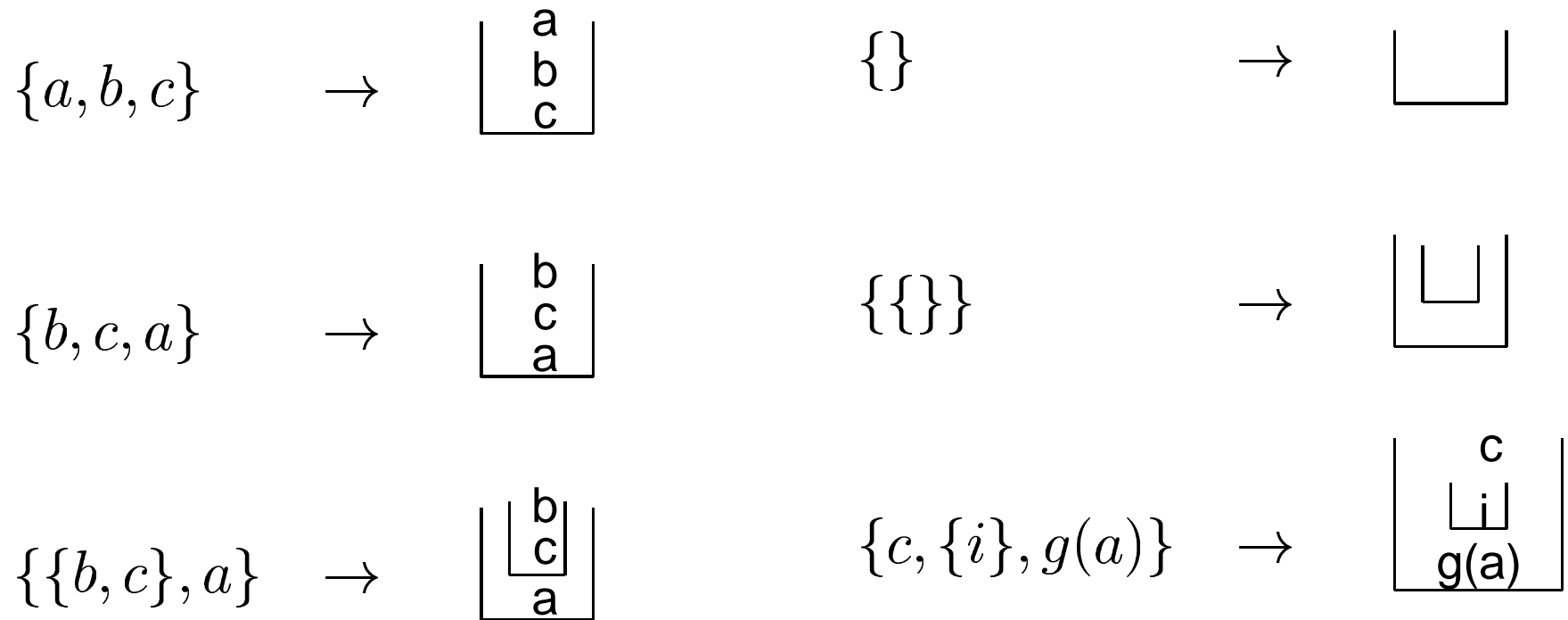


LISTEN

Liste

- Listen sind rekursive Datenstrukturen, die dazu dienen geordnete Mengen von Elementen zu beschreiben.
- Man kann sich Listen in Prolog als Behälter vorstellen, die Elementen verschiedener Typen behalten können.
- Die Reihenfolge, in der Behälter gefüllt werden, ist wichtig

Beispiele



Numeral

Die folgende rekursive Definition definiert eine natürliche Zahl:

1. 0 ist eine natürliche Zahl
2. $\text{succ}(X)$ ist eine natürliche Zahl wenn X eine natürliche Zahl ist

Definition von Liste

Die folgende rekursive Definition definiert eine Liste in Prolog:

1. `[]` ist eine Liste
2. `.(T,L)` ist eine liste wenn L eine Liste ist und T ein beliebiger Term

T heisst *head* der Liste, L heisst *tail*.

