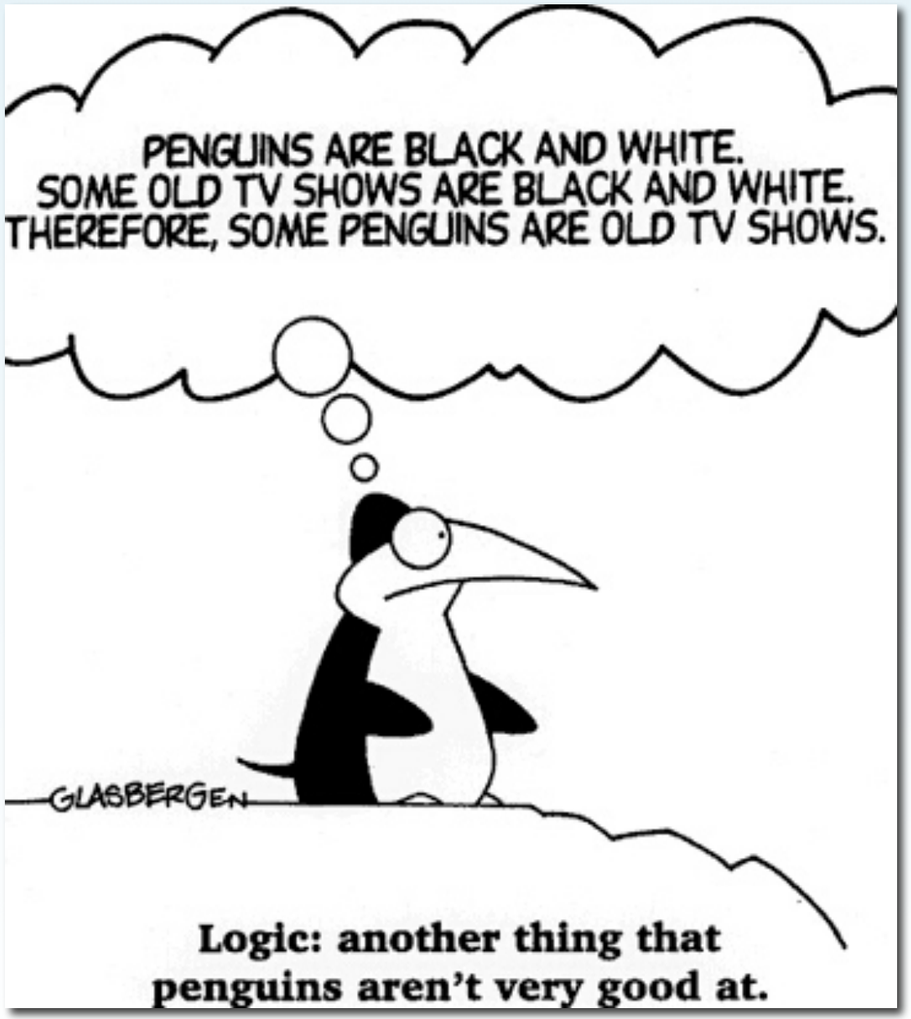
9 Prolog


PENGUINS ARE BLACK AND WHITE.
SOME OLD TV SHOWS ARE BLACK AND WHITE.
THEREFORE, SOME PENGUINS ARE OLD TV SHOWS.
GLASBERGEN
Logic: another thing that penguins aren't very good at.

9 Prolog

Introduction

Predicates & Goals

Rules

Satisfying Goals, Recursion & Backtracking

Equality

Terms & Lists

Arithmetic

The Cut & Negation

Prolog Pitfalls

9.1 Introduction

What is Prolog?

- *Untyped logic* programming language
- Computations are expressed by *rules* that define *relations* on objects
- Running a program/computing a value: Formulating a *goal* or *query*
- Result of program execution: Yes/No answer and a *binding of free variables*
- No higher-order predicates

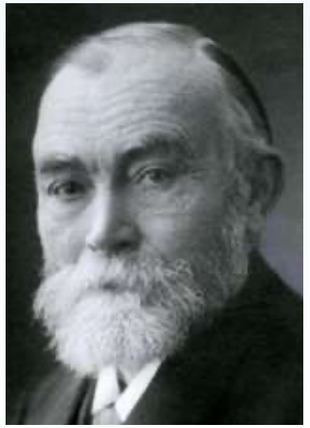
Logic: A Tool for Reasoning

Syllogism (logical argument) (*Aristotle, 350 BCE*)

E.g.: *All humans are mortal.*
Socrates is human.
Therefore: Socrates is mortal.

First-Order Logic (*Gottlob Frege, 1879 "Begriffsschrift"*)

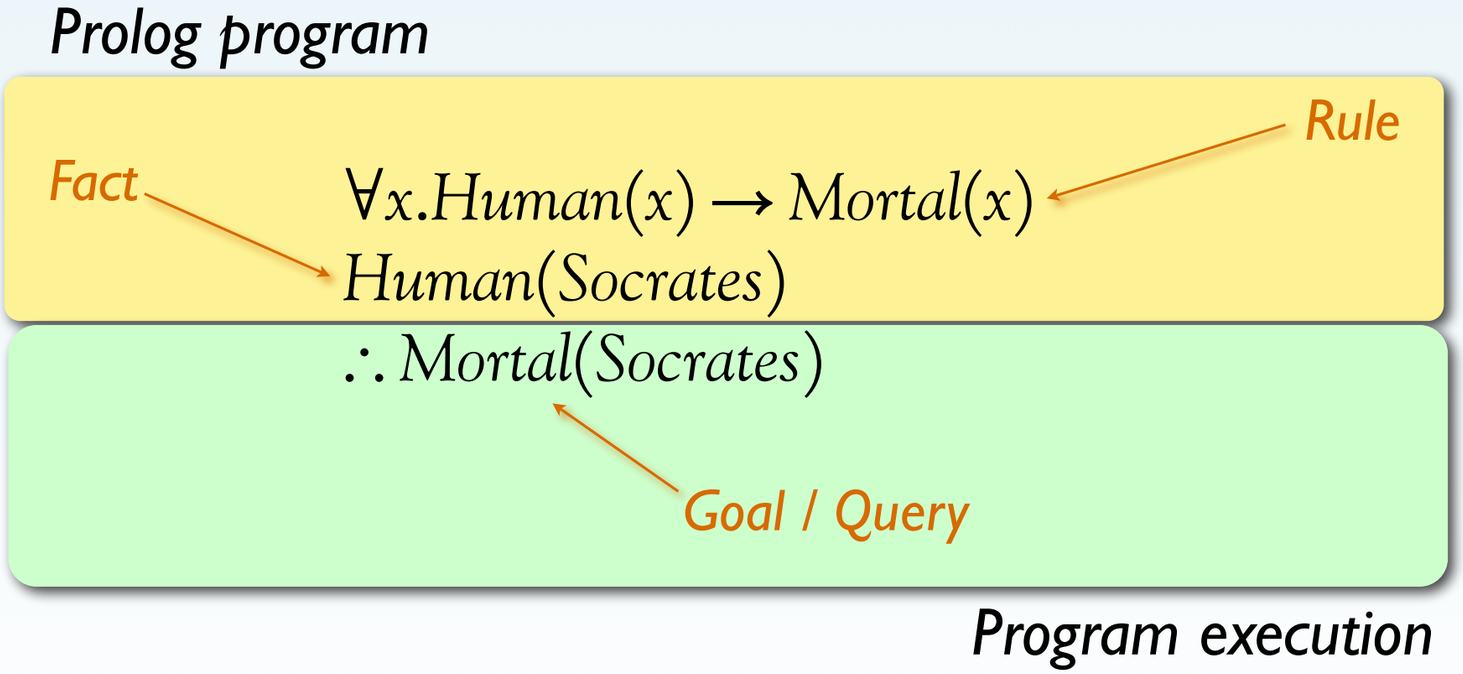
E.g.: $\forall x.Human(x) \rightarrow Mortal(x)$
$Human(Socrates)$
$\therefore Mortal(Socrates)$



Using Logic

Video Clip

Logic & Programming



Predicates, Functions & Sets

Nullary Predicate ≅ *Set A*

$$Term = \{\texttt{true}, \texttt{false}, \texttt{not true}, \texttt{not false}, \ldots\}$$

≈ *SQL query over table (with schema A, A×B, …)*

Unary Predicate (over A) ≅ *A* → *Bool* ≅ *Subset of A*

$$Even : \mathbb{N} \to \mathbb{B} = \{(\texttt{0}, \texttt{true}), (\texttt{1}, \texttt{false}), (\texttt{2}, \texttt{true}), \ldots\} \cong \{\texttt{0}, \texttt{2}, \texttt{4}, \ldots\} \subseteq \mathbb{N}$$

Binary Predicate (over A and B) ≅ *A* × *B* → *Bool* ≅ *Subset of A* × *B*

$$<: \mathbb{N} \times \mathbb{N} \to \mathbb{B} = \{((\texttt{0}, \texttt{0}), \texttt{false}), ((\texttt{0}, \texttt{1}), \texttt{true}), \ldots, ((\texttt{5}, \texttt{3}), \texttt{false}), \ldots, ((\texttt{5}, \texttt{7}), \texttt{true}), \ldots\}$$
$$\cong \{(\texttt{0}, \texttt{1}), \ldots, (\texttt{5}, \texttt{7}), \ldots\} \subseteq \mathbb{N} \times \mathbb{N}$$

SWI-Prolog

swipl [<option> ...] [-f <file> ...]

