Programming Paradigms – Prolog part

Summer Term

Prof. Janis Voigtländer
University of Duisburg-Essen

Programming Paradigms

Prolog Basics

Prolog in simplest case: facts and queries

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

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

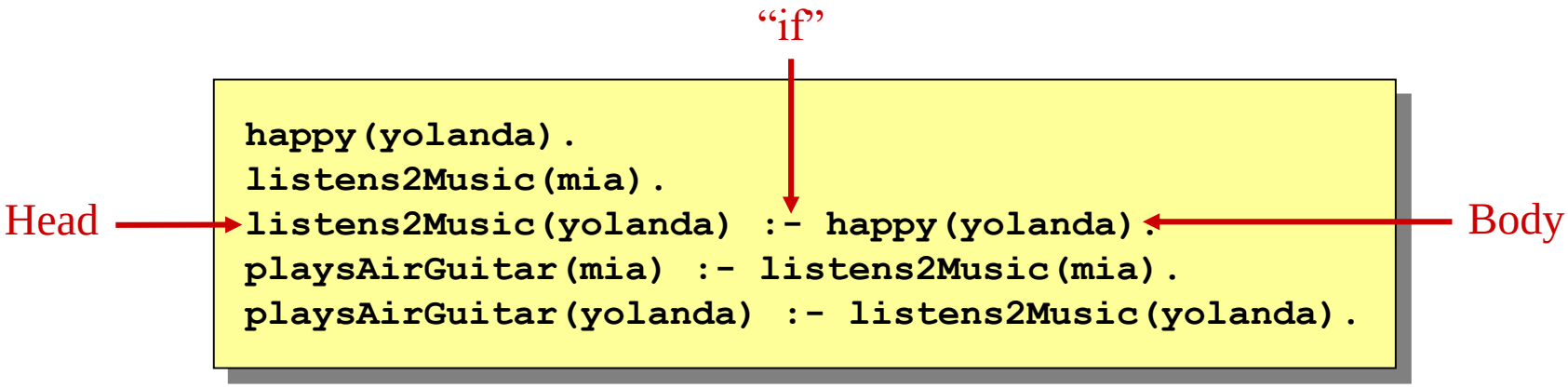
- Queries:

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

?- playsAirGuitar(jody).
true.

?- playsAirGuitar(mia).
false.

?- playsAirGuitar(vincent).
false.

?- playsPiano(jody).
false.
```

The dot is essential!

or an error message

Facts + simple implications



- 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.
```

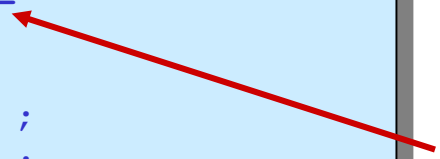
semicolon entered by user

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.
```



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), X \= Y.
```

important that at end

- Queries:

```
?- 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

```
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.

```
?- 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`**:
  if for every choice of **`Z`** holds: **`loves(X,Z)`**, and **`loves(Y,Z)`**, and **`X \= Y`**,
  then also holds: **`jealous(X,Y)`**

- Or, for arbitrary (but fixed) **`X`** , **`Y`**:
  for every choice of **`Z`** holds: if **`loves(X,Z)`**, and **`loves(Y,Z)`**, and **`X \= Y`**,
  then 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?)

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

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

- Logically equivalent, for arbitrary (but fixed) **`X`** , **`Y`**:
  if for any choice of **`Z`** holds: **`loves(X,Z)`** , and **`loves(Y,Z)`** , and **`X \= Y`**,
  then 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
```

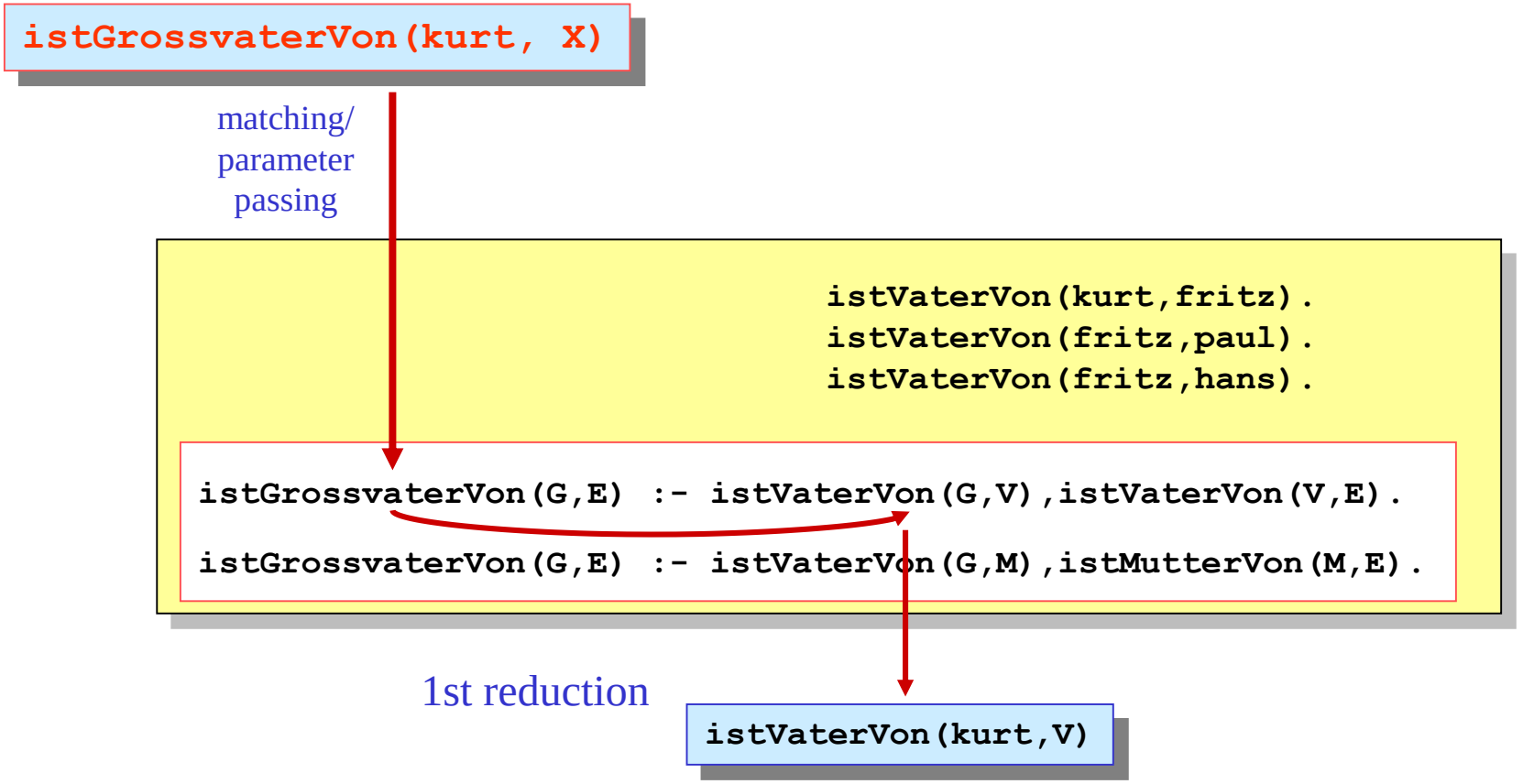
Input: a query

(repeated) resolution

Output: variable substitution(s)

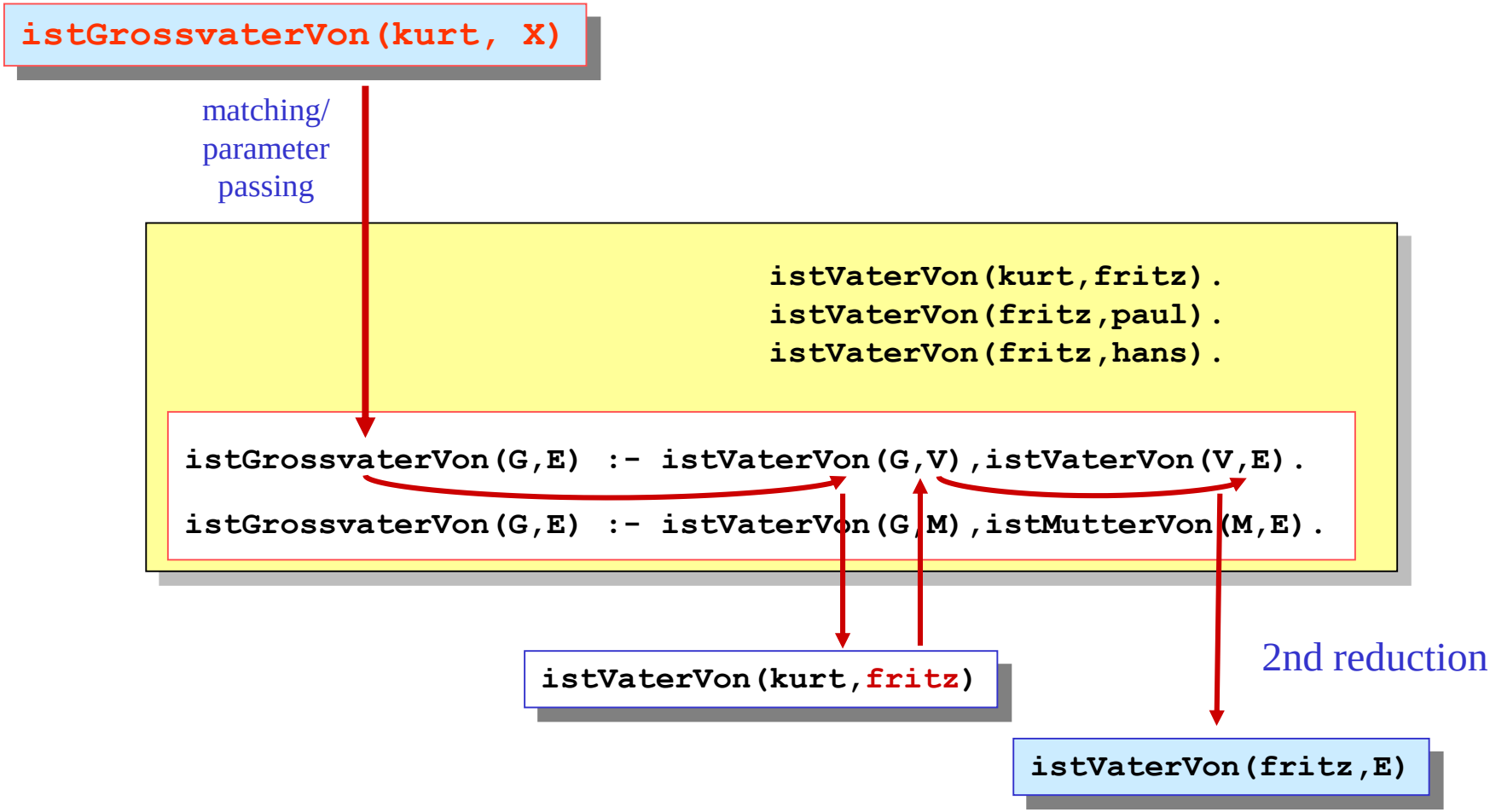
Operationalisation in Prolog (1)

Principle: reduction to subproblems



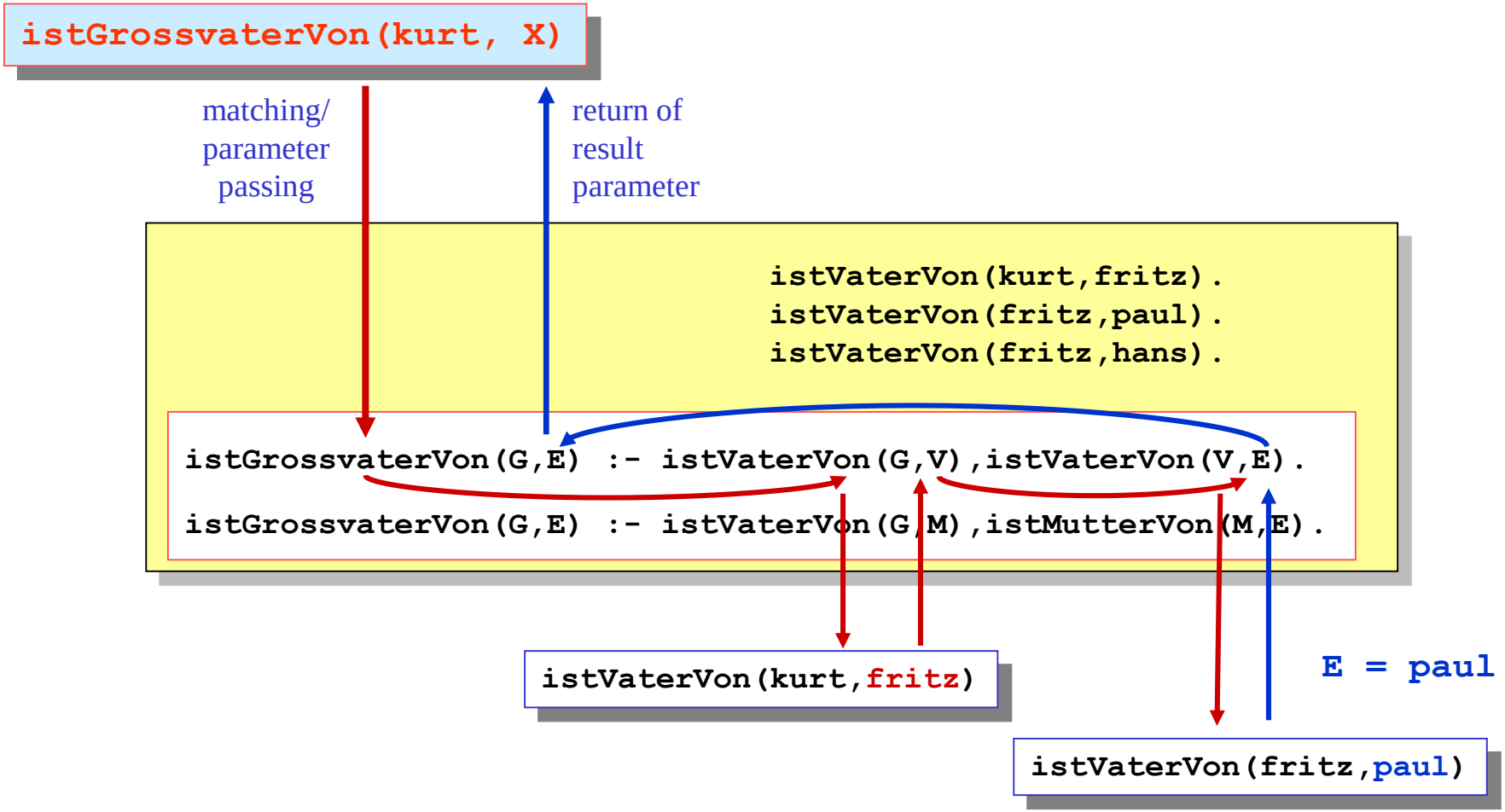
Operationalisation in Prolog (2)

Principle: reduction to subproblems, where new subqueries are found from left to right!



