

Prolog

5. Kapitel: Arithmetik

Dozentin: Wiebke Petersen

Kursgrundlage: Learn Prolog Now (Blackburn, Bos, Striegnitz)

Zusammenfassung Kapitel 4

- Wir haben Listen als mächtige Datenstrukturen in Prolog kennengelernt und mithilfe des Listenkonstruktors `|` dekonstruiert.
- Wir haben gelernt, Prädikate zu definieren, die Listen rekursiv verarbeiten und das wichtige Prädikat `member/2` kennengelernt.
- Wir haben die anonyme Variable `_` kennengelernt.
- **Keywords:** Listenkonstruktor, Kopf (Head), Restliste (Tail), rekursive Listenverarbeitung, `member/2`, anonyme Variable.
- **Wichtig:** Die rekursive Verarbeitung von Listen ist eine zentrale Programmiertechnik in Prolog.
- **Ausblick Kapitel 5:** Arithmetik

Arithmetik in Prolog

- Die meisten Prologimplementierungen stellen Operatoren zur Verarbeitung von Zahlen zur Verfügung.
- Hierzu gehören die arithmetischen Operatoren `+` (Addition), `-` (Subtraktion), `*` (Multiplikation), `/` (Division), `//` (ganzzahlige Division), `mod` (modulo) und `^` (Exponent).
- Alle Operatoren können auch als Funktoren verwendet werden: Statt `3+4` kann man auch `+(3,4)` schreiben.
- Die verwendeten Symbole für die Operatoren hängen von dem jeweiligen Prolog-Interpreter ab (hier angegeben für SWI-Prolog).

Vorsicht: Arithmetische Operationen gehören nicht zu den Kernkonzepten von Prolog. Mit ihnen verlässt man das auf Unifikation basierende Grundprinzip der deklarativen Programmierung.

Rechnen in Prolog

```
?- X is 3+4.
```

```
X = 7.
```

```
?- X is 3*4.
```

```
X = 12.
```

```
?- X is 3/4.
```

```
X = 0.75.
```

```
?- X is 13 mod 5.
```

```
X = 3.
```

% Prolog beherrscht Punkt- vor Strichrechnung:

```
?- X is 3+4*5.
```

```
X = 23.
```

% Klammern koennen wie ueblich verwendet werden:

```
?- X is (3+4)*5.
```

```
X = 35.
```

Arithmetische Operatoren und die Evaluation

- Arithmetische Ausdrücke werden in Prolog nicht evaluiert bzw. ausgewertet, sondern sind gewöhnliche zusammengesetzte Terme.

```
?- X = 2 + 3.  
X = 2+3.  
?- 2+3 = 2+3.  
true.  
?- 2+3 = +(2,3).  
true.
```

- Um arithmetische Ausdrücke in Prolog zu berechnen benötigt man den Infix-Operator `is`.

```
?- X is 2 + 3.  
X = 5  
?- is(X,2*3).  
X = 6.
```

Der Evaluationsoperator `is/2`

Vorsicht, da der Evaluationsoperator `is/2` außerhalb der normalen Programmlogik von Prolog steht, stellt er besondere Ansprüche:

- Der Evaluationsoperator `is/2` erzwingt die sofortige Auswertung des zweiten Arguments,
- daher muss das zweite Argument ein evaluierbarer arithmetischer Ausdruck sein:

```
?- X is 3+5.  
X = 8.  
?- 3+5 is X.  
ERROR: is/2: Arguments are not sufficiently instantiated  
?- X is 4+Y.  
ERROR: is/2: Arguments are not sufficiently instantiated  
?- X is a.  
ERROR: Arithmetic: 'a' is not a function
```

- Ist das zweite Argument nicht evaluierbar, so bricht Prolog mit einer Fehlermeldung ab.

► Übung

Vergleich `is/2` mit normalen Prologprädikaten

Der Evaluationsoperator `is/2` unterscheidet sich grundlegend von “normalen” Prologprädikaten wie `member/2`.

