Programming Paradigms

Summer Term 2018

Prof. Janis Voigtländer
**University of Duisburg-Essen**

Freeing the programmer from the necessity to explicitly plan and specify the computation process that leads to a problem solution: **"What instead of How"**
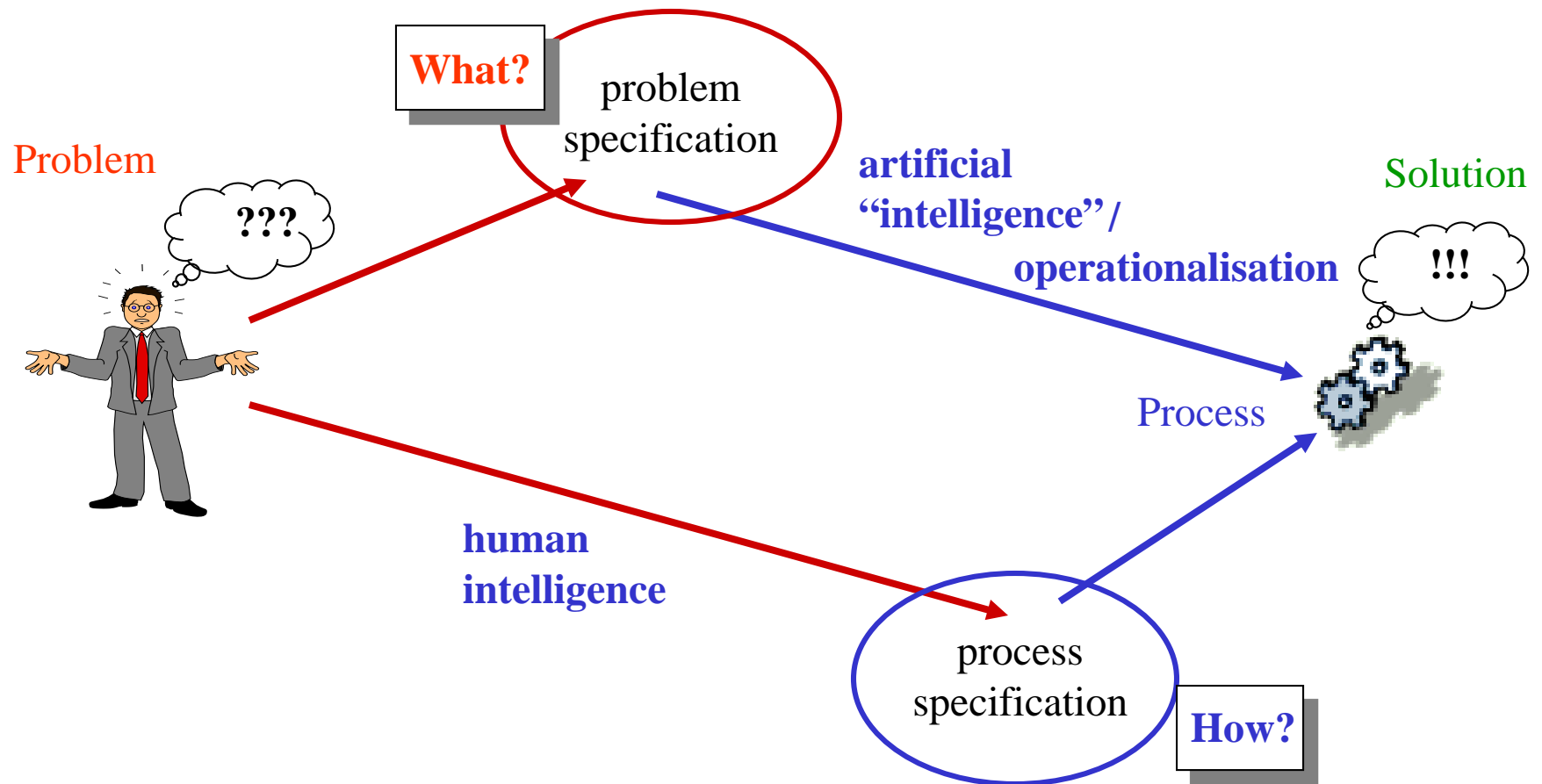
"There are five houses, each of a different color and inhabitated by a man of a different nationality with a different pet, drink and brand of smokes …"

("Einstein's Riddle", see http://en.wikipedia.org/wiki/Zebra_Puzzle)
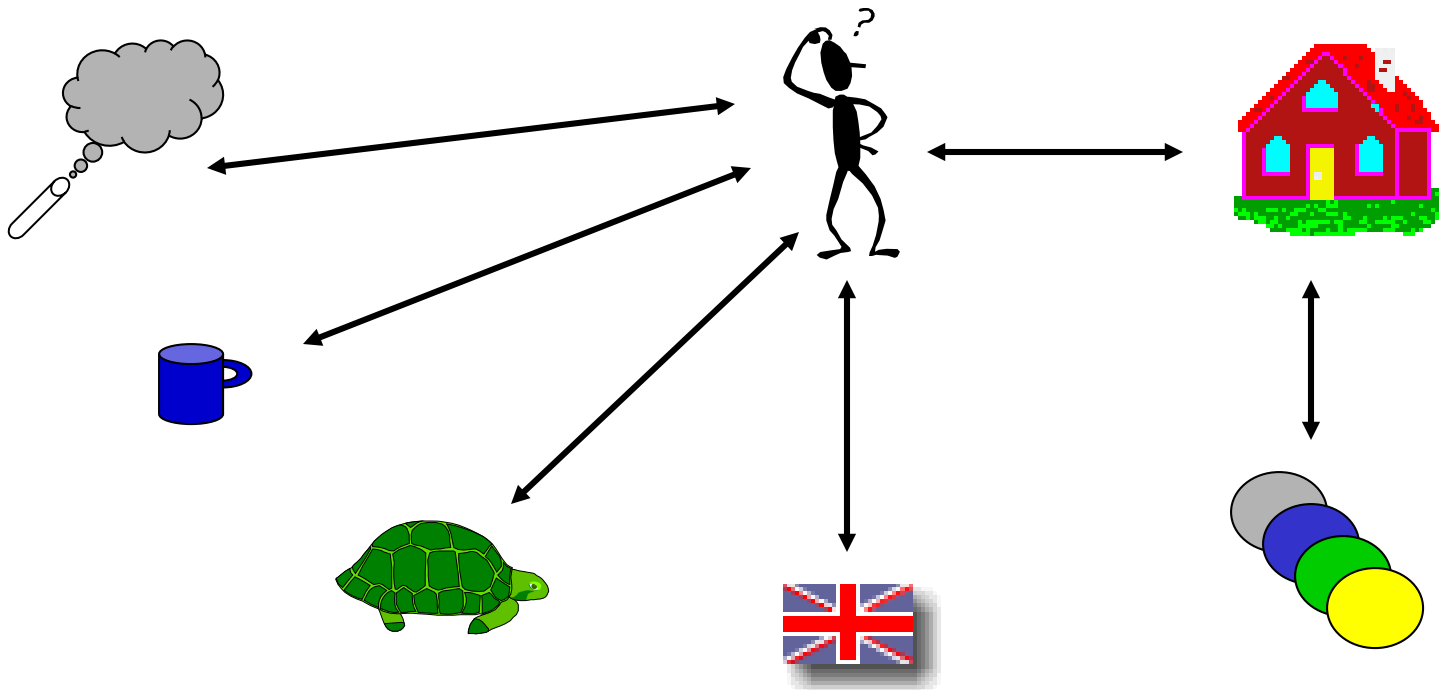
Puzzle (1)

Overall there are 14 clues that define the "world" of the puzzle:

1. The Englishman lives in the red house.
2. The Spaniard owns the dog.
3. Coffee is drunk in the green house.
4. The Ukrainian drinks tea.
5. The green house is immediately to the right of the ivory house.
6. The Winston smoker owns snails.
7. Kools are smoked in the yellow house.
8. Milk is drunk in the middle house.
9. The Norwegian lives in the leftmost house.
10. The man who smokes Chesterfield lives in the house next to the man with the fox.
11. Kools are smoked in the house next to the house where the horse is kept.
12. The Lucky Strike smoker drinks orange juice.
13. The Japanese smokes Parliaments.
14. The Norwegian lives next to the blue house.

The problem posed:

Who owns the zebra ?

Who drinks water ?
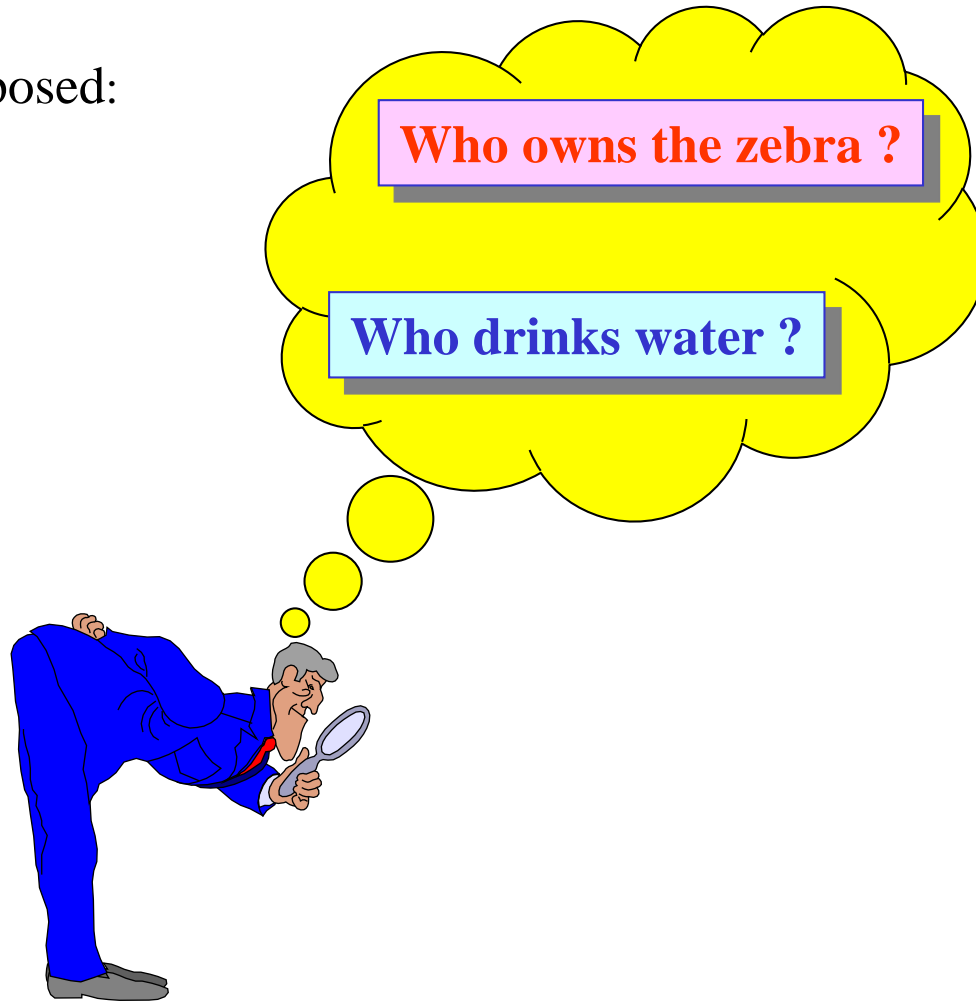
Systematic construction of the solution (by a human):

house	1	2	3	4	5

9.

14.

