Programming Paradigms – Prolog part

Summer Term 2025

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).
```

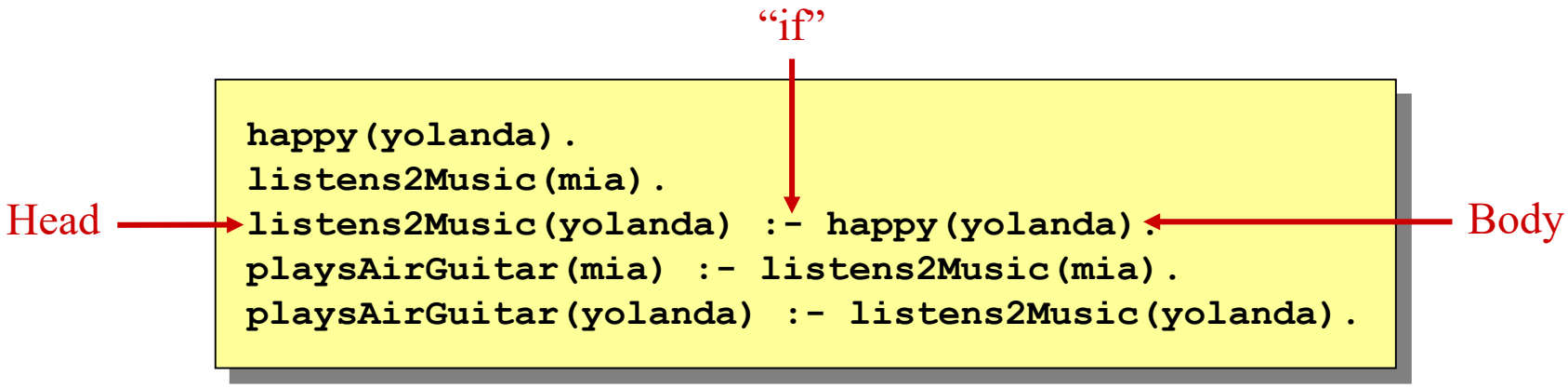
- Queries:

```
?- 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.
```

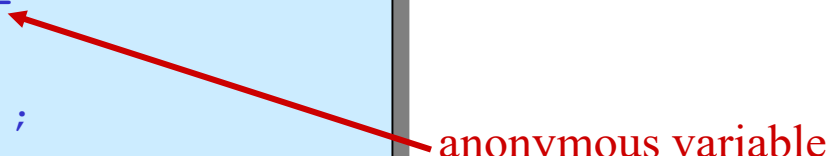
semicolon entered by user

