

Programming Paradigms

Summer Term 2019

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).
```

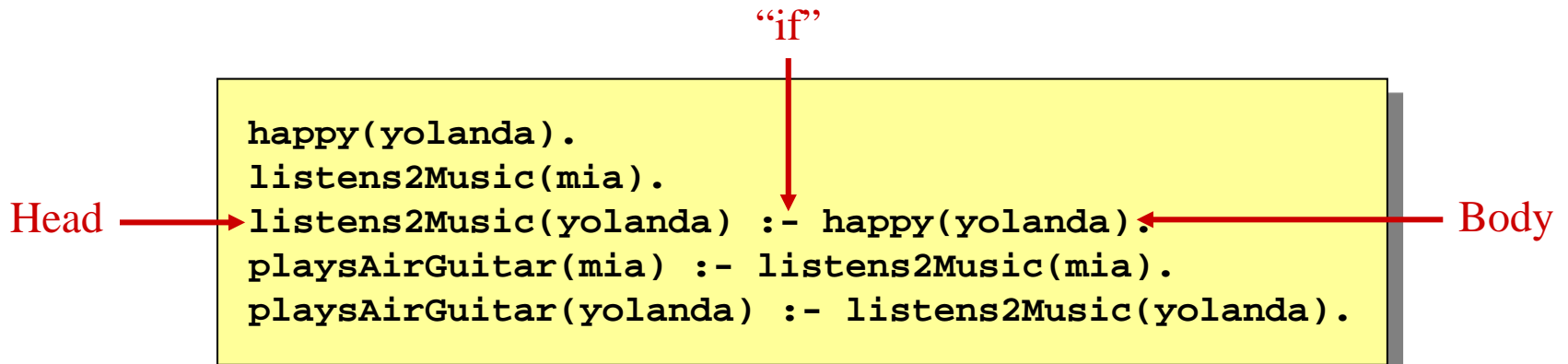
- Queries:

```
?- 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

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

“and”

Alternatives →

- Queries:

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

- Alternative notation:

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

“or”

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)

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

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

