

Mehr Listen und noch ein bisschen Arithmetik

- `append` und was man damit machen kann
- Arithmetik in Prolog
- Restrekursive Prädikate

Listen konkatenieren

```
?- append([1,2,3],[a,b,c],X).
```

```
X = [1,2,3,a,b,c]
```

```
yes
```

```
?- append([a,[foo,gibble],c],[1,[[],b]],X).
```

```
X = [a, [foo, gibble], c, 1, [[], b]]
```

```
yes
```

```
?- append([],[x,y],X).
```

```
X = [x,y]
```

```
yes
```

Definition von append/3 (1)

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

```
append( [H|T], L2, [H|L3] ) :-  
    append( T, L2, L3 ).
```

Definition von append/3 (1)

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

`append([H | T] , L2 , [H | L3]) :-
append(T , L2 , L3) .`

Input: **H** **T** + **L2**

What L3 is: **L3 = T + L2**

Result: **H** **L3**

Definition von append/3 (2)

2 Versionen:

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

```
append( [ H | T ] , L2 , [ H | L3 ] ) :-  
    append( T , L2 , L3 ) .
```

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

```
append( [ H | T ] , L2 , Result ) :-  
    append( T , L2 , L3 ) ,  
    Result = [ H | L3 ] .
```

Verwendung von append

- zum Konkatenieren von Listen (append(+,+,-))

```
append([a,b,c],[x,y,z],L)
```

- zum Testen, ob eine Liste die Konkatenation von zwei anderen Listen ist (append(+,+,+))

```
append([[a,b,c],[x,y,z],[a,b,c,x,y,z])
```

- zum Zerlegen von Listen (append(-,-,+), append(-,+,+) append(+,-,+))

```
append(X,Y,[a,b,c])
```

```
append(X,[b,c,d],[a,b,c,d])
```

```
append([a,b],X,[a,b,c,d])
```

Suffixe und Präfixe

Präfixe der Liste [a,b,c,d]:

[], [a], [a,b], [a,b,c], [a,b,c,d]

Suffixe der Liste [a,b,c,d]:

[], [d], [c,d], [b,c,d], [a,b,c,d]

Suffixe und Präfixe

Präfixe der Liste [a,b,c,d]:

[], [a], [a,b], [a,b,c], [a,b,c,d]

`prefix(P,L) :- append(P,_,L).`

Suffixe der Liste [a,b,c,d]:

[], [d], [c,d], [b,c,d], [a,b,c,d]

Suffixe und Präfixe

Präfixe der Liste [a,b,c,d]:

[], [a], [a,b], [a,b,c], [a,b,c,d]

`prefix(P,L) :- append(P,_,L).`

Suffixe der Liste [a,b,c,d]:

[], [d], [c,d], [b,c,d], [a,b,c,d]

`suffix(S,L) :- append(_,S,L).`

Sublisten

Sublisten der Liste [a,b,c]:

[], [a], [a,b], [a,b,c], [b], [b,c], [c]

Sublisten

Sublisten der Liste [a,b,c]:

[], [a], [a,b], [a,b,c], [b], [b,c], [c]

List:



Take suffix:



Take prefix to get sublist:



Sublisten

Sublisten der Liste `[a,b,c]`:

`[], [a], [a,b], [a,b,c], [b], [b,c], [c]`

List:



Take suffix:



Take prefix to get sublist:



```
sublist(SubL,L) :- suffix(S,L), prefix(SubL,S).
```

Listen umdrehen (naive Version)

```
?- reverse1([a,b,c],RevL).
```

```
RevL = [c,b,a]
```

```
yes
```

```
?- reverse1([],RevL).
```

```
RevL = []
```

Listen umdrehen (naive Version)

```
?- reverse1([a,b,c],RevL).
```

```
RevL = [c,b,a]
```

```
yes
```

```
?- reverse1([],RevL).
```

```
RevL = []
```

```
reverse1([],[]).
```

```
reverse1([H|T],R) :-  
    reverse1(T,RevT),  
    append(RevT,[H],R).
```

Warnung

Mit `append` kann man eine Menge machen, **aber:**

- Bei jedem Aufruf von `append` muss die Liste im ersten Argument komplett abgearbeitet werden.
- Dadurch können Programme, die viele Aufrufe von `append` enthalten, schnell ineffizient werden.
- Insbesondere ist es gefährlich, `append` in einem rekursiven Prädikat aufzurufen, wie z.B. in `reverse1`.

