d(B,_):-b(B).
```

Backtrack twice



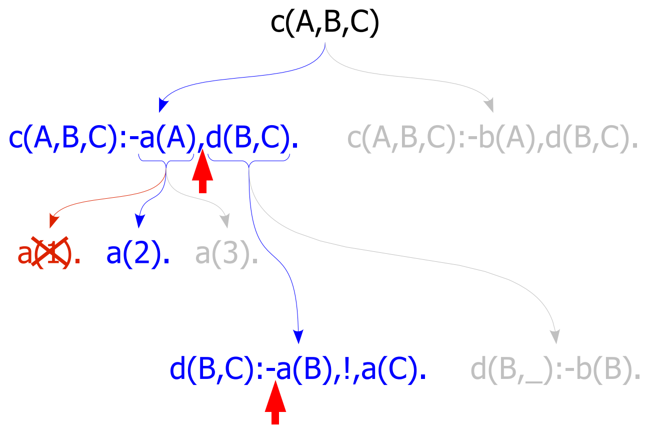
```
a(1).
a(2).
a(3).
b(apple).
b(orange).
c(A,B,C):-a(A),d(B,C).
c(A,B,C):-b(A),d(B,C).
d(B,C):-a(B),!,a(C).
d(B,_):-b(B).
```

Backtrack three times



```
a(1).
a(2).
a(3).
b(apple).
b(orange).
c(A,B,C):-a(A),d(B,C).
c(A,B,C):-b(A),d(B,C).
d(B,C):-a(B),!,a(C).
d(B,_):-b(B).
```

First a/1 has other solutions



```
a(1).
a(2).
a(3).
b(apple).
b(orange).
c(A,B,C):-a(A),d(B,C).
c(A,B,C):-b(A),d(B,C).
d(B,C):-a(B),!,a(C).
d(B,_):-b(B).
```

Can try to derive d/2 afresh...

Cut can change the logical meaning of your program

```
p :- a,b.
p :- c.
```

$p \Leftrightarrow (a \wedge b) \vee c$

```
p :- a,!,b.
p :- c.
```

$p \Leftrightarrow (a \wedge b) \vee (\neg a \wedge c)$

This is a red cut – **DANGER!** (p128)

Cut can be used for efficiency reasons

```
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).
```

If the second clause succeeds the third cannot

- we don't need to keep a choice point
- yet the interpreter cannot infer this on its own

Cut can be used for efficiency reasons

```
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).
```

Add a cut to make the orthogonality explicit

- This is a green cut – it just helps program execution go faster

We could go one step further at the expense of readability

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

The comparison in the third clause is no longer necessary

- but each clause no longer stands on its own
- stylistic preference – I avoid doing this

Cut gives us more expressive power

```
isDifferent(A,A) :- !,fail.
isDifferent(_,_).
```

isDifferent(A,B) is true iff A and B do not unify

Questions that you should be able to answer:

- Is this a red or a green cut?
- How can you define the fail/0 predicate?

Using cut, we can implement "not" (Negation by failure)

```
not(A) :- A,!,fail.
not(_).
```

not(A) is true if A cannot be shown to be true

- This is negation by failure (p124)

Negation by failure is based on the closed world assumption: (p129)

Everything that is true in the "world" is stated (or can be derived from) the clauses in the program

Negation Example

```
good_food(theWrestlers).
good_food(theCambridgeLodge).
expensive(theCambridgeLodge).

bargain(R) :- good_food(R),
              not(expensive(R)).
```

we can ask:

- bargain(R)

and Prolog replies:

- R = theWrestlers

Negation Gotcha!

```
good_food(theWrestlers).
good_food(theCambridgeLodge).
expensive(theCambridgeLodge).

bargain(R) :- not(expensive(R)),
              good_food(R).
```

Clause body terms have been swapped around!

we can ask the same query:

- bargain(R)

and Prolog replies:

- no

Why?

```
good_food(theWrestlers).
good_food(theCambridgeLodge).
expensive(theCambridgeLodge).

bargain(R) :- not(expensive(R)),
              good_food(R).
```

Prolog first tries to find an R such that expensive(R) is true.

- therefore not(expensive(R)) will fail if there are any expensive restaurants

We sometimes identify the way to use parameters of a rule

Prolog's non-logical properties can make it important whether or not an argument to a predicate is bound

% indicates a comment to the end of that line

```
% this comment in some hypothetical code is
% describing how to query myrule(+A,+B,-C,-D)
```

The convention for comments about rule parameters:

+X is a ground term

-X is a variable term

?X means it does not matter

Query "myrule" with two ground (input) terms A and B and two variable (output) terms C and D

Prolog variables and quantifiers

When R is not bound, quantifiers need attention

`expensive(R)`

- "There exists an R that is expensive".

`not(expensive(R))`

- "There does not exist an R that is expensive".
- In other words, "for all R, not expensive(R)".

Databases

Information can be stored as tuples in Prolog's internal database

```
tName(dme26,'David Eyers').
tName(awm22,'Andrew Moore').

tGrade(dme26,'IA',2.1).
tGrade(dme26,'IB',1).
tGrade(dme26,'II',1).
tGrade(awm22,'IA',2.1).
tGrade(awm22,'IB',1).
tGrade(awm22,'II',1).
```

Databases

We can now write a program to find all names:

```
qName(N) :- tName(_,N).
```

Or a program to find the full name and all grades for dme26.

```
qGrades(F,G) :- tName(C,F), tGrade(C,G).
```

Further exercises are in the problem sheet...

Prolog lecture 4

- Playing Countdown
- Iterative deepening
- Search

Countdown Numbers

Select 6 of 24 numbers tiles

- large numbers: 25,50,75,100
- small numbers: 1,2,3...10 (two of each)

Contestant chooses how many large and small

Randomly chosen 3-digit target number

Get as close as possible using each of the 6 numbers at most once and the operations of addition, subtraction, multiplication and division

- No floats or fractions allowed

Countdown Numbers

Strategy – generate and test

Maintain a list of symbolic arithmetic terms

- initially this list consists of ground terms e.g.: `[gnd(25),gnd(6),gnd(3),gnd(3),gnd(7),gnd(50)]`
- if the head of the list evaluates to the total then succeed
- otherwise pick two of the elements, combine them using one of the available arithmetic operations, put the result on the head of the list, and repeat

Countdown Numbers

Prerequisite predicates:

`eval(A,B)`

- true if the symbolic expression A evaluates to B

`choose(N,L,R,S)`

- true if R is the result of choosing N items from L and S is the remaining items left in L

`arithop(A,B,C)`

- true if C is a valid combination of A and B in the context of the game
  - e.g. arithop(A,B,plus(A,B)).

Countdown Numbers

```
%%% arith_op(+A, +B, -C)
%%% unify C with a valid binary operation
%%% of expressions A and B
arithop(A,B,plus(A,B)).
% minus is not commutative
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.
% don't allow mult by 1
arithop(A,B,mult(A,B)) :- eval(A,D), D \== 1,
                          eval(B,E), E \== 1.
% div is not commutative and don't allow div by 0 or 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.
```

Countdown Numbers

The code almost explains itself!

```
% Soln evaluates to the target number
countdown([Soln|_],Target,Soln) :-
      eval(Soln,Target).

% Combine from L to form new experiment
countdown(L,Target,Soln) :-
      choose(2,L,[A,B],R),
      arithop(A,B,C),
      countdown([C|R],Target,Soln).
```

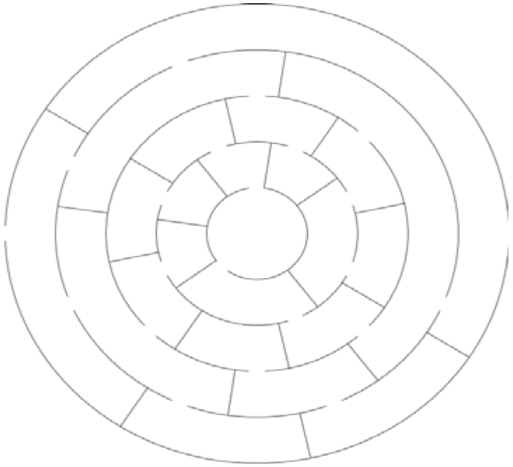
Closest Solution

No exact solutions? Find the closest solution instead.

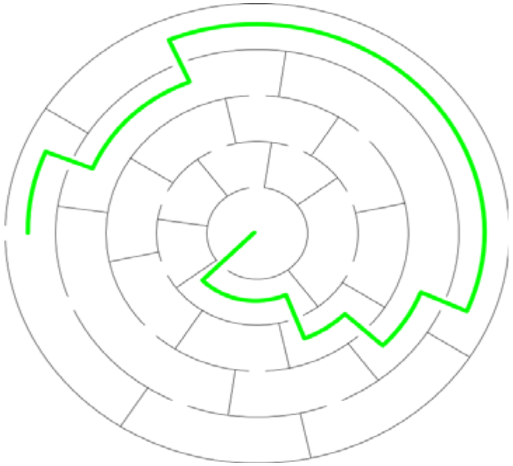
- This is iterative deepening and will be covered in your Artificial Intelligence course (p248)