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

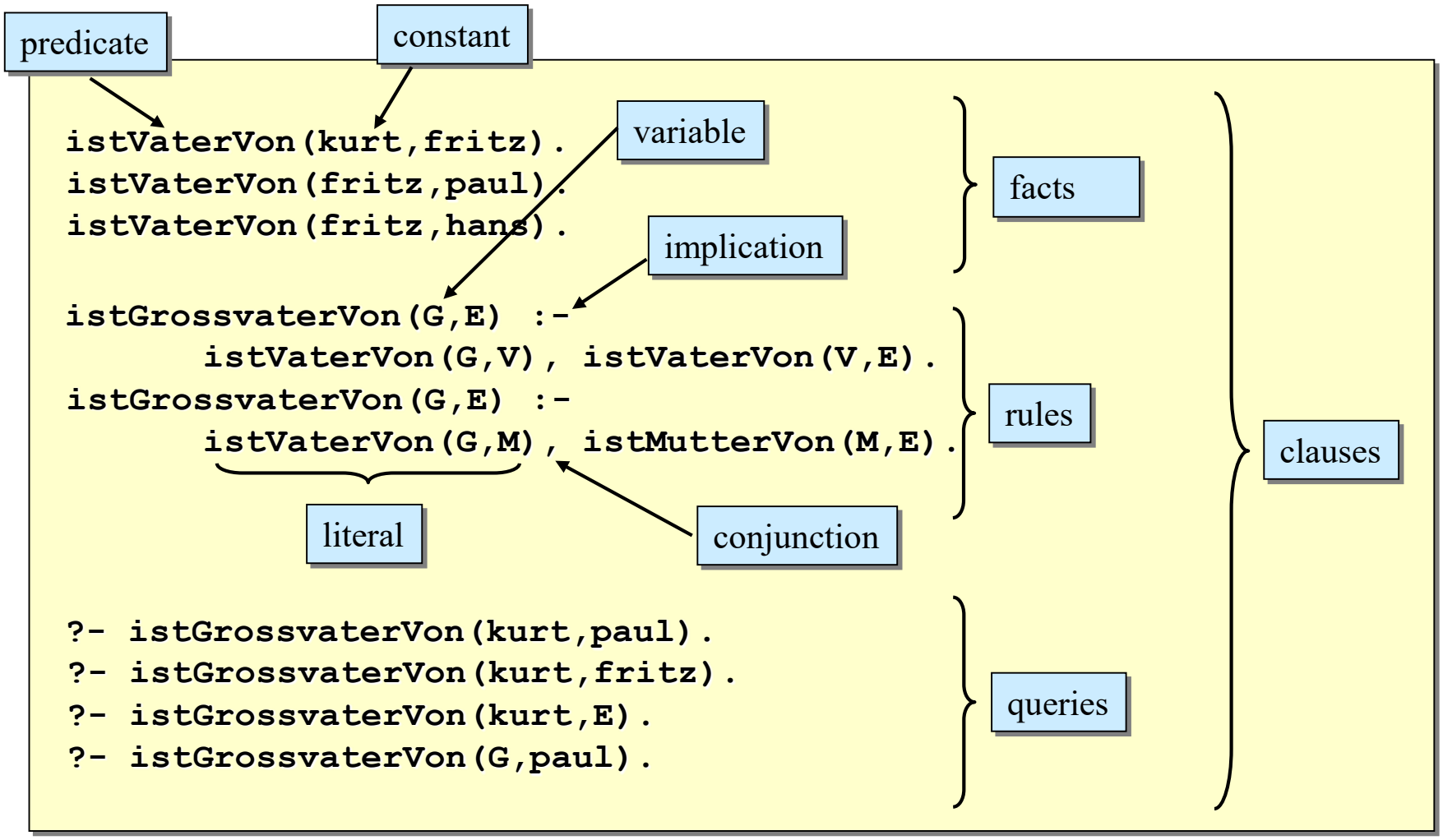
```
loves(vincent,mia).
loves(marsellus,mia).
loves(mia,vincent).

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

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”, …)

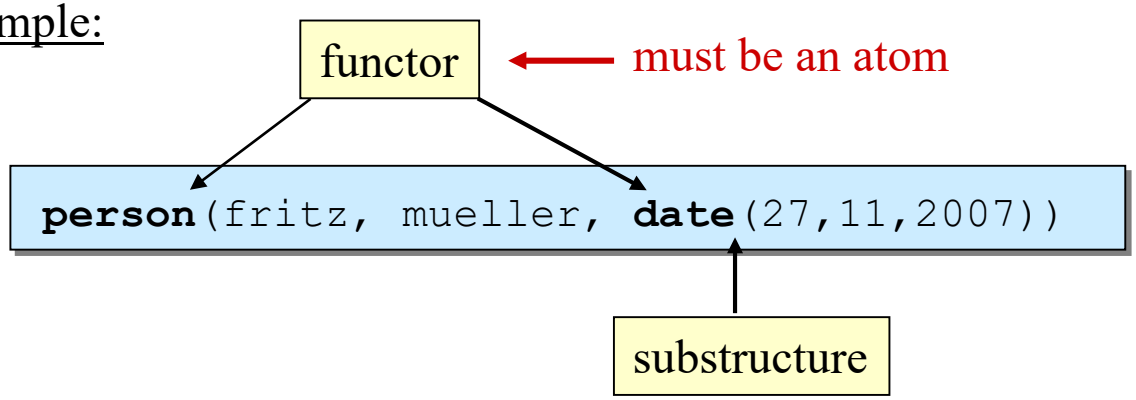
- Note: Prolog has no type system!