Werden “normale” Prologprädikate “falsch” instantiiert, kommt es zu keinem Programmabbruch. Die Aussage kann lediglich nicht bewiesen werden:

```
?- member(a,b).  
false.  
?- member([a,b],a).  
false.  
?- X is a.  
ERROR: Arithmetic: 'a' is not a function
```

arithmetische Vergleichsoperatoren

Neben dem Evaluationsoperator `is/2` gibt es weitere Operatoren, die das Evaluieren arithmetischer Ausdrücke erzwingen.

Die zweistelligen **Vergleichsoperatoren** `<` (kleiner), `=<` (kleiner gleich), `>` (größer), `>=` (größer gleich), `==` (gleich) und `\=` (ungleich) erzwingen die sofortige Evaluation beider Argumente.

```
?- 1+4 < 3*5.
```

```
true.
```

```
?- 1+7 =< 3*2.
```

```
false.
```

```
?- 1+3 == 2*2.
```

```
true.
```

```
?- 1+3 \= 2*3.
```

```
true.
```

```
?- X < 3.
```

```
ERROR: </2: Arguments are not sufficiently instantiated
```

```
?- 3 == 2+X.
```

```
ERROR: ==/2: Arguments are not sufficiently instantiated
```

Evaluation erzwingende Operatoren in Prädikatsdefinitionen

Evaluation erzwingende Operatoren können in Prädikatsdefinitionen eingesetzt werden.

Allerdings muss sichergestellt werden, dass beim Aufruf des Prädikats die zu evaluierenden Ausdrücke vollständig instantiiert sind.

```
% Definition
double_and_add3(X,Y):- Y is 2*X + 3.

% Aufrufe:
-? double_and_add3(3,9).
true.

-? double_and_add3(4,Y).
Y=11.

?- double_and_add3(X,11).
ERROR: is/2: Arguments are not sufficiently instantiated
```

Quiz-Time



Listenlänge bestimmen ohne Akkumulator

Die Länge einer Liste ist die Anzahl ihrer Elemente. Z.B. hat die Liste `[a,b,b,a]` die Länge 4.

rekursive Längendefinition

- ① Die leere Liste hat die Länge 0.
- ② Eine nichtleere Liste hat eine Länge, die um 1 höher ist als die Länge ihres Tails.

```
% len1/2
% len1(List, Length)
len1([],0).
len1([_|T],N):-
    len1(T,X),
    N is X+1.
```

```
?- len1([a,[b,e,[f,g]],food(cheese),X],4).
true.
?- len1([a,b,a],X).
X=3.
```

trace: Listenlänge ohne Akkumulator

Prädikatsdefinition:

```
% len1/2
% len1(List, Length)
len1([],0).
len1([_|T],N):-
    len1(T,X),
    N is X+1.
```

trace einer Beispielanfrage:

```
?- len1([a,a,a],Len).
Call: (7) len1([a,a,a], _X1) ?
Call: (8) len1([a,a], _X2) ?
Call: (9) len1([a], _X3) ?
Call: (10) len1([], _X4) ?
Exit: (10) len1([], 0) ?
Call: (10) _X3 is 0+1 ?
Exit: (10) 1 is 0+1 ?
Exit: (9) len1([a], 1) ?
Call: (9) _X2 is 1+1 ?
Exit: (9) 2 is 1+1 ?
Exit: (8) len1([a,a], 2) ?
Call: (8) _X1 is 2+1 ?
Exit: (8) 3 is 2+1 ?
Exit: (7) len1([a,a,a], 3) ?
Len = 3.
```

Listenlänge bestimmen mit Akkumulator

- **Akkumulatoren** (*accumulators*) dienen dem Aufsammeln von Zwischenergebnissen.
- Akkumulatoren ermöglichen eine effizientere Implementierung in Prolog, da Variablen früher instantiiert werden können.
- Rekursive Programmierung mit Akkumulatoren zählt zu den zentralen Programmier-techniken in Prolog.

```
% len2/2
% len2(List, Length)
len2(List,Length):- accLen(List,0,Length).

% accLen/3
% accLen(List,Accumulator,Length)
accLen([_|T],Acc,Length):-
    NewAcc is Acc+1,
    accLen(T,NewAcc,Length).
accLen([],Length,Length).
```

