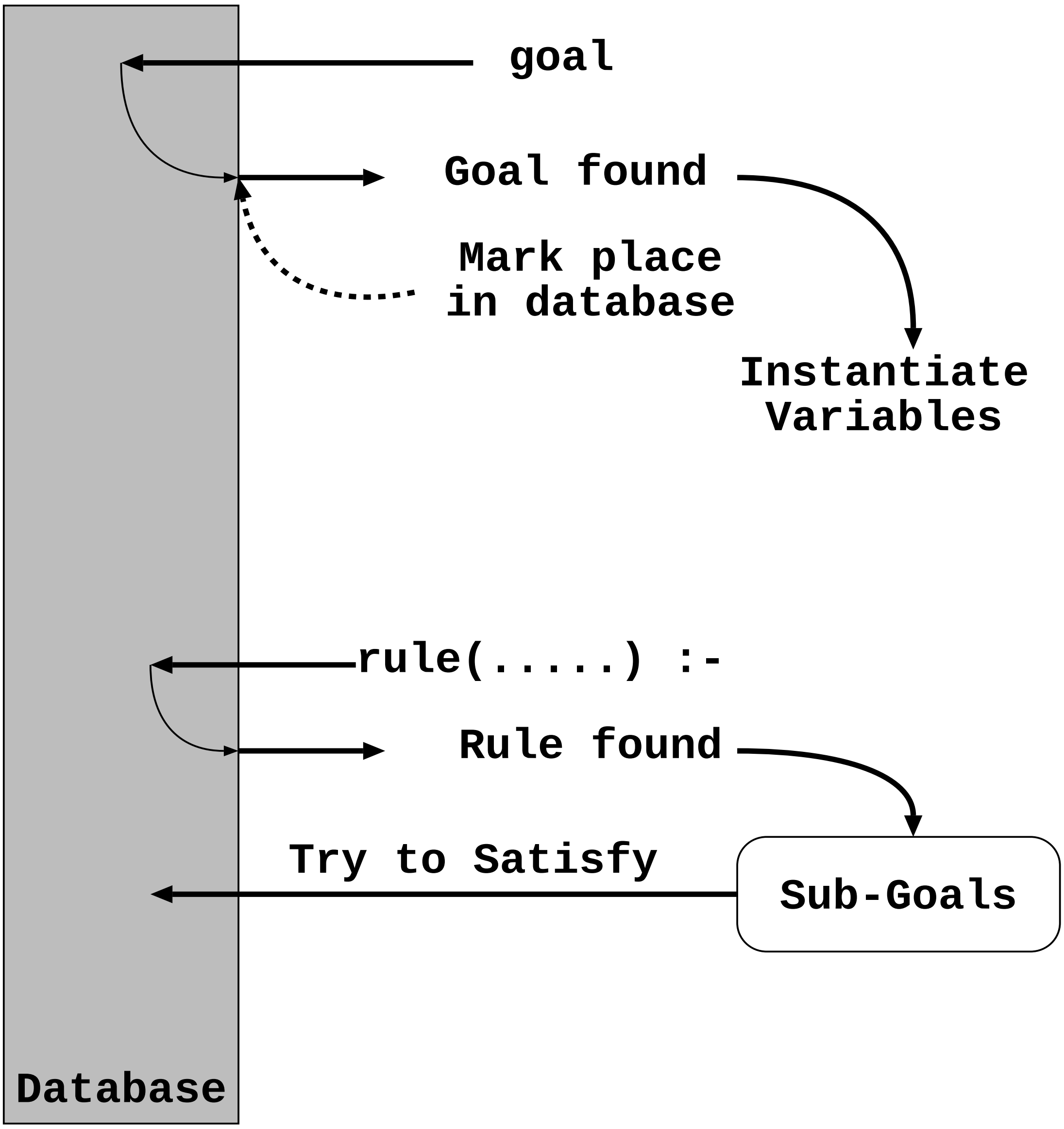
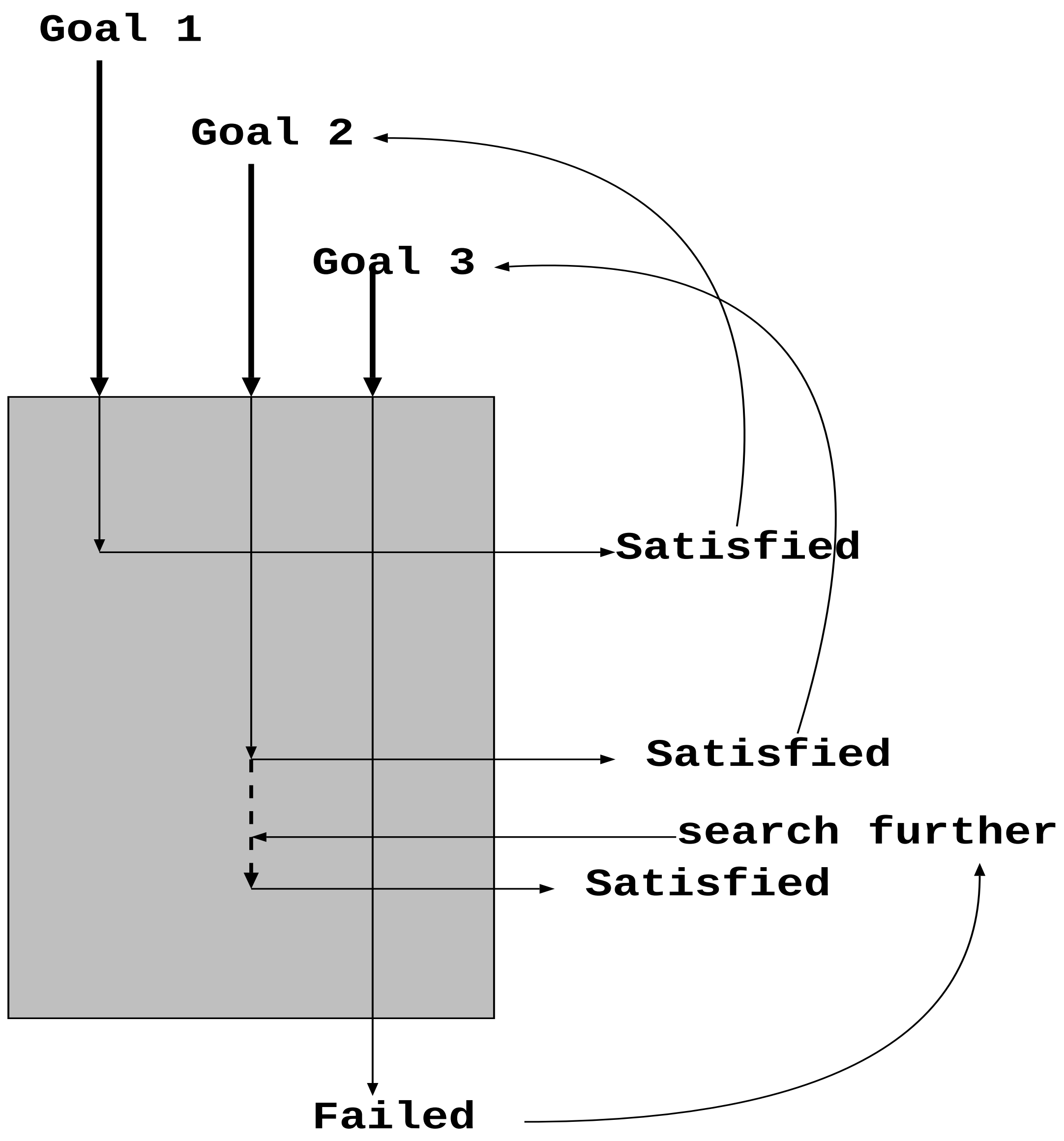
Matching