?- <goal>.	solve goal
?- [<file>].	load definitions from a file (file name w/o ".pl" extension)
?- help.	help on predicates
?- trace.	Turn on tracing of next goal
?- halt.	quit

On flip:
/usr/local/apps/swipl/current/bin/swipl

9.2 Predicates & Goals

The basic entities of Prolog are *predicates*
(Note: *predicate* ≅ *relation* ≅ *set*)

color = { red, blue }

```
color(red).
color(blue).
```

likes = { (john,red), (mary,red), (john,mary) }

fact

```
likes(john,red).
likes(mary,red).
likes(john,mary).
```

rules with empty bodies define facts

For each set S there is a corresponding predicate
$P_S(x) :\Leftrightarrow x \in S$
(*characteristic function*)

Exercises

```
likes(john,red).
likes(mary,red).
likes(john,mary).
```

Define the successor predicate on the first 5 natural numbers

Define the less than predicate on the first 3 natural numbers

Goals & Queries

A *goal* (or *query*) looks like a fact, but is a question:

Prolog prompt

Goal

```
?- color(red).
true.
```

Answer

```
?- likes(mary,red).
true.
```

```
?- color(green).
false.
```

closed world assumption

"No" means "cannot be derived" from the currently known facts

Answers are sought in a "*database*" of given facts (and rules):

Prolog program

```
color(red).
color(blue).
likes(john,red).
likes(mary,red).
likes(john,mary).
```

likes.pl

Variables

Goals may contain variables

```
?- color(X).
X = red ↵
```

satisfied with answer

```
?- color(X).
X = red ;
X = blue.
```

want more evidence

```
?- likes(X,blue).
false.
```

```
?- likes(john,X).
X = red ;
X = mary.
```

```
?- likes(X,Y).
X = john
Y = red ;
X = mary
Y = red ;
X = john
Y = mary.
```

enumerating a predicate

```
color(red).
color(blue).
likes(john,red).
likes(mary,red).
likes(john,mary).
```

Exercises

Which goal finds the successor of 3?

Which goal finds the predecessor of 2?

Write the goal to find all numbers that are greater than 1

```
succ(1,2).
succ(2,3).
succ(3,4).
succ(4,5).
succ(5,6).
```

```
lt(1,2).
lt(2,3).
lt(1,3).
```

Variables are "1st order"

Variables cannot be used for predicates:

```
?- X(red).
?- X(john,mary).
```

⇒ *We can only ask whether a particular predicate holds.*

⇒ *We cannot ask which predicate(s) hold for objects.*

Conjunctions

Do John and Mary like each other?

```
?- likes(john,mary), likes(mary,john).
false.
```

Comma denotes logical and

```
color(red).
color(blue).
likes(john,red).
likes(mary,red).
likes(john,mary).
```

Is there anything that John and Mary both like?

```
?- likes(john,X), likes(mary,X).
X = red ;
false.
```

X must be bound to the same value

No other solutions found

Which colors does John like?

```
?- likes(john,X), color(X).
X = red ;
false.
```

9.3 Rules

A *rule* consists of a *head* and a *body*

```
marry(X,Y) :- likes(X,Y), likes(Y,X).
```

:- denotes ⇐ (read "if")

```
?- marry(john,Y).
false.
```

```
color(red).
color(blue).
likes(john,red).
likes(mary,red).
likes(john,mary).
```

Rules can contain *free variables*

```
friends(X,Y) :- likes(X,Z), likes(Y,Z).
```

```
?- friends(X,Y).
X = john
Y = mary
```

Are there any other solutions? Which?

More on Rules

Predicates can be defined through multiple rules:

```
marry(X,Y) :- likes(X,Y), likes(Y,X).
marry(X,Y) :- likes(X,Y), isRich(X).
marry(john,Y).
```

Facts are rules without a body

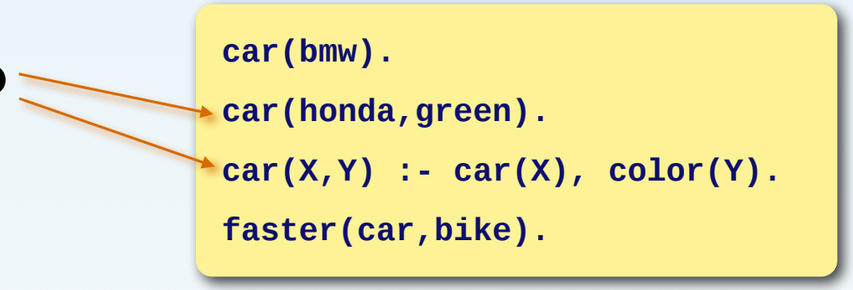
Facts are unconditional

```
color(red).
color(blue).
likes(john,red).
likes(mary,red).
likes(john,mary).
```

Overloading

Predicates are identified by name *and* arity.

car/2 refers to



```
car(bmw).
car(honda,green).
car(X,Y) :- car(X), color(Y).
faster(car,bike).
```

```
?- car(X).
X = bmw.
```

```
?- car(X,Y).
X = honda,
Y = green ;
X = bmw,
Y = red ;
X = bmw,
Y = blue.
```

References to predicates often include/require the predicate's arity

Recursive Rules

```
inside(X,Y) :- contains(Y,X).                  /* (1) */
inside(X,Y) :- contains(Y,Z), inside(X,Z).     /* (2) */
```

```
contains(house,bathroom).
contains(house,kitchen).
contains(kitchen,fridge).
contains(fridge,mouse).
```

```
?- inside(mouse,house).
true.
```

Why?

(2)

```
contains(house,kitchen), inside(mouse,kitchen).
```

(2)

```
contains(kitchen,fridge), inside(mouse,fridge).
```

(1)

```
contains(fridge,mouse).
```

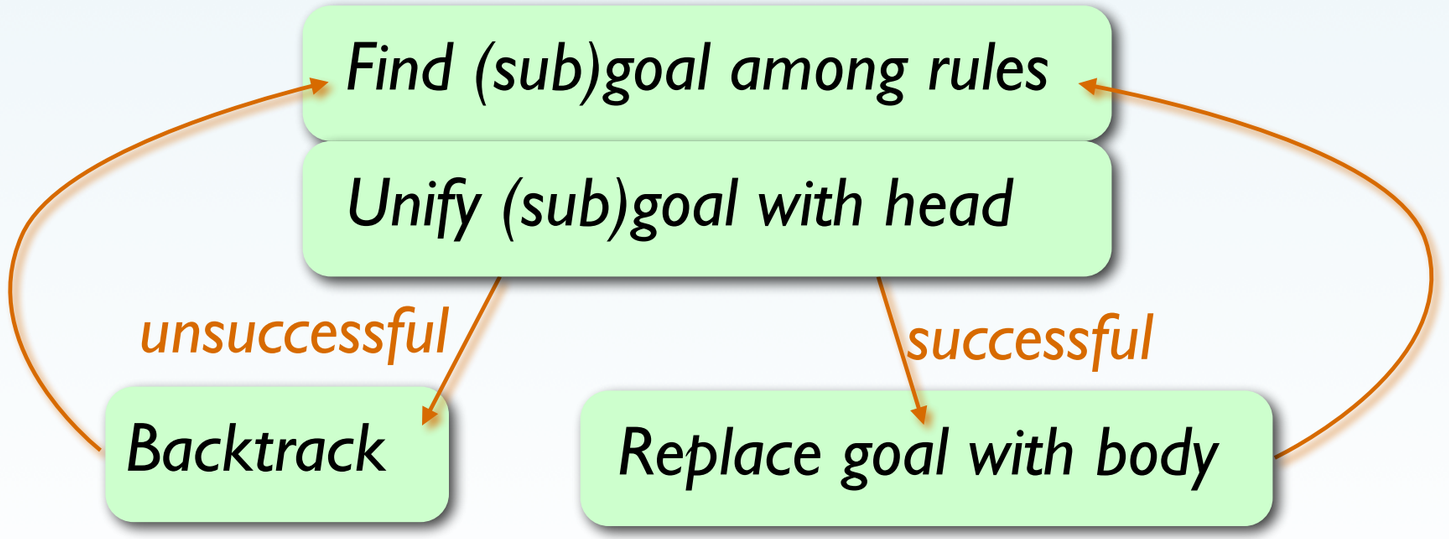
9.4 Satisfying Goals

Prolog's computational model

Logical view:

Try to refute negated goal (using SLD resolution)

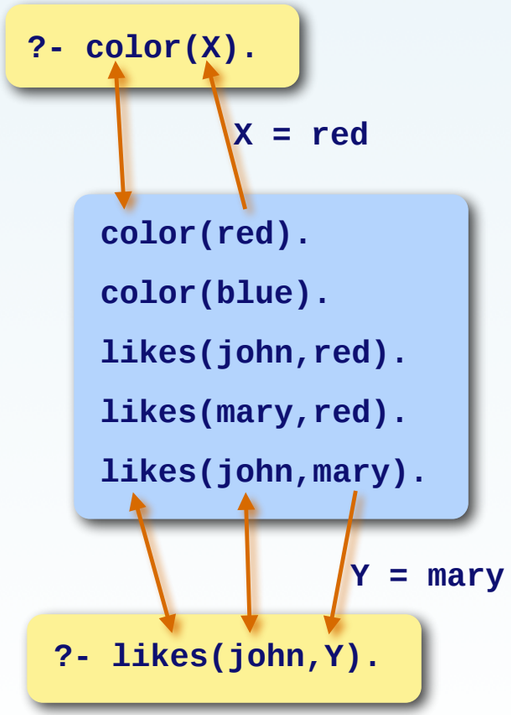
Operational view:



“Fact” Queries

Simplified view:

A goal/query is a pattern that is *matched* against *facts*.



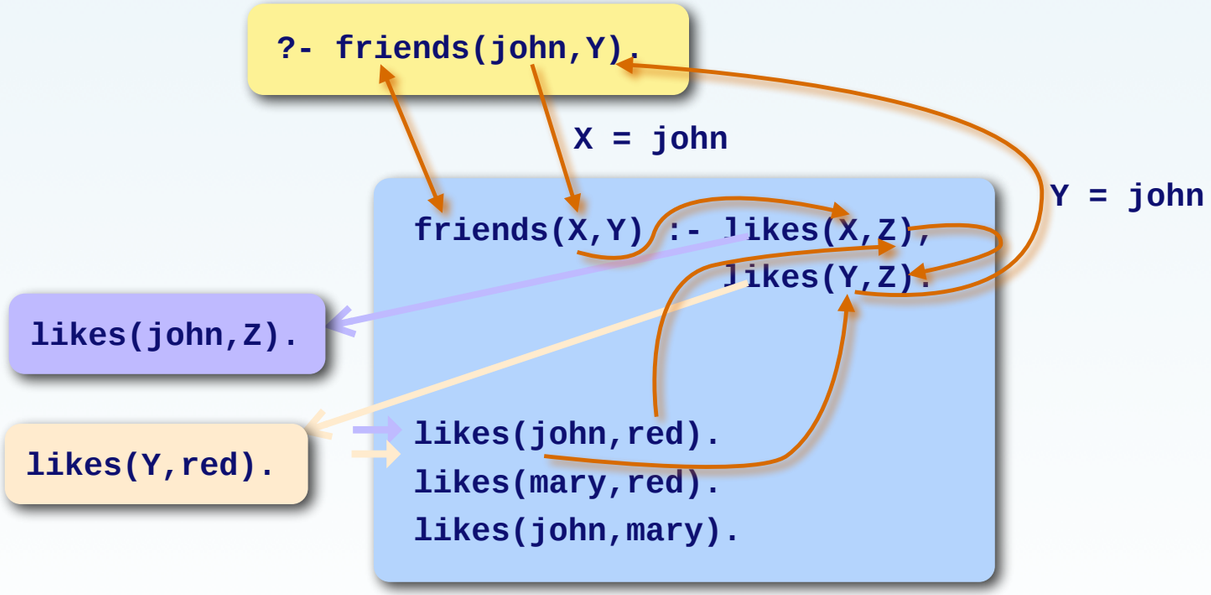
Variables in goals: Result parameters

Data flows from database to call

"View" Queries

General view:

A goal/query is a pattern that is *unified* with *rule heads*.

