Lists In Prolog

- [a, b, 1, ‘hi’, [X, 2]]
- But really represented as compound terms
  - [ ] is an atom
  - [a, b, c] is represented as .(a, .(b, .(c, [])))
- Matching over lists

```
?- [X, 1, Z] = [a, _, 17]
X = a,
Z = 17.
```

List Deconstruction

- Syntactically similar to Ocaml: [H|T] like h::t

```
?- [Head | Tail] = [a,b,c].
Head = a,
Tail = [b, c].

?- [1,2,3,4] = [ _, X | _].
X = 2
```

- This is sufficient for defining complex predicates
- Let's define concat(L1, L2, C)

```
?- concat([a,b,c], [d,e,f], X).
X = [a,b,c,d,e,f].
```

Example: Concatenating Lists

- To program this, we define the "rules" of concatenation
  - If L1 is empty, then C = L2

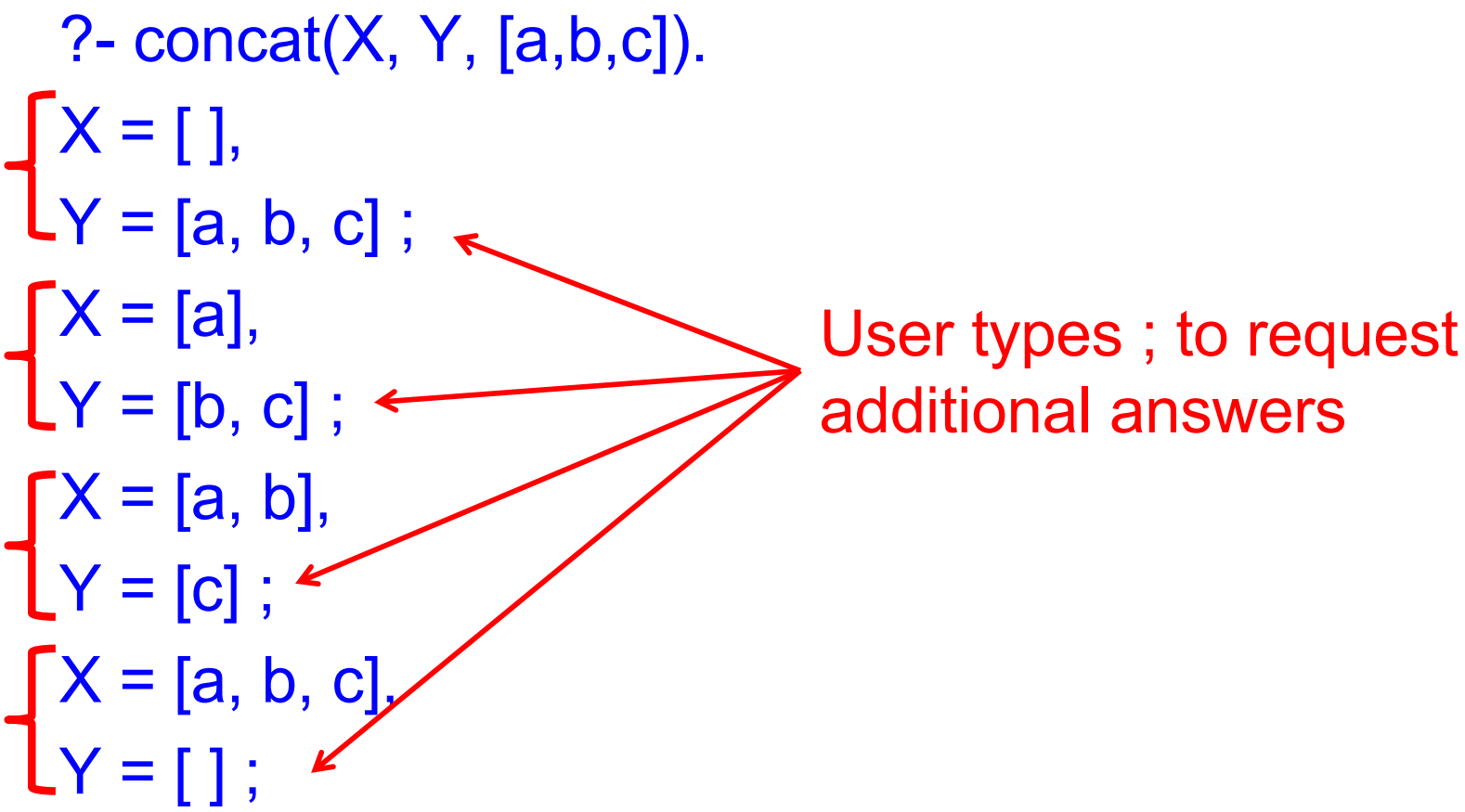
    concat([ ], L2, L2).
  - Prepending a new element to L1 prepends it to C, so long as C is the concatenation of L1 with some L2

```
concat( [E | L1], L2, [E | C] ) :-
    concat(L1, L2, C).
```