goal
Goal found
Mark place
in database
Instantiate
Variables
rule(.....) :-
Rule found
Try to Satisfy
Sub-Goals
Database

Failed


Goal 1
Goal 2
Goal 3
Satisfied
Satisfied
search further
Satisfied
Failed

Backtracking

A Failure was found

Backtracking

undoes step by step what has been done and tries the next approach

Step by Step

Go back one step or operation

See if another possibility exists

If yes

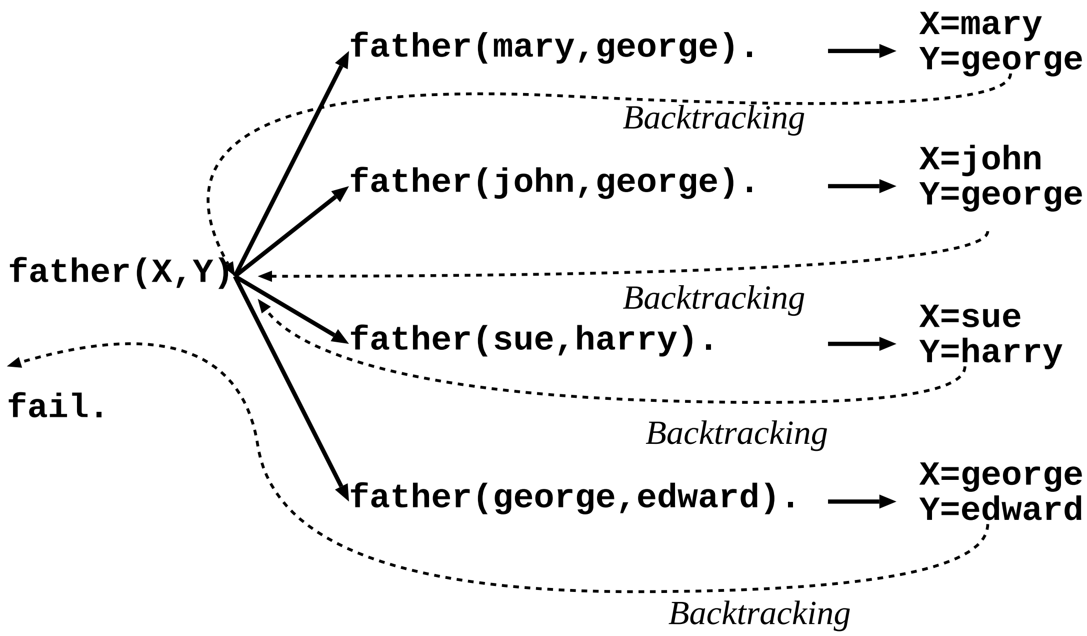
Try it

If no

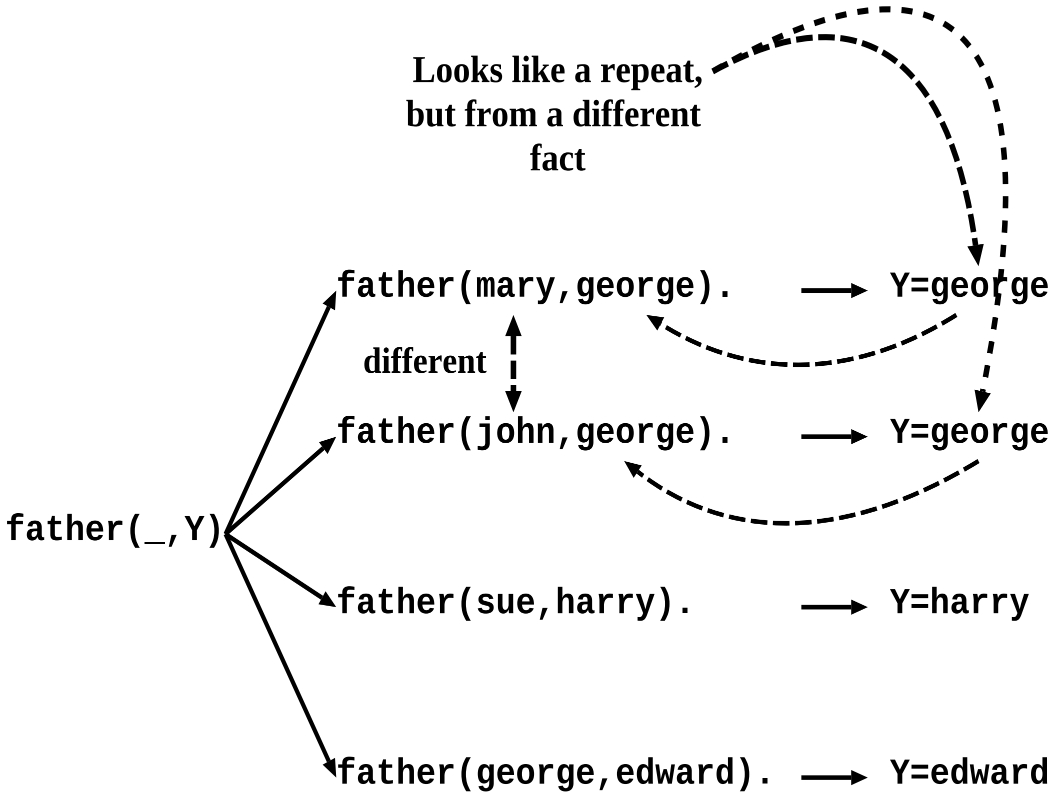
go back another step

All Possibilities

```
father(mary,george).
father(john,george).
father(sue,harry).
father(george,edward).
```



Backtrack


Looks like a repeat,
but from a different
fact
father(mary,george).
Y=george
different
father(john,george).
Y=george
father(_,Y)
father(sue,harry).