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 can be 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

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
```

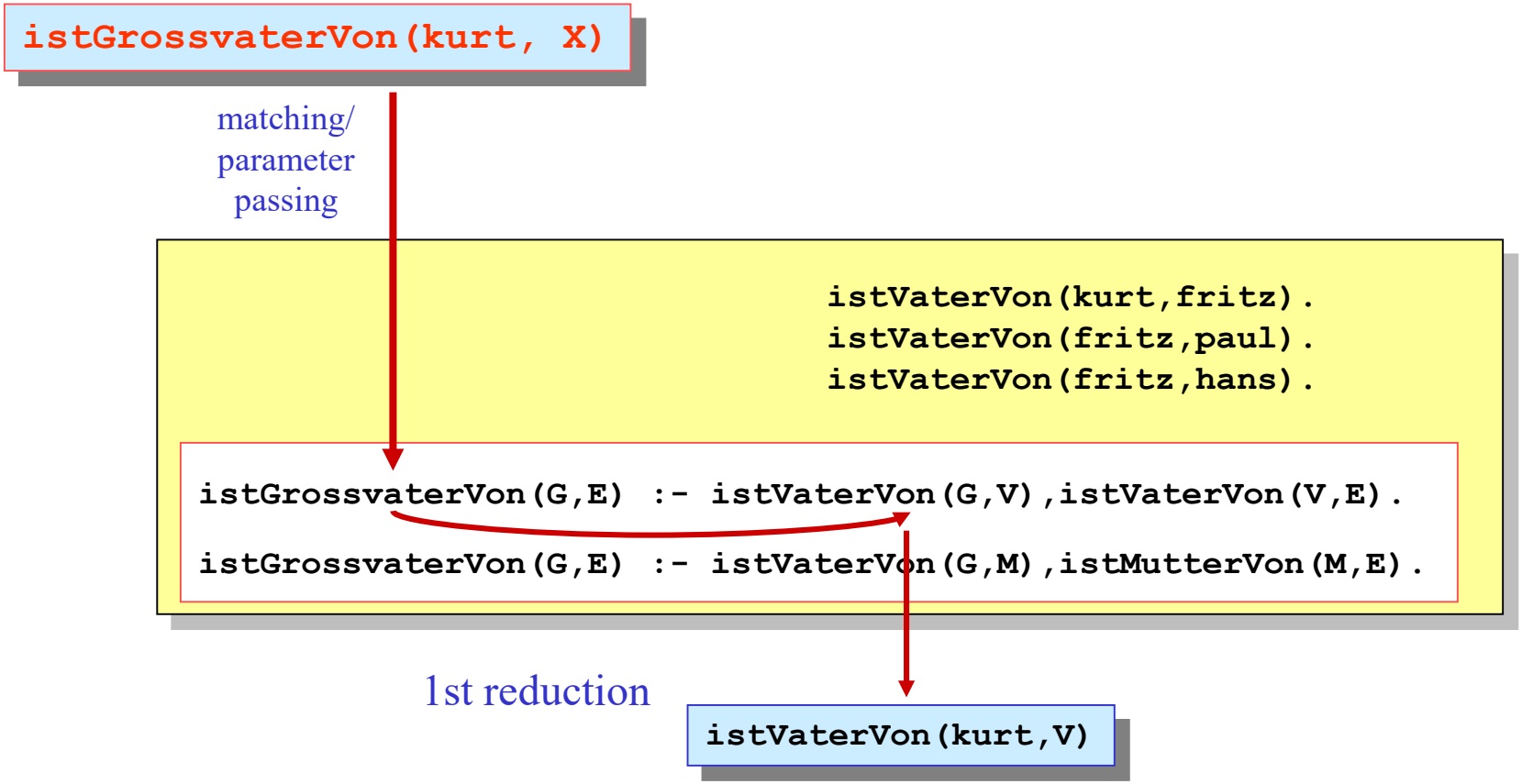