```
% our result value R is D different from the Target
solve([Soln|_],Target,Soln,D) :- eval(Soln,R),
                                 diff(Target,R,D).
% recursive case is akin to the equivalent countdown/3
solve(L,Target,Soln,D) :- choose(2,L,[A,B],R),
                          arithop(A,B,C),
                          solve([C|R],Target,Soln,D).
% search for a solution decreasingly close to the target
solve(L,Target,Soln) :- range(0,100,D),
                        solve(L,Target,Soln,D).
```

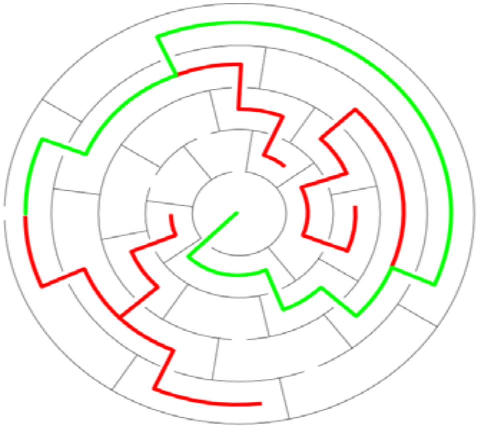
Searching



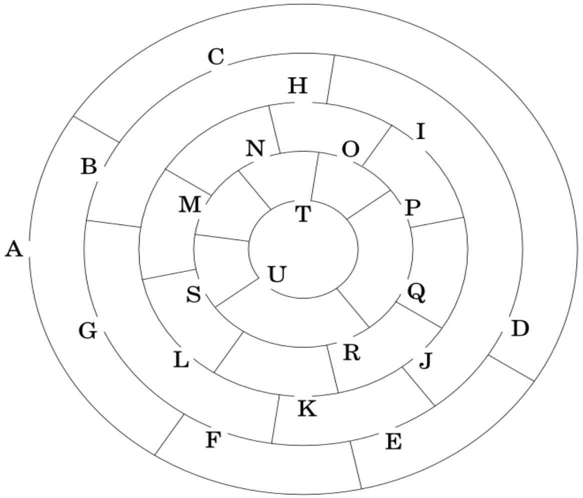
Searching: maze solution



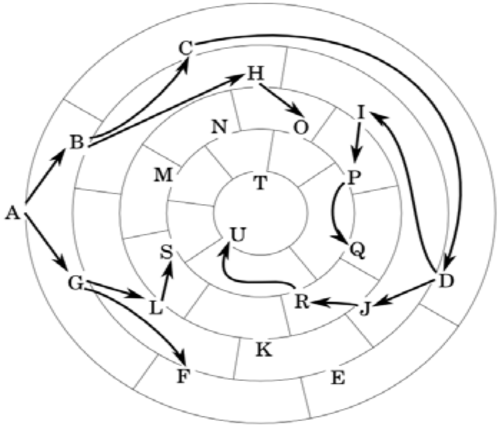
Searching: solution and failed paths



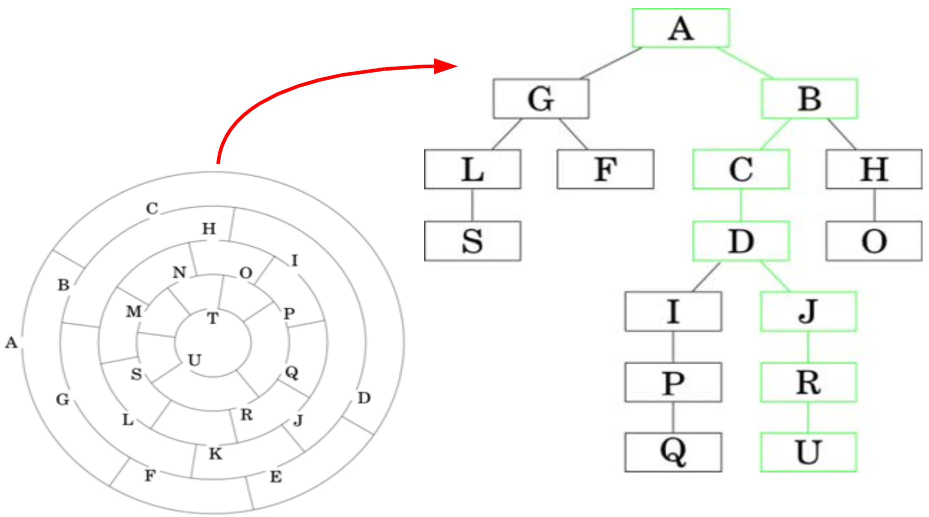
Searching: represent the problem



Searching: possible paths



Game search space as a tree



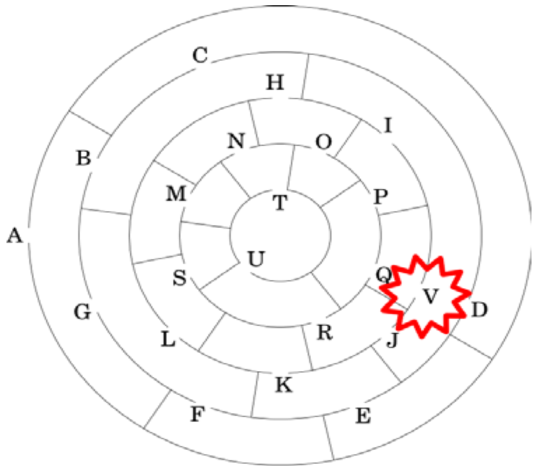
Finding a route through the maze

```
start(a).
finish(u).

route(a,g).
route(g,l).
route(l,s).
…
travel(A,A).
travel(A,C) :- route(A,B),travel(B,C).

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

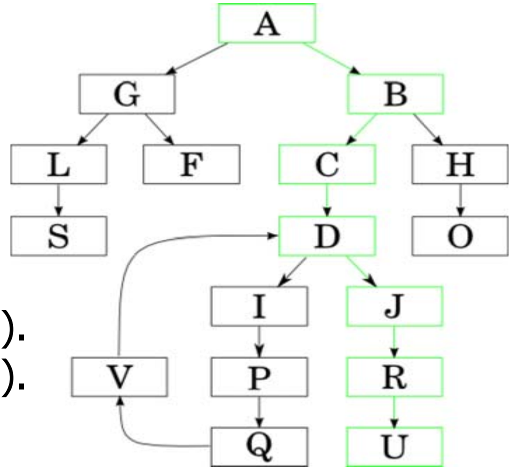
We need to remember the route

```
travellog(A,A,[]).
travellog(A,C,[A-B|Steps]) :-
    route(A,B), travellog(B,C,Steps).

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

What if we have a cyclic graph?



Cyclic Graphs



route(q,v).
route(v,d).

Searching

Solution: maintain a set of places we've already been – the closed set

- In SWI Prolog you can write \+ to mean not()

```
travelsafe(A,A,_).
travelsafe(A,C,Closed) :-
    route(A,B), \+ member(B,Closed),
    travelsafe(B,C,[B|Closed]).
```

Missionaries and Cannibals

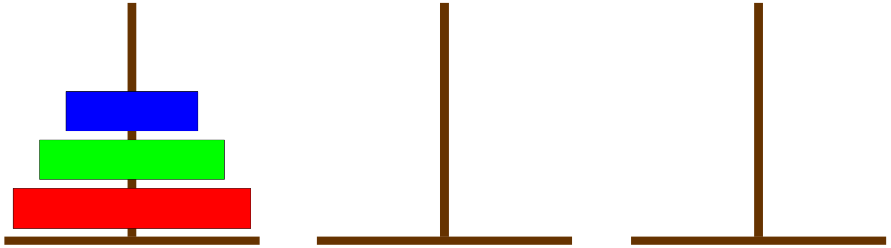
3 Missionaries

3 Cannibals

1 boat

- The boat carries two people
- If the Cannibals outnumber the Missionaries they will eat them
- Get them all from one side of the river to the other?

Towers of Hanoi



Umbrella problem

A group of 4 people, Andy, Brenda, Carl, & Dana, arrive in a car near a friend's house, who is having a large party. It is raining heavily, & the group was forced to park around the block from the house because of the lack of available parking spaces due to the large number of people at the party. The group has only 1 umbrella, & agrees to share it by having Andy, the fastest, walk with each person into the house, & then return each time. It takes Andy 1 minute to walk each way, 2 minutes for Brenda, 5 minutes for Carl, & 10 minutes for Dana. It thus appears that it will take a total of 19 minutes to get everyone into the house. However, Dana indicates that everyone can get into the house in 17 minutes by a different method. How? The individuals must use the umbrella to get to & from the house, & only 2 people can go at a time (& no funny stuff like riding on someone's back, throwing the umbrella, etc.).

Prolog lecture 5

- Data structures
- Difference lists
- Appendless append

Appending two Lists

Predicate definition is elegantly simple:

```
append([],L,L).
append([X|T],L,[X|R]) :- append(T,L,R).
```

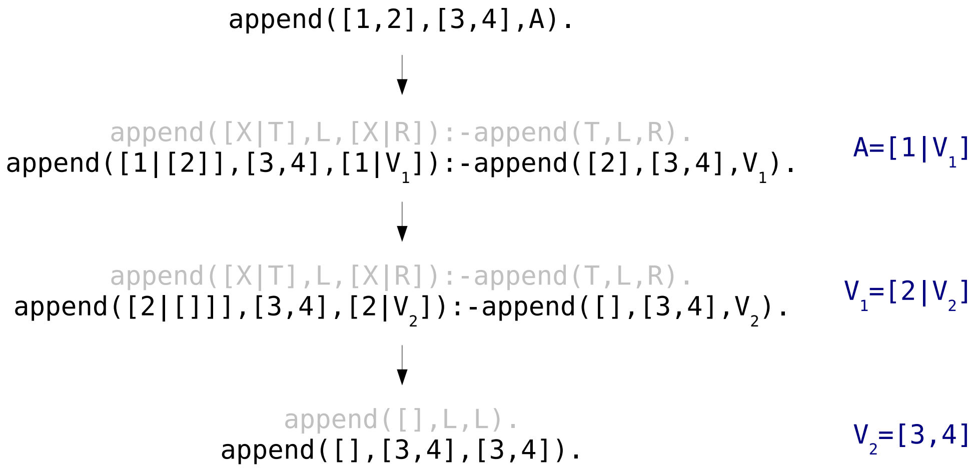
Run-time performance is not good though

- Procedural languages would not scan a list to append

Want to modify the end of the list directly

- Prolog can achieve this

```
append([],L,L).
append([X|T],L,[X|R]) :- append(T,L,R).
```



append([1,2],[3,4],A).

append([X|T],L,[X|R]):-append(T,L,R).
append([1|[2]],[3,4],[1|V_1]):-append([2],[3,4],V_1).

A=[1|V_1]

append([X|T],L,[X|R]):-append(T,L,R).
append([2|[]]],[3,4],[2|V_2]):-append([],[3,4],V_2).

V_1=[2|V_2]

append([],L,L).
append([],[3,4],[3,4]).

V_2=[3,4]

Difference Lists (p185)

Instead of storing one list, store two

- Represent our original list as the difference between these other two lists

We might represent the “normal” list [1,2,3] as

- [1,2,3,4,5]-[4,5] or
- [1,2,3,acr]-[acr] or
- [1,2,3|X]-X

It is the last form here that is key!

Difference List Append

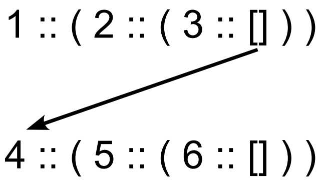
Append one list to another...

1 :: (2 :: (3 :: []))

4 :: (5 :: (6 :: []))

Difference List Append

... in a single list-linking step



1 :: (2 :: (3 :: []))

4 :: (5 :: (6 :: []))

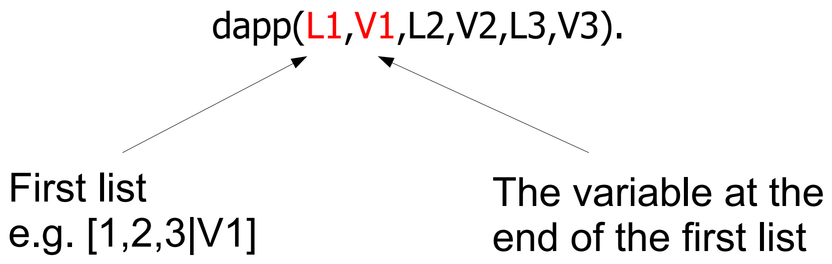
Difference List Append

$$1 :: (2 :: (3 :: A))$$

$$\underbrace{4 :: (5 :: (6 :: B))}_{A}$$

Prolog syntax for the first list is [1,2,3|A]

Difference List Append



dapp(L1,V1,L2,V2,L3,V3).

First list
e.g. [1,2,3|V1]

The variable at the
end of the first list

Difference List Append

By convention we write our difference list pair as

A-B

But we could also write:

differenceList(A,B)

A+B

A*B, etc

dapp(L1-V1,L2-V2,L3-V3)

- Append difference list L2-V2 to L1-V1 and unify the result with L3-V3.

Difference List Append (implementation)

L1 $l1_0::l1_1:: \cdots ::l1_n::V1$

L2 $l2_0::l2_1:: \cdots ::l2_n::V2$

L3 $l1_0:: \cdots ::l1_n::l2_0:: \cdots ::l2_n::V2$

```
dapp(L1-V1,L2-V2,L3-V3) :- V1=L2,
                           L3=L1,
                           V3=V2.
```

Difference List Append (implementation)

L1 $\boxed{l1_0::l1_1:: \cdots ::l1_n}::V1$

L2 $l2_0::l2_1:: \cdots ::l2_n::V2$

L3 $l1_0:: \cdot$

```
dapp(L1

                                       V3=V2.
```

This is the value of the list we want to represent and so our difference list has to be $[l1_0,l1_1,l1_2...|V1]-V1$

Difference List Append (implementation)

L1 $l1_0::l1_1:: \cdots ::l1_n::V1$

L2 $l2_0::l2_1:: \cdots ::l2_n::V2$

L3 $l1_0:: \cdots ::l1_n::l2_0:: \cdots ::l2_n::V2$

