

Prolog Lecture 2

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

Rules have a head that 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”

```
rule(X,Y) :- part1(X), part2(X,Y).
```

head body

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 \rightarrow head
- It's not as clean as a graphical arrow ...
- In practice Prolog is not as clean as logic either!

Note that quantifiers (\forall and \exists) are not explicitly expressed in Prolog

- (Also, in logic we could have multiple head terms, ...)

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.

- Recursion is a key Prolog programming technique

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

- Examples: `[Head|Tail]` or `[1|T]` or `[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  
% (NB: = should really be ==)  
:- 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.

(a good way to find out is to try it, obviously!)

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!

(Prolog's core is very small in terms of semantics)

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

? - $A = \text{money} + \text{power}.$

$A = \text{money} + \text{power}$

Yes

? - $\text{money} + \text{power} = A,$
 $A = +(\text{money}, \text{power}).$

$A = \text{money} + \text{power}$

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 $\text{len}([1,2],A)$.
- Apply $\text{len}([1 | [2]],A_0) :- \text{len}([2],M_0), A_0 \text{ is } M_0+1$

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

Stack Frame 1

Stack Frame 2

- Result $\text{len}([1,2],2)$
- 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 $\text{len2}([1,2],0,R)$
- Apply $\text{len2}([1| [2]],0,R) :- \text{AccNext is } 0+1, \text{len2}([2],\text{AccNext},R).$
 - Evaluate $\text{AccNext is } 0+1$ so $\text{AccNext} = 1$
 - Evaluate $\text{len2}([2],1,R)$
- Apply $\text{len2}([2| []],1,R) :- \text{AccNext is } 1+1, \text{len2}([],\text{AccNext},R).$
 - Evaluate $\text{AccNext is } 1+1$ so $\text{AccNext} = 2$
 - Evaluate $\text{len2}([],2,R).$
- Apply $\text{len2}([],2,2)$ so $R = 2$
- I didn't need to use any subscripts on variable instances!

Stack Frame 1

Stack Frame 2

Last Call Optimisation transforms 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