- 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.
```

important that at end

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) \equiv
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

`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)`

Operationalisation in Prolog (2)

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

```
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)
```

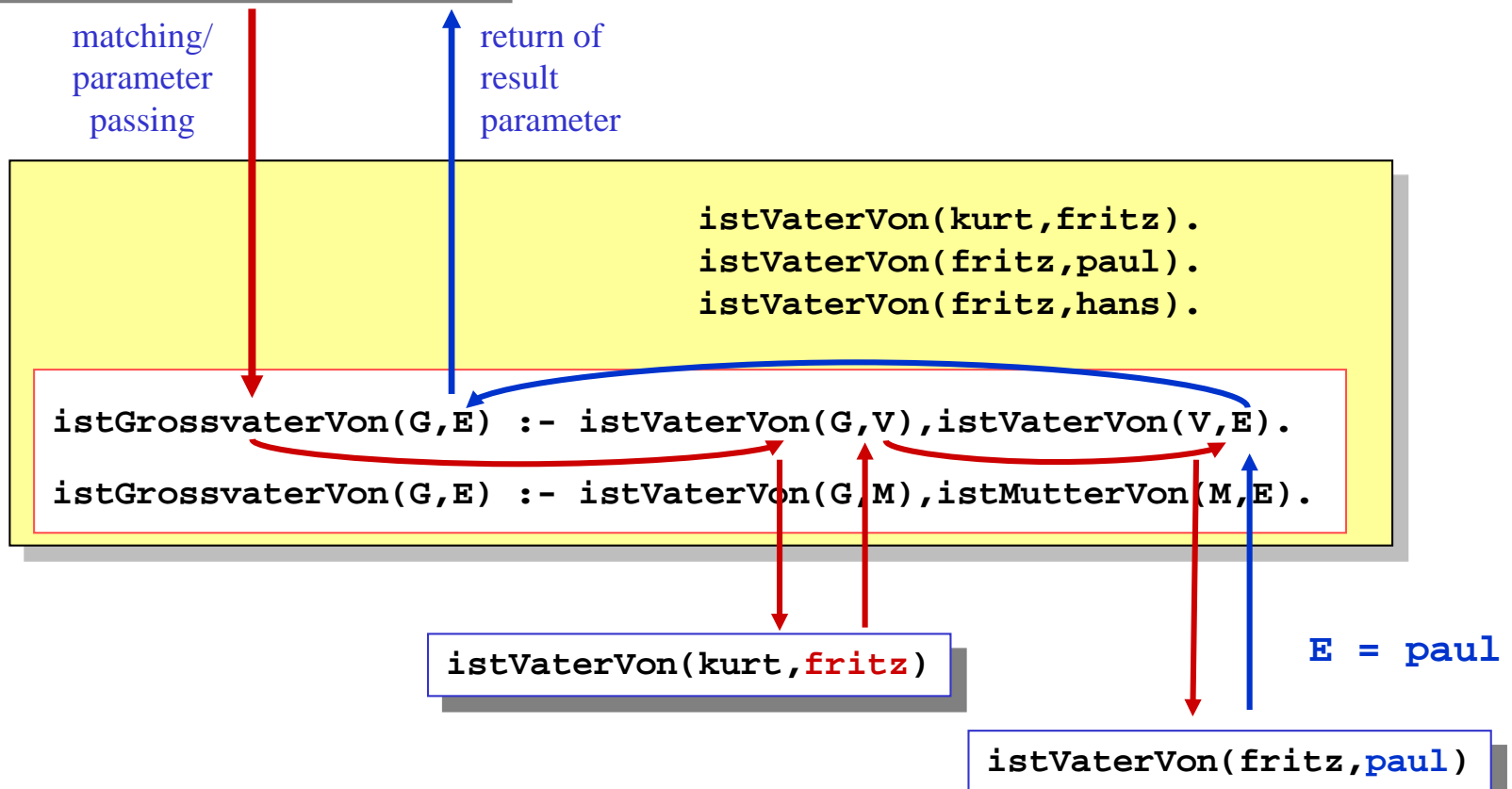
```
istVaterVon(fritz, E)
```

2nd reduction

Operationalisation in Prolog (3)

Principle: reduction to subproblems

`istGrossvaterVon(kurt, X)`



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