```
dapp(L1-V1,L2-V2,L3-V3) :- V1=L2,
                           L3=L1,
                           V3=V2.
```

Difference List Append (implementation)

```
dapp(L1-V1,L2-V2,L3-V3) :- V1=L2,
                           L3=L1,
                           V3=V2.
```

We know that V1 and L2 must be the same:

- replace all instances of V1 and L2 with new variable B
- (we can remove the B=B of course)

```
dapp(L1-B,B-V2,L3-V3) :- B=B,
                         L3=L1,
                         V3=V2.
```

Difference List Append (implementation)

So we have:

```
dapp(L1-B,B-V2,L3-V3) :- L3=L1,
                         V3=V2.
```

But we know that L3 and L1 must be the same

- Replace them with a new variable A:

```
dapp(A-B,B-V2,A-V3) :- A=A,
                       V3=V2.
```

Difference List Append (final implementation)

Now we have:

```
dapp(A-B,B-V2,A-V3) :- V3=V2.
```

But we know that V3 and V2 must be the same

- Substituting a new variable C:

```
dapp(A-B,B-C,A-C) :- C=C.
```

But that simplifies to the following final answer

- Gradual substitution like this is a useful technique

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

Representing empty Difference Lists

An empty difference list is an empty list with a variable at its end ready for later binding.

- Let us call this variable A
- We've seen lists like `[1,2|A]`

If you understand the `[⋯|L]` syntax you will appreciate that removing `1,2` leaves (simply):

```
A
```

We write this in the conventional notation as:

```
A-A
```

Another Difference List Example

Define a procedure `rotate(X,Y)` where both X and Y are represented by difference lists, and Y is formed by rotating X to the left by one element.

[14 marks]

1996-6-7

(This is the second example in your handout)

Determine an answer first that does not use Difference Lists

Take the first element off the first list (H) and append it after the tail (i.e. at the end) in the solution (R)

```
rotate([H|T],R) :- append(T,[H],R).
```

Rewrite with Difference Lists

Allocate “tail variables” to our original lists

- Give list [H|T] tail variable T1
- Give list R tail variable S

```
rotate([H|T],R) :- append(T,[H],R).
```

becomes:

```
rotate([H|T]-T1,R-S) :-
                dapp(T-T1,[H|L]-L,R-S).
```

Why is this term not `[H|T1]-T1` ?

Rename variables to incorporate difference list append

Recall: difference list append just shuffles vars

```
rotate([H|T]-T1,R-S) :-
            dapp(T-T1,[H|L]-L,R-S).
```

Rename T1 to be [H|L]: unify with B in dapp/3

```
rotate([H|T]-[H|L],R-S) :-
         dapp(T-[H|L],[H|L]-L,R-S).
```

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

Rename variables to incorporate difference list append

From the previous slide:

```
rotate([H|T]-[H|L],R-S) :-
         dapp(T-[H|L],[H|L]-L,R-S).
```

Rename R to be T: thus unifying with A in dapp/3

```
rotate([H|T]-[H|L],T-S) :-
         dapp(T-[H|L],[H|L]-L,T-S).
```

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

Rename variables to incorporate difference list append

From the previous slide:

```
rotate([H|T]-[H|L],T-S) :-
         dapp(T-[H|L],[H|L]-L,T-S).
```

Rename S to be L: thus unifying with C in dapp/3

```
rotate([H|T]-[H|L],T-L) :-
         dapp(T-[H|L],[H|L]-L,T-L).
```

dapp call is now redundant and can be removed!

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

Final Answer

```
rotate([H|T]-[H|A],T-A).
```

Beautifully concise... but also somewhat opaque!

It is recommended that you comment any line of Prolog like this really, really thoroughly!

Converting to difference lists

```
double([],[]).
double([H|T],[R|S]) :-
    R is H*2,
    double(T,S).
```

... add in the tail variables ...

```
double(A-A,B-B).
double([H|T]-T1,[R|S]-S1) :-
    R is H*2,
    double(T-T1,S-S1).
```

Question

What does `double([1,2,3|T]-T,R)` produce?

a) yes, R = [2,4,6|X]-X
b) no
c) yes, R = X-X
d) an exception

Towers of Hanoi Revisited

Move n rings from Src to Dest

- move n-1 rings from Src to Aux
- move the nth ring from Src to Dest
- move n-1 rings from Aux to Dest

Base case: move 0 rings from Src to Dest

End

- Next lecture: solving Sudoku,
- constraint logic programming
- and where to go next...

Prolog lecture 6

- Solving Sudoku puzzles
- Constraint Logic Programming
- Natural Language Processing

Playing Sudoku

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

Make the problem easier

		4	
	2		
		1	
	3		

We can model this problem in Prolog using list permutations

Each row must be a permutation of [1,2,3,4]

Each column must be a permutation of [1,2,3,4]

Each 2x2 box must be a permutation of [1,2,3,4]

Represent the board as a list of lists

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

```
[[A,B,C,D],
 [E,F,G,H],
 [I,J,K,L],
 [M,N,O,P]]
```

The sudoku predicate is built from simultaneous perm constraints