Y=harry
father(george,edward).
Y=edward

```
| ?- [user]
     .
| father(mary,george).
| father(john,george).
| father(sue,harry).
| father(george,edward).
| father(X) :- father(_,X).
| user consulted, 0 msec 880 bytes

yes
| ?- father(X).

X = george ? ;
X = george ? ;
X = harry ? ;
X = edward ? ;
no
| ?-
```

In What Order

```
person(adam).
person(X) :- mother(X,Y).
person(eve).

mother(cain,eve).
mother(abel,eve).
mother(jabal,adah).
mother(tubalcain,zillah).

person(X).
| ?- person(X).

X = adam ? ;
X = cain ? ;
X = abel ? ;
X = jabal ? ;
X = tubalcain ? ;
X = eve ? ;
no
| ?-
```

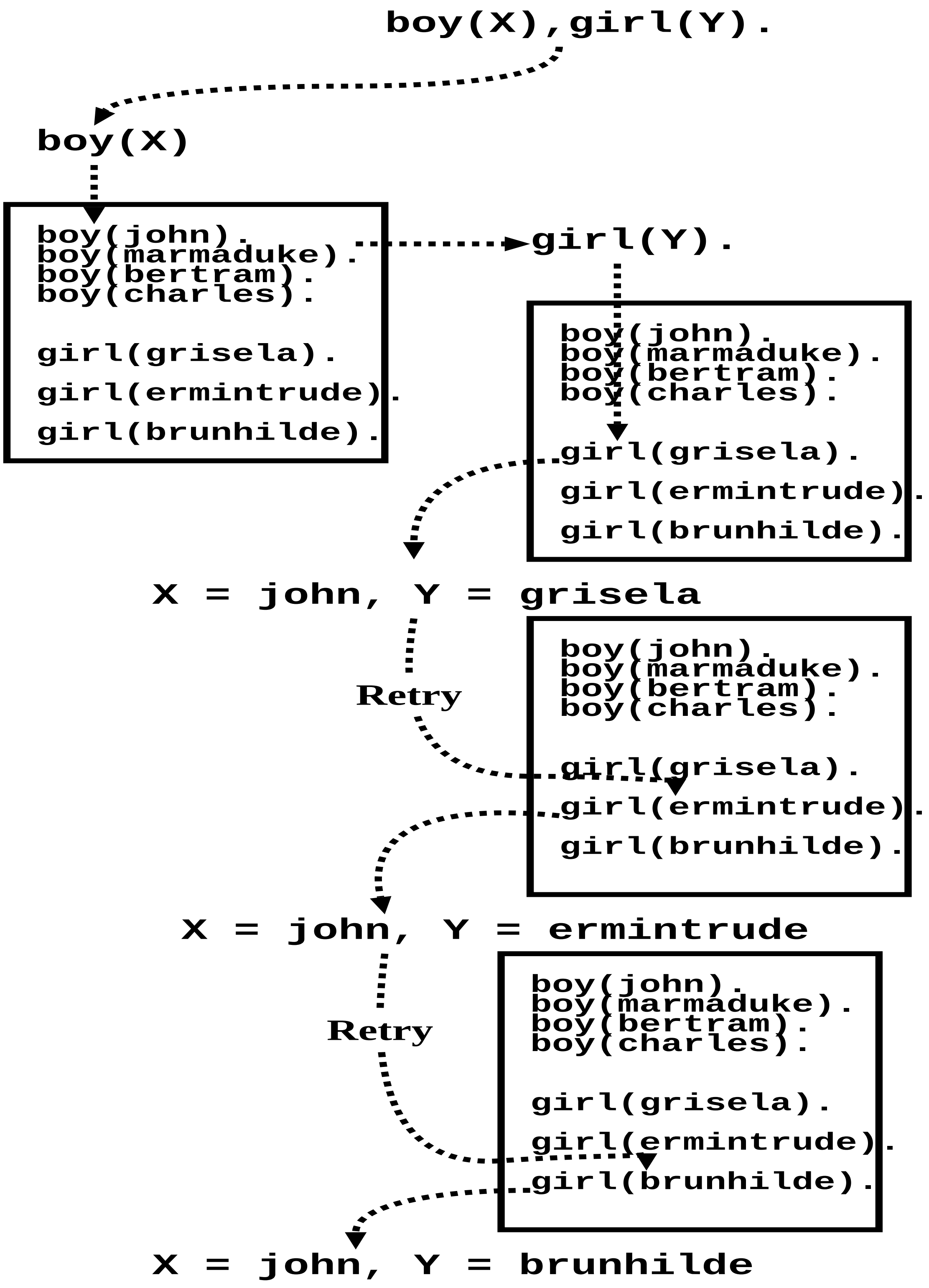
Conjunction

```
possible_pair(X,Y) :- boy(X), girl(Y).

boy(john).
boy(marmaduke).
boy(bertram).
boy(charles).

girl(grisela).
girl(ermintrude).
girl(brunhilde).
```

