

Prolog

References:

SWI Prolog: www.swi-prolog.org/

Learn Prolog Now (a tutorial)

<http://www.coli.uni-saarland.de/~kris/learn-prolog-now/html/>

<http://www.cs.nuim.ie/~jpower/Courses/PROLOG/>

Syntactic Elements of Prolog

- Facts, Rules and Queries

- built from *terms*.

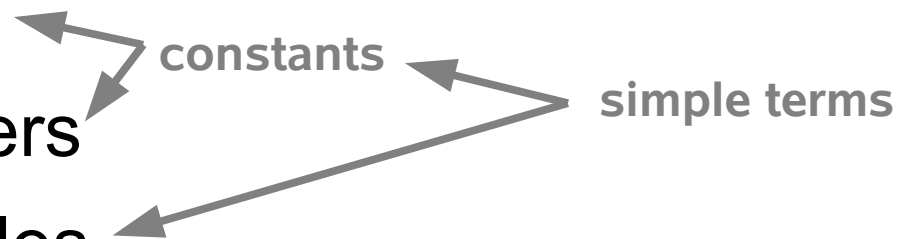
- Terms:

- atoms

- numbers

- variables

- complex terms



atoms can be any of:

- starts with lower case letter.
 - followed possibly by a sequence of letters, digits and/or underscore _.

joe dave_h apache2 wumpus

- any arbitrary sequence of characters enclosed by single quotes:

'Fred' '*&@ Joe'

- a sequence of special characters:

; :- @= =====>

Numbers

- Numeric constants (literals)

1

-3

3.2

1000000

Variables

- Start with either upper case letter, or underscore `_`.
 - followed by letters, digits, underscore.

X

Y

`_input`

Output

Complex Terms: Predicates

- The name of a predicate is called a *functor*.
 - functor is an atom
 - starts with lowercase letter.
- Each predicate has an *airity*
 - the number of arguments.
 - /1 means one argument
 - /2 means two arguments
 - /7 means 277 arguments, ...

You can have two predicates with the same functor (name), but different arity.

Predicate Arguments

- The arguments to a predicate can be any kind of term
 - atom, number, variable or another functor
- Example Predicates:

```
father(X, Y)
```

```
father(joe, sam)
```

```
playsAirGuitar(mia)
```

KB: Facts and Rules

- A KnowledgeBase is composed of Facts and Rules.
- Each fact is unconditionally true.
- Example Facts:

woman (mia) .

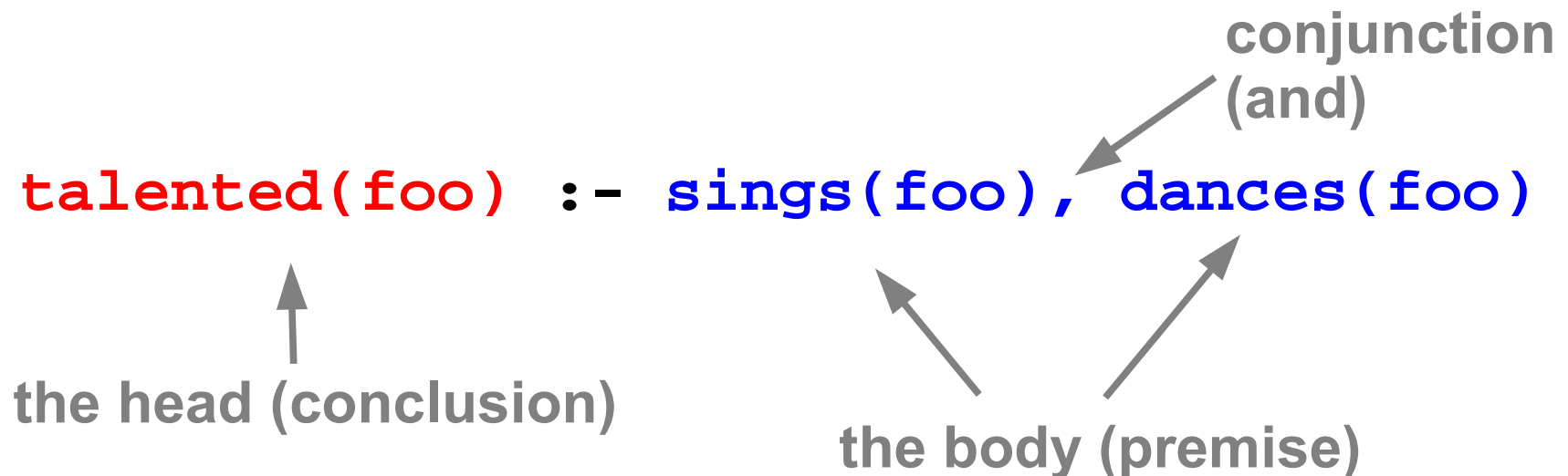
woman (jody) .

