
IV. Logische Programmierung

- 1. Grundkonzepte der logischen Programmierung
- 2. Syntax von Prolog
- 3. Rechnen in Prolog

Unifikation

`add(X, zero, X) .`

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

?- `add(succ(zero), succ(zero), U) .`

Unifikation

Existiert eine Substitution σ von Variablen mit Termen, so dass
 $\sigma(\text{Anfrage}) = \sigma(\text{Faktum})$ oder $\sigma(\text{Anfrage}) = \sigma(\text{Klauselkopf})$?

$\sigma(\text{add}(\text{succ}(\text{zero}), \text{succ}(\text{zero}), \text{U})) = \sigma(\text{add}(\text{X}, \text{zero}, \text{X}))$?

false

$\sigma(\text{add}(\text{succ}(\text{zero}), \text{succ}(\text{zero}), \text{U})) = \sigma(\text{add}(\text{X}, \text{succ}(\text{Y}), \text{succ}(\text{Z})))$?

$\sigma = \{X = \text{succ}(\text{zero}), Y = \text{zero}, U = \text{succ}(\text{Z})\}$

Unifikation

- **Substitution** σ : Abbildung von Variablen auf Terme
- s und t sind **unifizierbar**, falls es Substitution gibt mit $\sigma(s) = \sigma(t)$
- σ heißt dann **Unifikator** von s und t

$\sigma(\text{add}(\text{succ}(\text{zero}), \text{succ}(\text{zero}), U)) = \sigma(\text{add}(X, \text{zero}, X))$?

false

$\sigma(\text{add}(\text{succ}(\text{zero}), \text{succ}(\text{zero}), U)) = \sigma(\text{add}(X, \text{succ}(Y), \text{succ}(Z)))$?

$\sigma = \{X = \text{succ}(\text{zero}), Y = \text{zero}, U = \text{succ}(Z)\}$

Unifikation

$$\sigma(\text{add}(\text{succ}(\text{zero}), \text{succ}(\text{zero}), U)) = \sigma(\text{add}(X, \text{succ}(Y), \text{succ}(Z)))$$

Allgemeinster Unifikator (MGU)

$$\sigma_1 = \{X = \text{succ}(\text{zero}), Y = \text{zero}, U = \text{succ}(Z)\}$$

$$\sigma_2 = \{X = \text{succ}(\text{zero}), Y = \text{zero}, U = \text{succ}(\text{zero}), Z = \text{zero}\}$$

$$\sigma_3 = \{X = \text{succ}(\text{zero}), Y = \text{zero}, U = \text{succ}(\text{succ}(W)), Z = \text{succ}(W)\}$$

Unifikator μ ist MGU wenn alle anderen Unifikatoren σ aus μ entstehen, indem man Terme für seine Variablen einsetzt.

Formal:

Für alle anderen Unifikatoren σ muss eine Substitution τ existieren mit $\sigma = \tau \circ \mu$.

Berechnung des MGU

Eingabe: s und t

Ausgabe: MGU oder Fehlschlag

1. Falls s und t gleiche Variablen sind, dann $\sigma = \{ \}$.
2. Falls s Variable ist und t enthält s nicht, dann $\sigma = \{s = t\}$.
3. Falls t Variable ist und s enthält t nicht, dann $\sigma = \{t = s\}$.
4. Falls $s = f(s_1, \dots, s_n)$ und $t = f(t_1, \dots, t_n)$, dann:
 - a) Sei $\sigma_1 = \text{MGU}(s_1, t_1)$.
 - b) Für alle $2 \leq i \leq n$ sei
$$\sigma_i = \text{MGU}(\sigma_{i-1}(\dots(\sigma_1(s_i))\dots), \sigma_{i-1}(\dots(\sigma_1(t_i))\dots)).$$
 - c) Falls alle σ_i existieren, dann $\sigma = \sigma_n \circ \dots \circ \sigma_1$.
5. Sonst sind s und t nicht unifizierbar.

