

Kapitel 5.

Listenverarbeitung

Die Liste ist ein sehr flexibler Datentyp, der sich besonders auch für linguistische Anwendungen eignet. Allerdings können wir, mit Ausnahme des Kopfes, auf die Elemente einer Liste nicht direkt zugreifen. Zur Verarbeitung von Listen benötigen wir also eine Reihe von Prädikaten, die entweder als System- bzw. Bibliotheksprädikate zur Verfügung gestellt werden, oder die wir auf der Grundlage von elementaren vordefinierten Operationen selbst definieren können. Die Formulierung dieser Prädikate ist eigentlich gar nicht so schwer. Es ist aber wichtig, dafür immer im Hinterkopf zu behalten, wie die interne Struktur einer Liste aussieht, daß es sich bei Listen eben nicht um bloße Aneinanderreihungen von Elementen handelt, sondern um komplexe rekursive Strukturen.

5.1. Listenzerlegung

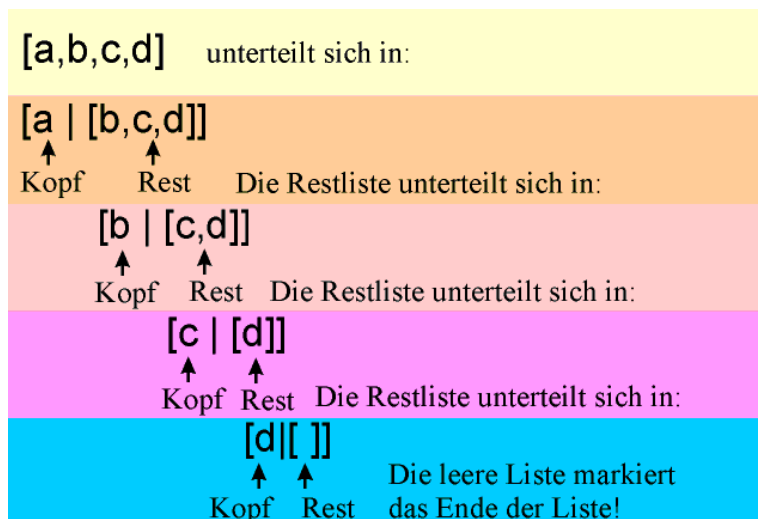
Die wichtigste Grundoperation, die allen anderen Listenoperationen zugrunde liegt, ist die Zerlegung einer Liste in den Listenkopf und den Listenrest. Dies geschieht sehr einfach mithilfe des vordefinierten Listenoperators '|'. Genauer gesagt: ist eine beliebige Liste L gegeben, muß diese Liste mit dem Listenstrukturmuster [Kopf | Rest] in Übereinstimmung gebracht werden: $L = [\text{Kopf} \mid \text{Rest}]$. Der Knackpunkt dabei ist, wie im letzten Kapitel deutlich wurde, die Tatsache, daß der Rest selber wieder eine Liste ist. Ist L beispielsweise die Liste [john, loves, mary], dann würde, da diese Liste mit Operator als [john | [loves, mary]] wiederzugeben wäre, die Variable Kopf mit john und die Variable Rest mit [loves, mary] unifiziert werden:

$$\begin{array}{l}
 [\text{Kopf} \mid \text{Rest}] \\
 = \\
 [\text{john} \mid [\text{loves, mary}]]
 \end{array}$$

Auf diesem Hintergrund und unter der Berücksichtigung der Tatsache, daß eine jede Liste stets mit der leeren Liste endet, ist die folgende Entsprechung gut nachzuvollziehen:

$$[a,b,c,d] \text{ entspricht } [a \mid [b \mid [c \mid [d \mid []]]]]$$

In einzelne Schritte unterteilt, gibt die nachstehende Grafik denselben Sachverhalt wieder:



Wie im vorherigen Kapitel aus der Definition für Listen deutlich wurde, sind diese (wie ja auch Operator-Operand-Strukturen) nur eine besondere Schreibweise der wohlbekannteren Funktor-Argument-Strukturen. Der Funktor ist im Falle der Liste ein einfacher Punkt: '.', dahinter stehen, in Klammern, die Argumente dieses Funktors. Intern übersetzt Prolog die Listen auch in Funktor-Argument-Strukturen. Um auf unser Beispiel zurückzukommen, kann also folgende Gleichung genannt werden:

$$[a,b,c,d] \text{ entspricht } [a \mid [b \mid [c \mid [d \mid []]]]] \text{ entspricht } .(a, .(b, .(c, .(d, []))))$$

Ein weiteres Beispiel, bei welchem das dritte Element eine eingebettete Liste ist:

```

[1,2,[a,b],3]
entspricht
[1|[2|[|[a|[b|[[]]]|3|[[]]]]]]
entspricht
.(1,.(2,.(a,.(b,[ ]),.(3,[ ])))

```

Man sieht, daß die jeweils erste Schreibweise am lesefreundlichsten ist – sie birgt aber möglicherweise die Gefahr, die komplexe rekursive Struktur, die eine Liste aufweist, zu übersehen. Die folgende kleine Aufwärmübung soll einzig das Ziel haben, sich dieser Tatsache bewußt zu werden, indem die einfachen Listenstrukturen jeweils übersetzt werden sollen in Listen mit Listenoperator einerseits und Funktor–Argument–Strukturen andererseits:



Wenn Ihr das Skript am PC durcharbeitet, könnt Ihr jetzt die **Aufgabe 1** am Kapitelende angehen

Soviel zu der internen Repräsentation von Listen. Es ging im wesentlichen darum, erneut ein wenig darauf herumzureiten, daß es sich bei Listen eben nicht um eine einfache Aneinanderreihung von Elementen handelt. In den nachfolgenden Abschnitten wird allerdings die einfache Schreibweise verwendet.

Als nächstes wollen wir eine Reihe von listenverarbeitenden Prädikaten definieren, die jeweils recht unterschiedliche Aufgaben erfüllen. Diese Prädikate werden nicht unbedingt alle später in anderen Programmen eingesetzt – einige dienen nur dazu, einen ersten Einstieg in den Umgang mit Listen zu erhalten und ein Gefühl für die Listenverarbeitung zu gewinnen – beispielsweise über die folgenden Prädikate der folgenden Übung:

kopf/2 und **rest/2**, die zu einer beliebigen Liste jeweils den Kopf bzw. den Rest ausgeben.

