Prolog and Logic Programming

Greg J. Badros
badros@cs.washington.edu
University of Washington, Seattle
CSE-341, Summer 1999
(C) 1999, Greg J. Badros—All Rights Reserved

Relations

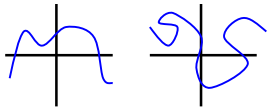
- A binary relation on two sets A and B is a subset of the Cartesian product $A \times B$
  - Cartesian product is just the set of all tuples (a,b) where a is in A, and b is in B
- Consider R = {(0,0),(1,1),(-1,-1),(2,2),(-2,-2)…}
- We call this R the equality relation, and write: 0=0, 1=1, etc.

Another relation

- Consider relation isSquareOf = $\{(a,b) \mid a = b^2\}$
- isSquareOf contains, e.g., {(0,0),(1,1),(1,-1),(4,2),(4,-2),(9,3),(9,-3)…}
- In particular, note that:
  4 isSquareOf 2 and
  4 isSquareOf -2

Functions vs. Relations

- Functions are a special case of relations in which each value from the left (from the domain) maps to exactly one value in the co-domain
- isSquareOf is a relation, but not a function
- Vertical line test



Prolog deals with relations

?- isSquareOf(9,3).	yes
?- isSquareOf(9,2).	no
?- isSquareOf(25,X).	X=5 ; X=-5
?- isSquareOf(X,-3).	X=9

Language metaphors

- Fortran, C, Pascal (Algol Family)
  - Von Neumann machine, changing state
- Scheme, Haskell, Lisp (Functional)
  - Function definition and application
- C++, Smalltalk, Java, Simula (OO)
  - Simulations of interacting entities
- Prolog, CLP(R) (Logic)
  - Theorem proving

Knuth on program correctness

"Beware of bugs in the above code;
I have only proved it correct,
not tried it."
—Donald Knuth

History of Prolog

- Developed in 1970s by Alan Colmeraur, Robert Kowalski, Phillip Roussel (University of Marseilles, France)
- David H. D. Warren provided foundations of modern implementation in the Warren Abstract Machine for DEC PDP-10 (University of Edinburgh)
- Prolog is basis for numerous other languages such as CLP(R), Prolog III, etc.

Not for general purpose programming

- More restricted computation model of proving assertions from collections of facts and rules
- Think of queries working on a database of facts with rules that permit inferring new facts
- Query is just a theorem to be proven

Why restrict applicability of a language?

- Prolog provides better built-in support for the algorithms and tasks especially useful in search problems (theorem proving is just a search problem)
- Search problems are incredibly important
  - Exponential complexity
  - But efficient techniques and heuristics help solve practical programs in a timely fashion

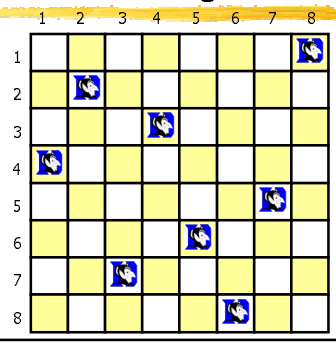
Example applications

- medical patient diagnosis
- theorem proving
- solving Rubik's cube
- type checking
- database querying

Eight Queens: A typical search problem

- Place eight (or n) queens on an $n \times n$ chessboard such that none of them are attacking any of the others
- Recursive solutions are naturally expressed using backtracking
- Solutions in C++, Pascal, Java, etc., are generally around 140–220 lines of uncommented code.

A Solution to Eight Queens



Eight Queens in Prolog

```
/* From Bratko's Prolog Programming for AI, p. 111 */
solution([]).
solution([X/Y | Others] ) :-
    solution(Others),
    member(Y, [1,2,3,4,5,6,7,8] ),
    noattack( X/Y, Others).

noattack(_, [] ).
noattack(X/Y, [X1/Y1 | Others]) :-
    Y =\= Y1,
    Y1-Y =\= X1-X,
    Y1-Y =\= X-X1,
    noattack( X/Y, Others).

template( [1/Y1,2/Y2,3/Y3,4/Y4,5/Y5,6/Y6,7/Y7,8/Y8] ).
```


Query for solution

```
?- template(S), solution(S).
  S = [1/4, 2/2, 3/7, 4/3, 5/6, 6/8, 7/5, 8/1] ;
  S = [1/5, 2/2, 3/4, 4/7, 5/3, 6/8, 7/6, 8/1] c
```

⋮

- 92 solutions in all!

A Prolog Program

- Facts and rules—
  a database of information, and rules to infer more facts
- Queries—
  the searches to perform

Example facts

```
female(karen).
male(joseph).
male(mark).
male(greg).
male(eric).
person(karen).
a_silly_fact.
```


Queries

```
?- a_silly_fact.          yes.
?- male(eric).            yes.
?- female(F).             F = karen.
?- male(bob).             no.
?- person(mark).          no.
```

Syntax for facts

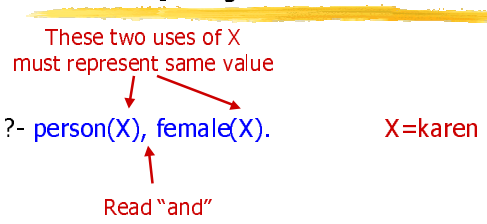
predicate.
predicate(arg1,arg2,...).

- Begin with lowercase letter
- End with a period (.)
- Numbers and underscores (_) are okay inside identifiers (also called atoms)

Variables

- Begin with an uppercase letter
- Either "instantiated" or "uninstantiated"
- X is instantiated means X stands for a particular value (similar to binding)
- Variables instantiations can be undone
- Multiple uses of the same variable in same scope must refer to same value

Variables are scoped within a query

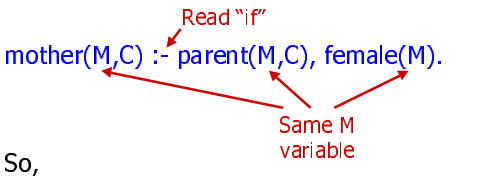


More facts

```
/* parent(P,C) means P is a parent of C */
parent(karen,greg).
parent(joseph,greg).
parent(karen,mark).
parent(joseph,mark).
```

- Interpretation of facts is imposed by the programmer
- Make assumptions clear!

A simple rule



So,

$(parent(M,C) \wedge female(M)) \rightarrow mother(M,C)$

Two interpretations of rule

- Declarative
  For a given M and C, M is the mother of C if M is the parent of C and M is female.
- Procedural
  Prove M is mother of C by proving subgoals that M is a parent of C and that M is female.

Predicate logic and English

If A then B	⇔	A → B
A only if B	⇔	A → B
A if B	⇔	B → A ⇔ A ← B
A if and only if B	⇔	A ↔ B ⇔ B ↔ A
A iff B	⇔	A ↔ B ⇔ B ↔ A

"She lives in Seattle only if she lives in Washington State."
lives_in_seattle(x) → lives_in_washington_state(x).

