Listen

Übersicht

- Listen allgemein
- Listen in Prolog
  - Schreibweise
  - Listen als rekursive Datenstruktur
  - Unifikation
- Programmiertechniken mit Listen
  - rekursive Suche
  - Abbilden

Zweck

- Erstellen von Prolog-Programmen mit Listen

Listen treten häufig auf:

- Teilnehmer eines Kurses
- Einzelstationen eines Weges
- Mitglieder einer Familie
- Wörter in einem Satz
- Buchstaben eines Wortes
- …

Eigenschaften

Listen besitzen bestimmte Eigenschaften

- Die einzelnen Elemente sind geordnet
- Dasselbe Element kann mehrmals vorkommen

  Beispielliste: Wörter im Satz »Wenn Fliegen hinter Fliegen fliegen, fliegen Fliegen Fliegen nach«

- Listen können beliebig lang werden
- Listen können auch leer sein (sog. *leere Liste*)

Listen in Prolog

Für Listen gibt es in Prolog eine eigene Schreibweise

- Elemente zwischen eckigen Klammern eingeschlossen
- durch Kommata getrennt
- Elemente = beliebige Terme

[apfel, Frucht]

Zwei Elemente
(ein Atom und eine Variable)

[kiwi]

Ein Element

[ ]

leere Liste

Listen in Prolog

Eigentlich sind Listen ganz normale Terme

- die Schreibweise mit [und] ist eine Abkürzung für eine etwas kompliziertere Struktur

  → später im Semester

- die einzelnen Elemente einer Liste können selber Listen sein

[[d,a,t,t,e,l], [ ], [k,o,k,o,s]]

Listen als Elemente von Listen

Unifikation von Listen

Zwei Listen können miteinander unifiziert werden:

- die einzelnen Elemente werden paarweise unifiziert
- die Länge beider Listen muss übereinstimmen

```
frisst(fido, [gulasch, Gemuese])
frisst(fido, [Fleisch,  gurke ])
```

Fleisch/gulasch

```
frisst(fido, [gulasch, Gemuese])
frisst(fido, [gulasch,  gurke ])
```

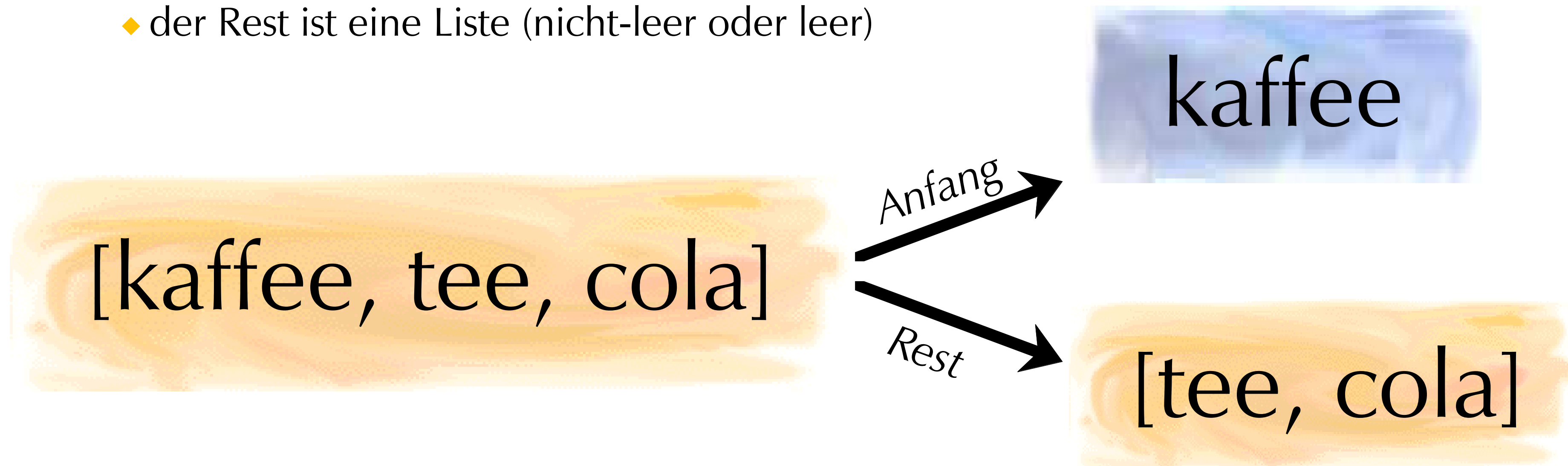
Gemuese/gurke

```
frisst(fido, [gulasch,  gurke ])
frisst(fido, [gulasch,  gurke ])
```