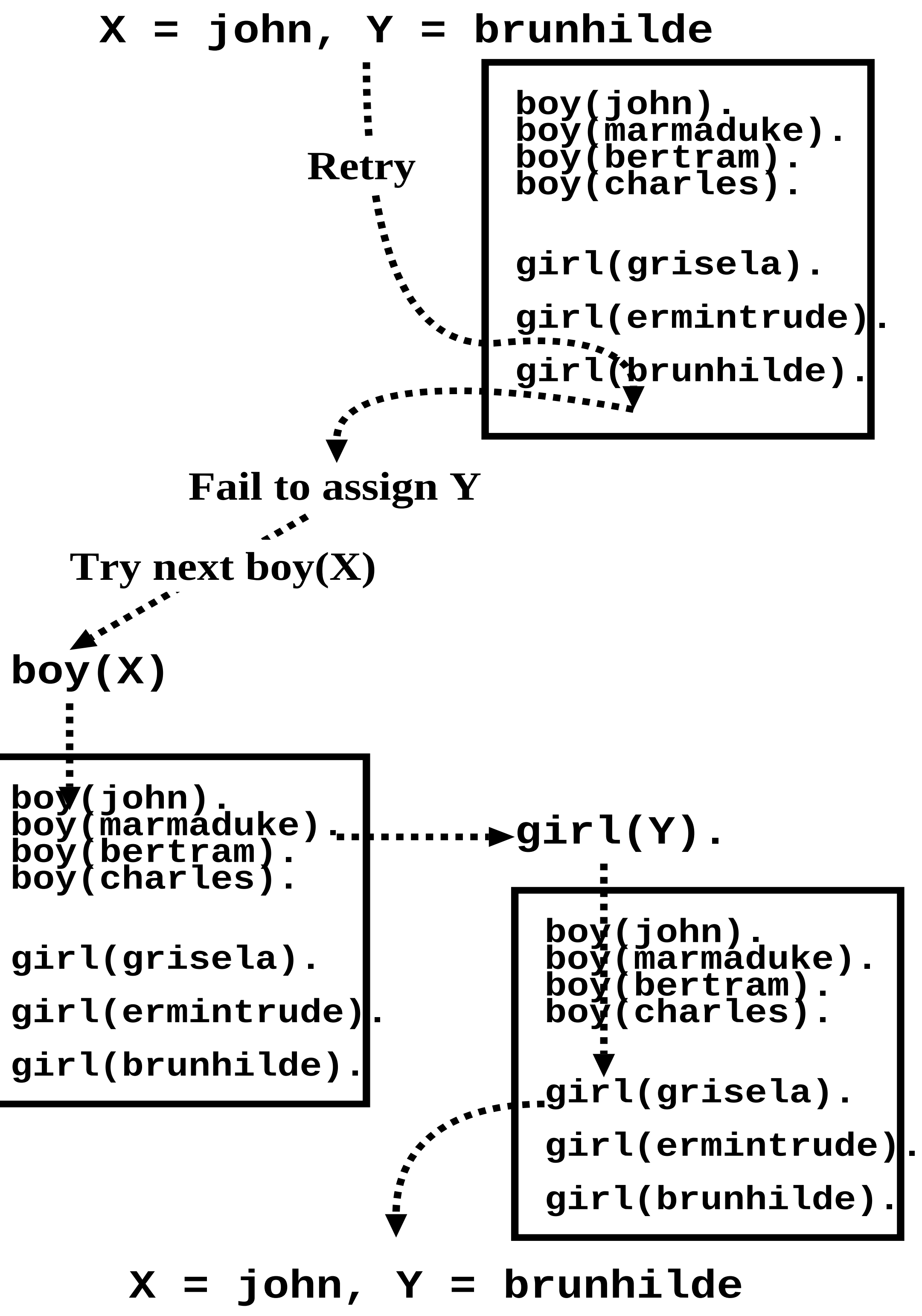
Conjunction Results

```
| ?- possible_pair(X,Y).

X = john, Y = grisela ? ;
X = john, Y = ermintrude ? ;
X = john, Y = brunhilde ? ;
X = marmaduke, Y = grisela ? ;
X = marmaduke, Y = ermintrude ? ;
X = marmaduke, Y = brunhilde ? ;
X = bertram, Y = grisela ? ;
X = bertram, Y = ermintrude ? ;
X = bertram, Y = brunhilde ? ;
X = charles, Y = grisela ? ;
X = charles, Y = ermintrude ? ;
X = charles, Y = brunhilde ? ;
```

Possible Pairs



Backtrack


X = john, Y = brunhilde
Retry
boy(john).
boy(marmaduke).
boy(bertram).
boy(charles).
girl(grisela).
girl(ermintrude).
girl(brunhilde).
Fail to assign Y
Try next boy(X)
boy(X)
boy(john).
boy(marmaduke).
boy(bertram).
boy(charles).
girl(grisela).
girl(ermintrude).
girl(brunhilde).
girl(Y).
boy(john).
boy(marmaduke).
boy(bertram).
boy(charles).
girl(grisela).
girl(ermintrude).
girl(brunhilde).
X = john, Y = brunhilde

Infinite

```
| ?- [user].
| is_integer(0).
| is_integer(X) :- is_integer(Y),
                   X is Y + 1.
| user consulted, 10 msec 368 bytes

yes
| ?- is_integer(X).

X = 0 ? ;
X = 1 ? ;
X = 2 ? ;
X = 3 ? ;
X = 4 ? ;
X = 5 ? ;
X = 6 ? ;
X = 7 ? ;
X = 8 ? ;
X = 9 ? ;
X = 10 ? ;
```

