

# Deskriptive Programmierung

## Prolog-Grundlagen/Syntax

## Prolog im einfachsten Fall: Fakten und Anfragen

- Eine Art Datenbank mit einer Reihe von Fakten:

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

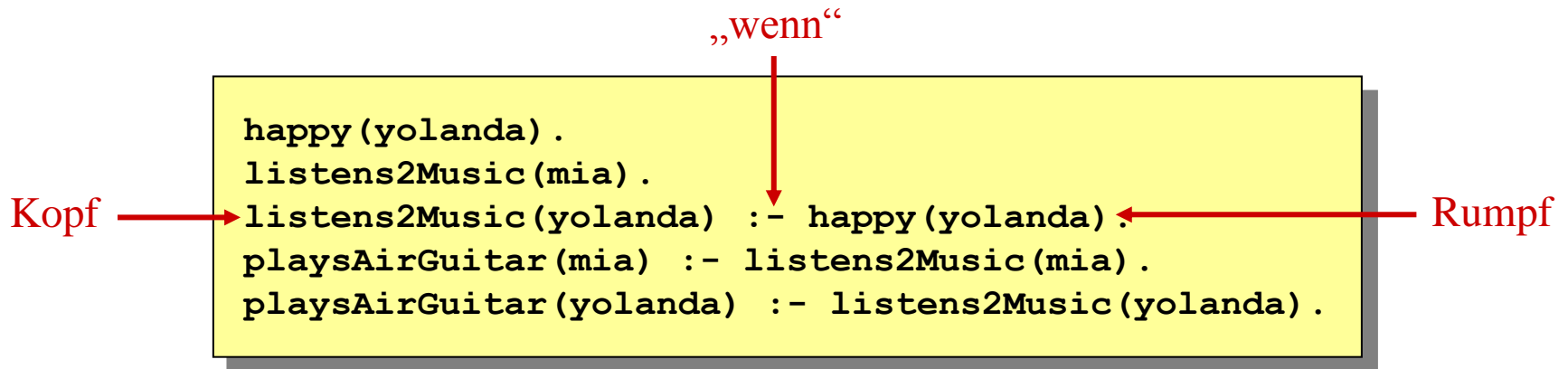
- Anfragen:

```
?- woman(mia) .  
true.  
  
?- playsAirGuitar(jody) .  
true.  
  
?- playsAirGuitar(mia) .  
false.  
  
?- playsAirGuitar(vincent) .  
false.  
  
?- playsPiano(jody) .  
false.
```

← Punkt wichtig!

← oder Fehlermeldung

## Fakten + einfache Implikationen



- Anfragen:

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

wegen:

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

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

„und“

Alternativen →

- Anfragen:

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

- alternative Schreibweise:

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

„oder“

```
woman(mia) .  
woman(jody) .  
woman(yolanda) .  
  
loves(vincent,mia) .  
loves(marsellus,mia) .  
loves(mia,vincent) .  
loves(vincent,vincent) .
```

mehrstelliges Prädikat

- Anfragen:

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

vom Nutzer einzugeben

## Variablen in Regeln (nicht nur in Anfragen)

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

- Anfragen:

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

anonyme Variable

## Variablen in Regeln (nicht nur in Anfragen)

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

- Anfragen:

```
?- 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.
```



wichtig dass am Ende

## Einige Beobachtungen zu Variablen

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

- Variablen in Regeln und Anfragen sind unabhängig voneinander.

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

- Innerhalb einer Regel oder Anfrage stehen gleiche Variablen für gleiche Objekte.
- Aber verschiedene Variablen stehen nicht notwendigerweise für verschiedene Objekte.
- Es sind auch mehrfache Vorkommen der gleichen Variable im Kopf möglich!
- In Regeln können im Rumpf Variablen auftauchen, die nicht im Kopf vorkommen!