Beispielsweise soll gelten:

`kopf([a, b, c,d], a)`

`rest([a, b, c,d], [b,c,d])`

Es geht also um die Definition zweier Prädikate, die von einer beliebigen Liste entweder den Kopf oder den Rest ausgeben. Das Prädikat `kopf/2` soll mithin die Anfrage `?- kopf([a,b,c,d],X)` wie folgt beantworten:

```

SWI-Prolog (version 2.9.7)
?- consult(liste).
liste compiled, 0.00 sec, 380 bytes.

Yes
?- kopf([a,b,c,d],X).

X = a

Yes
?-

```

Entsprechend `rest/2`:

```

SWI-Prolog (version 2.9.7)
?- consult(liste).
liste compiled, 0.00 sec, 560 bytes.

Yes
?- rest([a,b,c,d],X).

X = [b, c, d]

Yes
?-

```

Die Frage ist, wie man ein solches Prädikat definiert. Nun, auch hier bietet es sich an, die Sache zunächst umgangssprachlich anzugehen:

Ein beliebiger Term X ist der Kopf einer Liste L , falls – diese Liste in einen Kopf K und einen Rest R zerlegt wird und – X mit K unifiziert wird.²³

Die Zerlegung der Liste nun ist eine einfache Operation – einfach deshalb, weil dafür der vordefinierte Listenoperator '|' verwendet werden kann. Der umgangssprachliche Ausdruck wird also wie folgt in Prolog übersetzt, wobei statt der Variablen R die anonyme Variable '_' für den Rest benutzt wird (sonst kommt wieder die Meldung 'Singleton Variable'):

Prolog-Regel:	Zeilenweise Erläuterung:
$\text{kopf}(X, L):-$ $L = [K _],$ $X = K.$	X ist der Kopf einer Liste L falls L in einen Kopf K und einen Rest (anonyme Variable) zerlegt wird ²⁴ und X mit K unifiziert wird.

Diese Regel liefert das gewünschte Ergebnis. Entsprechend lautet auch die Regel für $\text{rest}/2$:

Prolog-Regel:	Zeilenweise Erläuterung:
$\text{rest}(X, L):-$ $L = [_ R],$ $X = R.$	X ist der Rest einer Liste L falls L in einen Kopf (anonyme Variable) und einen Rest R zerlegt wird und X mit R unifiziert wird.

Es gibt allerdings die Möglichkeit, das Ganze ein wenig einfacher und eleganter zu formulieren. Bei diesem Verfahren geht es darum,

- die Listenzerlegung nicht explizit auszudrücken (also mit $L = [K | R]$, Teilziel (1)); sondern diese bereits im Regelkopf zu erfassen, indem die Liste schon hier in Kopf und Rest zerlegt wird.
- auch die Unifikation von X mit K nicht explizit auszudrücken (also mit $X = K$, Teilziel (2)), sondern diese ebenfalls im Regelkopf zu erfassen – einfach dadurch, daß für beide dieselbe Variable verwendet wird.

Das bedeutet, daß wir statt der oben angegebenen Regel für $\text{kopf}/2$ stattdessen ein Fakt formulieren, welches die folgende Form hat: $\text{kopf}(X, [X | _])$. Dieses Fakt konstatiert die Tatsache, daß ein Term X Kopf einer Liste $[X|_]$ ist. Dieses Verfahren, also diese vorweggenommene Auswertung nennt man PARTIELLE EVALUATION. Dazu gleich noch mehr.

Als nächstes geht es darum, die Prädikate $\text{erstes}/2$, $\text{zweites}/2$, $\text{drittes}/2$, die jeweils das erste, zweite und dritte Element einer Liste identifizieren. Beispielsweise soll gelten:

(A) $\text{erstes}/2$

Die Formulierung des ersten Prädikates, $\text{erstes}/1$, ist einfach – sie ist eigentlich identisch zu der Definition von $\text{kopf}/2$ weiter oben:

Prolog-Regel:	Zeilenweise Erläuterung:
$\text{erstes}(X, L):-$ $L = [K _],$ $X = K.$	X ist das erste Element einer Liste L falls L in einen Kopf K und einen Rest (anonyme Variable) zerlegt wird und X mit K unifiziert wird.

Es gibt aber auch die Möglichkeit, das bereits definierte $\text{kopf}/2$ einzusetzen:

Prolog-Regel:	Zeilenweise Erläuterung:
$\text{erstes}(X, L):-$ $\text{kopf}(X,L).$	X ist das erste Element einer Liste L falls X der Kopf von L ist

Schließlich kann auch $\text{erstes}/2$ durch ein Fakt ausgedrückt werden: $\text{erstes}(X, [X|_])$.

²³ 'Unifikation' bedeutet entweder, daß K an X gebunden wird, oder aber, daß K und X identisch sein müssen.

²⁴ Genauer gesagt: wenn L mit einer Konstruktion unifiziert wird, die entsprechend in einen Kopf und Rest zerlegt wird. Die anonyme Variable wird in den folgenden Prädikaten immer dann verwendet, wenn sie einen Term bezeichnet, auf den es sozusagen nicht ankommt, da dessen Wert für das Prädikat irrelevant ist. Andernfalls erscheint die beliebte Warnung 'Singleton Variable'.

(B) zweites/2

Das nächste Prädikat, *zweites/2*, scheint eigentlich im Widerspruch zu der eingangs getroffenen Äußerung zu stehen, daß man direkten Zugriff nur auf den Kopf einer Liste hat: wenn dem so ist, kann man ja garnicht das zweite (oder dritte oder vierte usw.) Element isoliert identifizieren. Aber: – und das ist der Kerngedanke der Listenverarbeitung – **jede Liste ist zerlegbar in Kopf und Rest**, und der Rest ist selber wieder eine Liste – hat also auch einen Kopf, auf den man wiederum Zugriff hat.

Nehmen wir zur Verdeutlichung ein Beispiel. Das Element *kicks* steht in der Liste [*the, boy, kicks, the, ball*] an der dritten Position. Wird die Liste in Kopf und Rest zerteilt, erhalten wir eine Restliste wie folgt:

[Kopf | Rest]
 =
 [the | [boy, kicks, the, ball]]

In der Restliste steht *kicks* an der zweiten Position. Wird diese Restliste, also [*boy, kicks, the, ball*], selber in Kopf und Rest aufgeteilt (hier *Kopf1* und *Rest1* genannt), erhalten wir folgendes Ergebnis:

[Kopf1 | Rest1]
 =
 [boy | [kicks, the, ball]]

Nun steht *kicks* an der ersten Position. Durch die Aufteilung von *Rest1*, also [*kicks, the, ball*], in *Kopf2* und *Rest2*:

[Kopf2 | Rest2]
 =
 [kicks | [the, ball]]

haben wir *kicks* in einer Position, in der wir Zugriff auf dieses Element haben – als Kopf einer Liste (schlecht ausgedrückt als Kopf des Restes des Restes des Rests).

Mit diesen Erkenntnissen im Hinterkopf ist die Problemstellung von *zweites/2* nun sicher einfacher nachvollziehbar. Erstmal wieder umgangssprachlich:

Ein beliebiger Term X ist das zweite Element einer Liste L, falls diese Liste in einen Kopf K und einen Rest R zerlegt wird und der Rest R selber in einen Kopf K1 und einen Rest R1 und X mit K1 unifiziert wird.