- … and we're done

Why Is The Return Value An Argument?

- Now we can ask what inputs lead to an output



More Syntax: Built-in Predicates

- Equality (a.k.a. unification)

  X = Y f(1,X,2) = f(Y,3,_)

- fail and true
- "Consulting" (loading) programs

  ?- consult('file.pl') ?- ['file.pl']

- Output/Input

  ?- write('Hello world'), nl ?- read(X).

- (Dynamic) type checking

  ?- atom(elephant) ?- atom(Elephant)

- help

The == Operator

- For identity comparisons
- X == Y
  - Returns true if and only if X and Y are identical
- Examples

```
?- 9 == 9.
true.

?- X == 9.
False.

?- X == X.
true.

?- 9 == 7+2.
false.

?- X == Y.
false.

?- 7+2 == 7+2.
true.
```

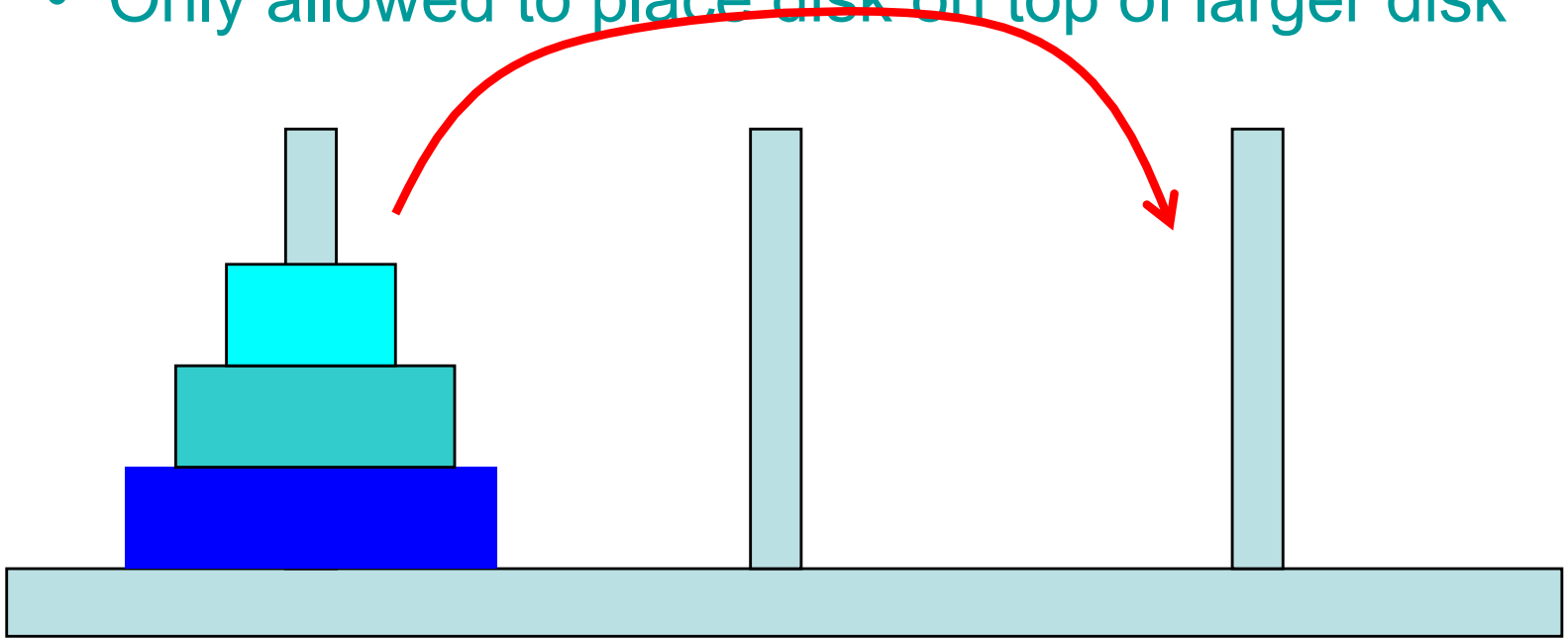
The =:= Operator

- For arithmetic operations
- "LHS =:= RHS"
  - Evaluate the LHS to value V1 (Error if not possible)
  - Evaluate the RHS to value V2 (Error if not possible)
  - Then match: V1 = V2
- Examples

```
?- 7+2 =:= 9.
true.

?- 7+2 =:= 3+6.
true.

?- X =:= 9.
Error: =:=/2: Arguments are not sufficiently instantiated

?- X =:= 7+2
```

Example – Towers of Hanoi

- Problem
  - Move stack of disks between pegs
  - Can only move top disk in stack
  - Only allowed to place disk on top of larger disk



Example – Towers of Hanoi

- To move a stack of n disks from peg X to Y
  - Base case
    - If n = 1, move disk from X to Y
  - Recursive step
    1. Move top n-1 disks from X to 3rd peg
    2. Move bottom disk from X to Y
    3. Move top n-1 disks from 3rd peg to Y

Iterative algorithm would take much longer to describe!

Towers of Hanoi

- Code

```
move(1,X,Y,_) :-
    write('Move top disk from '), write(X),
    write(' to '), write(Y), nl.
move(N,X,Y,Z) :-
    N>1,
    M is N-1,