Input: a query

(repeated) resolution

Output: variable substitution(s)

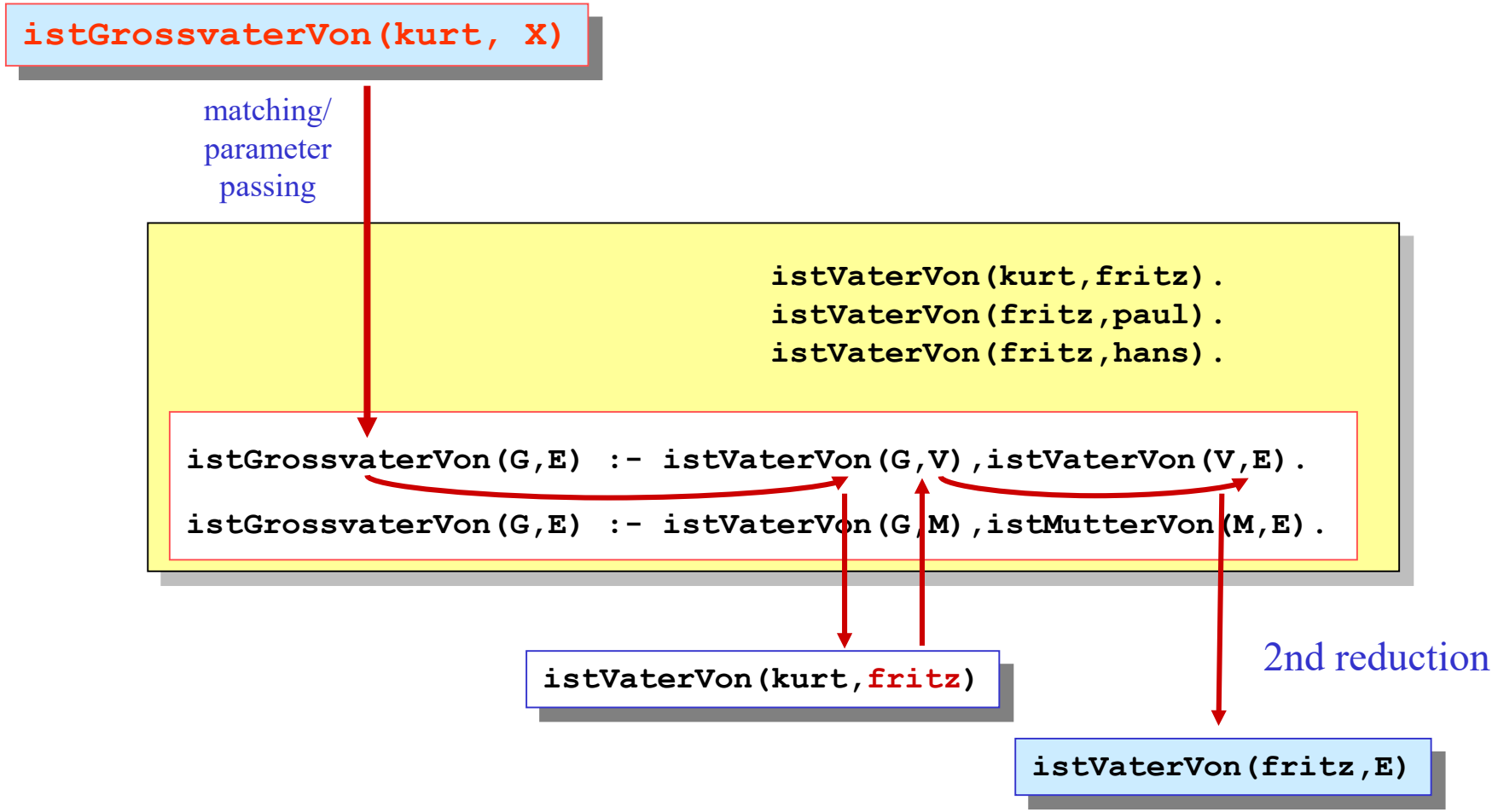
Operationalisation in Prolog (1)

Principle: reduction to subproblems



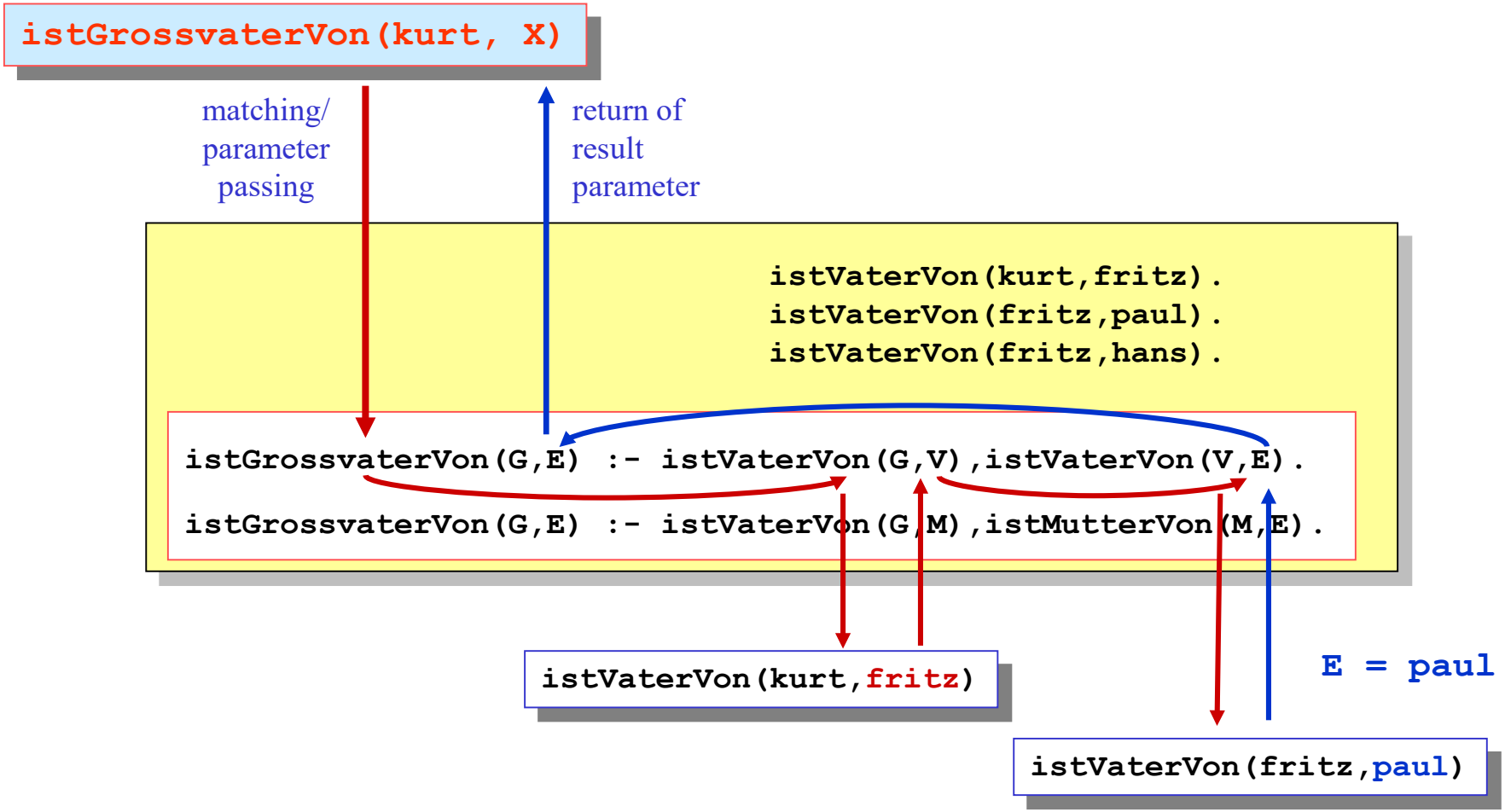