```
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)
```

Operationalisation in Prolog (6)

The **backtracking** also concerns further 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).
```

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).
```

X = paul:

X = hans:

```
?- 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).
```

X = paul:

X = hans:

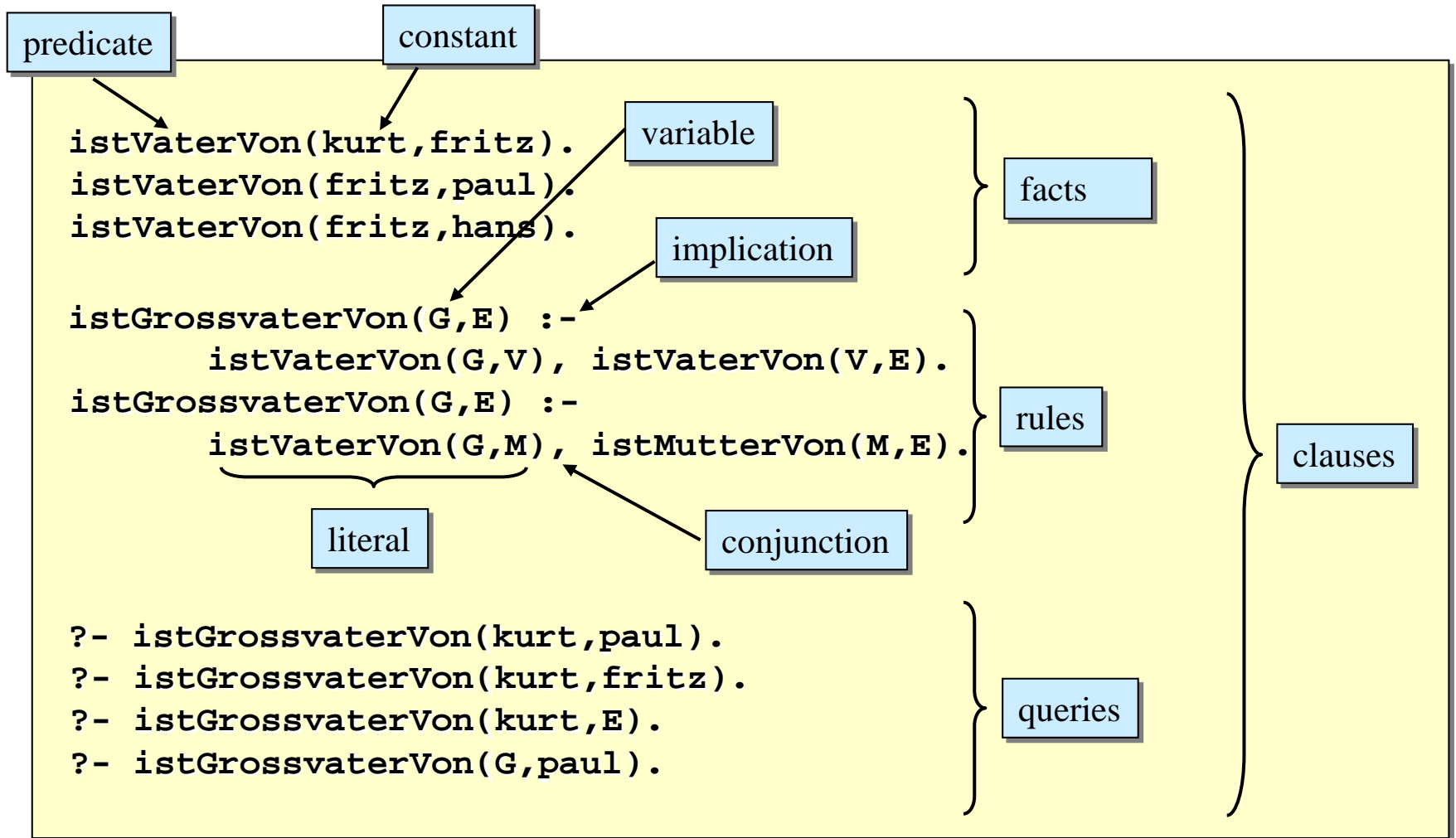
```
?- istGrossvaterVon(kurt, X).
?- istVaterVon(kurt, V), istVaterVon(V, X).
?- istVaterVon(fritz, X).
?- .
?- .
?- istVaterVon(kurt, M), istMutterVon(M, X).
?- istMutterVon(fritz, X).
```

Failure!

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

Programming Paradigms

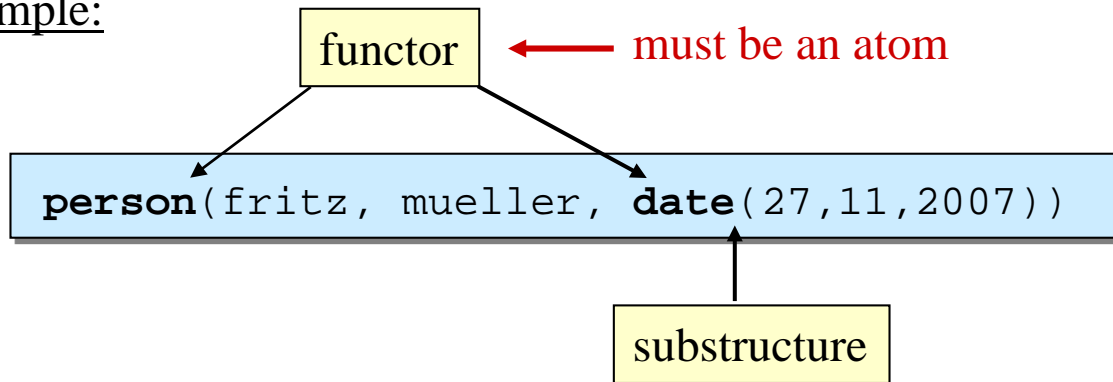
Syntactical ingredients of Prolog



- 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!

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))`

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:

add Zero $x = x$

add(Zero, x, x)

add(0, x, x).

add (Succ x) $y = \text{Succ (add x y)}$

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”

$p(x, y, z).$

- Systematically avoiding nested function calls:

`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

```
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,0).
```

???

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
```