```
sudoku( [[X11,X12,X13,X14],[X21,X22,X23,X24],
         [X31,X32,X33,X34],[X41,X42,X43,X44]]) :-
     %rows
     perm([X11,X12,X13,X14],[1,2,3,4]),
     perm([X21,X22,X23,X24],[1,2,3,4]),
     perm([X31,X32,X33,X34],[1,2,3,4]),
     perm([X41,X42,X43,X44],[1,2,3,4]),
     %cols
     perm([X11,X21,X31,X41],[1,2,3,4]),
     perm([X12,X22,X32,X42],[1,2,3,4]),
     perm([X13,X23,X33,X43],[1,2,3,4]),
     perm([X14,X24,X34,X44],[1,2,3,4]),
     %boxes
     perm([X11,X12,X21,X22],[1,2,3,4]),
     perm([X13,X14,X23,X24],[1,2,3,4]),
     perm([X31,X32,X41,X42],[1,2,3,4]),
     perm([X33,X34,X43,X44],[1,2,3,4]).
```

Scale up in the obvious way to 3x3

X11	X12	X13	X14	X15	X16	X17	X18	X19
X21	X22	X23	X24	X25	X26	X27	X28	X29
X31	X32	X33	X34	X35	X36	X37	X38	X39
X41	X42	X43	X44	X45	X46	X47	X48	X49
X51	X52	X53	X54	X55	X56	X57	X58	X59
X61	X62	X63	X64	X65	X66	X67	X68	X69
X71	X72	X73	X74	X75	X76	X77	X78	X79
X81	X82	X83	X84	X85	X86	X87	X88	X89
X91	X92	X93	X94	X95	X96	X97	X98	X99

Brute-force is impractically slow

There are very many valid grids:

6670903752021072936960 ≈ 6.671×10^{21}

Our current approach does not encode the interrelationships between the constraints

For more information on Sudoku enumeration:

http://www.afjarvis.staff.shef.ac.uk/sudoku/

Prolog programs can be viewed as constraint satisfaction problems

Prolog is limited to the single equality constraint:

- two terms must unify

We can generalise this to include other types of constraint

Doing so leads to Constraint Logic Programming

- and a means to solve Sudoku problems (p319)

Consider variables taking values from domains with constraints

Given:

- the set of variables
- the domains of each variable
- constraints on these variables

We want to find:

- an assignment of values to variables satisfying the constraints

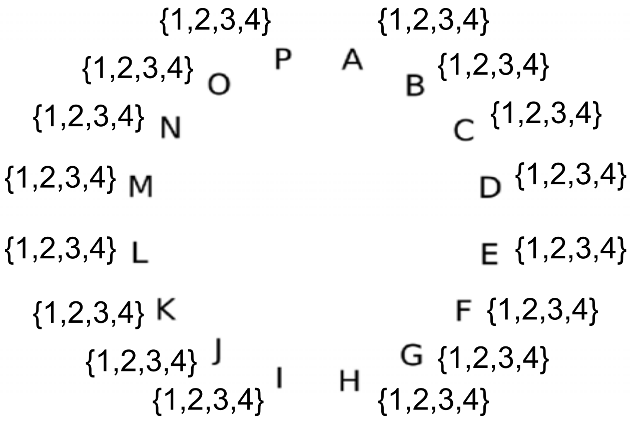
Sudoku can be expressed as constraints

First, we express the variables and domains

A ∈ {1,2,3,4}   B ∈ {1,2,3,4}
C ∈ {1,2,3,4}   D ∈ {1,2,3,4}
E ∈ {1,2,3,4}   F ∈ {1,2,3,4}
G ∈ {1,2,3,4}   H ∈ {1,2,3,4}
I ∈ {1,2,3,4}   J ∈ {1,2,3,4}
K ∈ {1,2,3,4}   L ∈ {1,2,3,4}
M ∈ {1,2,3,4}   N ∈ {1,2,3,4}
O ∈ {1,2,3,4}   P ∈ {1,2,3,4}

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

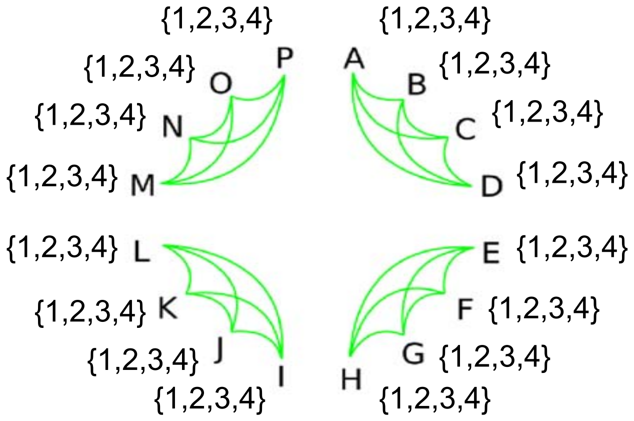
Express Sudoku as a Constraint Graph



A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

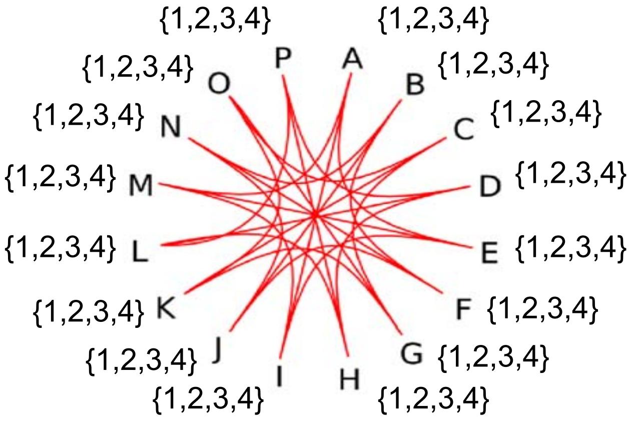
Constraints: All variables in rows are different

Edges represent inequalities between variables



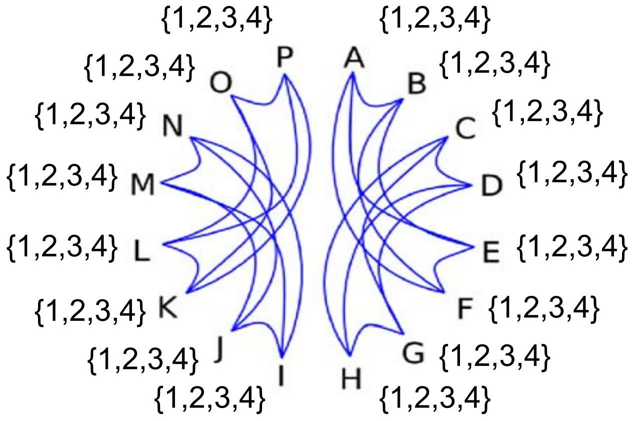
A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

Constraints: All variables in columns are different



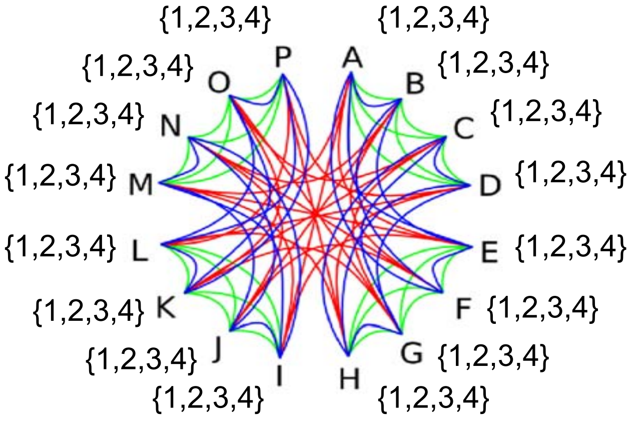
A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

Constraints: All variables in boxes are different