Operationalisation in Prolog (3)

Principle: reduction to subproblems



Operationalisation in Prolog (4)

- Prolog always looks for matching rules or facts from top to bottom 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-try: `istVaterVon(fritz,E)`

position of last solution – that is where search continues

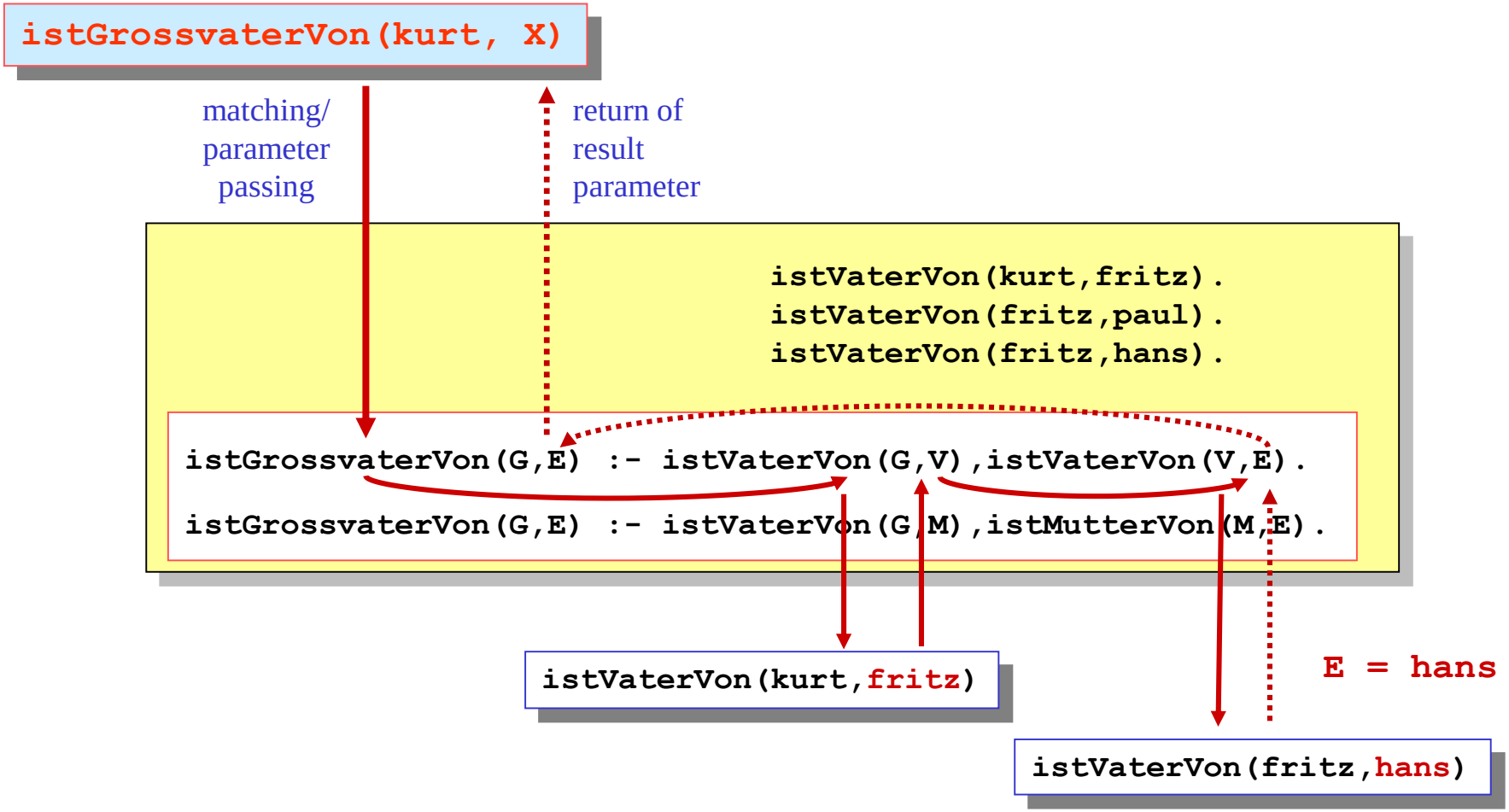
```
istVaterVon(kurt,fritz).
istVaterVon(fritz,paul).
istVaterVon(fritz,hans).
```

solution:

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

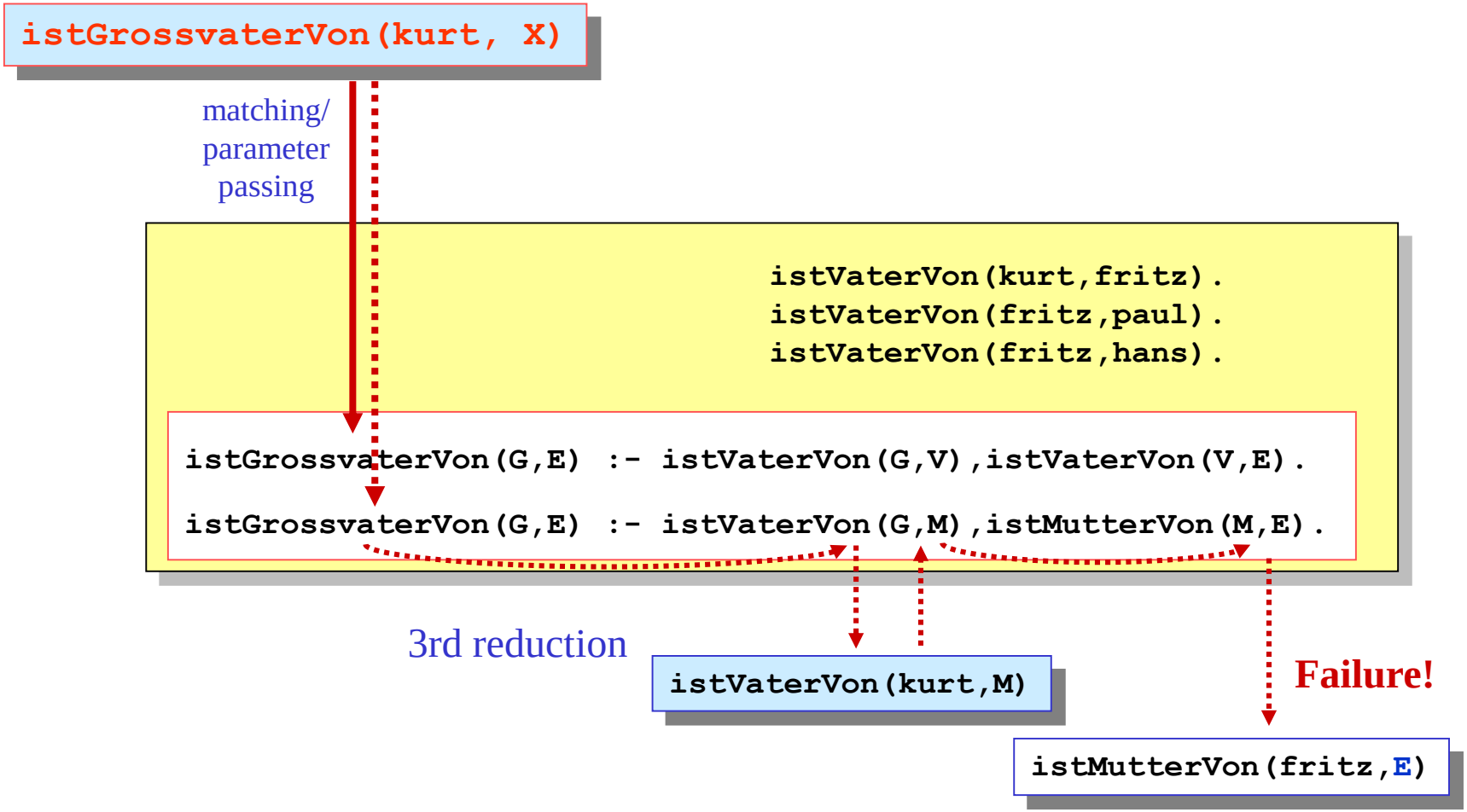
Operationalisation in Prolog (5)

Principle: reduction to subproblems



Operationalisation in Prolog (6)

The backtracking also concerns further matching rules:



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).
```

X = paul:

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

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).
```

X = paul:
X = hans:

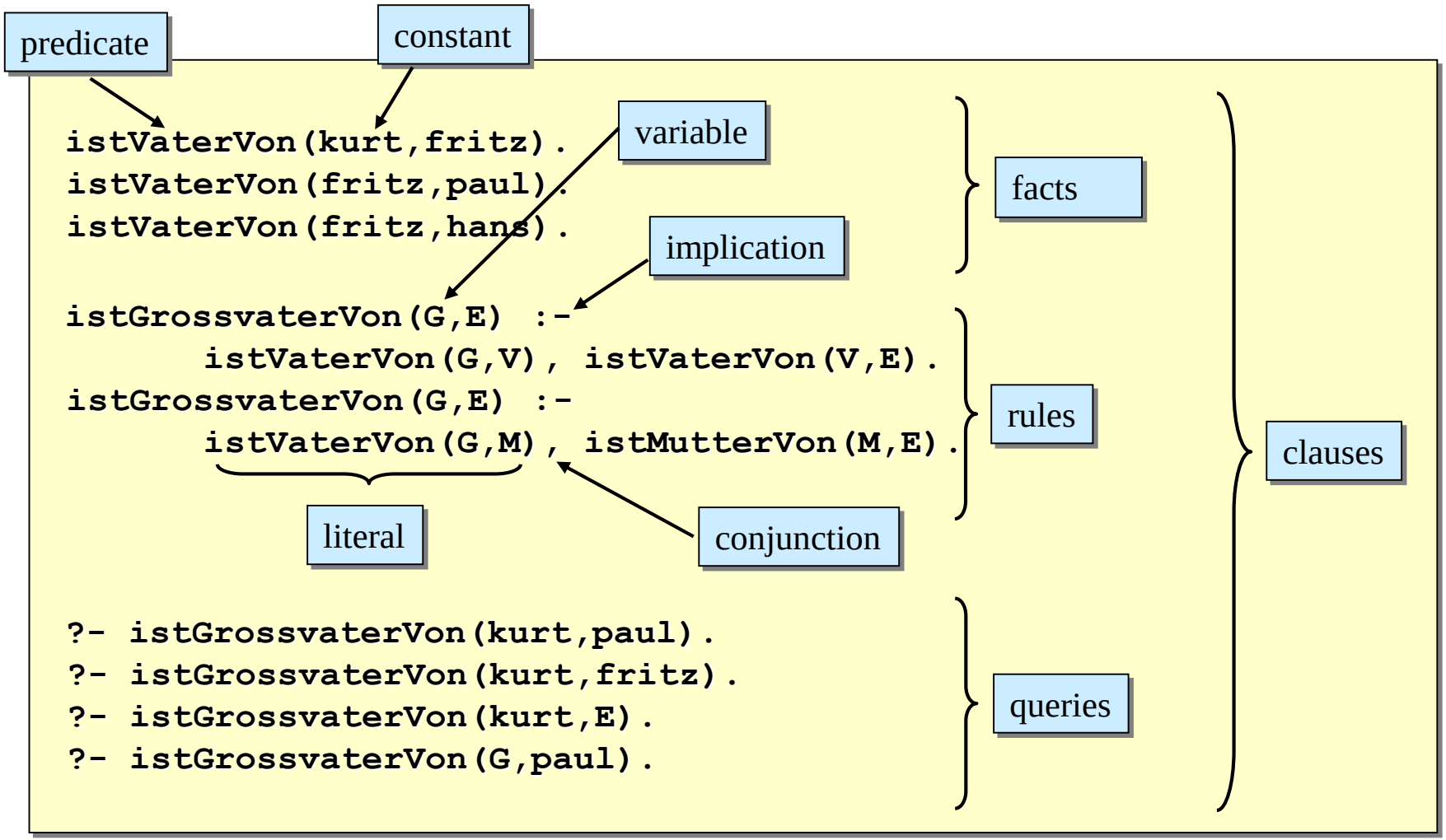
?- istGrossvaterVon(kurt, X).
?- istVaterVon(kurt, V), istVaterVon(V, X).
?- istVaterVon(fritz, X).
?- .
?- .
?- istVaterVon(kurt, M), istMutterVon(M, X).
?- istMutterVon(fritz, X).
Failure!