Arithmetik in Prolog

Arithmetik

$$8 + 2 = x$$

$$6 - 3 = x$$

$$6 * 2 = x$$

$$6/3 = x$$

Prolog

`x is 8 + 2`

`x is 6 - 2`

`x is 6 * 2`

`x is 6 / 3`

Ein paar Beispiele

```
?- X is 8 + 2.
```

```
X = 10
```

```
?- 12 is 6 * 2.
```

```
yes
```

```
?- 5 is 6 - 2.
```

```
no
```

```
?- X is 7 / 2.
```

```
X = 3.5
```

```
?- X is 2 * 3 + 1.
```

```
X = 7
```

```
?- X is 2 * (3 + 1).
```

```
X = 8
```

Prolog kennt die üblichen Konventionen zum Desambiguieren von arithmetischen Ausdrücken (Punkt- vor Strichrechnung).

Arithmetische Operationen in Prädikaten

```
add_3_and_double(X,Result) :- Result is (X + 3) * 2.
```

Arithmetische Ausdrücke sind Terme

- **Achtung!** In Prolog gibt es keine Funktionen. D.h. `+`, z.B., ist keine Funktion.
- `2 + 4` ist einfach ein komplexer Term (interne Darstellung: `+(2, 4)`).
- Wir brauchen das eingebaute Prädikat `is`, um diesen Term als arithmetischen Ausdruck zu interpretieren und den Wert zu berechnen.

'=' vs. 'is'

?- X = 2 * 5.

?- X is *(2,5).

'=' vs. 'is'

?- X = 2 * 5.

X = 2 * 5

?- X is *(2,5).

'=' vs. 'is'

```
?- X = 2 * 5.
```

```
X = 2 * 5
```

```
?- X is *(2,5).
```

```
X = 10
```

'=' vs. 'is'

```
?- X = 2 * 5.
```

```
X = 2 * 5
```

```
?- X is *(2,5).
```

```
X = 10
```

= matcht zwei beliebige Prolog-Terme.

is sagt Prolog, dass der Ausdruck, der rechts steht, ein arithmetischer Ausdruck ist und Prolog ihn auswerten soll. Das Ergebnis wird mit dem Term links von is gematcht.

Einschränkungen

- Nur der Ausdruck rechts von `is` wird ausgewertet.

```
?- 8 is 6 + 2.
```

```
yes
```

```
?- 6 + 2 is 8.
```

```
no
```

```
?- 6 + 2 is 6 + 2.
```

```
no
```

- Alle Variablen rechts von `is` müssen zum Zeitpunkt der Auswertung instantiiert sein.

```
?- 8 is 6 + X.
```

```
ERROR: Arguments are not sufficiently  
instantiated
```


Vergleichende Operatoren

Arithmetik

$$x < y$$

$$x > y$$

$$x \leq y$$

$$x \geq y$$

$$x = y$$

$$x \neq y$$

Prolog

$$X < Y$$

$$X > Y$$

$$X = < Y$$

$$X > = Y$$

$$X = : = Y$$

$$X = \backslash = Y$$

Beispiele

?- 4 < 7.

?- 7 >= 7.

?- 4 ::= 2 + 2.

?- 3 + 1 =\= 5.

?- X = 4, X < 7.

?- X is 3 + 4, X >= 7.

?- X ::= 3 + 4.

Beispiele

?- 4 < 7.

yes

?- 4 == 2 + 2.

?- X = 4, X < 7.

?- X == 3 + 4.

?- 7 >= 7.

?- 3 + 1 \= 5.

?- X is 3 + 4, X >= 7.

Beispiele

?- 4 < 7.

yes

?- 4 ::= 2 + 2.

?- X = 4, X < 7.

?- X ::= 3 + 4.

?- 7 >= 7.

yes

?- 3 + 1 =\= 5.

?- X is 3 + 4, X >= 7.

Beispiele

?- 4 < 7.

yes

?- 4 ::= 2 + 2.

yes

?- X = 4, X < 7.

?- 7 >= 7.

yes

?- 3 + 1 =\= 5.

?- X is 3 + 4, X >= 7.

?- X ::= 3 + 4.

Beispiele

?- 4 < 7.

yes

?- 4 ::= 2 + 2.

yes

?- X = 4, X < 7.

?- 7 >= 7.

yes

?- 3 + 1 =\= 5.

yes

?- X is 3 + 4, X >= 7.

?- X ::= 3 + 4.

Beispiele

?- 4 < 7.

yes

?- 4 == 2 + 2.

yes

?- X = 4, X < 7.

X = 4

yes

?- X == 3 + 4.

?- 7 >= 7.

yes

?- 3 + 1 \= 5.

yes

?- X is 3 + 4, X >= 7.

Beispiele

?- 4 < 7.

yes

?- 4 == 2 + 2.

yes

?- X = 4, X < 7.

X = 4

yes

?- X == 3 + 4.

?- 7 >= 7.

yes

?- 3 + 1 \= 5.

yes

?- X is 3 + 4, X >= 7.

X = 7

yes

Beispiele

?- 4 < 7.

yes

?- 4 == 2 + 2.

yes

?- X = 4, X < 7.

