Komposition und Differenzlisten

Übersicht

- Morphologie: Fallbeispiel Komposition
  - Komposition als Listenverkettung
  - Stämme und ihre Information
  - Komposition und Vererbung von Information
  - Effizienzprobleme mit append/3
- Differenzlisten
  - Offene Listen + Zugriff auf Restliste
  - Verkettung von Differenzlisten: append_dl/3
  - Implizites Verketten durch Variablenbindung

▸ Differenzlisten sind wichtigste Datenstruktur in Prolog für effiziente Verarbeitung sequentieller Daten!

Komposition als Verkettung

arbeit+s+zeit — **`[a,r,b,e,i,t]+[s]+[z,e,i,t]`**

Kompositionsanalyse — *Repräsentation in PROLOG*

Simple Idee zur Bildung und Analyse von Komposita

- Listen von Buchstabenatomen repräsentieren
  - Einfache Worte
  - Allfällige Fugen (z.B. s, en, n)
- Komposition ist Listenverkettung, sowohl für

▸ Analyse wie

```
?- append([b,r,o,t], N, [b,r,o,t,z,e,i,t]).
N = [z,e,i,t]
```

▸ Synthese

```
?- append([b,r,o,t], [z,e,i,t], K).
K = [b,r,o,t,z,e,i,t]
```


n_stamm/4 und n_comp/4

Nominalstämme: n_stamm/4

i. Zeichenfolge
ii. Flexionsklasse
iii. Genus
iv. Fugenforderung

```
n_stamm([a,r,b,e,i,t], 1, f, [s]).
n_stamm([z,e,i,t], 1, f, []).
n_stamm([b,r,o,t], 1, n, []).
n_stamm([p,a,u,s,e], 2, f, [n]).
```

Nominalkomposita: n_comp/4

i. Zeichenfolge
ii. Flexionsklasse des Kompositum
iii. Genus des Kompositum
iv. Fugenforderung des Kompositum

```
n_comp(Kompositum, Flexion, Genus, Fuge) :-
    n_stamm(Stamm1, _, _, Fuge1),
    append(Stamm1, Fuge1, Teil1),
    n_stamm(Stamm2, Flexion, Genus, Fuge),
    append(Teil1, Stamm2, Kompositum).
```


append/3

Listen verketten

```
append([], L2, L2).

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

Die Laufzeit ist proportional zur Länge der ersten Liste.

- rekursives Abarbeiten der ersten Liste
- für jedes Element der ersten Liste ein rekursiver Aufruf

Effizienzüberlegungen

Verwendung von append/3 führt zu grosser Ineffizienz

- um Fugenelement anzuhängen, muss das Erstglied vollständig dekomponiert werden

```
3  2  Call: append([a,r,b,e,i,t],[s],_522) ?
9  8  Exit: append([],[s],[s]) ?
3  2  Exit: append([a,r,b,e,i,t],[s],[a,r,b,e,i,t,s]) ?
```

- bevor falsches Erstglied bemerkt wird, werden alle Zweitglieder ausprobiert

```
11  2  Call: append([a,r,b,e,i,t,s],[a,r,b,e,i,t],[b,r,o,t,z,e,i,t])
11  2  Call: append([a,r,b,e,i,t,s],[z,e,i,t],[b,r,o,t,z,e,i,t])
11  2  Call: append([a,r,b,e,i,t,s],[b,r,o,t],[b,r,o,t,z,e,i,t])
11  2  Call: append([a,r,b,e,i,t,s],[p,a,u,s,e],[b,r,o,t,z,e,i,t])
```

- es werden blind alle Kombinationen versucht, bis allenfalls die passende Struktur erscheint

Verbesserungsmöglichkeiten

Wie optimieren?

- Fallunterscheidung, falls Fugenelement "leer" ist
- Effizienteres append/3 mit Akkumulatortechnik verwenden
  - Nur minime Verbesserung!
- Falsche Erstglieder sofort erkennen und nicht noch Zweitglied mutieren
- Bessere Datenstruktur wählen, die Listenverkettung zulässt, ohne dass eine Liste immer vollständig auseinander zu nehmen ist

Erstaunlicherweise gibt es eine Prolog-Datenstruktur, mit der die letzten 2 Punkte verbessert werden können:

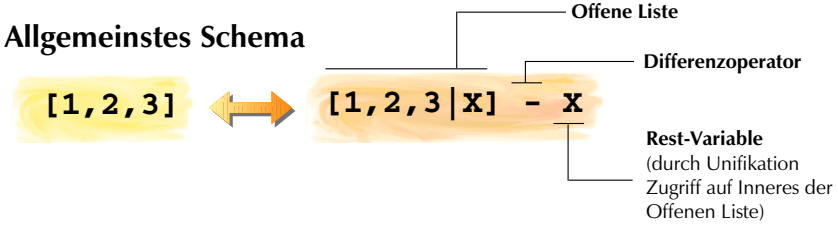
Differenzlisten

Differenzlisten

Die Liste [1,2,3] als Differenz unterschiedlicher Listen