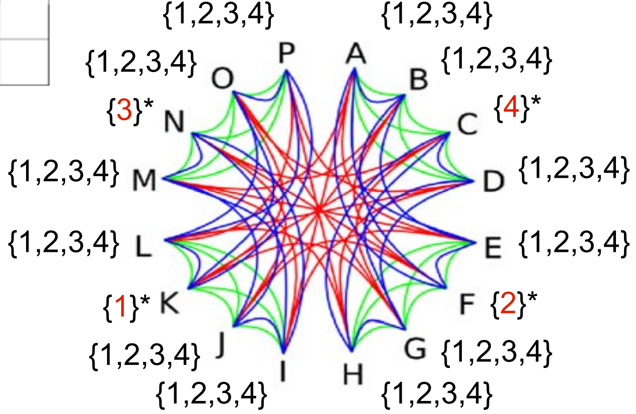
A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

All constraints shown together



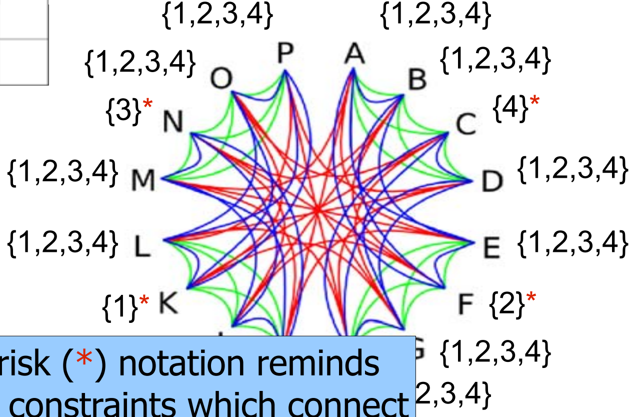
Reduce domains according to initial values

		4	
	2		
		1	
	3		



When a domain changes we update its constraints

		4	
	2		
		1	
	3		

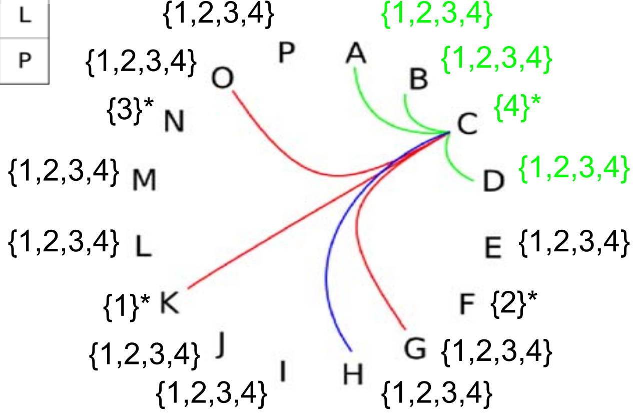


The asterisk (*) notation reminds us that all constraints which connect to this variable need updating

Update constraints connected to C

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

We will remove 4 from the domain of A, B and D

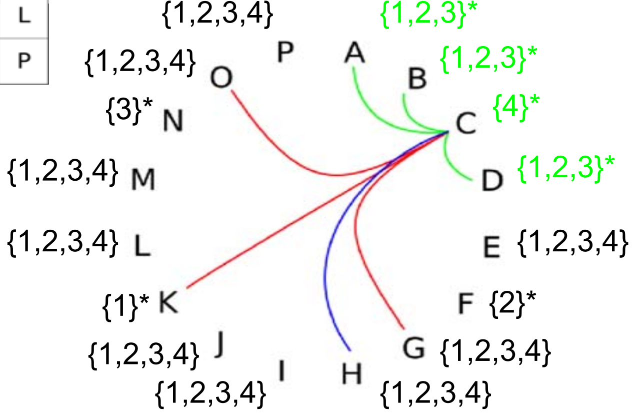


Update constraints connected to C

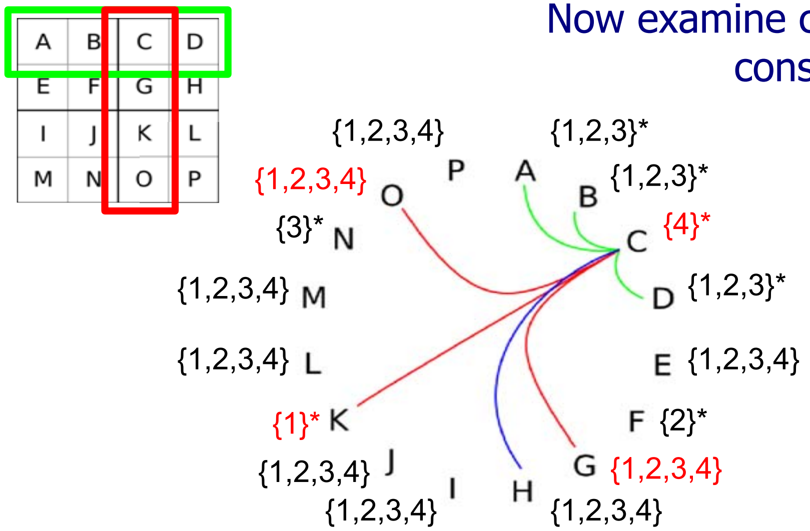
A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

We add asterisks to A, B and D

...but will defer looking at them

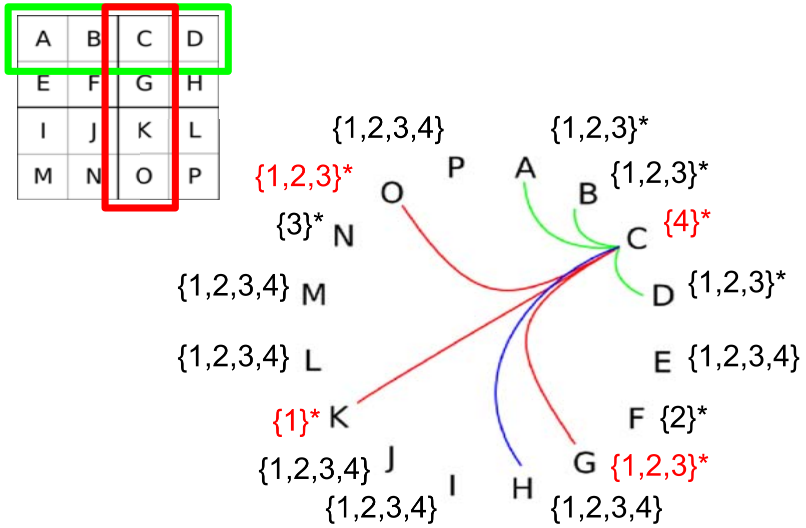


Update constraints connected to C

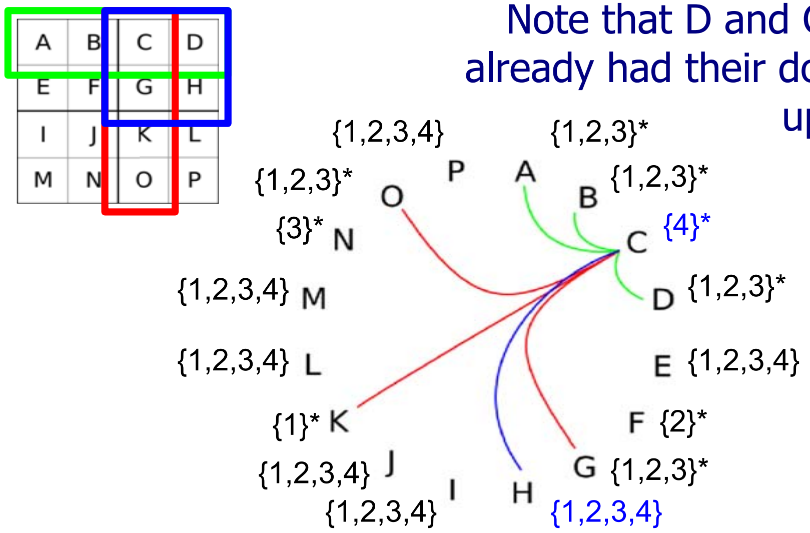


Now examine column constraints

Update constraints connected to C

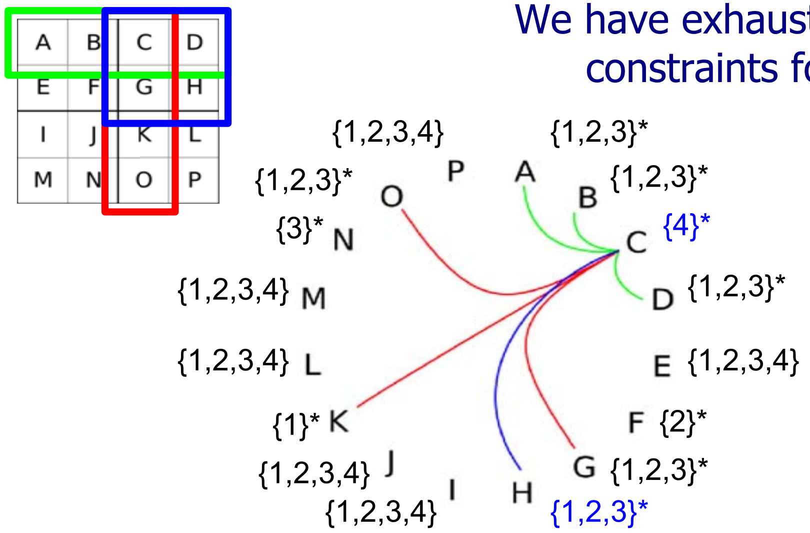


Update constraints connected to C



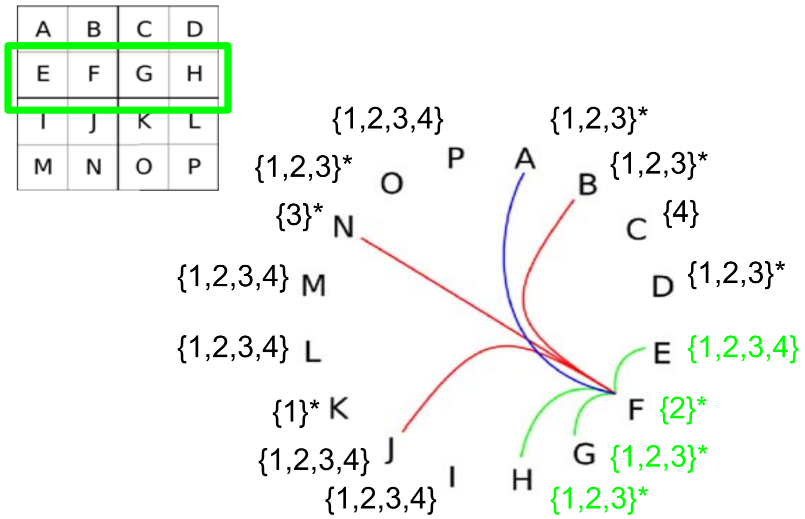
Note that D and G have already had their domains updated

Update constraints connected to C

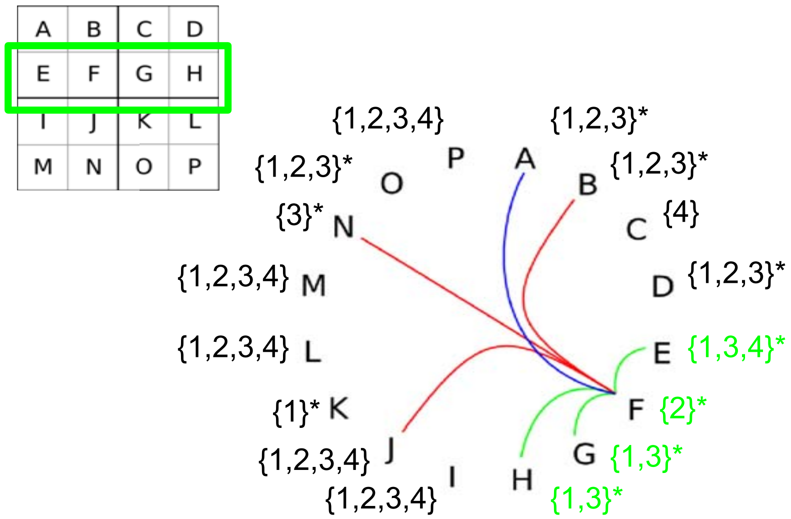


We have exhausted C's constraints for now

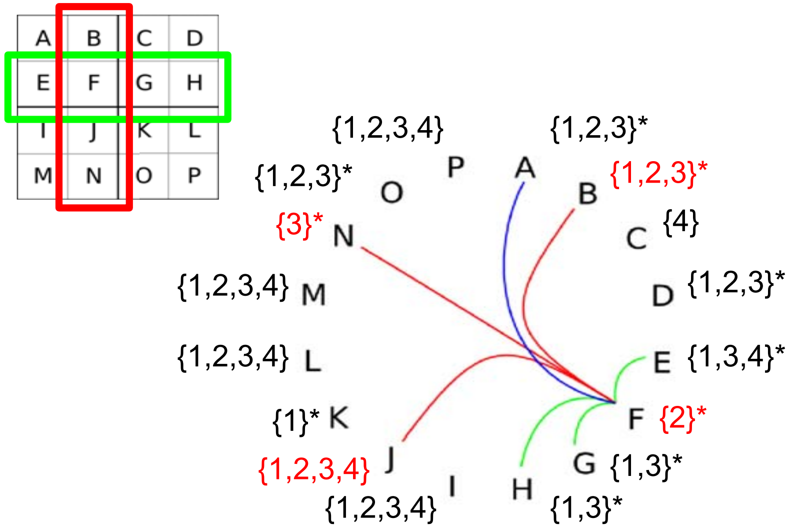
Update constraints connected to F



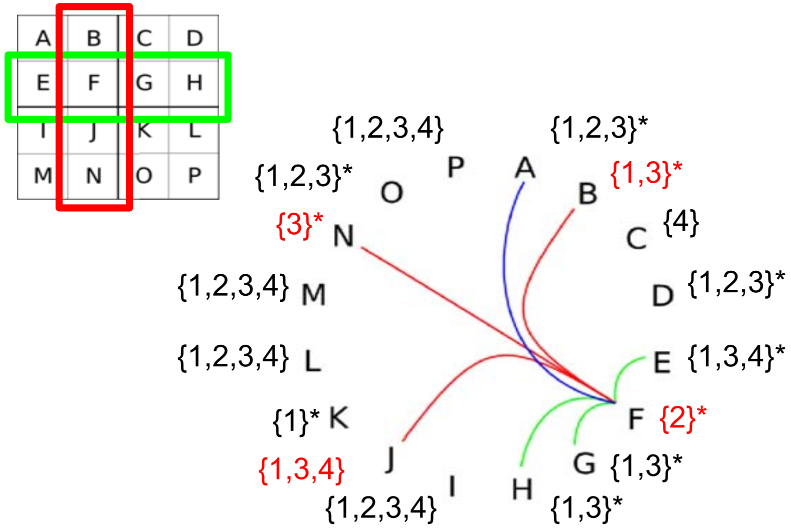
Update constraints connected to F



Update constraints connected to F

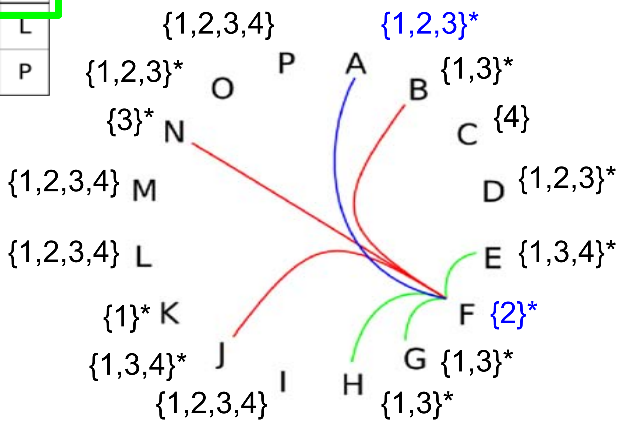


Update constraints connected to F



Update constraints connected to F

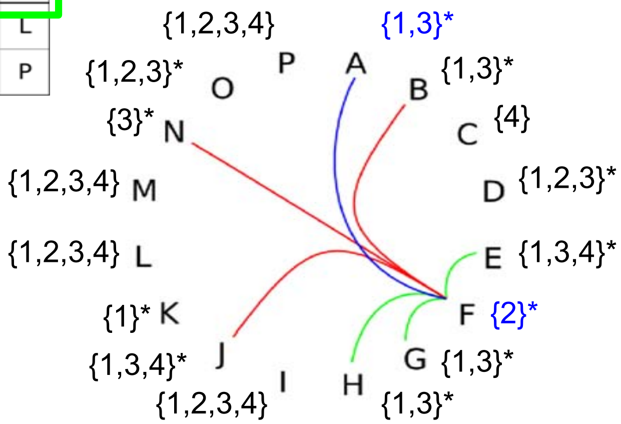
A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P



Update constraints connected to F

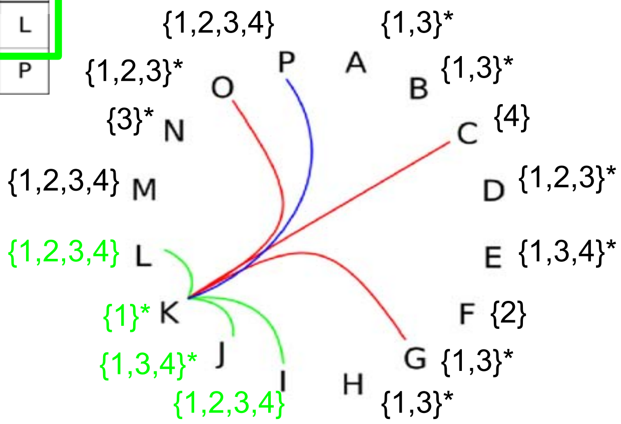
We have exhausted F's constraints for now

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P



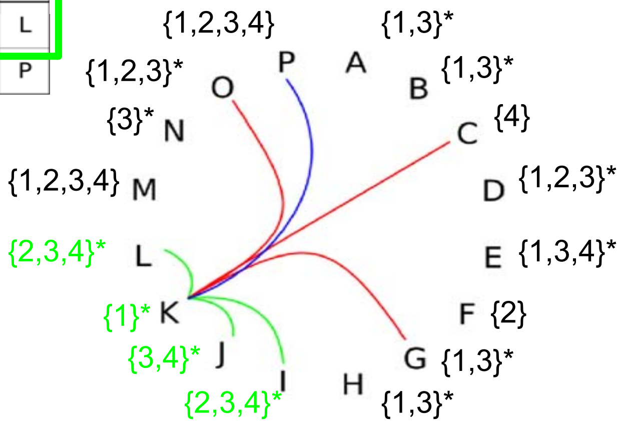
Update constraints connected to K

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

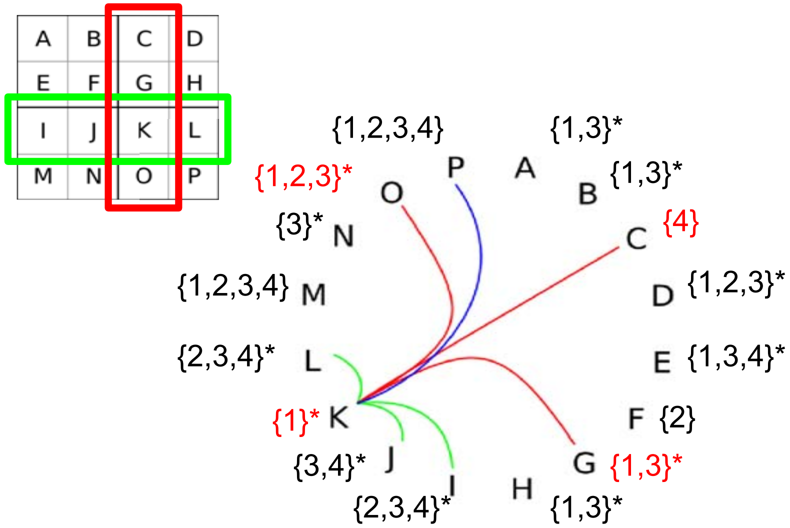


Update constraints connected to K

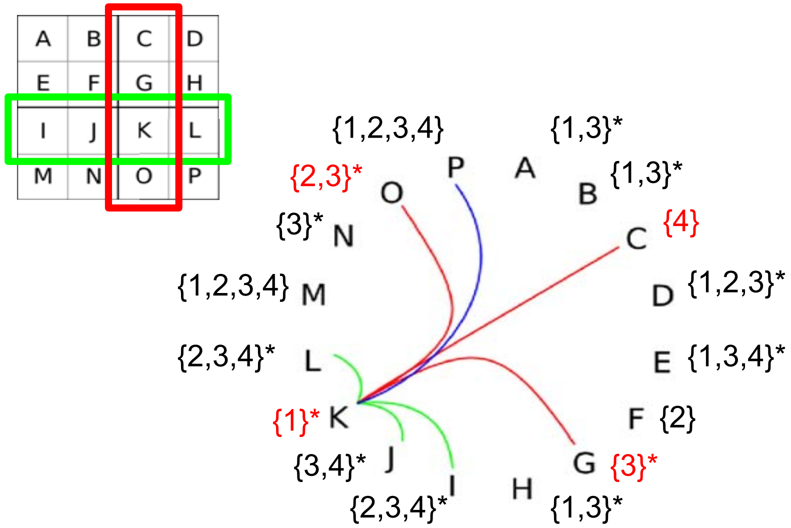
A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P



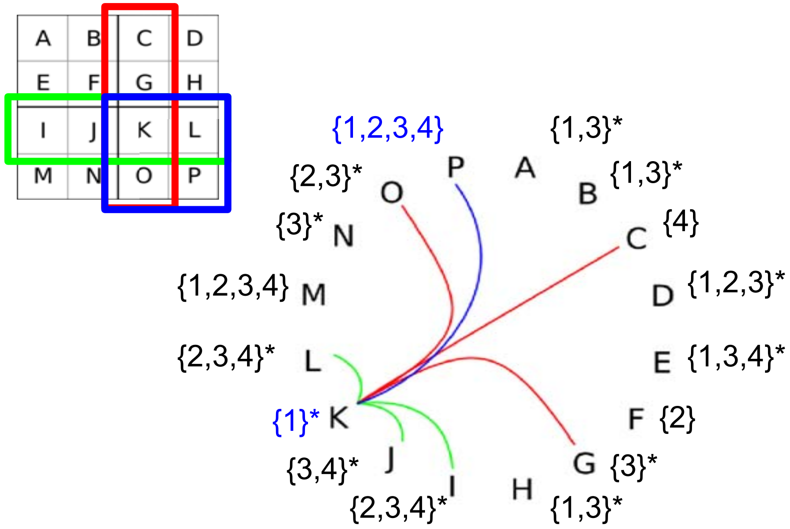
Update constraints connected to K



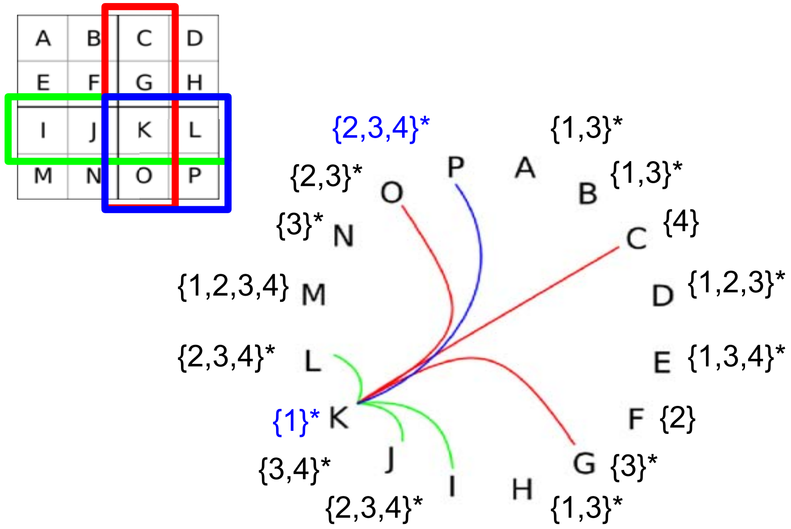
Update constraints connected to K



Update constraints connected to K

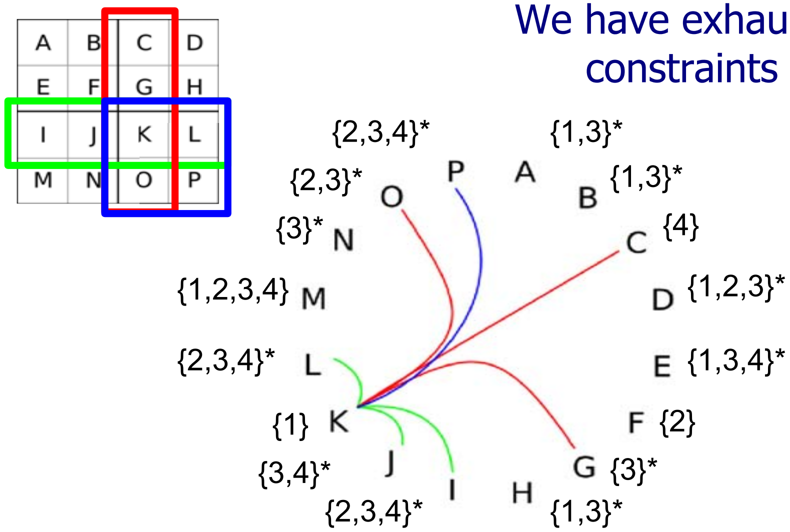


Update constraints connected to K



Update constraints connected to K

We have exhausted K's constraints for now



A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

{2,3,4}* P

{1,3}* A

{2,3}* O

B {1,3}*

{3}* N

C {4}

{1,2,3,4} M

D {1,2,3}*

{2,3,4}* L

E {1,3,4}*

{1} K

F {2}

