

# Rekursive Listenverarbeitung

---

## Übersicht

### Rekursion ist die wichtigste Programmieretechnik in Prolog!

- ◆ Rekursive Datenstrukturen
  - ◆ Einfache und rekursiv gebildete Strukturen
- ◆ Rekursive Datenstrukturen und rekursive Prädikate
  - ◆ Eine natürliche Kombination...
- ◆ Aufbau rekursiver Prädikate
  - ◆ Abbruchbedingung
  - ◆ Rekursionsschritt
- ◆ Rekursive Programmieretechniken mit Listen
  - ◆ Länge von Listen: `laenge/2`
  - ◆ Suche nach Elementen: `member/2`
  - ◆ Abbilden von Listen: `maplist/2`

# Listen: Rekursive Datenstrukturen

Bei Listen – wie bei allen rekursiven Datenstrukturen – gibt es 2 Arten von Definitionsregeln.

## I. Regeln für einfache Strukturen

- Die leere Liste ist eine Liste.

[ ]

- Atomaren Terme sind Prolog-Terme. Variablen sind Prolog-Terme.

## II. Regeln für rekursiv aufgebaute, komplexe Strukturen

- Nicht-leere Listen bestehen aus einem Element und einer Liste als Rest.

[E | Liste]

- Komplexe Terme sind Prolog-Terme, die aus einem Funktor und dessen Argumenten bestehen, die Prolog-Terme sind.

# Problem: Was sind wahre Listen?



`[a, b, c | [d]]`

`[a, b, c | d]`



## Listen sind eine bestimmte Sorte von Termen in Prolog

- ▶ Definiere ein Prädikate `is_list/1`, das genau dann wahr ist, wenn das Argument eine Liste ist!

### Problem

- ◆ Da Listen beliebig viele Elemente enthalten können, müssen beliebig viele Klauseln für `is_list/1` geschrieben werden!

```
is_list([]).  
is_list([_,_]).  
is_list([_,_,_]).  
...  
is_list([_,_,...,_]).
```

# Rekursive Lösung: Wahre Listen

## Rekursive Datenstrukturen + Rekursive Prädikate

- ▶ Passe die Strategie des Problemlösens der Struktur des Problems an!

### Also

- ◆ Prädikatsklausel für **einfache Struktur: Leere Liste**

```
is_list([]).
```

"Die leere Liste ist eine Liste."

- ◆ Prädikatsklausel für **rekursiven Strukturen: Nicht-Leere Liste**

```
is_list([E|Rest]) :-  
    is_list(Rest).
```

"Nicht-leere Listen bestehen aus einem Element und einer Liste als Rest."

# Rekursive Dekomposition

## Rekursive Datenstrukturen

- a. enthalten Teilstrukturen, die mit denselben Definitionsregeln aufgebaut wurden.

*Teilstrukturen sind immer weniger komplex als die sie enthaltenden Strukturen.*

- b. haben **elementare** Strukturen und **rekursive** Strukturen.

## Rekursive Prädikate

- a. lösen ein Problem, indem sie es auf Teilprobleme gleicher Art reduzieren, die deshalb mit dem gleichen Prädikat erschlagen werden können.

*Teilprobleme sind weniger komplex als das Problem, dessen Teil sie sind.*

- b. haben Klauseln für elementare oder abschliessende Fälle (**Abbruchbedingung**) und rekursive Fälle (**Rekursionsschritt**).

# Der Bau rekursiver Prädikate

Für den Aufbau und das Schreiben rekursive Prädikate kann oft ein gemeinsames Schema verwendet werden.

◆ **Zuerst:** Klauseln für **Abbruchbedingung**

- ▶ Terminiere den Beweis, falls die Abbruchbedingung erfüllt ist.
- ▶ Oft einfach zu finden und zu programmieren!

◆ **Danach:** Klauseln für **Rekursionschritte**

- ▶ Löse das Problem für einen einzelnen Schritt und wende auf das Restproblem dasselbe Prädikat rekursiv an.
- ▶ Oft erstaunlich einfache Definition, aber schwierig zu finden!

***Gefahr: Rekursive Prädikate ohne Abbruchbedingungen verhalten sich wie zirkulär definierte Prädikate!***

```
huhn :- ei.  
ei :- huhn.
```

# Beispiel 1: Länge von Listen

Schreibe ein rekursives Prädikat `laenge/2`, das die Länge einer Liste bestimmt.

