

Rekursive Programmieretechniken

Übersicht

- ◆ Linksrekursion
- ◆ Transitive Relationen berechnen
 - ◆ Hierarchische Beziehungen: Hyponymie
- ◆ Dekomposition eines (von mehreren) Arguments
 - ◆ Listen verketteten: append/3
- ◆ Akkumulatoren mit Zwischenresultaten gegen Ineffizienz
 - ◆ Listen umkehren: Naive vs. effiziente Version
- ◆ Doppelte Rekursion
 - ◆ verschachtelte Listen verflachen: flatten/2

Einfachste Rekursion mit Babuschka

Sinnvolle Definitionen von rekursiven Prozeduren müssen mindestens zwei Fälle abdecken.

- ◆ Abbruchbedingung(en)
- ◆ Rekursionsschritt(e)



Mit Babuschka spielen:

Die innerste Puppe kann nicht zerlegt werden.

Abbruchbedingung

Öffne die Puppe, lege Teile auf die Seite, spiele mit der enthaltenen Puppe weiter.

Rekursionsschritt

Linksrekursion mit Babuschka...

Linksrekursion liegt vor, wenn

- ◆ das erste Konjunkt im Rumpf dasselbe Prädikat ist wie im Kopf.

```
spiel(babuschka(Babuschka)) :-
    spiel(Babuschka).
spiel(babuschka).
```

Gefahr

- ◆ Prolog-Beweiser kommt in unendlich tiefen Suchast!

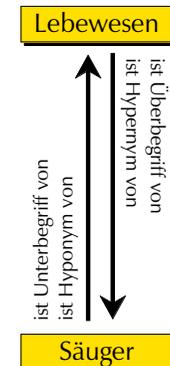
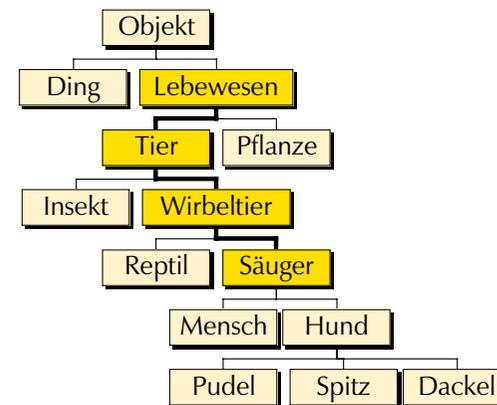
```
?- spiel(Babuschka).
{ERROR: Memory allocation failed}
{Execution aborted}
```

Ausweg

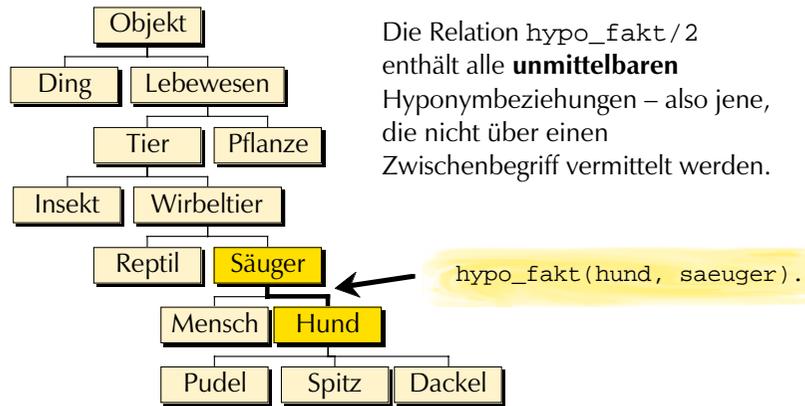
- ◆ Abbruchbedingungen vor die linksrekursive Klausel schreiben!

```
spiel(babuschka).
spiel(babuschka(Babuschka)) :-
    spiel(Babuschka).
?- spiel(Babuschka).
Babuschka = babuschka ? ;
Babuschka = babuschka(babuschka) ? ;
...
```