Prolog–Regel:	Zeilenweise Erläuterung:
$zweites(X, L):-$ $L = [_ R],$ $R = [K1 _],$ $X = K1.$	X ist das zweite Element einer Liste L falls L in einen Kopf (anonyme Variable) und einen Rest R zerlegt wird und R in einen Kopf K1 und einen Rest (anonyme Variable) zerlegt wird und X mit K1 unifiziert wird..

Mit partieller Evaluation hat das Prädikat die folgende, einfachere Form:

Prolog–Regel:	Zeilenweise Erläuterung:
$zweites(X, [_ R]):-$ $R = [X _].$	X ist das zweite Element einer Liste [$_ R$] falls X der Kopf von R ist.

Natürlich aber können wir *zweites/2* auch mit Bezug auf *erstes/2* ausdrücken:

Ein beliebiger Term X ist das zweite Element einer Liste L, falls – diese Liste in einen Kopf K und einen Rest R zerlegt wird und – X das erste Element des Restes R ist.

In Prolog sieht das so aus:

Prolog–Regel:	Zeilenweise Erläuterung:
$zweites(X, Liste):-$ $L = [_ R],$ $erstes(X,R).$	X ist das zweite Element einer Liste L falls L in einen Kopf (anonyme Variable) und einen Rest R zerlegt wird und X das erste Element von R ist.

(C) drittes/2

Auch das Prädikat *drittes/2* funktioniert nach demselben Prinzip. Das Beispiel von weiter oben ([the, boy, kicks, the ball]) hat ja bereits gezeigt, wie man das dritte Element einer Liste ermittelt:

Ein beliebiger Term X ist das dritte Element einer Liste L, falls – diese Liste in einen Kopf K und einen Rest R zerlegt wird, – der Rest R in einen Kopf K1 und einen Rest R1, – der Rest R1 in einen Kopf K2 und einen Rest – und X mit K2 unifiziert wird.

Prolog-Regel:	Zeilenweise Erläuterung:
$\text{drittes}(X, L):-$ $L = [_ R],$ $R = [_ R1]$ $R1 = [K2 _],$ $X = K2.$	X ist das dritte Element einer Liste L falls Liste in einen Kopf (anonyme Variable) und einen Rest R zerlegt wird und R in einen Kopf (anonyme Variable) und einen Rest R1 zerlegt wird und R1 in einen Kopf K2 und einen Rest (anonyme Variable) zerlegt wird und X mit K2 unifiziert wird..

Dasselbe wieder mit partieller Evaluation:

Prolog-Regel:	Zeilenweise Erläuterung:
$\text{drittes}(X, [_ R]):-$ $R = [_ R1],$ $R1 = [X, _].$	X ist das dritte Element einer Liste [_ R] falls R in einen Kopf (anonyme Variable) und einen Rest R1 zerlegt wird und X der Kopf von R1 ist.

Unter Bezug auf *zweites/2* schließlich könnte das Prädikat auch wie folgt formuliert werden:

Ein beliebiger Term X ist das dritte Element einer Liste [_ | R], falls X das zweite Element des Restes R ist.

Prolog-Regel:	Zeilenweise Erläuterung:
$\text{drittes}(X, [_ R]):-$ $\text{zweites}(X, R).$	X ist das dritte Element einer Liste [_ R] falls X das zweite Element von R ist.

Somit hätten wir die drei Prädikate formuliert. Um nun auch Prädikate wie *viertes/2*, *fünftes/2* usw. zu ermitteln, könnten wir theoretisch so weitermachen wie bisher. Was wir aber beispielsweise nicht können, ist ein Prädikat wie *letztes/2*, welches das letzte Element einer Liste ermittelt. Der Grund: Listen können beliebig viele Elemente enthalten – auf die oa. Weise wird aber immer nur eine endliche Zahl von Positionen 'abgearbeitet'. Es ist in der Tat so, das die o.a. Prädikate die eigentlich wirklich elegante Art der Listenverarbeitung verfehlen, nämlich die rekursive Listenverarbeitung. Diese ist der Gegenstand der nächsten Abschnitte.

5.2. Prinzipien der rekursiven Listenverarbeitung

Das Prinzip der Rekursion ist bereits in Kapitel 3 in Zusammenhang mit *unterbegriff/2* eingeführt worden. Dabei ist die folgende Aussage getroffen worden:

Das Prinzip bei der Rekursion ist daher, eine einfache 'Erfolgsbedingung' zu formulieren [...] und die rekursive Bedingung auf eine Art auszudrücken, daß sie quasi solange wiederholt wird, bis diese einfache Erfolgsbedingung erreicht wird.

Für die Verarbeitung von rekursiven Strukturen, also Strukturen wie Listen, bieten sich rekursive Prozeduren (Prädikate) an, die genau dieses Prinzip erfüllen. Die Prozedur wird solange durchgeführt, bis eine Erfolgsbedingung eintritt. Darunter muß hier verstanden werden, daß der Fall angegeben wird, an dem die Rekursion abbricht: bei Listen ist dieser Fall i.d.R. dann erreicht, wenn diese entweder leer sind oder nur ein Element enthalten.

Daraus ergibt sich folgendes allgemeines Verfahren zur Listenverarbeitung:

- a) Wende die Prozedur auf die leere Liste (bzw. die Einerliste) an (Erfolgsbedingung)
- b) Zerlege die Liste in Kopf und Rest
 - Behandle den Kopf
 - Wende die Prozedur **rekursiv** auf den Rest an

Dieses Verfahren ist am besten anhand eines konkreten Beispielen zu erklären. Dazu geht es um die Formulierung eines Prädikates `schreibe_liste/1`, welches die Elemente einer beliebigen Liste – egal, wie lang – zeilenweise auf dem Bildschirm ausgibt. Eine Anfrage wie `schreibe_liste([the,boy,kicks,the,ball])` soll also folgendes Ergebnis bringen:

```

SWI-Prolog [version 2.9.7]
Welcome to SWI-Prolog (Version 2.9.7)
Copyright (c) 1993-1997 University of Amsterdam. All rights reserved.

For help, use ?- help(Topic). or ?- apropos(Word).

?- consult(liste).
liste compiled, 0.00 sec, 588 bytes.

Yes
?- schreibe_liste([the,boy,kicks,the,ball]).
the
boy
kicks
the
ball

Yes
?-

```

Wiederum wollen wir die Problemstellung wie folgt umgangssprachlich formulieren:

Wenn die Liste leer ist, passiert gar nichts, die Prozedur wird abgebrochen.