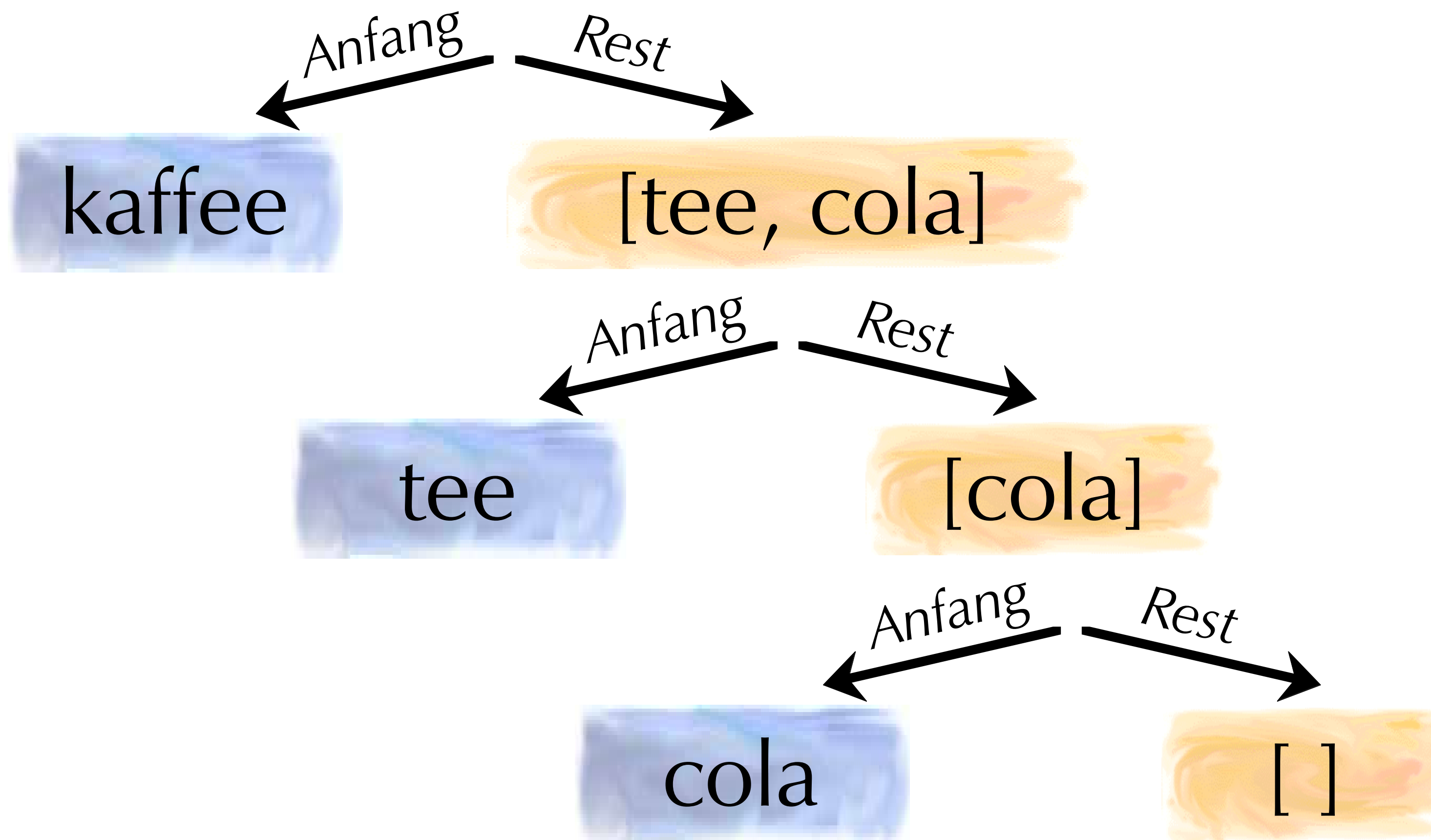
Listen: Eine rekursive Datenstruktur

Eine nicht-leere Liste besteht aus zwei Teilen:

- einem Anfang *(Head, First Element)*
  - das erste Element kann ein beliebiger Term sein
- einem Rest *(Rest, Tail)*
  - der Rest ist eine Liste (nicht-leer oder leer)



Listen: Eine rekursive Datenstruktur



Listen in Prolog: |

Mit | ist der Rest einer Liste erreichbar.