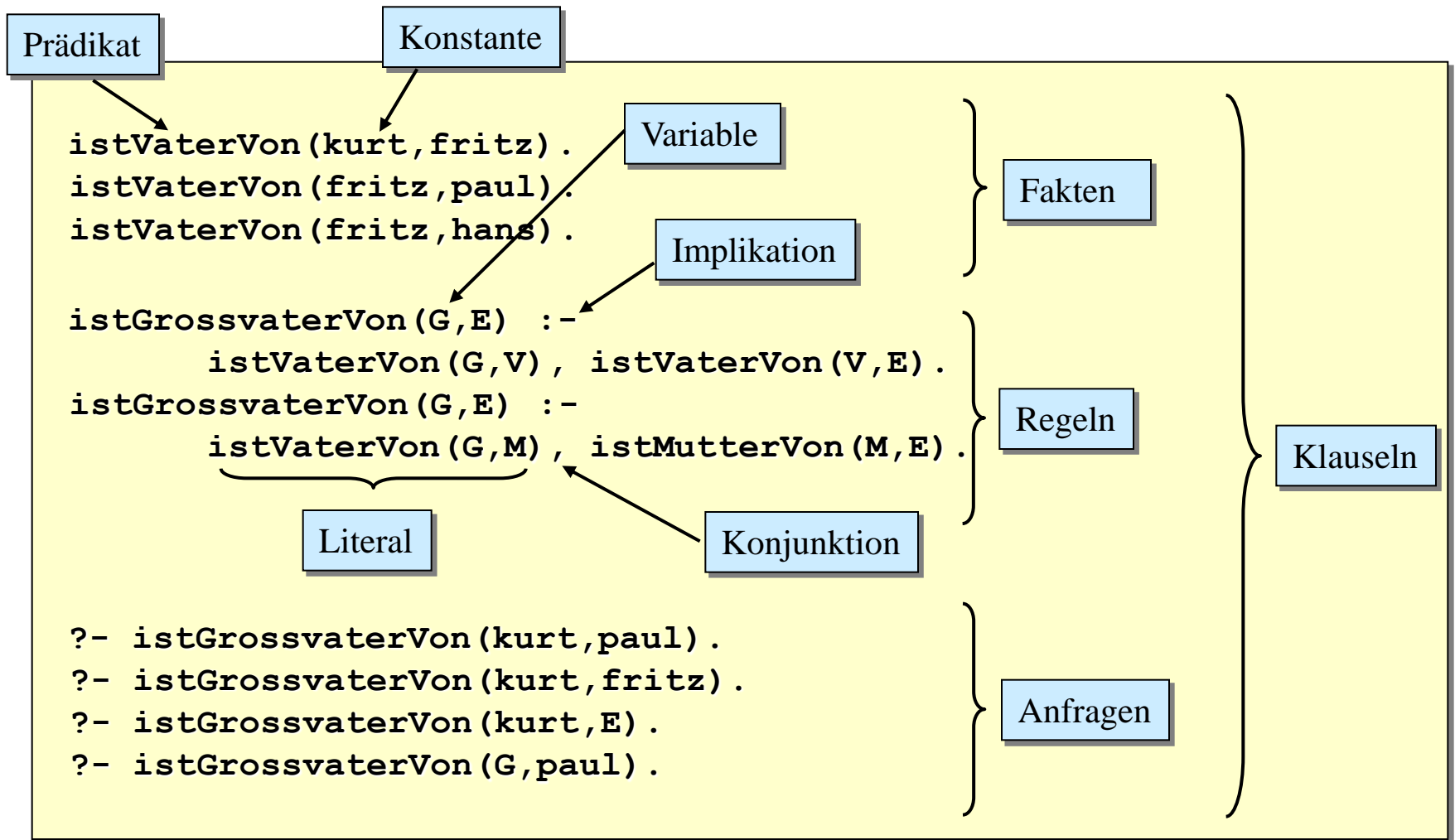
```
loves(vincent,mia) .  
loves(marsellus,mia) .  
loves(mia,vincent) .  
  
jealous(X,Y) :- loves(X,Z) , loves(Y,Z) , X \= Y.
```