Operationalisation in Prolog (2)

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



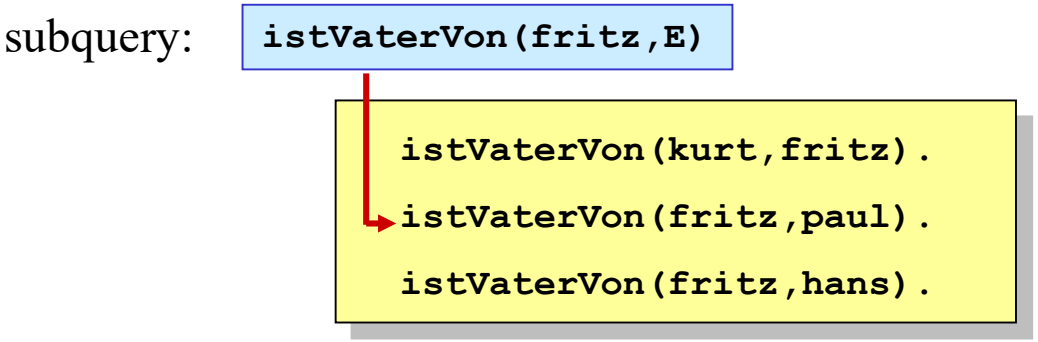
Operationalisation in Prolog (3)

Principle: reduction to subproblems