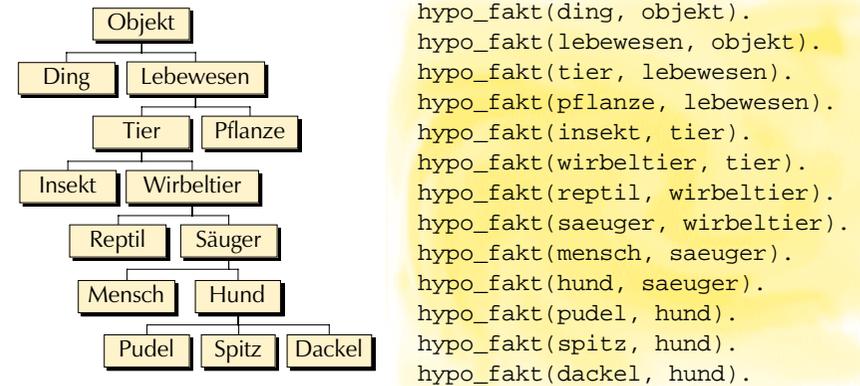
Hyponymie: Unterbegriffshierarchie



Unmittelbare Beziehung



hypo_fakt/2



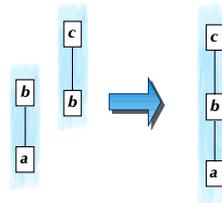
Unmittelbar und mittelbar hyponym

Problem: "Dackel" ist ein Hyponym von "Tier"

- Wie modellieren wir alle mittelbaren (=nicht unmittelbaren) Hyponymiebeziehungen, ohne sie alle einzeln aufzuzählen?

Idee: Hyponymie als Transitive Relation

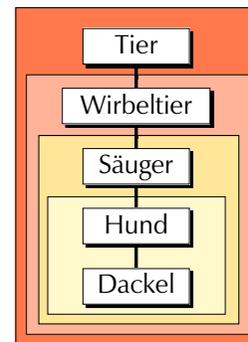
- Eine Relation R ist **transitiv**, falls gilt:
Wenn $R(a,b)$ und $R(b,c)$ besteht, dann besteht $R(a,c)$.



- Also:** Wenn a Unterbegriff von b ist, und b ist Unterbegriff von c , dann ist auch a Unterbegriff von c .

hypo = hypo1 + hypo2 + hypo3 + ...

Stufen der Hyponymie



`hypo1(X, Y) :- hypo_fakt(X, Y).`

`hypo2(X, Y) :- hypo_fakt(X, A), hypo_fakt(A, Y).`

`hypo3(X, Y) :- hypo_fakt(X, A), hypo_fakt(A, B), hypo_fakt(B, Y).`

`hypo4(X, Y) :- hypo_fakt(X, A), hypo_fakt(A, B), hypo_fakt(B, C), hypo_fakt(C, Y).`

`hypo1(X, Y) :- hypo_fakt(X, Y).`

`hypo2(X, Y) :- hypo_fakt(X, A), hypo1(A, Y).`

`hypo3(X, Y) :- hypo_fakt(X, A), hypo2(A, Y).`

`hypo4(X, Y) :- hypo_fakt(X, A), hypo3(A, Y).`

... hypo rekursiv

hypo2, ..., hypo4 liegt ein Schema zugrunde:

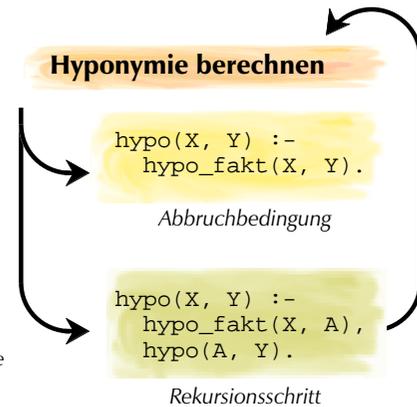
```
hypo1(X, Y) :-  
  hypo_fakt(X, Y).  
hypo2(X, Y) :-  
  hypo_fakt(X, A),  
  hypo1(A, Y).  
hypo3(X, Y) :-  
  hypo_fakt(X, A),  
  hypo2(A, Y).  
hypo4(X, Y) :-  
  hypo_fakt(X, A),  
  hypo3(A, Y).  
hypo(X, Y) :-  
  hypo_fakt(X, Y).  
hypo(X, Y) :-  
  hypo_fakt(X, A),  
  hypo(A, Y).
```

Rekursive Programmieretechniken – 9

Bestandteile von hypo/2

Die rekursive Definition von hypo/2:

- ◆ Abbruchbedingung
 - ▶ Unmittelbare Hyponymie
- ◆ Rekursionsschritt
 - ▶ Hyponymie, die über mehrere unmittelbare Stufen geht
 - ▶ Die Transitive "Hülle" der unmittelbaren Hyponymie wird rekursiv berechnet, d.h. die normale transitive Hyponymie-Relation.