[▶ Übung: len/2](#)[▶ Übung: gleichlange Listen](#)

trace: Listenlänge mit Akkumulator

Prädikatsdefinition:

```
% len2/2
% len2(List, Length)
len2(List,Length):-
    accLen(List,0,Length).

% accLen/3
% accLen(List,Acc,Length)
accLen([_|T],Acc,L):-
    NewAcc is Acc+1,
    accLen(T,NewAcc,L).
accLen([],Acc,Acc).
```

trace einer Beispielanfrage:

```
?- len2([a,a,a],Len).
Call: (7) len2([a,a,a], _X1) ?
Call: (8) accLen([a,a,a], 0, _X1) ?
Call: (9) _X is 0+1 ?
Exit: (9) 1 is 0+1 ?
Call: (9) accLen([a,a], 1, _X1) ?
Call: (10) _X is 1+1 ?
Exit: (10) 2 is 1+1 ?
Call: (10) accLen([a], 2, _X1) ?
Call: (11) _X is 2+1 ?
Exit: (11) 3 is 2+1 ?
Call: (11) accLen([], 3, _X1) ?
Exit: (11) accLen([], 3, 3) ?
Exit: (10) accLen([a], 2, 3) ?
Exit: (9) accLen([a,a], 1, 3) ?
Exit: (8) accLen([a,a,a], 0, 3) ?
Exit: (7) len2([a,a,a], 3) ?
Len = 3.
```

Vergleich Länge mit und ohne Akkumulator

ohne Akkumulator:

```
?- len1([a,a,a],Len).  
Call: (7) len1([a,a,a], _X1) ?  
Call: (8) len1([a,a], _X2) ?  
Call: (9) len1([a], _X3) ?  
Call: (10) len1([], _X4) ?  
Exit: (10) len1([], 0) ?  
Call: (10) _X3 is 0+1 ?  
Exit: (10) 1 is 0+1 ?  
Exit: (9) len1([a], 1) ?  
Call: (9) _X2 is 1+1 ?  
Exit: (9) 2 is 1+1 ?  
Exit: (8) len1([a,a], 2) ?  
Call: (8) _X1 is 2+1 ?  
Exit: (8) 3 is 2+1 ?  
Exit: (7) len1([a,a,a], 3) ?  
Len = 3.
```

mit Akkumulator

```
?- len2([a,a,a],Len).  
Call: (7) len2([a,a,a], _X1) ?  
Call: (8) accLen([a,a,a], 0, _X1) ?  
Call: (9) _X is 0+1 ?  
Exit: (9) 1 is 0+1 ?  
Call: (9) accLen([a,a], 1, _X1) ?  
Call: (10) _X is 1+1 ?  
Exit: (10) 2 is 1+1 ?  
Call: (10) accLen([a], 2, _X1) ?  
Call: (11) _X is 2+1 ?  
Exit: (11) 3 is 2+1 ?  
Call: (11) accLen([], 3, _X1) ?  
Exit: (11) accLen([], 3, 3) ?  
Exit: (10) accLen([a], 2, 3) ?  
Exit: (9) accLen([a,a], 1, 3) ?  
Exit: (8) accLen([a,a,a], 0, 3) ?  
Exit: (7) len2([a,a,a], 3) ?  
Len = 3.
```

maximales Listenelement bestimmen mit Akkumulator

```
1 % max1/2
2 % max1(List,ListMax)
3 max1([H|T],Max) :-
4     accMax(T,H,Max).
5
6 % accMax/3
7 % accMax(List,Accum.,ListMax)
8 accMax([H|T],Acc,Max) :-
9     H > Acc,
10    accMax(T,H,Max).
11
12 accMax([H|T],Acc,Max) :-
13     H =< Acc,
14     accMax(T,Acc,Max).
15
16 accMax([],Max,Max).
```

Grundidee: Die Liste wird von vorne nach hinten rekursiv aufgespalten. Die Variable `Acc` fasst das jeweils bis dato höchste Listenelement.

Zeile 4: Zu Beginn ist der Kopf der Liste das höchste bis dato gesehene Listenelement.