*Wenn die Liste nicht leer, so zerlege sie in einen Kopf und einen Rest, gebe den Kopf mit `write/1` auf dem Bildschirm aus, gehe mit `nl` in eine neue Zeile **und wende die Prozedur auf den Rest an.***

Der zentrale Punkt ist natürlich der erneute Aufruf des Prädikates mit dem Rest – der Teil, der in der Formulierung fettgedruckt ist. In Prolog hat das die folgende Form:

Prolog-Regel:	Zeilenweise Erläuterung:
<code>schreibe_liste([]).</code>	Wenn die Liste leer ist, passiert garnichts! (Abbruchbedingung)
<code>schreibe_liste(Liste):- Liste = [K R], write(K), nl, schreibe_liste(R).</code>	Wenn die Liste nicht leer ist, zerlege die Liste in einen Kopf K und einen Rest R, gebe K am Bildschirm aus, gehe in eine neue Zeile rufe <code>schreibe_liste/1</code> mit R auf (Rekursionsschritt!!)

Etwas einfacher, mit partieller Evaluation, sieht das Prädikat so aus:

```

schreibe_liste([ ]).
schreibe_liste([K|R]):-
  write(K),
  nl,
  schreibe_liste(R).

```

Anhand des folgenden kommentierten *Screen-Shots* wird gut deutlich, wie Prolog die gesamte Liste abarbeitet, bis schließlich die leere Liste erreicht ist und die Prozedur abbricht:

```

SWI-Prolog [version 2.9.7]
Yes
?- trace, schreibe_liste([a,b,c]).
Call: ( 7) schreibe_liste([a, b, c]) ? creep
Call: ( 8) write(a) ? creep
a Exit: ( 8) write(a) ? creep
Call: ( 8) nl ? creep

Exit: ( 8) nl ? creep
Call: ( 8) schreibe_liste([b, c]) ? creep
Call: ( 9) write(b) ? creep
b Exit: ( 9) write(b) ? creep
Call: ( 9) nl ? creep

Exit: ( 9) nl ? creep
Call: ( 9) schreibe_liste([c]) ? creep
Call: ( 10) write(c) ? creep
c Exit: ( 10) write(c) ? creep
Call: ( 10) nl ? creep

Exit: ( 10) nl ? creep
Call: ( 10) schreibe_liste([]) ? creep
Exit: ( 10) schreibe_liste([]) ? creep
Exit: ( 9) schreibe_liste([c]) ? creep
Exit: ( 8) schreibe_liste([b, c]) ? creep
Exit: ( 7) schreibe_liste([a, b, c]) ? creep

Yes
?-

```



Wenn Ihr das Skript am PC durcharbeitet, könnt Ihr jetzt die **Aufgaben 2 und 3** am Kapitelende angehen. Aufgabe 3 ist detailliert erläutert.

5.2.1. ABBILDUNG VON LISTEN AUF LISTEN

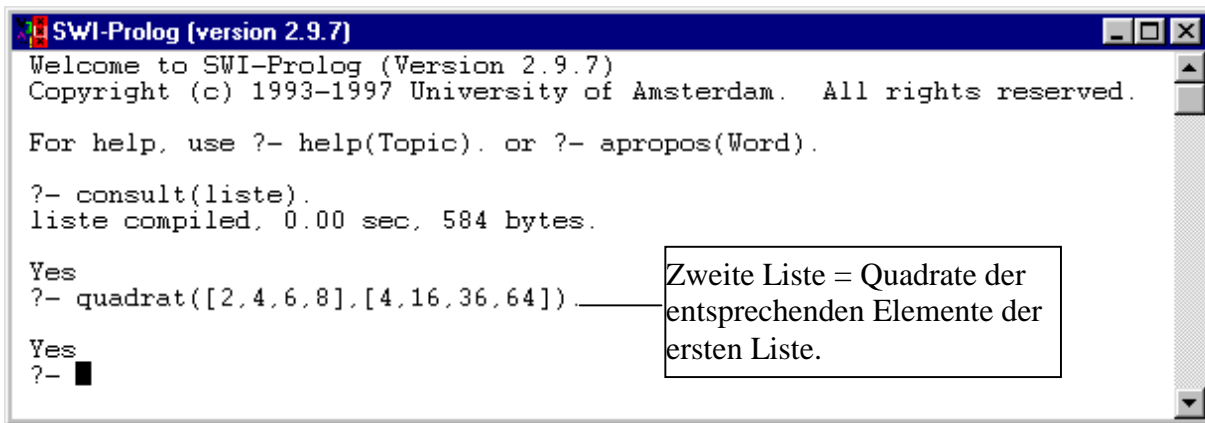
Bei den Prädikaten `erstes/2`, `element/2`, `laenge/2` usw. ging es immer um eine einzelne Liste. Ein sehr häufiges Prinzip der Listenverarbeitung ist aber die **ABBILDUNG** einer Liste auf eine zweite Liste (bzw. von mehreren Listen auf eine Liste). *Abbildung* bedeutet, daß ein jedes Element einer Liste L1 nach einem bestimmten Prinzip (nämlich durch eine **ABBILDUNGSFUNKTION**) dem entsprechenden Element einer Liste L2 zugeordnet wird.

$$\begin{array}{cccccc}
 \text{L1: [X1,} & \text{X2,} & \text{X3,} & \text{...,} & \text{X}_n \text{]:} & \\
 \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \\
 \text{L2: [Y1} & \text{Y2} & \text{Y3,} & \text{...,} & \text{Y}_n \text{]} &
 \end{array}$$

Im folgenden Beispiel bilden die Elemente der zweiten Liste jeweils das Quadrat der entsprechenden Elemente der ersten Liste. Sie sind also durch die Abbildungsfunktion $Y = X \cdot X$ definiert.

$$\begin{array}{cccccc}
 \text{L1: [2,} & \text{3,} & \text{4,} & \text{...,} & \text{X}_n \text{]:} & \\
 \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \\
 \text{L2: [4} & \text{9} & \text{16,} & \text{...,} & \text{X}_n \cdot \text{X}_n \text{]} &
 \end{array}$$

Betrachten wir dazu ein konkretes Prädikat, nämlich `quadrat/2`, welches genau diese Aufgabe erfüllt: es hat als Argumente zwei Listen, die Elemente der zweiten Liste bilden jeweils das Quadrat der entsprechenden Elemente der ersten Liste:



```

SWI-Prolog (version 2.9.7)
Welcome to SWI-Prolog (Version 2.9.7)
Copyright (c) 1993-1997 University of Amsterdam. All rights reserved.

For help, use ?- help(Topic). or ?- apropos(Word).

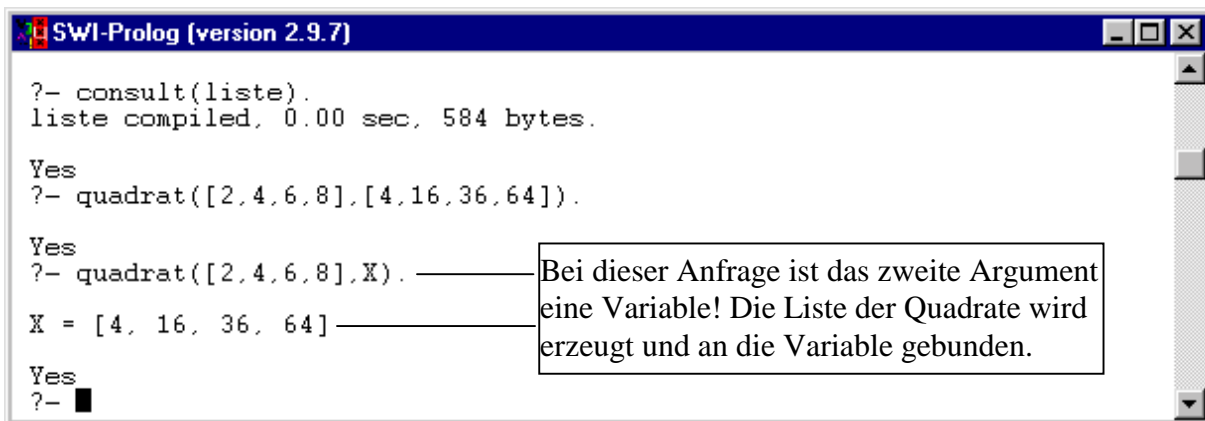
?- consult(liste).
liste compiled, 0.00 sec, 584 bytes.

Yes
?- quadrat([2,4,6,8],[4,16,36,64])
Yes
?-