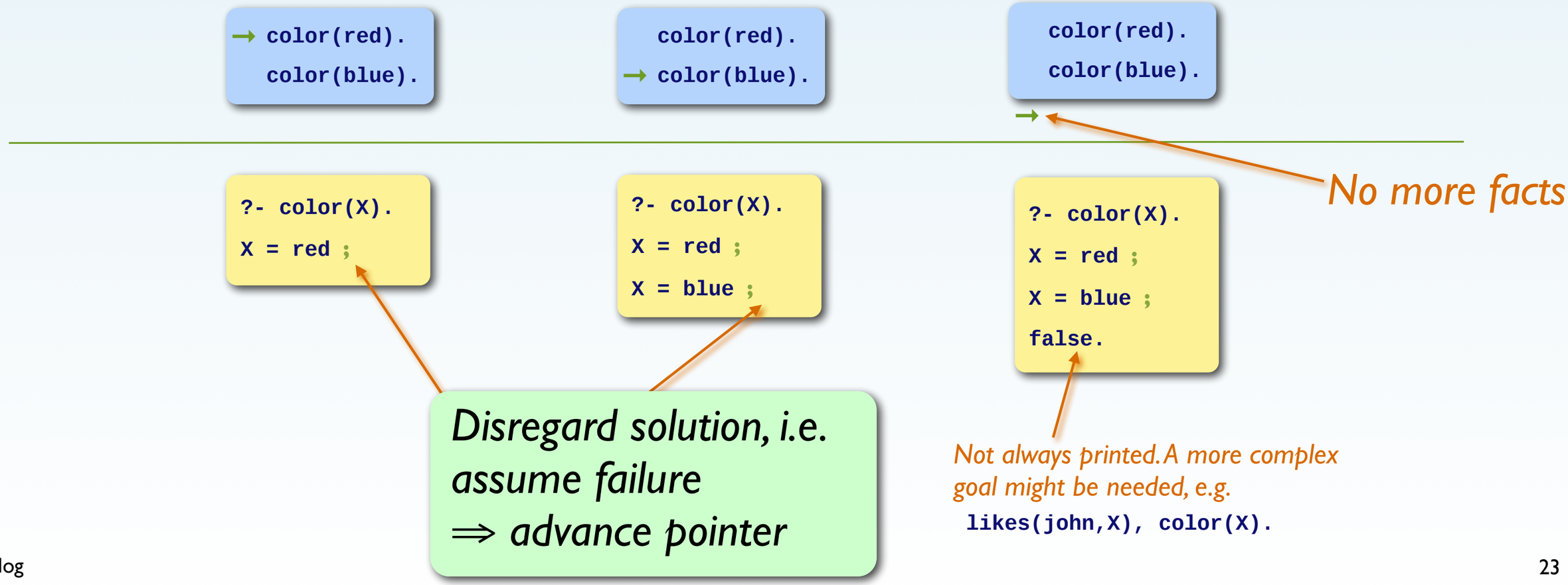


Variables in heads:
Value or result parameters

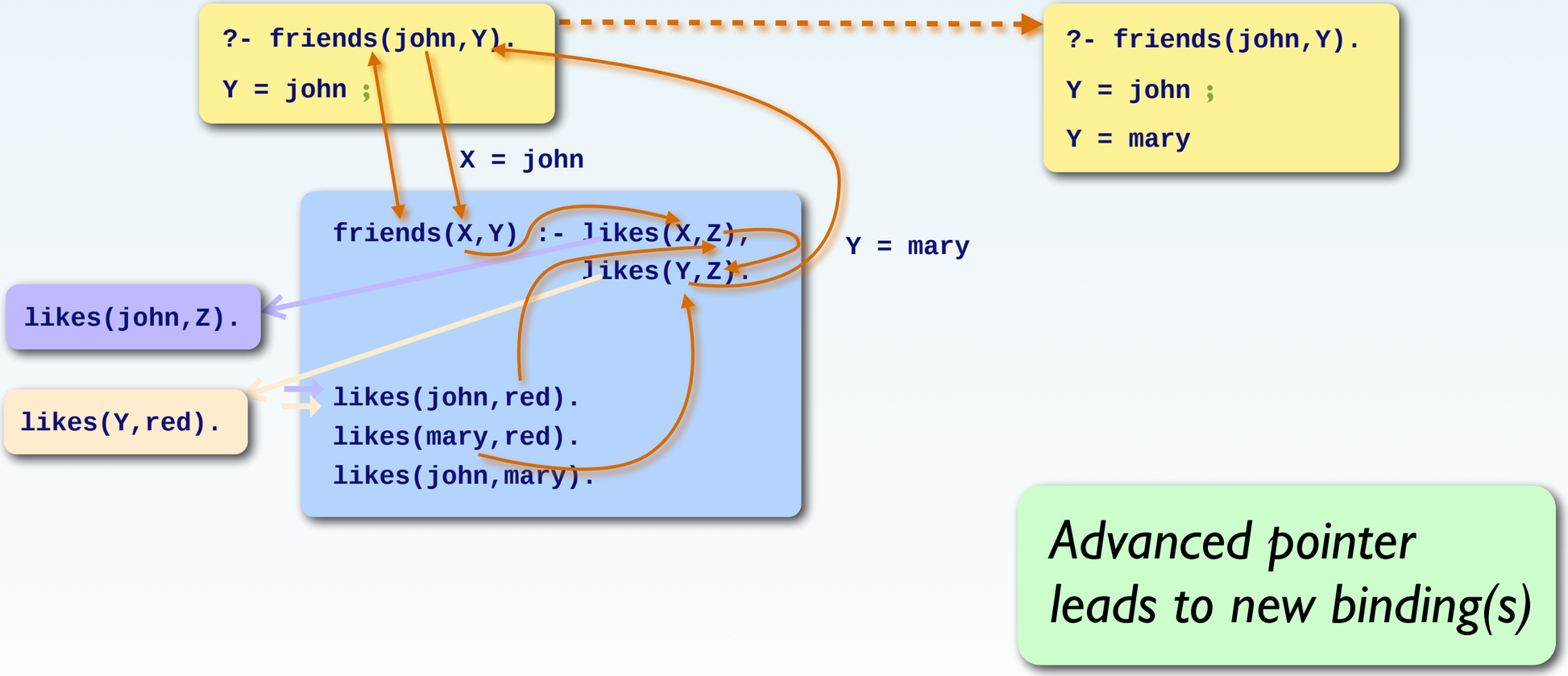
Data flows in both directions

Re-Satisfying Goals

For each (sub-)goal, imagine having a pointer "→" in the Prolog database.



Multiple Predicate Instances



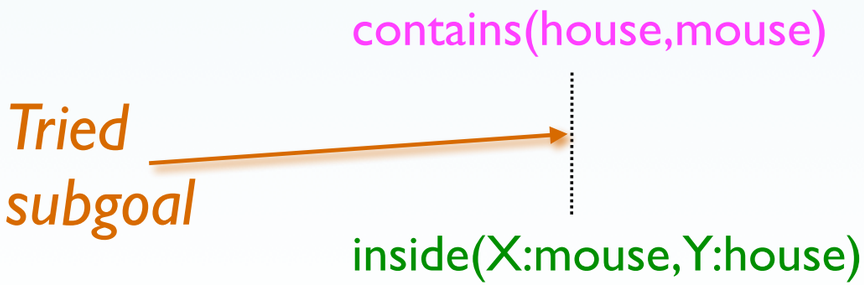
Advanced pointer leads to new binding(s)

Tree Model

A goal that is (tried to be) satisfied by a rule
H :- P_1, ..., P_n is extended by P_1, ..., P_n.
⇒ leads to a tree of goals.

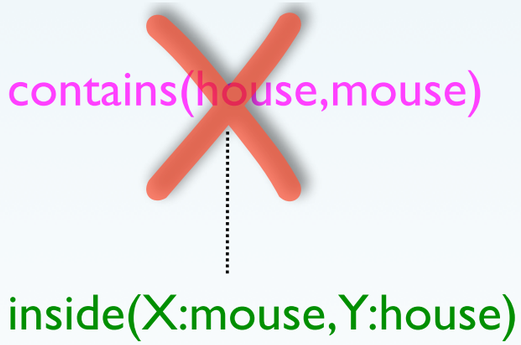
(1) To satisfy inside(mouse,house):
- select 1st rule for inside
- bind X and Y (to mouse & house)
- create subgoal contains(house,mouse)
  ⇒ create new marker "→"

```
→ contains(house,bathroom).
  contains(house,kitchen).
  contains(kitchen,fridge).
  contains(fridge,mouse).
→ inside(X,Y) :- contains(Y,X).
  inside(X,Y) :- contains(Y,Z),
                 inside(X,Z).
```



Revoking Subgoals

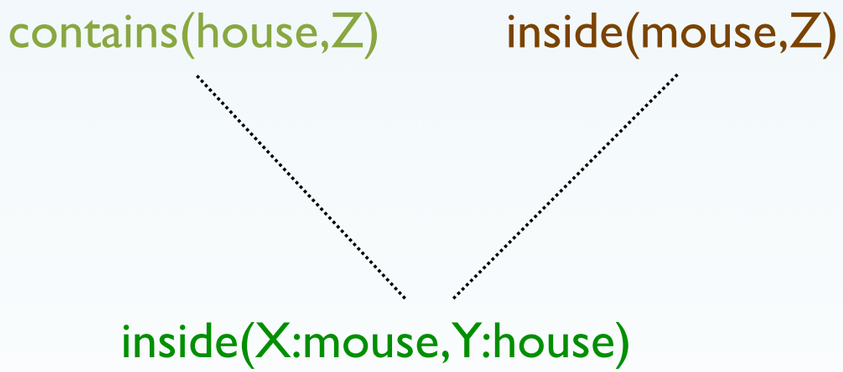
(2) Marker "→" for goal contains(house,mouse) scans the whole database, but isn't able to find such a fact (or a matching head of a rule)
⇒ goal fails and is revoked.




```
→ contains(house,bathroom).
  contains(house,kitchen).
  contains(kitchen,fridge).
  contains(fridge,mouse).
→ inside(X,Y) :- contains(Y,X).
  inside(X,Y) :- contains(Y,Z),
                 inside(X,Z).
```

Expanding Subgoals

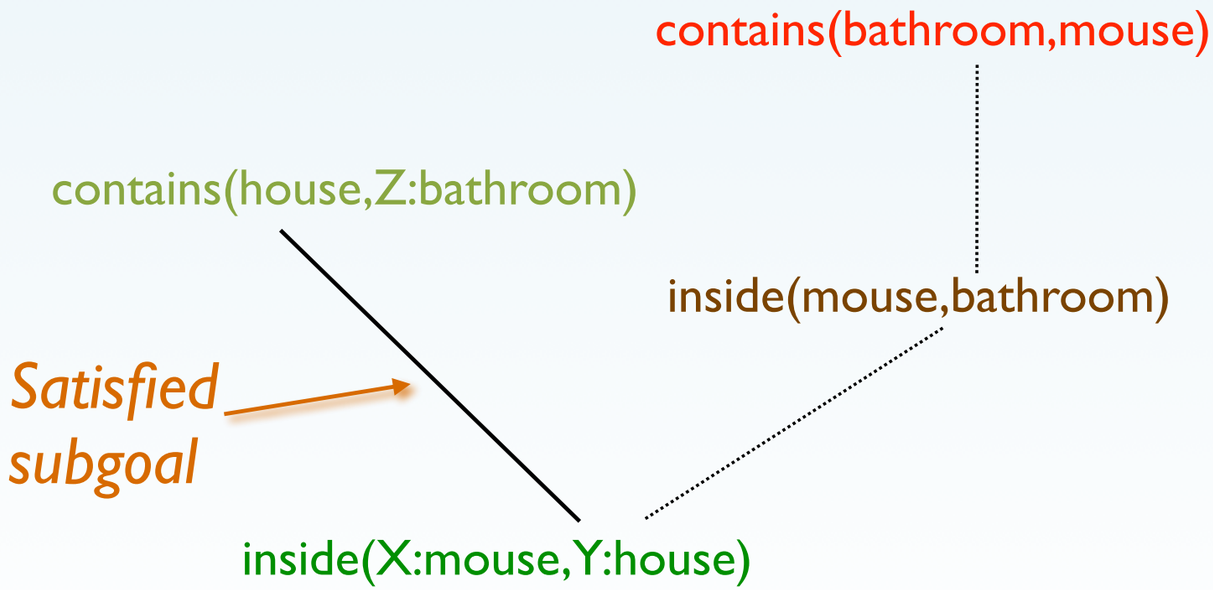
(3) Marker "→" for goal inside(mouse,house) is advanced
⇒ create new subgoals contains(house,Z) and inside(mouse,Z)
together with two new markers.




```
→ contains(house,bathroom).
  contains(house,kitchen).
  contains(kitchen,fridge).
  contains(fridge,mouse).
→ inside(X,Y) :- contains(Y,X).
  inside(X,Y) :- contains(Y,Z),
                 inside(X,Z).
```

Lots of Markers ...