Zeile 8-10: Ist der Kopf der aktuellen Liste größer als das bisherige Maximum, das im Akkumulator gespeichert ist, wird der Akkumulator durch den Kopf ersetzt.

Zeile 12-14: Ist der Kopf der aktuellen Liste nicht größer als das bisherige Maximum, das im Akkumulator gespeichert ist, bleibt der Akkumulator erhalten.

Zeile 16: Ist die Liste abgearbeitet, speichert der Akkumulator das maximale Listenelement.

maximales Listenelement bestimmen ohne Akkumulator

```
1 % max2/2 bestimmt das maximale
2 % Listenelement einer Liste
3 % mit nicht negativen Zahlen
4 % max2(List,ListMax)
5
6 max2([],0).
7
8 max2([H|T],H):-
9     max2(T,MaxT),
10    H>MaxT.
11
12 max2([H|T],MaxT):-
13     max2(T,MaxT),
14     H=<MaxT.
```

Zeile 6: Per Definition erklären wir, dass die leere Liste das Maximum 0 hat.

Zeile 8-10: Der Kopf einer Liste ist das Maximum der gesamten Liste, wenn er größer ist als das Maximum der Restliste.

Zeile 12-14: Ist der Kopf der Liste nicht größer als das Maximum der Restliste, dann ist das Maximum der Restliste das Maximum der gesamten Liste.

Experimentieren Sie mit dem Prädikat im Tracemodus: [► Übung](#)

Akkumulatoren: Struktur der Programme

Listenverarbeitung ohne Akkumulator

- Die eigentliche Verarbeitung beginnt am tiefsten Punkt der Rekursion.
- Die initiale Instanziierung der Lösungsvariable erfolgt am tiefsten Punkt der Rekursion.
- In jedem Schritt aus der Rekursion heraus erfolgt ein Verarbeitungsschritt.

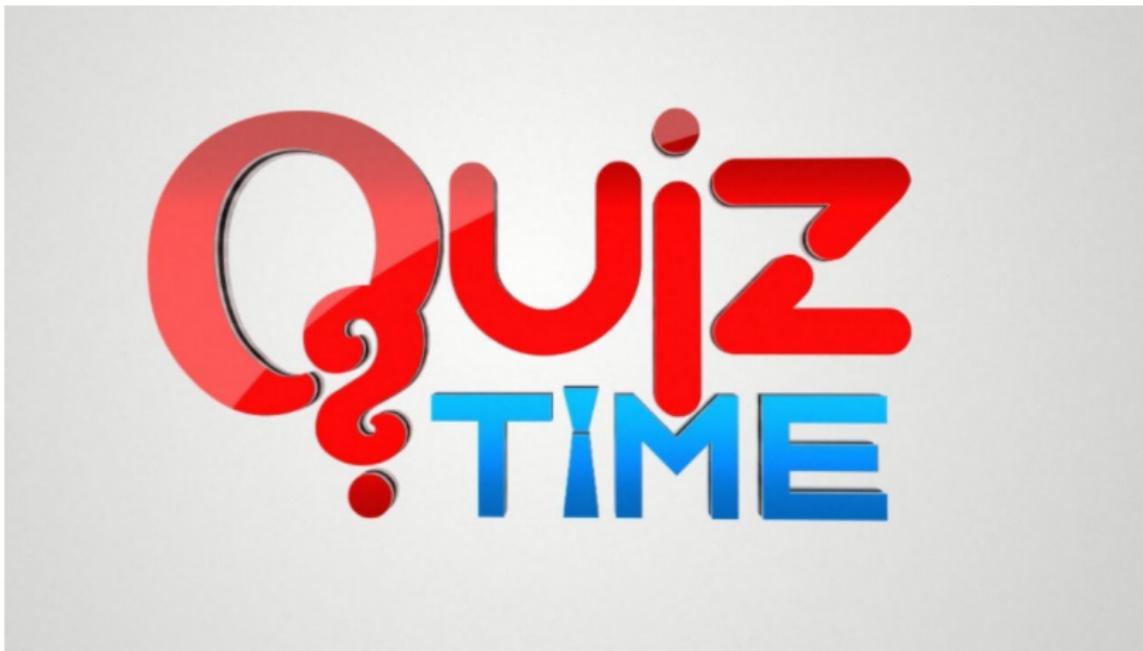
```
p([],Initial).  
  
p([H|T],Sol):-  
    p(T,NewSol),  
    ...,  
    Sol is ... NewSol ...,  
    ...  
    .
```