Implication

x	y	x→y ¬x ∨ y
False	False	True
False	True	True
True	False	False
True	True	True

We use this case to prove y

Our friend the list

?- append([1,2,3],[4,5],L). L=[1,2,3,4,5]
?- append([1,2],M,[1,2,3]). M=[3]
?- append(A,B,[1,2]). A=[], B=[1,2]
A=[1], B=[2]
A=[1,2], B=[]

- append works in multiple directions!

Declaration of append rule

```
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :-
  append(Xs,Ys,Zs).
```

- Think **declaratively!**
- Think **recursively!**

"Return value" is an argument of the rule

```
/* append(X,Y,Z)
   succeeds iff  Z is the list that is
   the list Y appended to the list X */
```

- Enables it to use any/all of the arguments to compute what's left
- Use uninstantiated variables (i.e., those starting with capital letters) to ask for a return value

Terminology

- simple term — number, variable, or atom
  e.g., -92 X greg
- compound term —
  atom + parenthesized subterms
  e.g., parent(joe,greg)
  functor sub-term arguments
- Facts, rules, and queries are made from terms… the functor is the predicate

Lists

- [] is the empty list
- . predicate is like Scheme's cons:
  ?- A = .(1, .(2, .(3, []))). A=[1, 2, 3]
- [...] shorthand syntax:
  ?- A = [1,2,3] A=[1, 2, 3]
- [E1... | Tail] notation
  ?- A = [1,2|3]. A=[1, 2|3]
  ?- A = [1,2|[3]]. A=[1, 2, 3]

Lists need not be homogeneous

?- A = "Hi". A=[72,105]
?- A = [1,"Hi",greg]. A=[1,[72,105],greg]
?- A = [1,g], B=[A,A]. A=[1,g] B=[[1,g],[1,g]]
?- A = [1,g], B=[A|A]. A=[1,g] B=[[1,g],1,g]

Unification of terms

- Similar to Haskell's pattern matching

Request to unify

?- p(X,foo) = p(bar,Y). X=bar, Y=foo
?- foo = X. X=foo
?- X = foo. X=foo
?- foo = bar. No

Unification of terms S and T

Terms *S* and *T* unify if and only if:

- *S* and *T* are both constants, and they are the same object; or
- *S* is uninstantiated. Substitute *T* for *S*; or
- *T* is uninstantiated. Substitute *S* for *T*; or
- *S* and *T* are structures, have same principal functor, and the corresponding components unify.

Recursive definition!

More unification examples

?- A=parent(joe,greg),A=parent(X,Y). X=joe, Y=greg
?- A=parent(joe,greg),A=parent(X). No
?- A=people([joe,greg]),A=people(X). X=[joe,greg]
?- A=[1,2,3,4],A=[X,Y|Z]. X=1,Y=2,Z=[3,4]

Unification is implicit in rule application

```
/* simple rule: X is the same as X. */
identity(X,X).

?- identity( p(X,foo), p(bar,Y) ).
                                X=bar, Y=foo

/* could have written the rule as: */
identity(X,Y) :- X = Y.
```

Unification in append rule

```
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :-
  append(Xs,Ys,Zs).
```

append([1,2,3],A,[1,2,3,4]). results in
[X|Xs] = [1,2,3], A=Ys, [X|Zs] = [1,2,3,4]
and thus:
X=1, Xs=[2,3], Zs=[2,3,4]
so must prove: append([2,3],A,[2,3,4])

Trace of app([1,2,3],A,[1,2,3,4])

```
T Call: ( 7) app([1, 2, 3], _G235, [1, 2, 3, 4])
T Call: ( 8) app([2, 3], _G235, [2, 3, 4])
T Call: ( 9) app([3], _G235, [3, 4])
T Call: ( 10) app([], _G235, [4])
T Exit: ( 10) app([], [4], [4])
T Exit: ( 9) app([3], [4], [3, 4])
T Exit: ( 8) app([2, 3], [4], [2, 3, 4])
T Exit: ( 7) app([1, 2, 3], [4], [1, 2, 3, 4])
```


Trace of app([1,2,3],[4],Z)

```
T Call: ( 7) app([1, 2, 3], [4], _G191)
T Call: ( 8) app([2, 3], [4], _G299)
T Call: ( 9) app([3], [4], _G302)
T Call: ( 10) app([], [4], _G305)
T Exit: ( 10) app([], [4], [4])
T Exit: ( 9) app([3], [4], [3, 4])
T Exit: ( 8) app([2, 3], [4], [2, 3, 4])
T Exit: ( 7) app([1, 2, 3], [4], [1, 2, 3, 4])
```


Arithmetic in Prolog

?- X=2+3, X=5. No

⇨2+3 does not unify with 5
- 2+3 is an **unevaluated** expression
- that expression is not the same as the literal 5

?- X **is** 2+3, X=5. Yes

Force arithmetic evaluation

Arithmetic only works forward in Prolog

```
sum(X,Y,Z) :- Z is X + Y.
```

?- sum(4,5,Z). Z=9
?- sum(4,Y,9). Error!

Constraint Logic Programming

- Built on top of Prolog's foundations
- Developed by Jaffar and Lassez at Monash University in Melbourne, Australia
- Includes domain-specific constraint solvers to augment the logical deduction algorithm
- Different domains are targeted with different specialized solvers
  - CLP(FD), for finite domains
  - CLP(R), for real number

Importance of Constraint Logic Programming

"Were you to ask me which programming paradigm is likely to gain most in commercial significance over the next 5 years I'd have to pick Constraint Logic Programming…"
— Dick Pountain

CLP(R) can do arithmetic in all directions!

```
sum(X,Y,Z) :- Z = X + Y.
```

= instead of **is**

```
?- sum(4,5,Z).        Z = 9
?- sum(4,Y,9).         Y=5
?- sum(4,Y,Z).        Y=Z-4
```

Symbolic solution!

```
/* rule could have been: */
sum(X,Y,X+Y).
```


member rule

```
/* member(X,Y) succeeds iff
   X is a member of the list Y. */
```

Example uses:

```
?- member(1,[1,2,3]).      Yes
?- member(7,[1,2,3]).      No
?- member(3,foo).          No
?- member(foo,[]).         No
```


Definition of member

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

- X is a member of a list starting with X; and X is a member of a list starting with anything as long as it is a member of the rest of the list

The declarative interpretation falls short…

Two queries:

```
X = [1,2,3], member(7,X).
member(7,X), X = [1,2,3].
```

} Identical declarative semantics

```
?- X = [1,2,3], member(7,X).      No
?- member(7,X), X=[1,2,3].        …
```

Infinite computation

More uses of member

```
?- member(X,[]).                  No
?- member(X,[1,2,3]).              1 ;
                                   2 ;
                                   3 ;
                                  No
?- member(7,X).          X = [7|_G219] ;
                  X = [_G218, 7|_G222] ;
       X = [_G218, _G221, 7|_G225] c
```

How Prolog tries to prove

Rule order:

- Select the first applicable rule from top to bottom

Goal order:

- Prove subgoals from left to right

Evidence of top to bottom rule order

```
?- male(X).          X=joseph ;
                     X=mark ;
                     X=greg ;
                     X=eric ;
                     No
```

- Order of results mirrors the order that the facts appeared in the database

Order of rules matters!

```
mem1(X,[X|_]).          /* A */
mem1(X,[_|T]) :- mem1(X,T).   /* B */
```

versus

```
mem2(X,[_|T]) :- mem2(X,T).   /* A */
mem2(X,[X|_]).          /* B */
```

- Which is more efficient?