8.

color   blue

nationality   Norwegian

drink   milk

pet
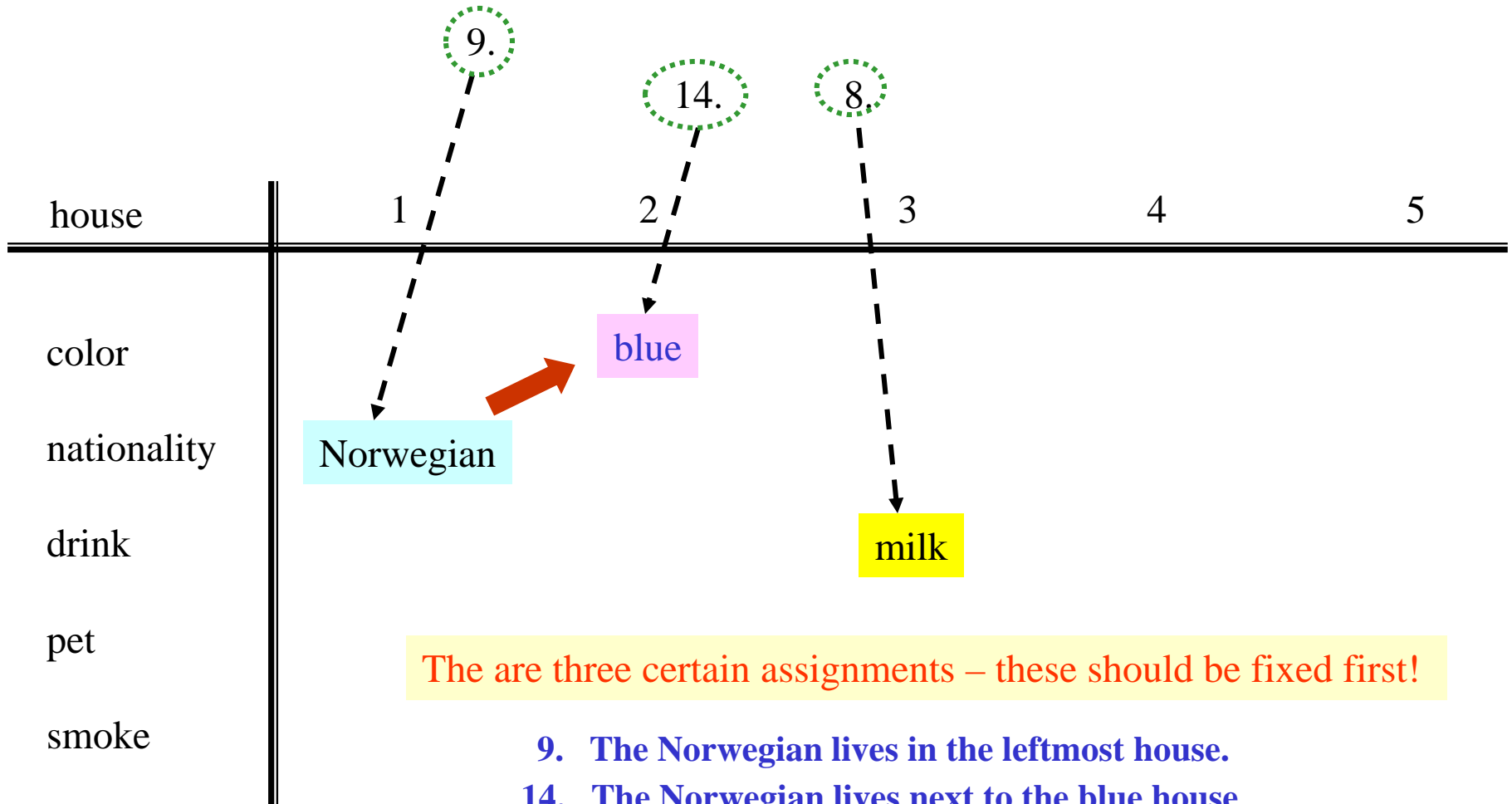
The are three certain assignments – these should be fixed first!

smoke

9. The Norwegian lives in the leftmost house.
14. The Norwegian lives next to the blue house.
8. Milk is drunk in the middle house.

Puzzle (4)

Condition 5:

5. The green house is immediately to the right of the ivory house.

… allows only two possibilities:

house	1	2	3	4	5
color		blue		ivory green	green
nationality	Norwegian				
drink			milk	coffee	coffee
pet					
smoke					

5.

?

3.

?

A direct consequence of this would be:

3. Coffee is drunk in the green house.
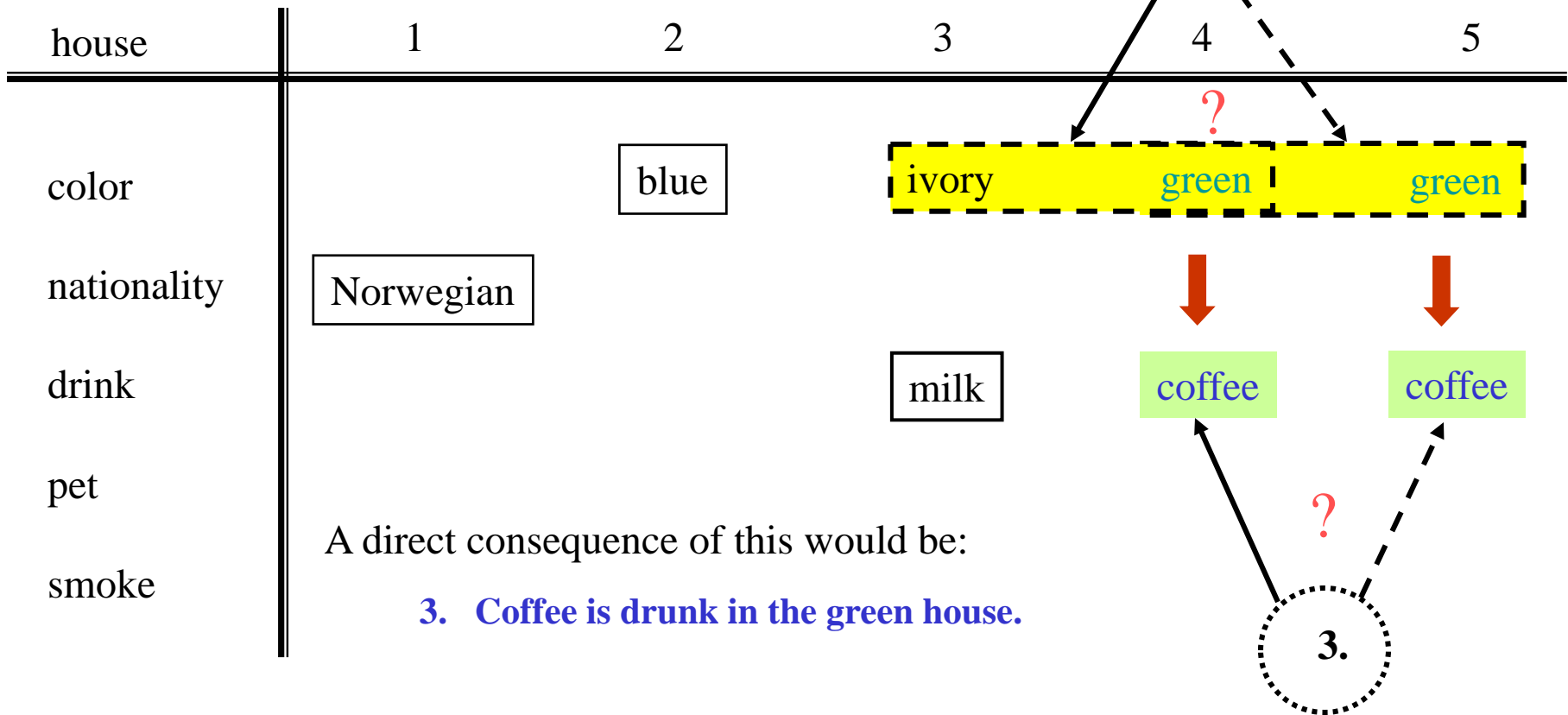
If the 1. solution for ivory/green is correct, necessarily the position for red:

1.

then also necessarily

1. The Englishman lives in the red house.

house	1	2	3	4	5
color	yellow	blue	ivory green		red
nationality	Norwegian				English
drink			milk	coffee	
pet		horse			
smoke	Kools				

7. Kools are smoked in the yellow house.

7.

11.

11. Kools are smoked in the house next to the house where the horse is kept.

etc. …

Puzzle (6)

Unique solution of the puzzle (to be found via several backtracking steps):

house	1	2	3	4	5
color	yellow	blue	red	ivory	green
nationality	Norwegian	Ukrainian	English	Spanish	Japanese
drink	**water**	tea	milk	juice	coffee
pet	fox	horse	snails	dog	**zebra**
smoke	Kools	Chesterfield	Winston	Lucky Strike	Parliaments

Puzzle: <u>one</u> possible specification in Prolog

```
right_of(R, L, [ L | [ R | _ ] ]).
right_of(R, L, [ _ | Rest ]) :- right_of(R, L, Rest).
next_to(X, Y, List) :- right_of(X, Y, List).
next_to(X, Y, List) :- right_of(Y, X, List).
zebra(Zebra_Owner) :-
```

8. ∧ 9.	Houses = [[_, norwegian, _, _, _], _, [_, _, milk, _, _], _, _],
1.	member([red, englishman, _, _, _], Houses),
2.	member([_, spaniard, _, dog, _], Houses),
3.	member([green, _, coffee, _, _], Houses),
4.	member([_, ukrainian, tea, _, _], Houses),
5.	right_of([green, _, _, _, _], [ivory, _, _, _, _], Houses),
6.	member([_, _, _, snails, winston], Houses),
7.	member([yellow, _, _, _, kools], Houses),
10.	next_to([_, _, _, _, chesterfield], [_, _, _, fox, _], Houses),
11.	next_to([_, _, _, _, kools], [_, _, _, horse, _], Houses),
12.	member([_, _, juice, _, lucky], Houses),
13.	member([_, japanese, _, _, parliaments], Houses),
14.	next_to([_, norwegian, _, _, _], [blue, _, _, _, _], Houses),
?	member([_, Zebra_Owner, _, zebra, _], Houses),
?	member([_, _, water, _, _], Houses).

History of Prolog

- Prolog as name is abbreviated from "Programming with logic".

- It is the most common logic programming language.

- Some history on Prolog:

1965:	John Alan Robinson provides theoretical foundations for theorem provers using the resolution calculus.
1972:	Alain Colmerauer (Marseilles) and his group develop Prolog.
in the '70s:	David D.H. Warren builds the first Prolog compiler.
1981–92:	5th Generation Computer Project in Japan (made Prolog "popular")

Literature on Prolog

- A lot of books and tutorials exist.