- ◆ Erstes Argument (*Input*): Liste, deren Länge zu bestimmen ist
- ◆ Zweites Argument (*Output*): Die Länge (d.h. Anzahl der Elemente)

```
?- laenge([], X).  
X = 0  
yes
```

```
?- laenge([a,b,c], X).  
X = 3  
yes
```

Wie immer bei rekursiven Prädikaten unterscheiden wir

- ◆ Abbruchbedingung
- ◆ Rekursionsschritt

# Definition: Länge von Listen

## Abbruchbedingung

- ◆ Die Länge der leeren Liste ist 0.

```
laenge([], 0).
```

## Rekursionsschritt

- ◆ Die Länge einer nicht-leeren Liste ist die Länge ihres Rests plus 1.

```
laenge([_|Rest], Ergebnis) :-  
    laenge(Rest, RestLaenge),  
    Ergebnis is RestLaenge + 1.
```

## Hinweis

*In SICStus Prolog gibt es ein eingebautes Prädikat `length/2`, das die Länge einer Liste berechnen kann wie unser Prädikat `laenge/2`. Zusätzlich kann es aber noch verwendet werden, um Listen bestimmter Länge zu generieren.*



# Beispiel II: Suche nach einem Element

Schreibe ein Prädikat `member/2`, das wahr ist, falls ein Term Element einer Liste ist.

```
?- member(sittich, [spitz,dackel,terrier]).  
no
```

```
?- member(dackel, [spitz,dackel,terrier]).  
yes
```

```
?- member(X, [spitz,dackel,terrier]).  
X = spitze ;  
X = dackel ;  
X = terrier ;  
no
```

# Suche: Rekursive Dekomposition...

---

**Das gesuchte Element ist das erste Element der Liste.**

- ◆ **Abbruchbedingung:** Das vorderste Element ist mit dem gesuchten Term unifizierbar.

```
member(dackel, [dackel, terrier])
```

**Das gesuchte Element befindet sich vielleicht im Rest der Liste.**

- ◆ **Rekursionsschritt:** Suche im Listenrest weiter (d.h. ohne das Anfangs-Element)

```
member(dackel, [mops, dackel, terrier])
```

# Abbruchbedingung: Gefunden!

```
member(dackel, [dackel, terrier])
```

## Die Klausel für die erfolgreiche Suche...

- ◆  $X$  ist Element der Liste, wenn  $X$  das erste Element in der Liste ist.

```
member(X, Liste) :-  
    Liste = [X | IrgendeinRest].
```

- ◆ Einfacher geschrieben:

```
member(X, [X | IrgendeinRest]).
```

- ◆ Eigentlich interessiert uns der Rest gar nicht:

```
member(X, [X | _]).
```

# Rekursiver Fall: Weitersuchen!

```
member(dackel, [mops, dackel, terrier])
```

**Das Element könnte noch im Rest enthalten sein, d.h. im Rest weitersuchen!**

- ◆ Nimm den Rest der Liste und prüfe, ob  $X$  im Rest enthalten ist.

```
member(X, [Anfang | Rest]) :-  
    member(X, Rest).
```

- ◆ Eigentlich interessiert uns der Anfang gar nicht.

```
member(X, [_ | Rest]) :-  
    member(X, Rest).
```

# Rekursive Suche: member/2

---

```
member(X, [X|_]).  
member(X, [_|Rest]) :-  
    member(X, Rest).
```

## Deklarative Verdeutschung

- ◆ Ein Term ist Element einer Liste, falls der Term Kopf der Liste ist.
- ◆ Ein Term ist Element einer Liste, falls der Term Element des Rests der Liste ist.

# Listen: Analyse und Konstruktion

---

## Listen-Analyse

- ◆ In rekursiven Listenprädikaten wird meist eine Eingabe-Liste rekursiv auseinandergenommen (analysiert)

## Listen-Konstruktion

- ◆ In rekursiven Listenprädikaten wird oft zugleich eine Ausgabe-Liste rekursiv aufgebaut (konstruiert), die das gewünschte Resultat enthält

## Für Dekonstruktion wie Konstruktion wird Unifikation verwendet!

- ▶ *Pattern Matching spielt Doppelrolle der analytischen Fallunterscheidung und Resultatskonstruktion!*

# Beispiel III: Abbilden (*Mapping*)

Listenmapping mit Prädikat `papagei/2` ist Beispiel für gleichzeitige Listen-Analyse und -Konstruktion.