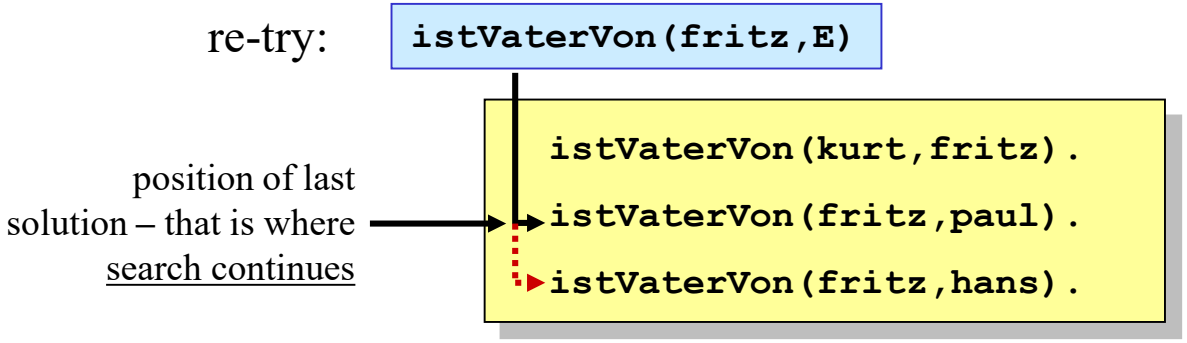


Operationalisation in Prolog (4)

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

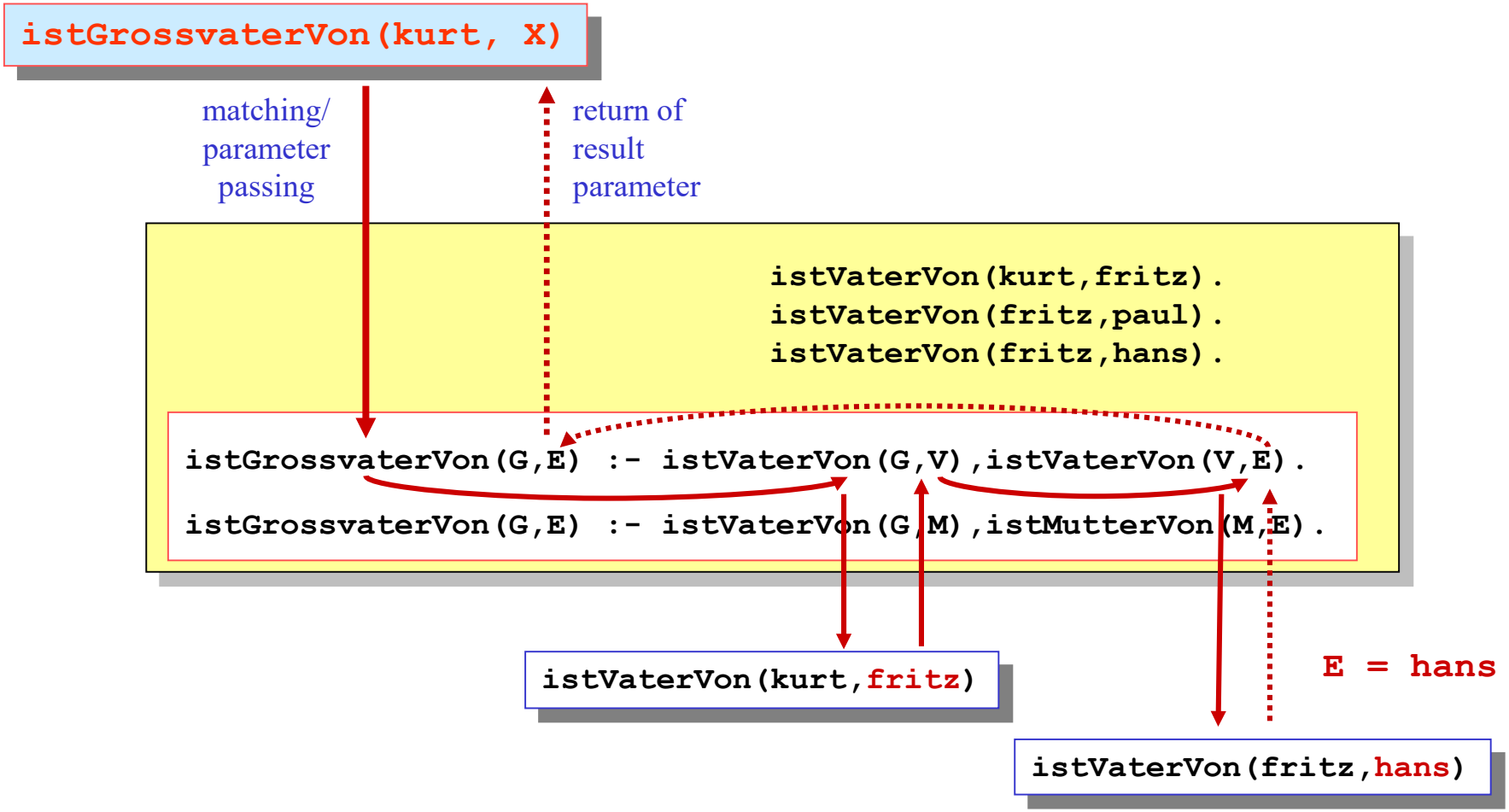


- Since a relation generally is not a unique