Trace of mem1(2,[1,2,3])

```
T Call:  (  8) mem1(2, [1, 2, 3])   matches B
T Call:  (  9) mem1(2, [2, 3])      matches A
T Exit:  (  9) mem1(2, [2, 3])      A succeeds
T Exit:  (  8) mem1(2, [1, 2, 3])   B succeeds
```

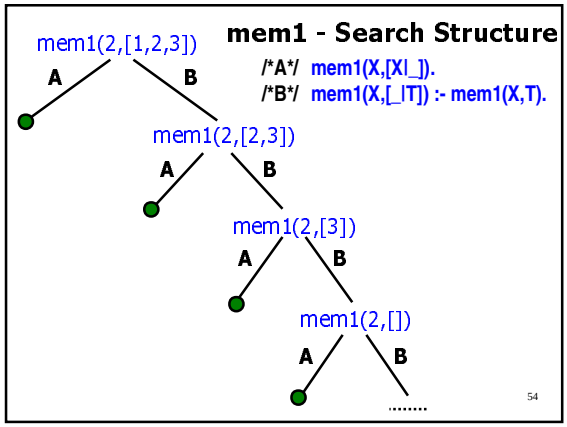

Trace of mem2(2,[1,2,3])

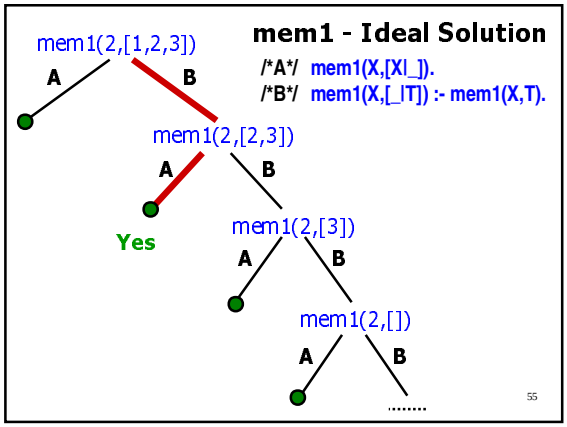
```
T Call:  (  8) mem2(2, [1, 2, 3])   matches A
T Call:  (  9) mem2(2, [2, 3])      matches A
T Call:  ( 10) mem2(2, [3])         matches A
T Call:  ( 11) mem2(2, [])          matches A
T Fail:  ( 11) mem2(2, [])          no match
T Redo:  ( 10) mem2(2, [3])         match B?
T Fail:  ( 10) mem2(2, [3])         no match
T Redo:  (  9) mem2(2, [2, 3])      match B?
T Exit:  (  9) mem2(2, [2, 3])      B succeeds
T Exit:  (  8) mem2(2, [1, 2, 3])   A succeeds
```

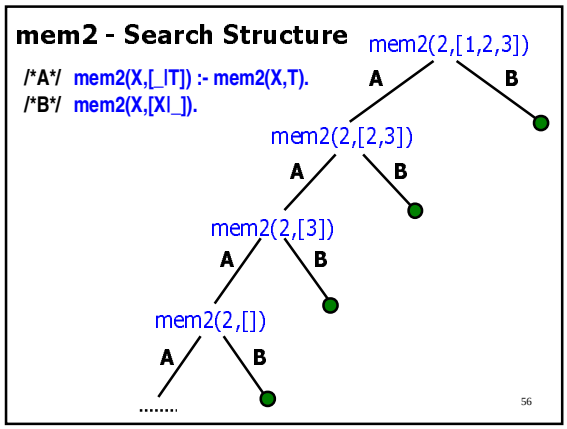

mem1 was faster

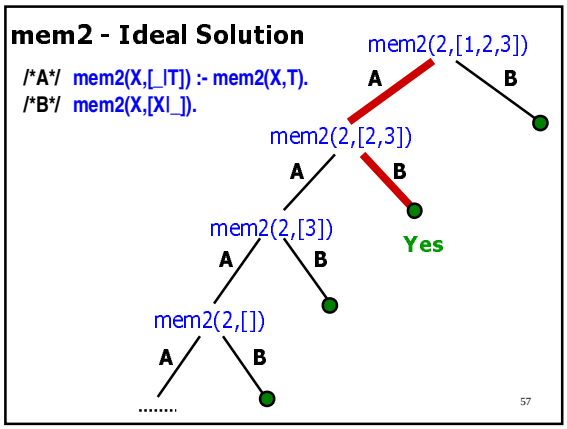
- mem1 had the base case listed first
- base case had no sub-goals

- Let's see how the searches compare more visually...









Prolog is not an oracle!

- First applicable rule
  - Always takes left branch if its applicable
- Prove subgoals in order
  - Only one subgoal for **A** so each node in the tree from an **A** branch has only one subgoal
  - No subgoals for **B** so each node in the tree from a **B** branch is a solution

14-July-1999 G. Badros -- CSE 341, Prolog 58

How Prolog Proceeds - 1

/*A*/ mem2(X,[_|T]) :- mem2(X,T).
/*B*/ mem2(X,[Xl_]).

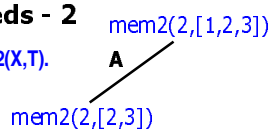
mem2(2,[1,2,3])

Matches **A**,
Unify [_|T]
with [1,2,3]
So T = [2,3],
and must prove
mem2(X,T).

59

How Prolog Proceeds - 2

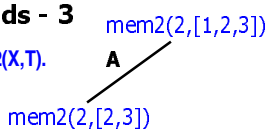
```
/*A*/ mem2(X,[_|T]) :- mem2(X,T).
/*B*/ mem2(X,[X|_]).
```



60

How Prolog Proceeds - 3

```
/*A*/ mem2(X,[_|T]) :- mem2(X,T).
/*B*/ mem2(X,[X|_]).
```

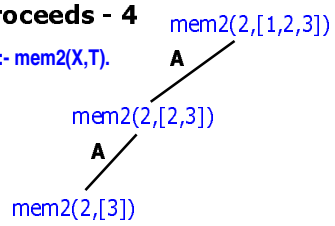


Matches **A**,
Unify [_|T]
with [2,3]
So T = [3],
and must prove
mem2(X,T).