Beweisverfahren von Prolog: Resolution

Algorithmus SOLVE

Eingabe: Anfrage $?- G_1, \dots, G_m$

Ausgabe: Antwortsubstitution σ oder Fehlschlag

1. Wenn $m = 0$, dann terminiere mit $\sigma = \{ \}$.
2. Sonst: Suche nach der nächsten Programmklausel
 $H :- B_1, \dots, B_n.$
so dass G_1 und H unifizierbar (mit MGU μ) sind.
Gibt es keine, dann terminiere mit Fehlschlag.
3. Rufe SOLVE mit der folgenden Anfrage auf:
 $?- \mu(B_1), \dots, \mu(B_n), \mu(G_2), \dots, \mu(G_m).$
4. Falls dieser Aufruf Antwortsubst. τ berechnet,
dann: Terminiere mit $\sigma = \tau \circ \mu.$
sonst: Gehe zurück zu Schritt 2.

Beweisbaum

`add(X, zero, X) .`

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

`?- add(succ(zero), succ(zero), U) .`

Resolution mit:

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

`X = succ(zero)
Y = zero
U = succ(Z)`

`?- add(succ(zero), zero, Z) .`

Resolution mit:
`add(X1, zero, X1) .`

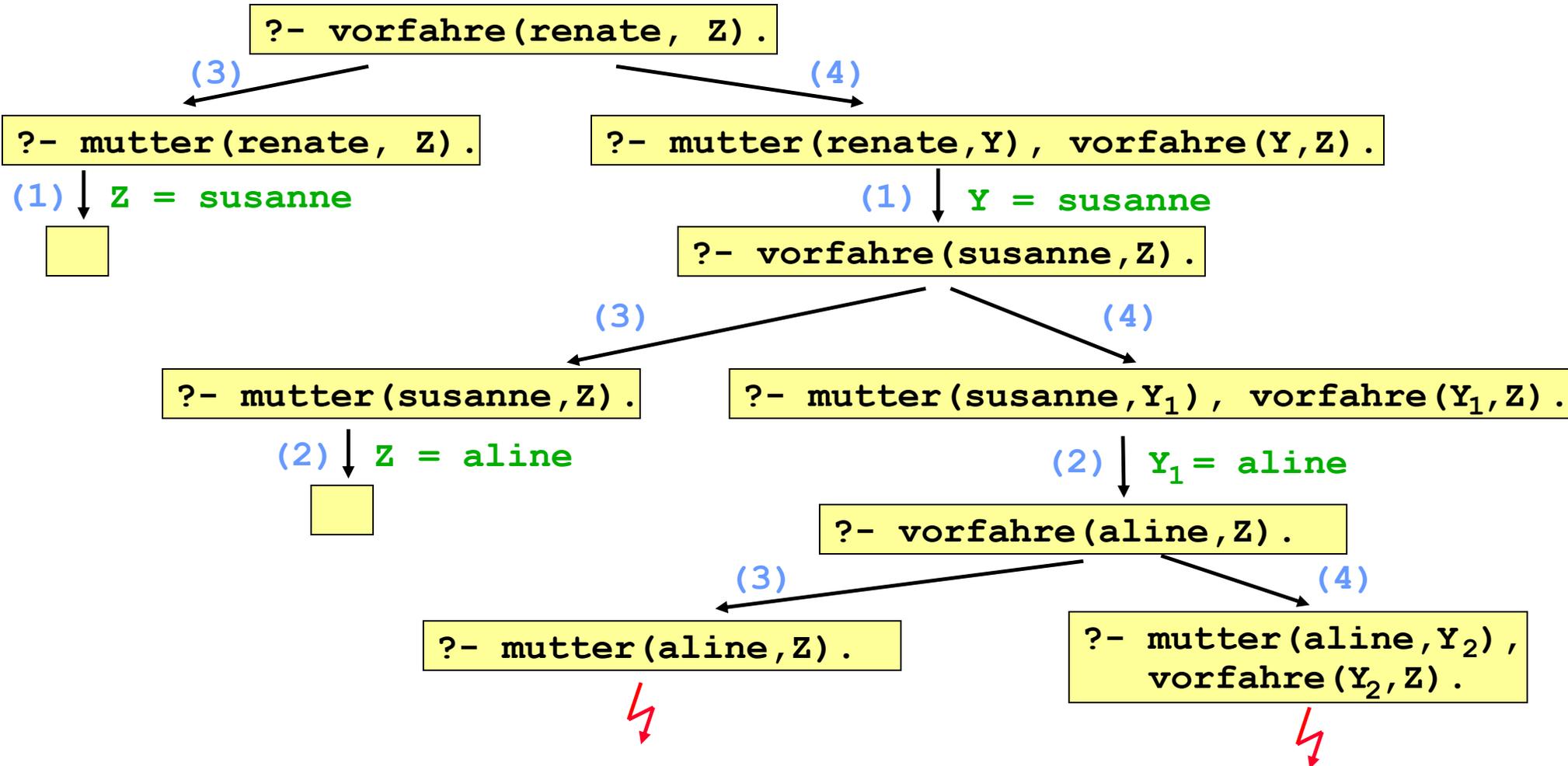
`X1 = succ(zero)
Z = succ(zero)`

*In jedem Reso-
lutionsschritt
werden die Variablen
in der Prog.-Klausel
ggf. umbenannt.*