Infinite Lists

```
| ?- [user].
| member(X,[X|_]).
| member(X,[_,Y]) :- member(X,Y).
| user consulted, 20 msec 624 bytes
| ?- member(a,X).
X = [a|_A] ? ;
X = [_A,[a|_B]] ? ;
X = [_A,[_B,[a|_C]]] ? ;
X = [_A,[_B,[_C,[a|_D]]]] ? ;
X = [_A,[_B,[_C,[_D,[a|_E]]]]] ? ;
X = [_A,[_B,[_C,[_D,[_E,[a|_F]]]]]] ?


| ?- member(a,[a,b,r,a,c,a,d,a,b,r,a]).
yes
```

The Cut

`member(a,[a,b,r,a,c,a,d,a,b,r,a]).`

Can succeed 5 times

But the yes or no needs only to succeed once

Can tell PROLOG to discard choices
with
The Cut
predicate

The cut says
Upon backtracking
fail at this point
The result
Everything above the cut
will be matched
once and only once

Uses of Cut

Faster

The backtracking will not waste time attempting to satisfy goals that you know beforehand will never succeed.

Memory

Backtracking choices above the cut do not need to be saved

Example of Cut

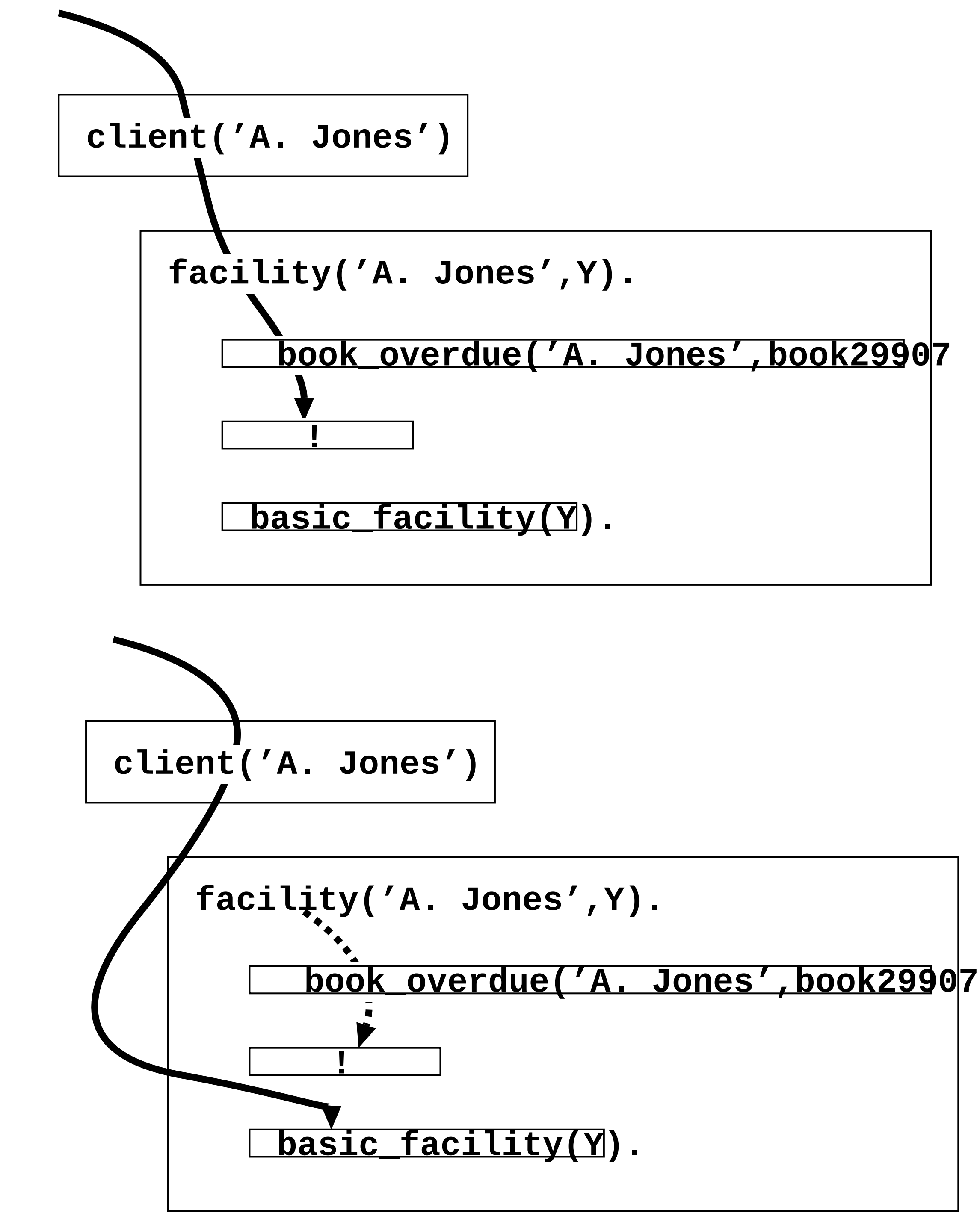
Reference Library Determine Which
Facilities are available

If one has an overdue book
Can only use the *basic facilities*

Otherwise
Can use the *general facilities*

```
facility(Pers,Fac) :-
    book˙overdue(Pers,Book),
    !,
    basic˙facility(Fac).

facility(Pers,Fac) :- general˙facility(Fac).

basic˙facility(reference).
basic˙facility(enquiries).

additional˙facility(borrowing).
additional˙facility(inter˙library˙loan).
general˙facility(X) :- basic˙facility(X).
general˙facility(X) :- additional˙facility(X).

book˙overdue('C. Watzer', book10089).
book˙overdue('A. Jones', book29907).
client('C. Watzer').
client('A. Jones').

client(X), facility(X,Y).
```

Effect


client('A. Jones')
facility('A. Jones',Y).
book_overdue('A. Jones',book29907
!
basic_facility(Y).
client('A. Jones')
facility('A. Jones',Y).
book_overdue('A. Jones',book29907
!
basic_facility(Y).