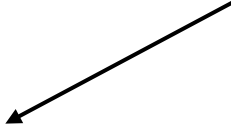
Computing the length of a list in Prolog:

```
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]
```

list with 3 arbitrary
(variable) elements



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



```
quicksort [] = []  
quicksort (h : t) = list  
  where (l1, l2) = partition h t  
        ls = quicksort l1  
        lg = quicksort l2  
        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 (these 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: $\{0, s(0), s(s(0)), s(s(s(0))), \dots\}$ (without predicate symbols!)

the Herbrand base: $\{add(0, 0, 0), add(0, 0, s(0)), add(0, s(0), 0), \dots\}$

(**all** applications of predicate symbols on terms from Herbrand universe)

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, \dots, L_n exist in I such that $L_0 :- L_1, L_2, \dots, L_n$ 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

$$\emptyset, T_P(\emptyset), T_P(T_P(\emptyset)), T_P(T_P(T_P(\emptyset))), \dots$$

Smallest Herbrand model

On the example:

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

$$T_P(\emptyset) = \{\text{add}(0,0,0), \text{add}(0,s(0),s(0)), \text{add}(0,s(s(0)),s(s(0))), \dots\}$$

$$T_P(T_P(\emptyset)) = T_P(\emptyset) \cup \{\text{add}(s(0),0,s(0)), \text{add}(s(0),s(0),s(s(0))), \text{add}(s(0),s(s(0)),s(s(s(0))))), \dots\}$$

$$T_P(T_P(T_P(\emptyset))) = T_P(T_P(\emptyset)) \cup \{\text{add}(s(s(0)),0,s(s(0))), \text{add}(s(s(0)),s(0),s(s(s(0))))), \text{add}(s(s(0)),s(s(0)),s(s(s(s(0))))), \dots\}$$

...

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

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 $\backslash=$, 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).
```

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

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

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).
```

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

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

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

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

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(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!

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:

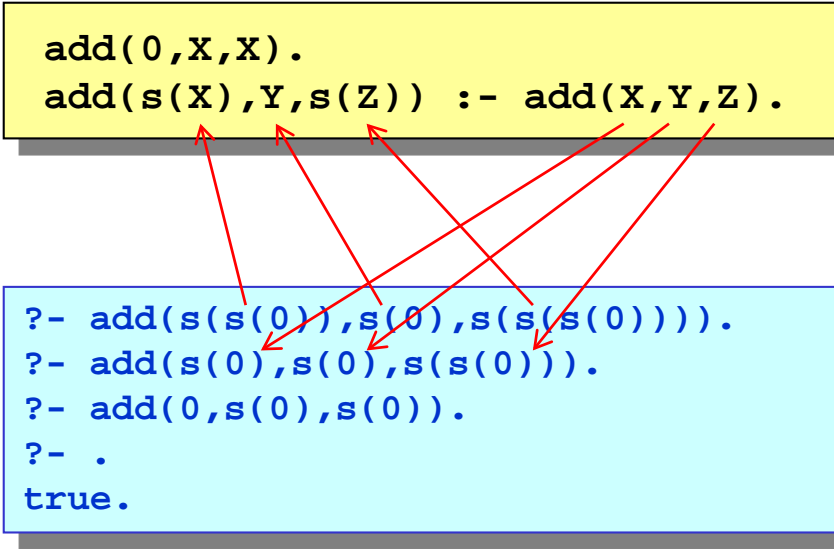
1. Unification
2. Resolution
3. Derivation trees

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”?

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

?



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

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:

<code>europa = europa ?</code>	<code>yes</code>
<code>person(fritz,mueller) = person(fritz,mueller) ?</code>	<code>yes</code>
<code>person(fritz,mueller) = person(mueller,fritz) ?</code>	<code>no</code>
<code>5 = 2 ?</code>	<code>no</code>
<code>5 = 2 + 3 ?</code>	<code>no</code>
<code>2 + 3 = +(2, 3) ?</code>	<code>yes</code>

⇒ 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(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) ?  
...
```

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:
$$U = \{\text{Lastname} / \text{mueller}, \text{MM} / 11\}$$
- This substitution U changes only the variables **Lastname** and **MM** (in context), everything else stays unchanged.
- $U(\text{person}(\text{fritz}, \text{Lastname}, \text{datum}(27, 11, 2007)))$
 $\quad == \text{person}(\text{fritz}, \text{mueller}, \text{datum}(27, 11, 2007))$

Unification concepts, somewhat formally (2)

- Unifier:

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

$$\begin{aligned} &U(\text{person}(\text{fritz}, \text{Lastname}, \text{date}(27, 11, 2007))) \\ &== U(\text{person}(\text{fritz}, \text{mueller}, \text{date}(27, \text{MM}, 2007))) \end{aligned}$$

- Most general unifier:

- unifier that leaves as many variables as possible unchanged, and does not introduce specific terms where variables suffice
- Example: $\text{date}(\text{DD}, \text{MM}, 2007)$ and $\text{date}(\text{D}, 11, \text{Y})$

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

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

- Prolog always looks for a most general unifier.