```

Zweite Liste = Quadrate der entsprechenden Elemente der ersten Liste.

Die Bezeichnung 'Abbildung' ist nun vielleicht insofern mißverständlich, als sie möglicherweise suggeriert, beide Listen würden bereits vorgegeben sein – so wie gerade in diesem Beispiel, bei welchem `quadrat/2` mit zwei instantiierten Argumenten aufgerufen wurde (d.h. es wurden keine Variablen verwendet). Interessant wird die Sache aber insbesondere dadurch, daß Prädikate, die Listen aufeinander abbilden, auch verwendet werden, um Listen sozusagen neu aufzubauen – wie im nachfolgenden Screen-Shot, bei welchem die zweite Liste (die sogenannte ZIELLISTE) erst ermittelt werden soll:



```

SWI-Prolog (version 2.9.7)
?- consult(liste).
liste compiled, 0.00 sec, 584 bytes.

Yes
?- quadrat([2,4,6,8],[4,16,36,64]).

Yes
?- quadrat([2,4,6,8],X).
X = [4, 16, 36, 64]
Yes
?-

```

Bei dieser Anfrage ist das zweite Argument eine Variable! Die Liste der Quadrate wird erzeugt und an die Variable gebunden.

Das Verfahren, das zwei Listen aufeinander abbildet, arbeitet völlig analog zu dem der allgemeinen rekursiven Listenverarbeitung:

1. Formuliere eine Endbedingung (z.B.: beide Listen sind leer)
2. Zerlege beide Listen in Kopf und Rest
 - a. Wende die Abbildungsfunktion auf die Listenköpfe an;
 - b. Wende die Prozedur rekursiv auf die Listenreste an.

Zur Illustration dient ein Beispiel, nämlich genau das Prädikat `quadrat/2`:

Definiert ein Prädikat `quadrat/2`, das dann erfolgreich ist, wenn jedes Element y_i von Liste2 das Quadrat des entsprechenden Elementes x_i von Liste1 ist, d.h. $y_i = x_i \cdot x_i$.

Problembeschreibung:

Wenn die Ausgangsliste leer ist, ist auch die Zielliste leer.

Zerlege die Ausgangs- und Zielliste jeweils in Kopf und Rest.

Der Kopf der Zielliste ist das Quadrat des Kopfes der Ausgangsliste.

Der Rest der Zielliste ist die 'Quadrat'-liste des Restes der Ausgangsliste.

Diese Problembeschreibung wird direkt in Prolog übersetzt:

```

quadrat([ ], [ ]).
quadrat ([K1 | R1], [K2 | R2]):-
    K2 is K1 * K1,
    quadrat(R1, R2).

```


Die erste Regel (das erste Fakt) ist die Endbedingung – hier also die leeren Listen. In der rekursiven Regel wird zunächst der Kopf 'bearbeitet', und zwar nach der Abbildungsfunktion $Y = X \cdot X$, dann wird das Prädikat erneut mit den Restlisten aufgerufen.



Wenn Ihr das Skript am PC durcharbeitet, könnt Ihr jetzt die **Aufgaben 5 – 7** am Kapitelende angehen.

Übungen zu Kapitel 5

Aufgabe.1.

Übersetzt (auf Papier) die folgenden Listen (einfache Schreibweise) in 'explizite' Listen (mit Listenoperator und leeren Listen) einerseits und in Funktor-Argument-Strukturen andererseits:

[the, boy, slept]

[a, b, [1,2], c] (Liste mit eingebetteter Liste).

Übersetzt (auf Papier) die folgende Funktor-Argument-Struktur in eine Liste mit einfacher Schreibweise:

.(der, .(junge, .(lacht, [])))

Ein kleiner Tip: man kann diese Übung mit Prolog überprüfen. Das Prädikat `write/1` nämlich kann dazu benutzt werden, Funktor-Argument-Strukturen in Listen zu 'übersetzen':

```

SWI-Prolog (version 2.9.7)
Welcome to SWI-Prolog (Version 2.9.7)
Copyright (c) 1993-1997 University of Amsterdam. All rights reserved.

For help, use ?- help(Topic). or ?- apropos(Word).

?- write(.(dies, .(ist, .(eine, .(liste, [ ])))).
[dies, ist, eine, liste]
Yes
?-
  
```

Das

Prädikat `display/1` dagegen kann eingesetzt werden, um für einen Term (hier also eine Liste) anzuzeigen, wie dieser intern in Prolog repräsentiert wird:

```

SWI-Prolog (version 2.9.7)
Welcome to SWI-Prolog (Version 2.9.7)
Copyright (c) 1993-1997 University of Amsterdam. All rights reserved.

For help, use ?- help(Topic). or ?- apropos(Word).

?- display([dies, ist, auch, eine, liste]).
.(dies, .(ist, .(auch, .(eine, .(liste, [ ])))))
Yes
?-
  
```

Die nächsten beiden Übungen sind rekursive Prädikate zur Listenverarbeitung. Auch hier ist es so, daß der Hauptnutzen der Übungen darin liegt, ein Gefühl für den Umgang mit Prolog-Listen zu entwickeln – der tatsächliche Nutzen der zu erstellenden Prädikate für spätere Programme ist also eher mittelbar.

Aufgabe 2

Definiert ein Prädikat `element/2`. Das erste Argument ist ein Term, das zweite Argument eine Liste. Das Prädikat `element/2` soll für den Term ermitteln, ob er Element in der Liste ist oder nicht. Beispielsweise soll gelten:
`element(c, [a,b,c,d]).`

Lösungstip: der 'einfache' Fall, die Erfolgsbedingung in diesem Beispiel ist nicht mit Bezug auf die leere Liste formuliert, sondern die Tatsache, daß der Kopf einer Liste natürlich Element dieser Liste ist. Man könnte es so ausdrücken:

Ein Term X ist Element einer Liste L, wenn diese in Kopf und Rest zerlegt wird und X der Kopf von L ist.