- ◆ **Eingabe:** Liste
- ◆ **Ausgabe:** Eingabeliste, in der bestimmte Element ersetzt sind

```
?- papagei([du,bist,nett], Echo).  
Echo = [ich,bin,nett]
```

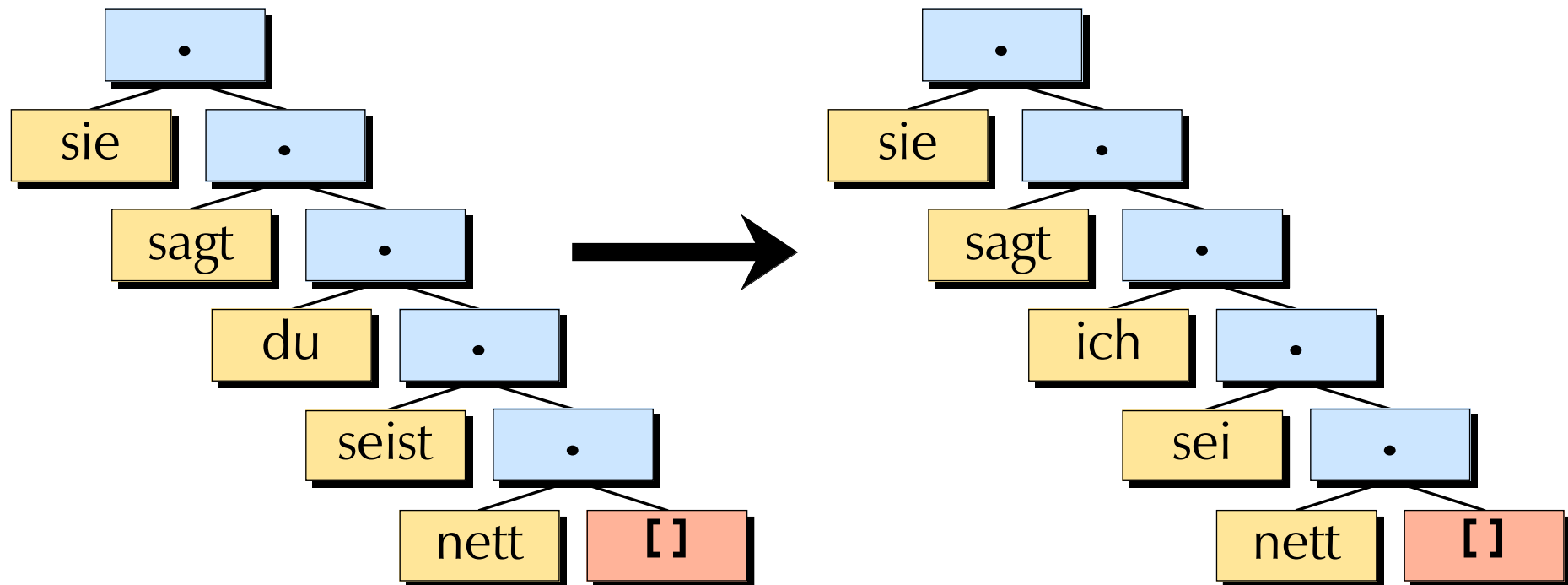
```
?- papagei([sie,sagt,du,seist,nett], Echo).  
Echo = [sie,sagt,ich,sei,nett]
```

Im Beispiel ist die Ausgabe gleich der Eingabe, ausser für die Listenelemente *du*, *ich*, *bist*, *bin*, *sei*, *seist*.

# Abbilden der Liste

## Vorgehen: Analyse und Konstruktion

- ◆ Gehe schrittweise durch die Liste (»traversieren«)
- ◆ Bilde jedes einzelne Element auf das entsprechende Ergebnis ab.



