CMSC 330: Organization of Programming Languages

Prolog Advanced Topic: Cut

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) $\cap$ range(θ) = $\varnothing$
  - Variables are terms, so a substitution can map variables to other variables, but circularly
- 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

Quick Quiz

- Which of these are ground terms?

jedi(luke)	ground
jedi(yoda)	ground
sith(X)	not ground

- Which of these is an instance of fight(A,B)?

jedi(luke)	no (heads don’t match)
fight(C,D)	yes, $\theta = \{ A \rightarrow C, B \rightarrow D \}$
fight(A,luke)	no (A → A not allowed)
fight(luke,yoda)	yes, $\theta = \{ A \rightarrow \text{luke}, B \rightarrow \text{yoda} \}$

Prolog’s Algorithm Solve()

```
Solve(goal $G$, program $P$, substitution $\theta$) =
  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 $\theta_1$ 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 $\theta'$ then return $\theta'$
      (else it has failed, so we continue the for loop)
    (else unification has failed, so try another rule)
  If loop exits return fail
  Output: $\theta$ s.t. $G\theta$ can be deduced from $P$, or fail
```

Starts as empty

Chooses goals in order

Most General Unifier

Implements backtracking

Example

```
on_vacation(mary).
on_vacation(peter).
has_money(peter).
travel(X) :- on_vacation(X),
  has_money(X).
```

?= travel(Y).

  - θ = X → Y1, Y → Y1
- on_vacation(X), has_money(X)
  - θ = X → Y1, Y → Y1, Y1 → mary
- has_money(X)
  - X = mary fails. Backtrack.
- on_vacation(X), has_money(X)
  - θ = X → Y1, Y → Y1, Y1 → peter
- has_money(X)
  - X = peter succeeds with
    θ = X → Y1, Y → Y1, Y1 → peter
  - Output is thus Y = peter.

Some Additional Built-in Predicates

- “Consulting” (loading) programs

  ?- consult(‘file.pl’) ?- [‘file.pl’] ?- [file]

- Output/Input

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

- (Dynamic) type checking

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

- help
- fail and true

! : 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 $A \text{ :- } B_1, \ldots, B_k, !, \ldots B_n$.
  - If the current goal unifies with A, and $B_1, \ldots, B_k$ 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