- The slides use a lot of examples from this book:
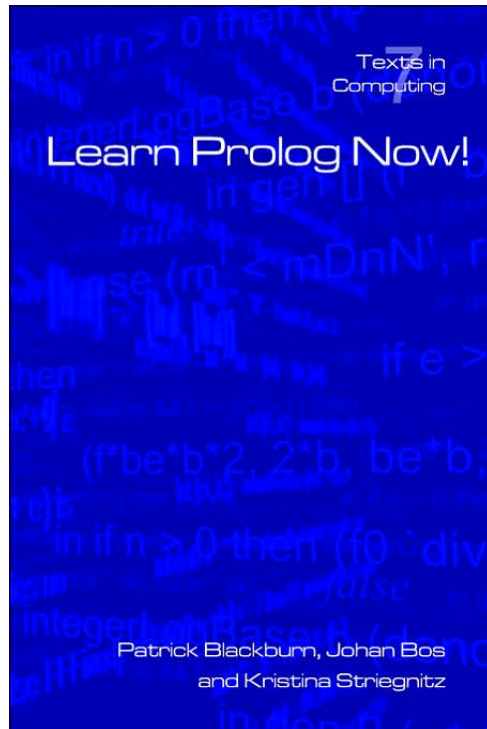
Patrick Blackburn, Johan Bos,
Kristina Striegnitz:
"Learn Prolog Now!"
College Publications, 2006

Programming Paradigms

Prolog Basics

- A kind of data base with a number of facts:

```prolog
woman(mia).
woman(jody).
woman(yolanda).
playsAirGuitar(jody).
```
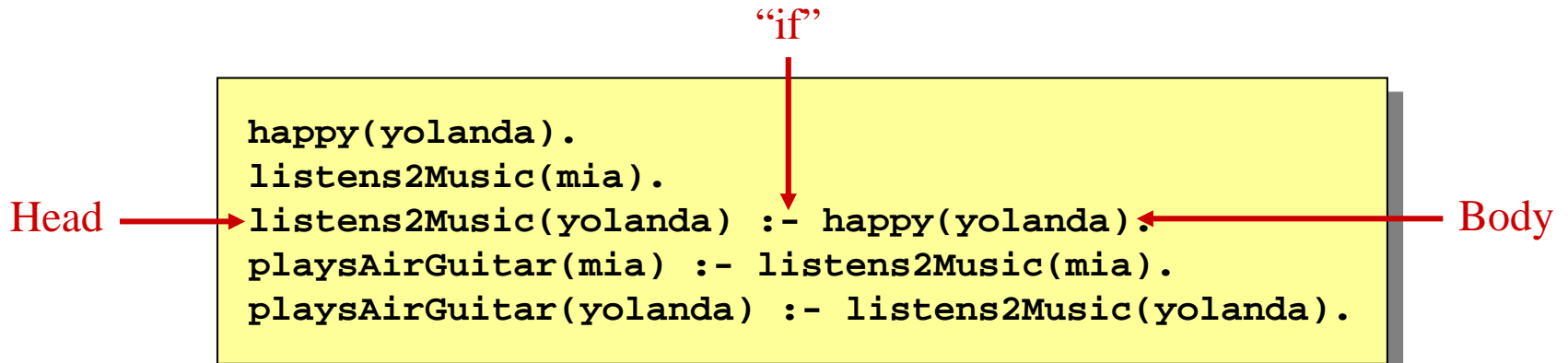
- Queries:

```prolog
?- woman(mia).          ← The dot is essential!
true.

?- playsAirGuitar(jody).
true.

?- playsAirGuitar(mia).
false.

?- playsAirGuitar(vincent).
false.

?- playsPiano(jody).
false.                  ← or an error message
```

"if"

```
happy(yolanda).
listens2Music(mia).
listens2Music(yolanda) :- happy(yolanda).
playsAirGuitar(mia) :- listens2Music(mia).
playsAirGuitar(yolanda) :- listens2Music(yolanda).
```

Head   Body

- Queries:

```
?- playsAirGuitar(mia).
true.

?- playsAirGuitar(yolanda).
true.
```

because of:

```
happy(yolanda)
    ⇒ listens2Music(yolanda)
    ⇒ playsAirGuitar(yolanda)
```

More complex rules

"and"

```
happy(vincent).
listens2Music(butch).
playsAirGuitar(vincent) :- listens2Music(vincent),
                           happy(vincent).
playsAirGuitar(butch) :- happy(butch).
playsAirGuitar(butch) :- listens2Music(butch).
```

Alternatives →

- Queries:

```
?- playsAirGuitar(vincent).
false.

?- playsAirGuitar(butch).
true.
```

- Alternative notation:

"or"

```
...
playsAirGuitar(butch) :- happy(butch);
                         listens2Music(butch).
```

Relations, and more complex queries

```
woman(mia).
woman(jody).
woman(yolanda).

loves(vincent,mia).
loves(marsellus,mia).
loves(mia,vincent).
loves(vincent,vincent).
```

multi-ary (concretely, binary) predicate

- Queries:

```
?- woman(X).
X = mia ;
X = jody ;
X = yolanda.

?- loves(vincent,X).
X = mia ;
X = vincent.

?- loves(vincent,X), woman(X).
X = mia ;
false.
```

semicolon entered by user

Variables in rules (not just in queries)

```
loves(vincent,mia).
loves(marsellus,mia).
loves(mia,vincent).

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

- Queries:

```
?- jealous(marsellus,X).
X = vincent ;
X = marsellus ;
false.

?- jealous(X,_).
X = vincent ;
X = vincent ;
X = marsellus ;
X = marsellus ;
X = mia.
```

anonymous variable

Variables in rules (not just in queries)

```prolog
loves(vincent,mia).
loves(marsellus,mia).
loves(mia,vincent).

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

important that at end

- Queries:

```prolog
?- jealous(marsellus,X).
X = vincent ;
false.

?- jealous(X,_).
X = vincent ;
X = marsellus ;
false.

?- jealous(X,Y).
X = vincent,
Y = marsellus ;
X = marsellus,
Y = vincent ;
false.
```

Some observations on variables

```prolog
loves(vincent,mia).
loves(marsellus,mia).
loves(mia,vincent).

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

- Variables in rules and in queries are independent from each other.

```prolog
?- jealous(marsellus,X).
X = vincent ;
false.
```

- Within a rule or a query, the same variables represent the same objects.

- But different variables do not necessarily represent different objects.

- It is possible to have several occurrences of the same variable in a rule's head!

- In a rule's body there can be variables that do not occur in its head!

Intuition on "free" variables

```
loves(vincent,mia).
loves(marsellus,mia).
loves(mia,vincent).

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

- What is the "logical" interpretation of **Z** above? (or of the whole rule?)

- Possibly, for arbitrary (but fixed) **X**, **Y**:
  <u>if</u> for every choice of **Z** holds: **loves(X,Z)**, and **loves(Y,Z)**, and **X \= Y**,
  <u>then</u> also holds: **jealous(X,Y)**

- Or, for arbitrary (but fixed) **X**, **Y**:
  for every choice of **Z** holds: <u>if</u> **loves(X,Z)**, and **loves(Y,Z)**, and **X \= Y**,
  <u>then</u> also holds: **jealous(X,Y)**

???

```
loves(vincent,mia).
loves(marsellus,mia).
loves(mia,vincent).

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

- What is the "logical" interpretation of **Z** above? (or of the whole rule?)

- Possibly, for arbitrary (but fixed) **X** , **Y**:
  <u>if</u> for every choice of **Z** holds: **loves(X,Z)**, and **loves(Y,Z)**, and **X \= Y**,
  <u>then</u> also holds: **jealous(X,Y)**

- Or, for arbitrary (but fixed) **X** , **Y**:
  for every choice of **Z** holds: <u>if</u> **loves(X,Z)**, and **loves(Y,Z)**, and **X \= Y**,
  <u>then</u> also holds: **jealous(X,Y)**

Intuition on "free" variables

```
loves(vincent,mia).
loves(marsellus,mia).
loves(mia,vincent).

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

- What is the "logical" interpretation of **Z** above? (or of the whole rule?)

- Or, for arbitrary (but fixed) **X** , **Y**:
  for <u>every</u> choice of **Z** holds: <u>if</u> **loves(X,Z)**, and **loves(Y,Z)**, and **X \= Y**,
  <u>then</u> also holds: **jealous(X,Y)**

- Logically equivalent, for arbitrary (but fixed) **X** , **Y**:
  <u>if</u> for <u>any</u> choice of **Z** holds: **loves(X,Z)**, and **loves(Y,Z)**, and **X \= Y**,
  <u>then</u> also holds: **jealous(X,Y)**

Programming Paradigms

Operational intuition for Prolog

Operationalisation?

Specification (program) ≡
  relation definitions

```
istVaterVon(kurt,fritz).
istVaterVon(fritz,paul).
istVaterVon(fritz,hans).

istGrossvaterVon(G,E) :-
      istVaterVon(G,V),istVaterVon(V,E).
istGrossvaterVon(G,E) :-
      istVaterVon(G,M),istMutterVon(M,E).
```

```
?- istGrossvaterVon(kurt,X)
   ⤳ ...
   ⤳ ...
   ⤳ ...
   ⤳ ...
   ⤳  X = paul ;  X = hans
```

Input: a query

(repeated) reduction

Output: variable substitution(s)

Principle: reduction to a subproblem

istGrossvaterVon(kurt, X)

matching/
parameter
passing

```
                              istVaterVon(kurt,fritz).
                              istVaterVon(fritz,paul).
                              istVaterVon(fritz,hans).

istGrossvaterVon(G,E) :- istVaterVon(G,V),istVaterVon(V,E).

istGrossvaterVon(G,E) :- istVaterVon(G,M),istMutterVon(M,E).
```

1st reduction

istVaterVon(kurt,V)

Principle: reduction to a subproblem, where new subqueries are found <u>from left to right</u>!

istGrossvaterVon(kurt, X)

matching/
parameter
passing

```
                            istVaterVon(kurt,fritz).
                            istVaterVon(fritz,paul).
                            istVaterVon(fritz,hans).

istGrossvaterVon(G,E) :- istVaterVon(G,V),istVaterVon(V,E).

istGrossvaterVon(G,E) :- istVaterVon(G,M),istMutterVon(M,E).
```

istVaterVon(kurt,fritz)

2nd reduction

istVaterVon(fritz,E)

Principle: reduction to a subproblem

```
istGrossvaterVon(kurt, X)
```

matching/
parameter
passing

return of
result
parameter

```
                              istVaterVon(kurt,fritz).
                              istVaterVon(fritz,paul).
                              istVaterVon(fritz,hans).


istGrossvaterVon(G,E) :- istVaterVon(G,V),istVaterVon(V,E).

istGrossvaterVon(G,E) :- istVaterVon(G,M),istMutterVon(M,E).
```

```
istVaterVon(kurt,fritz)
```

E = paul

```
istVaterVon(fritz,paul)
```

- Prolog always looks for matching rules or facts <u>from top to bottom</u> in the program.

subquery:

```
istVaterVon(fritz,E)
```

```
istVaterVon(kurt,fritz).
istVaterVon(fritz,paul).
istVaterVon(fritz,hans).
```

solution:

```
E = paul
```

- Since a relation generally is not a unique mapping, further answers for a (sub)query may exist. Prolog finds those using backtracking:

re-evaluation:

```
istVaterVon(fritz,E)
```

position of last solution – that is where <u>search continues</u>

```
istVaterVon(kurt,fritz).
istVaterVon(fritz,paul).
istVaterVon(fritz,hans).
```

solution:

```
E = paul;
E = hans
```

Principle: reduction to a subproblem

istGrossvaterVon(kurt, X)

matching/
parameter
passing

return of
result
parameter

```
                            istVaterVon(kurt,fritz).
                            istVaterVon(fritz,paul).
                            istVaterVon(fritz,hans).

istGrossvaterVon(G,E) :- istVaterVon(G,V),istVaterVon(V,E).

istGrossvaterVon(G,E) :- istVaterVon(G,M),istMutterVon(M,E).
```

istVaterVon(kurt,fritz)

E = hans

istVaterVon(fritz,hans)

The backtracking also concerns other matching rules:

```
istGrossvaterVon(kurt, X)
```

matching/
parameter
passing

```
istVaterVon(kurt,fritz).
istVaterVon(fritz,paul).
istVaterVon(fritz,hans).

istGrossvaterVon(G,E) :- istVaterVon(G,V),istVaterVon(V,E).

istGrossvaterVon(G,E) :- istVaterVon(G,M),istMutterVon(M,E).
```

3rd reduction

```
istVaterVon(kurt,M)
```

Failure!

```
istMutterVon(fritz,E)
```

Operationalisation on the example, presented differently

```
istVaterVon(kurt,fritz).
istVaterVon(fritz,paul).
istVaterVon(fritz,hans).

istGrossvaterVon(G,E) :-
      istVaterVon(G,V),istVaterVon(V,E).
istGrossvaterVon(G,E) :-
      istVaterVon(G,M),istMutterVon(M,E).
```

?- istGrossvaterVon(kurt, X).
?- istVaterVon(kurt, V), istVaterVon(V, X).
?- istVaterVon(fritz, X).