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

Programming Paradigms

Syntactical ingredients of Prolog

Syntax / notions in Prolog



Syntactical objects in Prolog

- To build clauses, Prolog uses different pieces:
  - constants (numbers, atoms – mainly lowercase identifiers, …)
  - variables (X,Y, ThisThing, _, _G107…)
  - operator terms (… 1 + 3 * 4 …)
  - structures (date(27,11,2007), person(fritz, mueller), …
    composite, recursive, “infinite”, …)

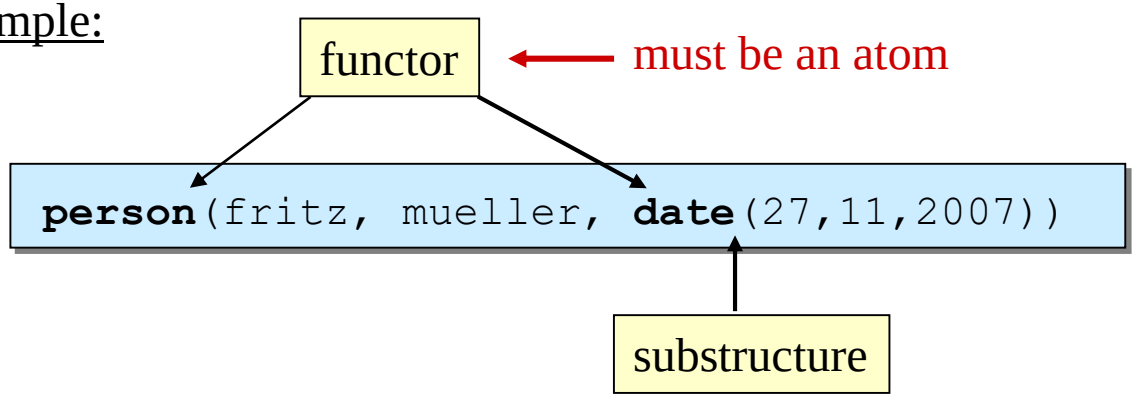
- Note: Prolog has no type system!

Syntactical objects in Prolog

Structures in Prolog

- Structures represent objects that are made up of other objects (like trees and subtrees).

- Example:



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.

Syntactical 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:

"Prolog" = [80, 114, 111, 108, 111, 103]
= .(80, .(114, .(111, .(108, .(111, .(103, [ ])))))

Operators:

- Operators are functors/atoms made from symbols and can be 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))`

Syntactical objects in Prolog

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)
```

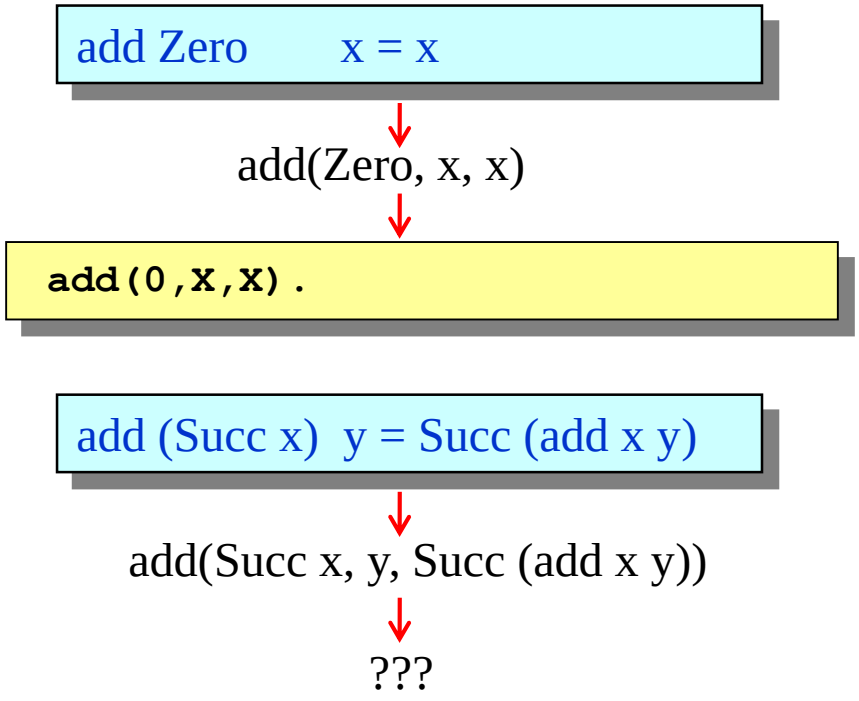
Systematic connection/derivation?

- Essential difference Haskell/Prolog:

  Functions vs. Predicates/Relations

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

- First a somewhat naïve attempt to exploit this correspondence:



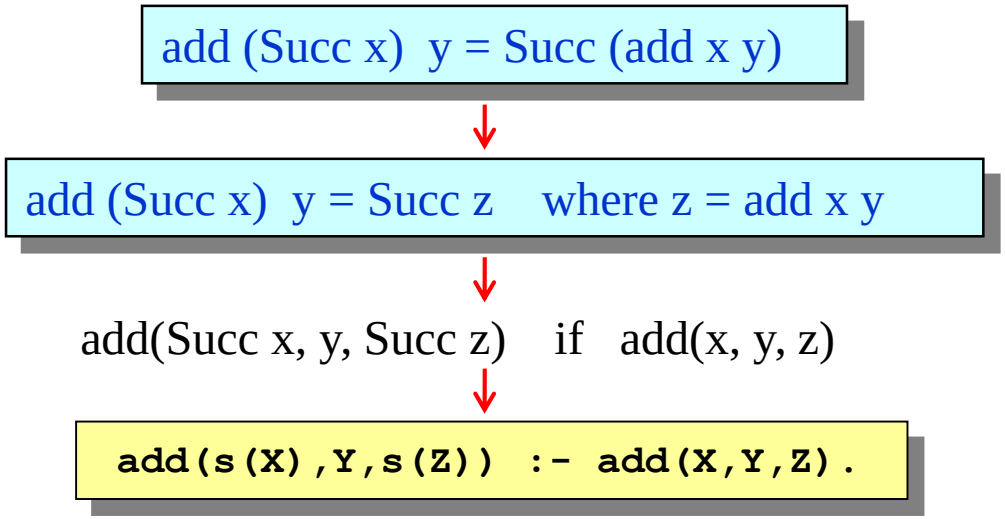
Systematic connection/derivation?

- 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>:



On the flexibility of Prolog predicates

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

```
?- 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

```
add(0,X,X).
add(s(X),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(M) ;
false.
```

Another example

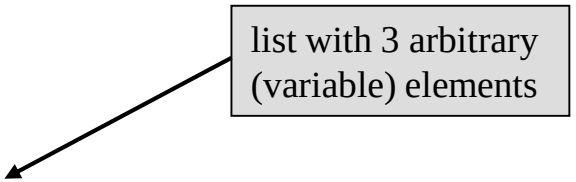
Computing the length of a list in Haskell:

```
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]
```



Arithmetics vs. symbolic operator terms

Caution: 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].
```

An example corresponding to several nested calls

```
partition :: Int → [Int] → ([Int], [Int])
…

quicksort [ ]     =  [ ]
quicksort (h : t) =  quicksort l1 ++ h : quicksort l2
                     where (l1, l2) = partition h t
```

```
quicksort [ ]     =  [ ]
quicksort (h : t) =  ls ++ h : quicksort l2
                     where (l1, l2) = partition h t
                           ls = quicksort l1
```

```
quicksort [ ]     =  [ ]
quicksort (h : t) =  ls ++ h : lg
                     where (l1, l2) = partition h t
                           ls = quicksort l1
                           lg = quicksort l2
```

```
quicksort [ ]     =  [ ]
quicksort (h : t) =  list
                     where (l1, l2) = partition h t
                           ls = quicksort l1
                           lg = quicksort l2
                           list = ls ++ h : lg
```

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

lesson: “inner subexpressions first”

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:

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

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 (these specific interpretations are called models)

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)))`, …} (without predicate symbols!)
the Herbrand base: {`add(0,0,0)`, `add(0,0,s(0))`, `add(0,s(0),0)`, …}
(all applications of predicate symbols on terms from Herbrand universe)

Smallest Herbrand model

Can one compute, in a constructive fashion, the smallest (via the subset relation) Herbrand interpretation that is a 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 $L_1, L_2, \ldots, L_n$ exist in I such that L_0 `:-` $L_1, L_2, \ldots, L_n$ is a complete instantiation (i.e., no variables left) of any of the given program clauses (facts/rules).

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).
```

$T_P(\emptyset)$ = {`add(0,0,0)`, `add(0,s(0),s(0))`, `add(0,s(s(0)),s(s(0)))`, …}

$T_P(T_P(\emptyset))$ = $T_P(\emptyset)$ ∪ {`add(s(0),0,s(0))`, `add(s(0),s(0),s(s(0)))`,
`add(s(0),s(s(0)),s(s(s(0))))`, …}

$T_P(T_P(T_P(\emptyset)))$ = $T_P(T_P(\emptyset))$ ∪ {`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)))))`, …}

…

In the limit: { `add(`s^i`(0),`s^j`(0),`s^{i+j}`(0))` | $i, j \geq 0$ }

Applicability of the semantics based on Herbrand models

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).
```

$T_P(\varnothing)$ = {**`add(0,0,0)`**, **`add(0,s(0),s(0))`**, …} ∪ {**`mult(0,0,0)`**, **`mult(0,s(0),0)`**, …} ∪ {**`mult(s(0),0,0)`**, …}

$T_P(T_P(\varnothing))$ = $T_P(\varnothing)$ ∪ {**`add(s(0),0,s(0))`**, **`add(s(0),s(0),s(s(0)))`**, …} ∪ {**`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).
```

$T_P(\emptyset) = \{\texttt{add(0,0,0)}, \texttt{add(0,s(0),s(0))}, \ldots\} \cup \{\texttt{mult(0,0,0)}, \texttt{mult(0,s(0),0)}, \ldots\} \cup \{\texttt{mult(s(0),0,0)}, \ldots\}$

$T_P(T_P(\emptyset)) = T_P(\emptyset) \cup \{\texttt{add(s(0),0,s(0))}, \texttt{add(s(0),s(0),s(s(0)))}, \ldots\} \cup \{\texttt{mult(s(0),s(0),s(0))}\}$

$T_P(T_P(T_P(\emptyset))) = T_P(T_P(\emptyset)) \cup \{\texttt{add(s(s(0)),0,s(s(0)))}, \ldots\} \cup \{\texttt{mult(s(0),s(s(0)),s(s(0)))}, \texttt{mult(s(s(0)),s(0),s(s(0)))}\}$

$T_P^4(\emptyset) = T_P^3(\emptyset) \cup \{\texttt{add(s}^3\texttt{(0),0,s}^3\texttt{(0))}, \texttt{add(s}^3\texttt{(0),s(0),s}^4\texttt{(0))}, \ldots\} \cup \{\texttt{mult(s(0),s}^3\texttt{(0),s}^3\texttt{(0))}, \texttt{mult(s}^2\texttt{(0),s}^2\texttt{(0),s}^4\texttt{(0))}, \texttt{mult(s}^3\texttt{(0),s(0),s}^3\texttt{(0))}\}$

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

```
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).
```

```
?- 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 concerning 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(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(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?

Moreover, caution needed when using/positioning negative literals

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:

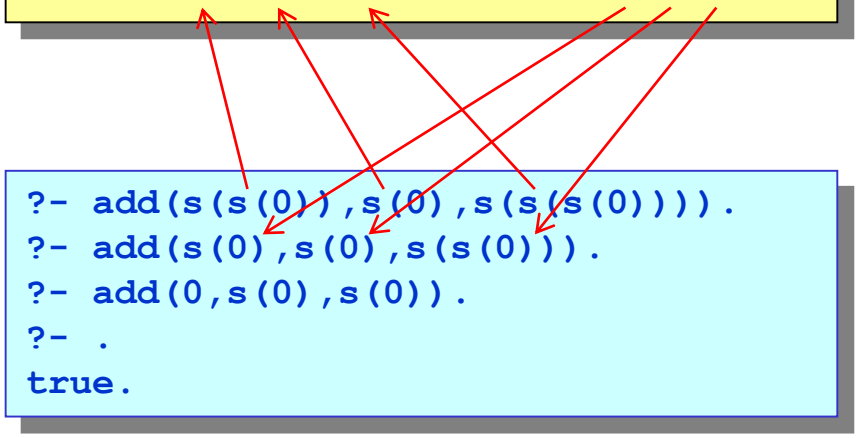
1. Unification
2. Resolution
3. Derivation trees

Programming Paradigms

Unification

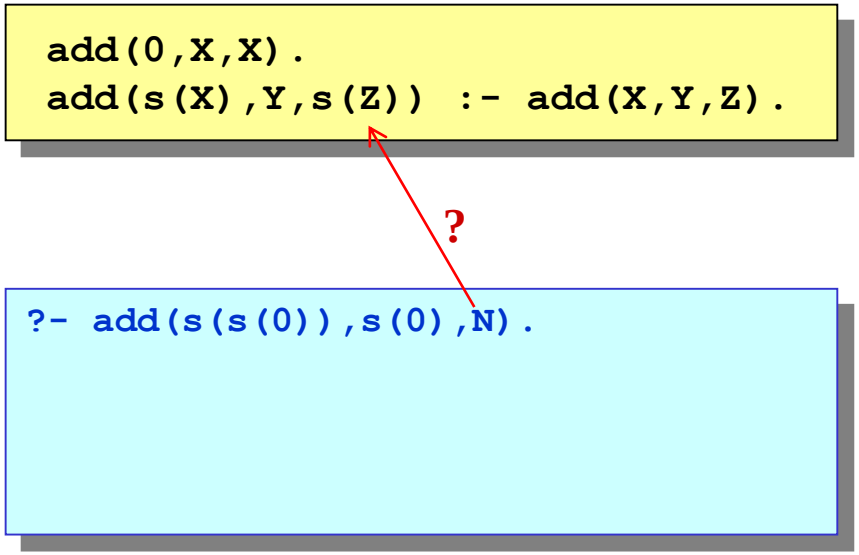
Analogy to Haskell: Pattern matching

```
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”?



In some sense, we need a form of “bidirectional pattern matching”, that can also combine and propagate variable bindings.