Rekursive Programmieretechniken – 10

Listen verketteten: append/3

`[a, b, c]` + `[d, e]` = `[a, b, c, d, e]`

Das Verketteten zweier Listen wird häufig gebraucht.

- ◆ Das Prädikat `append/3` drückt die Verkettungsbeziehung aus.
 - ◆ Manchmal heißt es auch `concat/3` in Programmibibliotheken.

```
?- append([a,b,c], [d,e], Ergebnis).  
Ergebnis = [a,b,c,d,e]
```

Rekursive Programmieretechniken – 11

Rekursive Dekomposition

Abbruchbedingung

- ◆ Die Verkettung der leeren Liste mit einer Liste L ergibt wieder L .

```
append([], L, L).
```

Rekursionsschritt

- ◆ Um eine nicht leere Liste $[X|L1]$ mit einer Liste $L2$ zu verketteten, verkette $L1$ mit $L2$ zu $L3$ und stelle X als Anfangselement zu $L3$.

```
append([X|L1], L2, [X|L3]) :-  
  append(L1, L2, L3).
```

Rekursive Programmieretechniken – 12

Grammatikregeln als Listenverkettung

Grammatikregeln lassen sich als Listenverkettungen von Wörtern formulieren.

$S \rightarrow NP VP$

$[fido, frisst] \rightarrow [fido] [frisst]$

- ◆ Eine Wortliste X ist ein syntaktisch korrekter Satz, falls sie aus 2 Teillisten Y und Z besteht, so dass gilt:
 - ◆ X ist die Verkettung der Liste Y und Z (in dieser Reihenfolge)
 - ◆ die Wortliste Y ist eine syntaktisch korrekte Nominalphrase
 - ◆ die Wortliste Z ist eine syntaktisch korrekt Verbalphrase.

Parsen mit append/3

Die Grammatikregeln lassen sich mit `append/3` direkt formulieren und funktionieren als *Parser* (Programm zur syntaktischen Analyse).

```
% S --> NP VP
s(X) :-
    append(Y, Z, X),
    np(Y),
    vp(Z).
```

```
?- s([fido,frisst]).
yes
```

"Fido frisst" ist ein syntaktisch korrekter Satz.

```
% NP --> Eigename
np([fido]).
```

```
?- s([frisst,frisst]).
no
```

"frisst frisst" ist kein syntaktisch korrekter Satz.

```
% VP --> Vintransitiv
vp([frisst]).
```

Listen umkehren

$[k, a, n, u] \rightarrow [u, n, a, k]$

Gelegentlich ist es nötig, eine Liste umzudrehen.

- ◆ Version I `naive_reverse/2`: »naives« Umkehren
 - ▶ Laufzeit verhält sich kubisch zur Listenlänge (n^3)!
 - ▶ Um eine Liste der Länge 100 umzudrehen, brauchts etwa 1'000'000 Schritte.
- ◆ Version II `reverse_aku/3`: Akkumulator für Zwischenresultat
 - ▶ Laufzeit verhält sich linear zur Listenlänge (n)!
 - ▶ Um eine Liste der Länge 100 umzudrehen, brauchts etwa 100 Schritte.

Akkumulatoren sind ein wichtiges Mittel zur Effizienzsteigerung!

Listen umkehren: »Naive« Version

Abbruchbedingung

- ◆ Die Umkehrung der leeren Liste ist die leere Liste.

```
naive_reverse([], []).
```

Rekursionsschritt

- ◆ Um eine nicht leere Liste $[X|Rest]$ umzukehren, kehre den *Rest* um und verkette ihn mit der Einerliste $[X]$.

```
naive_reverse([X|Rest], Ergebnis) :-
    naive_reverse(Rest, RevRest),
    append(RevRest, [X], Ergebnis).
```

Listen umkehren: Akkumulator

Akkumulatoren dienen dem Festhalten und Weitergeben von Zwischenergebnissen bei rekursiven Prädikaten.