mapping, further answers for a (sub)query may exist. Prolog finds those using backtracking:



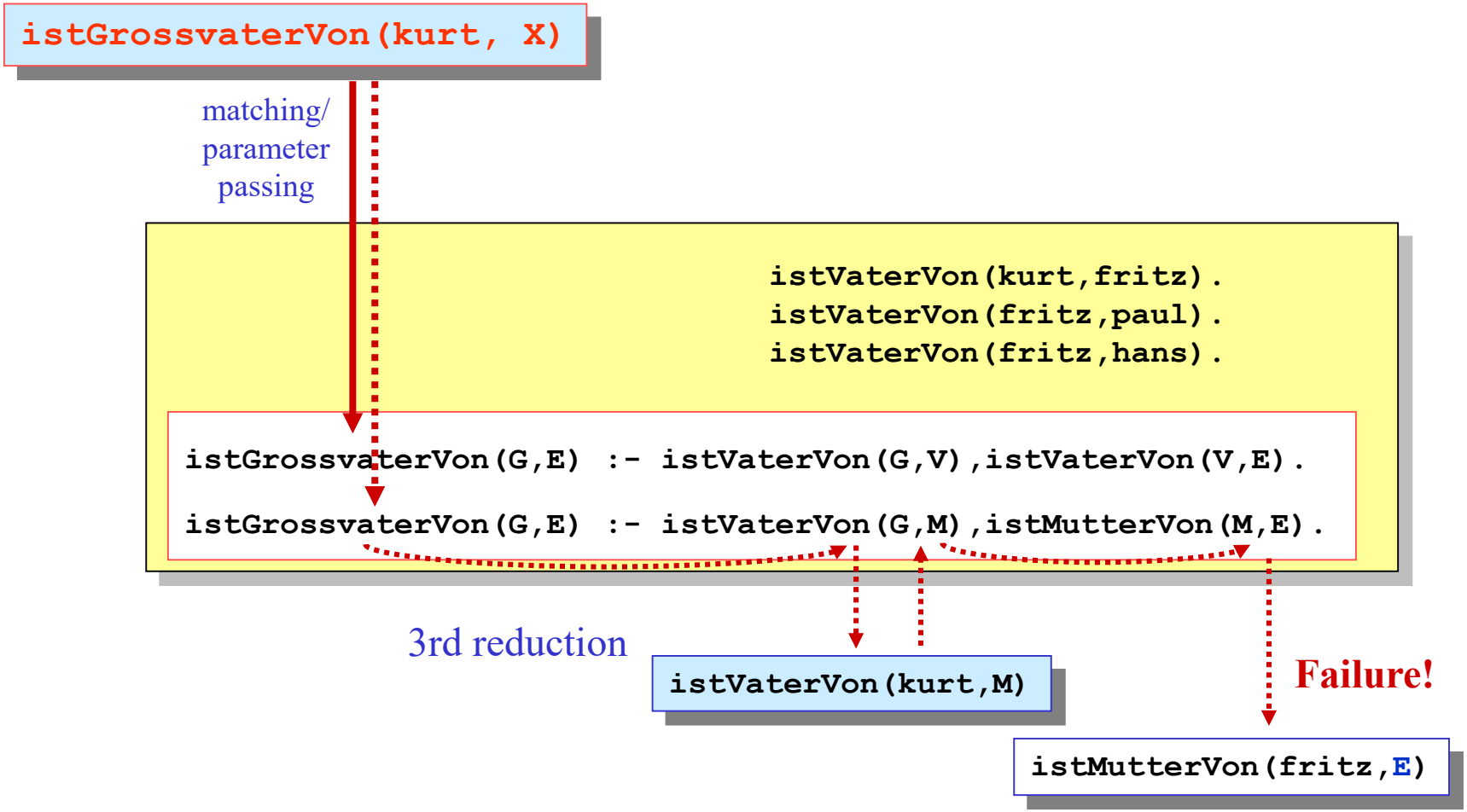
Operationalisation in Prolog (5)

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).