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
```

$\Rightarrow$ Equality of terms means structural equality.

Terms are not “evaluated” before a comparison!

Equality of terms (2)

- 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`.
  - After this substitution both terms are equal.

Equality of terms (3)

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)

Equality of terms (4)

Some further (problematic) cases:

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

Unification concepts, somewhat formally (1)

Substitution:

- Replacing variables by other variables or other kinds of terms (constants, structures, …)
- Extended to a function which uniquely maps each term to a new term, where the new term differs from the old term only by the replacement of variables.
- Notation: *U* = {`Lastname / mueller, MM / 11`}
- This substitution *U* changes only the variables `Lastname` and `MM` (in context), everything else stays unchanged.
- *U*(`person(fritz, Lastname, datum(27, 11, 2007))`)
  `==` `person(fritz, mueller, datum(27, 11, 2007))`

Unification concepts, somewhat formally (2)

- Unifier:
  - substitution that makes two terms equal
  - e.g., application of the substitution U = { `Lastname/mueller, MM/11` } :

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

- Most general unifier:
  - unifier that leaves as many variables as possible unchanged, and does not introduce specific terms where variables suffice
  - Example: `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

We 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 it has been practiced in that course’s exercises.

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**.
(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 (though it is relevant in Prolog implementations).

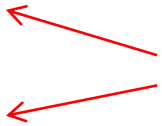
Unification – Computing a most general unifier

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



"occurs check"

Unification – Computing a most general unifier

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

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

   If all these unifiers exist, then

   $U = U_n \circ U_{n-1} \circ ... \circ U_1$ (function composition of the unifiers, always applied recursively along term structure)

5. Otherwise: T_1 and T_2 are not unifiable.

Unification algorithm – Examples

```
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(`4`) and U_1(`4`)

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

3. Find a most general unifier U_3 for $U_2(U_1$(`1985`)) and $U_2(U_1$(`Year`))

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

A most general unifier overall is:

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

Unification algorithm – Examples

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

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

1. Find a most general unifier U_1 for `marsellus` and `X`

   $\Rightarrow$ constant vs. variable, thus U_1 = {`X` /`marsellus`}

2. Find a most general unifier U_2 for U_1(`mia`) = `mia` and U_1(`X`) = `marsellus`

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

Consequently, also no unifier exists for the original terms!

Unification algorithm – Examples

`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:

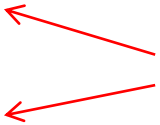
$U = U_2 \circ U_1 = \{$**`E`**/**`J`** , **`F`**/**`J`**$\}$

Relevance of the “occurs check”

As a reminder:

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

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



“occurs check”

So, for example:

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

$\Rightarrow$ No unifier exists.

But in Prolog this check is actually not performed by default (in can be enabled in implementations, though)!

Relevance of the “occurs check”

Without “occurs check”:

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

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 $U_1 = \{\texttt{X}/\texttt{q(X)}\}$

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

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

Programming Paradigms

Resolution

Resolution in Prolog (1)

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 $n \geq 0$).

- 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 L_0.

Programming Paradigms

Derivation trees

Reminder: Motivation for considering operational semantics…

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
```

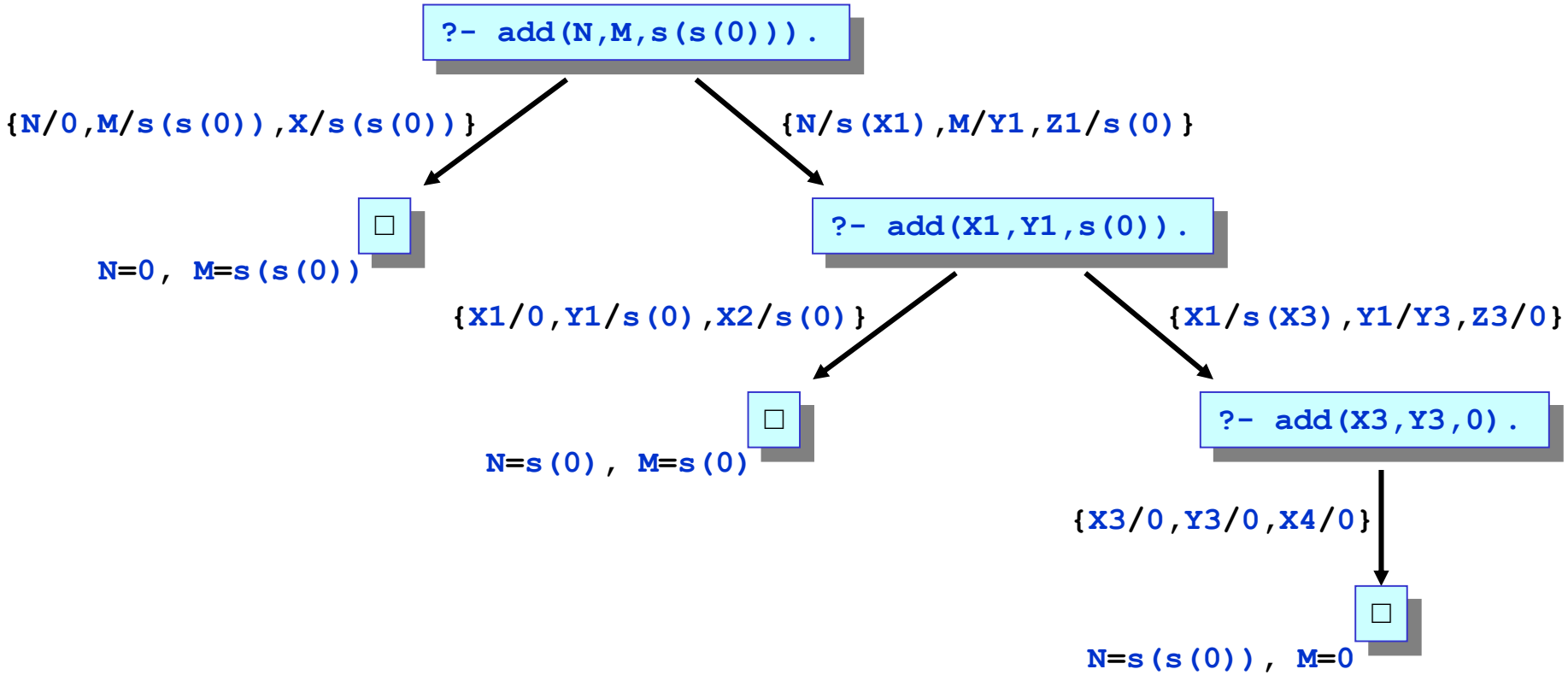
otherwise infinite search

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:



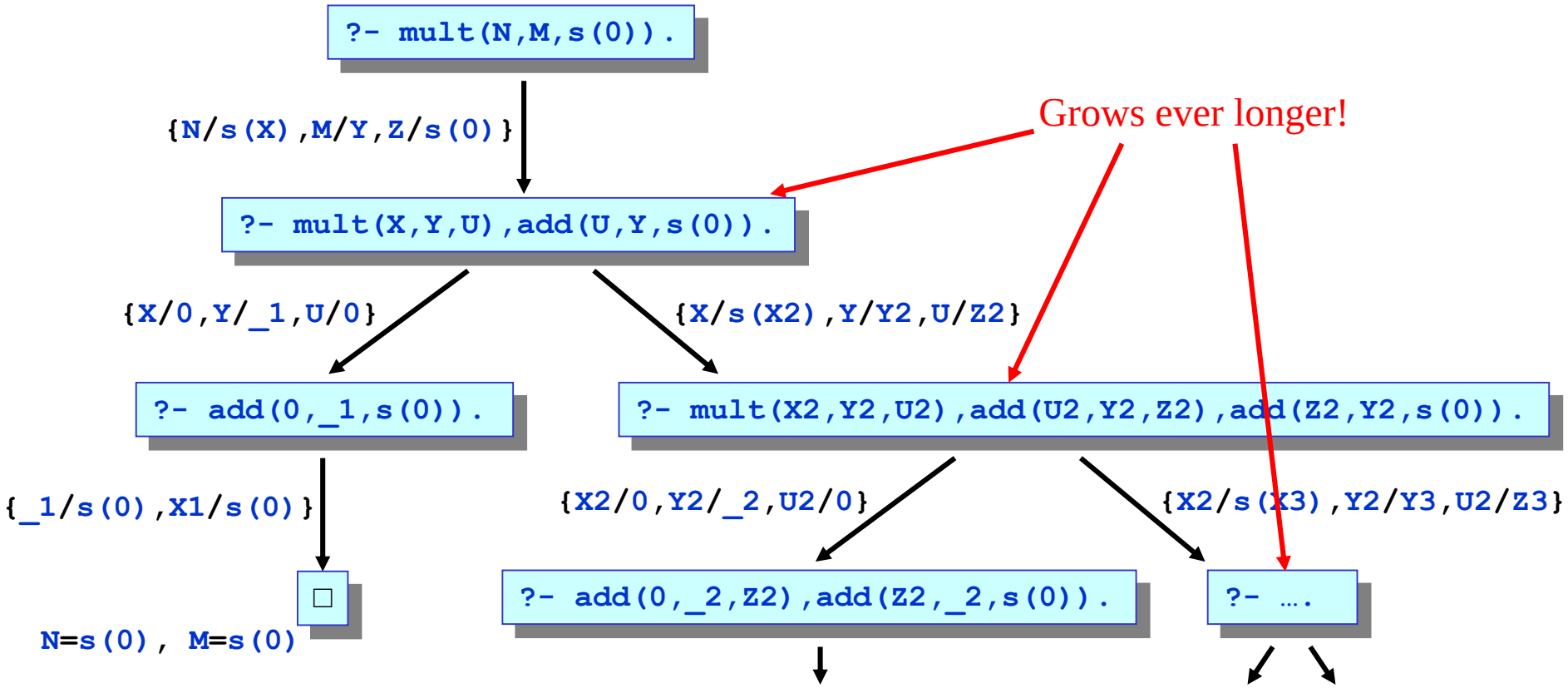
Detailed description of the generation of derivation trees

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 (the ones that are finished as empty), annotate the solution (the composition of unifiers – as functions on terms – along the path from the root, applied to all variables that occurred in the original query)

An example with infinite search

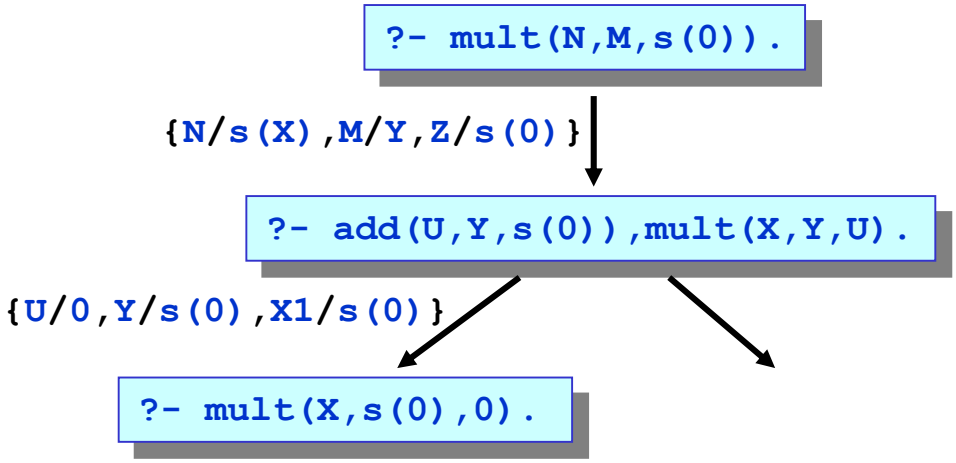
```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

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



Experiment with changed order of literals

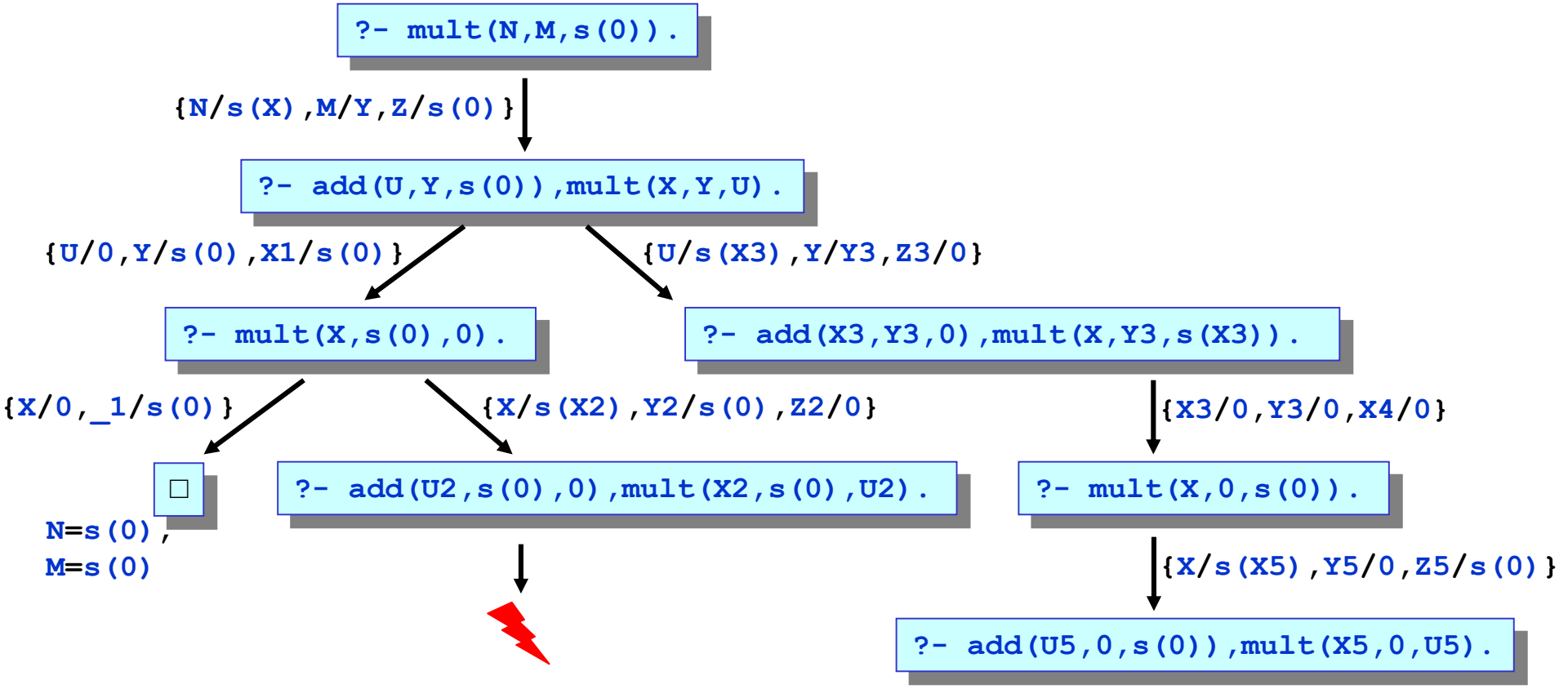
```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

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).
```