    move(M,X,Z,Y),
    move(1,X,Y,_),
    move(M,Z,Y,X).
```

Prolog Terminology

- A query, goal, or term where variables do not occur is called ground; else it's nonground
  - foo(a,b) is ground; bar(X) is nonground
- A substitution θ is a partial map from variables to terms where domain(θ) ∩ range(θ) = ∅
  - Variables are terms, so a substitution can map variables to other variables, but not to themselves
- A is an instance of B if there is a substitution such that A = Bθ ← `The substitution θ applied to B`
- C is a common instance of A and B if it is an instance of A and an instance of B

Prolog's Algorithm Solve()

Starts as empty

Solve(goal G, program P, substitution θ) =

- Suppose G is $A_1,\ldots,A_n$. Choose goal A_1.
- For each clause A :- $B_1,B_2,\ldots,B_k$ in P,
  - if θ_1 is the mgu of A and $A_1\theta$ then
    - If Solve($\{B_1,\ldots,B_k,A_2,\ldots,A_n\}$, P, $\theta\cdot\theta_1$) = some θ' then **return** θ'
      - (else it has failed, so we continue the for loop)
  - (else unification has failed, so try another rule)
- If loop exits return fail
- Output: θ s.t. Gθ can be deduced from P, or fail

Chooses goals in order

Most General Unifier

Implements backtracking

! : a.k.a. “cut”

- When a ! is reached, it succeeds and commits Prolog to all the choices made since the parent goal was unified with the head of the clause the cut occurs in
  - Suppose we have clause C which is
    A :- B_1,…,B_k,!,…B_n.
  - If the current goal unifies with A, and B_1,…,B_k further succeed, the program is committed to the choice of C for the goal.
    - If any B_i for i > k fail, backtracking only goes as far as the cut.
    - If the cut is reached when backtracking, the goal fails