Listenverarbeitung mit Akkumulator

- Die Instanziierung der Akkumulatorvariable erfolgt beim ersten Aufruf.
- Am tiefsten Punkt der Rekursion wird die Lösungsvariable mit dem Akkumulator unifiziert.
- In jedem Schritt in die Rekursion hinein erfolgt ein Verarbeitungsschritt.

```
p(List,Sol):- p(List,InitialAcc,Sol).  
  
p([H|T],Acc,Sol):-  
    NewAcc is ... Acc ...,  
    ...,  
    p(T,NewAcc,Sol),  
    ...  
    .  
  
p([],Sol,Sol).
```

Quiz-Time



Zusammenfassung Kapitel 5

- Wir haben gesehen, wie wir mit Prolog rechnen können.
- Wir haben arithmetische Vergleichsoperatoren kennengelernt.
- Wir haben gelernt, wie Akkumulatoren in der rekursiven Listenverarbeitung eingesetzt werden können, um effizienter Prädikate zu erhalten.
- **Keywords:** Rechnen in Prolog mit dem Evaluationsoperator `is`, arithmetische Vergleichsoperatoren, Akkumulatoren.
- **Wichtig:** Die rekursive Verarbeitung von Listen mit Akkumulatoren ist eine zentrale Programmiertechnik in Prolog.
- **Vorsicht:** Die arithmetischen Vergleichsoperatoren und der Operator `is` fordern zwingend sofort evaluierbare Terme. Uninstantiierte Terme führen zu einem Abbruch mit Fehlermeldung.
- **Ausblick Kapitel 6:** Weitere Listenprädikate

Übung: arithmetische Operationen

Was antwortet Prolog auf folgende Anfragen?

```
1  ?- X = 3*4.
2  ?- X is 3*4.
3  ?- 4 is X.
4  ?- X = Y.
5  ?- 3 is 1+2.
6  ?- 3 is +(1,2).
7  ?- 3 is X+2.
8  ?- X is 1+2.
9  ?- 1+2 is 1+2.
10 ?- is(X,+(1,2)).
11 ?- 3+2 = +(3,2).
12 ?- *(7,5) = 7*5.
13 ?- *(7,+(3,2)) = 7*(3+2).
14 ?- *(7,(3+2)) = 7*(3+2).
15 ?- *(7,(3+2)) = 7*(+(3,2)).
```

▶ zurück

Übung: Prädikate mit arithmetischen Operationen

- 1 Schreiben Sie ein 3-stelliges Prädikat `produkt/3`, das wahr ist, wenn dessen drittes Argument das Produkt der ersten beiden ist.

```
?- produkt(2,4,8).  
true.  
?- produkt(2,4,6).  
false.  
?- produkt(3,4,X).  
X=12.
```

- 2 Schreiben Sie ein 2-stelliges Prädikat `nachfolger/2`, das wahr ist, wenn das zweite Argument um 1 größer ist als das erste.

```
?- nachfolger(2,3).  
true.  
?- nachfolger(2,1).  
false.  
?- nachfolger(3,X).  
X=4.
```

Übung: Listenlänge bestimmen

Warum führen die folgenden beiden Prädikate bei der Anfrage `?- len1([a,b,c],L)` bzw. `?- len2([a,b,c],L)` zu einem Abbruch? Schauen sie sich den Trace an.

```
% ohne Akkumulator:
len1([],0).
len1(_|T,N):-
    N is X+1,
    len1(T,X).

% mit Akkumulator:
len2(List,Int):-
    accLen(List,0,Int).

accLen([],Acc,Acc).
accLen(_|T,Acc,L):-
    accLen(T,NewAcc,L),
    NewAcc is Acc+1.
```

▶ zurück

Übung: Prädikate für den Vergleich von Listenlängen

Schreiben Sie ein Prädikat `sameLength/2` das zwei Listen akzeptiert, wenn sie dieselbe Länge haben.