woman (yolanda) .

playsAirGuitar (jody) .

Rules

- Each rule states something that is conditionally true.
 - an implication.
- Each rule has a head (the conclusion) and a body (the premise of the implication).



Rules and Modus Ponens

```
talented(foo) :- sings(foo), dances(foo)
```

The rule states that if `sings(foo)` is true and `dances(foo)` is true, then `talented(foo)` is also true.

This is inferred knowledge.

Deductive Inference using modus ponens

Simple Prolog KB and Queries

- You can put a collection of facts and rules in a file and tell prolog to use that as the KB.
- You can now issue queries that are checked against the KB.
 - If prolog can prove that the query is true based on the KB it will tell you the query is true.
 - If prolog can't prove the query is true, it will tell you the query is false.
 - *false* means that either:
 - the KB contradicts the query
 - the KB doesn't have enough information to know whether it's true or not.

Queries

- At the prompt (`?-`) you type in a fact
 - make sure there is a period at the end!
- Prolog says "Yes" or "No"
- Examples:

```
?- easyTest(ai) .
```

```
no
```

```
?- minMaxAlphaBetaQuestion(test1) .
```

```
yes
```

Simple KB

```
happy (vincent) .
```

```
listensToMusic (butch) .
```

```
playsAirGuitar (vincent) :-  
    listensToMusic (vincent) ,  
    happy (vincent) .
```

```
playsAirGuitar (butch) :-  
    happy (butch) .
```

```
playsAirGuitar (butch) :-  
    listensToMusic (butch) .
```

Could also be:

```
playsAirGuitar (butch) :-  
    happy (butch) ;  
    listensToMusic (butch) .
```

Simple KB Queries

```
happy(vincent) .  
listensToMusic(butch) .  
playsAirGuitar(vincent) :- listensToMusic(vincent), happy(vincent) .  
playsAirGuitar(butch) :- happy(butch) .  
playsAirGuitar(butch) :- listensToMusic(butch) .
```

?- playsAirGuitar(vincent) .

no ← no means that this fact cannot be
inferred given the KB.

?- playsAirGuitar(butch) .

yes ← yes means that this fact is a logical
conclusion of the rules and facts in the KB.

Another KB and Query

```
loves (vincent, mia) .  
loves (marcellus, mia) .  
loves (pumpkin, honey_bunny) .  
loves (honey_bunny, pumpkin) .  
jealous (X, Y) :- loves (X, Z), loves (Y, Z) .
```

```
?- jealous (marcellus, W) .
```

```
W = vincent
```

Derivation

`jealous (marcellus ,W) .`

`jealous (X,Y) :- loves (X,Z) , loves (Y,Z) .`

`X = marcellus, Y=W`

`jealous (marcellus ,W) :- loves (marcellus ,Z) , loves (W,Z) .`

`Z= mia`

`jealous (marcellus ,W) :- loves (marcellus ,mia) , loves (W,mia) .`

`W= vincent`

`jealous (marcellus ,vincent) :-
loves (marcellus ,mia) , loves (vincent ,mia) .`

SWI Prolog

Command Line interactions.