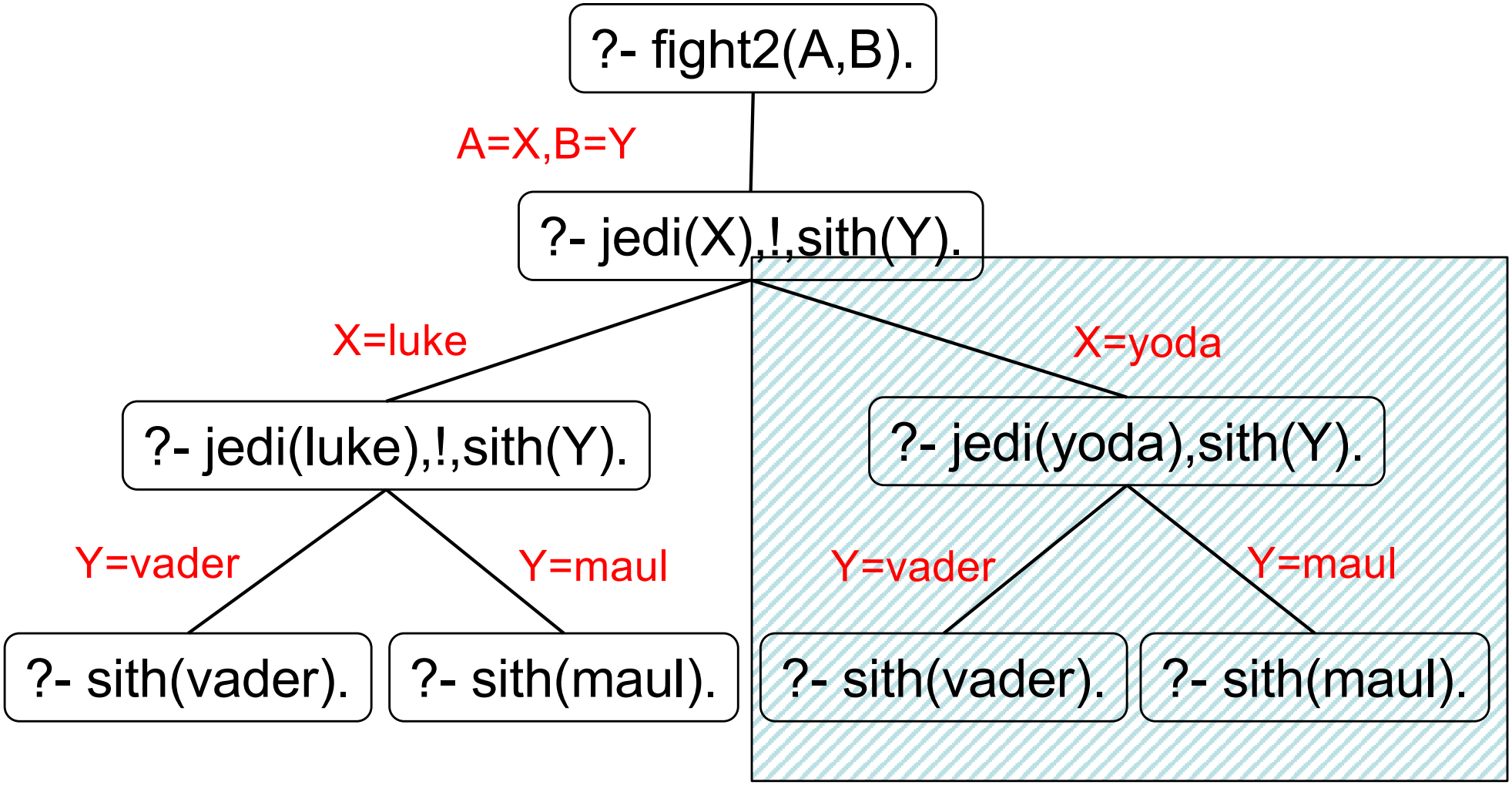
Cut

- Limits backtracking to predicates to right of cut
- Example

```
jedi(luke).
jedi(yoda).
sith(vader).
sith(maul).
fight2(X,Y) :- jedi(X), !, sith(Y).
fight3(X,Y) :- jedi(X), sith(Y), !.
```

```
?- fight2(A,B).
A=luke,
B=vader;
A=luke,
B=maul.
?- fight3(A,B).
A=luke,
B=vader.
```

Prolog Search Tree Limited By Cut



What Exactly Is Cut Doing?