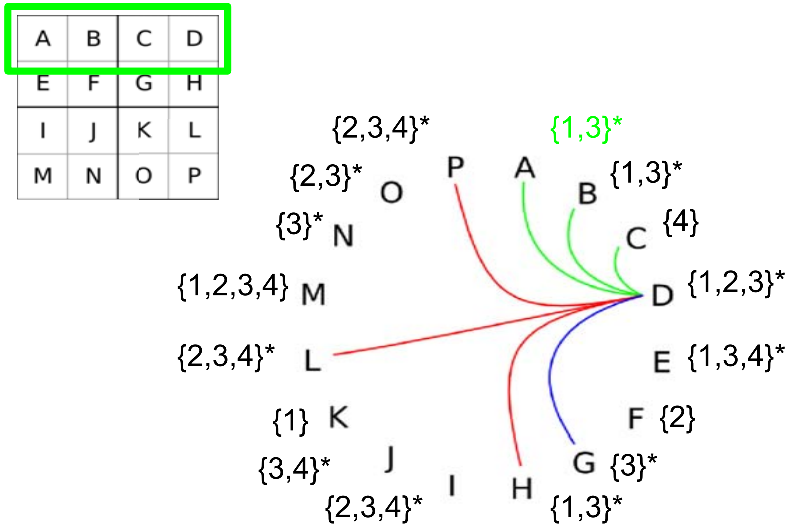
{3,4}* J

G {3}*

{2,3,4}* I

H {1,3}*

Update constraints connected to D



A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

{2,3,4}* P

{1,3}* A

{2,3}* O

B {1,3}*

{3}* N

C {4}

{1,2,3,4} M

D {1,2,3}*

{2,3,4}* L

E {1,3,4}*

{1} K

F {2}

{3,4}* J

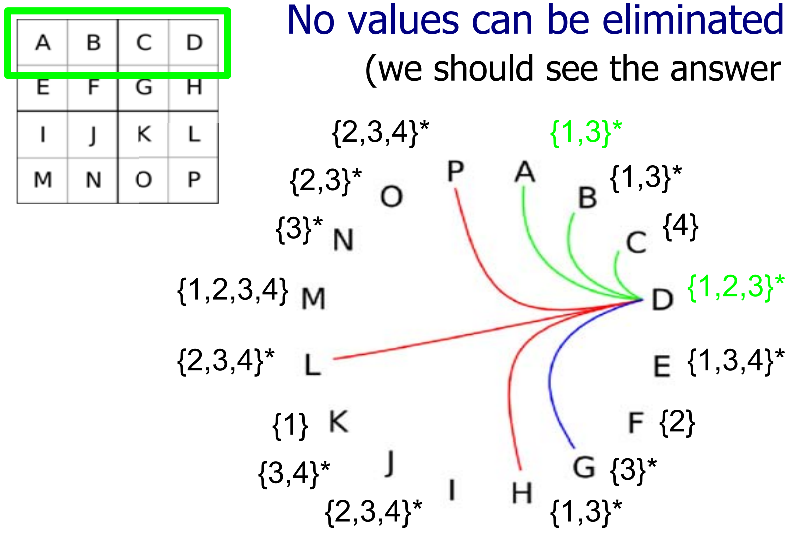
G {3}*

{2,3,4}* I

H {1,3}*

Update constraints connected to D

No values can be eliminated directly
(we should see the answer though!)



A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

{2,3,4}* P

{1,3}* A

{2,3}* O

B {1,3}*

{3}* N

C {4}

{1,2,3,4} M

D {1,2,3}*

{2,3,4}* L

E {1,3,4}*

{1} K

F {2}

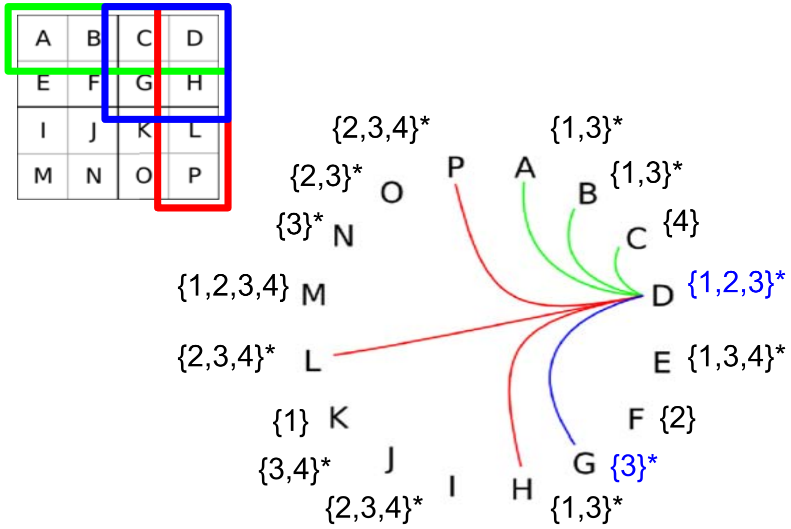
{3,4}* J

G {3}*

{2,3,4}* I

H {1,3}*

Update constraints connected to D



A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

{2,3,4}* P

{1,3}* A

{2,3}* O

B {1,3}*

{3}* N

C {4}

{1,2,3,4} M

D {1,2,3}*

{2,3,4}* L

E {1,3,4}*

{1} K

F {2}

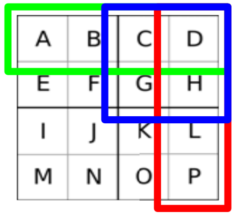
{3,4}* J

G {3}*

{2,3,4}* I

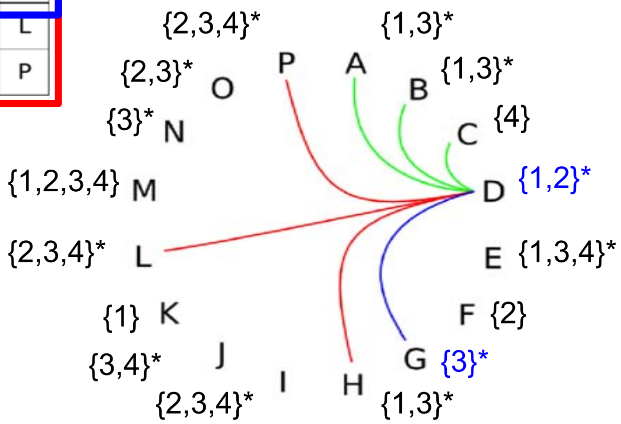
H {1,3}*

Change can occur in source domain



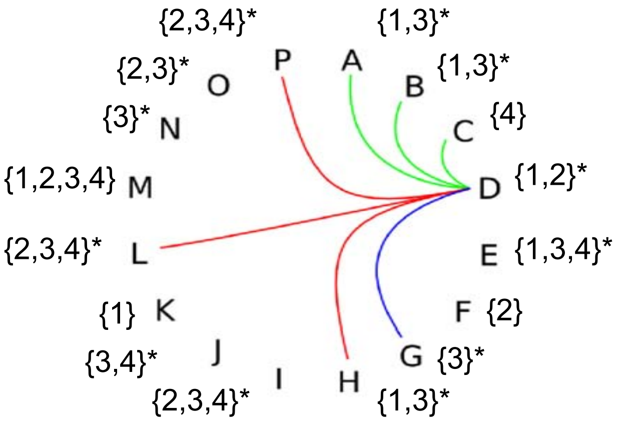
A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

Single 3 at G eliminates D's 3



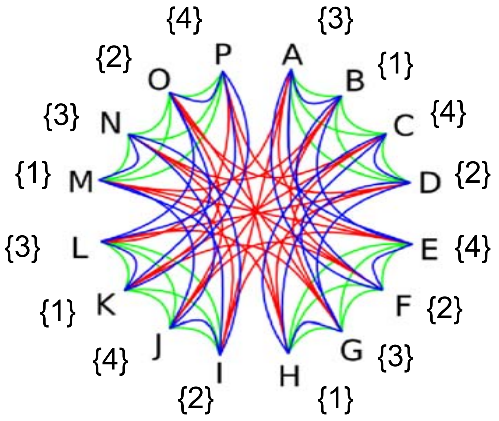
Change can occur in source domain

If the source domain changes we mark all its constraints for update again



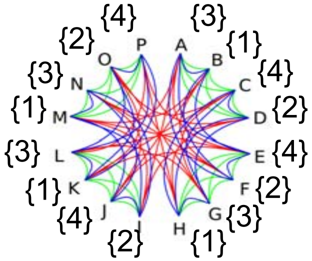
Iterate the algorithm to convergence (no further changes occur)

Why will the algorithm eventually converge?



Outcome 1: Single valued domains

We have found a unique solution to the problem



3	1	4	2
4	2	3	1
2	4	1	3
1	3	2	4

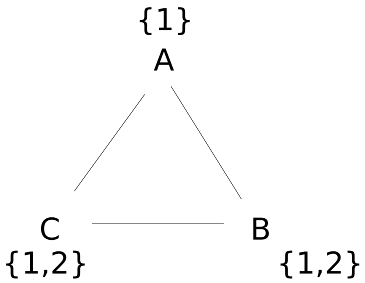
Outcome 2: Some empty domains

Our constraints are shown to be inconsistent

- therefore there is no solution to this problem

Variables
- A ∈ {1}
- B ∈ {1,2}
- C ∈ {1,2}

Constraints
- A ≠B, A ≠C, B ≠C



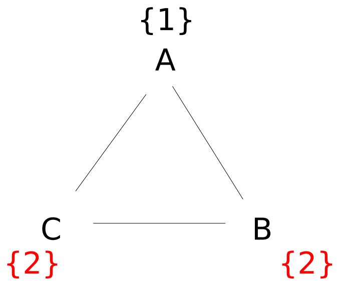
Outcome 2: Some empty domains

Our constraints are shown to be inconsistent

- therefore there is no solution to this problem

Variables
- A ∈ {1}
- B ∈ {1,2}
- C ∈ {1,2}

Constraints
- A ≠B, A ≠C, B ≠C



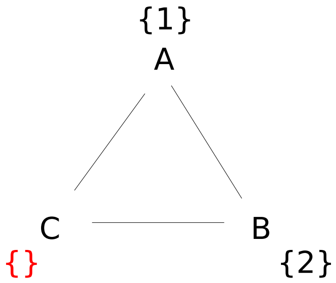
Outcome 2: Some empty domains

Our constraints are shown to be inconsistent

- therefore there is no solution to this problem

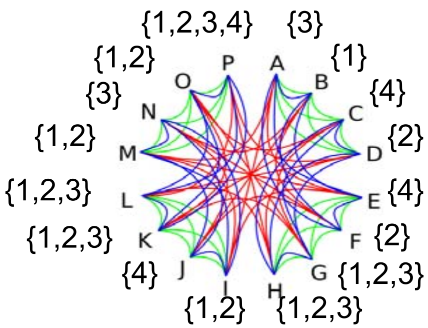
Variables
- A ∈ {1}
- B ∈ {1,2}
- C ∈ {1,2}

Constraints
- A ≠B, A ≠C, B ≠C



Outcome 3: Some multivalued domains

		4	
	2		
	3		



Not all combinations of these variable assignment possibilities are global solutions though...

Outcome 3: Hypothesise labellings

To find global solutions from the narrowed domains we hypothesise a solution in a domain and propagate the changes

Backtrack if something goes wrong

Using CLP in Prolog