Listen in Prolog

```
/*
```

```
Das Predikat liste implementiert die folgende  
rekursive Definition von Liste:
```

```
1. [] ist eine Liste
```

```
2. .(T,L) ist eine Liste wenn L eine Liste ist
```

```
*/
```

```
% Basisklausel
```

```
liste([]).
```

```
% Recursive Klausel
```

```
liste(.(T,L)) :- liste(L).
```

Beispiele

$\{a, b, c\} \rightarrow .(a, .(b, .(c, [])))$

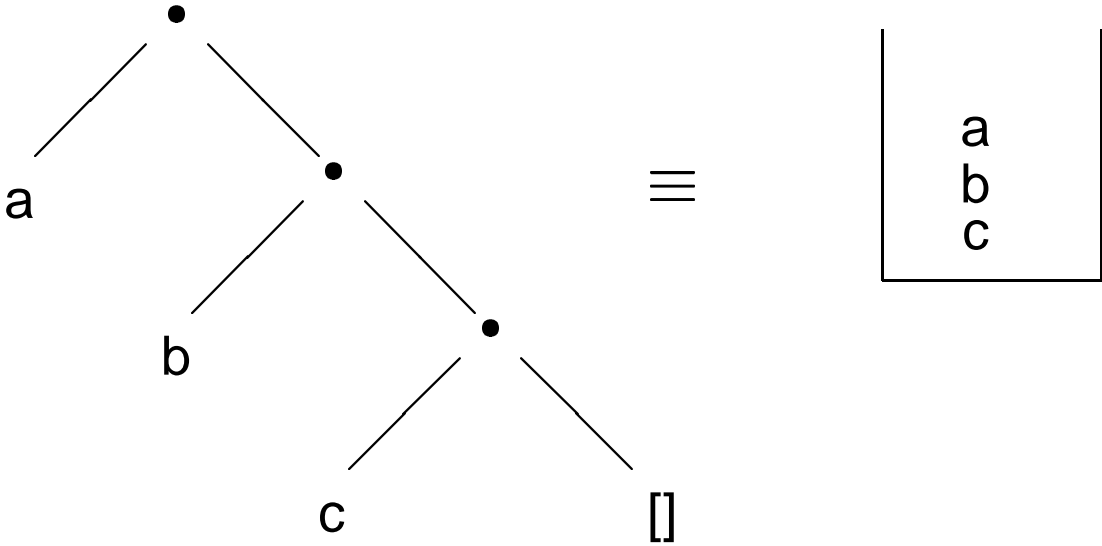
$\{a \{b, c\}\} \rightarrow .(a, .(. (b, .(c, [])), []))$

$\{\} \rightarrow []$

$\{\{\}\} \rightarrow .([], [])$

$\{c, \{i\}, g(a)\} \rightarrow .(c, .(. (i, []), .(g(a), [])))$

Listen als Bäume



Länge einer Liste

```
/*
```

Das Predikat **count_length** implementiert die folgende rekursive Definition von Länge ($|L|$) einer Liste:

1. $|[]| = 0$

2. $|\cdot(T,L)| = 1 + |L|$

```
*/
```

```
% Basisklausel
```

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

```
% Recursive Klausel
```

```
count_length(·(T,L), succ(X)) :- count_length(L,X).
```

Studenten = {martin, anna, hans}

Studenten = { martin, anna, hans }

In Prolog:

Studenten = .(martin, .(anna, .(hans, []))).

Studenten = { martin, anna, hans }

In Prolog:

Studenten = .(martin, .(anna, .(hans, []))).

0. count_length(Studenten, N) .

Studenten = {martin, anna, hans}

In Prolog:

Studenten = .(martin, .(anna, .(hans, []))).

0. count_length(Studenten, N).

1. count_length(. (martin, . (anna, . (hans, []))) , N) .

Studenten = { martin, anna, hans }

In Prolog:

Studenten = .(martin, .(anna, .(hans, []))).

0. count_length(Studenten, N) .

1. count_length(. (martin, . (anna, . (hans, []))) , N) .