To load a KB (stored in the file `foo.pl`)

```
?- ['foo.pl'].
```

```
% foo.pl compiled 0.00 sec, 0 bytes
```

```
Yes
```

Issue a query:

```
?- jealous(marcellus,W).
```

```
W = vincent
```

Matching

- Prolog attempts to prove each query by matching rules and facts.
 - which are made of terms.
- Two terms match:
 - if they are the same
 - or-
 - if they contain variables that can be instantiated in such a way that the resulting terms are the same.

More precise definition

- Two numbers match only if they are the same number.
- Two atoms match only if they are the same atom.
- Two variables match
 - replace one variable with the other
- A variable and a constant match
 - replace the variable with the constant.
- Complex terms match
 - if the functors match and
 - arity is the same
 - all arguments match
 - consistent matching of arguments

Matching Example

$f(a).$

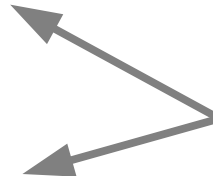
$f(b).$

$g(a).$

$g(b).$

$h(b).$

$k(X) :- f(X), g(X), h(X).$

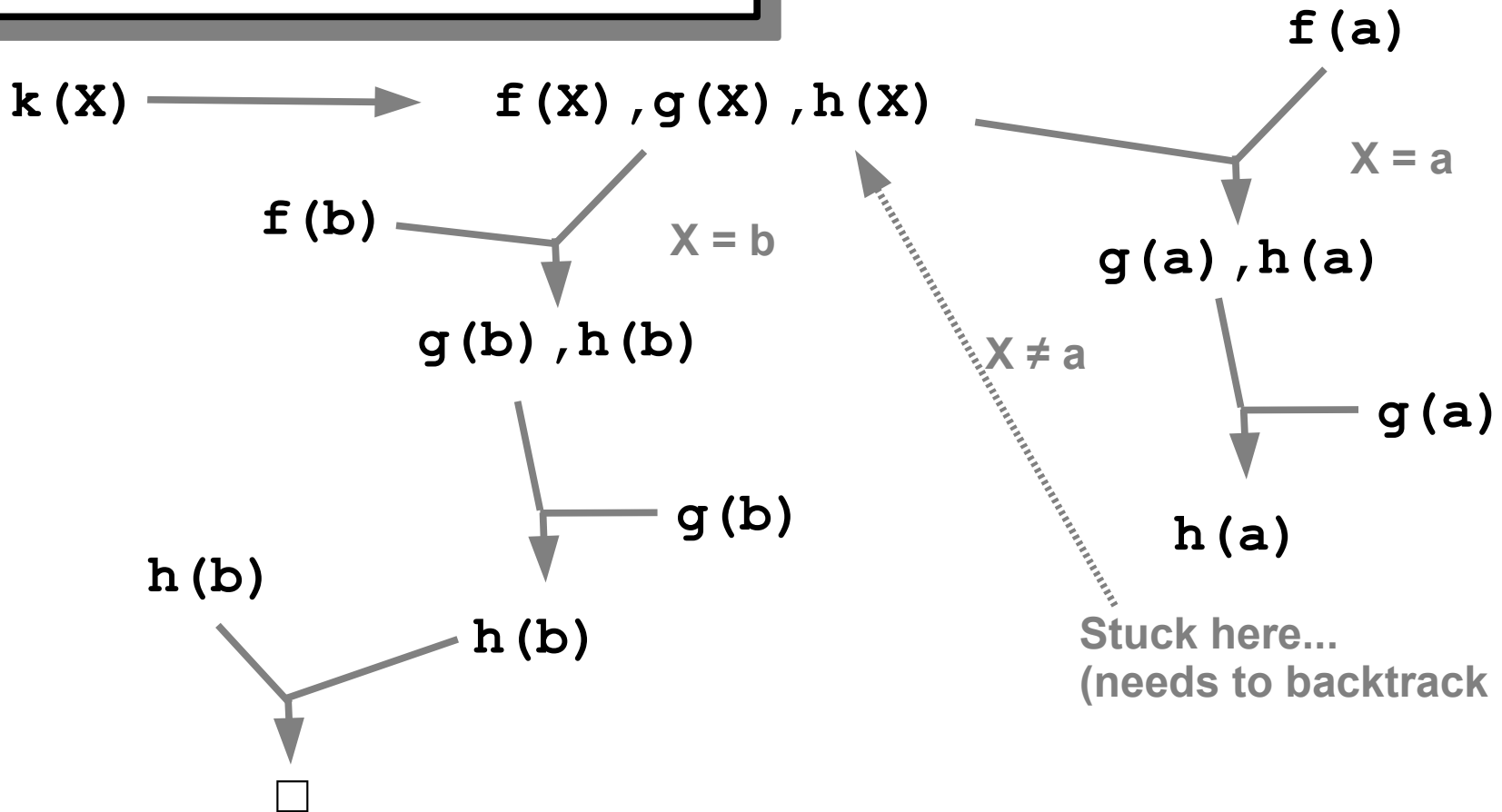
$?- k(X).$  **To prove $k(X)$, we need to prove $f(x), g(x), h(x)$ for some value of x .**

```

f(a) .
f(b) .
g(a) .
g(b) .
h(b) .
k(X) :- f(X), g(X), h(X) .