- Was ist die „logische“ Interpretation von **Z** oben? (bzw. der gesamten Regel?)
- Denkbar, für beliebige (aber feste) **X** , **Y**:  
    wenn für jede Wahl von **Z** gilt: **loves(X,Z)** , und **loves(Y,Z)** , und **X \= Y**,  
    dann gilt auch: **jealous(X,Y)**
- Oder, für beliebige (aber feste) **X** , **Y**:  
    für jede Wahl von **Z** gilt: wenn **loves(X,Z)** , und **loves(Y,Z)** , und **X \= Y**,  
    dann gilt auch: **jealous(X,Y)**

???

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

- Was ist die „logische“ Interpretation von **Z** oben? (bzw. der gesamten Regel?)
- Oder, für beliebige (aber feste) **X** , **Y**:  
für jede Wahl von **Z** gilt: wenn **loves(X,Z)** , und **loves(Y,Z)** , und **X \= Y** ,  
dann gilt auch: **jealous(X,Y)**
- Logisch äquivalent, für beliebige (aber feste) **X** , **Y**:  
wenn für irgendeine Wahl von **Z** gilt: **loves(X,Z)** , und **loves(Y,Z)** , und **X \= Y** ,  
dann gilt auch: **jealous(X,Y)**



- Zum Aufbau von Klauseln verwendet Prolog verschiedene Objekte.
- Diese unterteilen sich in:
  - **Konstanten** ( Zahlen, Zeichenfolgen, ...)
  - **Variablen** ( X,Y, InLand, ...)
  - **Operatortermine** ( ... 1 + 3 \* 4 ...)
  - **Strukturen** ( datum(27,11,2007), person(fritz, mueller), ...  
zusammengesetzt, rekursiv, „unendlich“, ...)
- Achtung: Prolog hat kein Typsystem

## Konstanten in Prolog

- **Zahlen**

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

- **Atome**, d.h. Zeichenfolgen, die einer der folgenden drei Regeln genügen:

1. Die Zeichenfolge beginnt mit einem Kleinbuchstaben, gefolgt von beliebig vielen Klein- und Großbuchstaben, Ziffern und Unterstrichen '\_'.
2. Die Zeichenfolge beginnt und endet mit einem Apostroph ( ' ). Dazwischen können beliebige Zeichen stehen. Soll ein Apostroph selbst in der Zeichenkette vorkommen, muss er doppelt angegeben werden.
3. Die Zeichenfolge besteht nur aus Sonderzeichen.

richtig:	<b>fritz</b>	<b>new_york</b>	<b>:-</b>	<b>--&gt;</b>	<b>'I don''t know!'</b>
falsch:	<b>Fritz</b>	<b>new-york</b>	<b>_xyz</b>	<b>123</b>	

## Variablen in Prolog

- **Variablen:**
  - Name beginnt mit einem Großbuchstaben oder einem Unterstrich '\_'.