Antwortsubstitution: `{U = succ(succ(zero))}`

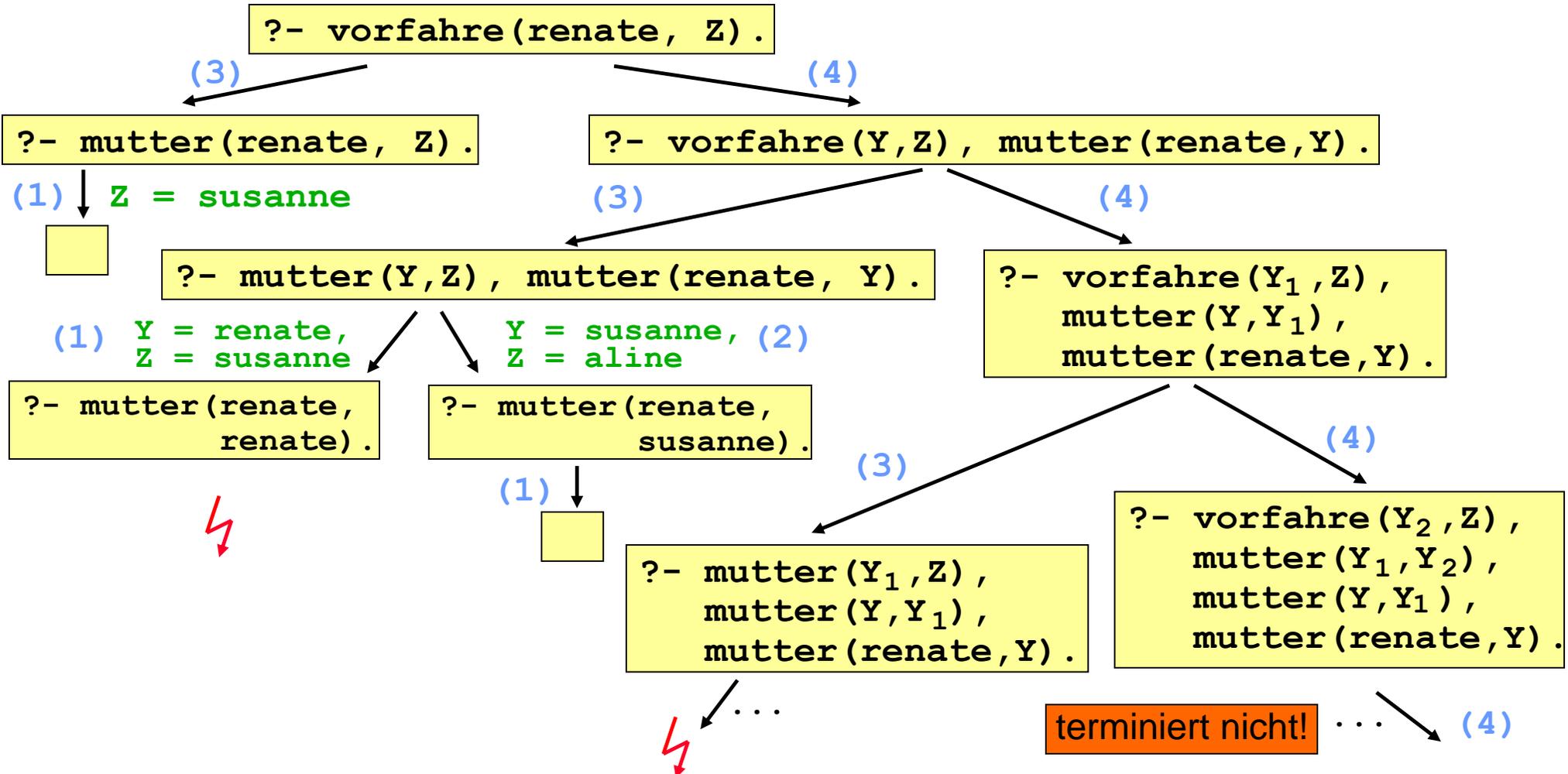
Beweisbaum

(1) `mutter(renate, susanne).` (3) `vorfahre(V, X) :- mutter(V, X).`
(2) `mutter(susanne, aline).` (4) `vorfahre(V, X) :- mutter(V, Y),
vorfahre(Y, X).`



Beweisbaum

(1) `mutter(renate, susanne).` (3) `vorfahre(V,X) :- mutter(V,X).`
 (2) `mutter(susanne, aline).` (4) `vorfahre(V,X) :- vorfahre(Y,X),
 mutter(V,Y).`



Beweisbaum

- (1) `mutter(renate, susanne)` . (3) `vorfahre(V, X) :- vorfahre(Y, X),
mutter(V, Y)` .
(2) `mutter(susanne, aline)` . (4) `vorfahre(V, X) :- mutter(V, X)` .

`?- vorfahre(renate, Z)` .

(3)

`?- vorfahre(Y, Z), mutter(renate, Y)` .

(4)

`?- mutter(renate, Z)` .

(1) $Z = \text{susanne}$

`□`

(3)

`?- vorfahre(Y1, Z),
mutter(Y, Y1),
mutter(renate, Y)` .

(4)

`?- mutter(Y, Z), mutter(renate, Y)` .

(1) $Y = \text{renate}, Z = \text{susanne}$

$Y = \text{susanne}, (2)$
 $Z = \text{aline}$

`?- mutter(renate, renate)` .

`?- mutter(renate, susanne)` .

(3)

(4)

(1)

`□`

`?- vorfahre(Y2, Z),
mutter(Y1, Y2),
mutter(Y, Y1),
mutter(renate, Y)` .

`?- mutter(Y1, Z),
mutter(Y, Y1),
mutter(renate, Y)` .

(3)

... **terminiert nicht!**

... ⚡

Gleichheit in Prolog: =

`gleich(X,X).`

Ist in Prolog vordefiniert, heißt dort "="

`?- succ(X) = succ(succ(Y)).`