```

Matching and Backtracking



Asking Prolog to match

- You can ask prolog to do some matching for you using =:

?- a = b.

No

?- a = X.

X = a

Yes

?- foo(a) = foo(Z) .

Z = a

Yes

Using Trace

?- trace.

Yes

[trace] ?- k(X).

Call: (7) k(_G312) ?

Call: (8) f(_G312) ?

Exit: (8) f(a) ?

Call: (8) g(a) ?

Exit: (8) g(a) ?

Call: (8) h(a) ?

Fail: (8) h(a) ?

Redo: (8) f(_G312) ?

...

```
f(a) .
```

```
f(b) .
```

```
g(a) .
```

```
g(b) .
```

```
h(b) .
```

```
k(X) :- f(X), g(X), h(X) .
```

new (unique) variable name

subgoal

_G312 matches a, f(a) is true

new subgoal

g(a) is true

new subgoal

stuck (failed)

backtrack

Cut

- Sometimes it is important to tell prolog not to backtrack.
 - perhaps finding one solution (variable assignments) is all we want.
- The ! symbol (atom) tells prolog not to backtrack.
- Cut can be used in goals and in rules.
- Cut means "stop once you find one satisfying assignment".
 - otherwise prolog will attempt to find all satisfying assignments.

Simple Cut Example

KB:
p(a) .
p(b) .
p(c) .

?- p(x) .
x = a ;
x = b ;
x = c ;
No
?-

?- p(x) , !
x = a ;
No
?-

Another Cut Example

```
?- r(X),s(Y) .
```

```
X = a  
Y = a ;
```

```
X = a  
Y = b ;
```

```
X = a  
Y = c ;
```

```
X = b  
Y = a ;
```

```
...
```

```
?- r(X),s(Y),!.  
.
```

```
X = a  
Y = a ;
```

```
No  
?-
```

KB:

```
r(a) .
```

```
r(b) .
```

```
s(a) .
```

```
s(b) .
```

```
s(c) .
```

More interesting example

```
?- r(X), !, s(Y) .
```

```
X = a
```

```
Y = a ;
```

```
X = a
```

```
Y = b ;
```

```
X = a
```

```
Y = c ;
```

```
No
```

```
?-
```

KB:

```
r(a) .
```

```
r(b) .
```

```
s(a) .
```

```
s(b) .
```

```
s(c) .
```

Cut is dangerous

- Depending on how you use it, cut can change the meaning of a program.
 - generally we only want to use it to improve efficiency (to avoid unnecessary searching for satisfying assignments of variables).
- Example: a max predicate:
- $\text{max}(X, Y, Y) \text{ :- } Y \geq X.$
- $\text{max}(X, Y, X) \text{ :- } X > Y.$

Cut Example: a Max predicate.

max(X, Y, Y) :- Y >= X.

max(X, Y, X) :- X > Y.