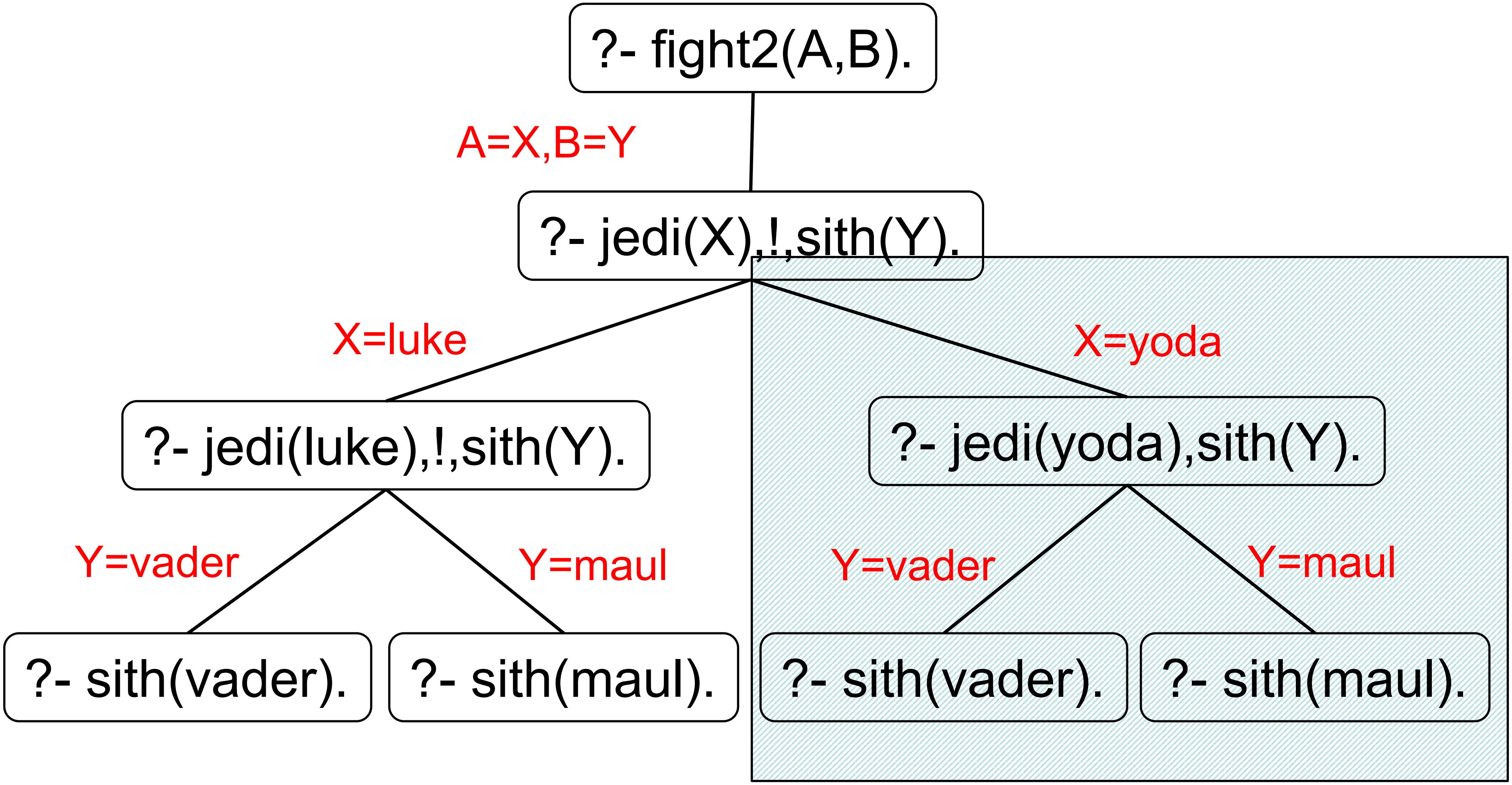
Cut

- 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

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

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) :- !.~~

Quiz 1: What does this query return?

```
s(c).
s(m).
s(d).
solve(X) :- s(X), !.
solve(other_solution).
?– solve(X).
```

A. X = c; X=m; X = d; X = other_solution.
B. X = c
C. X = c; X=m; X = d;
D. true

Quiz 1: What does this query return?

```
s(c).
s(m).
s(d).
solve(X) :- s(X), !.
solve(other_solution).
?– solve(X).
```

A. X = c; X=m; X = d; X = other_solution.
B. X = c
C. X = c; X=m; X = d;
D. true

Quiz 2: What does this query return?

```
check(_, []) :- !.
check(E, [H|T]) :- E > H, check(E, T).
 ?− check(10, [4, 3, 2]).
```

A. false.
B. true;   false.
C. true.
D. false;   true.

Quiz 2: What does this query return?

```
check(_, []) :- !.
check(E, [H|T]) :- E > H, check(E, T).
 ?– check(10, [4, 3, 2]).
```

A. false.
B. true;   false.
C. true.
D. false;   true.

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 the search to a different solution
  - Change the meaning of the program
    - Intentionally returning only subset of possible solutions
  - These are **red cuts**

Quiz 3: Is this a green or red cut?

```
s(c).
s(m).
s(d).
solve(X) :- s(X), !.
solve(other_solution).
?– solve(X).
```

A. Green
B. Red

Quiz 3: Is this a green or red cut?

```
s(c).
s(m).
s(d).
solve(X) :- s(X), !.
solve(other_solution).
?– solve(X).
```