X = 4

yes

?- X == 3 + 4.

ERROR: Arguments are not sufficiently instantiated

?- 7 >= 7.

yes

?- 3 + 1 =\= 5.

yes

?- X is 3 + 4, X >= 7.

X = 7

yes

= vs. is vs. ==

= / 2

Nimmt zwei beliebige Terme als Argumente. Testet, ob sie matchen und macht ggf. die nötigen Variableninstanziierungen.

is / 2

1. Argument: Variable oder Zahl; 2. Argument: ein arithmetische Ausdruck ohne uninstanzierte Variablen.

Berechnet den Wert des arithmetischen Ausdrucks und matcht ihn mit dem 1. Argument.

== / 2

Beide Argumente sind Zahlen oder arithmetische Ausdrücke ohne uninstanzierte Variablen. Prolog berechnet den Wert der arithmetischen Ausdrücke und testet, ob sie gleich sind.

max/2 (1)

```
?- max([2,5,4],X).
```

```
X = 5
```

```
yes
```

```
?- max([],X).
```

```
X = 0
```

```
yes
```

Basisfall: Wenn die Liste leer ist, dann gib 0 zurück.

Rekursion: Wenn die Liste nicht leer ist, dann berechne das Maximum des Tails MT. Wenn der Kopf größer als MT ist, dann gib den Kopf zurück.

Wenn der Kopf kleiner/gleich MT ist, dann gib MT zurück.

max/2 (1)

```
?- max([2,5,4],X).
```

```
X = 5
```

```
yes
```

```
?- max([],X).
```

```
X = 0
```

```
yes
```

Basisfall: Wenn die Liste leer ist, dann gib 0 zurück.

Rekursion: Wenn die Liste nicht leer ist, dann berechne das Maximum des Tails MT. Wenn der Kopf größer als MT ist, dann gib den Kopf zurück.

Wenn der Kopf kleiner/gleich MT ist, dann gib MT zurück.

max/2 (1)

`max([], 0).`

`max([H|T], H) :-
 max(T, MaxT),
 H > MaxT.`

`max([H|T], MaxT) :-
 max(T, MaxT),
 H =< MaxT.`

accMax/3

`accMax([], A, A) .`

`max([], 0) .`

`accMax([H | T], A, Max) :-`

`max([H | T], H) :-`

`H > A,`

`max(T, MaxT),`

`accMax(T, H, Max) .`

`H > MaxT .`

`accMax([H | T], A, Max) :-`

`max([H | T], MaxT) :-`

`H =< A,`

`max(T, MaxT),`

`accMax(T, A, Max) .`

`H =< MaxT .`

- Das zweite Argument ist ein Akkumulator, in dem Zwischenergebnisse gespeichert werden.
- Am Anfang initialisieren wir diesen Akkumulator mit 0.

?- `accMax([1, 2, 3], 0, Result)`

Akkumulatoren und Rest-Rekursion

- Akkumulatoren speichern Zwischenergebnisse.
- Wenn wir bei der Basisklausel angekommen sind, enthält der Akkumulator das Endergebnis.
- Ein Prädikat ist **restrekursiv** (oder tail recursive), wenn nach dem Erreichen der Basisklausel keine neuen Prädikatsaufrufe mehr folgen.
- Gute Prologimplementationen können restrekursive Prädikate effizienter verarbeiten als andere rekursive Prädikate.

Aufgaben

1. Hier ist eine Definition des Prädikats `length/2`, das die Länge einer Liste berechnet.

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

```
length([_|T], Length) :-
```

```
    length(T, TLength),
```

```
    Length is TLength + 1.
```

Schreibt eine restrekursive Variante `length/3`.

2. Schreibt eine restrekursive Variante des Prädikats `reverse1` zum Umdrehen von Listen.

len/2 und accLen/3

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

```
len([_|T],N) :-  
    len(T,X),  
    N is X+1.
```

len/2 und accLen/3

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

```
len(_|T,N) :-  
    len(T,X),  
    N is X+1.
```

```
accLen([],A,A).
```

```
accLen(_|T,A,L) :-  
    Anew is A+1,  
    accLen(T,Anew,L).
```

Listen umdrehen (effizienter)

```
accRev([ ],A,A).
```

```
accRev([H|T],A,R) :- accRev(T,[H|A],R).
```

Zusammenfassung

Heute haben wir gesehen,

- wie man in Prolog (mit richtigen Zahlen) rechnen kann,
- das man mit `append` eine Menge machen kann, aber vorsichtig sein muss, weil es schnell ineffizient wird, und
- was rest-rekursive Prädikate sind.

Nächste Woche: Klausur. Und danach: Context-freie Grammatiken in Prolog.

Übungsaufgaben: Das Übungsblatt ist auf der Web-Seite. Ihr müsst es erst am Freitag in zwei Wochen abgeben.