Prunes all clauses below it

Prunes alternative solutions to its left

Does *not* affect the goals to its right

```
merge([X|Xs], [Y|Ys], [X|Zs]) :-
    X < Y, !, merge(Xs, [Y|Ys], Zs).

merge([X|Xs], [Y|Ys], [X,Y|Zs]) :-
    X =:= Y, !, merge(Xs,Ys,Zs).

merge([X|Xs], [Y|Ys], [Y|Zs]) :-
    X > Y, !, merge([X|Xs],Ys,Zs).

merge(Xs, [], Xs) :- !.

merge([], Ys, Ys) :- !.
```

Note: Cut only affects this call to merge. Does not affect backtracking of functions calling merge, or later recursive call to merge past cut

Why Use Cuts?

- Save time and space, or eliminate redundancy
  - Prune useless branches in the search tree
  - If sure these branches will not lead to solutions
  - These are green cuts
- Guide to the search to a different solution
  - Change the meaning of the program
  - Intentionally returning only subset of possible solutions
  - These are red cuts

Negation As Failure

- Cut may be used implement negation (not)
- Example

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

- If X succeeds, then the cut is reached, committing it; fail causes the whole thing to fail
- If X fails, then the second rule is reached, and the overall goal succeeds.
  - FYI, X here refers to an arbitrary goal
  - Effect of not depends crucially on rule order

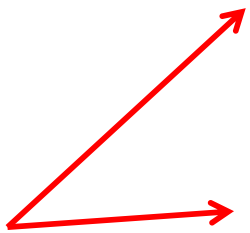
Not

- Not is tricky to use
  - Does not mean “not true”
  - Just means “not provable at this time”
- Example

```
jedi(luke).
jedi(vader).
sith(vader).
```

```
?- not(sith(luke)).
true.
?- not(sith(vader)).
false.
?- not(jedi(leia)).
true.
?- not(sith(leia)).
true.
```

Cannot prove either jedi(leia) or sith(leia) are true, so not( ) returns true



Not (cont.)

- Not is tricky to use
  - Does not mean "not true"
  - Just means "not provable at this time"
- Example

```
jedi(luke).
jedi(vader).
sith(vader).
```

```
?- not(sith(X)).
false.
```

Huh? Why not return X=luke?

Because not(sith(X)) does not mean "Can prove sith(X) is false for some X"

```
not(sith(X)) :- sith(X), !, fail.
not(sith(X)).
```

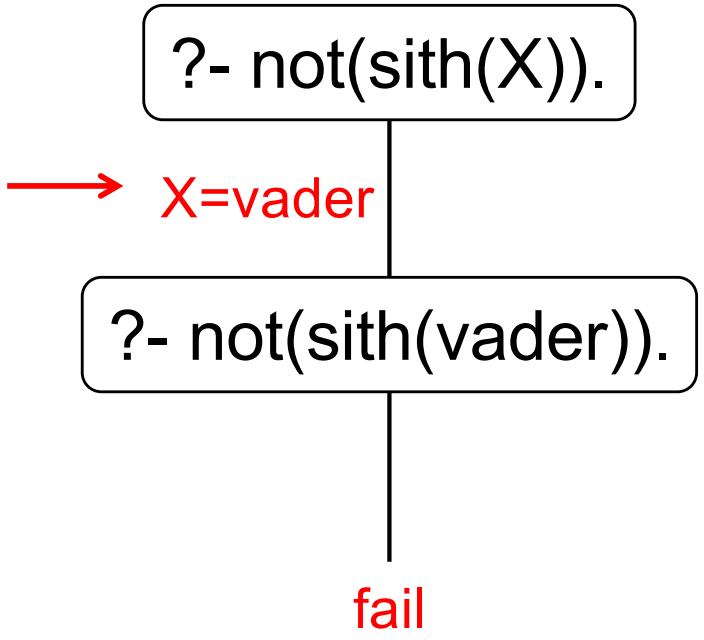
Instead, it means "Cannot prove sith(X) is true for some X". So X=vader causes not(sith(X)) to fail and return false