Example of Cut

```
| ?- client(X), facility(X,Y).

X = 'C. Watzer',
Y = reference ? ;

X = 'C. Watzer',Y = enquiries ? ;
X = 'A. Jones',Y = reference ? ;
X = 'A. Jones',Y = enquiries ? ;

no
```

Trace of Cut

```
— ?- client(X), facility(X,Y).
 + 1 1 Call: client(_61) ?
 + 1 1 Exit: client(C. Watzer) ?
 + 2 1 Call: facility(C. Watzer,_94) ?
 + 3 2 Call: book_overdue(C. Watzer,_712) ?
 + 3 2 Exit: book_overdue(C. Watzer,book10089) ?
 + 4 2 Call: basic_facility(_94) ?
 + 4 2 Exit: basic_facility(reference) ?
 + 2 1 Exit: facility(C. Watzer,reference) ?

X = 'C. Watzer',
Y = reference ? ;
 + 2 1 Redo: facility(C. Watzer,reference) ?
 + 4 2 Redo: basic_facility(reference) ?
 + 4 2 Exit: basic_facility(enquiries) ?
 + 2 1 Exit: facility(C. Watzer,enquiries) ?
X = 'C. Watzer',
Y = enquiries ? ;
```

```
+ 2 1 Redo: facility(C. Watzer,enquiries) ?
+ 4 2 Redo: basic_facility(enquiries) ?
+ 4 2 Fail: basic_facility(_94) ?
+ 2 1 Fail: facility(C. Watzer,_94) ?
+ 1 1 Redo: client(C. Watzer) ?
+ 1 1 Exit: client(A. Jones) ?
+ 2 1 Call: facility(A. Jones,_94) ?
+ 3 2 Call: book_overdue(A. Jones,_712) ?
+ 3 2 Exit: book_overdue(A. Jones,book29907) ?
+ 4 2 Call: basic_facility(_94) ?
+ 4 2 Exit: basic_facility(reference) ?
+ 2 1 Exit: facility(A. Jones,reference) ?

X = 'A. Jones',
Y = reference ?
```

Explanation

If a client is found to have an overdue book

Use of only basic facilities

After all the

`basic_facility(X)`

have been matched

The Cut

says to not to match

The predicates above the cut

`book_overdue`

The rest of the parent goal

`facility`

Common Uses

If you get this far
you have picked the correct rule
for this goal

If you get here
you should stop trying to satisfy this goal

If you get to here, you have found the only solution to this problem and there is no point in ever looking for alternatives

Sum of Numbers

```
| ?- [user].
| sum_to(1,1).
| sum_to(N,Res) :-
     N1 is N - 1,
     sum_to(N1,Res1),
     Res is Res1 + N.


| ?- sum_to(5,X).
```

Trace of sum_to

```
— ?- sum˙to(5,X).
 + 1  1  Call: sum˙to(5,˙75) ?
 + 2  2  Call: ˙345 is 5-1 ?
 + 2  2  Exit: 4 is 5-1 ?
 + 3  2  Call: sum˙to(4,˙337) ?
 + 4  3  Call: ˙958 is 4-1 ?
 + 4  3  Exit: 3 is 4-1 ?
 + 5  3  Call: sum˙to(3,˙950) ?
.
.
.
 + 5  3  Exit: sum˙to(3,6) ?
 + 12  3  Call: ˙337 is 6+4 ?
 + 12  3  Exit: 10 is 6+4 ?
 + 3  2  Exit: sum˙to(4,10) ?
 + 13  2  Call: ˙75 is 10+5 ?
 + 13  2  Exit: 15 is 10+5 ?
 + 1  1  Exit: sum˙to(5,15) ?
X = 15 ? ;
```

And Next?

```
X = 15 ? ;
 + 1  1  Redo: sum˙to(5,15) ?
 + 13  2  Redo: 15 is 10+5 ?
 + 13  2  Fail: ˙75 is 10+5 ?
 + 3  2  Redo: sum˙to(4,10) ?
 + 12  3  Redo: 10 is 6+4 ?
 + 12  3  Fail: ˙337 is 6+4 ?
 + 5  3  Redo: sum˙to(3,6) ?
 + 11  4  Redo: 6 is 3+3 ?
 + 11  4  Fail: ˙950 is 3+3 ?
 + 7  4  Redo: sum˙to(2,3) ?
 + 10  5  Redo: 3 is 1+2 ?
 + 10  5  Fail: ˙1563 is 1+2 ?
 + 9  5  Redo: sum˙to(1,1) ?
 + 10  6  Call: ˙2797 is 1-1 ?
 + 10  6  Exit: 0 is 1-1 ?
 + 11  6  Call: sum˙to(0,˙2789) ?
 + 12  7  Call: ˙3410 is 0-1 ?
 + 12  7  Exit: -1 is 0-1 ?
 + 13  7  Call: sum˙to(-1,˙3402) ?
```

What Happened?

The first solution was found
when the base case
`sum_to(1,1)`
was found

When looking for the next solution?

`sum_to(1,1)`
fails

Thus,
the second predicate can work
`sum_to(1,N)`

`N1` is now zero
Call `sum_to(0,Res1)`

But N less than zero will never match a predicate

Runs forever

Use of Cut

The first solution is the only solution

This means that
by this recursive definition
when the base case is reached
stop

```
| ?- sum_to(5,X).

 + 1  1  Call: sum_to(5,_75) ?
...
 + 1  1  Exit: sum_to(5,15) ?
X = 15 ? ;
 + 1  1  Redo: sum_to(5,15) ?
 + 13  2  Redo: 15 is 10+5 ?
 + 13  2  Fail: _75 is 10+5 ?
 + 3  2  Redo: sum_to(4,10) ?
 + 12  3  Redo: 10 is 6+4 ?
 + 12  3  Fail: _337 is 6+4 ?
 + 5  3  Redo: sum_to(3,6) ?
...
 + 5  3  Fail: sum_to(3,_950) ?
 + 4  3  Redo: 3 is 4-1 ?
 + 4  3  Fail: _958 is 4-1 ?
 + 3  2  Fail: sum_to(4,_337) ?
 + 2  2  Redo: 4 is 5-1 ?
 + 2  2  Fail: _345 is 5-1 ?
 + 1  1  Fail: sum_to(5,_75) ?
no
```

General

Two Cases

The number is 1
and otherwise

With Lists

The empty list `[]`
or `[A|B]`

Not Easy

The two cases

Equal to one

Not equal to one

Cannot match

Not Equal to One

Gave a rule with one
and then, if rule fails
(since PROLOG matches in order)
Then the other case will be tried

But with Backtracking
Reconsideration of `sum_of(1,1)`
Considers the second case, nevertheless
(even when equal to one).