?- max(1, 2, Z).

Z = 2

Yes

max(X, Y, Y) :- Y >= X, !.

max(X, Y, X).

?- max(1, 2, 1).

yes

max(X, Y, Y) :- Y >= X, !.

max(X, Y, X) :- X > Y.

?- max(1, 2, 1).

no

Close Inspection of Towers of Hanoi program

```
hanoi(N) :- move(N, left, right, center).
```

```
move(0, _, _, _) :- !.
```

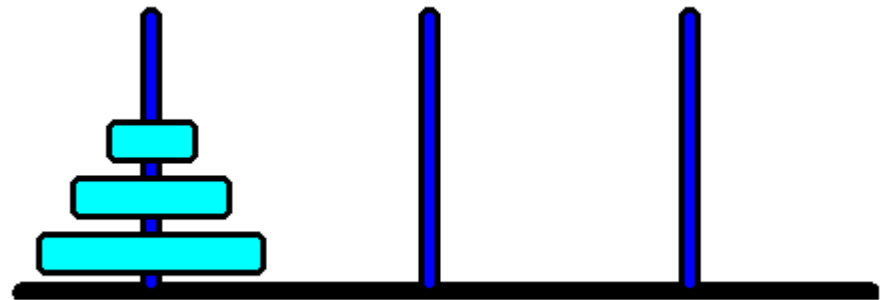
```
move(N, A, B, C) :-
```

```
    M is N-1,
```

```
    move(M, A, C, B),
```

```
    inform(A, B),
```

```
    move(M, C, B, A).
```



```
inform(X, Y) :-
```

```
    write('move a disc from the '),
```

```
    write(X),
```

```
    write(' pole to the '),
```

```
    write(Y),
```

```
    write(' pole'),
```

```
    nl.
```

Rule for `hanoi(N)`

```
hanoi(N) :- move(N, left, right, center).
```

- If prolog finds that it needs to prove `hanoi(3)`, it will now try to prove:

```
move(3, left, right, center).
```

which reads something like:

"move 3 discs from the `left` pole to the `right` pole, using the `center` pole to hold things temporarily.

Rule Inform (X, Y)

- This is actually used for side effects (it can always be satisfied).
 - the side effects are generating output.

```
inform(X, Y) :-
```

```
    write('move a disc from the '),
```

```
    write(X),
```

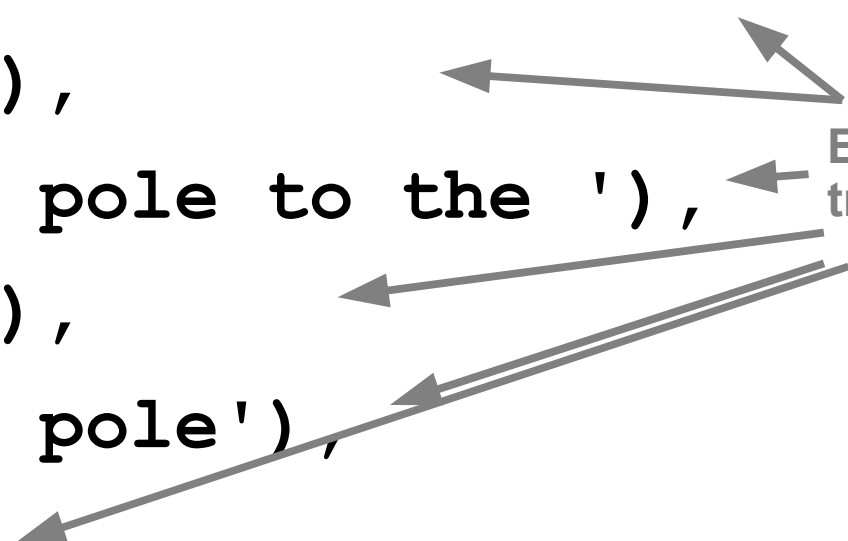
```
    write(' pole to the '),
```

```
    write(Y),
```

```
    write(' pole'),
```

```
    nl.
```