A. Green

B. Red

Quiz 4: Is this a green or red cut?

```
check(_, []) :- !.
check(E, [H|T]) :- E > H, !, check(E, T).
 ?– check(10, [4, 3, 2]).
```

A. Green
B. Red

Quiz 4: Is this a green or red cut?

```
check(_, []) :- !.
check(E, [H|T]) :- E > H, !, check(E, T).
 ?– check(10, [4, 3, 2]).
```

A. Green
B. Red

Quiz 5: Is this a green or red cut?

```
if_then_else(P,Q,_) :- P, !, Q.
if_then_else(_,_,R) :- R.
```

A. Green
B. Red

Quiz 5: Is this a green or red cut?

```
if_then_else(P,Q,_) :- P, !, Q.
if_then_else(_,_,R) :- R.
```

A. Green

B. Red

Negation As Failure

- (Red) cut used to 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(X) :- X, !, fail.
not(X).
```

- Not is tricky to use
  - Does not mean “not true”
  - Just means “not provable”
- 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(X) :- X, !, fail.
not(X).
```

- Not is tricky to use
  - Does not mean “not true”
  - Just means “not provable”
- 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”

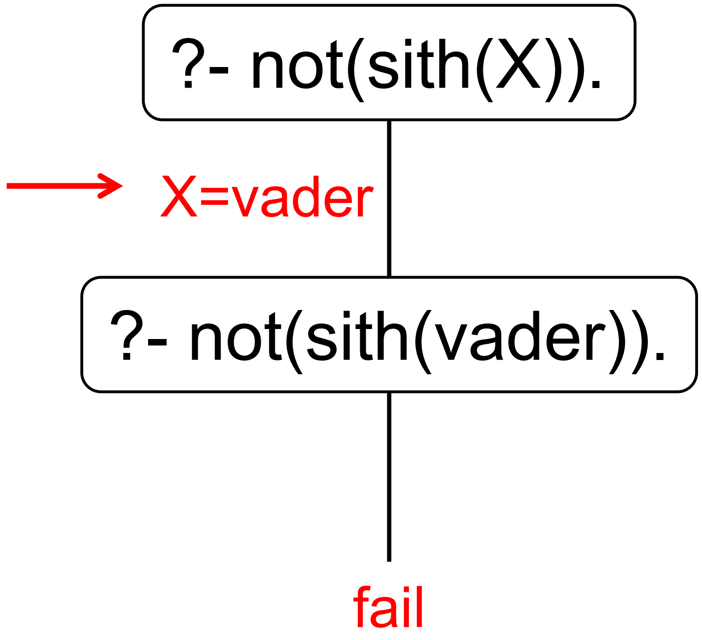
```
not(sith(X)) :- sith(X), !, fail.
not(sith(X)).
```

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

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



?- not(sith(X)).

X=vader

?- not(sith(vader)).

fail

Not (cont.)