X = paul: ?- .

Compare (within a Prolog system): use of ?- trace.

Operationalisation on the example, presented differently

```
istVaterVon(kurt,fritz).
istVaterVon(fritz,paul).
istVaterVon(fritz,hans).

istGrossvaterVon(G,E) :-
        istVaterVon(G,V),istVaterVon(V,E).

istGrossvaterVon(G,E) :-
        istVaterVon(G,M),istMutterVon(M,E).
```
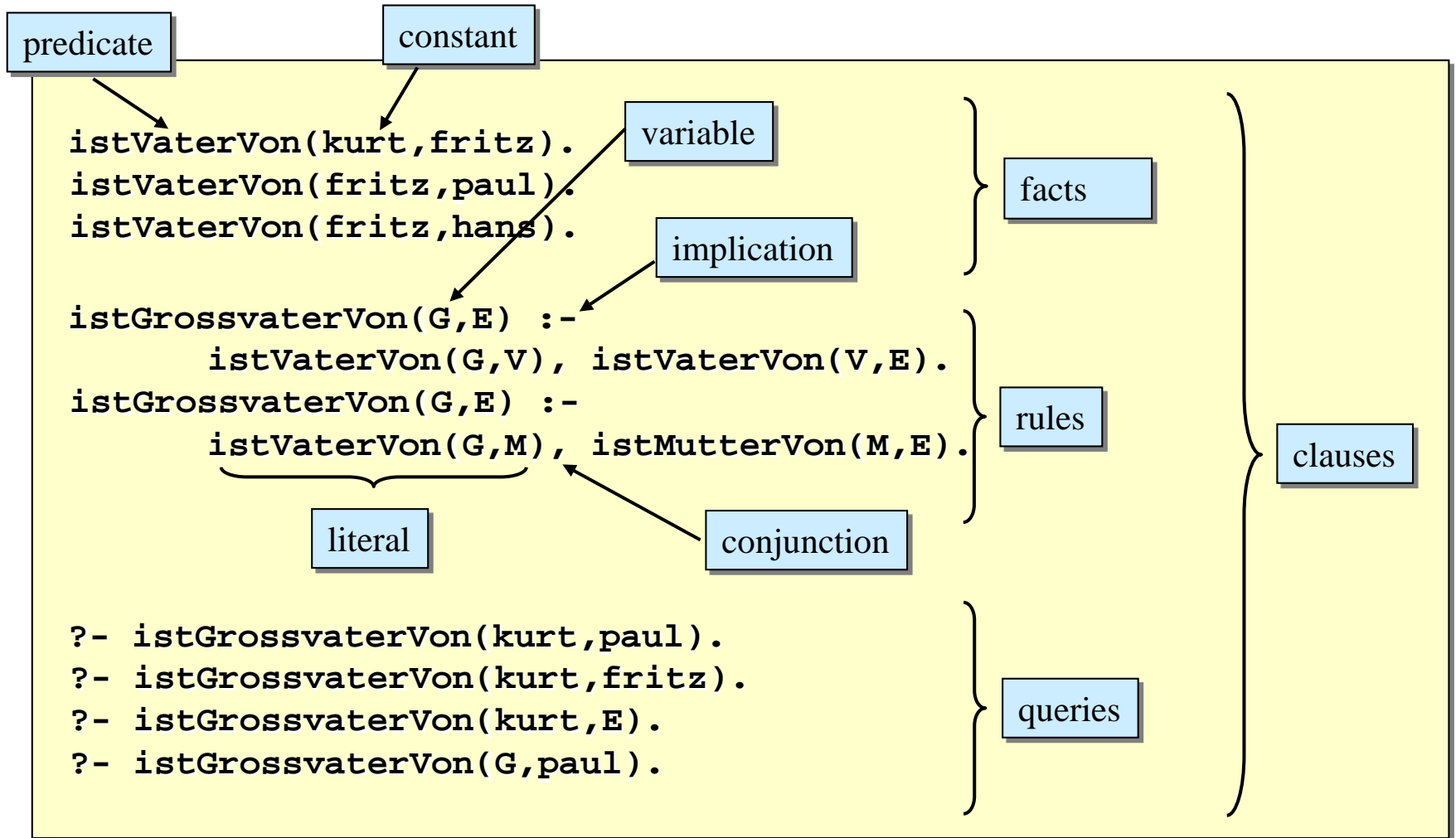
?- istGrossvaterVon(kurt, X).
?- istVaterVon(kurt, V), istVaterVon(V, X).
?- istVaterVon(fritz, X).

X = paul: ?- .
X = hans: ?- .

Compare (within a Prolog system): use of ?- trace.

Operationalisation on the example, presented differently

```
istVaterVon(kurt,fritz).
istVaterVon(fritz,paul).
istVaterVon(fritz,hans).

istGrossvaterVon(G,E) :-
        istVaterVon(G,V),istVaterVon(V,E).
istGrossvaterVon(G,E) :-
        istVaterVon(G,M),istMutterVon(M,E).
```

?- istGrossvaterVon(kurt, X).
?- istVaterVon(kurt, V), istVaterVon(V, X).
?- istVaterVon(fritz, X).

X = paul: ?- .
X = hans: ?- .
?- istVaterVon(kurt, M), istMutterVon(M, X).
?- istMutterVon(fritz, X).
Failure!

Compare (within a Prolog system): use of ?- trace.

Programming Paradigms

Syntax ingredients for Prolog

Syntax / notions in Prolog

predicate | constant

```
istVaterVon(kurt,fritz).
istVaterVon(fritz,paul).
istVaterVon(fritz,hans).
```

variable

facts

```
istGrossvaterVon(G,E) :-
      istVaterVon(G,V), istVaterVon(V,E).
istGrossvaterVon(G,E) :-
      istVaterVon(G,M), istMutterVon(M,E).
```

implication

rules

clauses

literal | conjunction

```
?- istGrossvaterVon(kurt,paul).
?- istGrossvaterVon(kurt,fritz).
?- istGrossvaterVon(kurt,E).
?- istGrossvaterVon(G,paul).
```

queries

Language objects in Prolog

- To build clauses, Prolog uses different kinds of objects:

  - constants   (numbers, strings, …)

  - variables   (X,Y, ThisThing, …)

  - operator terms   (… 1 + 3 * 4 …)

  - structures   (date(27,11,2007), person(fritz, mueller), …

                        composite, recursive, "infinite", …)

- Note: Prolog has no type system!

Constants in Prolog

- Numbers

  ```
  -17   -2.67e+021   0   1   99.9     512
  ```

- Atoms, i.e. strings that satisfy one of these rules:

  1. The string starts with a lower case letter, followed by arbitrarily many lower or upper case letters, numbers and underscores '_'.

  2. The string starts and ends with an apostrophe ('). In between, there can be arbitrary characters. If an apostrophe should appear in the string, it has to be denoted twice.

  3. The string consists only of symbols.
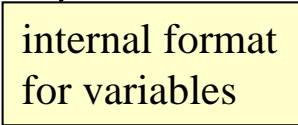
correct:	**fritz**	**new_york**	**:-**	**-->**	**'I don''t know!'**
wrong:	**Fritz**	**new-york**	**_xyz**	**123**	

Variables in Prolog

- Variables:

    - Name starts with an upper case letter or an underscore '_'.

    - Examples: | **Country   Year   M   V   _45   _G107   _europe** |

        internal format
        for variables

- Anonymous variables (simply '_', even if several anonymous variables):
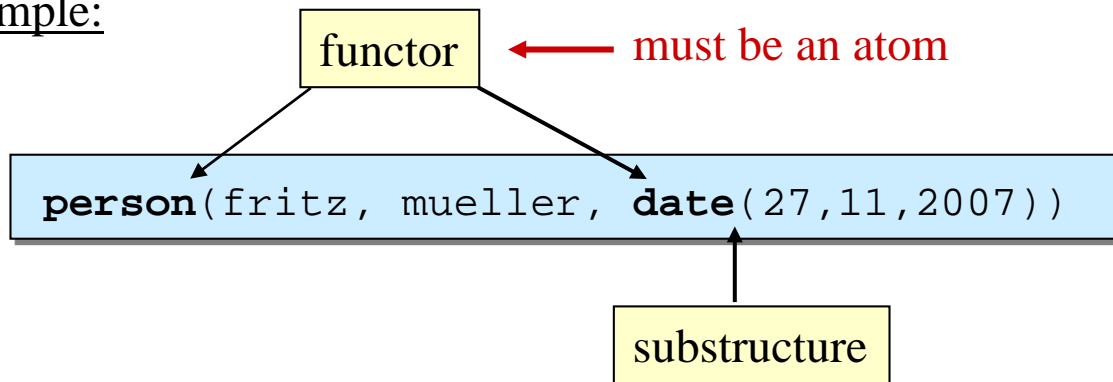
    - if the object is not of interest:

        ```
        ?- istVaterVon(_,fritz).
        ```

Structures in Prolog

- Structures represent objects that are made up of several other objects.

- Example:


```
functor  ←  must be an atom

person(fritz, mueller, date(27,11,2007))

                                  substructure
```

functors: **person/3, date/3** (notation for arity)

- Through this, modelling of essentially "algebraic data types" – but not actually typed. So, **person(1,2,'a')** would also be a legal structure.

- Arbitrary nesting depth allowed – in principle infinite.

Language objects in Prolog

Predefined syntax for special structures:

- There is a predefined "list type" as recursive data structure:

```
[1,2,a]   .(1,.(2,.(a,[])))   [1|[2,a]]   [1,2|[a]]   [1,2|.(a,[])]
```

- Character strings are represented as lists of ASCII-Codes:

$$\text{"Prolog"} = [80, 114, 111, 108, 111, 103]$$
$$= .(80, . (114, . (111, . (108, . (111, . (103, [ ])))))))$$

Operators:

- Operators are functors made from symbols and written infix.

- Example: in arithmetic expressions

  - Mathematical functions are defined as operators.

  - `1 + 3 * 4` is to be read as this structure: `+(1,*(3,4))`

Collective notion "terms":

- Terms are constants, variables or structures:

```
fritz
27
MM
[europe, asia, africa | Rest]
person(fritz, Lastname, date(27, MM, 2007))
```

- A ground term is a term that does not contain variables:

```
person(fritz, mueller, date(27, 11, 2007))
```

Programming Paradigms

More Prolog examples

Simple example for working with data structures

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

```
?- add(s(0),s(0),s(s(0))).
true.

?- add(s(0),s(0),N).
N = s(s(0)) ;
false.
```

- Recall, in Haskell:

```
data Nat = Zero | Succ Nat

add :: Nat → Nat → Nat
add Zero      x = x
add (Succ x)  y = Succ (add x y)
```

Systematic connection/derivation?

- Essential difference Haskell/Prolog:

<div align="center">

Functions   vs.   Predicates/Relations

f x y = z   "corresponds to"   `p(X,Y,Z).`

</div>

- First a somewhat naïve attempt to exploit this correspondence:

<div align="center">

add Zero   x = x

↓

add(Zero, x, x)

↓

`add(0,X,X).`

add (Succ x) y = Succ (add x y)

↓

add(Succ x, y, Succ (add x y))

↓

???