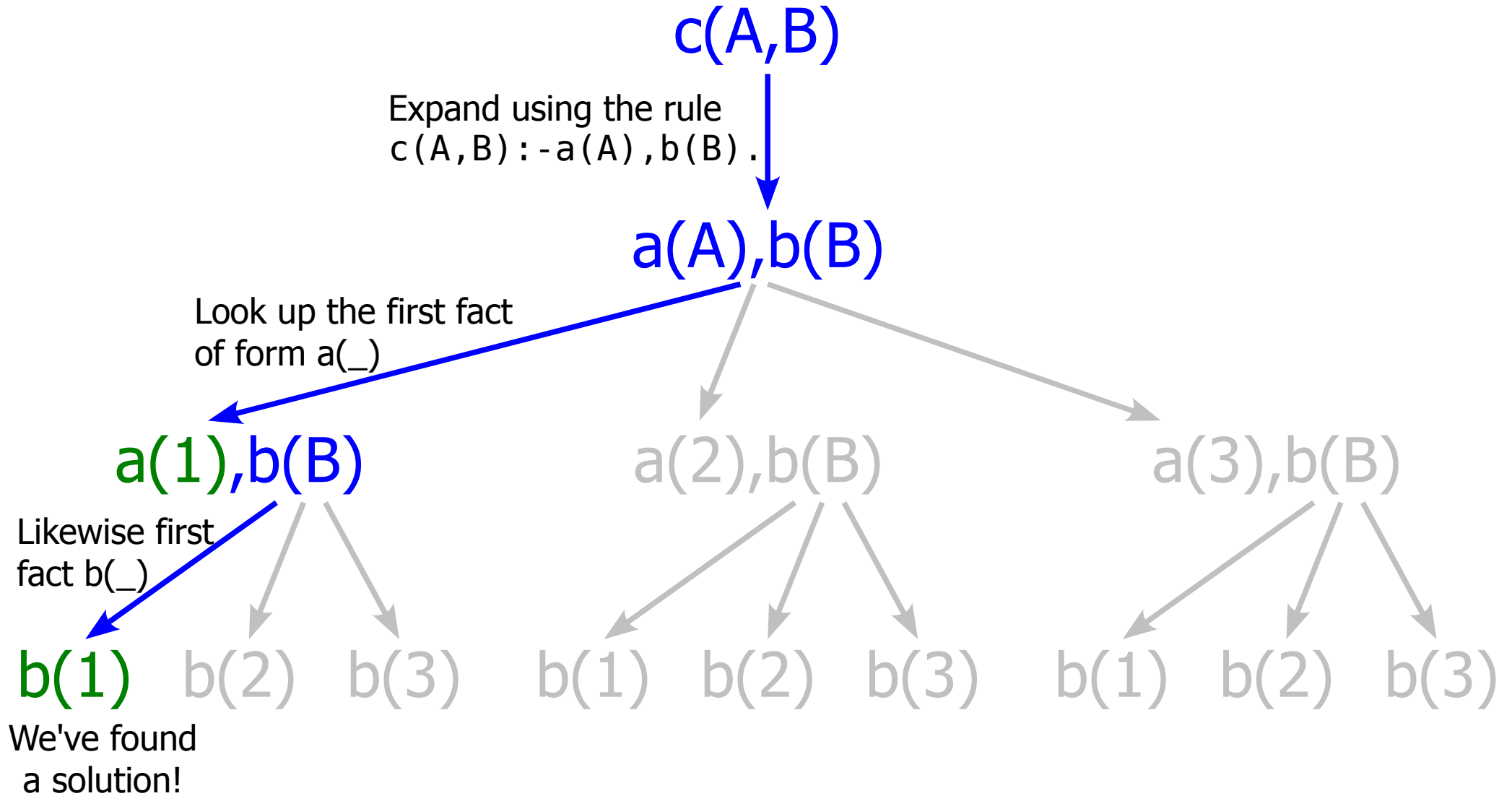
Here is a (boring) program:

```
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?

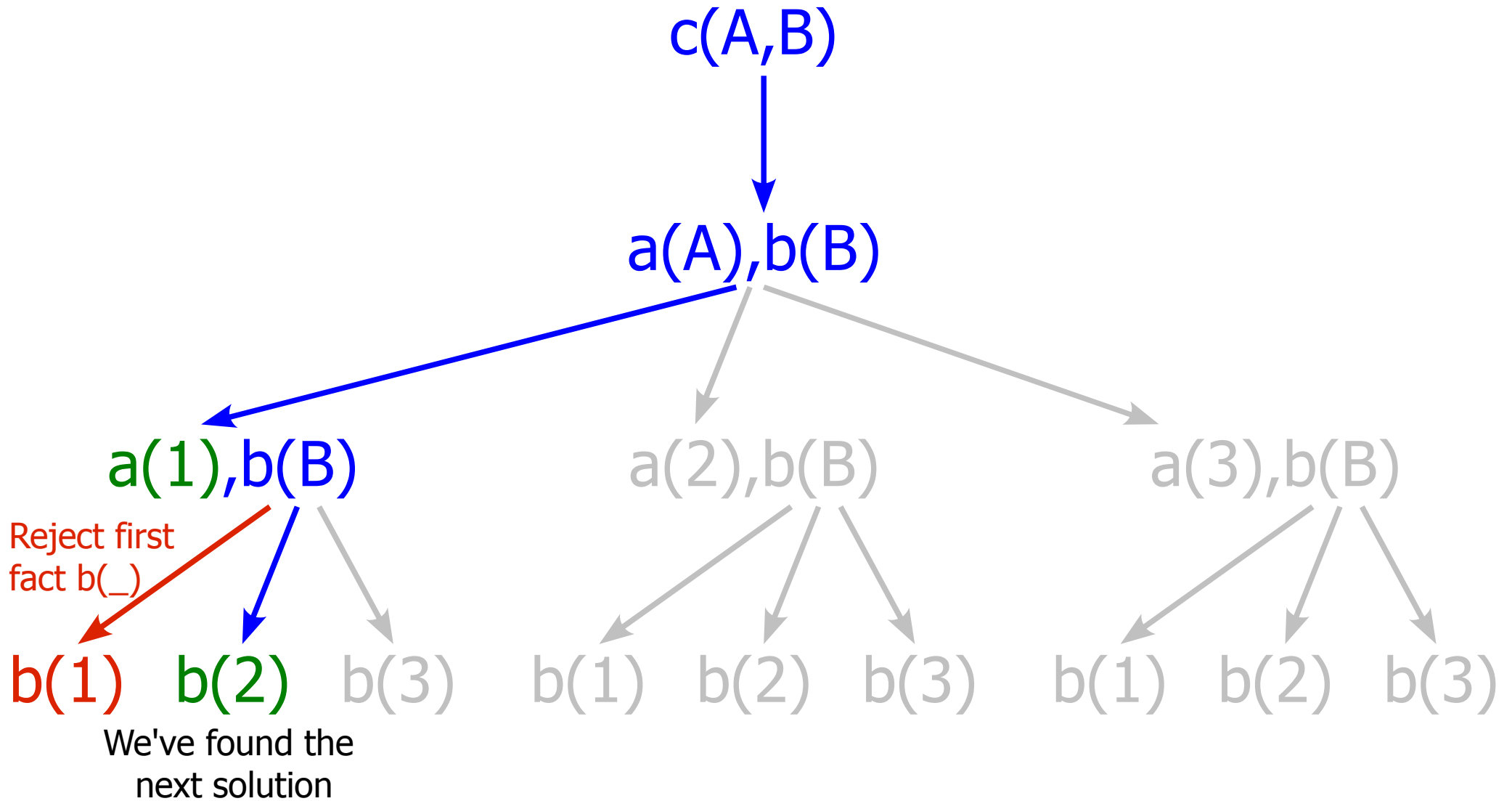
`c(A,B).`

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



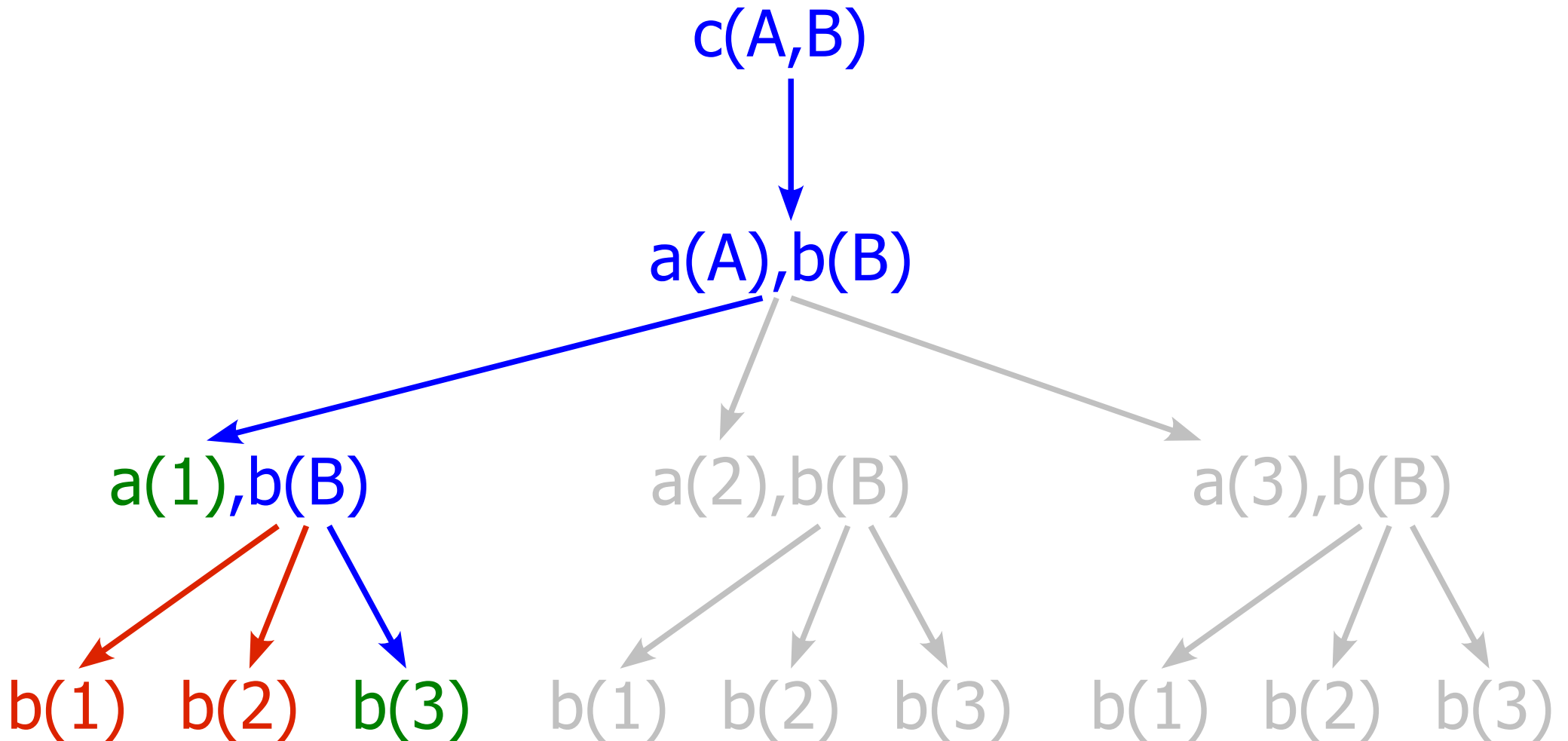
Variable bindings: $A=1, B=1$

Backtrack to find the next solution



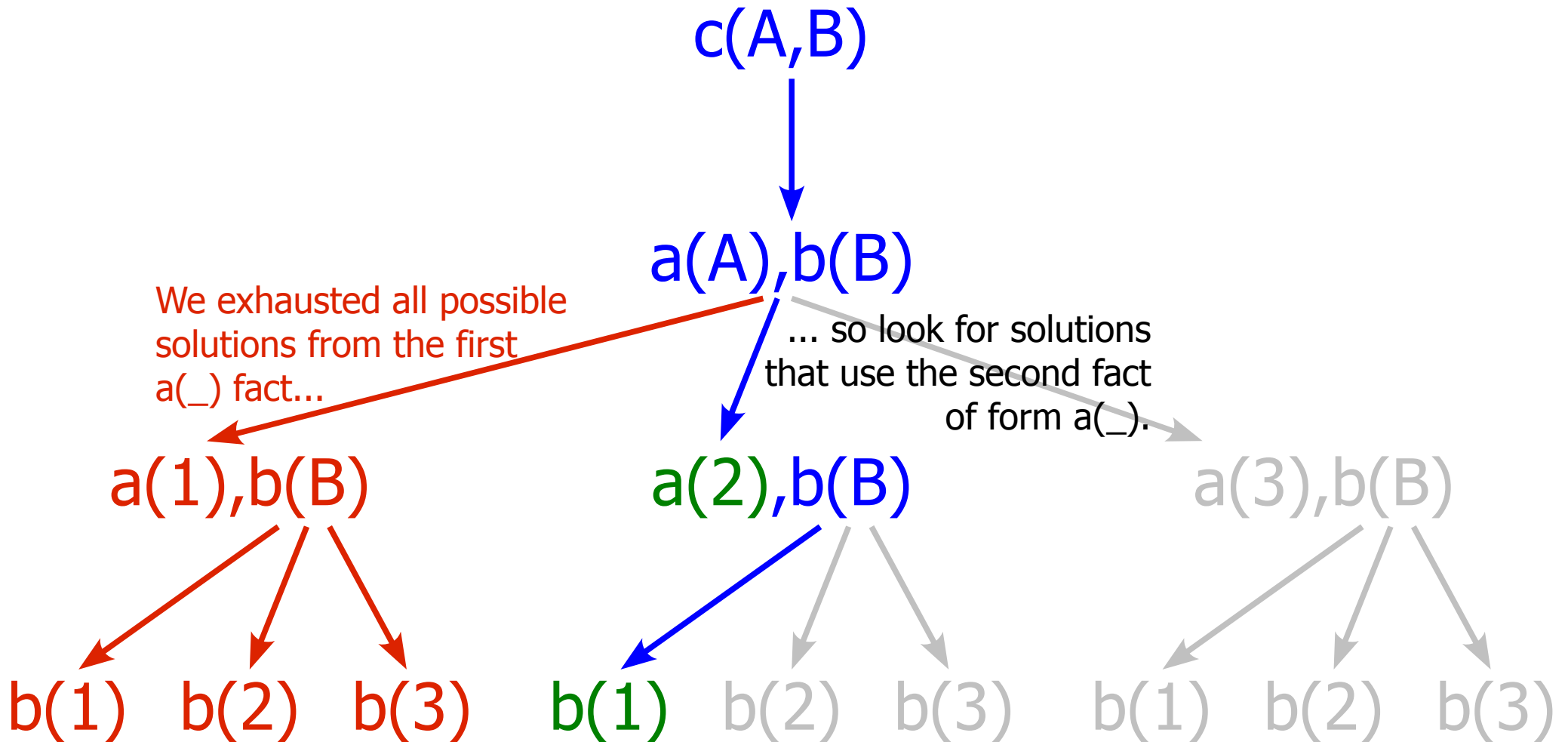
Variable bindings: $A=1, B=2$

Backtrack to find another solution



Variable bindings: $A=1, B=3$

Backtrack to find another solution



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]],1,[2,3]).

From the "fact"
take/3 clause

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