- ◆ Es muss eine Argumentstelle für den Akkumulator geschaffen werden.

```
reverse(Liste, UmgekehrteListe) :-  
    reverse_akku(Liste, [], UmgekehrteListe).
```

- ◆ reverse/2 wird auf reverse_akku/3 reduziert.
- ◆ Am Anfang ist der Akkumulator leer, d.h. die leere Liste.

Akkumulator: Abbruchbedingung

Abbruchbedingung

- ◆ Bei der Umkehrung der leeren Liste enthält das akkumulierte Zwischenresultat das Endresultat.

```
reverse_akku([], Akku, Akku).
```



- ▶ Für leere Listen wird reverse/2 korrekt durch reverse_akku/3 abgebildet...

```
?- reverse([], RL).  
RL = []
```



```
?- reverse_akku([], [], RL).  
RL = []
```

Akkumulator: Rekursionsschritt

Rekursionsschritt

- ◆ Um eine nicht leere Liste $[X|Rest]$ umzukehren, kehre den $Rest$ mit dem neuen Zwischenresultat $[X|Akku]$ um.

```
reverse_akku([X|Rest], Akku, Ergebnis):-  
    reverse_akku(Rest, [X|Akku], Ergebnis).
```

- ▶ Die Eingabeliste schrumpft mit jedem rekursiven Aufruf.
- ▶ Die Akkumulatorliste wächst mit jedem rekursiven Aufruf und stellt dann beim Abbruch das Ergebnis dar.
- ▶ Metapher: Die Elemente der Eingabeliste werden im Akkumulator umgestapelt!

Programmieretechnik Akkulatoren

Akkumulatoren werden normalerweise

- ◆ **initialisiert** beim ersten Aufruf
- ◆ **akkumuliert** bei rekursiven Schritten
- ◆ **unifiziert** zum Endresultat beim Erreichen der Abbruchbedingung

```
reverse(Liste, UmgekehrteListe) :-  
    reverse_akku(Liste, [], UmgekehrteListe).
```

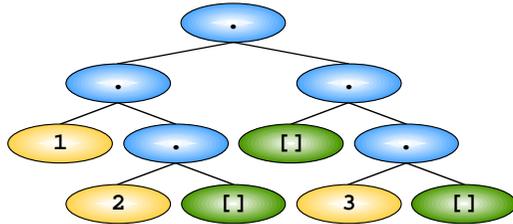
```
reverse_akku([], Akku, Akku).
```

```
reverse_akku([X|Rest], Akku, Ergebnis):-  
    reverse_akku(Rest, [X|Akku], Ergebnis).
```

Doppelrekursion: Verschachtelte Listen

Da Listen normale Terme sind, kann eine Liste auch Element einer anderen Liste sein.

```
?- [[1,2],[[],3]] = '.'('1','2,[ ]'), '.'([],'3,[ ]').  
yes
```



Rekursive Programmieretechniken – 21

Verschachtelte Listen verflachen

```
?- flatten([[1,2],[[],3]], Liste).  
Liste = [1,2,3]
```

Das Prädikat `flatten/2` wandelt eine verschachtelte Liste in flache Listen um.

◆ Was ist eine flache Liste?

Eine flache Liste ist eine Liste, deren Elemente keine Listen sind.

◆ Was ist eine verschachtelte Liste?

Eine verschachtelte Liste ist eine Liste, deren Elemente zum Teil aus Listen bestehen.

■ Listen sind verschachtelte Datenstrukturen, also sind verschachtelte Listen verschachtelte verschachtelte Datenstrukturen...

Rekursive Programmieretechniken – 22

Definition von `flatten/2`

Abbruchbedingung "leere Liste"

```
flatten([],[]).           % Leere Liste ist flach
```

Abbruchbedingung "keine Liste"

```
flatten(X,[X]):-         % Terme in flache Listen packen,  
  \+ is_list(X).         % die keine Listen sind
```

Rekursionsschritt "Doppelte Rekursion"

```
flatten([Kopf|Rest], FlacheListe):-  
  flatten(Kopf, FlacherKopf),           % Kopf verflachen  
  flatten(Rest, FlacherRest),           % Rest verflachen  
  append(FlacherKopf, FlacherRest, FlacheListe).  
                                          % und verketteten
```

Rekursive Programmieretechniken – 23