Ein Term X ist Element der Liste L, wenn es Element des Restes ist.

Durch eine entsprechende Prolog-Formulierung wird die Liste solange abgearbeitet, bis irgendwann der Fall eingetreten ist, bei welchem diejenige Restliste erreicht wird, deren Kopf das gesuchte Element X ist (so es denn tatsächlich in der Liste vorkommt). Übrigens hat dieses Prädikat das Fließmuster `element(?Term,+Liste)` (zu Fließmuster siehe Kapitel 4, *Notationskonventionen für Prädikate*). Das heißt, daß bei einer Anfrage das erste Argument von `element/2` auch eine Variable sein kann. Dann gibt Prolog alle Werte aus, an die diese Variable gebunden werden kann – mithin alle Elemente der Liste:

```

SWI-Prolog (version 2.9.7)
Yes
?- element(X, [1, 2, 3]).

X = 1 ;
X = 2 ;
X = 3 ;

No
?- █

```

Aufgabe 3

Definiert ein Prädikat `letztes/2`, welches das letzte Element einer Liste ermittelt. Das erste Argument ist ein Term, das zweite Argument eine Liste. `letztes/2` soll für den Term ermitteln, ob er das letzte Element in der Liste ist oder nicht. Beispielsweise soll gelten: `letztes(d, [a,b,c,d])`.

Auch diese Aufgabenstellung ist nur durch die rekursive Listenverarbeitung zu bewältigen – einfach deshalb, weil man nie genau weiß, wieviele Elemente die Liste aufweist, und also keine Prädikate formulieren kann, die das Listenende in einer vorher festgelegten Zahl von Schritten ermitteln.

Der einfache Fall, die Erfolgs- oder Abbruchbedingung ist in diesem Prädikat die EINERLISTE. Diese wird wie folgt notiert: `[X]` (siehe dazu auch die Definition von Liste im letzten Kapitel). Umgangssprachlich wird die Problemstellung wie folgt umschrieben:

Wenn eine Liste nur ein Element aufweist, so ist dieses Element das letzte Element der Liste.

Wenn eine Liste mehr als ein Element aufweist, so zerlege sie in Kopf und Rest. Das gesuchte Element ist das letzte Element des Restes.

Komplexere Fälle der Listenverarbeitung entstehen, wenn zusätzliche Argumente mitgeführt werden. Ein solcher Fall kann beispielsweise dann eintreten, wenn die Länge einer Liste ermittelt werden soll – die Frage ist, wie das geht, denn dafür muß Prolog die Elemente der Liste ja irgendwie 'zählen':

Aufgabe 4

Definiert ein Prädikat `laenge/2`, welches die Länge einer Liste ermittelt. Das erste Argument ist eine Liste, das zweite Argument ist eine Zahl, die die Länge der Liste angibt. Beispielsweise soll gelten: `laenge([a,b,c,d,e],5)`.

Für dieses Prädikat benötigen wir entsprechende Operatoren, in diesem Fall den Additionsoperator '+' und den Auswertungsoperator 'is'. Dazu erstmal ein paar Anmerkungen.

Exkurs: Auswertung arithmetischer Ausdrücke

In Standard-Prolog sind eine Reihe von arithmetischen Operatoren definiert: +, -, *, / etc. die in entsprechenden Operator-Operand-Strukturen verwendet werden können, z.B. `2 + 3`, `5 - 2`, `A * B`, `N / 3`, etc. Dabei muß man jedoch berücksichtigen, daß es sich hier um bloße syntaktische Konstruktionen handelt, die nicht automatisch ausgewertet werden. Durch den Ausdruck `N = 2 + 3` wird N nicht etwa mit

der Zahl 5 unifiziert, sondern mit der Struktur $2 + 3$ (\equiv $'+(2, 3)$). Wenn arithmetische Ausdrücke ausgewertet werden sollen, muß ein eigener Auswertungsoperator `is/2` verwendet werden: `Ausdruck1 is Ausdruck2`. Dieser Operator ist so definiert, daß `Ausdruck2` ausgewertet wird und das Ergebnis mit `Ausdruck1` unifiziert wird. `Ausdruck1` ist entweder eine Variable oder eine Ganzzahl, `Ausdruck2` muß ein syntaktisch korrekter arithmetischer Ausdruck sein.

```
?- N is 2+3
      Auswertung
N = 5
?- 5 is 2 + 3
Yes
```

Wichtig: Ein arithmetischer Ausdruck kann nur dann ausgewertet werden, wenn er keine ungebundenen Variablen enthält.

Damit können wir zu unserer Aufgabenstellung zurückkehren, die Länge einer Liste zu ermitteln. Das Prädikat `laenge/2` soll also folgendes leisten: eine Anfrage wie `laenge([a,b,c,d],X)` führt dazu, daß Prolog die Länge der Liste ermittelt und `X` an den entsprechenden Wert bindet:

```
SWI-Prolog [version 2.9.7]
Welcome to SWI-Prolog (Version 2.9.7)
Copyright (c) 1993-1997 University of Amsterdam. All rights reserved.

For help, use ?- help(Topic). or ?- apropos(Word).

?- consult(liste).
liste compiled, 0.00 sec, 552 bytes.

Yes
?- laenge([a,b,c,d],X).
X = 4

Yes
?-
```

Um uns dem Verfahren, welches dafür angewendet wird, zu nähern, betrachten wir zunächst die folgenden Listen. Die erste Liste ist leer – ihre Länge ist entsprechend null. Die zweite Liste enthält zwei Elemente, die dritte Liste drei und die vierte Liste fünf – wobei das vierte Element selber eine Liste ist:

```
[ ]           Länge = 0.
[a, b]        Länge = 2.
[a, b, c]     Länge = 3.
[a, b, [1, 2, 3], c] Länge = 4.
```

Der entscheidende Fingerzeig für die Beschreibung des Problem es kann nun wie folgt durch die Aufteilung der Listen in Kopf und Rest dargestellt werden:

```
[ ]           Länge = 0.
[a | [b]]     Länge = 2 – bzw. 1 + 1
[a | [b, c]]  Länge = 3 – bzw. 1 + 2
[a | [b, [1, 2, 3], c]] Länge = 4 – bzw. 1 + 3
```

Wir sehen: die Länge einer Liste ist immer um eins größer als die Länge der Restliste. Die Länge der leeren Liste ist immer null. Genau diesen Sachverhalt gilt es, in Prolog umzusetzen:

Die Länge der leeren Liste ist null.