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

- 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).
```

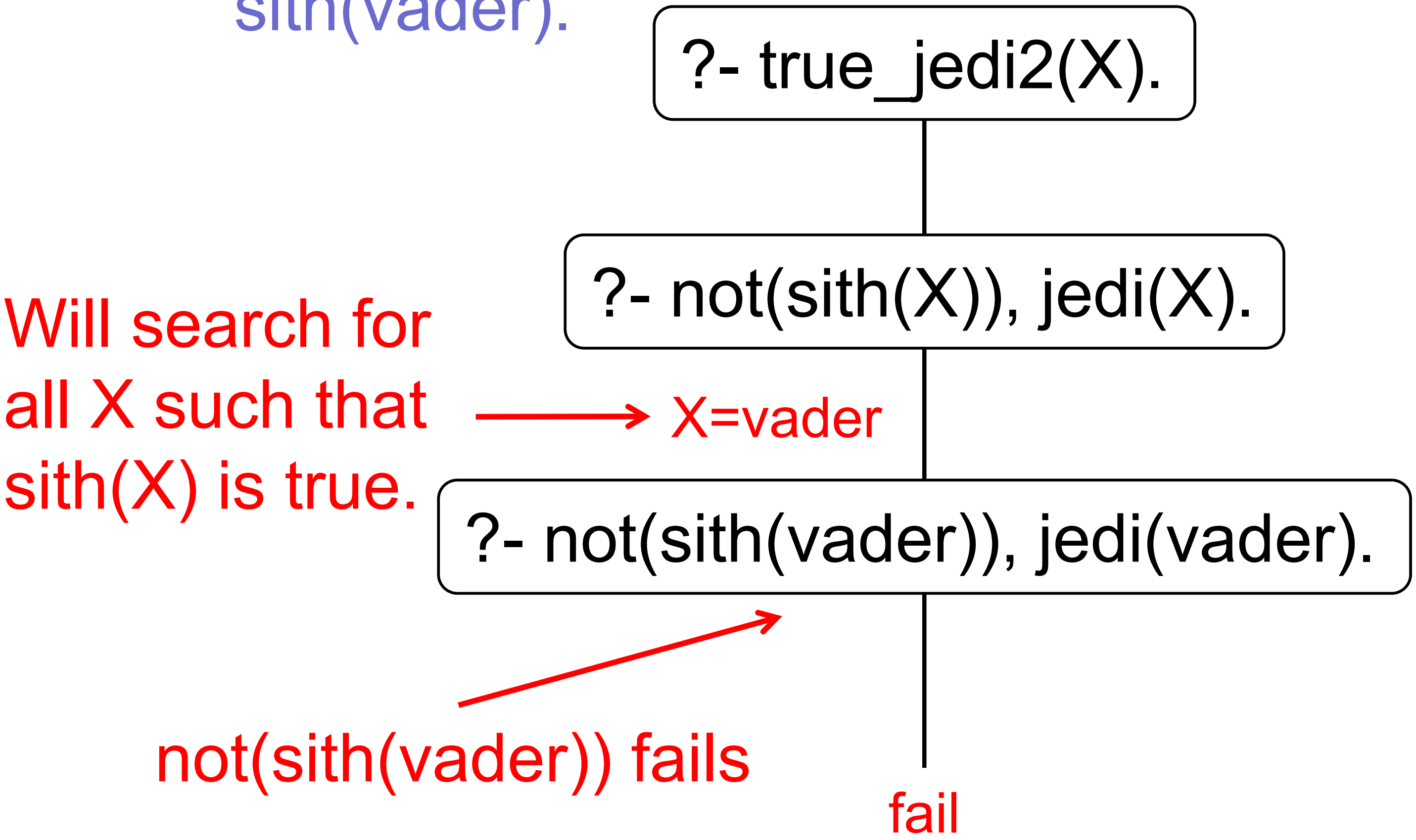
```
?- true_jedi1(luke).
true.
?- true_jedi1(X).
X=luke.
?- true_jedi2(luke).
true.
?- true_jedi2(X).
false.
```

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)).
```



?- true_jedi2(X).

?- not(sith(X)), jedi(X).

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

X=vader

?- not(sith(vader)), jedi(vader).