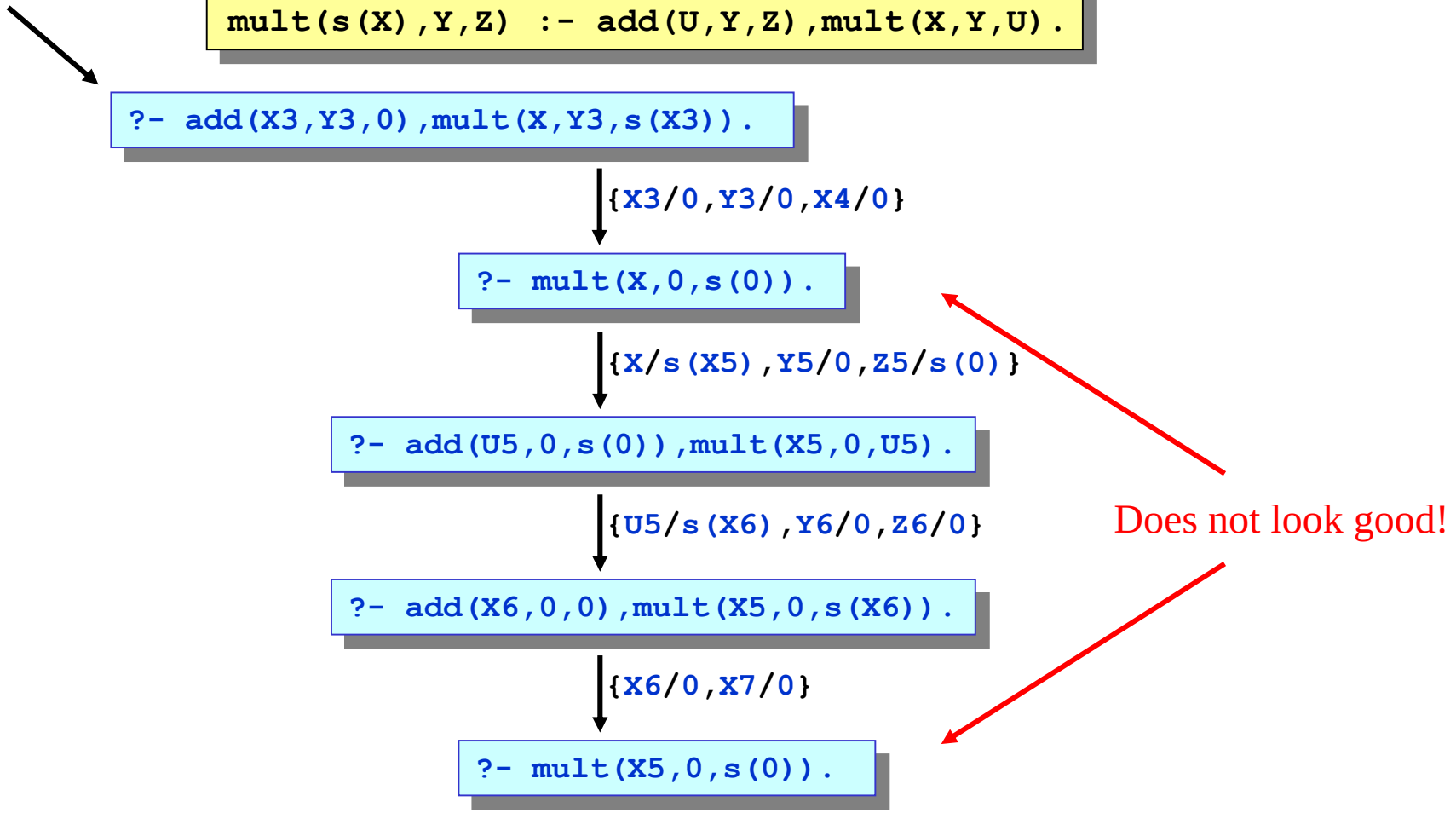
Experiment with changed order of literals

```
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).
```



Experiment with changed order of literals

```
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).
```



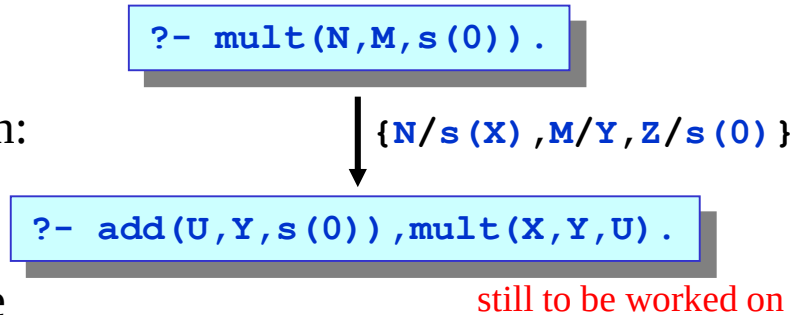
Detailed description of the generation of derivation trees

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:

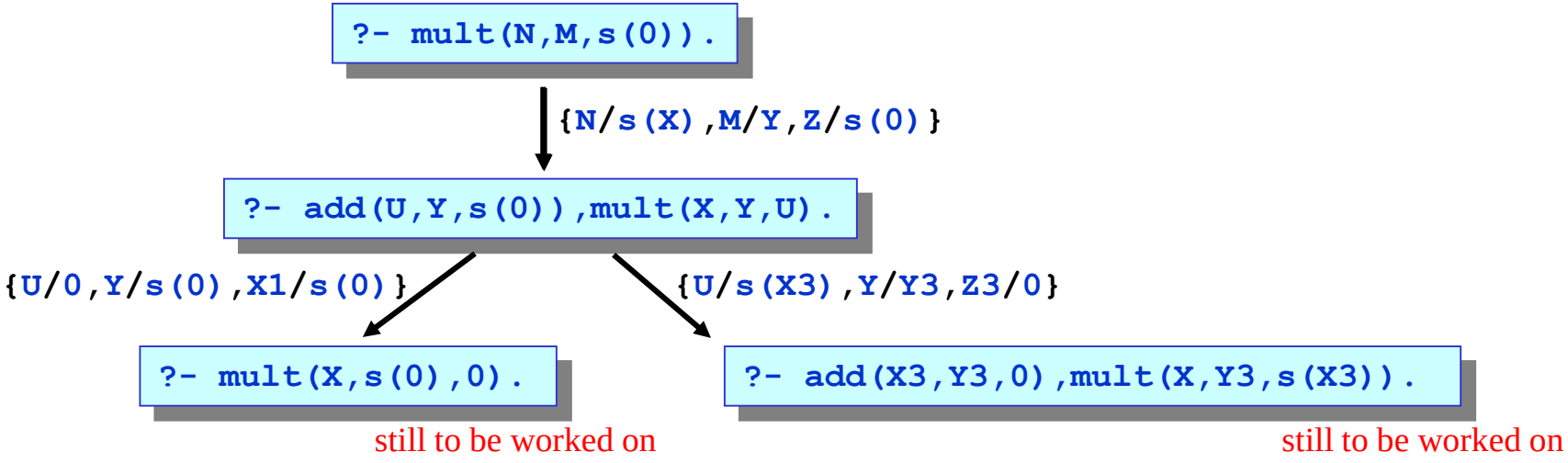
1. Generate root node with query, remember it as still to be worked on.
2. As long as there are still nodes to be worked on:
   - select left-most such node
   - determine all facts/rules (with consistently renamed variables) whose head is unifiable with the left-most literal in that node
   - generate 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



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).
```

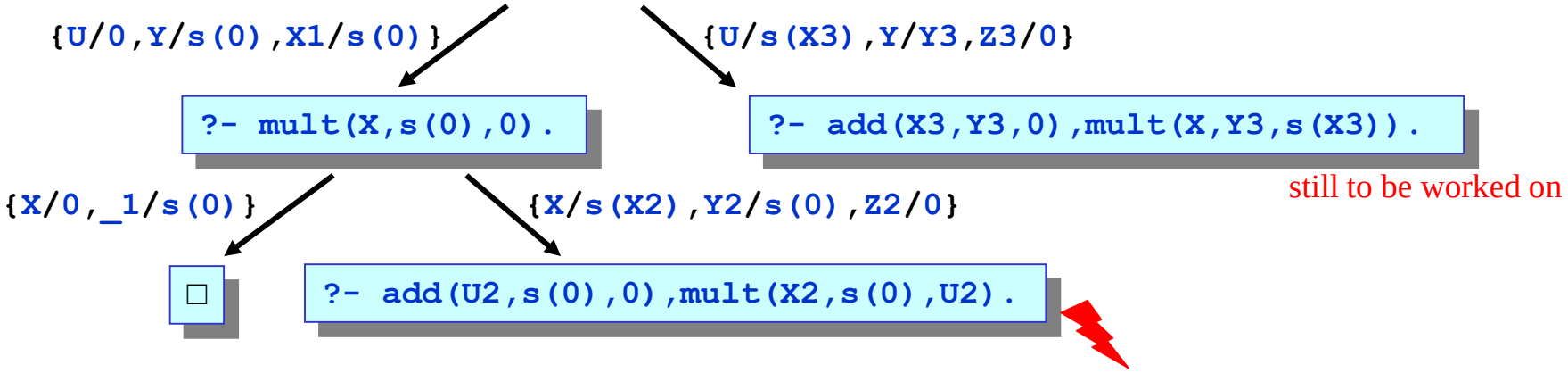
2. As long as there are still nodes to be worked on:
   - select left-most such node
   - determine all facts/rules (w. cons. renamed variables) whose head is unifiable with the left-most literal in that node
   - generate for each such fact/rule a (still to be worked on) successor node via a resolution 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



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).
```

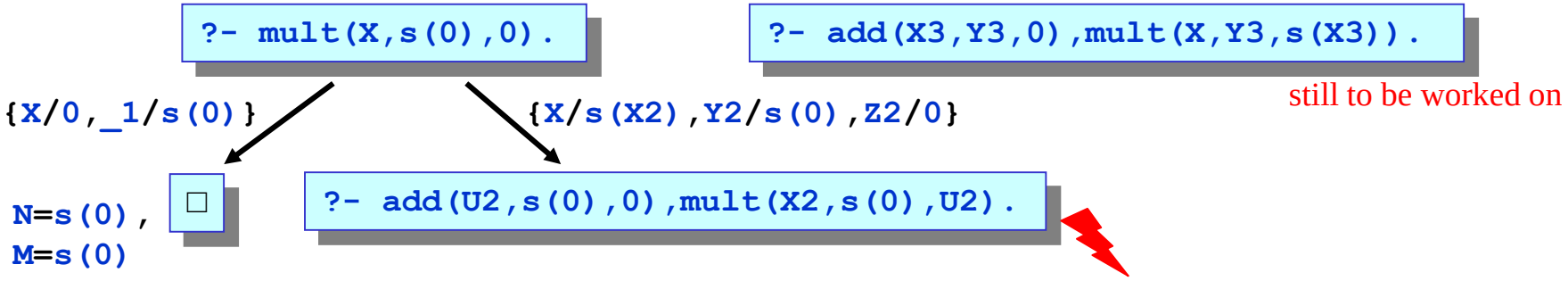
2. As long as there are still nodes to be worked on:
   - select left-most such node
   - determine all facts/rules (w. cons. renamed variables) whose head is unifiable with the left-most literal in that node
   - generate for each such fact/rule a (still to be worked on) successor node via a resolution 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 (↯)