The cut prevents this

Extra Conditions

Last Case

Cannot Distinguish between cases

More Usual

Cannot Specify Pattern
when extra cases are present
to decide which rule

Example

```
sum_to(N,1) :- N =< 1, !.
sum_to(N,R) :-
     N1 is N - 1,
     sum_to(N1,R1),
     R is R1 + N.
```

Better Formulation

```
sum_to(N,1) :- N =< 1, !.
```

Slightly Better
Produces an answer
(rather than running infinitely)
if less than or equal 0

Only if not true
is the second rule tried

The cut says
Don't try the second rule
if less than or equal to one

Remember
Encontering a cut
No more goal predicates are tried
(in this case: `sum_to`)

Equivalent: not

```
sum_to(1,1).
sum_to(N,R) :-
      not(N=1),
      N1 is N - 1,
      sum_to(N1,R1),
      R is R1 + N.
```

or

```
sum_to(N,1) :- N =< 1.
sum_to(N,R) :-
      not(N =< 1),
      N1 is N - 1,
      sum_to(N1,R1),
      R is R1 + N.
```

Equivalent: numeric

```
sum_to(1,1).
sum_to(N,R) :-
     N \= 1,
     N1 is N - 1,
     sum_to(N1,R1),
     R is R1 + N.

or

sum_to(N,1) :- N =< 1.
sum_to(N,R) :-
     N > 1,
     N1 is N - 1,
     sum_to(N1,R1),
     R is R1 + N.
```

Style

Better to use `not`
than cut.

The cut is hard to read and interpret

However

Use of `not`
says that the goal inside the `not` still has to be evaluated to check if can be satisfied

Could be inefficient

Double Evaluation

B has to be evaluated here

```
A :- B, C.
A :- not(B), D.
```

and here

Use of cut

```
A :- B, !, C.

A :- D.
```

Not evaluated twice,
but not as readable
(not logical)

Efficiency

```
append([],X,X).
append([A|B],C,[A|D]) :- append(B,C,D).

append([],X,X) :- !.
append([A|B],C,[A|D]) :- append(B,C,D).
```

The cut says
when the first solution is reached,
don't need to try the second

More efficient for the implementation
(better use of storage)

Cut-Fail Combo

`fail`

when reached,

the predicate will fail automatically (forces backtracking)

Average Taxpayer

should fail if foreigner

close, but not correct

```
average_taxpayer(X) :- foreigner(X),
    fail.
average_taxpayer(X) :- ...
```

In this case the second goal would be tried

Correct

```
average_taxpayer(X) :- foreigner(X), !,
    fail
```

Full Example

```
average_taxpayer(X) :-
     foreigner(X), !, fail.
average_taxpayer(X) :-
     spouse(X,Y),
     gross_income(Y,Inc),
     Inc > 3000,
     !, fail.
average_taxpayer(X) :-
     gross_income(X,Inc),
     Inc < 3000,
     Inc > 20000.
gross_income(X,Y) :-
     receives_pension(X,P),
     P < 5000,
     !, fail.
gross_income(X,Y) :-
     gross_salary(X,Z),
     investment_income(X,W),
     Y is Z+W.
investiment_income(X,Y) :- ...
```

With Not

```
average_taxpayer(X) :-
   not(foreigner(X)),
   not((spouse(X,Y),
        gross_income(Y,Inc),
        Inc>3000)),
   gros_income(X,Inc1),
  .....
```

Definition of not

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

`not(P)` succeeds

if and only if

`P` cannot be proven

closed world assumption

it is not always safe

to assume that

something is not true

if we are unable to prove it

Generate and Test

One of the simpliest AI search techniques

Generate

Generate all possible solutions to a problem

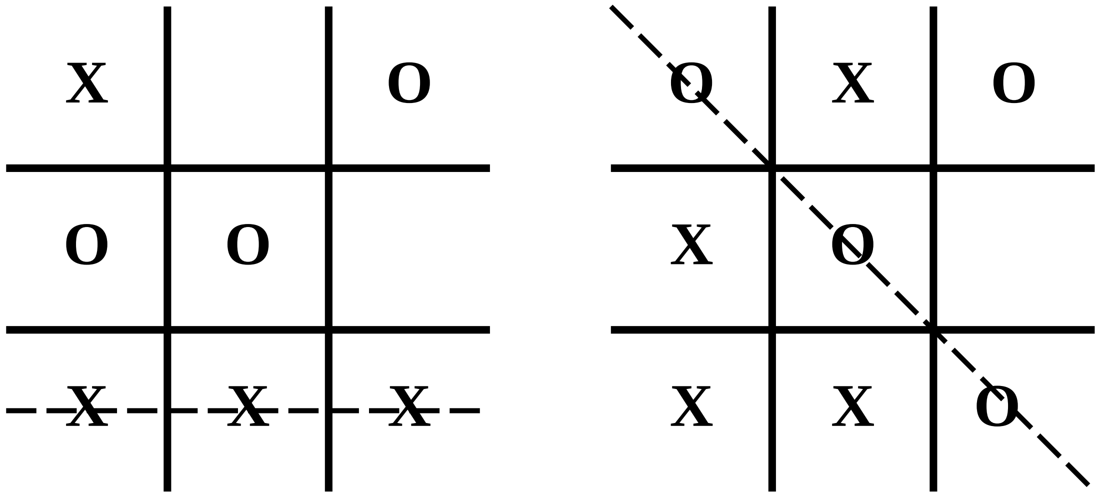
Test

Test each to see whether they are a solution

A possible solution
is generated
then tested
if the test succeeds
a solution is found
otherwise, backtrack to next possible solution