2. N = 1 + I1 =succ(I1)

Studenten = { martin, anna, hans }

In Prolog:

Studenten = .(martin, .(anna, .(hans, []))).

0. count_length(Studenten, N) .
1. count_length(. (martin, . (anna, . (hans, []))) , N) .
2. N = 1 + I1 =succ(I1)
3. count_length(. (anna, . (hans, [])) , I1) .

Studenten = { martin, anna, hans }

In Prolog:

Studenten = .(martin, .(anna, .(hans, []))).

0. count_length(Studenten, N) .
1. count_length(. (martin, . (anna, . (hans, []))) , N) .
2. N = 1 + I1 = succ(I1)
3. count_length(. (anna, . (hans, [])) , I1) .
4. I1 = 1 + I2 = succ(I2)

Studenten = { martin, anna, hans }

In Prolog:

Studenten = .(martin, .(anna, .(hans, []))).

0. count_length(Studenten, N) .
1. count_length(. (martin, . (anna, . (hans, []))) , N) .
2. N = 1 + I1 = succ(I1)
3. count_length(. (anna, . (hans, [])) , I1) .
4. I1 = 1 + I2 = succ(I2)
5. count_length(. (hans, []) , I2) .

Studenten = { martin, anna, hans }

In Prolog:

Studenten = .(martin, .(anna, .(hans, []))).

0. count_length(Studenten, N) .
1. count_length(. (martin, . (anna, . (hans, []))) , N) .
2. N = 1 + I1 = succ(I1)
3. count_length(. (anna, . (hans, [])) , I1) .
4. I1 = 1 + I2 = succ(I2)
5. count_length(. (hans, []) , I2) .
6. I2 = 1 + I3 = succ(I3)

Studenten = { martin, anna, hans }

In Prolog:

Studenten = .(martin, .(anna, .(hans, []))).

0. count_length(Studenten, N) .
1. count_length(. (martin, . (anna, . (hans, []))) , N) .
2. N = 1 + I1 = succ(I1)
3. count_length(. (anna, . (hans, [])) , I1) .
4. I1 = 1 + I2 = succ(I2)
5. count_length(. (hans, []), I2) .
6. I2 = 1 + I3 = succ(I3)
7. count_length([], I3)

Studenten = { martin, anna, hans }

In Prolog:

Studenten = .(martin, .(anna, .(hans, []))).

0. count_length(Studenten, N) .
1. count_length(. (martin, . (anna, . (hans, []))) , N) .
2. N = 1 + I1 = succ(I1)
3. count_length(. (anna, . (hans, [])) , I1) .
4. I1 = 1 + I2 = succ(I2)
5. count_length(. (hans, []), I2) .
6. I2 = 1 + I3 = succ(I3)
7. count_length([], I3)
8. count_length([], 0) .

Studenten = { martin, anna, hans }

In Prolog:

Studenten = .(martin, .(anna, .(hans, []))).

0. count_length(Studenten, N) .
1. count_length(. (martin, . (anna, . (hans, []))) , N) .
2. N = 1 + I1 = succ(I1)
3. count_length(. (anna, . (hans, [])) , I1) .
4. I1 = 1 + I2 = succ(I2)
5. count_length(. (hans, []), I2) .
6. I2 = 1 + I3 = succ(I3)
7. count_length([], I3)
8. count_length([], 0) .
9. I3 = 0

Studenten = { martin, anna, hans }

In Prolog:

Studenten = .(martin, .(anna, .(hans, []))).

0. count_length(Studenten, N) .
1. count_length(. (martin, . (anna, . (hans, []))) , N) .
2. N = 1 + I1 = succ(I1)
3. count_length(. (anna, . (hans, [])) , I1) .
4. I1 = 1 + I2 = succ(I2)
5. count_length(. (hans, []), I2) .
6. I2 = 1 + I3 = succ(I3)
7. count_length([], I3)
8. count_length([], 0) .
9. I3 = 0
10. I2 = 1

Studenten = { martin, anna, hans }

In Prolog:

Studenten = .(martin, .(anna, .(hans, []))).