</div>

Systematic connection/derivation?

- Essential difference Haskell/Prolog:

  Functions   vs.   Predicates/Relations

  $f\ x\ y = z$ "corresponds to"   `p(X,Y,Z).`

- Systematically <u>avoiding nested function calls</u>:

  add (Succ x) y = Succ (add x y)

  ↓

  add (Succ x) y = Succ z   where z = add x y

  ↓

  add(Succ x, y, Succ z)   if   add(x, y, z)

  ↓

  `add(s(X),Y,s(Z)) :- add(X,Y,Z).`

On the flexibility of Prolog predicates

```prolog
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

```prolog
?- add(N,M,s(s(0))).
N = 0,
M = s(s(0)) ;
N = s(0),
M = s(0) ;
N = s(s(0)),
M = 0 ;
false.

?- add(N,s(0),s(s(0))).
N = s(0) ;
false.

?- add(N,M,O).
```

???

On the flexibility of Prolog predicates

```prolog
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

sub(X,Y,Z) :- add(Z,Y,X).
```

```prolog
?- sub(s(s(0)),s(0),N).
N = s(0) ;
false.

?- sub(N,M,s(0)).
N = s(M) ;
false.
```

Another example

Computing the length of a list in Haskell:

```
length []     =   0
length (x:xs) =   length xs + 1
```
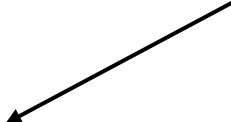
Computing the length of a list in Prolog:

```
length([],0).
length([X|Xs],N) :- length(Xs,M), N is M+1.
```

```
?- length([1,2,a],Res).
   Res = 3.

?- length(List,3).
   List = [_G331, _G334, _G337]
```

list with 3 arbitrary
(variable) elements

<u>Caution</u>: If instead of:

```
length([],0).
length([X|Xs],N) :- length(Xs,M), N is M+1.
```

we use:

```
length([],0).
length([X|Xs],M+1) :- length(Xs,M).
```

then:
```
?- length([1,2,a],Res).
   Res = 0+1+1+1.

?- length(List,3).
   false.

?- length(List,0+1+1+1).
   List = [_G331, _G334, _G337].
```

partition :: Int $\rightarrow$ [Int] $\rightarrow$ ([Int], [Int])

…

quicksort [ ]   = [ ]
quicksort (h : t) = quicksort l_1 ++ h : quicksort l_2
           where (l_1, l_2) = partition h t

```
quicksort([], []).
quicksort([H|T], List) :-
    partition(H, T, L1, L2),
    quicksort(L1, LS),
    quicksort(L2, LG),
    append(LS, [H|LG], List).
```

quicksort [ ]   = [ ]
quicksort (h : t) = ls ++ h : quicksort l_2
           where (l_1, l_2) = partition h t
                 ls = quicksort l_1

quicksort [ ]   = [ ]
quicksort (h : t) = ls ++ h : lg
           where (l_1, l_2) = partition h t
                 ls = quicksort l_1
                 lg = quicksort l_2

quicksort [ ]   = [ ]
quicksort (h : t) = list
           where (l_1, l_2) = partition h t
                 ls = quicksort l_1
                 lg = quicksort l_2
                 list = ls ++ h : lg

Programming Paradigms

Declarative semantics of Prolog

Declarative semantics of Prolog

What is the "mathematical" meaning/semantics of a Prolog program?

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

Logical interpretation:

$$(\forall X.\ \text{add}(0,X,X))$$
$$\wedge\ (\forall X, Y, Z.\ \text{add}(X,Y,Z) \Rightarrow \text{add}(s(X),Y,s(Z)))$$

To give meaning to such formulas, the study of logics uses models:

- starting from a set of mathematical objects
- interpretation of constants (like "0") as elements of the above set, and of functors (like "s(…)") as functions thereover
- interpretation of predicates (like "add(…)") as relations between objects
- assignment of truth values to formulas according to certain rules
- consideration only of interpretations that make all given formulas true

Semantics of a program would be given by all statements/relationships that hold in all models for the program.

Herbrand models

Important: There is always a kind of "universal model".

Idea: Interpretation as simple as possible, namely purely syntactic.
            Neither functors nor predicates really "do" anything.   the Herbrand universe

So: set of objects   = all ground terms (over implicitly given signature)
            interpretation of functors   = syntactical application on terms
            interpretation of predicates = some set of applications of predicate symbols
                                         on ground terms

a Herbrand interpretation

Example:

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

Signature: `0` (of arity 0), `s` (of arity 1)
Herbrand universe: $\{0, s(0), s(s(0)), s(s(s(0))), \ldots\}$ (without predicate symbols!)
the Herbrand base: $\{add(0,0,0), add(0,0,s(0)), add(0,s(0),0), \ldots\}$
               (all applications of predicate symbols on terms from Herbrand universe)

Herbrand models

A Herbrand interpretation is some subset of the Herbrand base.

Example:

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

Herbrand interpretation 1: $\{$`add(0,0,0)`, `add(0,0,s(0))`, `add(0,s(0),0)`, ...$\}$

Herbrand interpretation 2: $\varnothing$

Herbrand interpretation 3: $\{$`add(0,0,0)`, `add(0,s(0),s(0))`,
                        `add(s(0),0,s(0))`, `add(s(0),s(0),s(s(0)))`, ...$\}$

Our aim is a Herbrand interpretation that makes true all formulas given by the program, but does not unnecessarily make anything else additionally true.

A Herbrand interpretation is a model for a program if for every complete instantiation (i.e., no variables left)

$$L_0 \text{ :- } L_1, \; L_2, \; \ldots, \; L_n$$

of each fact/rule it holds: if $L_1, \; L_2, \; \ldots, \; L_n$ is in the interpretation, then so is L_0.

Example:

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

- The Herbrand base is (always) a model.

- The Herbrand interpretation $\varnothing = \{\}$ is (here) no model.

- The interpretation $\{$ `add(s`^{i}`(0),s`^{j}`(0),s`$^{i+j}$`(0))` $\mid$ i , $j \geq 0$ $\}$ is here a model.

Smallest Herbrand model

The declarative meaning of a Prolog program is its smallest (via the subset relation) Herbrand interpretation that is a model!

For the example:

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

$$\{ \mathtt{add(s^i(0),s^j(0),s^{i+j}(0))} \mid \mathtt{i}, \mathtt{j} \geq 0 \}$$

Generally:

Is there always such a smallest model?

Yes, since models for programs consisting of so-called Horn clauses (exactly the kind of clauses in <u>Prolog without negation</u>) are closed under intersection!

Smallest Herbrand model

Can one actually compute, in a constructive fashion, the smallest Herbrand model?

Yes, using the "immediate consequence operator": T_P

Definition: T_P takes a Herbrand interpretation I and produces all ground literals
(elements of the Herbrand base) `L`$_0$ for which `L`$_1$`, ` `L`$_2$`, ...,` `L`$_n$
exist in I such that `L`$_0$` :- ` `L`$_1$`, ` `L`$_2$`, ...,` `L`$_n$ is a complete instantiation
of any of the given program clauses (facts/rules).

Obviously: A Herbrand interpretation I is a model if and only if $T_P(I)$ is a subset of I.

Moreover: The smallest Herbrand model is obtained as fixpoint/limit of the sequence

$$\varnothing \, , \; T_P(\varnothing) \, , \; T_P(T_P(\varnothing)) \, , \; T_P(T_P(T_P(\varnothing))) \, , \ldots$$

Smallest Herbrand model

On the example:

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

$T_P(\varnothing) = \{add(0,0,0), add(0,s(0),s(0)), add(0,s(s(0)),s(s(0))), \ldots\}$

$T_P(T_P(\varnothing)) = T_P(\varnothing) \cup \{add(s(0),0,s(0)), add(s(0),s(0),s(s(0))),$
$add(s(0),s(s(0)),s(s(s(0)))), \ldots\}$

$T_P(T_P(T_P(\varnothing))) = T_P(T_P(\varnothing)) \cup \{add(s(s(0)),0,s(s(0))),$
$add(s(s(0)),s(0),s(s(s(0)))),$
$add(s(s(0)),s(s(0)),s(s(s(s(0))))), \ldots\}$

…

For which kind of Prolog programs can one work with the T_P -semantics?
- no arithmetics, no **is**
- no **\\=**, no **not**
- generally, none of the "non-logical" features (not introduced in the lecture)

But for example programs like this (and would also work for mutual recursion):

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(_),0,0).
mult(s(X),s(Y),s(Z)) :- mult(X,s(Y),U), add(Y,U,Z).
```

$T_P(\varnothing) = \{\mathbf{add(0,0,0)}, \mathbf{add(0,s(0),s(0))}, ...\} \cup \{\mathbf{mult(0,0,0)},$
$\qquad \mathbf{mult(0,s(0),0)}, ...\} \cup \{\mathbf{mult(s(0),0,0)}, ...\}$

$T_P(T_P(\varnothing)) = T_P(\varnothing) \cup \{\mathbf{add(s(0),0,s(0))}, \mathbf{add(s(0),s(0),s(s(0)))}, ...\}$
$\qquad \cup \{\mathbf{mult(s(0),s(0),s(0))}\}$

Applicability of the semantics based on Herbrand models

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(_),0,0).
mult(s(X),s(Y),s(Z)) :- mult(X,s(Y),U), add(Y,U,Z).
```

$T_P(\varnothing) = \{$add(0,0,0), add(0,s(0),s(0)), $\dots\} \cup \{$mult(0,0,0),
            mult(0,s(0),0), $\dots\} \cup \{$mult(s(0),0,0), $\dots\}$

$T_P(T_P(\varnothing)) = T_P(\varnothing) \cup \{$add(s(0),0,s(0)), add(s(0),s(0),s(s(0))), $\dots\}$
            $\cup \{$mult(s(0),s(0),s(0))$\}$

$T_P(T_P(T_P(\varnothing))) = T_P(T_P(\varnothing)) \cup \{$add(s(s(0)),0,s(s(0))), $\dots\}$
            $\cup \{$mult(s(0),s(s(0)),s(s(0))),
                mult(s(s(0)),s(0),s(s(0)))$\}$

$T_P{}^4(\varnothing) = T_P{}^3(\varnothing) \cup \{$add(s^3(0),0,s^3(0)), add(s^3(0),s(0),s^4(0)), $\dots\}$
            $\cup \{$mult(s(0),s^3(0),s^3(0)), mult(s^2(0),s^2(0),s^4(0)),
                mult(s^3(0),s(0),s^3(0))$\}$

Applicability of the semantics based on Herbrand models

The declarative semantics:

- is only applicable to certain, "purely logical", programs

- does not directly describe the behaviour for queries containing variables

- is mathematically simpler than the still to be introduced operational semantics

- can be related to that operational semantics appropriately

- is insensitive against changes to the order of, and within, facts and rules (!)

Programming Paradigms

Operational semantics of Prolog

Motivation: Observing some not so nice (not so "logical"?) effects

```prolog
direct(frankfurt,san_francisco).
direct(frankfurt,chicago).
direct(san_francisco,honolulu).
direct(honolulu,maui).

connection(X, Y) :- direct(X, Y).
connection(X, Y) :- direct(X, Z), connection(Z, Y).
```

```prolog
?- connection(frankfurt,maui).
true.

?- connection(san_francisco,X).
X = honolulu ;
X = maui ;
false.