Not – Search Tree

```
jedi(luke).
jedi(vader).
sith(vader).
```

```
not(sith(X)) :- sith(X), !, fail.
not(sith(X)).
```

Will search for all X such that sith(X) is true.



Not (cont.)

- Ordering of clauses matters
- Example

```
jedi(luke).
jedi(vader).
sith(vader).
true_jedi1(X) :-
    jedi(X), not(sith(X)).
true_jedi2(X) :-
    not(sith(X)), jedi(X).
```

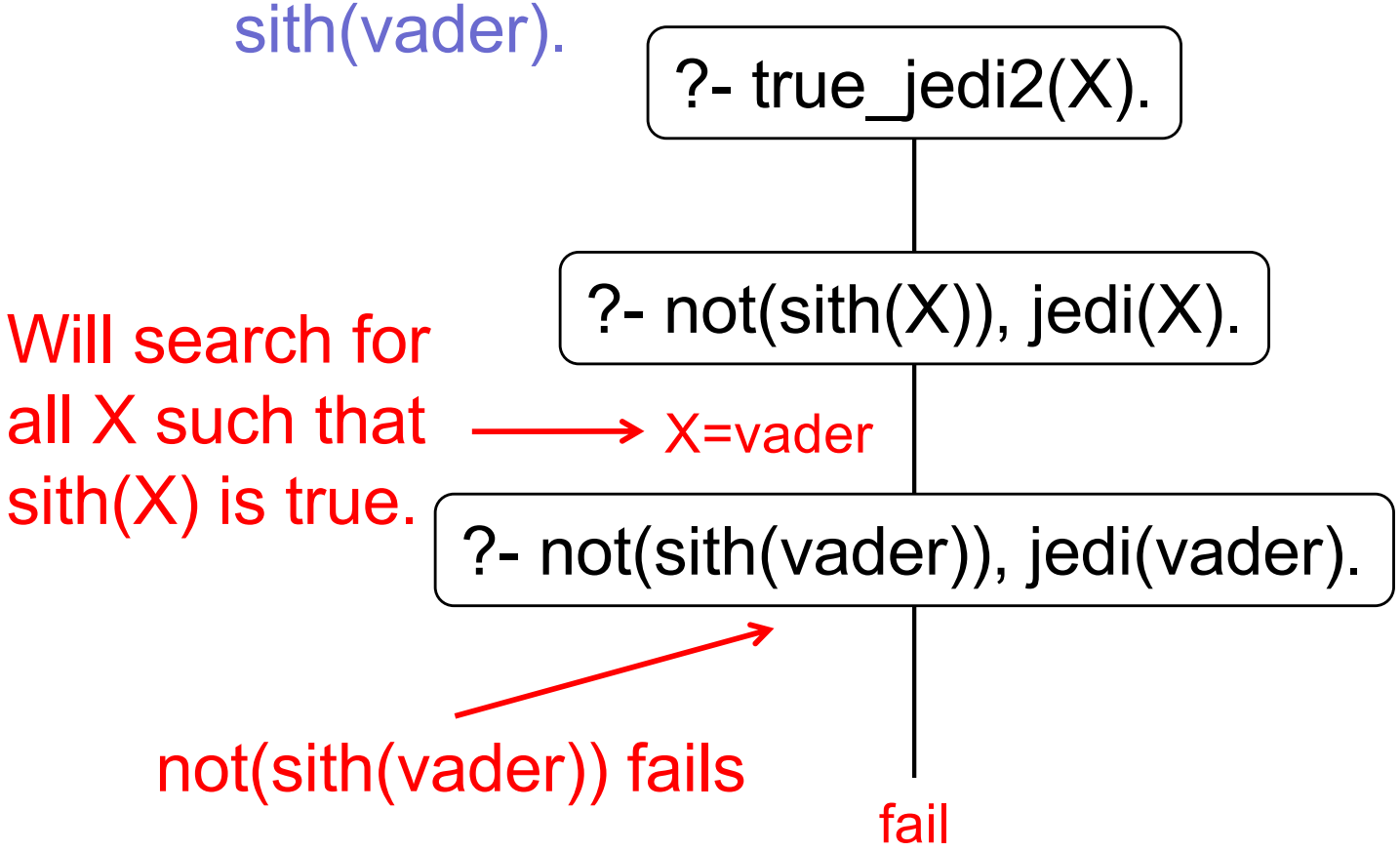
```
?- true_jedi1(luke).
true.
?- true_jedi1(X).
X=luke.
?- true_jedi2(luke).
true.
?- true_jedi2(X).
false.
```