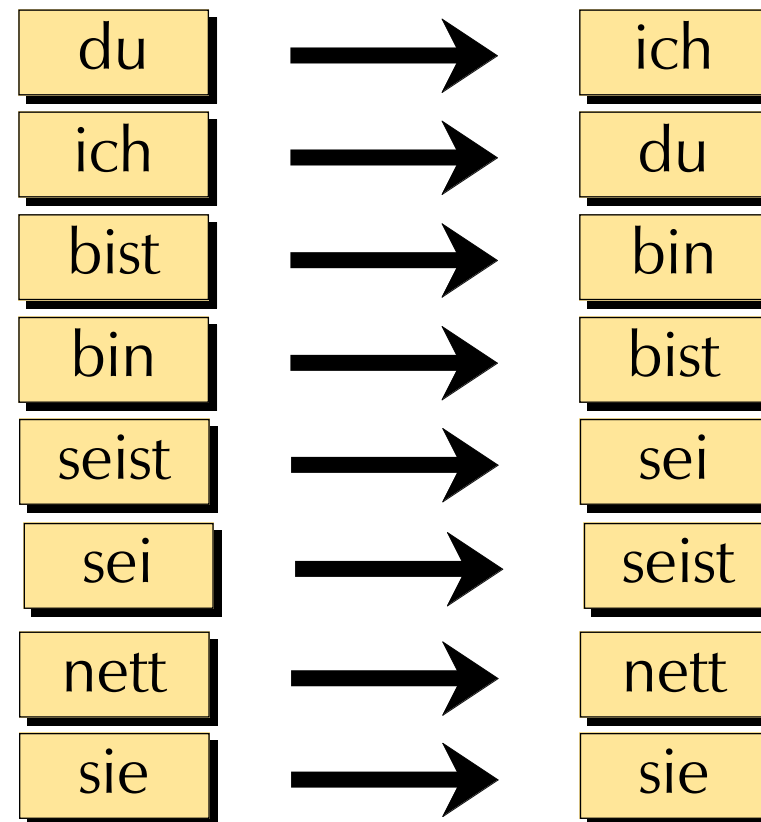


# Abbilden der Elemente: map/2

Zum Austauschen der einzelnen Listenelemente definieren wir das Hilfsprädikat map/2.

```
map(du, ich).  
map(ich, du).  
map(bist, bin).  
map(bin, bist).  
map(seist, sei).  
map(sei, seist).
```

```
map(X, X).
```



# Zuerst Abbruchbedingung...

Das Prädikat `papagei/2` bildet eine Liste in eine andere ab.

## Abbruchbedingung

- ◆ Bilde die leere Liste auf die leere Liste ab.

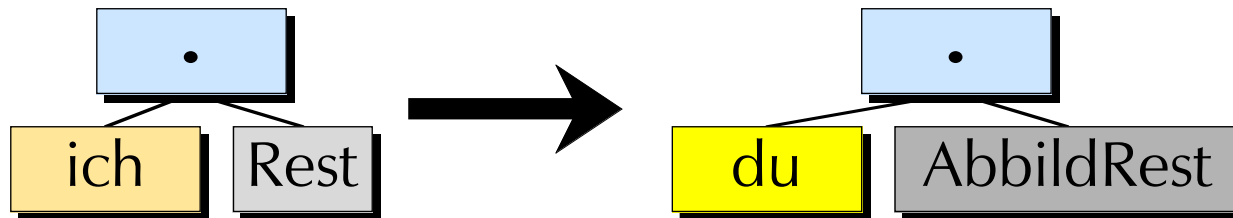
*Da kein Element vorhanden ist, muss keines ausgetauscht werden.*

```
papagei([], []).
```



- ▶ Die Abbruchbedingung steht normalerweise vor dem rekursiven Fall.

# ... dann Rekursionsschritt



## Rekursionsschritt

- ◆ Nimm Anfangselement und bilde es mit map/2 ab.
- ◆ Rufe papagei/2 rekursiv auf, um den Rest der Liste abzubilden
- ◆ Konstruiere die Resultatsliste, die besteht aus
  - ◆ einem Anfangs-Element: der Abbildung des ersten Elements
  - ◆ einer Restliste: der Abbildung des Rests

```
papagei([E|Rest], [AbbE|AbbildRest]) :-  
    map(E, AbbE),  
    papagei(Rest, AbbRest).
```

# Abbilden im Überblick: papagei/2

**% Abbildung bestimmter Terme.**

```
map(du, ich).  
map(bist, bin).  
map(ich, du).  
map(bin, bist).  
map(seist, sei).  
map(sei, seist).
```

**% Wenn es keiner der obigen Terme ist, ist  
% das Abbild gleich dem Original.**

```
map(X, X).
```

**% Die Abbildung einer leeren Liste ergibt  
% eine leere Liste.**

```
papagei([], []).
```

**% Die Abbildung eines Terms, gefolgt von einer Rest-Liste  
% ist die Abbildung des Terms, gefolgt von der Abbildung  
% der Rest-Liste.**

```
papagei([E|Rest], [AbbE|AbbRest]) :-  
    map(E, AbbE),  
    papagei(Rest, AbbRest).
```