- Dies wird oft im Rahmen einer Unifikation benutzt

```
?- [a, b, c, d] = [a, b | Rest].
Rest = [c, d]
```

```
?- [a, b, c, d] = [a, b, c, d | Rest].
Rest = []
```

```
?- [Anfang | Rest] = [a, b | [c, d]].
Anfang = a, Rest = [b, c, d]
```

Rekursive Suche

Schreibe ein Prolog-Prädikat `member/2`, das bestimmt, ob ein Term in einer Liste enthalten ist.

Beispiele für die Anwendung:

- ```
  ?- member(dackel, [spitz, dackel, terrier]).
  yes
  ```
- ```
  ?- member(sittich, [spitz, dackel, terrier]).
  no
  ```
- ```
  ?- member(X, [spitz, dackel, terrier]).
  X = spitz ;
  X = dackel ;
  X = terrier ;
  no more solutions
  ```
  ```

Rekursive Suche

Die Lösung geht rekursiv vor, indem jeweils das vorderste Element der Liste mit dem gesuchten Term verglichen wird.

- **Abbruchbedingung:** Das vorderste Element ist mit dem gesuchten Term unifizierbar.
- **Rekursiver Fall:** Suche im Rest der Liste (d.h. ohne das vorderste Element) weiter.

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

Rekursive Suche: Abbruchbedingung

Wie bei allen rekursiven Problemen steht die Abbruchbedingung zuerst im Programm.

- *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 | _]).
```

Beispiel: `member(dackel, [dackel, terrier]).`

Rekursive Suche: Rekursiver Fall

Zweite Möglichkeit: X könnte im Rest enthalten sein.

- 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).
```

Beispiel:

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

Rekursive Suche: member / 2

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

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

Abbilden (*Mapping*)

Eingabe: Eine rekursive Prolog-Struktur (z. B. Liste).

Ausgabe: Ähnliche Struktur, jedoch in bestimmter Weise verändert.

Beispiel:

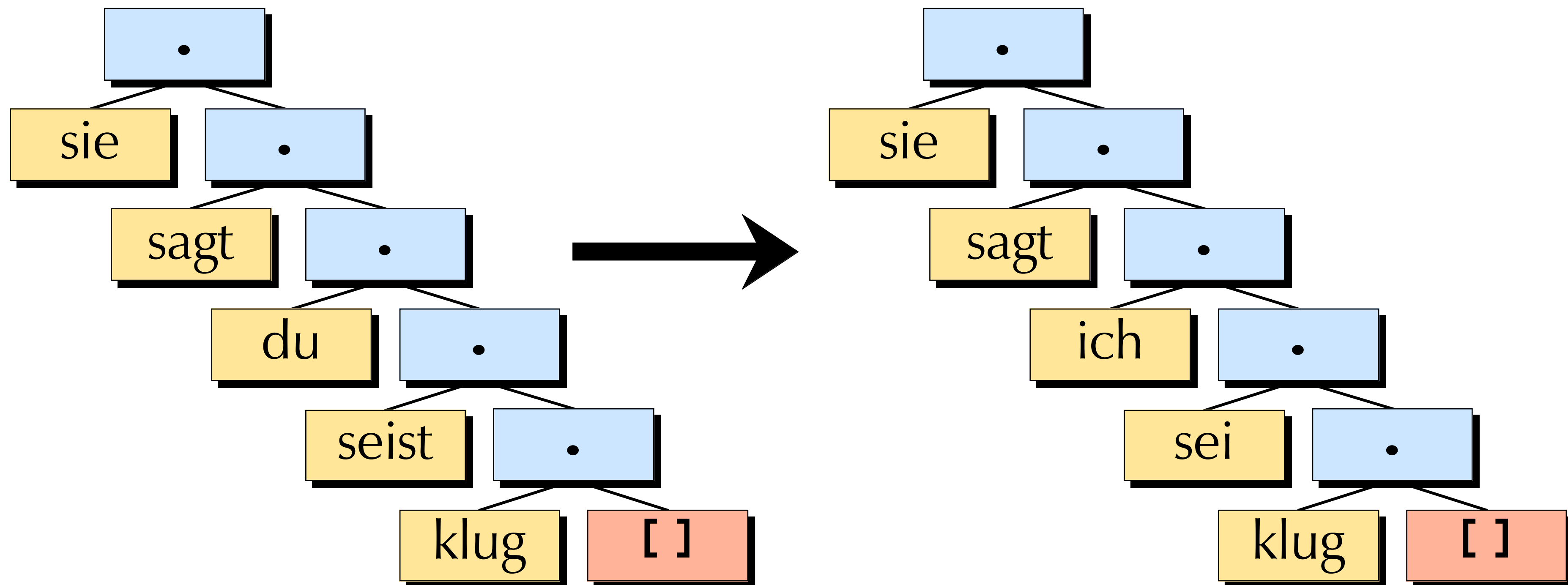
- [du, bist, nett] $\Rightarrow$ [ich, bin, nett]
- [ich, bin, schön] $\Rightarrow$ [du, bist, schön]
- [sie, sagt, du, seist, klug] $\Rightarrow$ [sie, sagt, ich, sei, klug]

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

Abbilden (*Mapping*)

Vorgehen: Rekursion

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

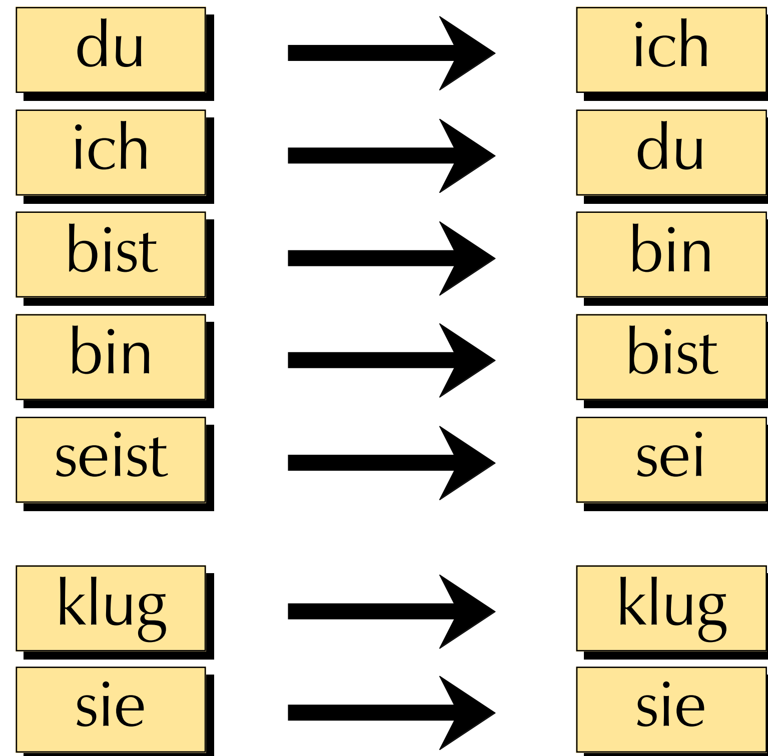


Abbilden: change/2

Zum Abbilden der einzelnen Listenelemente definieren wir uns ein Hilfsprädikat change/2.

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

change(X, X).
```