`X = succ(Y)`

`?- .(a,L) = [X,b|K].`

`X = a, L = [b|K]`

`?- s = t`

berechnet MGU von s und t

```
mem(X, [Y|_]) :- X = Y.  
mem(X, [_|L]) :- mem(X,L).
```

`?- mem(X, [1,2,3]).`

`X = 1 ;`

`X = 2 ;`

`X = 3`

```
mem(X, [X|_]).  
mem(X, [_|L]) :- mem(X,L).
```

Gleichheit in Prolog: `is`

```
?- X = 2 + 5.
```

```
X = 2 + 5
```

```
?- 7 = 2 + 5.
```

```
false
```

"=" berechnet nur syntaktische Unifikation

Vordefiniertes Prädikat "`is`": `s is t`

- `t` muss vollständig instantiiertes arithmetisches Ausdruck sein
- rechne erst `t` aus
- dann wird das Ergebnis der Auswertung mit `s` unifiziert

```
?- X is 2 + 5.
```

```
X = 7
```

```
?- 7 is 2 + 5.
```

```
true
```

```
?- 2 + 5 is 7.
```

```
false
```

```
len([], zero).  
len([_ | Rest], succ(N)) :- len(Rest, N).
```

```
len([], 0).  
len([_ | Rest], M) :- len(Rest, N), M is N + 1.
```

Vorlesung „Programmierung“

■ Inhalt der Vorlesung

- Was ist ein Programm?
- Was sind grundlegende Programmierkonzepte?
- Wie konstruiert (entwickelt) man ein Programm?
- Welche Programmier Techniken und -paradigmen gibt es?

■ Teil I: Einleitung und Grundbegriffe

■ Teil II: Imperative & objektorientierte Programmierung (*Java*)

■ Teil III: Funktionale Programmierung (*Haskell*)

■ Teil IV: Logische Programmierung (*Prolog*)