- Verwenden Sie für das Prädikat zunächst das Prädikat `len2/2`.
- Versuchen Sie anschließend auf die Verwendung von `len2/2` oder andere arithmetische Prädikate zu verzichten.

Welche Ihrer Prädikatsversionen ist effizienter?

Definieren Sie folgende Listenprädikate:

- `shorter/2`: gelingt wenn die erste Liste kürzer ist als die zweite;
- `longer/2`: gelingt wenn die erste Liste länger ist als die zweite;

▶ zurück

Übung: maximales Listenelement bestimmen

- Bestimmen Sie mit den beiden Prädikaten `max1/2` (mit Akkumulator) und `max2/2` (ohne Akkumulator) die maximalen Listenelemente der Listen `[1,4,9]` und `[9,4,1]` im Tracemodus. Was fällt Ihnen auf?
- Das Prädikat `max2/2` (ohne Akkumulator) funktioniert nur für Listen positiver Zahlen. Können Sie es so reparieren, dass es auch für Listen wie `[-5,-3,-7]` die korrekte Antwort liefert?

▶ zurück

Übung: Listenelemente verdoppeln

Schreiben Sie ein Prädikat `double_elements/2`, das gelingt, wenn beide Argumente Listen von Zahlen sind und die zweite Liste genau aus den verdoppelten Zahlen der ersten Liste besteht.

```
?- double_elements([1,4,3],[2,8,6]).  
true.  
?- double_elements([3,4,1],[2,8,6]).  
false.  
?- double_elements([],[]).  
true.
```

Skalarmultiplikation

Die Skalarmultiplikation ist eine Operation auf einem Skalar (einer Zahl) und einem Vektor. Das Ergebnis ist der Vektor, den man erhält, wenn man jedes Element des Inputvektors mit dem Skalar multipliziert. Beispiel:

$$4 * \begin{pmatrix} 3 \\ 5 \\ 6 \end{pmatrix} = \begin{pmatrix} 12 \\ 20 \\ 24 \end{pmatrix}$$

Schreibe ein dreistelliges Prädikat `scalarMult/3` mit `scalarMult(Skalar,Vektor,Resultatvektor)`, so dass `Resultatvektor` das Resultat der Sakalarmultiplikation von `Skalar` und `Vektor` ist.

```
?- scalarMult(3,[2,7,4],Result).  
Result = [6,21,12]
```

Skalarprodukt/Kreuzprodukt

Das Skalarprodukt ist eine Operation auf zwei gleichdimensionalen Vektoren. Das Ergebnis ist der Skalar, den man erhält, wenn man die einander entsprechenden Elemente der Vektoren miteinander multipliziert und die Ergebnisse aufaddiert. Beispiel:

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \times \begin{pmatrix} 3 \\ 4 \\ 2 \end{pmatrix} = 1 * 3 + 2 * 4 + 3 * 2 = 17$$

Schreibe ein dreistelliges Prädikat `dot/3` mit `dot(Vektor1,Vektor2,Resultat)`, so dass `Resultat` das Resultat des Skalarprodukts von `Vektor1` und `Vektor2` ist.

```
?- dot([2,5,6],[3,4,1],Result).  
Result = 32
```

Zusatzaufgaben

- 1 Schreiben Sie ein zweistelliges Prädikat `arit/2`, das gelingt, wenn das erste Argument eine Liste von Zahlen und das zweite das arithmetische Mittel der Zahlen ist (arithmetisches Mittel = Durchschnitt).
- 2 Schreiben Sie ein zweistelliges Prädikat `minimum/2`, das gelingt, wenn das erste Argument eine Liste von Zahlen und das zweite das Minimum der Liste ist.
- 3 Schreiben Sie ein zweistelliges Prädikat `gerade/2`, das gelingt, wenn das erste Argument eine Liste von Zahlen ist und das zweite Argument die Anzahl der geraden Zahlen ist, die in der Liste vorkommen.