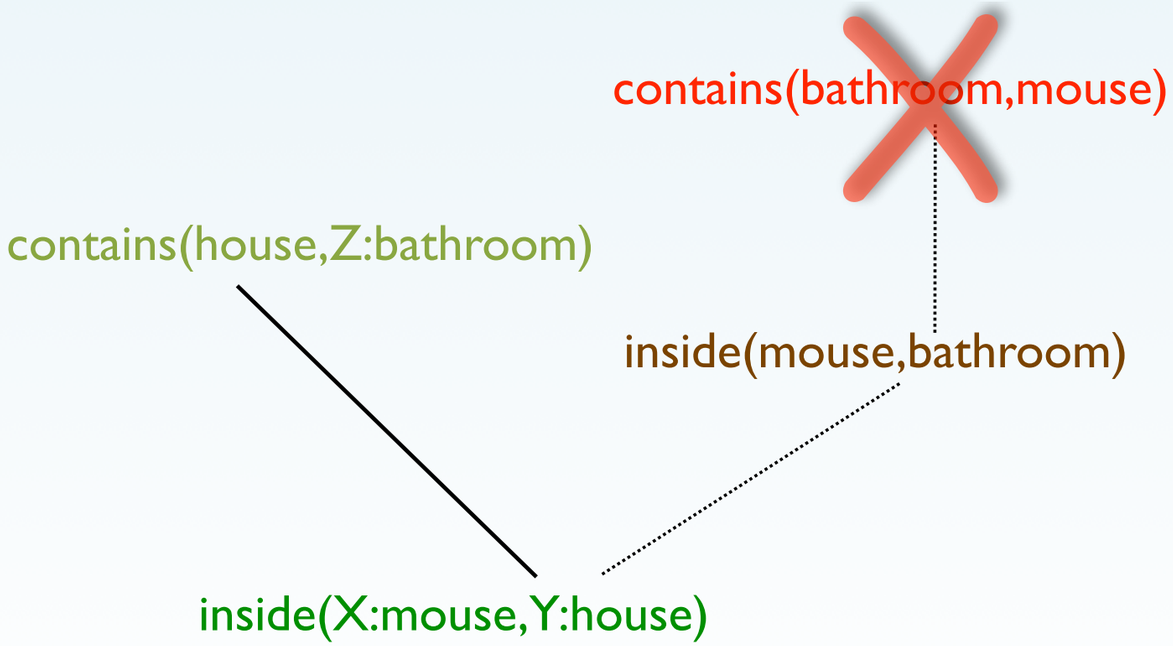
(4) Z is bound to bathroom, which causes the "contains" goal to succeed.
The goal inside(mouse,bathroom) is expanded to
contains(bathroom,mouse) (plus new marker "→").




```
contains(house,bathroom).
contains(house,kitchen).
contains(kitchen,fridge).
contains(fridge,mouse).
inside(X,Y) :- contains(Y,X).
inside(X,Y) :- contains(Y,Z),
               inside(X,Z).
```

Lots of Markers ...

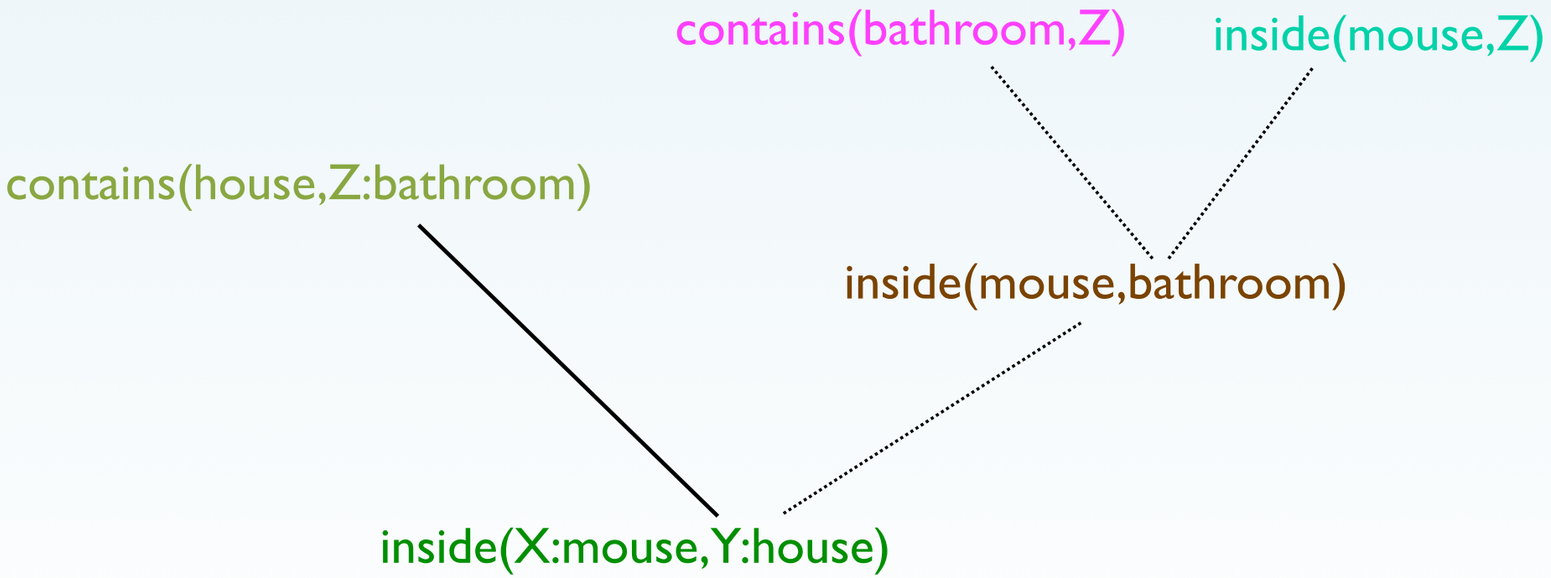
(5) contains(bathroom,mouse) cannot be satisfied and must be revoked; marker "→" for goal inside(mouse,bathroom) is advanced.




```
contains(house,bathroom).
contains(house,kitchen).
contains(kitchen,fridge).
contains(fridge,mouse).
inside(X,Y) :- contains(Y,X).
inside(X,Y) :- contains(Y,Z),
               inside(X,Z).
```

Even more markers ...