Tic-Tac-Toe

Object is to get three in a row



Representation

1	2	3
4	5	6
7	8	9

aline

Representation

1	2	3
4	5	6
7	8	9

O	O	O

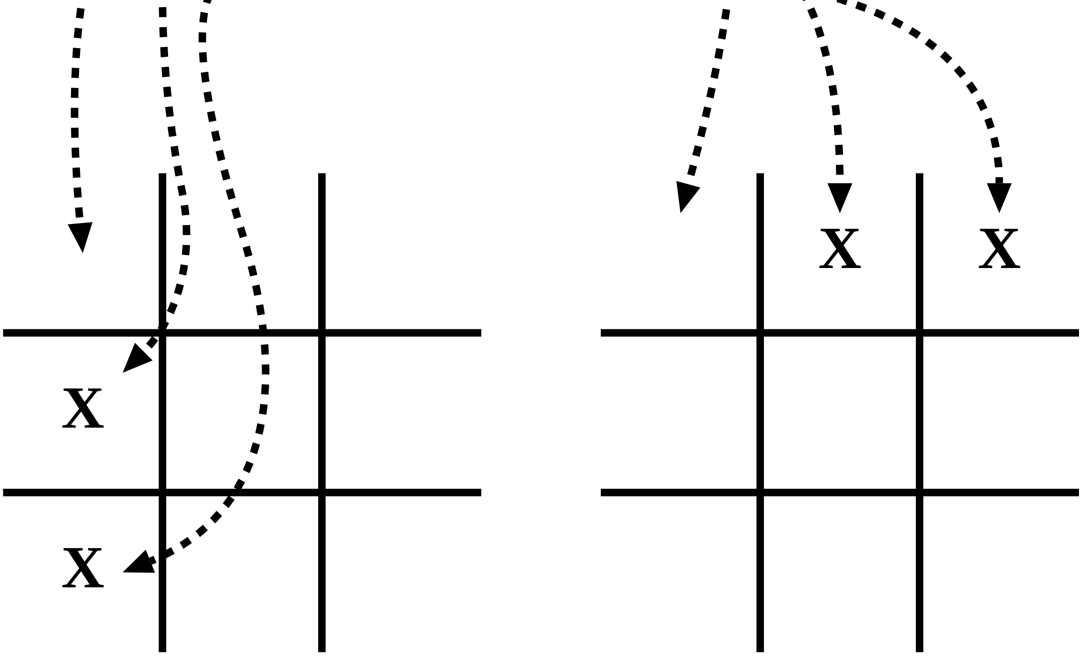
```
aline([4,5,6]).
```

		X
	X	
X		

```
aline([3,5,7]).
```

threatening

```
threatening([X,Y,Z],B,X) :-
        empty(X,B),
        cross(Y,B),
        cross(Z,B).
```



PROLOG

```
forced_move(Board,Sq) :-
   aline(Squares),
   threatening(Squares,Board,Sq),
   !.

aline([1,2,3]).
aline([4,5,6]).
aline([7,8,9]).
aline([1,4,7]).
aline([2,5,8]).
aline([3,6,9]).
aline([1,5,9]).
aline([3,5,7]).

threatening([X,Y,Z],B,X) :-
   empty(X,B), cross(Y,B), cross(Z,B).
threatening([X,Y,Z],B,X) :-
   empty(Y,B), cross(X,B), cross(Z,B).
threatening([X,Y,Z],B,X) :-
   empty(Z,B), cross(X,B), cross(Y,B).
```

forced_move

Moves Generated by

`alines`

all possible ways that X can win

Moves Tested by

`threatening`

If X can win in the next move

If no forced moves are found,
then the predicate fails
and some other predicate
would decide what move to make

Cut

Suppose embedded in a larger program

If `forced_move` successfully finds a move
then `Sq` becomes instantiated to the move

If, later, a failure occurs
(after this instantiation)
`forced_move` would retry

Can prevent PROLOG to search further
(which would be futile)
and not waste time
When I look for forced moves
it is only the first solution that is important

Problems with Cut

```
 append([],X,X) :- !.
 append([A|B],C,[A|D]) :- append(B,C,D).

| ?- append([a,b,c],[d,e],X).
X = [a,b,c,d,e] ? ;
no
| ?- append([a,b,c],X,Y).
Y = [a,b,c|X] ? ;
no
| ?- append(X,Y,[a,b,c]).
X = [],
Y = [a,b,c] ? ;
no
| ?-
```

Should be

```
append([],X,X).
append([A|B],C,[A|D]) :- append(B,C,D).



| ?- append(X,Y,[a,b,c]).
X = [],
Y = [a,b,c] ? ;

X = [a],
Y = [b,c] ? ;

X = [a,b],
Y = [c] ? ;

X = [a,b,c],
Y = [] ? ;
```

A Wrong Answer

```
number_of_parents(adam,0) :- !.
number_of_parents(eve,0) :- !.
number_of_parents(_,2).

| ?- number_of_parents(adam,X).
X = 0 ? ;
no
| ?- number_of_parents(eve,X).
X = 0 ? ;
no
| ?-  number_of_parents(john,X).
X = 2 ? ;
no
| ?- number_of_parents(eve,2).
```

Correct Implementations

```
number_of_parents(adam,N) :- !, N = 0.
number_of_parents(eve,N) :- !, N = 0.
number_of_parents(_,2).

number_of_parents(adam,0).
number_of_parents(eve,0).
number_of_parents(X,2) :-
      X \= adam,
      X \= eve.
```

Moral

If you introduce Cuts
to obtain correct behaviour
when goals are of one form

The is no guarentee
that anything sensible will happen
if goals of another form start appearing

Not a purely logical consequence