61

How Prolog Proceeds - 4

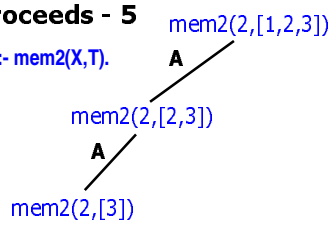
```
/*A*/ mem2(X,[_|T]) :- mem2(X,T).
/*B*/ mem2(X,[X|_]).
```



62

How Prolog Proceeds - 5

```
/*A*/ mem2(X,[_|T]) :- mem2(X,T).
/*B*/ mem2(X,[X|_]).
```

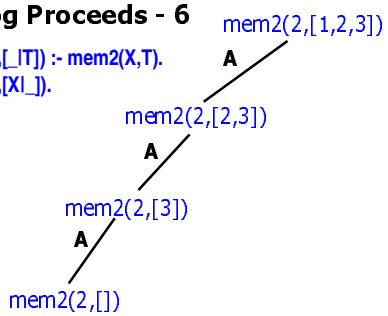


Matches **A**,
Unify [_|T]
with [3]
So T = [],
and must prove
mem2(X,T).

63

How Prolog Proceeds - 6

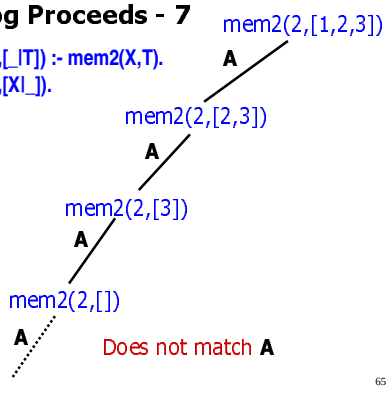
```
/*A*/ mem2(X,[_|T]) :- mem2(X,T).
/*B*/ mem2(X,[X|_]).
```



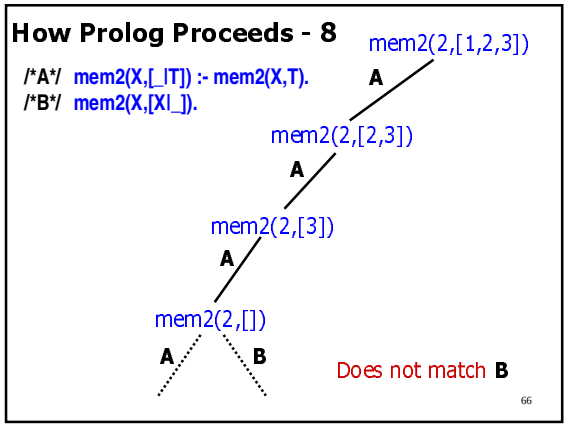
64

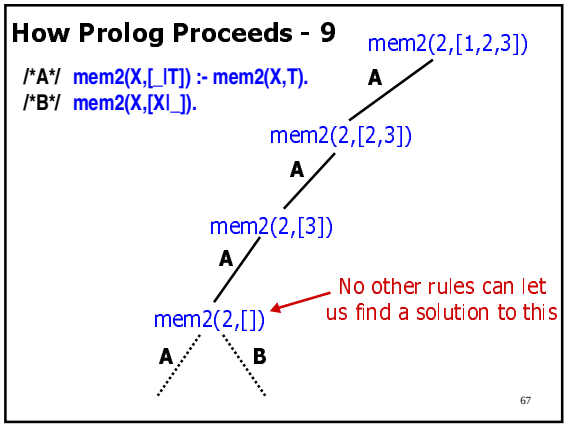
How Prolog Proceeds - 7

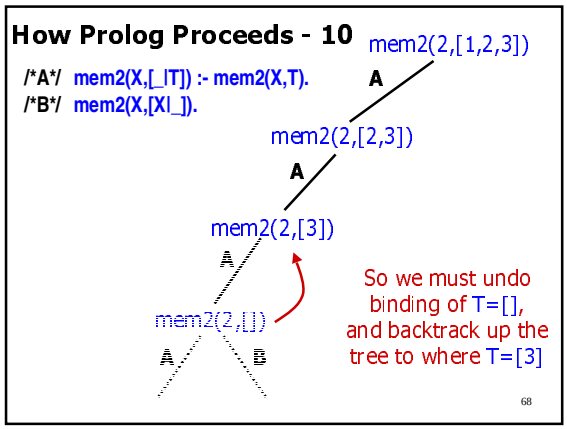
```
/*A*/ mem2(X,[_|T]) :- mem2(X,T).
/*B*/ mem2(X,[X|_]).
```

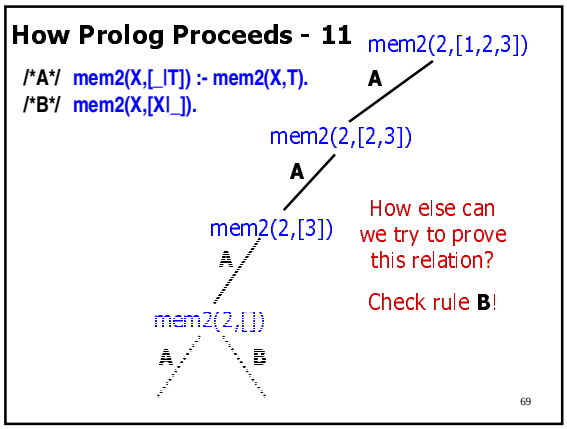


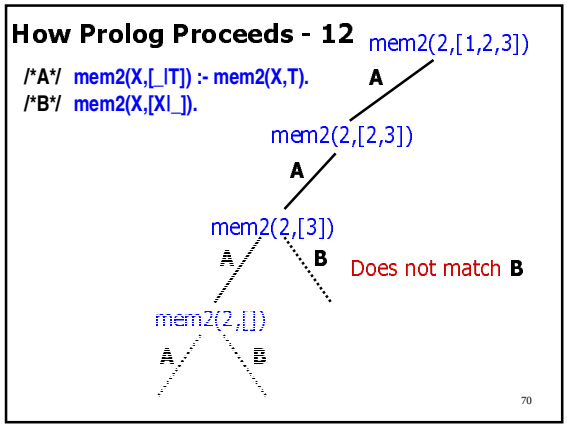
65

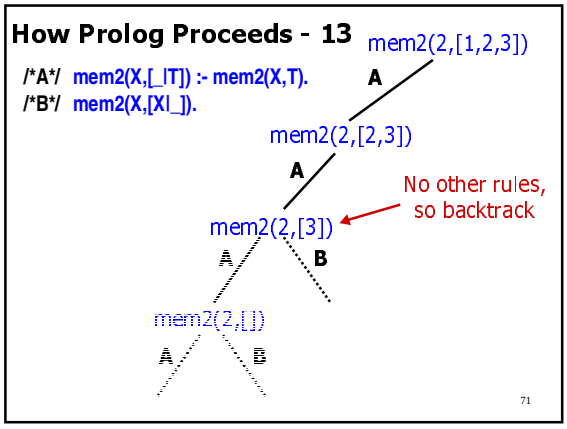

How Prolog Proceeds - 8
/*A*/ mem2(X,[_|T]) :- mem2(X,T).
/*B*/ mem2(X,[X|_]).
mem2(2,[1,2,3])
A
mem2(2,[2,3])
A
mem2(2,[3])
A
mem2(2,[])
A
B
Does not match B
66