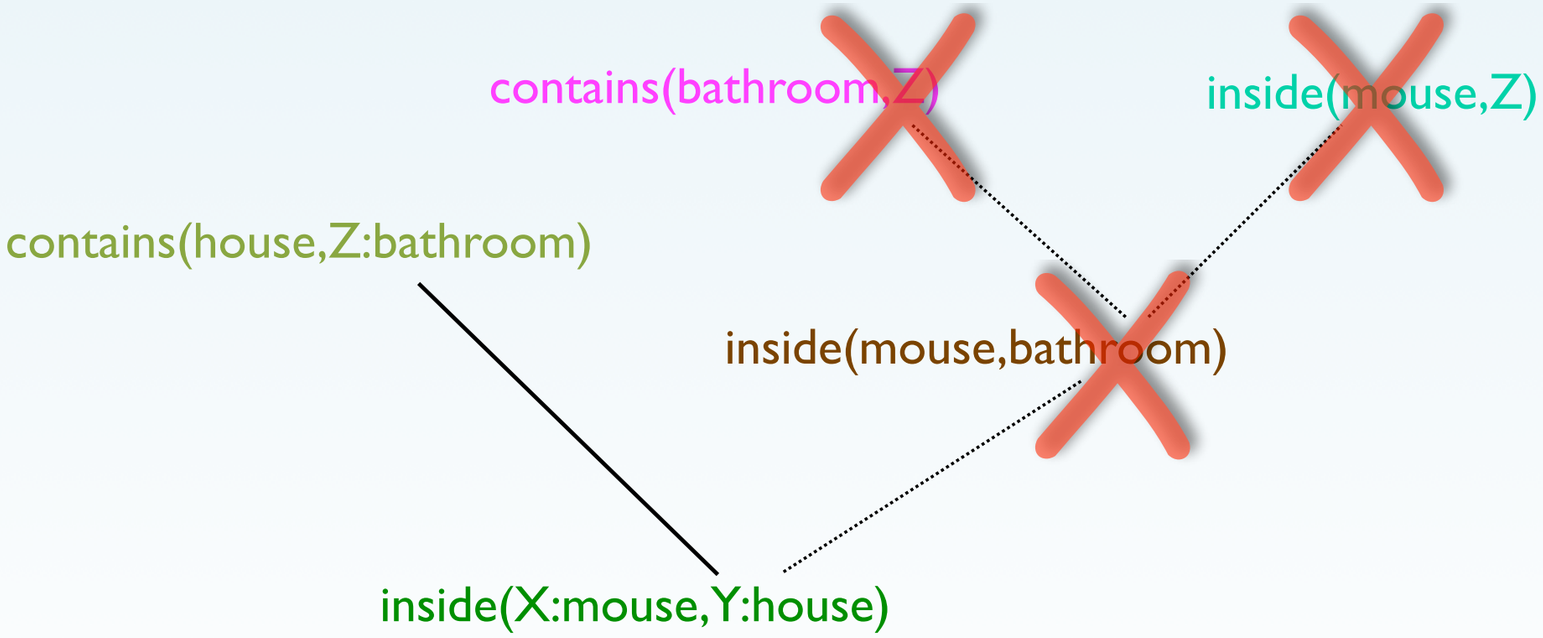
(6) 2nd rule for inside(mouse,bathroom) is expanded (and two more markers are created), but cannot succeed because no fact matches contains(bathroom,...).




```
contains(house,bathroom).
contains(house,kitchen).
contains(kitchen,fridge).
contains(fridge,mouse).
inside(X,Y) :- contains(Y,X).
inside(X,Y) :- contains(Y,Z),
               inside(X,Z).
```

Backtracking

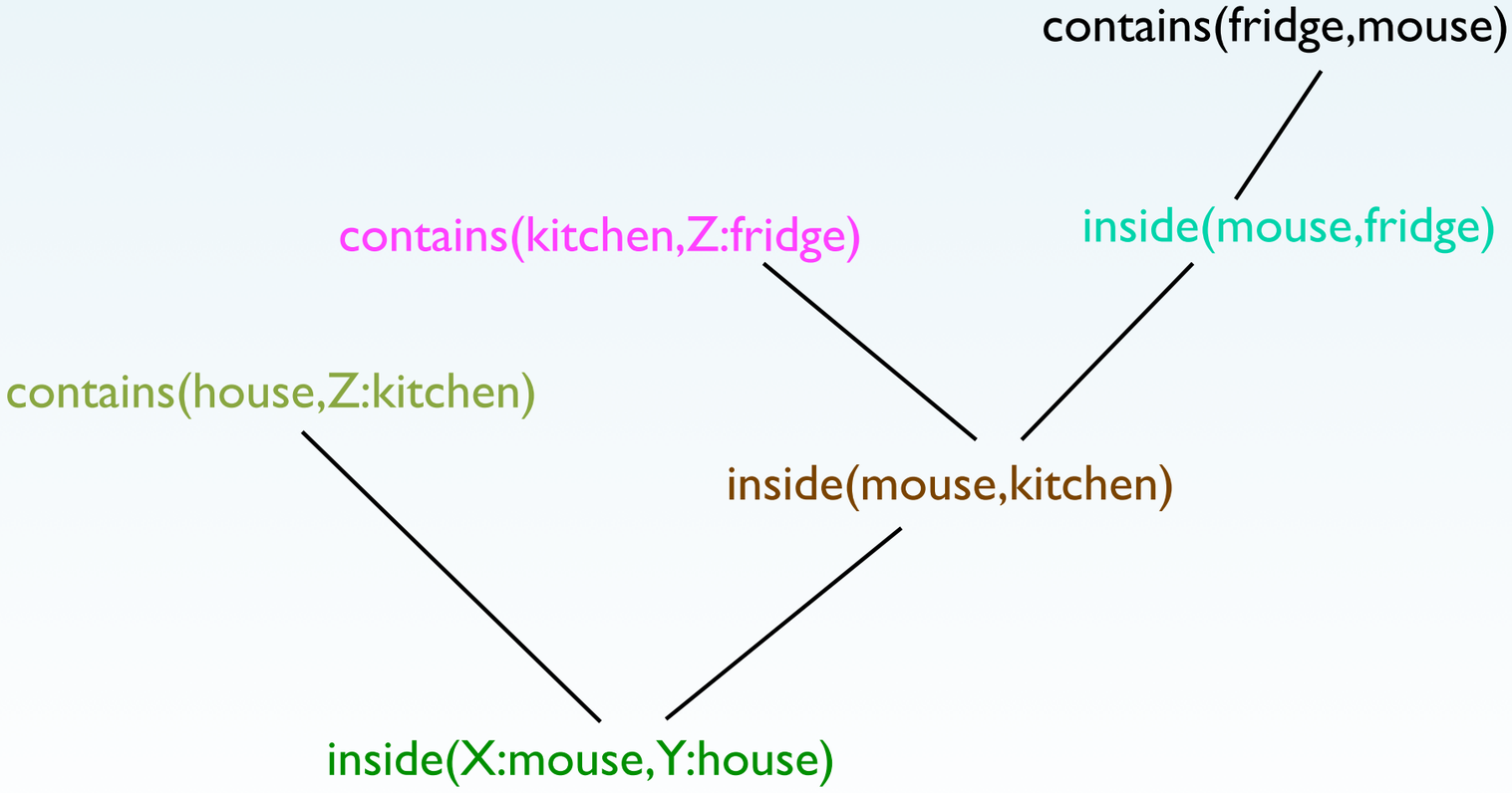
(7) Both subgoals are revoked. Since there is no other way of satisfying inside(mouse,bathroom), this goal is revoked, too. This fact forces the advance of the "→" pointer.




```
contains(house,bathroom).
contains(house,kitchen).
contains(kitchen,fridge).
contains(fridge,mouse).
inside(X,Y) :- contains(Y,X).
inside(X,Y) :- contains(Y,Z),
               inside(X,Z).
```

Resatisfying Goals

(8) The previous steps are repeated, now with "kitchen" bound to Z. In the recursive call to "contains", Z will be bound to fridge; the first rule for inside(mouse,Z) succeeds.




```
contains(house,bathroom).
contains(house,kitchen).
contains(kitchen,fridge).
contains(fridge,mouse).
inside(X,Y) :- contains(Y,X).
inside(X,Y) :- contains(Y,Z),
               inside(X,Z).
```

Exercises

Given the predicates female/1, male/1, and parent/2, define the following predicates.

(1) father/2 and mother/2

(2) grandfather/2, and grandmother/2

(3) child/2, son/2, and daughter/2

```
male(tywin).
male(jaime).
male(tyrion).
male(joffrey).
male(tommen).

female(joanna).
female(cersei).
female(myrcella).
```

```
parent(tywin,jaime).
parent(tywin,cersei).
parent(tywin,tyrion).
parent(joanna,jaime).
parent(joanna,cersei).
parent(joanna,tyrion).
parent(jaime,joffrey).
parent(jaime,tommen).
parent(jaime,myrcella).
parent(cersei,joffrey).
parent(cersei,tommen).
parent(cersei,myrcella).
```

Exercises

Given the predicates female/1, male/1, and parent/2, define the following predicates.

(4) grandchild/2, grandson/2, and granddaughter/2

(5) ancestor/2

(6) sibling/2, brother/2, and sister/2

```
male(tywin).
male(jaime).
male(tyrion).
male(joffrey).
male(tommen).

female(joanna).
female(cersei).
female(myrcella).
```

```
parent(tywin,jaime).
parent(tywin,cersei).
parent(tywin,tyrion).
parent(joanna,jaime).
parent(joanna,cersei).
parent(joanna,tyrion).
parent(jaime,joffrey).
parent(jaime,tommen).
parent(jaime,myrcella).
parent(cersei,joffrey).
parent(cersei,tommen).
parent(cersei,myrcella).
```

Exercises

Given the predicates female/1, male/1, and parent/2, define the following predicates.