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 $U = \emptyset$
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”
←

Algorithm (cont.):

4. If $T_1 = f(T_{1,1}, \dots, T_{1,n})$ and $T_2 = f(T_{2,1}, \dots, T_{2,n})$ 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
 $U_{n-1}(\dots(U_1(T_{1,n})\dots))$ and $U_{n-1}(\dots(U_1(T_{2,n}))\dots)$

If all these unifiers exist, then

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

5. Otherwise: T_1 and T_2 are not unifiable.

`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 $U_1 = \emptyset$
2. Find a most general unifier U_2 for $U_1(\mathbf{4})$ and $U_1(\mathbf{4})$
 \Rightarrow same constants, thus $U_2 = \emptyset$
3. Find a most general unifier U_3 for $U_2(U_1(\mathbf{1985}))$ and $U_2(U_1(\mathbf{Year}))$
 \Rightarrow constant vs. variable, thus $U_3 = \{\mathbf{Year}/\mathbf{1985}\}$

A most general unifier overall is:

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

`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 $U_1 = \{x/\text{marcellus}\}$
2. Find a most general unifier U_2 for $U_1(\text{mia}) = \text{mia}$ and $U_1(x) = \text{marcellus}$
 \Rightarrow **different** constants, hence U_2 does not exist!

Consequently, also no unifier exists for the original terms!

$$d(\mathbf{E}, g(\mathbf{H}, \mathbf{J})) = d(\mathbf{F}, g(\mathbf{H}, \mathbf{E})) \quad ?$$

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

1. Find a most general unifier U_1 for \mathbf{E} and \mathbf{F}
 \Rightarrow different variables, thus $U_1 = \{\mathbf{E}/\mathbf{F}\}$
2. Find a most general unifier U_2 for $U_1(g(\mathbf{H}, \mathbf{J}))$ and $U_1(g(\mathbf{H}, \mathbf{E}))$

$$g(\mathbf{H}, \mathbf{J}) = g(\mathbf{H}, \mathbf{F}) \quad ?$$

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

- Find a most general unifier $U_{2,1}$ for \mathbf{H} and \mathbf{H}
 \Rightarrow same variables, thus $U_{2,1} = \emptyset$
- Find a most general unifier $U_{2,2}$ for $U_{2,1}(\mathbf{J})$ and $U_{2,1}(\mathbf{F})$
 \Rightarrow different variables, thus $U_{2,2} = \{\mathbf{F}/\mathbf{J}\}$

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

A most general unifier overall is:

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

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:

$$x = q(x) ?$$

⇒ No unifier exists.

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

Relevance of the “occurs check”

Without “occurs check”:

$$p(\mathbf{x}) = p(q(\mathbf{x}))?$$

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

1. Find a most general unifier U_I for \mathbf{x} and $q(\mathbf{x})$
 \Rightarrow variable vs. term, thus $U_I = \{\mathbf{x}/q(\mathbf{x})\}$

$$U = U_I = \{\mathbf{x}/q(\mathbf{x})\} !$$

Although it actually is not true that $U(p(\mathbf{x}))$ and $U(p(q(\mathbf{x})))$ 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

$?- P, L_1, L_2, \dots, L_n, Q.$

provided that $L :- L_1, L_2, \dots, L_n.$ 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

$?- U(P), U(L_1), U(L_2), \dots, U(L_n), U(Q).$

provided that:

- there is a program clause $L_0 :- L_1, L_2, \dots, L_n.$ (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(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
```

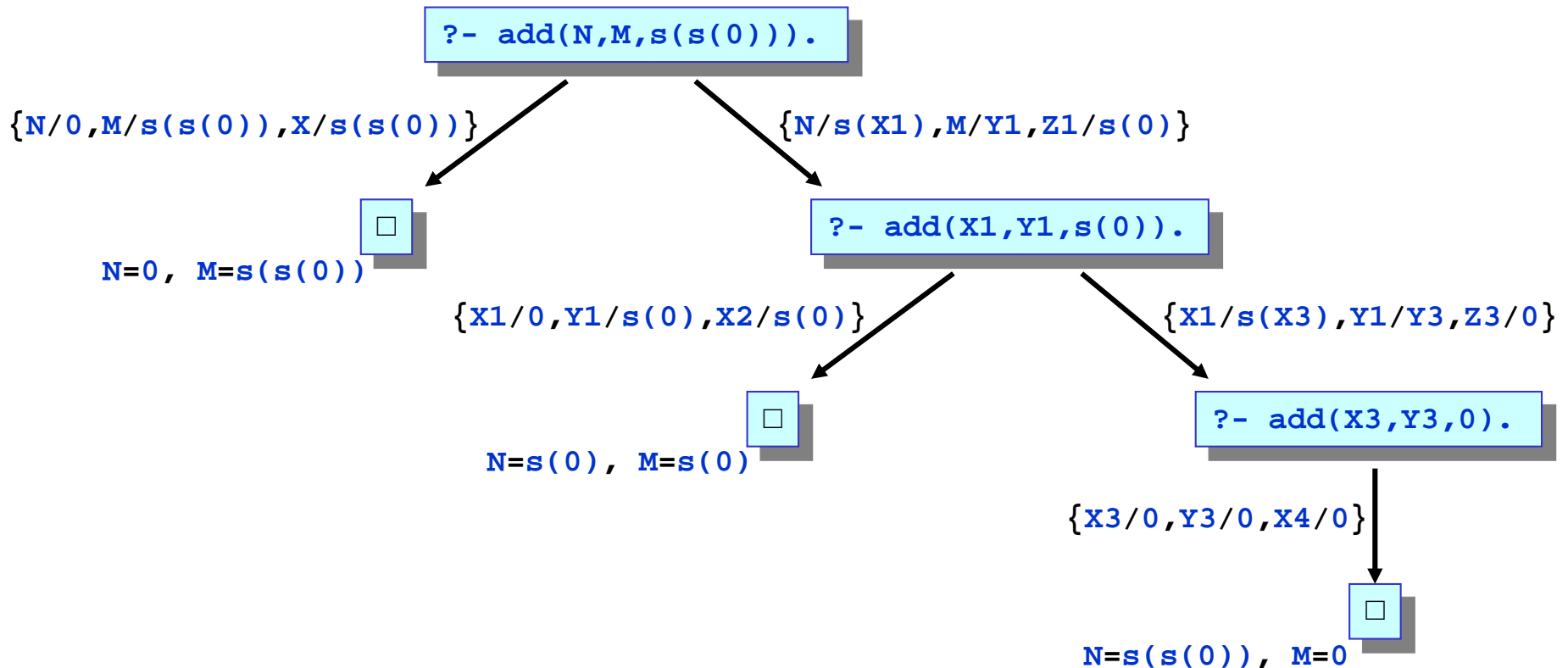
otherwise infinite search

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:



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 – 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).
```

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

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

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

```
{X/0,Y/_1,U/0}
```

```
?- add(0,_1,s(0)).
```

```
{_1/s(0),X1/s(0)}
```



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

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

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

```
{X2/0,Y2/_2,U2/0}
```

```
?- add(0,_2,Z2),add(Z2,_2,s(0)).
```

```
{X2/s(X3),Y2/Y3,U2/Z3}
```

```
?- ...
```

Grows ever longer!

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).
```

```
?- 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).
```


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).
```

```
?- 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).
```

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).
```

?- 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}

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

{X6/0,X7/0}

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

Does not look good!

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