*Die Gesamtlänge der Elemente einer Liste $[K | R]$ ist um eins größer als die Gesamtlänge der Restliste R , d.h. **Gesamtlänge = Restlänge + 1**.*

Salopp könnte man es so sagen: *Wenn ich weiß, wie lang die Restliste ist, weiß ich auch, wie lang die ganze Liste ist.* Diese Problembeschreibung läßt sich direkt in Prolog übersetzen. Für die Formulierung benötigen wir die beiden arithmetischen Operatoren `is/2` und `+/2`.

```

laenge([ ],0).
laenge([K | R],X):-
    laenge(R,Y),
    X is Y + 1.

```

Das klingt ja zu schön, um wahr zu sein – auch hier mag man sich, ähnlich wie bereits bei der Formulierung der Prädikate *unterbegriff/2*, *vorfahr/2* usw., fragen, wie das den funktionieren soll, woher also Prolog wissen kann, wie lang die Restliste ist? Schließlich wollen wir das Prädikat *laenge/2* ja erst definieren, verwenden es aber hier bereits im Regelrumpf der rekursiven Regel, um die Länge des Restes zu ermitteln.

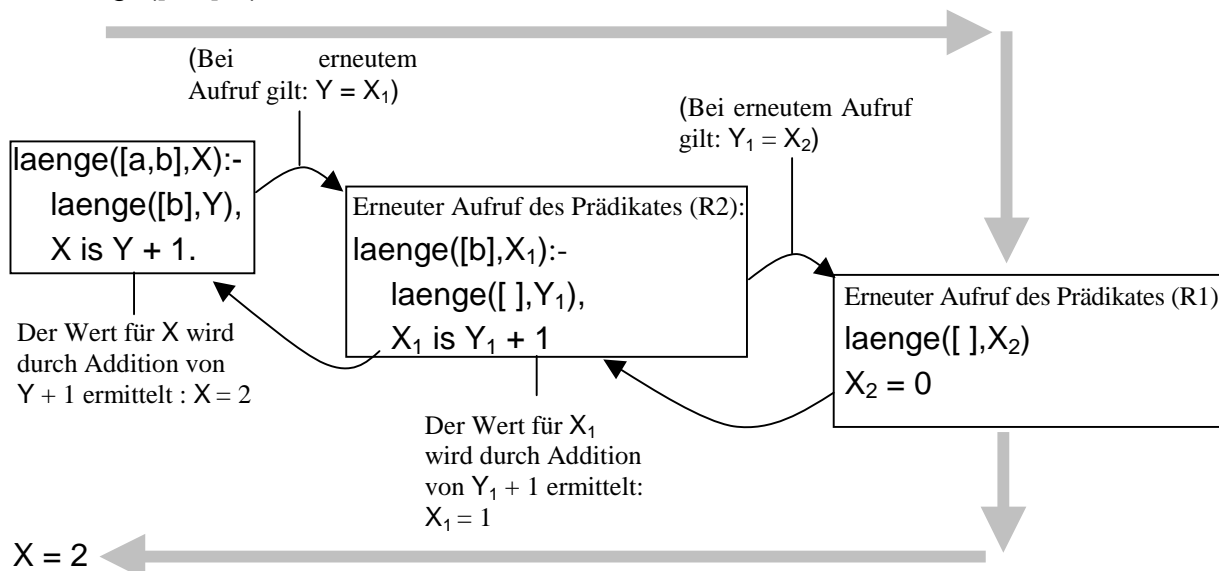
Was also passiert, wenn dieses Prädikat mit einem entsprechenden Argument aufgerufen wird? Nun, ist dieses Argument eine leere Liste, so ist die Antwort schnell gefunden:

?- *laenge*([],X). Da diese Liste leer ist, kommt Regel 1 zur Anwendung:

X = 0

Ist die Liste aber nicht leer, wird das Prädikat erneut aufgerufen – diesmal mit dem Rest der Liste. Schematisch könnte das wie folgt dargestellt werden, die Variablen sind mit Indizes versehen, da auch Prolog die Variablen bei erneutem Aufruf des Prädikates umbenennt.

?- *laenge*([a,b],X). Da diese Liste nicht leer ist, Aufruf von R2:

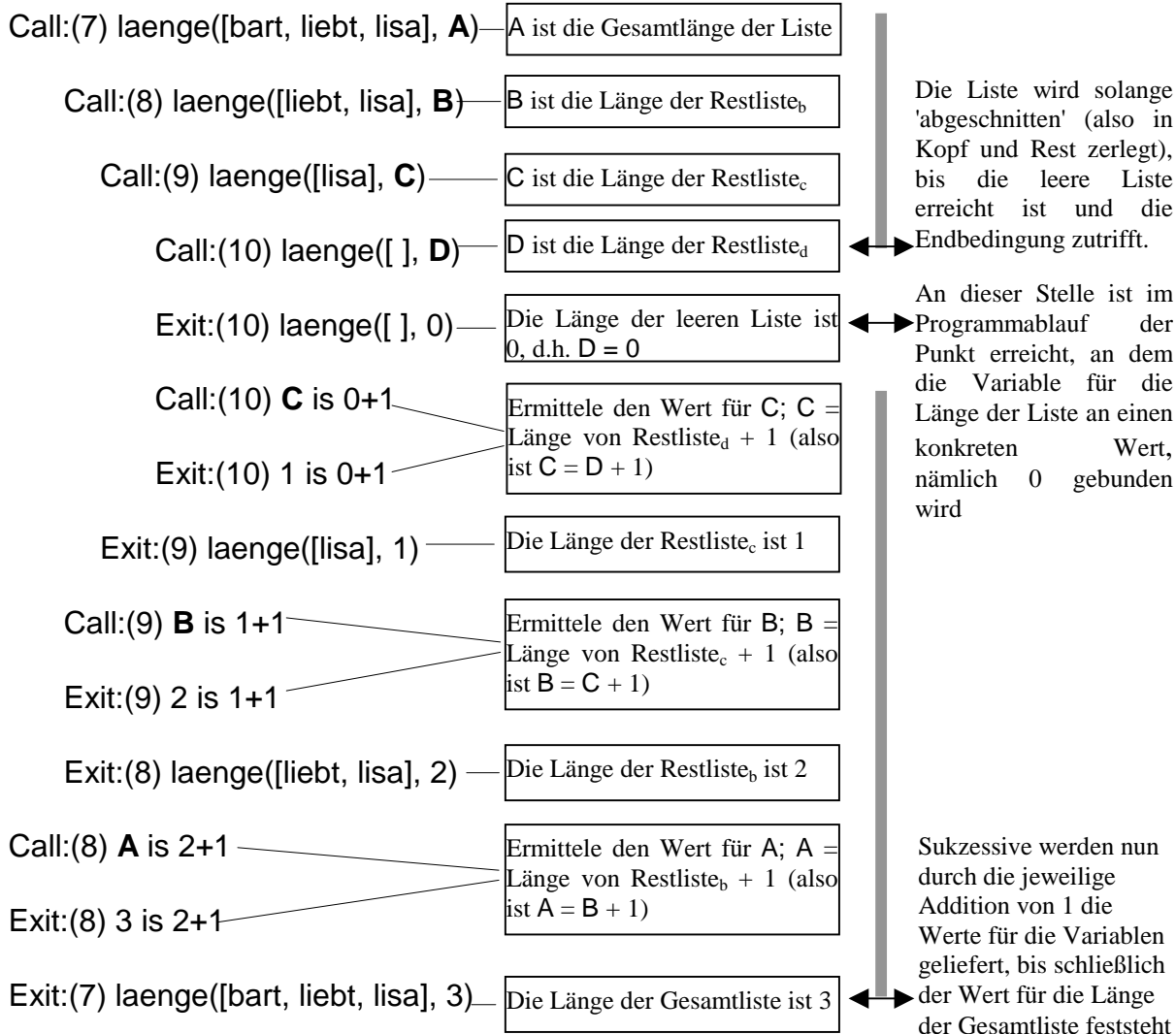


(Kleine Anmerkung: falls diese Graphik den Geist mehr verwirrt – akzeptiert)