How Prolog Proceeds - 9
/*A*/ mem2(X,[_|T]) :- mem2(X,T).
/*B*/ mem2(X,[X|_]).
mem2(2,[1,2,3])
A
mem2(2,[2,3])
A
mem2(2,[3])
A
mem2(2,[])
No other rules can let us find a solution to this
A
B
67

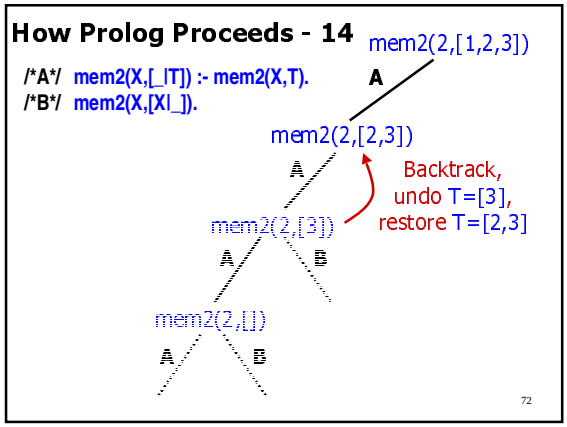

How Prolog Proceeds - 10
/*A*/ mem2(X,[_|T]) :- mem2(X,T).
/*B*/ mem2(X,[X|_]).
mem2(2,[1,2,3])
A
mem2(2,[2,3])
A
mem2(2,[3])
A
mem2(2,[])
So we must undo binding of T=[], and backtrack up the tree to where T=[3]
A
B
68


How Prolog Proceeds - 11
/*A*/ mem2(X,[_|T]) :- mem2(X,T).
/*B*/ mem2(X,[X|_]).
mem2(2,[1,2,3])
A
mem2(2,[2,3])
A
mem2(2,[3])
How else can we try to prove this relation?
Check rule B!
A
mem2(2,[])
A
B
69


How Prolog Proceeds - 12
/*A*/ mem2(X,[_|T]) :- mem2(X,T).
/*B*/ mem2(X,[X|_]).
mem2(2,[1,2,3])
A
mem2(2,[2,3])
A
mem2(2,[3])
A
B
Does not match B
mem2(2,[])
A
B
70

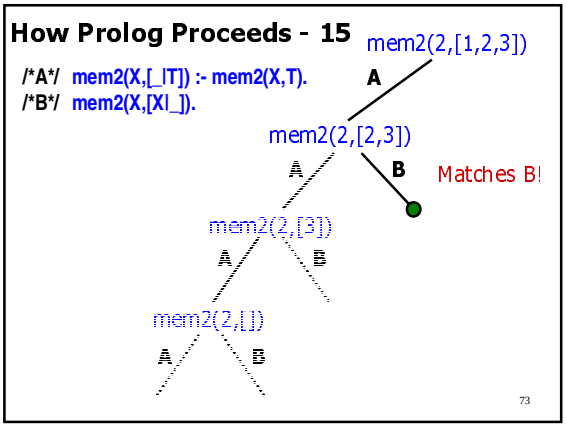

How Prolog Proceeds - 13
/*A*/ mem2(X,[_|T]) :- mem2(X,T).
/*B*/ mem2(X,[X|_]).
mem2(2,[1,2,3])
A
mem2(2,[2,3])
A
No other rules, so backtrack
mem2(2,[3])
A
B
mem2(2,[])
A
B
71

How Prolog Proceeds - 14



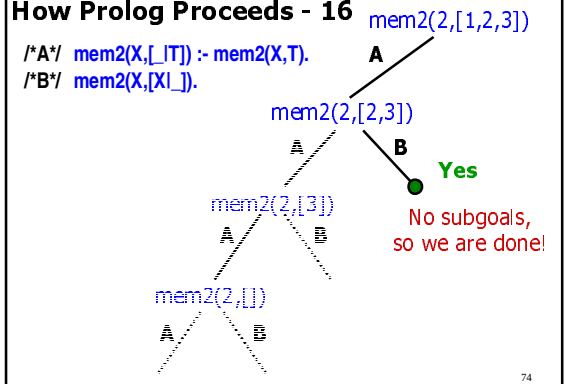
72

How Prolog Proceeds - 15



73

How Prolog Proceeds - 16



74

Search algorithm

- Depth first visitation of the nodes in the search tree
- What about rules with multiple goals?

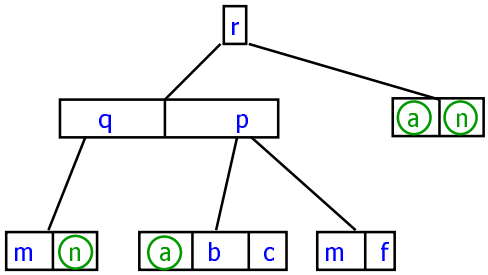
Multiple sub-goals

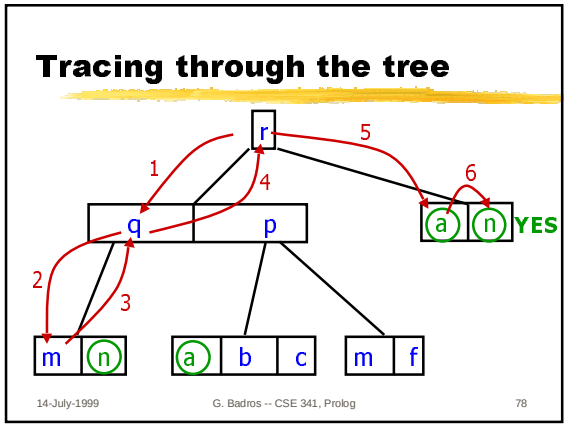
```
p :- a, b, c.
p :- m, f.
q :- m, n.
r :- q, p.
r :- a, n.
n.
a.

?- r.
```