(7) aunt/2 and uncle/2

(8) mate/2

(9) incest/2

```
male(tywin).
male(jaime).
male(tyrion).
male(joffrey).
male(tommen).

female(joanna).
female(cersei).
female(myrcella).
```

```
parent(tywin,jaime).
parent(tywin,cersei).
parent(tywin,tyrion).
parent(joanna,jaime).
parent(joanna,cersei).
parent(joanna,tyrion).
parent(jaime,joffrey).
parent(jaime,tommen).
parent(jaime,myrcella).
parent(cersei,joffrey).
parent(cersei,tommen).
parent(cersei,myrcella).
```

9.5 Equality

Terms can be "made" equal through substitution

Terms are identical

Terms can be evaluated to the same number

Different forms of equality between terms:

(1) *Unification* = and \=

(2) *Equivalence* == and \==

(3) *Evaluation* =:= and =\=

Unification Equality

Unification equality

Two terms are equal when they can be instantiated so that they become identical.

```
?- 3=3.
true.
```

```
?- X=3.
X = 3.
```

```
?- X=Y.
X = Y.
```

```
?- likes(X,red)=likes(john,Y).
X = john,
Y = red.
```

Term

```
?- car(red)=car(X).
X = red.
```

Compare with evaluation equality

```
?- 3=4.
false.
```

```
?- 3+1=4.
false.
```

Unification

Mapping from variables to terms

A *unifier* for two terms T and T' is a *substitution* for variables, σ, such that σ(T) = σ(T').

```
?- 3=3.
true.
```

σ={}
σ(3) = 3 = σ(3)

```
?- X=3.
X = 3.
```

σ={X→3}
σ(X) = 3 = σ(3)

```
?- X=Y.
X = Y.
```

σ={X→Y}
σ(X) = Y = σ(Y)

```
?- likes(X,red)=likes(john,Y).
X = john,
Y = red.
```

σ={X→john,Y→red}
σ(likes(X,red)) = likes(john,red) = σ(likes(john,Y))

```
?- likes(X,red)=Y.
Y = likes(X,red).
```

σ={Y→likes(X,red)}
σ(likes(X,red)) = likes(X,red) = σ(Y)

Equivalence

Equivalence

Two terms are equivalent if they are identical.

```
?- 3==3.
true.
```

```
?- X==3.
false.
```

```
?- X==Y.
false.
```

```
?- X=3, X==3.
X = 3.
```

```
?- X=Y, X==Y.
X = Y.
```

```
?- X==Y, X=Y.
false.
```

Different from object/ reference equality

Evaluation Equality

Evaluation Equality

Two terms are evaluation equal if they evaluate to the same number.

```
?- 3+1=:=4.
true.
```

```
?- X=3, X*2=:=X+3.
X = 3.
```

```
?- 3+1==4.
false.
```

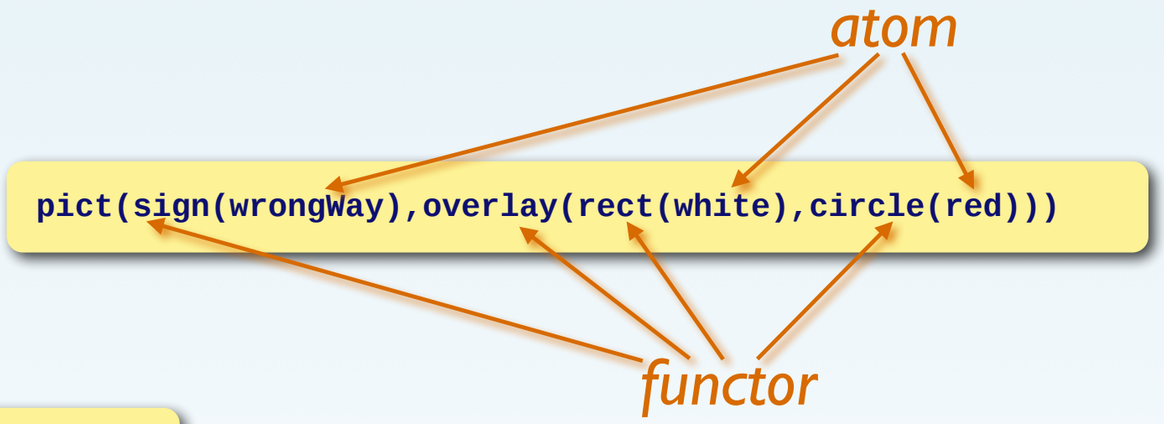
```
?- 3+1=4.
false.
```

```
?- 3=:=X.
ERROR: =:=/2: Arguments are not
sufficiently instantiated
```

```
?- X*2=:=X+3, X=3.
ERROR: =:=/2: Arguments are …
```

9.6 Terms & Lists

Terms denote structured objects




```
?- owns(john,X).
X = bike(blue).
```

```
?- owns(john,bike(X)).
X = blue.
```

```
?- owns(X,bike(Y)).
X = john,
Y = blue ;
X = mike,
Y = black.
```

```
owns(john,bike(blue)).
owns(mike,bike(black)).
owns(ann,caddilac(black)).
owns(ann,caddilac(silver)).
```

Variables cannot be used to match functors

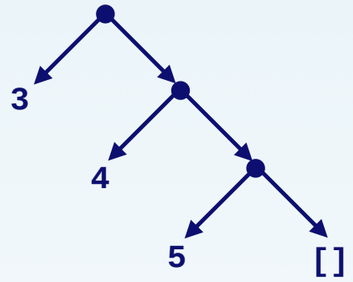
```
?- owns(john,X(blue)).
```

Lists

Lists are terms with special syntax

`[]`

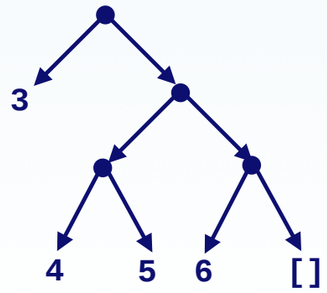
Empty list



`[3,4,5]` = `.(3,.(4,.(5,[])))`

Cons functor

`[3,[4,5],6]`



Lists can be heterogeneous

```
[3, little, pigs]
```

```
has(mary,[lamb(little),lamp(big),lame(pig)])
```

```
[walk(4),wait(2),bus(7),walk(2)]
```

List Patterns

Head *Tail*

```
?- story([X|Y]).
X = 3,
Y = [little, pigs].
```

```
story([3,little,pigs]).
```

```
?- story([X,Y|Z]).
X = 3,
Y = little,
Z = [pigs].
```

```
?- story([X,Y,Z]).
X = 3,
Y = little,
Z = pigs.
```

```
?- story([X,Y,Z|V]).
X = 3,
Y = little,
Z = pigs,
V = [].
```

```
?- story([X,Y,Z,V]).
false.
```

Pattern in Prolog		*Pattern in Haskell*
`[X,Y|Z]`	≈	`x:y:z`
`[X,Y,Z]`	≈	`[x,y,z]`

List Predicates

Wildcard, matches anything

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

Test

```
?- member(3,[2,3,4,3]).
true ;
true.
```

```
?- member(3,[2,3,4,3,1]).
true ;
true ;
false.
```

```
?- member(2,[2,3,4,3]).
true ;
false.
```