```
?- 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

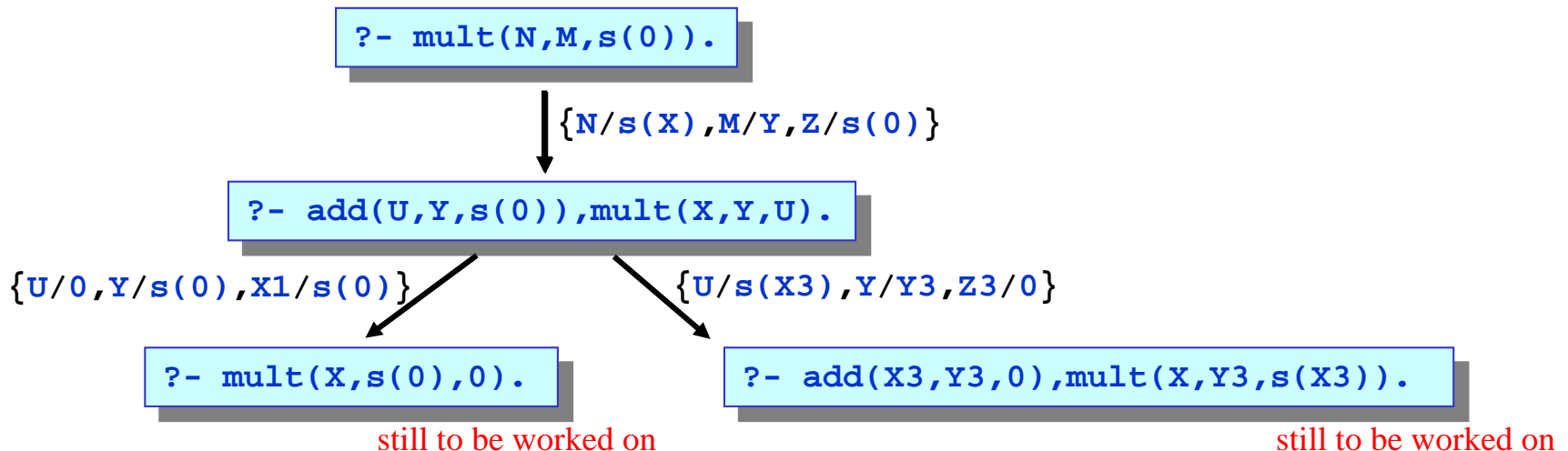
Detailed description of the generation of derivation trees

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



```
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).
```



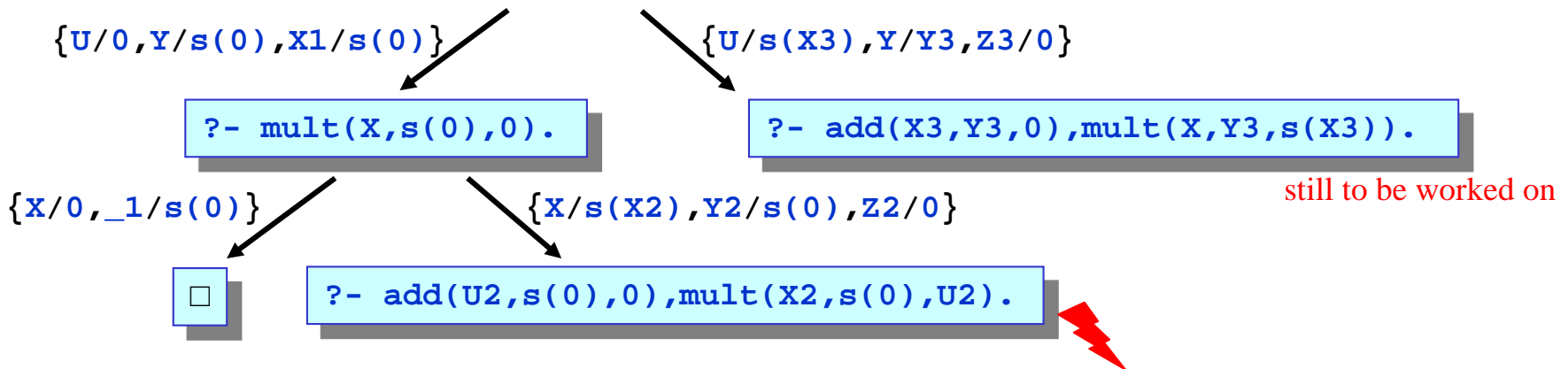
Detailed description of the generation of derivation trees

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 ()

```
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

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)

```
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(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),  
M=s(0)
```



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



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).
```

?- 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}

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

{X6/0,X7/0}

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

Does not look good!

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).
```

```
?- 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)}

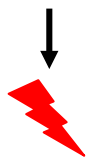
{X/s(X2),Y2/s(0),Z2/0}

{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!

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).
    
```

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

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

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

Does not look good!

```
{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 all call modes
work well, except
`mult(?X,?Y,?Z)!`

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