?- connection(maui,X).
false.
```

Motivation: Observing some not so nice (not so "logical"?) effects

```
direct(frankfurt,san_francisco).
direct(frankfurt,chicago).
direct(san_francisco,honolulu).
direct(honolulu,maui).

connection(X, Y) :- connection(X, Z), direct(Z, Y).
connection(X, Y) :- direct(X, Y).
```

```
?- connection(frankfurt,maui).
ERROR: Out of local stack
```

- Apparently, the implicit logical operations are not commutative.

- So related to program execution, there must be more than the purely logical reading.

Somewhat more subtle…

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

sub(X,Y,Z) :- add(Z,Y,X).
```

```
?- sub(N,M,s(0)).
N = s(M) ;
false.
```

```
add(X,0,X).
add(X,s(Y),s(Z)) :- add(X,Y,Z).

sub(X,Y,Z) :- add(Z,Y,X).
```

```
?- sub(s(s(0)),s(0),N).
N = s(0) ;
false.

?- sub(N,M,s(0)).
N = s(0),
M = 0 ;
N = s(s(0)),
M = s(0) ;
```

So the choice/treatment of
the order of arguments in
definitions affects the
quality of results.

...

… and (thus) sometimes less flexibility than desired

The nicely descriptive solution:

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- mult(X,Y,U),add(U,Y,Z).
```

works very well for various kinds of queries:

```
?- mult(s(s(0)),s(s(s(0))),N).
N = s(s(s(s(s(s(0)))))).

?- mult(s(s(0)),N,s(s(s(s(0))))).
N = s(s(0)) ;
false.
```

We say that `mult` supports the "call modes" `mult(+X,+Y,?Z)` and `mult(+X,?Y,+Z)`

But there are also "outliers":

```
?- mult(N,M,s(s(s(s(0))))).
N = s(0),
M = s(s(s(s(0)))) ;
N = s(s(0)),
M = s(s(0)) ;
abort
```

… but not `mult(?X,?Y,+Z)`.

otherwise infinite search

… and (thus) sometimes less flexibility than desired

Whereas with just addition:

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

the analogous call mode seemed to work pretty well:

```
?- add(N,M,s(s(s(0)))).
N = 0,
M = s(s(s(0))) ;
N = s(0),
M = s(s(0)) ;
N = s(s(0)),
M = s(0) ;
N = s(s(s(0))),
M = 0 ;
false.
```

Indeed, `add` supports all call modes, even `add(?X,?Y,?Z)`.

1. So why the difference?

2. And what can we do to also let `mult` function this way?

And now it gets really "strange":

```
loves(vincent,mia).
loves(marsellus,mia).
loves(mia,vincent).

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

small change

```
...

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

```
?- jealous(marsellus,X).
false.

?- jealous(X,_).
false.

?- jealous(X,Y).
false.
```

Whereas before the small change, we got meaningful results for these queries!

Operational semantics of Prolog

To investigate all these phenomena, we have to consider the concrete execution mechanism of Prolog.

Ingredients for this discussion of the operational semantics, considered in what follows:

1. Unification

2. Resolution

3. Derivation trees

Programming Paradigms

Unification

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

```
?- add(s(s(0)),s(0),s(s(s(0)))).
?- add(s(0),s(0),s(s(0))).
?- add(0,s(0),s(0)).
?- .
true.
```

But what about "output variables"?

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

?

```
?- add(s(s(0)),s(0),N).
```

Equality "**=**" as binary Prolog predicate that accomplishes a lot:

- performing comparisons on ground terms (terms without variables), e.g.:

s(0)=s(0)	$\Rightarrow$ **true**
s(0)=s(s(0))	$\Rightarrow$ **false**

- accepting bindings of variables, e.g.:

N=0	$\Rightarrow$ **true**
N=s(U)	$\Rightarrow$ **true**
s(0)=N	$\Rightarrow$ **true**
M=V	$\Rightarrow$ **true**

- structurally matching and binding, e.g.:

s(s(0))=s(V)	$\Rightarrow$ **V=s(0)**
s(U)=s(0)	$\Rightarrow$ **U=0**

- "collecting"/combining bindings, e.g.:

N=s(V), M=V	$\Rightarrow$ **N=s(M)**

Equality of terms (1)

- Checking equality of ground terms:

```
europe = europe ?                                    yes

person(fritz,mueller) = person(fritz,mueller) ?      yes

person(fritz,mueller) = person(mueller,fritz) ?      no

5 = 2 ?                                               no

5 = 2 + 3 ?                                           no

2 + 3 = +(2, 3) ?                                     yes
```

⇒ Equality of terms means structural equality.

Terms are not "evaluated" before a comparison!

- Checking equality of terms with variables:

```
person(fritz, Lastname, datum(27, 11, 2007))
    = person(fritz, mueller, datum(27, MM, 2007)) ?
```

- For a variable, any term may be substituted:

    - in particular **mueller** for **Lastname** and **11** for **MM**.

    - <u>After</u> this substitution both terms are equal.

Which variables have to be substituted how, in order to make the terms equal?

```
          date(1, 4, 1985) = date(1, 4, Year) ?

     date(Day, Month, 1985) = date(1, 4, Year) ?

  a(b,C,d(e,F,g(h,i,J))) = a(B,c,d(E,f,g(H,i,K))) ?

                          X = Y + 1 ?

          [[the, Y]|Z] = [[X, dog], [is, here]] ?
```

As a reminder, list syntax:

```
[1,2,a] = [1|[2,a]] = [1,2|[a]] = [1,2|.(a,[])] = .(1,.(2,.(a,[])))
```

And what about:

```
p(X) = p(q(X)) ?
```

"occurs check" (implementation detail)

Some further (problematic) cases:

```
        loves(vincent, X) = loves(X, mia) ?

    loves(marcellus, mia) = loves(X, X) ?

 a(b,C,d(e,F,g(h,i,J))) = a(B,c,d(E,f,p(H,i,K))) ?

                  p(b,b) = p(X) ?

                         …
```

<u>Substitution</u>:

- Replacing variables by other variables or other kinds of terms (constants, structures, …)

- A function which uniquely maps each term to a new term, where the new term differs from the old term only by replacement of variables.

- <u>Notation</u>: $U = \{$`Lastname / mueller, MM / 11`$\}$

- The substitution U changes only the variables `Lastname` and `MM`, everything else stays unchanged!

- $U($`person(fritz, Lastname, datum(27, 11 2007))`$)$
      == `person(fritz, mueller, datum(27, 11, 2007))`

Unification, formally (2)

- <u>Unifier</u>**:**

    - substitution that makes two terms equal

    - e.g., application of the substitution $U = \{$ **Lastname**/**mueller, MM**/**11** $\}$:

    $$U(\textbf{person(fritz,Lastname,date(27,11 2007)))}$$
    $$== U(\textbf{person(fritz,mueller,date(27,MM,2007)))}$$

- <u>Most general unifier</u>:

    - unifier that leaves as many variables as possible unchanged,
      and does not introduce specific terms where variables suffice

    - <u>Example</u>: **date(DD,MM,2007)** and **date(D,11,Y)**

        - $U_1 = \{$ **DD**/**27, D**/**27, MM**/**11, Y**/**2007** $\}$ ✘

        - $U_2 = \{$ **DD**/**D, MM**/**11, Y**/**2007** $\}$ ✔

- Prolog always looks for a most general unifier.

Unification, formally

I will now skip over some slides with a description of a concrete algorithm for computing most general unifiers.

The main reason is that the lecture "Logik" has already introduced an algorithm for this purpose.

And for our consideration of the operational semantics of Prolog you do not need to learn a specific unification algorithm by heart, you only need to be able to determine what the most general unifier for a pair of terms **is**, not how exactly it is obtained. (We will encounter various examples.)

Aside: The issue of the "occurs check" will not come up in any examples considered in lecture, exercises or exam.

Unification, formally (3) – Computing a most general unifier

Input: two terms T_1 and T_2 (in general possibly containing common variables)

Output: a most general unifier U for T_1 and T_2 in case T_1 and T_2 are unifiable, otherwise failure

Algorithm:

1. If T_1 and T_2 are the same constant or variable,
    then $U = \varnothing$

2. If T_1 is a variable that does not occur in T_2,
    then $U = \{T_1 / T_2\}$

    "occurs check"

3. If T_2 is a variable that does not occur in T_1,
    then $U = \{T_2 / T_1\}$

Algorithm (cont.):

4. If $T_1 = f(T_{1,1},...,T_{1,n})$ and $T_2 = f(T_{2,1},...,T_{2,n})$ are structures with the same functor and the same number of components, then

    1. Find a most general unifier U_1 for $T_{1,1}$ and $T_{2,1}$
    2. Find a most general unifier U_2 for $U_1(T_{1,2})$ and $U_1(T_{2,2})$

        …

    n. Find a most general unifier U_n for

        $$U_{n-1}(...(U_1(T_{1,n})...) \text{ and } U_{n-1}(...(U_1(T_{2,n}))...)$$

    If all these unifiers exist, then

    $$U = U_n \circ U_{n-1} \circ ... \circ U_1 \qquad \text{(function composition of the unifiers)}$$

5. Otherwise: T_1 and T_2 are not unifiable.

$$\texttt{date(1, 4, 1985) = date(1, 4, Year)} \; ?$$

Structures with the same functor, same number of components, hence:

1. Find a most general unifier U_1 for **1** and **1**

   $\Rightarrow$ same constants, thus $U_1 = \varnothing$

2. Find a most general unifier U_2 for $U_1(\textbf{4})$ and $U_1(\textbf{4})$

   $\Rightarrow$ same constants, thus $U_2 = \varnothing$

3. Find a most general unifier U_3 for $U_2(U_1(\textbf{1985}))$ and $U_2(U_1(\textbf{Year}))$

   $\Rightarrow$ constant vs. variable, thus $U_3 = \{\textbf{Year}\,/\textbf{1985}\}$

A most general unifier overall is:

$$U = U_3 \circ U_2 \circ U_1 = \{\textbf{Year}\,/\textbf{1985}\}$$

$$\texttt{loves(marcellus, mia) = loves(X, X)} \; ?$$

Structures with the same functor, same number of components, hence:

1. Find a most general unifier U_1 for **marcellus** and **X**

   $\Rightarrow$ constant vs. variable, thus $U_1 = \{\texttt{X}\,/\texttt{marcellus}\}$

2. Find a most general unifier U_2 for $U_1(\texttt{mia}) = \texttt{mia}$ and $U_1(\texttt{X}) = \texttt{marcellus}$

   $\Rightarrow$ different constants, hence U_2 does not exist!

Consequently, also no unifier exists for the original terms!

$$d(E,g(H,J)) = d(F,g(H,E)) \ ?$$

Structures with the same functor, same number of components, hence:

    1.   Find a most general unifier U_1 for E and F

        $\Rightarrow$ different variables, thus $U_1 = \{E/F\}$

    2.   Find a most general unifier U_2 for $U_1(g(H,J))$ and $U_1(g(H,E))$

$$g(H,J) = g(H,F) \ ?$$

        $\Rightarrow$ Structures with the same functor, same number of components, hence:

            - Find a most general unifier $U_{2,1}$ for H and H

                $\Rightarrow$ same variables, thus $U_{2,1} = \varnothing$

            - Find a most general unifier $U_{2,2}$ for $U_{2,1}(J)$ and $U_{2,1}(F)$

                $\Rightarrow$ different variables, thus $U_{2,2} = \{F/J\}$

    $U_2 = U_{2,2} \circ U_{2,1} = \{F/J\}$

A most general unifier overall is:

    $U = U_2 \circ U_1 = \{E/J \ , \ F/J\}$

As a reminder:

2. If T_1 is a variable that does not occur in T_2,
    then $U = \{T_1 \, / \, T_2\}$

    "occurs check"

3. If T_2 is a variable that does not occur in T_1,
    then $U = \{T_2 \, / \, T_1\}$

So, for example:

$$\texttt{X = q(X)} \text{ ?}$$

$\Rightarrow$ No unifier exists.

But in Prolog this check is actually not performed by default!

Without "occurs check":

$$p(X) = p(q(X))?$$

Structures with the same functor, same number of components, hence:

    1.   Find a most general unifier U_1 for X and $q(X)$

        $\Rightarrow$ variable vs. term, thus $U_1 = \{X/q(X)\}$

    $U = U_1 = \{X/q(X)\}$!

Although it actually is <u>not</u> true that $U(p(X))$ and $U(p(q(X)))$ are equal!

Programming Paradigms

Resolution

Resolution (proof principle) – without variables

One can reduce proving the query

`?- P, L, Q.`   (let `L` be a variable free literal and `P` and `Q` be sequences of such)

to proving the query

`?- P, L₁, L₂, ... , Lₙ, Q.`

provided that `L :- L₁, L₂, ..., Lₙ.` is a clause in the program (where n ≥ 0).

- The choice of the literal `L` and the clause to use are in principle arbitrary.

- If n = 0, then the query becomes smaller by the resolution step.

Resolution – with variables

One can reduce proving the query

   `?- P, L, Q.`   (let **L** be a literal and **P** and **Q** be sequences of literals)

to proving the query

   `?- U(P), U(L₁), U(L₂), ... , U(Lₙ), U(Q).`

provided that:

> - there is a program clause `L₀:- L₁, L₂, ..., Lₙ.` (where $n \geq 0$),
>   with – just in case – renamed variables (so that there is no overlap with
>   those in **P, L, Q**),

> - and U is a most general unifier for **L** and **L₀**.

Programming Paradigms

Derivation trees

We wanted to understand why, for example, for

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- mult(X,Y,U),add(U,Y,Z).
```