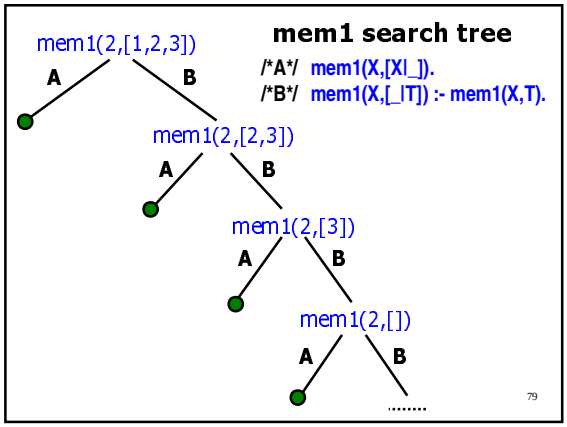
Nodes will now contain multiple sub-goals each the root of a new tree

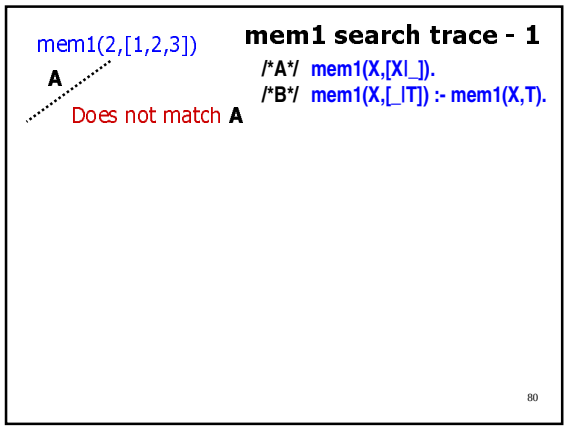
Must prove **q**, and then prove **p**

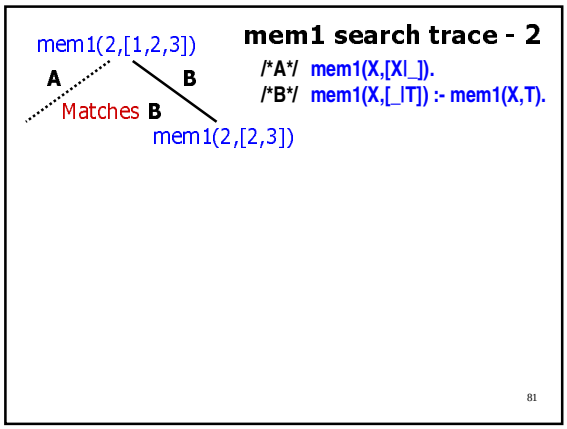
Another search tree...

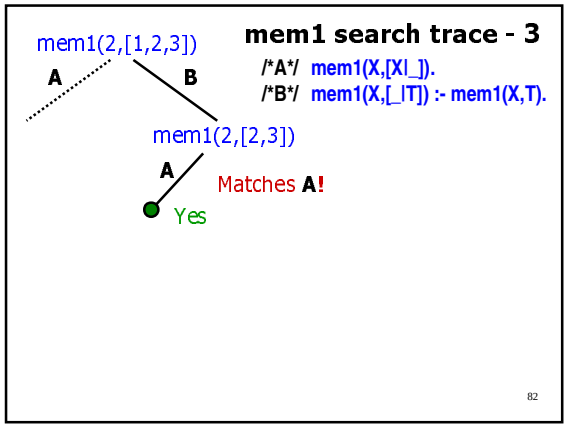


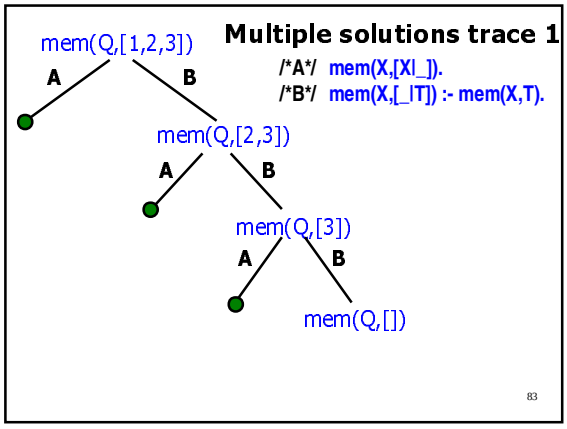

Tracing through the tree
r
q
p
a
n
YES
m
n
a
b
c
m
f
1
2
3
4
5
6
14-July-1999
G. Badros -- CSE 341, Prolog
78

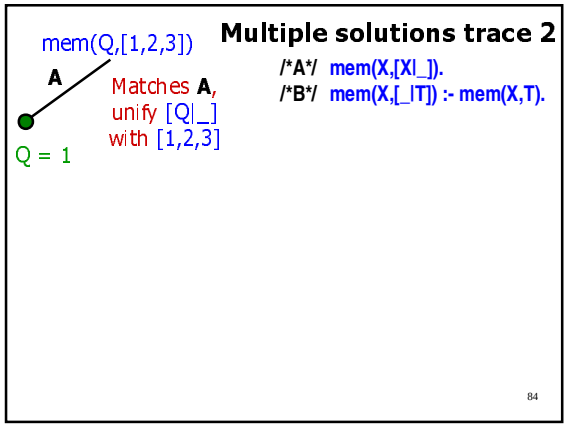

mem1 search tree
mem1(2,[1,2,3])
/*A*/ mem1(X,[X|_]).
/*B*/ mem1(X,[_|T]) :- mem1(X,T).
A
B
mem1(2,[2,3])
mem1(2,[3])
mem1(2,[])
79

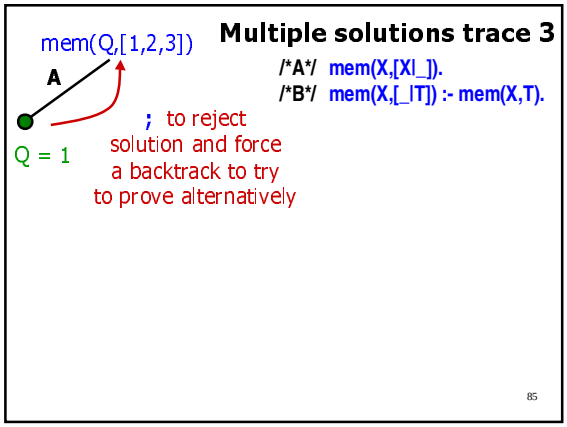

mem1 search trace - 1
mem1(2,[1,2,3])
/*A*/ mem1(X,[X|_]).
/*B*/ mem1(X,[_|T]) :- mem1(X,T).
A
Does not match A
80