Detailed description of the generation of derivation trees

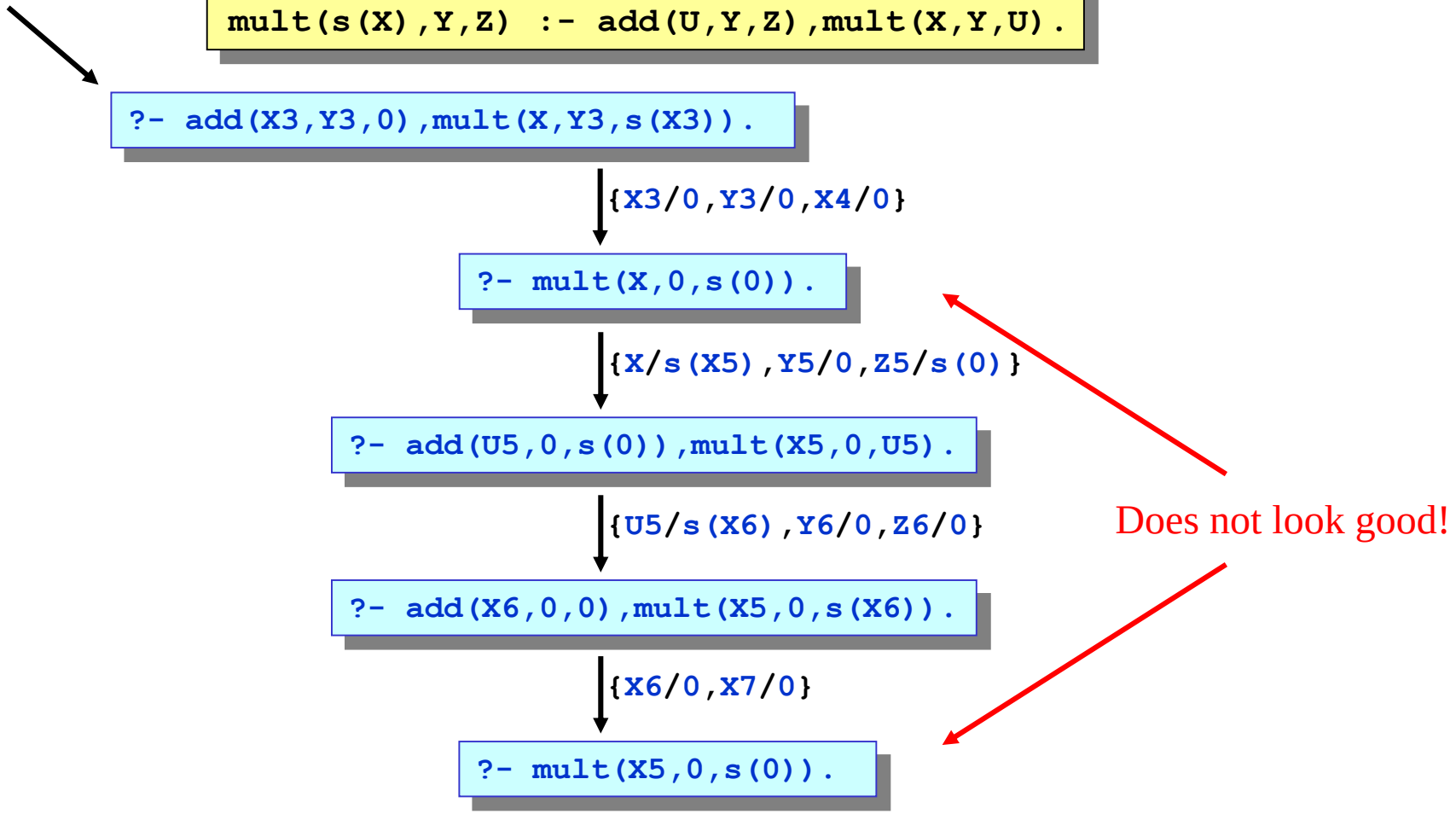
```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

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
   - 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)



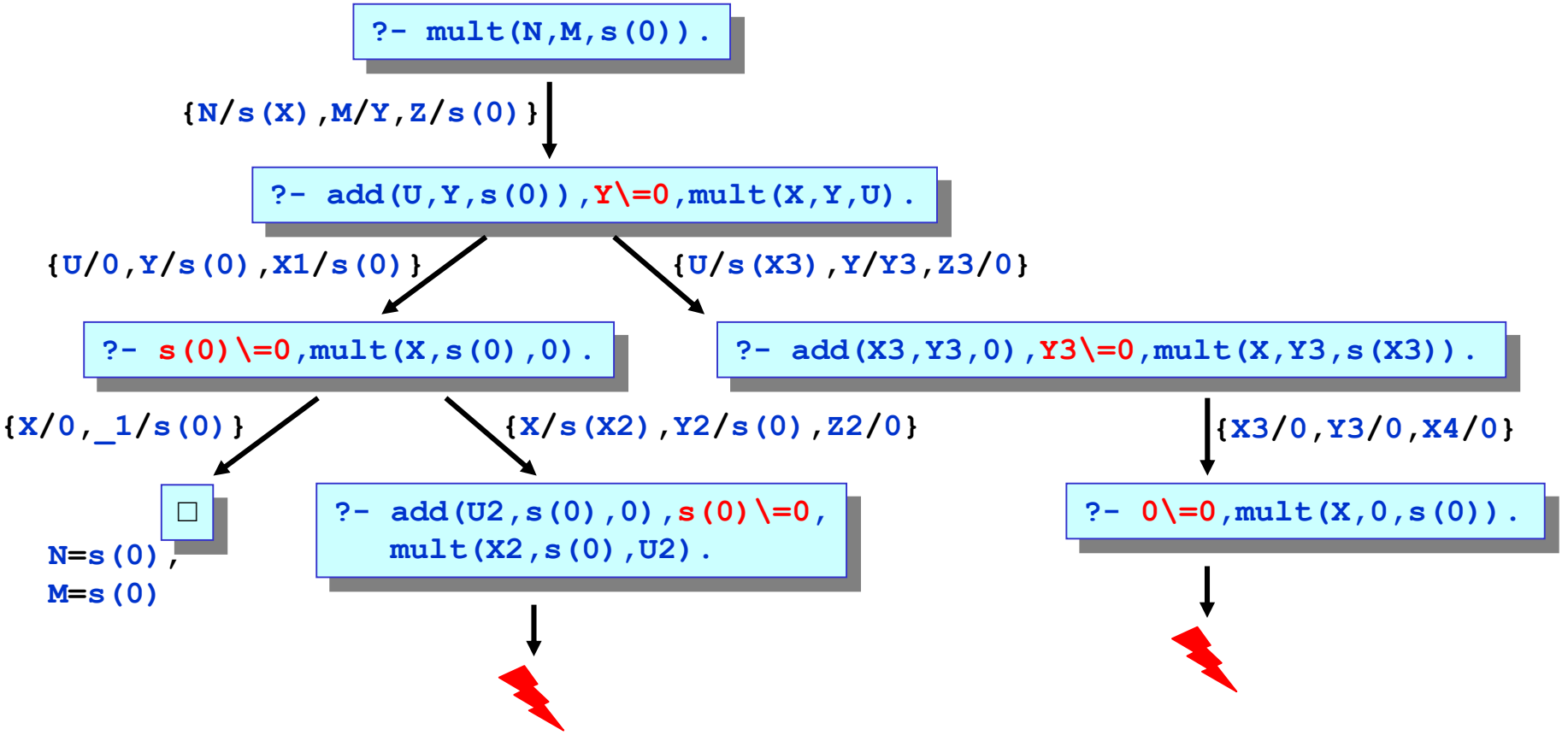
Back to the example: What to do?

```
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).
```



Attempt: introducing an extra test

```
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).
```



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!

Yet another “repair”

```
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):

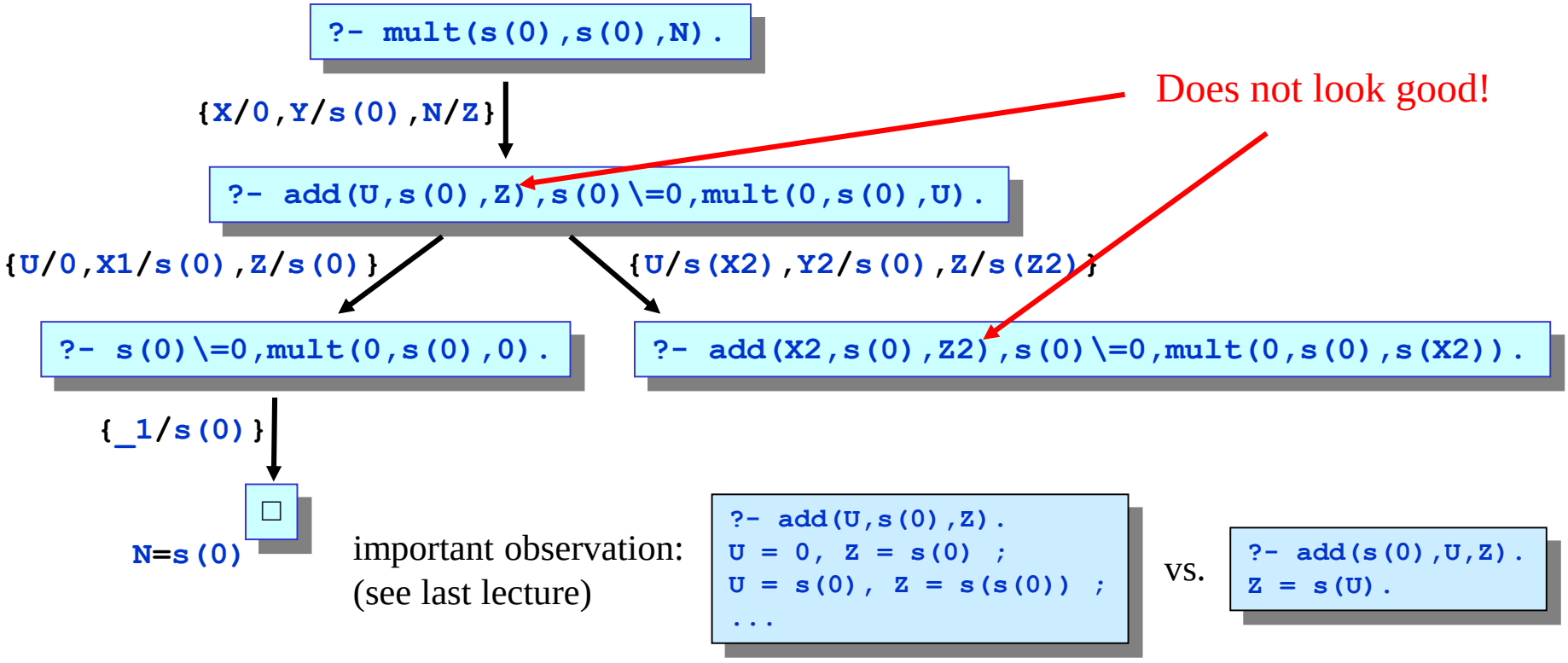
```
?- mult(s(0),s(0),N).
N = s(0) ;
abort
```

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).
```



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)

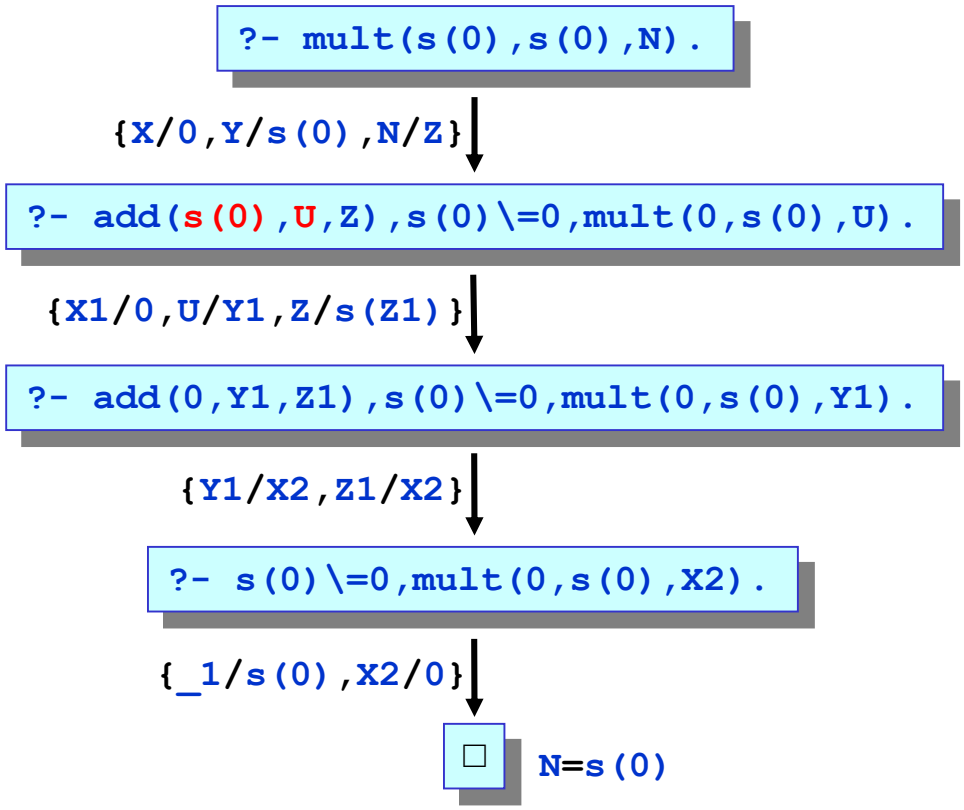
```
?- add(U,s(0),Z).
U = 0, Z = s(0) ;
U = s(0), Z = s(s(0)) ;
...
```

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).
```



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 all call modes work well, except `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

Programming Paradigms

Negation in Prolog

Negation (1)

- Logic programming is primarily based on a positive logic.

  A literal is provable if it can be reduced (possibly via several resolution steps) to the validity of known facts.

- But Prolog also offers the possibility to use **negation**.
  - However, Prolog negation is not fully compatible with the expected logical meaning.
  - `\+ Goal`, or `not(Goal)`, is provable if and only if `Goal` is not provable.

Example: `\+ member(4,[2,3])` is provable, since `member(4,[2,3])` is not provable, i.e., it exists a "finite failure tree".

Caution:

```
?- member(X,[2,3]).          ⇒ X = 2; X = 3.
?- \+ member(X,[2,3]).       ⇒ false.
?- \+ \+ member(X,[2,3]).    ⇒ true.
```

(Negation does not yield variable bindings.)