- Beispiele: `Land Jahr M V _45 _G107 _europa`

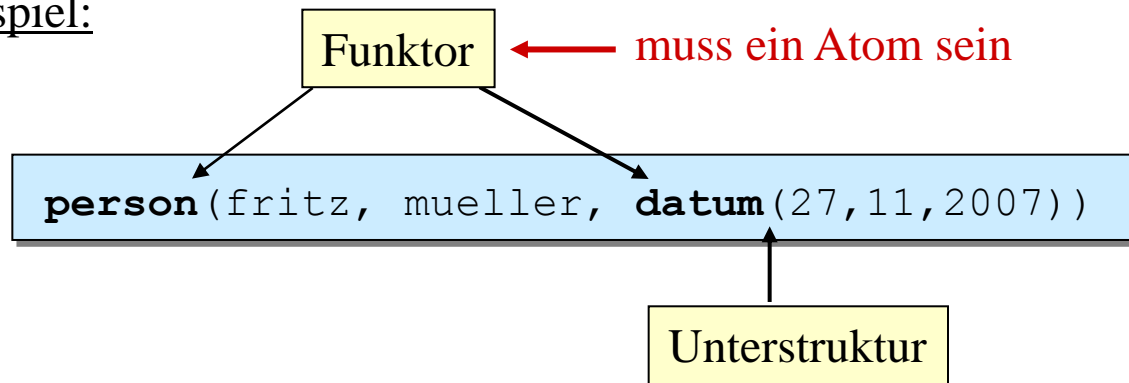
internes Format  
für Variablen

- Anonyme Variablen (Darstellung mit lediglich '\_'):
  - wenn das Objekt nicht interessiert:

`?- istVaterVon(_,fritz).`

## Strukturen in Prolog

- **Strukturen** repräsentieren Objekte, die aus mehreren anderen Objekten zusammengesetzt sind.
- Beispiel:



Funktoren: `person/3`, `datum/3`

- Dadurch Modellierung von algebraischen Datentypen – aber keine Typisierung. Somit wäre `person(1,2,'a')` auch eine legale Struktur.
- Beliebige **Schachtelungstiefe** ist erlaubt – im Prinzip unendlich.

### Vordefinierte Syntax für spezielle Strukturen:

- Es gibt einen vordefinierten „Listentyp“ als rekursive Datenstruktur:

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

- Zeichenketten werden als Listen von ASCII-Codes dargestellt:

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

### Operatoren:

- Operatoren sind Funktoren in Operatorschreibweise.
- Beispiel: arithmetische Ausdrücke
  - Mathematische Funktionen sind als Operatoren definiert.

• `1 + 3 * 4` ist als folgende Struktur zu sehen: `+ (1, * (3, 4))`



## Einfaches Beispiel für Arbeit mit Datenstrukturen

```
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.
```

- Zur Erinnerung, in Haskell:

```
data Nat = Zero | Succ Nat  
  
add :: Nat → Nat → Nat  
add Zero    m = m  
add (Succ n) m = Succ (add n m)
```

## Systematischer Zusammenhang/Herleitung?

- Wesentlicher Unterschied Haskell/Prolog:

Funktionen

vs.

Prädikate/Relationen

$f\ x\ y = z$

„entspricht“

$p(x, y, z)$

- Zunächst etwas naiver Versuch, diesen Zusammenhang auszunutzen:

add Zero m = m

add(Zero,m,m)

add(0,x,x) .

add (Succ n) m = Succ (add n m)

add(Succ n,m,Succ (add n m))

???

## Systematischer Zusammenhang/Herleitung?

- Wesentlicher Unterschied Haskell/Prolog:

Funktionen

vs.

Prädikate/Relationen

$f\ x\ y = z$

„entspricht“

$p(x, y, z) .$

- Systematische Vermeidung verschachtelter Aufrufe:

`add (Succ n) m = Succ (add n m)`



`add (Succ n) m = Succ m' where m' = add n m`



`add(Succ n,m,Succ m')` wenn `add(n,m,m')`



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

## Zur Flexibilität von Prolog-Prädikaten

```
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) .
```

???

## Zur Flexibilität von Prolog-Prädikaten

```
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.
```

## Ein weiteres Beispiel

Länge einer Liste in Haskell:

```
length []      = 0
length (x:xs)  = length xs + 1
```

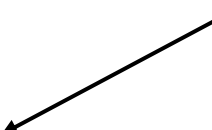
Länge einer Liste in Prolog:

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

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

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

Liste mit 3 beliebigen  
(variablen) Elementen



Vorsicht: wenn statt:

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

Verwendung von:

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

dann:

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

```
?- length(Liste,3).  
    false.
```

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

## Ein Beispiel entsprechend mehrerer verschachtelter Aufrufe

```
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).
```