various kinds of queries/"call modes" work very well:

```
?- mult(s(s(0)),s(s(s(0))),N).
N = s(s(s(s(s(s(0)))))).

?- mult(s(s(0)),N,s(s(s(s(0))))).
N = s(s(0)) ;
false.
```

but others don't:

```
?- mult(N,M,s(s(s(s(0))))).
N = s(0),
M = s(s(s(s(0)))) ;
N = s(s(0)),
M = s(s(0)) ;
abort
```
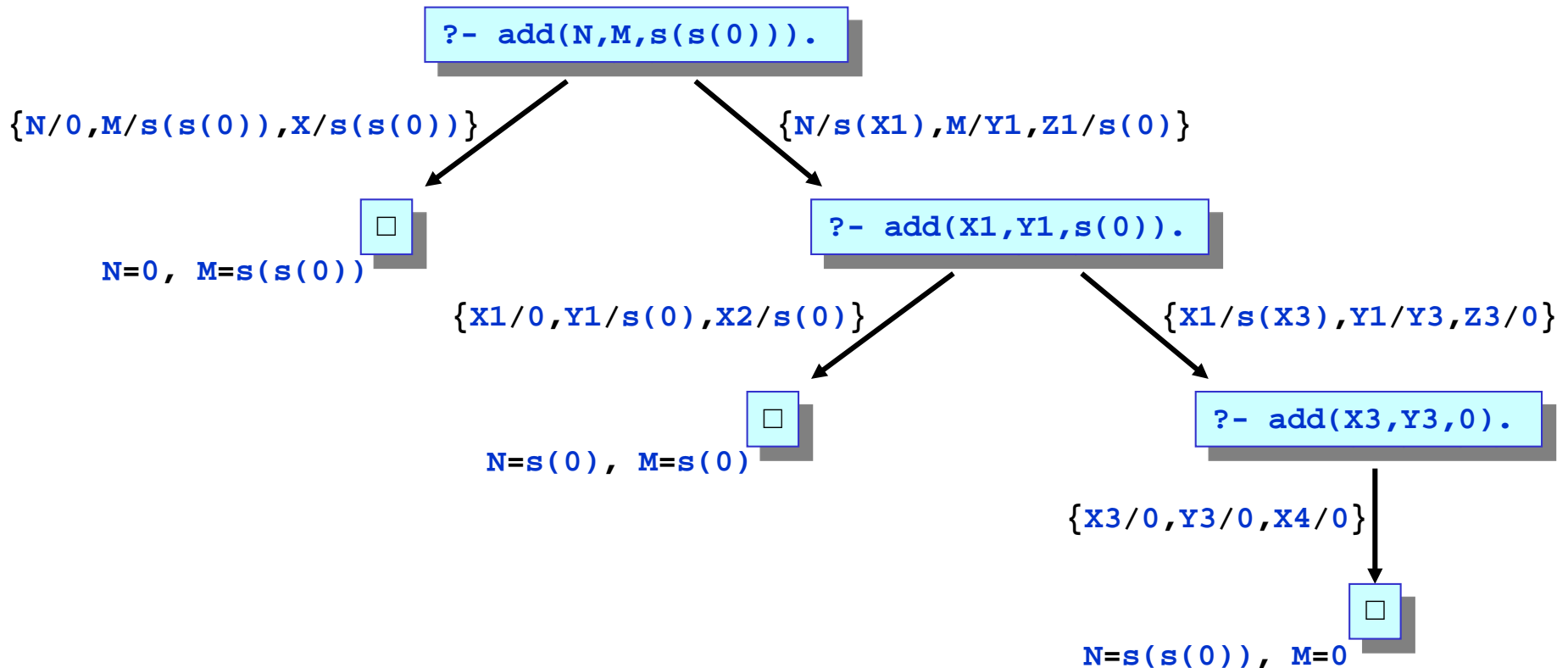
otherwise infinite search

Explicit enumeration of solutions

Let us start with a simple example just for addition:

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

Exhaustive search:

```
?- add(N,M,s(s(0))).
```

{N/0,M/s(s(0)),X/s(s(0))}   {N/s(X1),M/Y1,Z1/s(0)}

□

N=0, M=s(s(0))

```
?- add(X1,Y1,s(0)).
```

{X1/0,Y1/s(0),X2/s(0)}   {X1/s(X3),Y1/Y3,Z3/0}

□

N=s(0), M=s(0)

```
?- add(X3,Y3,0).
```

{X3/0,Y3/0,X4/0}

□

N=s(s(0)), M=0

Detailed description of the generation of derivation trees

1. Generate root node with query, remember it as still to be worked on.

2. As long as there are still nodes to be worked on:
   - select left-most such node
   - determine all facts/rules (with consistently renamed variables) whose head is unifiable with the left-most literal in that node
   - generate for each such fact/rule a (still to be worked on) successor node via a resolution step
   - arrange successor nodes from left to right according to the order of appearance of the used facts/rules in the program (from top to bottom)
   - annotate the unifier used in each case
   - mark nodes as finished if they are empty or if their left-most literal is not unifiable with any fact/rule head
   - at successful nodes, annotate the solution (the composition of unifiers along the path from the root, applied to all variables that occurred in the original query)

An example with infinite search

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- mult(X,Y,U),add(U,Y,Z).
```

```
?- mult(N,M,s(0)).
```

{N/s(X),M/Y,Z/s(0)}

Grows ever longer!

```
?- mult(X,Y,U),add(U,Y,s(0)).
```

{X/0,Y/_1,U/0}   {X/s(X2),Y/Y2,U/Z2}

```
?- add(0,_1,s(0)).
```
```
?- mult(X2,Y2,U2),add(U2,Y2,Z2),add(Z2,Y2,s(0)).
```

{_1/s(0),X1/s(0)}   {X2/0,Y2/_2,U2/0}   {X2/s(X3),Y2/Y3,U2/Z3}

□

N=s(0), M=s(0)
```
?- add(0,_2,Z2),add(Z2,_2,s(0)).
```
```
?- ….
```

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- mult(X,Y,U),add(U,Y,Z).
```

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- add(U,Y,Z),mult(X,Y,U).
```

```
?- mult(N,M,s(0)).
```

`{N/s(X),M/Y,Z/s(0)}`

```
?- add(U,Y,s(0)),mult(X,Y,U).
```

`{U/0,Y/s(0),X1/s(0)}`

```
?- mult(X,s(0),0).
```

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- add(U,Y,Z),mult(X,Y,U).
```

?- mult(N,M,s(0)).

{N/s(X),M/Y,Z/s(0)}

?- add(U,Y,s(0)),mult(X,Y,U).

{U/0,Y/s(0),X1/s(0)}   {U/s(X3),Y/Y3,Z3/0}

?- mult(X,s(0),0).   **?- add(X3,Y3,0),mult(X,Y3,s(X3)).**

{X/0,_1/s(0)}   {X/s(X2),Y2/s(0),Z2/0}   {X3/0,Y3/0,X4/0}

□   **?- add(U2,s(0),0),mult(X2,s(0),U2).**   **?- mult(X,0,s(0)).**

N=s(0),
M=s(0)   {X/s(X5),Y5/0,Z5/s(0)}

?- add(U5,0,s(0)),mult(X5,0,U5).

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- add(U,Y,Z),mult(X,Y,U).
```

```
?- add(X3,Y3,0),mult(X,Y3,s(X3)).
```

| {X3/0,Y3/0,X4/0}

```
?- mult(X,0,s(0)).
```

| {X/s(X5),Y5/0,Z5/s(0)}

```
?- add(U5,0,s(0)),mult(X5,0,U5).
```

| {U5/s(X6),Y6/0,Z6/0}   Does not look good!

```
?- add(X6,0,0),mult(X5,0,s(X6)).
```

| {X6/0,X7/0}

```
?- mult(X5,0,s(0)).
```

Input: query and program,
           for example
           `mult(N,M,s(0))` and:

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- add(U,Y,Z),mult(X,Y,U).
```

Output: tree, generated by following steps:

1. Generate root node with query, remember it as still to be worked on.

2. As long as there are still nodes to be worked on:
   - select left-most such node
   - determine all facts/rules (with consistently renamed variables) whose head is unifiable with the left-most literal in that node
   - generate for each such fact/rule a (still to be worked on) successor node via a resolution step
   - arrange successor nodes from left to right according to the order of appearance of the used facts/rules in the program (from top to bottom)
   - annotate the unifier used in each case