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

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

take([1|2,3],E,[1|S₁])
take([2,3],E,S₁)

take([2|[3]],E,[2|S₂])
take([3],E,S₂)

take([3|[]],E,[3|S₃])
take([],E,S₃)

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]],1,[2,3]).

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

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

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

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

take([1|2,3],E,[1|S₁])

take([2,3],E,S₁)

take([2|[3]],E,[2|S₂])

take([3],E,S₂)

take([3|[[]],E,[3|S₃])

take([[]],E,S₃)

Variable bindings: E=2, Rest=[1,3], S₁=[3]

Backtrack for another solution

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

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

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

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

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

take([1|2,3],E,[1|S₁])

take([2,3],E,S₁)

take([2|[3]],E,[2|S₂])

take([3],E,S₂)

take([3|[]],E,[3|S₃])

take([],E,S₃)

Variable bindings: E=3, Rest=[1,2], S₁=[2], S₂=[]

Prolog says "no"

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

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

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

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

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

take([1|2,3],E,[1|S₁])

take([2,3],E,S₁)

take([2|[3]],E,[2|S₂])

take([3],E,S₂)

take([3|[]],E,[3|S₃])

take([],E,S₃)

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]

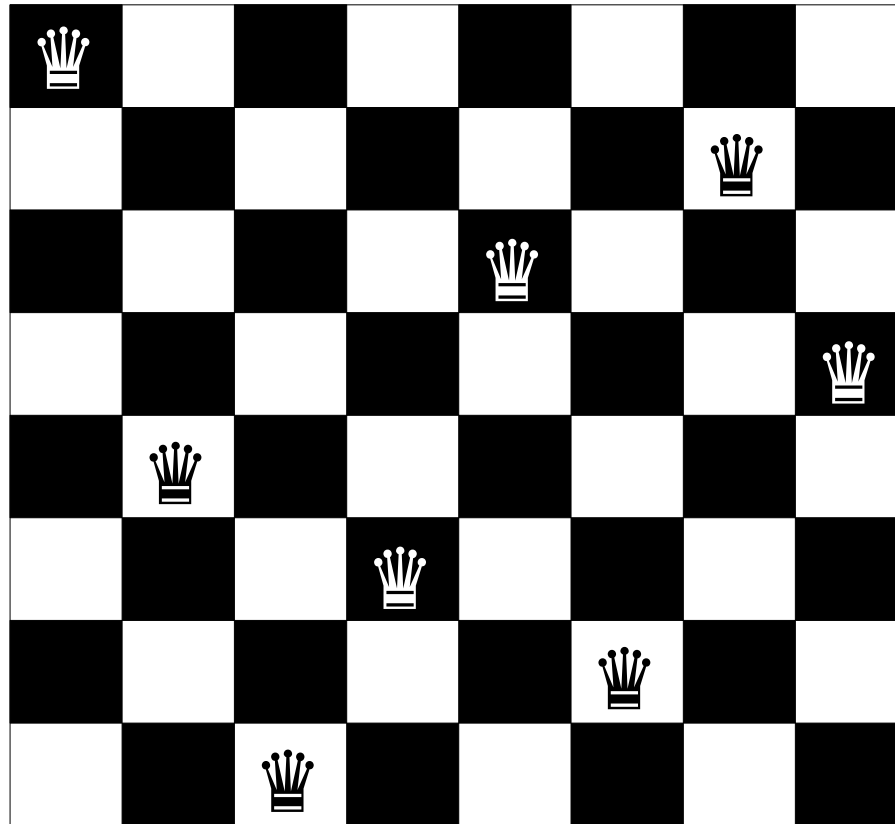
“Generate and Test” is a technique for solving problems like this

- (1) Generate a solution
- (2) Test if it is valid
- (3) 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 eight queens so that
none can take any other



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

Generate and Test works for Eight 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:

– i.e. use facts like: `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