```
:- use_module(library(bounds)).

valid4(L) :- L in 1..4, all_different(L).

sudoku2([[X11,X12,X13,X14],[X21,X22,X23,X24],
         [X31,X32,X33,X34],[X41,X42,X43,X44]]) :-
  valid4([X11,X12,X13,X14]),valid4([X21,X22,X23,X24]), Rows
  valid4([X31,X32,X33,X34]),valid4([X41,X42,X43,X44]),
  valid4([X11,X21,X31,X41]),valid4([X12,X22,X32,X42]), Cols
  valid4([X13,X23,X33,X43]),valid4([X14,X24,X34,X44]),
  valid4([X11,X12,X21,X22]),valid4([X13,X14,X23,X24]), Boxes
  valid4([X31,X32,X41,X42]),valid4([X33,X34,X43,X44]),
  labeling([],[X11,X12,X13,X14,X21,X22,X23,X24,
               X31,X32,X33,X34,X41,X42,X43,X44]).
```

Chapter 14 of the textbook has more information

Prolog can be used for parsing context-free grammars (p555)

Here is a simple grammar:

s → 'a' 'b'
s → 'a' 'c'
s → s s

Terminal symbols:

a, b

Non-terminal symbols:

s

Parsing by consumption

Write a predicate for each non-terminal that:

- consumes as much of the first list as is necessary to match the non-terminal, and
- returns the remaining elements in the second list

These predicate evaluations will thus be true:

- s([a,b],[])
- s([a,b,c,d],[c,d])

A Prolog program that accepts sentences from our grammar

s → 'a' 'b'
s → 'a' 'c'
s → s s

```
% match a single character
c([X|T],X,T).

% grammar predicates
s(In,Out) :- c(In,a,In2),
             c(In2,b,Out).
s(In,Out) :- c(In,a,In2),
             c(In2,c,Out).
s(In,Out) :- s(In,In2),
             s(In2,Out).
```

Prolog DCG syntax

s → 'a' 'b'
s → 'a' 'c'
s → s s

Prolog provides us with a shortcut for encoding Definite Clause Grammar (DCG) syntax.

```
s --> [a],[b].
s --> [a],[c].
s --> s,s.
```

This will both test and generate:

- s([a,c,a,b],[]).
- s(A,[]).

Building a parse tree

```
% match a single character
c([X|T],X,T).

% grammar predicates
s(ab,In,Out) :- c(In,a,In2),
                c(In2,b,Out).
s(ac,In,Out) :- c(In,a,In2),
                c(In2,c,Out).
s(t(A,B),In,Out) :- s(A,In,In2),
                    s(B,In2,Out).

:- s(Result,[a,c,a,b,a,b],[]).
```

Prolog's DCG syntax helps us again

Unfortunately the DCG syntax is not part of the ISO Prolog standard

- Almost all modern compilers will include it though

```
s(ab) --> [a],[b].
s(ac) --> [a],[c].
s(t(A,B)) --> s(A),s(B).
```

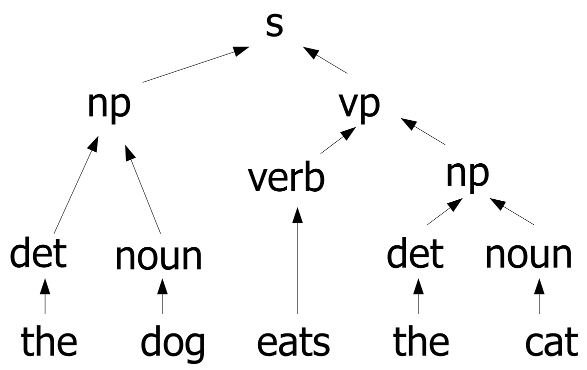
Parsing Natural Language (back to Prolog's roots)

```
s   --> np,vp.
np  --> det,n.
vp  --> v.
vp  --> v,np.

n   --> [cat].
n   --> [dog].
v   --> [eats].
det --> [the].
```

This is a very limited English grammar subset.

Things get complicated very quickly!

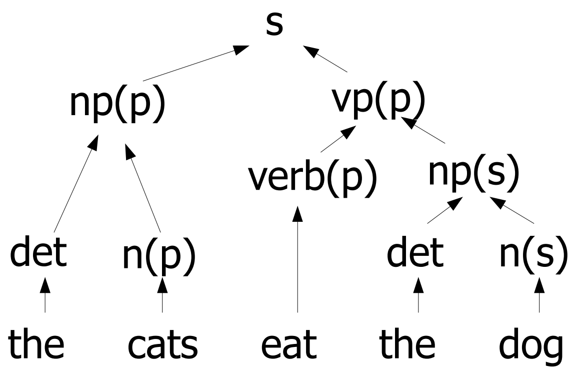
- see the Natural Language Processing course next year (Prolog is not a pre-requisite)



We can also handle agreement

```
s(N)  --> np(N),vp(N).
np(N) --> det,n(N).
vp(N) --> v(N).
vp(N) --> v(N),np(_).