Each of these will be true (for any X, Y)



Inform(X,Y)

?- inform(left,center) .

move a disc from the left pole to the center pole

Yes

?- inform(right,left) .

move a disc from the right pole to the left pole

Yes

?- inform(north,south) .

move a disc from the north pole to the south pole

Yes

?-

Rule: `move (0, _, _, _) :- !.`

- Move 0 discs from any pole to any pole using any pole is true and should stop any backtracking.
 - whatever the poles are, this rule is true (will match any poles). The only restriction is the number of discs (0).
 - This is effectively the base case of a recursive definition.

NOTE: The special variable `'_'` is a *don't care* variable, it can match anything, but no assignment results from this matching.

The rest of move

```
move(N, A, B, C) :-  
    M is N-1,  
    move(M, A, C, B),  
    inform(A, B),  
    move(M, C, B, A).
```

To move N discs from A to B ,
first move the top $N-1$ discs from A to C ,
then move the bottom disc from A to B ,
then move the $N-1$ discs from C to B .

Simplest Trace

`move (1 , left , right , center)`

`move (0 , left , center , right)`

`inform (left , right)`

`move (0 , center , right , south)`

Prolog If-Then-Else

(A -> B ; C)

If A can be proved, then try to prove B.

If A can't be proved, then try to prove C.

```
max(X, Y, Z) :-  
  ( X =< Y  
  -> Z = Y  
  ; Z = X  
  ).
```

Lists

- Prolog includes facilities for processing lists

- A list looks like this:

```
[ 1, joe, X, father(joe), [3, sam], []]
```

- list elements can be many things:
 - constants, atoms, complex terms, lists
- The empty list is []

More Lists

- A non-empty lists have a *head* and *tail*.
- | is used to extract the head and/or tail:

[X | Y] = [0, a, 1, b, 2, c].

X = 0

Y = [a, 1, b, 2, c]

Fun with lists

```
?- [Hard,Impossible|Easy] =  
    [ ai, opsys, cs1, cs2, modcomp].
```

```
Hard = ai
```

```
Impossible = opsys
```

```
Easy = [cs1, cs2, modcomp]
```

List-o-mania

?- [_,'_','_',Fourth|_] = [a,b,c,d,e,f,g].

Fourth = d

?- [_|[_|[_|[Fourth|_]]]]=[a,b,c,d,e,f,g].

Fourth = d

List Membership

```
member (X, [X|T]) .
```

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

```
?- member (joe, [sally, joe, sam]) .
```

```
yes
```

```
?- member (ai, [opsys, cs1, cs2]) .
```

```
no
```

Another List Membership

```
member (X, [X|_]) .
```

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

A general rule of thumb: don't name anything you don't need to.

Recursive List Comparison

```
listequal([], []).
```

```
listequal([Ha|Ta], [Hb|Tb]) :-  
    Ha = Hb, listequal(Ta, Tb).
```

```
?- listequal([a,b], [a,b]).
```

yes

```
?- [a,b] = [a,b].
```

yes

There is really no need for `listequal`, since this will work as well...

Recursive List Generation

```
makelist(0,_,[]) :- !.  
makelist(Len,Start,[Start|Tail]) :-  
    M is Len-1,  
    K is Start+1,  
    makelist(M,K,Tail).
```

```
?- makelist(4,1,X).
```

```
X = [1, 2, 3, 4]
```

```
?- makelist(5,22,Z).
```

```
Z = [22, 23, 24, 25, 26]
```