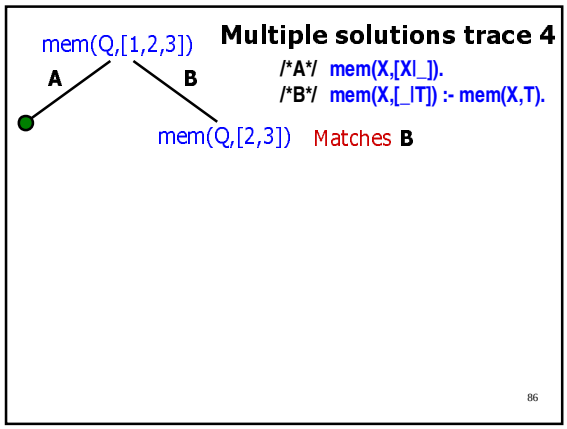

mem1 search trace - 2
mem1(2,[1,2,3])
/*A*/ mem1(X,[X|_]).
/*B*/ mem1(X,[_|T]) :- mem1(X,T).
A
B
Matches B
mem1(2,[2,3])
81

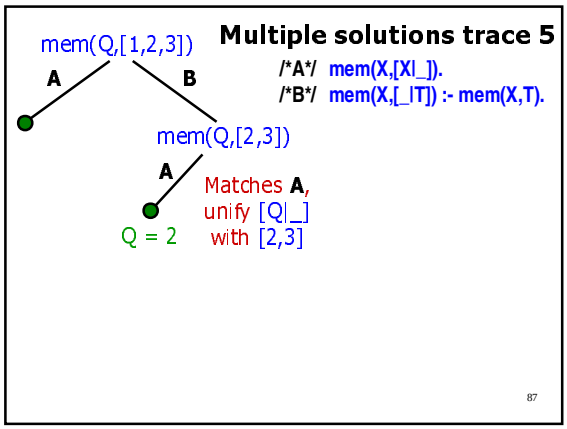

mem1 search trace - 3
mem1(2,[1,2,3])
/*A*/ mem1(X,[X|_]).
/*B*/ mem1(X,[_|T]) :- mem1(X,T).
A
B
mem1(2,[2,3])
A
Matches A!
Yes
82

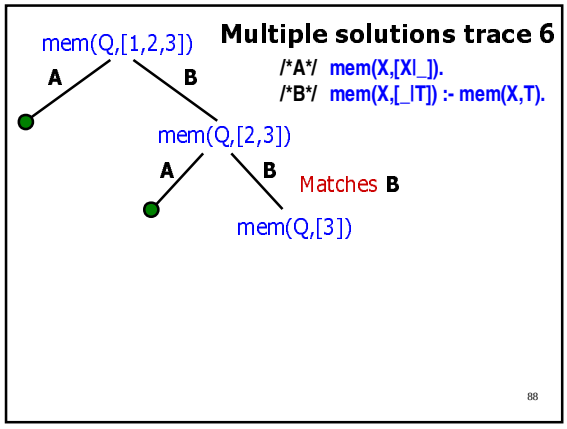

Multiple solutions trace 1
mem(Q,[1,2,3])
/*A*/ mem(X,[X|_]).
/*B*/ mem(X,[_|T]) :- mem(X,T).
A
B
mem(Q,[2,3])
mem(Q,[3])
mem(Q,[])
83

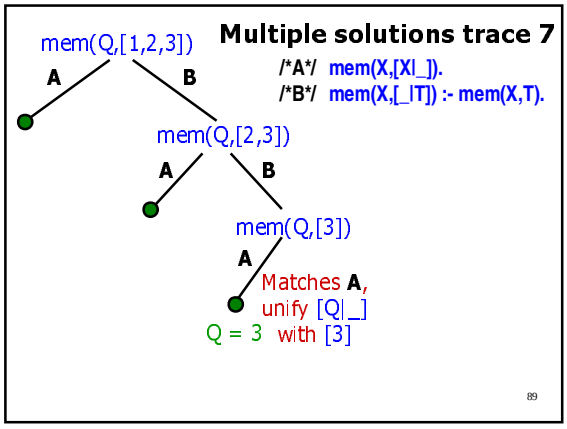

Multiple solutions trace 2
mem(Q,[1,2,3])
A
Matches A,
unify [Q|_]
with [1,2,3]
Q = 1
/*A*/ mem(X,[X|_]).
/*B*/ mem(X,[_|T]) :- mem(X,T).
84

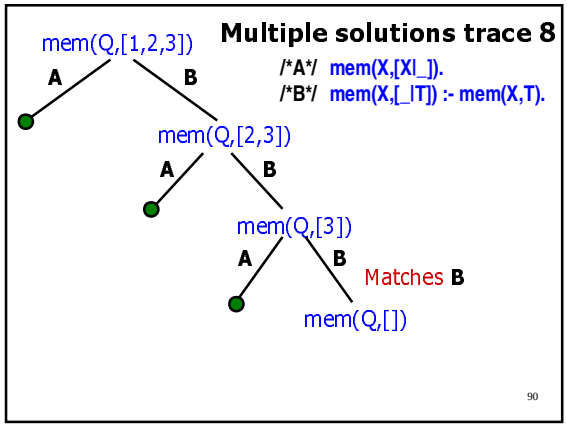

Multiple solutions trace 3
mem(Q,[1,2,3])
A
; to reject
solution and force
a backtrack to try
to prove alternatively
Q = 1
/*A*/ mem(X,[X|_]).
/*B*/ mem(X,[_|T]) :- mem(X,T).
85

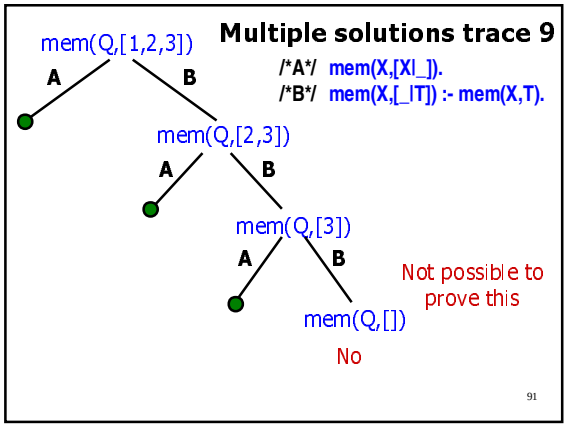

Multiple solutions trace 4
mem(Q,[1,2,3])
A
B
mem(Q,[2,3])
Matches B
/*A*/ mem(X,[X|_]).
/*B*/ mem(X,[_|T]) :- mem(X,T).
86