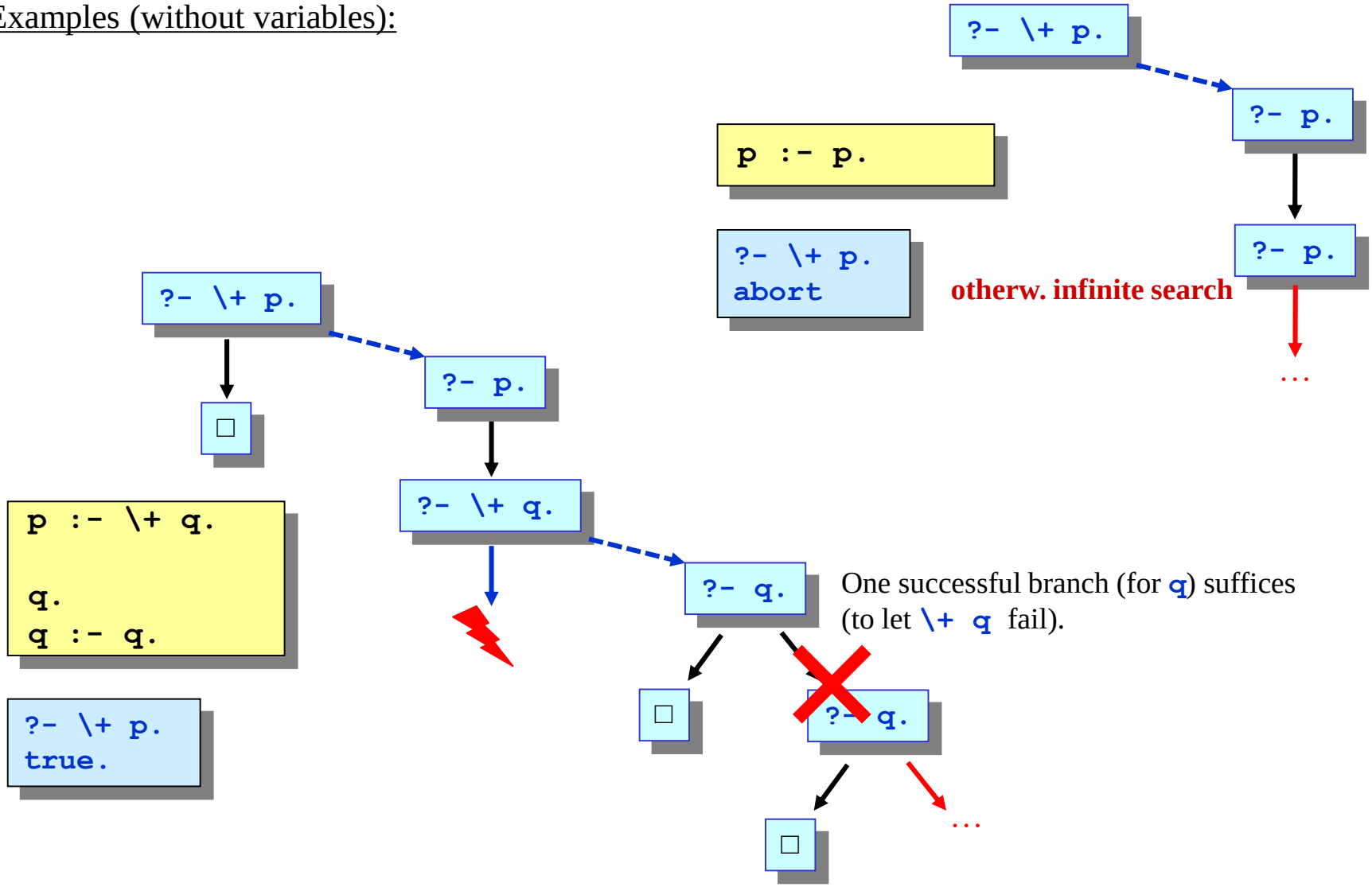
Negation (2)

- Why “finite failure tree”?
  - We cannot, in general, show that from the clauses of a program a certain negative statement follows.
  - We can only show that a certain positive statement can not be deduced. (negation as failure)
  - Here, “show” means to attempt a proof of the positive statement but to fail.
  - That any such attempt will necessarily fail (for some given positive statement) can only be said with certainty if the search space is finite.
- Underlying assumption:

closed world assumption

Negation (3)

Examples (without variables):



Negation (4)

Examples with variables:

```
human(marsellus).
human(vincent).
human(mia).

married(vincent,mia).
married(mia,vincent).

single(X) :- human(X), \+ married(X,Y).
```

```
?- single(X).
X = marsellus.

?- single(marsellus).
true.

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

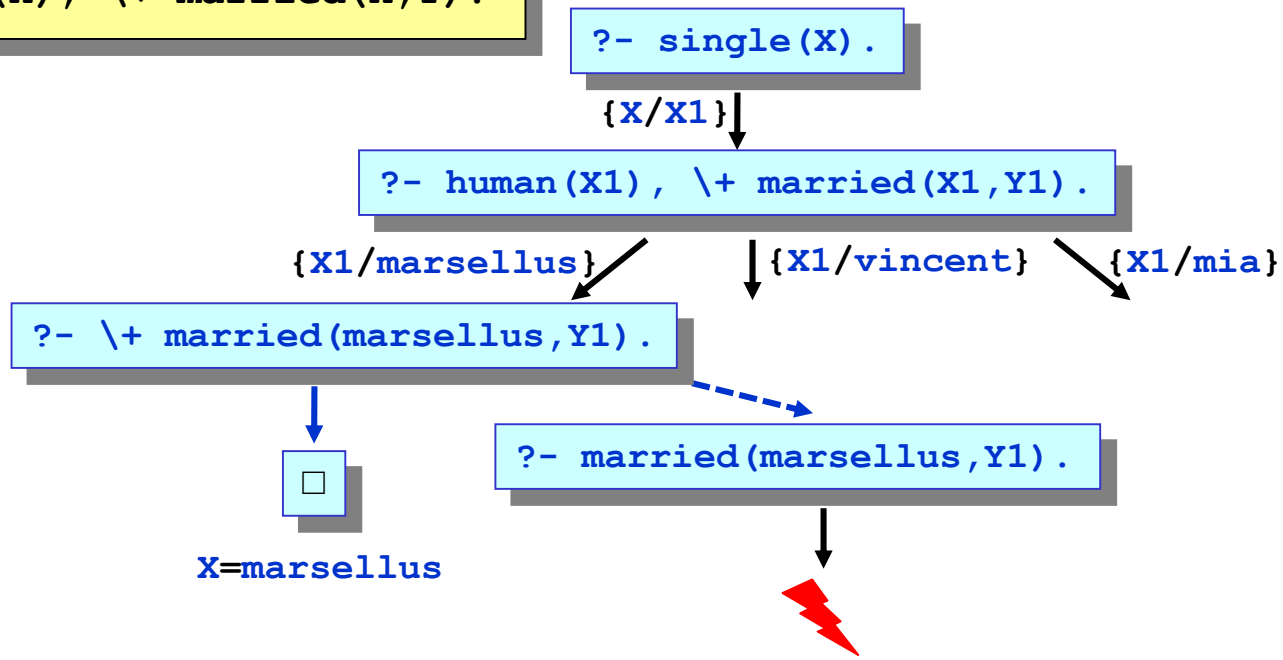
```
human(marsellus).
human(vincent).
human(mia).

married(vincent,mia).
married(mia,vincent).

single(X) :- \+ married(X,Y), human(X).
```

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

?- single(marsellus).
true.

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

Negation (5)

Examples with variables:

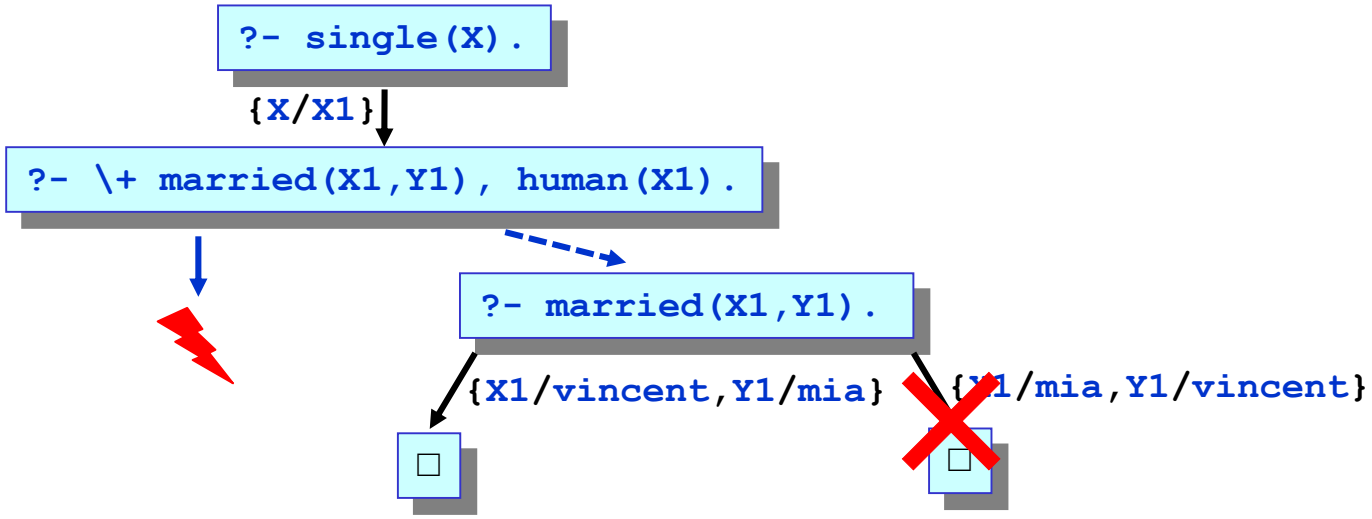
```
human(marsellus).
human(vincent).
human(mia).

married(vincent,mia).
married(mia,vincent).

single(X) :- human(X), \+ married(X,Y).
```



Negation (6)

Examples with variables:

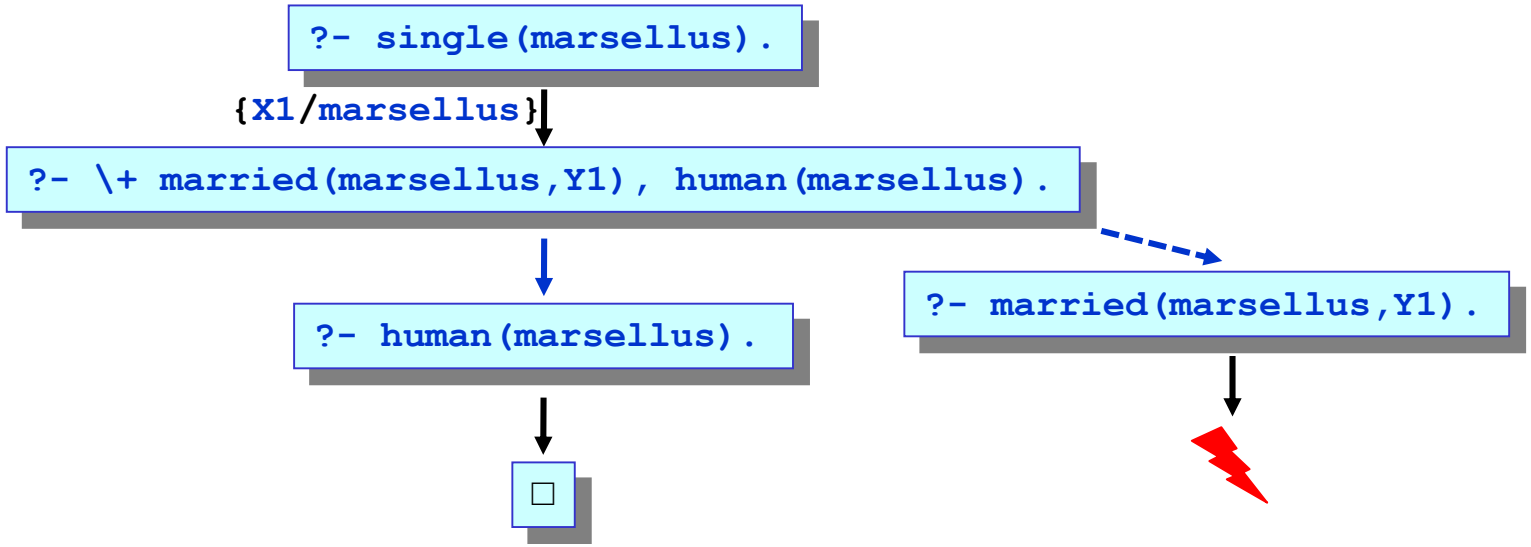
```
human(marsellus).
human(vincent).
human(mia).

married(vincent,mia).
married(mia,vincent).

single(X) :- \+ married(X,Y), human(X).
```



Negation (7)

Examples with variables:

```
human(marsellus).
human(vincent).
human(mia).

married(vincent,mia).
married(mia,vincent).

single(X) :- \+ married(X,Y), human(X).
```



Negation (8)

Explanation from “logical perspective” :

Under the assumptions that **`X`** is originally unbound and by **`human(X)`** will always be bound, this:

```
single(X) :- human(X), \+ married(X,Y).
```

means: $\forall$ X : human(X) $\wedge \neg(\exists$ Y : married(X,Y)) $\Rightarrow$ single(X).

But under the same assumptions, this:

```
single(X) :- \+ married(X,Y), human(X).
```

means: $\forall$ X : $\neg(\exists$ X,Y : married(X,Y)) $\wedge$ human(X) $\Rightarrow$ single(X).

Summary on Negation

- no real logical negation: instead, negation as failure
- proof search in “side branch”, does not bind variables to the outside
- can only be truly understood procedurally/operationally
- problems with attempting a declarative perspective:
  - not compositional
  - sensitive against changes to the order of, and within, facts and rules
  - T_P-operator would be non-monotone

Potentielle Probleme mit Rekursion

Alte Beispielaufgabe:

Given is an arbitrary database of facts about (true) lines between points in the plane, for example:

```
line(a, b). line(c, b). line(d, a).
line(b, d). line(d, c). line(d, e).
```

Implement predicates triangle with arity 3 and tetragon with arity 4, for the (true) triangles and tetragons created by the lines in the database. A line or triangle or tetragon is "true" if no two listed points are the same.

Also note that the line relation given above is not symmetric, even though lines between points should conceptually be considered to be so.

Lösungsversuch:

```
triangle(X,Y,Z) :- line(X,Y),line(Y,Z),line(Z,X).

tetragon(X,Y,Z,U) :- line(X,Y),line(Y,Z),line(Z,U),
                     line(U,X), X \= Z, Y \= U.
```

Potentielle Probleme mit Rekursion

Um die „fehlenden“ (per Symmetrie) Fakten wie „line(b,a)“ etc. zu berücksichtigen, liegt folgende Ergänzung nahe:

```
line(X,Y) :- line(Y,X).
```

Allerdings ist das leider „zu rekursiv“ (bei Ausführung).

Besser hier, Einführung eines gesonderten Prädikates und dann:

```
sline(X,Y) :- line(X,Y).
sline(Y,X) :- line(X,Y).

triangle(X,Y,Z) :- sline(X,Y), sline(Y,Z), sline(Z,X).

tetragon(X,Y,Z,U) :- sline(X,Y), sline(Y,Z), sline(Z,U),
                     sline(U,X), X \= Z, Y \= U.
```

Lösung bestand hier also im Verzicht auf Rekursion.

Potentielle Probleme mit Rekursion


```
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).
```

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

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

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

Potentielle Probleme mit Rekursion


```
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
```

Potentielle Probleme mit Rekursion


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

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

```
?- connection(san_francisco,Y).
Y = honolulu ;
Y = maui ;
Y = san_francisco ;
Y = honolulu ;
Y = maui ;
Y = san_francisco ;
Y = honolulu ;
Y = maui ; …
```

Ziel sollte sein: Endlossuche vermeiden

Potentielle Probleme mit Rekursion

Idee: schon bereiste Zwischenstationen merken, zum Beispiel als Liste:

```
direct(frankfurt,san_francisco).
…
direct(honolulu,san_francisco).

connection(X, Y) :- connection1(X, Y, [X]).

connection1(X, Y, _) :- direct(X, Y).
connection1(X, Y, L) :- direct(X, Z), not(member(Z,L)),
                        append([Z], L, L1),
                        connection1(Z, Y, L1).
```

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

Spezielle Datenstruktur: Listen

Neben Konstanten und per Schachtelung von Datenkonstruktoren wie `s/1` und `z/0` zu erhaltenden Datenstrukturen, wurden auch Listen mit Syntax wie `[1,2,3,4,5]` und `[duisburg,X,essen]` zuvor bereits kurz erwähnt.

Zur Arbeit mit Listen hält Prolog diverse Prädikate bereit, zum Beispiel:

- `member/2`, um auszudrücken, dass ein Element in einer Liste vorkommt
- `append/3`, um auszudrücken, dass eine Liste die Aneinanderhängung zweier bestimmter Listen ist
- `length/2`, um auszudrücken, welche Länge eine Liste hat

Vordefinierte Prädikate auf Listen

Interessant dabei ist, dass (ganz im Sinne „unseres“ add/3-Prädikates) diverse Aufrufmodi der Listenprädikate funktionieren. Zum Beisiel:

```
?- member(3,[1,2,3,4,5]).
true.