0. count_length(Studenten, N).
1. count_length(. (martin, . (anna, . (hans, []))) , N).
2. N = 1 + I1 = succ(I1)
3. count_length(. (anna, . (hans, [])) , I1).
4. I1 = 1 + I2 = succ(I2)
5. count_length(. (hans, []), I2).
6. I2 = 1 + I3 = succ(I3)
7. count_length([], I3)
8. count_length([], 0).
9. I3 = 0
10. I2 = 1
11. I1 = 2

Studenten = { martin, anna, hans }

In Prolog:

Studenten = .(martin, .(anna, .(hans, []))).

0. count_length(Studenten, N) .
1. count_length(. (martin, . (anna, . (hans, []))) , N) .
2. N = 1 + I1 = succ(I1)
3. count_length(. (anna, . (hans, [])) , I1) .
4. I1 = 1 + I2 = succ(I2)
5. count_length(. (hans, []), I2) .
6. I2 = 1 + I3 = succ(I3)
7. count_length([], I3)
8. count_length([], 0) .
9. I3 = 0
10. I2 = 1
11. I1 = 2
12. N = 3

Aufgaben

1. Definiere ein Predikat **länger_than/2**, das als Argumente zwei Listen nimmt und entscheidet ob die erste länger als die zweite ist.
(Tipp: ähnlich wie **greater_than**)
2. Definiere ein Predikat **append/3**, das drei Listen als Argumente nimmt und die Konkatenation der ersten zwei Listen in dem dritten Argument speichert. Zum Beispiel bei der Anfrage

```
?- append(.(a,.(b,[ ])),.(1,.(2,[ ])),X)
```

soll Prolog soll $X = .(a,.(b,.(1,.(2,[]))))$ antworten.
(Tipp: ähnlich wie **add**)
3. Definiere ein Predikat **member/2**, das als Argumente ein Term und eine Liste nimmt und testet ob der Term in der Liste enthalten ist.

Alternative Darstellung für Listen

?- .(a, .(b,[])) = X.

X = [a,b]

Listen können in Prolog ähnlich wie Mengen geschrieben werden:

[a,b,c,[1]]

[1, 2, [*student(hans), r*], *f*, *g(a)*, []]

[[[]]]

[*a*, *a*, *a*]

Prolog stellt eine benutzerfreundlichere Darstellung für Listen zur Verfügung, die wie folgt definiert ist:

1. `[]` ist eine Liste
2. `[H|T]` ist eine Liste wenn T eine Liste ist.

Zugriff auf Listen

$[a,b,c] = [X|T]$

$X = a$

$T = [b,c]$

$[g(a)] = [X|Y]$

$X = g(a)$

$Y = []$

$[a,b,c] = [X,Y|T]$

$X = a$

$Y = b$

$T = [c]$

$[] = [X|T]$

no

Die anonyme Variable _

- Jedes vorkommen dieser Variable ist unabhängig von den anderen Vorkommen, d.h jedes mal das diese Variable vorkommt wird sie neu instanziiert.
- Die Bindungen dieser Variable sind unsichtbar, d.h. sie werden nicht gespeichert und werden auch nicht von Prolog ausgegeben.

?- f(a,b,c)=f(X,b,X)

no

?- f(a,b,c)=f(_,b,_)

yes

?- [a,b,c,d] = [H|T]

H = a

T = [b,c,d]

?- [a,b,c,d] = [H|_]

H = a

Member

```
/*
```

```
Das Predikat member/2 testet ob ein Objekt  
sich in einer Liste befindet
```

```
*/
```

```
% Basisklausel
```

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

```
% Recursive Klausel
```

```
member(X, [Y|T] :- member(X,T).
```

Append

```
/*
```

```
Das Predikat append/3 konkateniert zwei  
Listen
```

```
*/
```

```
% Basisklausel
```

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

```
% Recursive Klausel
```

```
append([X|T],Y,[X|C]) :- append(T,Y,C).
```

Zusammenfassung

- Listen sind rekursive Datenstrukturen mit denen geordnete Mengen von Objekten dargestellt werden können
- Anonyme variable

Nächste Woche: Arithmetik und mehr zu Listen.

Übungsaufgaben: Das Übungsblatt ist auf der Web-Seite.