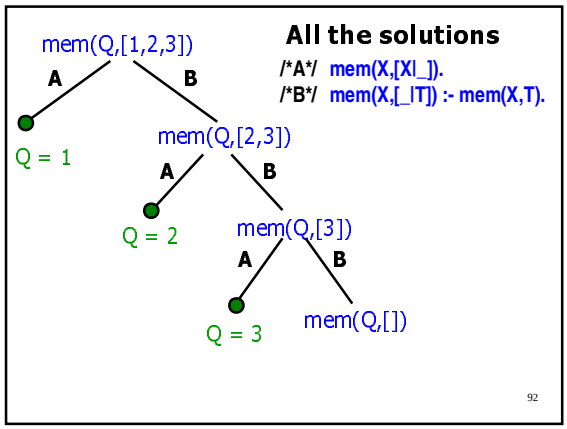

Multiple solutions trace 5
mem(Q,[1,2,3])
A
B
mem(Q,[2,3])
A
Matches A,
unify [Q|_]
with [2,3]
Q = 2
/*A*/ mem(X,[X|_]).
/*B*/ mem(X,[_|T]) :- mem(X,T).
87


Multiple solutions trace 6
mem(Q,[1,2,3])
A
B
mem(Q,[2,3])
A
B
Matches B
mem(Q,[3])
/*A*/ mem(X,[X|_]).
/*B*/ mem(X,[_|T]) :- mem(X,T).
88


Multiple solutions trace 7
mem(Q,[1,2,3])
A
B
mem(Q,[2,3])
A
B
mem(Q,[3])
A
Matches A,
unify [Q|_]
Q = 3 with [3]
/*A*/ mem(X,[X|_]).
/*B*/ mem(X,[_|T]) :- mem(X,T).
89







Backtracking is not always what we want

- Patterns may match where we do not intend
- Backtracking is expensive— we may know more about our problem and can help the algorithm be "smarter"
- We may want to specify a situation that we know definitively results in failure

delete_all example

```
/* delete_all(List,E,Result) means that
   Result is a a list just like List except all
   elements E are missing. */
delete_all([], E, []).

delete_all([E|Tail], E, Res) :-
      delete_all(Tail, E, Res).

delete_all([Head|Tail], E, [Head|Res]) :-
      delete_all(Tail, E, Res).
```


A query for delete_all

```
?- delete_all([1,2,3],2,R).      R=[1,3] ;
                                 R=[1,2,3] ;
                                 No
```

Why this solution?

delete_all can succeed in any of three ways...

```
delete_all([], E, []).

delete_all([E|Tail], E, Res) :-
      delete_all(Tail, E, Res).

delete_all([Head|Tail], E, [Head|Res]) :-
      delete_all(Tail, E, Res).
```

- Order in file only tells which rules are attempted first—later matching rules can be used after backtracking!

delete_all has multiple matching rules

```
delete_all([E|Tail], E, Res) :-
      delete_all(Tail, E, Res).

delete_all([Head|Tail], E, [Head|Res]) :-
      delete_all(Tail, E, Res).
```

delete_all([2,3],2,R).

- Can be proven using either of the above!
  R=[3], or R=[2,3]

Third rule contained implicit assumption

```
delete_all([Head|Tail], E, [Head|Res]) :-
      delete_all(Tail, E, Res).
```

- Want above rule to apply only when Head is not E
- That is exactly the complement of rule 2
- So we can make the algorithm only try rule 3 if rule 2 did not succeed

Use a "cut" — !

- We can make rule 2 prevent backtracking with the "cut" operator, written **!**

```
delete_all([E|Tail], E, Res) :-
      delete_all(Tail, E, Res), !.
```

- Now the search algorithm will not try any other rules for delete_all after the above rule succeeds

The query again

```
?- delete_all([1,2,3],2,R).        R=[1,3] ;
                                   No
```

- Now we get only the single correct solution!

Cut divides problem into backtracking regions

foo := a, b, c, !, d, e, f.

- Search may try various ways to prove a, b, and c, backtracking freely while solving those sub-goals
- Once a, b, and c are proved, that sub-answer is frozen, and d, e, f must be proved without changing a, b, or c

Controversy over cut

- Prolog is meant to be declarative
- cut operator alters the behaviour of the built-in searching algorithm
- No declarative interpretation for cut—you must think about the algorithm to understand its effects

cut and not

- We can write the not predicate using a cut operator:

```
not(P) :- P, !, fail.
not(P).
```

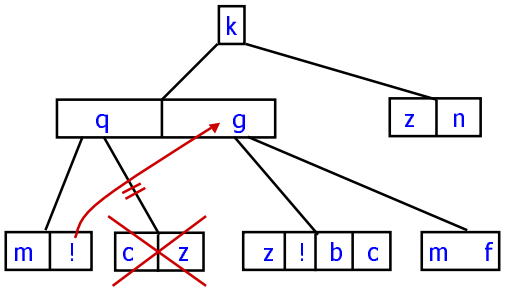
- Uses built-in fail predicate that always fails
- Cut operator prevents the search algorithm from backtracking to use the second rule to satisfy P when the first rule already failed

!, fail combination

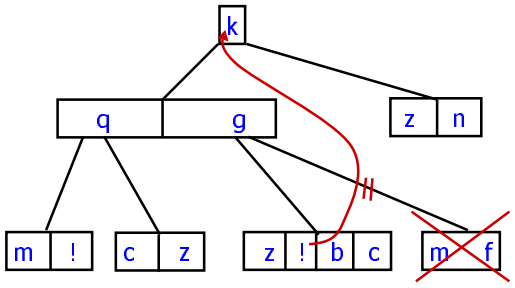
- Another common use of the cut is with fail
- Use to force failure in special cases that are easy to rule out immediately

```
average_taxpayer(X) :-
  lives_in_bermuda(X), !, fail.
average_taxpayer(X) :-
  /* complicated rules here... */
```


Tree view of a cut



Tree view of a cut



Difference lists— an incomplete data structure

- Access to start of a list is constant time
- Access to the end of a list is slow—linear time to iterate through all elements.
- Represent one conceptual list with two lists:

```
dl([a,b], []).
dl([a,b,c], [c]).
dl([a,b|E],E).
dl([a,b,c|F], [c|F).
```

All represent [a,b]

Hole variables for accessing end of the list

- dl(L,E) represents the list containing all elements in L that are not in E
- Typically, L and E are "open" lists that have an uninstantiated variable at the tail: dl([a,b|E], E).

Hole variable

- That variable is a hole variable—permits constant time access

Appending elements to a difference list

```
dl_app(dl(X,Y), dl(Y,Z), dl(X,Z)).

?- dl_app(dl([1,2,3|M],M),
          dl([4,5,6|N],N),
          dl(A,P))
          A = [1,2,3,4,5,6|N], P = N
```


Unifications in dl_app

```
dl_app(dl(X,Y), dl(Y,Z), dl(X,Z)).

?- dl_app(dl([1,2,3|M],M),
          dl([4,5,6|N],N),
          dl(A,P))
          A = [1,2,3,4,5,6|N], P = N
```


Another way to query

```
dl_app(dl(X,Y), dl(Y,Z), dl(X,Z)).

?- dl_app(dl([1,2,3|M],M),
          dl([4,5,6|N],N),
          dl(A,[]))
          A = [1,2,3,4,5,6]
```