?- member(X,[1,2,3]).
X = 1 ;
X = 2 ;
X = 3.

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

Vordefinierte Prädikate auf Listen

Auch für die anderen Listenprädikate, zum Beispiel:

```
?- append([1,2,3],[4,5],L).
L = [1,2,3,4,5].

?- append(X,Y,[a,b]).
X = [], Y = [a,b] ;
X = [a], Y = [b] ;
X = [a,b], Y = [].

?- append(X,X,[a,b]).
false.

?- append(X,X,[a,Y]).
X = [a], Y = a.
```

Vordefinierte Prädikate auf Listen

Oder:

```
?- length([a,b,c],N).
N = 3.

?- length(L,3).
L = [_1570, _1576, _1582].

?- length(L,3),append(X,X,L).
false.

?- length(L,4),append(X,X,L).
L = [_1610, _1616, _1610, _1616], X = [_1610, _1616].

?- length(L,2),member(a,L),member(b,L),member(c,L).
false.
```

Definition von Prädikaten auf Listen

UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

Definiert werden Prädikate auf Listen typischerweise durch Verwendung bereits vorhandener:

```
insert(X,L,R) :- append(U,V,L),append(U,[X],Y),
                 append(Y,V,R).
```

und/oder Rekursion:

```
permutation([],[]).
permutation(L,P) :- append([X],Y,L),permutation(Y,Z),
                    insert(X,Z,P).
```

Mit obigen Definitionen, zum Beispiel:

```
?- permutation([1,2,3],L)
L = [1,2,3] ;
L = [2,1,3] ;
...
```

Generate and Test – Was und Wie

Angenommen, wir möchten Listen sortieren können, also zum Beispiel Anfragen wie

```
?- sortingTo([4,2,6,9,1],R).
```

stellen können, und darauf als Antwort

```
R = [1,2,4,6,9].
```

erhalten.

Also **was** wollen wir genau?

Generate and Test – Was und Wie

Die gewünschte Eigenschaft der Ergebnisliste können wir relativ leicht als Prolog-Prädikat ausdrücken:

```
isSorted([]).
isSorted([_]).
isSorted(Xs) :- append([X,Y],Ys,Xs), X =< Y,
                append([Y],Ys,Zs),isSorted(Zs).
```

Dann gilt zum Beispiel:

```
?- isSorted([4,2,6,9,1]).
false.
?- isSorted([1,2,4,6,9]).
true.
```

Und **wie** können wir eine solche passende Liste erhalten bzw. herstellen?

Generate and Test – Was und Wie

Nun, eine recht naive, aber funktionierende Lösung wäre:

```
sortingTo(Xs,Ys) :- permutation(Xs,Ys),isSorted(Ys).
```

Dann in der Tat:

```
?- sortingTo([4,2,6,9,1],R).
R = [1,2,4,6,9].
```

Prinzip hier:

Um eine Regel auf Eingaben zu formulieren, die genau dann **true** liefert, wenn eine gültige Lösung des Problems vorliegt, Zerlegung in zwei Teile:

- **Generate**-Teil definiert einen Suchraum.
- **Test**-Teil definiert die Bedingung, die erfüllt sein muss.

Generate and Test – Weitere Beispiele

Aufgabe: Ermittle alle Möglichkeiten, bei dreimaligem Würfeln insgesamt 15 Punkte zu erzielen.

Lösung:

```
?- W = [1,2,3,4,5,6], member(A,W), member(B,W),
   member(C,W), A + B + C =:= 15.
```

Generate and Test – Weitere Beispiele

Aufgabe: Ermittle alle Möglichkeiten, bei dreimaligem Würfeln mit verschiedenen Augenzahlen insgesamt 15 Punkte zu erzielen.

Lösung:

```
?- W = [1,2,3,4,5,6], member(A,W), member(B,W),
   member(C,W), A \= B, A \= C, B \= C,
   A + B + C =:= 15.
```

oder:

```
?- permutation([1,2,3,4,5,6],[A,B,C,_,_,_]),
   A + B + C =:= 15.
```

Generate and Test – Weitere Beispiele

Aufgabe: Ermittle alle Möglichkeiten, bei dreimaligem Würfeln mit verschiedenen Augenzahlen in aufsteigender Reihenfolge insgesamt 15 Punkte zu erzielen.

Lösung:

```
?- permutation([1,2,3,4,5,6],[A,B,C,_,_,_]),
   isSorted([A,B,C]), A + B + C =:= 15.
```

Generate-and-Test ist sinnvoll einzusetzen bei nicht-trivialen kombinatorischen Problemen, wenn

- die Menge der potentiellen Lösungen endlich oder besser sogar recht klein ist, oder
- man keine Vorstellung darüber hat, wie systematisch schneller eine Lösung gefunden werden könnte.

Beispiel: Krypto-Arithmetik

ABB	**−**	**CD**	**=**	**EED**
–		**–**		*****
FD	**+**	**EF**	**=**	**CE**
=		**=**		**=**
EGD	*****	**FH**	**=**	**?**

Jeder Buchstabe entspreche einer anderen Ziffer.

Wie lautet eine gültige Belegung?

Beispiel: Krypto-Arithmetik


```
solve(A,B,C,D,E,F,G,H) :- generate(A,B,C,D,E,F,G,H),
                          test(A,B,C,D,E,F,G,H).

generate(A,B,C,D,E,F,G,H) :-
    permutation([0,1,2,3,4,5,6,7,8,9],
                [A,B,C,D,E,F,G,H,_,_]).

test(A,B,C,D,E,F,G,H) :- ???
```

Zum Beispiel die erste Zeile entspricht:

```
(A * 100 + B * 10 + B) - (C * 10 + D)
    =:= E * 100 + E * 10 + D
```

Und die erste Spalte:

```
(A * 100 + B * 10 + B) - (F * 10 + D)
    =:= E * 100 + G * 10 + D
```

Beispiel: Krypto-Arithmetik

Zweite Zeile und zweite Spalte:

$$(F * 10 + D) + (E * 10 + F) =:= C * 10 + E,$$
$$(C * 10 + D) - (E * 10 + F) =:= F * 10 + H$$

Schließlich noch die Bedingung, dass gleiches Ergebnis in letzter Zeile und letzter Spalte:

$$(E * 100 + E * 10 + D) * (C * 10 + E)$$
$$=:= (E * 100 + G * 10 + D) * (F * 10 + H)$$

Beispiel: Krypto-Arithmetik

Insgesamt für den Test-Teil:

```
test(A,B,C,D,E,F,G,H) :-
    (A * 100 + B * 10 + B) - (C * 10 + D)
        =:= E * 100 + E * 10 + D,
    (A * 100 + B * 10 + B) - (F * 10 + D)
        =:= E * 100 + G * 10 + D,
    (F * 10 + D) + (E * 10 + F) =:= C * 10 + E,
    (C * 10 + D) - (E * 10 + F) =:= F * 10 + H,
    (E * 100 + E * 10 + D) * (C * 10 + E)
        =:= (E * 100 + G * 10 + D) * (F * 10 + H).
```

Als eindeutige erfüllende Belegung findet Prolog mit der Anfrage

```
?- solve(A,B,C,D,E,F,G,H).
```

dies: A = 2, B = 0, C = 8, D = 5, E = 1, F = 6, G = 3, H = 9.

Beispiel: Krypto-Arithmetik

200	−	85	=	115
−		−		*
65	+	16	=	81
=		=		=
135	*	69	=	9315

Beispiel: Einstein's Riddle

Zur Erinnerung:

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.

Beispiel: Einstein's Riddle

Versuchen wir, das Rätsel per Generate-and-Test zu lösen.

Für den Generate-Teil wäre zunächst einfach denkbar:

```
Houses = [ _, _, _, _, _ ]
```

Oder auch bereits:

```
Houses = [ [ _, _, _, _, _ ]
         , [ _, _, _, _, _ ]
         , [ _, _, _, _, _ ]
         , [ _, _, _, _, _ ]
         , [ _, _, _, _, _ ] ]
```

Beispiel: Einstein's Riddle

UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

Für den Test-Teil nehmen wir uns die einzelnen Hinweise vor.

Zum Beispiel:

1. The Englishman lives in the red house.

Unter der Festlegung, dass wir die einzelnen Attribute jeweils in der Reihenfolge „color“, „nationality“, „drink“, „pet“, „smoke“ angeben werden, können wir diesen ersten Hinweis wie folgt ausdrücken:

```
member([ red, englishman, _, _, _ ], Houses)
```

Analog:

2. The Spaniard owns the dog.

wird zu:

```
member([ _, spaniard, _, dog, _ ], Houses)
```

Beispiel: Einstein's Riddle

Die nächsten beiden Hinweise:

3. Coffee is drunk in the green house.
4. The Ukrainian drinks tea.

werden auch analog behandelt:

```
member([ green, _, coffee, _, _ ], Houses)
```

bzw.:

```
member([ _, ukrainian, tea, _, _ ], Houses)
```

Beispiel: Einstein's Riddle

Dann wird es wieder etwas interessanter:

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

Das könnten wir so ausdrücken:

```
rightOf([ green, _, _, _, _ ],
        [ ivory, _, _, _, _ ],
        Houses)
```

wenn wir ein solches Prädikat hätten.

Definieren wir es uns doch einfach:

```
rightOf(R,L,List) :- append(Prefix,_,List),
                     append(_,[L,R],Prefix).
```

Beispiel: Einstein's Riddle

Nun kommen nochmal zwei sehr einfach umzusetzende Hinweise:

6. The Winston smoker owns snails.
7. Kools are smoked in the yellow house.

Dann wieder spannender:

8. Milk is drunk in the middle house.
9. The Norwegian lives in the leftmost house.

Diese beiden können wir umsetzen, indem wir

```
Houses = ...
```

verfeinern zu:

```
Houses = [ [ _, norwegian, _, _, _ ], _
         , [ _, _, milk, _, _ ], _, _ ]
```

Beispiel: Einstein's Riddle

Für den nächsten Hinweis:

10. The man who smokes Chesterfield lives in the house next to the man with the fox.

brauchen wir nochmal ein Hilfsprädikat:

```
nextTo([ _, _, _, _, chesterfield ],
       [ _, _, _, fox, _ ],
       Houses)
```

welches wir wie folgt definieren können:

```
nextTo(X,Y,List) :- rightOf(X,Y,List).
nextTo(X,Y,List) :- rightOf(Y,X,List).
```

Beispiel: Einstein's Riddle

Die restlichen Hinweise:

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.

lassen sich dann alle analog zu schon vertrauten umsetzen.

Es bleibt noch, letztlich den Zebra-Besitzer und den Wasser-Trinker zu bestimmen.

Dazu können Variablen und weitere member-Aufrufe verwendet werden.

Beispiel: Einstein's Riddle – Gesamtlösung


```
rightOf(R, L, List) :- append(Prefix, _, List), append(_, [L, R], Prefix).
nextTo(X, Y, List) :- rightOf(X, Y, List).
nextTo(X, Y, List) :- rightOf(Y, X, List).
solve(ZebraOwner, WaterDrinker) :-
        Houses = [ [ _, norwegian, _, _, _ ], _, [ _, _, milk, _, _ ], _, _ ],
        member([ red, englishman, _, _, _ ], Houses),
        member([ _, spaniard, _, dog, _ ], Houses),
        member([ green, _, coffee, _, _ ], Houses),
        member([ _, ukrainian, tea, _, _ ], Houses),
        rightOf([ green, _, _, _, _ ], [ ivory, _, _, _, _ ], Houses),
        member([ _, _, _, snails, winston ], Houses),
        member([ yellow, _, _, _, kools ], Houses),
        nextTo([ _, _, _, _, chesterfield ], [ _, _, _, fox, _ ], Houses),
        nextTo([ _, _, _, _, kools ], [ _, _, _, horse, _ ], Houses),
        member([ _, _, juice, _, lucky ], Houses),
        member([ _, japanese, _, _, parliaments ], Houses),
        nextTo([ _, norwegian, _, _, _ ], [ blue, _, _, _, _ ], Houses),
        member([ _, ZebraOwner, _, zebra, _ ], Houses),
        member([ _, WaterDrinker, water, _, _ ], Houses).
```