Prolog ruft das Prädikat solange auf, bis der einfache Fall eingetreten ist – bis also die leere Liste erreicht wird. Danach geht es quasi wieder zurück – angefangen bei Null werden die Werte für die Längen der jeweiligen Restlisten durch die sukzessive Addition von Eins zu den Restlistenlängen (auahauaha!) ermittelt.

Ein Blick auf die nachfolgende Darstellung soll ebenfalls Erleuchtung bringen, indem gezeigt wird, wie Prolog eine entsprechende Anfrage abarbeitet. Es handelt sich um eine leicht modifizierte Auflistung der Ausgabe, die Prolog liefert, wenn die Anfrage *laenge*([bart, liebt, lisa],A). mit der *Trace*-Funktion aufgerufen würde. Sie ist insofern modifiziert, als die jeweiligen Variablen, deren Wert es zu ermitteln gilt, nicht als *_L112* oder *_G4301* dargestellt sind (wie bei der Original-Trace-Funktion), sondern von A–D gegliedert sind. Es lohnt sich, in den sauren Apfel zu beißen und diese zunächst etwas unübersichtlich scheinende Darstellung genau durchzuarbeiten – ihr ist nämlich genau zu entnehmen, wie die Verarbeitung dieses rekursiven Listenprädikates funktioniert.

?- trace, laenge([bart,liebt,lisa], A).



X = 3

Yes

Man sieht, was passiert: Prolog arbeitet die Liste sukzessive ab (d.h. zerlegt sie in Kopf und Rest), und – das ist ein ganz wichtiger Punkt – legt die nicht-beantwortbaren Ziele, hier also die Werte der Variable A – C, solange auf Eis, bis schließlich die Endbedingung erreicht ist, und ein konkreter Zahlenwert für die Variable D ermittelt ist. Danach werden die zuvor zurückgestellten Ziele mit dem Additionsverfahren nacheinander abgearbeitet, bis endlich die Länge der Gesamtliste ermittelt ist.

Soviel zur internen Verarbeitung rekursiver Listenprädikate. In vielen gängigen Prolog-Einführungen wird dieser Punkt nicht explizit aufgegriffen – denn dem eigentlichen Ziel aller Einführungen, der Vermittlung der deklarativen Programmierung, bei welcher ein Programm aus einer reinen Problembeschreibung besteht (wie in diesem Skript immer die kursiv gedruckten Textstellen), dessen Lösung dann Prolog obliegt, trägt die explizite Angabe der dabei durchgeführten Verarbeitungsschritte nicht unbedingt bei. Im vorliegenden Skript wird aber dennoch versucht, zu zeigen, was intern passiert – einfach deshalb, weil erfahrungsgemäß im Zusammenhang mit rekursiven Prädikaten immer wieder die Frage 'wie kann das denn funktionieren' aufkommt. Aber auch hier, bei der rekursiven Listenverarbeitung, verhält es sich wie im Abschnitt über rekursive Prädikate im vierten Kapitel: das eigentliche Verarbeitungsverfahren ist doch irgendwie einfacher, als es die komplizierten Ausführungen vermuten lassen. Entsprechend wird auf die animierte Darstellung in der Veranstaltung verwiesen, die dann hoffentlich zum weiteren Verständnis dieser eleganten Programmieretechnik beiträgt.

Aufgabe 5

Definiert die Prädikate `mal_zwei/2`, `mal_drei/2` und `mal_vier/2`, die dann erfolgreich sind, wenn jedes Element y_i von Liste2 entweder das Doppelte, das Dreifache oder das Vierfache des entsprechenden Elementes x_i von Liste1 ist, d.h. $y_i = x_i \cdot 2$, $y_i = x_i \cdot 3$ und $y_i = x_i \cdot 4$.

Diese Prädikate sollen Anfragen wie die folgenden ermöglichen:

```

SWI-Prolog (version 2.9.7)
liste compiled, 0.06 sec, 1,588 bytes.

Yes
?- mal_zwei([2,3,4,5,6],X).

X = [4, 6, 8, 10, 12]

Yes
?- mal_drei([2,3,4,5,6],X).

X = [6, 9, 12, 15, 18]

Yes
?-
  
```

Problembeschreibung für `mal_zwei/2`:

Wenn die Ausgangsliste leer ist, ist auch die Zielliste leer.

Zerlege die Ausgangs- und Zielliste jeweils in Kopf und Rest.

Der Kopf der Zielliste ist das Doppelte des Kopfes der Ausgangsliste.

Der Rest der Zielliste ist die Liste der doppelten Elemente des Restes der Ausgangsliste.

Diese drei Prädikate sind im Grunde ja identisch – bis auf den Faktor der Multiplikation, dieser ist entweder zwei, drei oder vier. Schöner wird das Ganze, wenn der Benutzer diesen Faktor bei der Anfrage selber bestimmen kann – z.B. so:

```

SWI-Prolog (version 2.9.7)
?- consult(liste).
liste compiled, 0.00 sec, 1,920 bytes.

Yes
?- multipliziere([2,3,4,5,6],X,9).
X = [18, 27, 36, 45, 54]

Yes
?-
  
```

Das dritte Argument ist der Faktor der Multiplikation – in diesem konkreten Fall 9.

Aufgabe 6

Definiert ein Prädikat `multipliziere/3`, also `multipliziere(Liste1, Liste2, N)`, welches dann erfolgreich sind, wenn jedes Element y_i von der Zielliste das N-fache des entsprechenden Elementes x_i von Liste1 ist, d.h. $y_i = x_i \cdot N$.

Aufgabe 7

Legt in einer Datei `lexikon.pl` eine Reihe von Fakten `lex/2` über die Zugehörigkeit von Wörtern zu lexikalischen Kategorien an (z.B. `lex(kicks,verb)`). Schreibt ein Prädikat `lexkat/2`, das jedem Wort einer Liste von Wörtern seine Wortklasse zuordnet. Beispielsweise soll gelten:

`lexkat([the, boy], [det, nomen])`