n(s)  --> [cat].
n(s)  --> [dog].
n(p)  --> [cats].
v(s)  --> [eats].
v(p)  --> [eat].
det   --> [the].
```

We consider only third-person constructions here!



Real Natural Language Processing

Things get much more complicated very quickly

Ambiguities, special cases and noise all make the approach we have demonstrated hard to scale

- Although people have definitely tried!

Closing Remarks

Declarative programming is different to Functional or Procedural programming

- Foundations of Computer Science & Programming in Java

Prolog is built on logical deduction

- formal explanation in Logic & Proof

It can provide concise implementations of algorithms such as sorting or graph search

- Algorithms I & Algorithms II

Closing Remarks

Foundations of Functional Programming (Part IB)

- Building computation from first principles

Databases (Part 1B)

- Find out more about representing data and SQL

Artificial Intelligence (Part 1B)

- Search, constraint programming and more

C & C++ (Part 1B)

- Doing useful stuff in the real world

Natural Language Processing (Part II)

- Parsing natural language

End

C

You shoot yourself in the foot.

C++

You accidentally create a dozen instances of yourself and shoot them all in the foot. Providing emergency medical care is impossible since you can't tell which are bitwise copies and which are just pointing at others and saying, "That's me over there."

Prolog

You explain in your program that you want to be shot in the foot. The interpreter figures out all the possible ways to do it, but then backtracks completely, instead destroying the gun.

Prolog supervision work

Michaelmas 2008
David Eyers <David.Eyers@cl.cam.ac.uk>
Original author Dr Andrew Rice

1 Introduction

These questions form the suggested supervision work for the Prolog course. All students should attempt the basic questions. Ideally, students should ensure that they have access to a Prolog enviornment, and test their work within it. Those questions marked with an asterisk are more difficult although all students should be able to answer them with the help of their supervisor. Questions marked with a double asterisk are particularly challenging and are beyond the level required for producing a good answer in the exam.

Prolog contains a number of features and facilities not covered in the lectures such as: assert, findall and retract. Students should limit themselves to using only the features covered in the lecture course and are not expected to know about anything further. All questions can be successfully answered using only the lectured features.

Students are encouraged to contact me by email with bug-reports and solutions to double-asterisk questions.

2 Lecture 1

2.1 Unification

1. Unify these two terms by hand:
   - tree(tree(tree(1,2),A,B),tree(C,tree(E,F,G)))
   - tree(C,tree(Z,C))
2. Explain Prolog's behaviour when you unify a(A) with A.
3. Relate unification with ML type inference

2.2 The Zebra Puzzle

1. Implement and test the Zebra Puzzle solution
2. Explain how Clue 1 has been expressed in the Zebra Puzzle query

3 Lecture 2

3.1 Encoding arithmetic in Prolog

The **is** operator in Prolog evaluates arithmetic expressions. This builtin functionality can also be modelled within Prolog's logical framework.

Let the atom i represent the identity (1) and the compound term s(A) represent the successor of A. For example 4 = s(s(s(i)))

Implement and test the following rules:

1. prim(A,B) which is true if A is a number and B is its primitive representation
2. plus(A,B,C) which is true if C is A+B (all with primitive representations, A and B are both ground terms)
3. mult(A,B,C) which is true if C is A*B (all with primitive representations, A and B are both ground terms)

The development of arithmetic (and general computation) from first principles is considered more formally in the Foundations of Functional Programming course.

3.2 List Operations

1. Explain the operation of the append/3 clause

   ```
   append([],A,A).
   append([H|T],A,[H|R]) :- append(T,A,R).
   ```

2. Draw the Prolog search tree for perm([1,2,3],A).
3. Implement a clause choose(N,L,R,S) that chooses N items from L and puts them in R with the remaining elements in L left in S
4. * What it the purpose of the following clauses:

   ```
   a([H|T]) :- a([H|T],H).
   a([],_).
   a([H|T],Prev) :- H >= Prev, a(T,H).
   ```

5. * What does the following do and how does it work?:

   ```
   b(X,X) :- a(X).
   b(X,Y) :- append(A,[H1,H2|B],X), H1 > H2, append(A,[H2,H1|B],X1),
             b(X1,Y).
   ```

3.3 Generate and Test

The description of these problems will be given in the lecture.

1. Complete the Dutch National Flag solution
2. * Complete the 8-Queens solution
3. * Generalise 8-Queens to n-Queens
4. Complete the Anagram generator. In what situations is it more efficient to Test-and-Generate rather than Generate-and-Test?

4 Lecture 3

4.1 Symbolic Evaluation

1. Explain what happens when you put the clauses of the symbolic evaluator in a different order
2. Add additional clauses to the symbolic evaluator for subtraction and integer division (this is the // operator in Prolog i.e. 2 is 6//3)

4.2 Negation

State and explain Prolog's response to the following queries:

1. X=1.
2. not(X=1).
3. not(not(X=1)).
4. not(not(not(X=1))).

In those cases where Prolog says 'yes' your answer should include the unified result for X.

4.3 Databases

We can use facts entered into Prolog as a general database for storing and querying information. This question considers the construction of a database containing information about students, their colleges and their grades in the various parts of the CS Tripos.

Each fact in our Prolog database corresponds to a row in a table of data. A table is constructed from rows produced by facts with the same name. The initial database of facts is as follows:

```
tName(dme26,'David Eyers').
tName(awm22,'Andrew Moore').

tCollege(dme26, 'King''s').
tCollege(awm22, 'Corpus Christi').

tGrade(dme26,'IA',2.1).
tGrade(dme26,'IB',1).
tGrade(dme26,'II',1).
tGrade(awm22,'IA',2.1).
tGrade(awm22,'IB',1).
tGrade(awm22,'II',1).
```

As an example, this database contains a table called 'tName' which contains two rows of two columns. The first column is the CRSID of the individual and the second column is their full name.

4.3.1 Part 1

1. Add your own details to the database.
2. Add a new table tDOB that contains CRSID and DOB.
3. Alter the database such that for some users their college is not present (this final step is necessary for testing your answers to the questions in Part 2)

4.3.2 Part 2

The next task is to provide rules and show queries that implement various queries of the database. You should answer each question with the Prolog facts and rules required to implement the query and also an example invocation of those rules.

For example:

```
% The full name of each person in the database
qFullName(A) :- tName(_,A).

% Example invocation
% qFullName(A).
```

Each query should return one row of the answer at a time, subsequent rows should be returned by backtracking.

For the example above:

```
?- qFullName(A).

A = 'David Eyers' ;

A = 'Andrew Moore'

Yes
```

- The descriptions that follow provide a plain English description of the query that you should implement, followed by the same query in SQL.
- SQL (Structured Query Language) is the industry standard language currently used to query relational databases—you will see more on this in the Databases course.
- The '?' notation in the SQL statements derives from the use of prepared statements in relational databases where (for efficiency) a single statement is sent to the database server and repeatedly evaluated with different values replacing the '?'. Interested students can consult the Java PreparedStatement documentation.

1. Full name and College attended.
   SELECT name,college FROM tName, tCollege WHERE tName.crsid = tCollege.crsid
2. Full name and College attended only including entries where the user can choose a single CRSID to include in the results.
   SELECT name,college FROM tName, tCollege WHERE tName.crsid = tCollege.crsid AND tName.crsid = ?
3. Full name and College attended or blank if the college is unknown.
   SELECT name,college FROM tName LEFT OUTER JOIN tCollege ON tName.crsid = tCollege.crsid
4. Full name and College attended. The full name or the college should be blank if either is unknown.
   SELECT name,college FROM tName FULL OUTER JOIN tDOB ON
   tName.crsid = tCollege.crsid
5. * Find the lowest grade where the CRSID is specified by the user. Note that this predicate should only return one result even when backtracking.
   SELECT min(grade) FROM tGrade WHERE crsid = ?

6. ** Find the number of people with a First class mark
   SELECT count(grade) FROM tGrade WHERE grade = 1

7. ** Find the number of First class marks awarded to each person. Your output should consist of a tuple (CRSID,NumFirsts) that iterates through all CRSIDs which have at least one First class mark upon backtracking
   SELECT crsid,count(grade) FROM tGrade WHERE grade=1 GROUP BY crsid
   Hint: This is not the number of rows with First class marks in the tGrade table. You will need build a list of First class CRSIDs by repeatedly querying tGrade and checking if the result is already in your list. Every time you find a new unique CRSID, increment an accumulator which will form the result.

5 Lecture 4

5.1 Countdown Numbers Game

1. Type in the example code which finds exact solutions from the lectures and test it;

2. Implement the predicate range(Min,Max,Val) which unifies Val with Min on the first evaluation, and then all values up to Max-1 when backtracking;

3. Get the iterative deepening version of the numbers game working.

5.2 Graph searching

Implement search-based solutions for:

1. Missionaries and Cannibals: there are three missionaries, three cannibals and who need to cross a river. They have one boat which can hold at most two people. If, at any point, the cannibals outnumber the missionaries then they will eat them. Discover the procedure for a safe crossing.

2. * Towers of Hanoi: you have N rings of increasing size and three pegs. Initially the three rings are stacked in order of decreasing size on the first peg. You can move them between pegs but you must never stack a big ring onto a smaller one. What is the sequence of moves to move from all the rings from the first to the the third peg.

3. * Umbrella: A group of 4 people, Andy, Brenda, Carl, & Dana, arrive in a car near a friend's house, who is having a large party. It is raining heavily, & the group was forced to park around the block from the house because of the lack of available parking spaces due to the large number of people at the party. The group has only 1 umbrella, & agrees to share it by having Andy, the fastest, walk with each person into the house, & then return each time. It takes Andy 1 minute to walk each way, 2 minutes for Brenda, 5 minutes for Carl, & 10 minutes for Dana. It thus appears that it will take a total of 19 minutes to get everyone into the house. However, Dana indicates that everyone can get into the house in 17 minutes by a different method. How? The individuals must use the umbrella to get to & from the house, & only 2 people can go at a time (& no funny stuff like riding on someone's back, throwing the umbrella, etc.). (This puzzle included with kind permission from http://www.puzz.com/)

6 Lecture 5

6.1 Sorting

Implement the following sorting algorithms in Prolog:

1. Finding the minimum element from the list and recursively sorting the remainder.
2. Quicksort.
3. * Quicksort where the partitioning step divides the list into three groups: those items less than the pivot, those items equal to the pivot and those items greater than the pivot. Explain in what situations this additional complexity might be desirable.
4. * Quicksort with the append removed using difference lists. Note: you do not need to alter the partitioning clauses.
5. * Mergesort

6.2 Towers of Hanoi

The Towers of Hanoi problem can be solved without requiring an inefficient graph search. Discover the algorithm required to do this from the lecture notes or the web and implement it. Once you have a simple list-based implementation rewrite it to use difference lists.

6.3 Dutch Flag

Earlier in the course we solved the dutch flag problem using generate and test. A more efficient approach is to make one pass through the initial list collecting three separate lists (one for each colour). When you reach the end of the initial list you append the three separate collection lists together and you are done. Implement this algorithm using normal lists and then rewrite it to use difference lists.

7 Lecture 6

7.1 Sudoku Solver

Extend the CLP-based 2x2 Sudoku solver given in the lecture to a 3x3 grid and test it.

7.2 Cryptarithmetic

Here is a classic example of a cryptarithmetic puzzle:

$$\begin{array}{cccccc} & & S & E & N & D \\ + & & M & O & R & E \\ \hline & M & O & N & E & Y \end{array}$$

The problem is to find an assignment of the numbers 0–9 (inclusive) to the letters S,E,N,D,M,O,R,E,Y such that the arithmetic expression holds and the numeric value assigned to each letter is unique.

We can formulate this problem in CLP as follows:

```
solve1([S,E,N,D],[M,O,R,E],[M,O,N,E,Y]) :-
  Var = [S,E,N,D,M,O,R,Y],
  Var in 0..9, all_different(Var),
            1000*S + 100*E + 10*N + D +
            1000*M + 100*O + 10*R + E #=
  10000*M + 1000*O + 100*N + 10*E + Y,
  labeling([],Var).
```

1. Get the example above working in your Prolog interpreter. How many unique solutions are there?
2. A further requirement of these types of puzzle is that the leading digit of each number in the equation is not zero. Extend your program to incorporate this. The CLP operator for arithmetic not-equals is `#\=`. How many unique solutions remain now?
3. * Extend your program to work in an arbitrary base (rather than base 10), the domain of your variables should change to reflect this. How many solutions of the puzzle above are there in base 16?
4. * Consult the Prolog documentation regarding the findall predicate. Use findall within a predicate count(Base,N) that unifies N with the number of solutions in base Base.
5. * Use the range/3 predicate from Lecture 4 to extend your count predicate to try all values from 1 to 50 as the base.
6. * Plot a graph of the number of solutions against the chosen base.

7.3 **Findall

The findall predicate is an extra-logical predicate for backtracking and collecting the results into a list. The implementation within Prolog is something along the lines of:

```
findall(Template,Goal,Solutions) :- call(Goal),
                                    assertz(findallsol(Template)),
                                    fail.
findall(Template,Goal,Solutions) :- collect(Solutions).

collect([Template|RestSols]) :- retract(findallsol(Template)),
                                !,
                                collect(RestSols).
collect([]).
```

1. ** Consult the Prolog documentation and work out what the above is doing. The predicate assertz adds a new clause to the end of the running Prolog program (i.e. the Prolog database) and the predicate retract removes a clause that unifies with its argument.
2. ** Develop an alternative to findall that does not use extra-logical predicates such as assertz and retract. This alternative will necessarily take the form of a pattern which you can apply to your clause to find all the possible results. The algorithm for the alternative findall proceeds as follows: maintain a list of solutions found so far, evaluate the target clause and repeatedly backtrack through it until it returns a value not in your list of results, add the result to the list and repeat.
3. ** Comment on the runtime complexity of the alternate findall compared to the builtin version.