not(sith(vader)) fails

fail

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 \=

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

- 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.
```

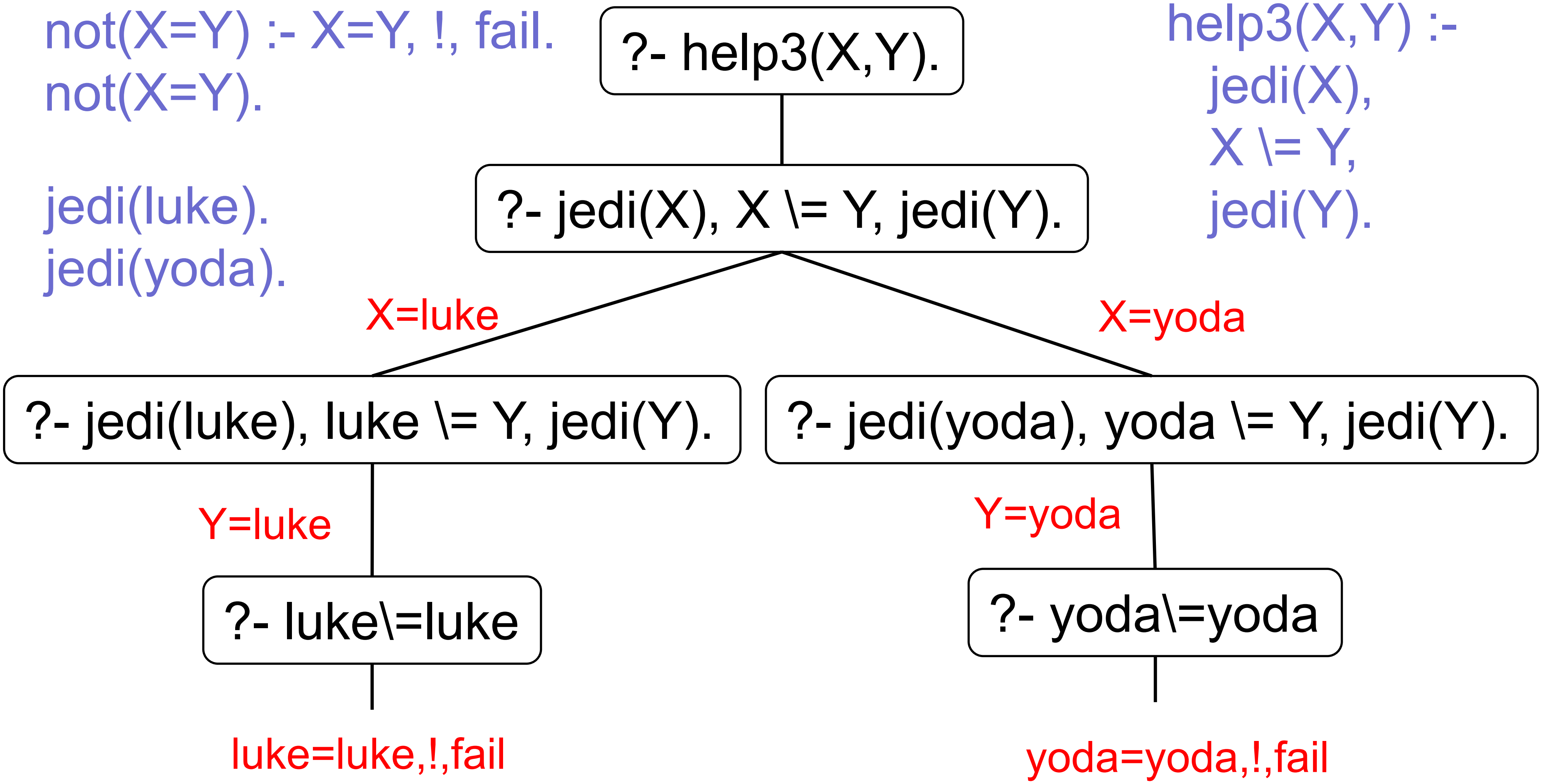
After selecting X, can choose Y=X and fail X \= Y.

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).
```



?- help3(X,Y).

?- jedi(X), X \= Y, jedi(Y).

X=luke

?- jedi(luke), luke \= Y, jedi(Y).

Y=luke

?- luke\=luke

luke=luke,!,fail

X=yoda

?- jedi(yoda), yoda \= Y, jedi(Y).

Y=yoda

?- yoda\=yoda

yoda=yoda,!,fail

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.
```

Quiz 6: What does this query return?

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

A. X = luke
B. false
C. true
D. X = vader

Quiz 6: What does this query return?

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

A. X = luke
B. false
C. true
D. X = vader

Quiz 7: What does this query return?

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

A. X = vader
B. X = luke
C. false
D. true

Quiz 7: What does this query return?

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

A. X = vader
B. X = luke
C. false
D. true

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