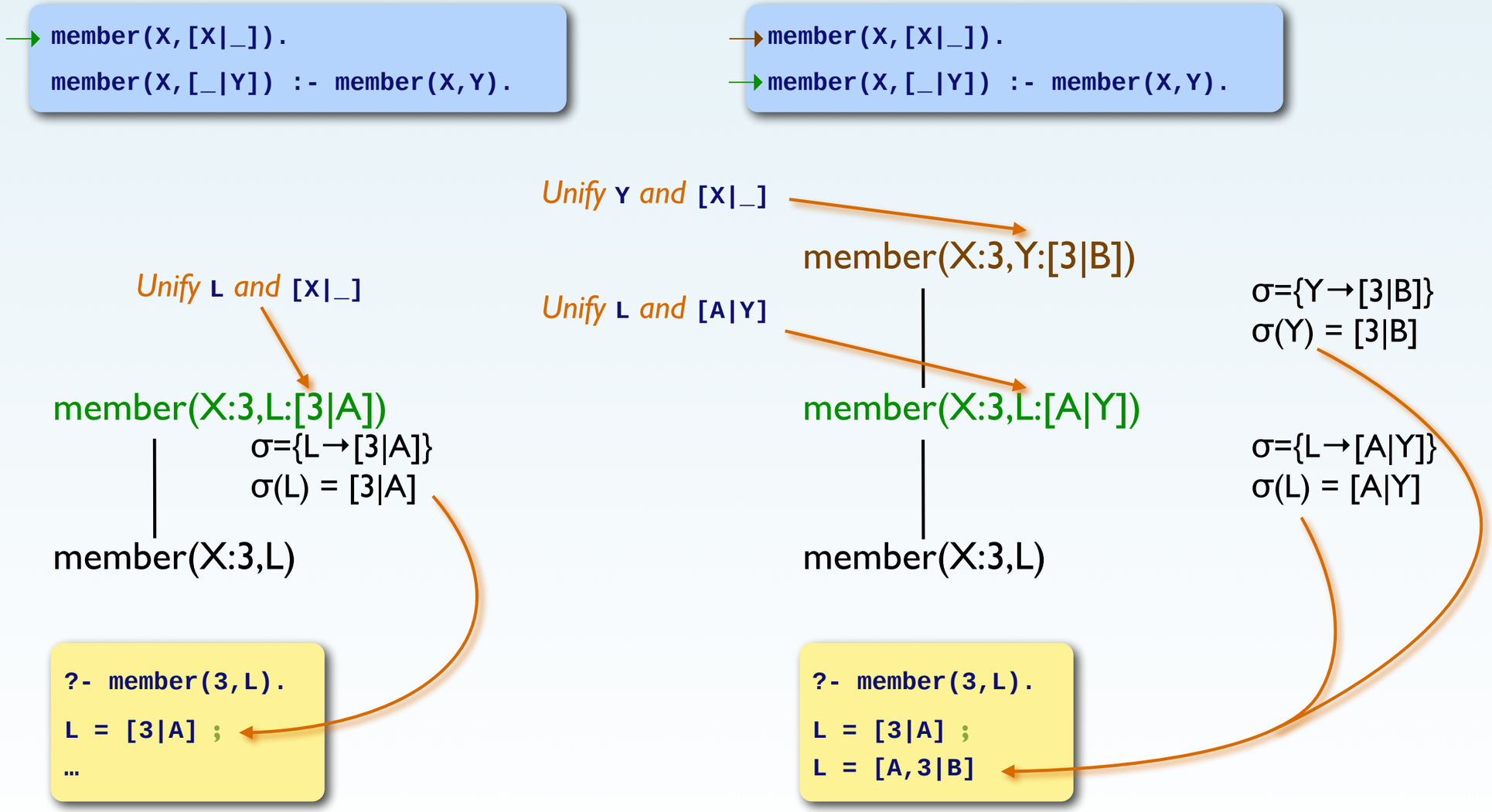
Traversal

```
?- member(X,[3,4]).
X = 3 ;
X = 4.
```

```
?- member(3,L).
L = [3|A] ;
L = [A,3|B] ;
L = [A,B,3|C]
…
```

Generation

Solving member(3,L).



The append/3 predicate

Constructing result in argument

```
append([], L, L).
append([X|L1], L2, [X|L3]) :- append(L1, L2, L3).
```

Only one way (of several) to read the definition!

```
?- append([2,3], [a,[c],d], L).
L = [2,3,a,[c],d].
```

```
?- append([2,3], Y, [2,3,a,[c],d]).
Y = [a,[c],d].
```

```
?- append(X, [a,[c],d], [2,3,a,[c],d]).
X = [2,3].
```

```
?- append(X, Y, [3,4,5]).
X = [],
Y = [3,4,5] ;
X = [3],
Y = [4,5] ;
X = [3,4],
Y = [5] ;
X = [3,4,5],
Y = [] ;
false.
```

Exercises

Define predicate del/3, such that in del(X, L, M) M is equal to the list L with the first occurrence of X removed.

What is the output of del(a, [a,b], L)?

Exercises

Avoid unwanted results in the definition of del(X, L, M).

```
del(X, [],    []).
del(X, [X|L], L).
del(X, [Y|L], [Y|M]) :- del(X, L, M).
```

What is the output of del(a, [a,b], L)?

Exercises

Define a predicate sublist/2, such that sublist(S, L) holds if the list S is a sublist of L.

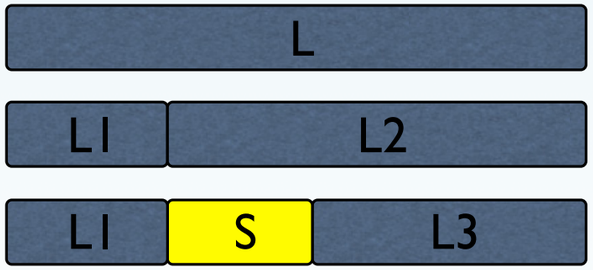
Examples:

sublist([c,d,e], [a,b,c,d,e,f]) yes

sublist([c,e], [a,b,c,d,e,f]) no

Hint: use append/3

Hint: S is a sublist of L if L can be decomposed into L1 and L2 and L2 can be decomposed into S and L3.



9.7 Arithmetic

Prolog Operators: =, \=, >, ... +, -, * , ...

X is <exp> binds result of <exp> to X

Unification vs. evaluation equality

```
?- X is 3*5.
X = 15.
```

```
?- X = 3*5.
X = 3*5.
```

`=:=` *evaluate and check*

`is` *evaluate and create binding*

```
?- 8 is X*2.
ERROR: is/2: Arguments are not
sufficiently instantiated
```

```
?- 8 = X*2.
false.
```

```
?- 4*2 = X*2.
X = 4.
```

```
?- fac(X,6).
ERROR: fac/2: Arguments are not
sufficiently instantiated
```

```
fac(1,1).
fac(N,M) :- K is N-1, fac(K,L), M is L*N.
```

Exercises

Define a predicate length/2, such that N in length(L, N) is equal to the length of the list L.

Can we swap the goals in the body of the second rule?

What does the goal length([a,b,c],N) compute if we use the following, alternative rule?

```
length([_|L], N) :- length(L, M), N = M+1.
```

Exercises

Simplify the following rule for length.

```
length([_|L], N) :- length(L, M), N = M+1.
```

Using the above version, which goal can compute the length of a list ?

9.8 The Cut & Negation

The cut "!" prevents backtracking:

- *do not resatisfy goals*
- *never advance any preceding markers*

The cut is a **non-logical**, **imperative** feature!

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

```
?- member(3,[3,4,3,3]).
true ;
true ;
true.
```

```
?- member(3,L).
L = [3|A] ;
L = [A,3|B] ;
L = [A,B,3|C]
…
```

`memberchk` *in SWIPL*

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

```
?- member(3,[3,4,3,3]).
true.
```

```
?- member(3,L).
L = [3|A].
```

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

```
?- member(3,[3,4,3,3]).
true ;
true.
```

```
?- member(3,L).
L = [3|A] ;
L = [A,3|B].
```

Negation

... can be tricky

Video Clip

Negation Predicate

```
likes(mary,X) :- animal(X), not(snake(X)).
```

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

If `P` *is true, then* `not(P)` *is false*

Otherwise, `not(P)` *is true*

```
animal(dog).
animal(python).
snake(python).
```

```
?- likes(mary,dog).
true.
```

```
?- likes(mary,cobra).
false.
```

```
?- likes(mary,python).
false.
```

```
?- likes(mary,cat).
false.
```

Negation as Failure

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

```
mother(mary).
mother(eve).
hadSex(eve).
```

```
virginBirth(X) :- mother(X), not(hadSex(X)).
```

```
?- virginBirth(eve).
false.
```

```
?- virginBirth(mary).
true.
```

```
?- virginBirth(X).
X = mary ;
false.
```

```
virginBirth(X) :- not(hadSex(X)), mother(X).
```

```
?- virginBirth(X).
false.
```

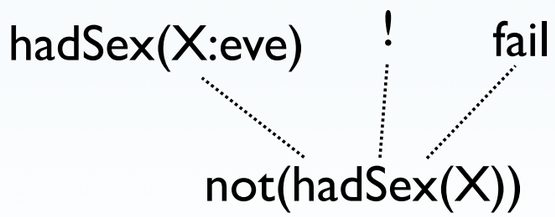
Negation as Failure

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

```
mother(mary).
mother(eve).
hadSex(eve).
```

```
?- virginBirth(X).
false.
```

```
virginBirth(X) :- not(hadSex(X)), mother(X).
```



not(hadSex(X))

=

not(∃X. hadSex(X))

=

∀X. not(hadSex(X))

≠

∃X. not(hadSex(X))

Barber Paradox

In a town, the barber shaves all males that do not shave themselves. Does the barber shave himself?

```
shaves(barber,X) :- male(X), not(shaves(X,X)).

male(barber).
```

```
?- shaves(barber,barber).
ERROR: Out of local stack
```

More on Negation

Video Clip

9.9 Prolog Pitfalls

- Misspelled predicates
- Wrong arity of predicates
- Missing rule cases
- Too general rules (e.g. 'catch-all' rules)
- Unintentional unification (use one name for variables that should be different)
- Left-recursion in rules:

```
path(A,C) :- path(A,B), edge(B,C).
path(A,A).
```

- Infinite unification (as in: X = [X]) (use unify_with_occurs_check)