```

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

X = paul:
X = hans:

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

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:

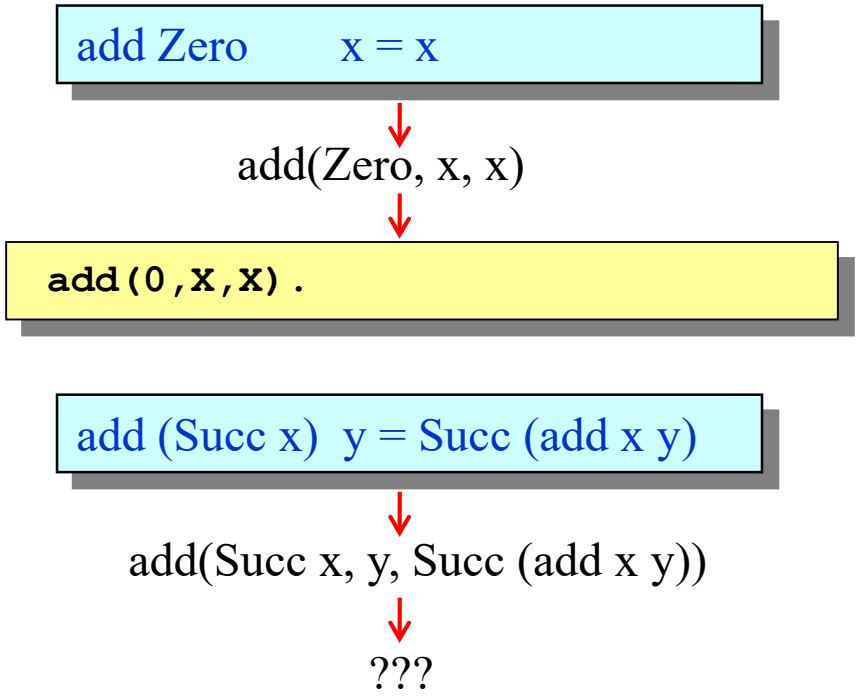
```
data Nat = Zero | Succ Nat

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:



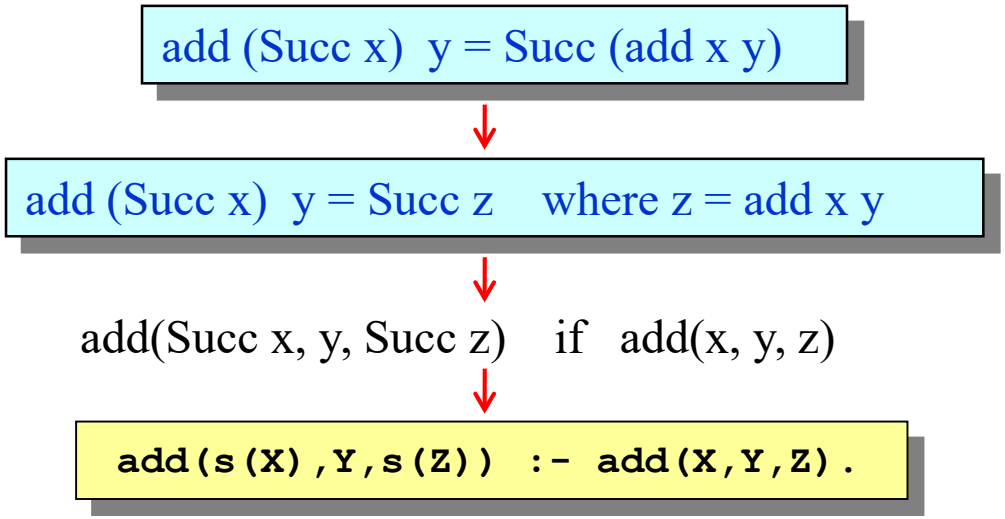
Systematic connection/derivation?

- Essential difference Haskell/Prolog:

  Functions vs. Predicates/Relations

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

- Systematically avoiding nested function calls:



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

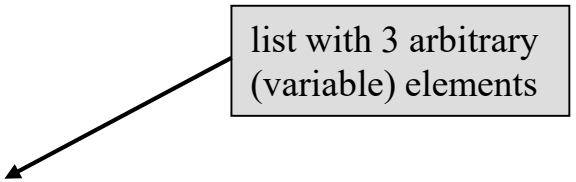
Computing the length of a list in Haskell:

```
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”