```
[1,2,3]        [1,2,3,4,5] - [4,5]
[1,2,3]   <=>  [1,2,3|[bla]] - [bla]
[1,2,3]        [1,2,3] - [ ]
```

Allgemeinstes Schema



Differenzlisten II

Listen können leicht in Differenzlisten mit denselben Elementen umgewandelt werden:

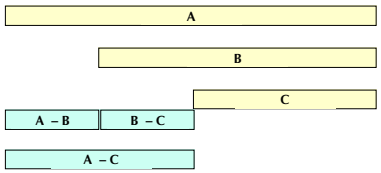
```
[1,2,…]   <=>  [1,2,…|X] - X
[]        <=>  X - X
```

```
n_stamm([z,e,i,t], 1, f, []).
<=>
n_stamm_dl([z,e,i,t|X]-X, 1, f, Y-Y).
```

append_dl/3

Listen verketten mit Differenzlisten

```
append_dl(A-B, B-C, A-C).
```




```
?- append_dl([1,2|X]-X,
             [4,5]-[],
             Z).
```

```
?- append_dl([1,2|X]-X,
             [4,5|Y]-Y,
             Z).
```

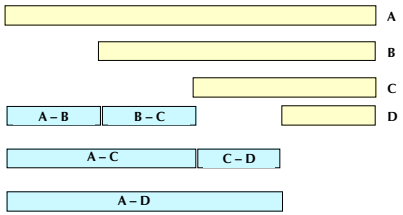
Laufzeit ist konstant

- keine Rekursion, sondern simple Term-Unifikation

Anwendung von append_dl/3

```
n_comp_dl(A-D, Flexion, Genus, Fuge-E) :-
   n_stamm_dl(A-B, _, _, B-C),
   append_dl(A-B, B-C, A-C),
   n_stamm_dl(C-D, Flexion, Genus, Fuge-E),
   append_dl(A-C, C-D, A-D).
```

```
?- n_comp_dl([a,r,b,e,i,t,s,z,e,i,t]-[], Flex, Gen, Fuge-E).
```



n_stamm_dl/4 und n_comp_dl/4

Nominalstämme

- Zeichenfolge als Differenzliste
- Fugenforderung als Differenzliste

```
n_stamm_dl([a,r,b,e,i,t|X]-X, 1, f, [s|Y]-Y).
n_stamm_dl([z,e,i,t|X]-X, 1, f, Y-Y).
n_stamm_dl([b,r,o,t|X]-X, 1, n, Y-Y).
n_stamm_dl([p,a,u,s,e|X]-X, 2, f, [n|Y]-Y).
```

Nominalkomposita: n_comp_dl/4

i. Zeichenfolge als Differenzliste
ii. Flexionsklasse
iii. Genus
iv. Fugenforderung als Differenzliste

```
n_comp_dl(A-D, Flexion, Genus, Fuge-E) :-
   n_stamm_dl(A-B, _, _, B-C),
   append_dl(A-B, B-C, A-C),
   n_stamm_dl(C-D, Flexion, Genus, Fuge-E),
   append_dl(A-C, C-D, A-D).
```


Redundanz von append_dl/3

Die beiden Aufrufe von append_dl sind redundant, da alle Variablen schon vorher identisch instantiiert sind.

```
n_comp_dl(A-D, Flexion, Genus, Fuge-E) :-
   n_stamm_dl(A-B, _, _, B-C),
   append_dl(A-B, B-C, A-C),
   n_stamm_dl(C-D, Flexion, Genus, Fuge-E),
   append_dl(A-C, C-D, A-D).
```

Deshalb kurz und elegant:

```
n_comp_dl(A-D, Flexion, Genus, Fuge-E) :-
   n_stamm_dl(A-B, _, _, B-C),
   n_stamm_dl(C-D, Flexion, Genus, Fuge-E).
```