X=vader causes not(sith(X)) to fail;
Will not backtrack to X=luke, since sith(luke) is not a fact

True_jedi2 – Search Tree

```
jedi(luke).
jedi(luke).
sith(vader).
```

```
not(sith(X)) :- sith(X), !, fail.
not(sith(X)).
```



Not and \=

- Built-in operators
  - \+ is not
  - X \= Y is same as not(X=Y)
  - X \== Y is same as not(X==Y)
- So be careful using \=
  - Ordering of clauses matters
  - Try to ensure operands of \= are instantiated

Example Using \=

- Example

```
jedi(luke).
jedi(yoda).
help2(X,Y) :- jedi(X), jedi(Y), X \= Y.
help3(X,Y) :- jedi(X), X \= Y, jedi(Y).
help4(X,Y) :- X \= Y, jedi(X), jedi(Y).
```

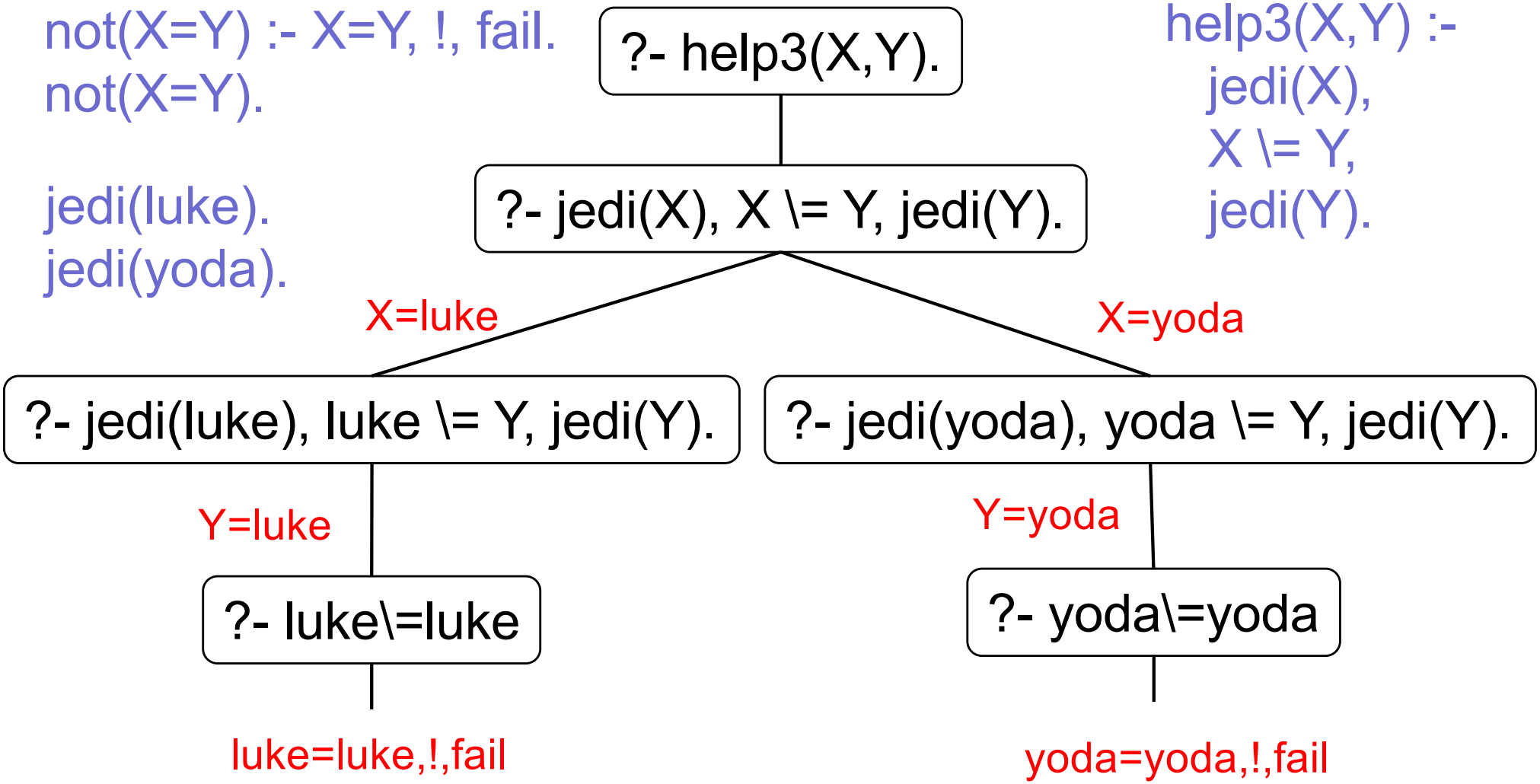
```
?- help2(X,Y).
X=luke,
Y=yoda;
X=yoda,
Y=luke.
?- help3(X,luke).
X=yoda.
?- help3(X,Y).
false.
```