`?- mult(N,M,s(0)).`

$\{$`N/s(X),M/Y,Z/s(0)`$\}$
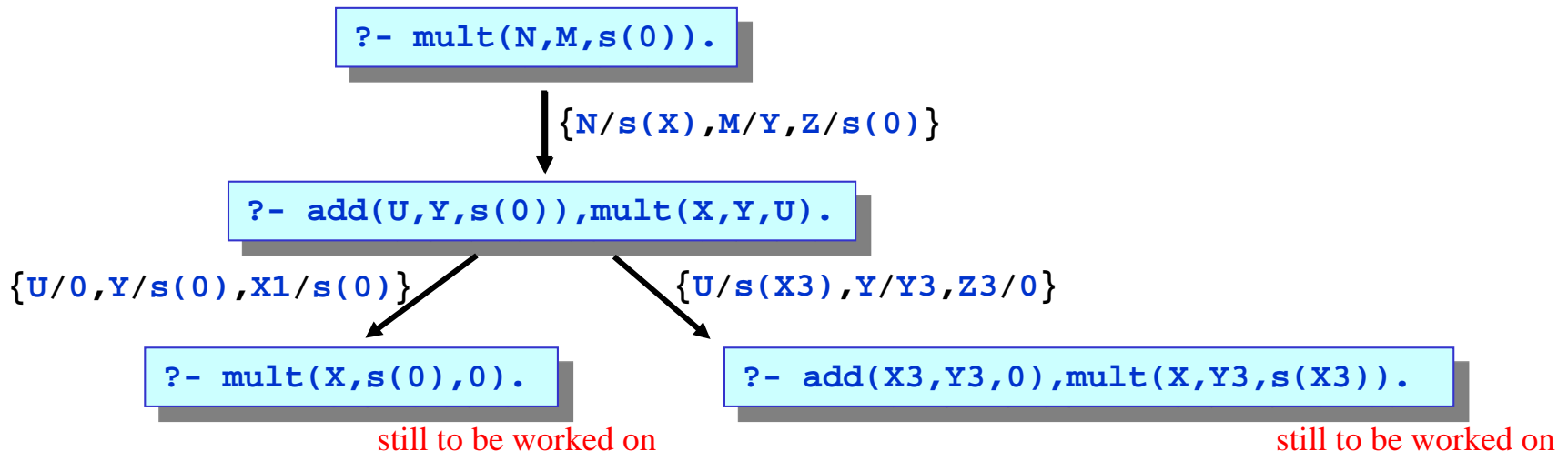
`?- add(U,Y,s(0)),mult(X,Y,U).`

still to be worked on

Detailed description of the generation of derivation trees

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- add(U,Y,Z),mult(X,Y,U).
```

2. As long as there are still nodes
    to be worked on:
    - select left-most such node
    - determine all facts/rules (w. cons.
      renamed variables) whose head is unifiable with the left-most literal in that node
    - generate for each such fact/rule a (still to be worked on)
      successor node via a resolution step
    - arrange successor nodes from left to right according to the order
      of appearance of the used facts/rules in the program (from top to bottom)
   - annotate the unifier used in each case

```
?- mult(N,M,s(0)).
```
{N/s(X),M/Y,Z/s(0)}

```
?- add(U,Y,s(0)),mult(X,Y,U).
```
{U/0,Y/s(0),X1/s(0)}   {U/s(X3),Y/Y3,Z3/0}

```
?- mult(X,s(0),0).
```
still to be worked on

```
?- add(X3,Y3,0),mult(X,Y3,s(X3)).
```
still to be worked on

Detailed description of the generation of derivation trees

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- add(U,Y,Z),mult(X,Y,U).
```
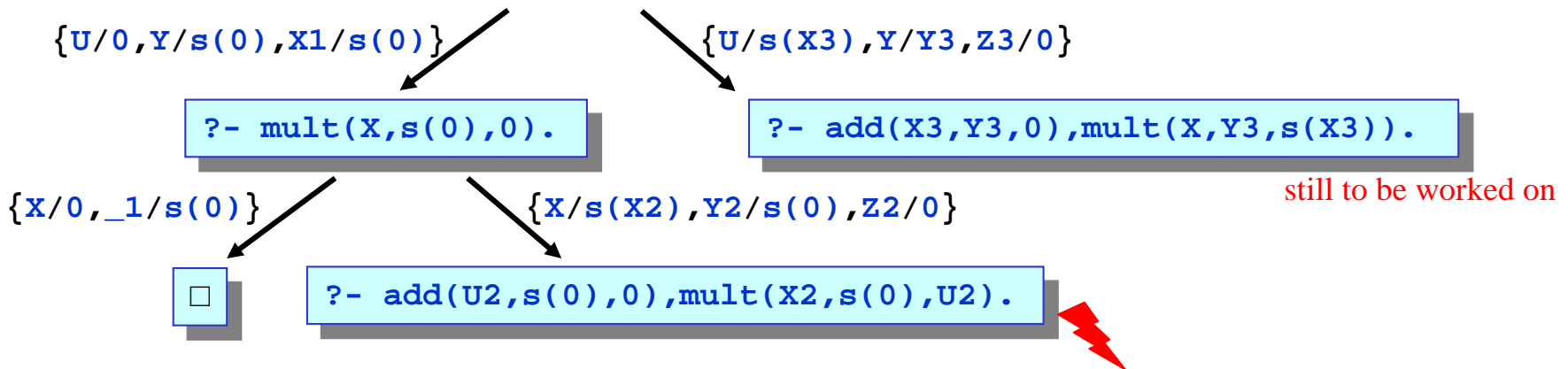
2. As long as there are still nodes to be worked on:
   - select left-most such node
   - determine all facts/rules (w. cons. renamed variables) whose head is unifiable with the left-most literal in that node
   - generate for each such fact/rule a (still to be worked on) successor node via a resolution step
   - arrange successor nodes from left to right according to the order of appearance of the used facts/rules in the program (from top to bottom)
   - annotate the unifier used in each case
   - mark nodes as finished if they are empty (□) or if their left-most literal is not unifiable with any fact/rule head (⚡)

{U/0,Y/s(0),X1/s(0)}   {U/s(X3),Y/Y3,Z3/0}

?- mult(X,s(0),0).   ?- add(X3,Y3,0),mult(X,Y3,s(X3)).

{X/0,_1/s(0)}   {X/s(X2),Y2/s(0),Z2/0}   still to be worked on

□   ?- add(U2,s(0),0),mult(X2,s(0),U2).

2. As long as there are still nodes
    to be worked on:

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- add(U,Y,Z),mult(X,Y,U).
```

- select left-most such node
- determine all facts/rules (w. cons.
  renamed variables) whose head is unifiable with the left-most literal in that node
- generate for each such fact/rule a (still to be worked on)
  successor node via a resolution step
- arrange successor nodes from left to right according to the order
  of appearance of the used facts/rules in the program (from top to bottom)
- annotate the unifier used in each case
- mark nodes as finished if they are empty
  or if their left-most literal is not unifiable with any fact/rule head
- at successful nodes, annotate the solution (the composition of unifiers along the
  path from the root, applied to all variables that occurred in the original query)

`?- mult(X,s(0),0).`   `?- add(X3,Y3,0),mult(X,Y3,s(X3)).`

$\{X/0,\_1/s(0)\}$ $\{X/s(X2),Y2/s(0),Z2/0\}$ still to be worked on

`N=s(0),`   □   `?- add(U2,s(0),0),mult(X2,s(0),U2).`
`M=s(0)`

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- add(U,Y,Z),mult(X,Y,U).
```

```
?- add(X3,Y3,0),mult(X,Y3,s(X3)).
```

$\{$X3/0,Y3/0,X4/0$\}$

```
?- mult(X,0,s(0)).
```

$\{$X/s(X5),Y5/0,Z5/s(0)$\}$

```
?- add(U5,0,s(0)),mult(X5,0,U5).
```

$\{$U5/s(X6),Y6/0,Z6/0$\}$

Does not look good!

```
?- add(X6,0,0),mult(X5,0,s(X6)).
```

$\{$X6/0,X7/0$\}$

```
?- mult(X5,0,s(0)).
```

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- add(U,Y,Z),Y\=0,mult(X,Y,U).
```

```
?- mult(N,M,s(0)).
```

{N/s(X),M/Y,Z/s(0)}

```
?- add(U,Y,s(0)),Y\=0,mult(X,Y,U).
```

{U/0,Y/s(0),X1/s(0)}   {U/s(X3),Y/Y3,Z3/0}

```
?- s(0)\=0,mult(X,s(0),0).
```
                              ```
                              ?- add(X3,Y3,0),Y3\=0,mult(X,Y3,s(X3)).
                              ```
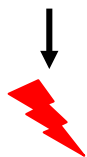
{X/0,_1/s(0)}   {X/s(X2),Y2/s(0),Z2/0}   {X3/0,Y3/0,X4/0}

□

N=s(0),
M=s(0)

```
?- add(U2,s(0),0),s(0)\=0,
     mult(X2,s(0),U2).
```
                              ```
                              ?- 0\=0,mult(X,0,s(0)).
                              ```

Only partial success

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- add(U,Y,Z),Y\=0,mult(X,Y,U).
```

```
?- mult(N,M,s(s(s(s(0))))).
N = s(0),
M = s(s(s(s(0)))) ;
N = s(s(0)),
M = s(s(0)) ;
N = s(s(s(s(0)))),
M = s(0) ;
false.
```

```
?- mult(s(0),0,0).
false.
```

New results found, old results lost!

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(_),0,0).
mult(s(X),Y,Z) :- add(U,Y,Z),Y\=0,mult(X,Y,U).
```

Now this works:

```
?- mult(s(0),0,0).
true.
```

And it even works generally
`mult(?X,?Y,+Z)`.

But unfortunately (only noticed now):

```
?- mult(s(0),s(0),N).
N = s(0) ;
abort
```

otherwise infinite search

So `mult(+X,+Y,?Z)`.
does not anymore work.

A new "infinity trap"

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(_),0,0).
mult(s(X),Y,Z) :- add(U,Y,Z),Y\=0,mult(X,Y,U).
```

```
?- mult(s(0),s(0),N).
```

$\{$`X/0,Y/s(0),N/Z`$\}$

Does not look good!

```
?- add(U,s(0),Z),s(0)\=0,mult(0,s(0),U).
```

$\{$`U/0,X1/s(0),Z/s(0)`$\}$ $\{$`U/s(X2),Y2/s(0),Z/s(Z2)`$\}$

```
?- s(0)\=0,mult(0,s(0),0).
```

```
?- add(X2,s(0),Z2),s(0)\=0,mult(0,s(0),s(X2)).
```

$\{$`_1/s(0)`$\}$

□

`N=s(0)`

important observation:
(see last lecture)

```
?- add(U,s(0),Z).
U = 0, Z = s(0) ;
U = s(0), Z = s(s(0)) ;
...
```

vs.

```
?- add(s(0),U,Z).
Z = s(U).
```

Exploiting commutativity

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(_),0,0).
mult(s(X),Y,Z) :- add(Y,U,Z),Y\=0,mult(X,Y,U).
```

important observation:
(see last lecture)

```
?- add(U,s(0),Z).
U = 0, Z = s(0) ;
U = s(0), Z = s(s(0)) ;
...
```

vs.

```
?- add(s(0),U,Z).
Z = s(U).
```

Exploiting commutativity

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(_),0,0).
mult(s(X),Y,Z) :- add(Y,U,Z),Y\=0,mult(X,Y,U).
```

```
?- mult(s(0),s(0),N).
```

{X/0,Y/s(0),N/Z}

```
?- add(s(0),U,Z),s(0)\=0,mult(0,s(0),U).
```

{X1/0,U/Y1,Z/s(Z1)}

```
?- add(0,Y1,Z1),s(0)\=0,mult(0,s(0),Y1).
```

{Y1/X2,Z1/X2}

```
?- s(0)\=0,mult(0,s(0),X2).
```

{_1/s(0),X2/0}

□   N=s(0)

Indeed a generally useful definition

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(_),0,0).
mult(s(X),Y,Z) :- add(Y,U,Z),Y\=0,mult(X,Y,U).
```

```
?- mult(N,M,s(s(s(s(0))))).
N = s(0),
M = s(s(s(s(0)))) ;
N = s(s(0)),
M = s(s(0)) ;
N = s(s(s(s(0)))),
M = s(0) ;
false.

?- mult(s(0),s(0),N).
N = s(0).

?- add(X,0,X),not(mult(s(s(_)),s(s(_)),X)).
...
```

Now <u>all</u> call modes
work well, <u>except</u>
`mult(?X,?Y,?Z)`!

Conclusion

The operational semantics:

- reflects the actual Prolog search process, with backtracking

- makes essential use of unification and resolution steps

- enables understanding of effects like non-termination

- gives insight into impact of changes to the order of, and within, facts and rules