Abbilden: Abbruchbedingung

Das Prädikat `papagei/2` bildet eine Liste in eine andere ab, wobei es sich `change/2` bedient.

Wie bei allen Rekursionen steht die Abbruchbedingung *vor* dem rekursiven Fall.

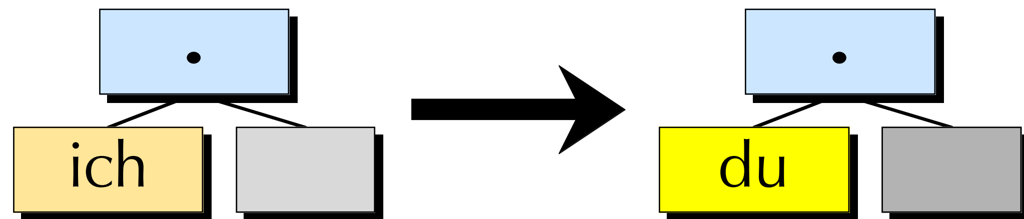
- **Abbruchbedingung:** Eine leere Liste wird auf eine leere Liste abgebildet.

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

Abbilden: Rekursiver Fall

Der rekursive Fall …

- benutzt `change/2`, um ein einzelnes Listenelement abzubilden
- ruft `papagei/2` rekursiv auf, um den Rest der Liste abzubilden
- gibt als Resultat eine Liste zurück, die besteht aus:
  - einem erstes Element: die Abbildung des ersten Elements
  - einem Rest: die Abbildung des Rests




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

Abbilden: papagei/2

```
% Abbildung bestimmter Terme.
change(du, ich).
change(bist, bin).
change(ich, du).
change(bin, bist).
change(seist, sei).

% Wenn es keiner der obigen Terme ist, ist
% die Abbildung gleich dem Original.
change(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]) :-
  change(E, AbbE),
  papagei(Rest, AbbRest).
```