After selecting X, can choose Y=X and fail X \= Y.

Help3 – Search Tree

```
not(X=Y) :- X=Y, !, fail.
not(X=Y).

jedi(luke).
jedi(yoda).
```

```
help3(X,Y) :-
    jedi(X),
    X \= Y,
    jedi(Y).
```



Using \=

- In fact, given X \= Y
  - will always fail if X or Y are not both instantiated

```
X \= a    // fails for X=a
a \= Y    // fails for Y=a
X \= Y    // fails for X=Y
```

Example Using \=

- Example

```
jedi(luke).
jedi(yoda).
help2(X,Y) :- jedi(X), jedi(Y), X \= Y.
help3(X,Y) :- jedi(X), X \= Y, jedi(Y).
help4(X,Y) :- X \= Y, jedi(X), jedi(Y).
```

```
?- help4(X,luke).
false.
?- help4(yoda,luke).
true.
```

Built-in List Predicates

- length(List,Length)

  ?- length([a, b, [1,2,3]], Length).
  Length = 3.

- member(Elem,List)

  ?- member(duey, [huey, duey, luey]).
  true.

  ?- member(X, [huey, duey, luey]).
  X = huey; X = duey; X = luey.

- append(List1,List2,Result)

  ?- append([duey], [huey, duey, luey], X).
  X = [duey, huey, duey, luey].

Built-in Predicates

- sort(List,SortedList)

  ?- sort([2,1,3], R).
  R= [1,2,3].

- findall(Elem,Predicate,ResultList)

  ?- findall(E,member(E,[huey, duey, luey]),R).
  R=[huey,duey,luey].

- setof(Elem,Predicate,ResultSortedList)

  ?- setof(E,member(E,[huey, duey, luey]),R).
  R=[duey,huey,luey].

- See documentation for more
  - http://www.swi-prolog.org/pldoc/man?section=builtin

Prolog Summary

- General purpose logic programming language
  - Associated with AI, computational linguistics
  - Also used for theorem proving, expert systems
- Declarative programming
  - Specify facts & relationships between facts (rules)
  - Run program as queries over these specifications
- Natural support for
  - Searching within set of constraints
  - Backtracking