Logische und funktionale Programmierung

Vorlesung 5: Listen

Babeş-Bolyai Universität, Department für Informatik, Cluj-Napoca
csacarea@cs.ubbcluj.ro

10. November 2017

Listen

Listen sind eine Datenstruktur, die sehr gebräuchlich in der nicht-numerischen Programmierung ist.

Liste:

'geordnete Folge von Elementen beliebiger Länge'

- Elemente: beliebige Terme, Konstanten, Variablen, Strukturen $\rightarrow$ insbesondere andere Listen.
- Wichtig: Listen können praktisch jede Art von Struktur repräsentieren, die man in der symbolischen Programmierung verwenden will.

Beispiel:

Parserbäume, Grammatiken, Stadtpläne, Formeln, Funktionen.

Listen

Listen als spezieller Baum:

- eine Liste ist entweder die leere Liste [ ] oder eine Struktur mit zwei Komponenten: head und tail
- das Ende der Liste ist tail als leere Liste
- head und tail sind Komponenten des Funktors '.'

Listen

In PROLOG können Listen auch mit dem Listenoperator . (Punkt) geschrieben werden:
`[tick, trick, track]` entspricht
`.(tick,.(trick,.(track,[])))`

Liste aus einem Element

'a': .(a,[ ]) oder [a]
.(a,.(b,.(c,[ ]))) oder [a,b,c] komma-getrennt in eckigen Klammern.

Beispiel legaler Listen

```
[ ]
[the, men, [like,to,fish] ]
[a,V1,b,[X,Y]]
```

Jeder horizontale Level ist eine Liste mit einer bestimmten Anzahl von Elementen.

```
[a,V1,b,[X,Y]]
```

1. Level 1: 4
2. Level 2: 2

Zugriff auf Listen

Eine Hauptaufgabe in PROLOG Prädikaten besteht darin, auf Elemente einer Liste zuzugreifen. Da es in PROLOG keine Möglichkeit gibt über eine INDEX Position auf die Elemente einer Liste zuzugreifen, muss sich der Programmierer auf die `head` und `tail` Funktionalität des Restlistenoperators | beschränken. Die `head` und `tail` Funktionalität wird schon beim Aufruf eines Prädikats eingesetzt um über Argumente auf die Elemente der Liste zuzugreifen.

Restlistenoperator

- Liste in `Head` und `Tail` aufteilen.
- Kopf der Liste ist das erste Element des '.' Funktors bzw. das erste Element nach der öffnenden eckigen Klammer.
- Head: erstes Element
- Tail: alle bis auf das erste Element
- Die leere Liste hat weder head noch tail
- Spezielle Notation $[X \mid Y]$: Liste mit Kopf X und tail Y, '|' heisst Restlistenoperator

Beispiele:

```
p([1,2,3]).
p([the, cat, sat, [on,the,mat]]).
?- p([X|Y]).
X=1 Y= [2,3]
X=the, Y=[cat,sat,[on,the,mat]].
```

Listen und Matching

List 1	List 2	Instantiations
[X, Y, Z]	[john, likes, fish]	X = john Y = likes Z = fish
[cat]	[X|Y]	X = cat Y = []
[X, Y|Z]	[mary, likes, wine]	X = mary Y = likes Z = [wine]
[[the, Y]|Z]	[[X, hare], [is, here]]	X = the Y = hare Z = [[is,here]]
[golden|T]	[golden, norfolk]	T = [norfolk]
[vale, horse]	[horse, X]	(none)
[white|Q]	[P|horse]	P = white Q = horse

Operationen mit Listen

- Testen, ob ein Objekt Element einer Liste ist, oder nicht.
- Konkatenation von Listen.
- Zufügen von Objekte in Listen/Löschen von Objekte aus Listen.

MEMBERSHIP

`member(X, L).`

X Objekt, L Liste. Das goal `member(X, L)` is wahr, falls X Element der Liste L ist.

Beispiel:

Membership

X gehört zu L falls

1. X ist `head` von L, oder
2. X ist ein Element des `tails` von L/

In Klauselform:

```
member(X, [X|Tail]).
member(X, [Head|Tail]):- member(X, Tail).
```

KONKATENATION

`conc(L1, L2, L3)`

`conc([a,b], [c,d], [a,b,c,d])` → wahr, aber
`conc([a,b], [c,d], [a,b,a,c,d])` → `falsch`.

Fälle

1. Die leere Liste `conc([ ], L, L).`
2. `[X| L1]:conc([X| L1], L2, [X|L3]):- conc(L1, L2, L3).`

```
?- conc( [a,[b,c],d], [a,[ ],b], L).
L = [a, [b,c], d, a, [ ], b]
```

KONKATENATION

```
?- conc( L1, L2, [a,b,c] ).
L1 = [ ]
L2 = [a,b,c];

L1 = [a] L2 = [b,c];

L1 = [a,b]
L2 = [c];

L1 = [a,b,c]
L2 = [ ];

no
```

Es sind insgesamt 4 Zerlegungen/backtracking.

Pattern Suche

```
?- conc( Before, [may | After],
[jan,feb,mar,apr,may,jun,jul,aug,sep,oct,nov,dec]
).
Before = [jan,feb,mar,apr]
After = [jun,jul,aug,sep,oct,nov,dec].
```

oder

```
?- conc(_, [Monat1,may,Monat2 |_],
[jan,feb,mar,apr,may,jun,juI,aug,sep,oct,nov,dec]
).
Month 1 = apr
Month2 = jun
```

LÖSCHEN

```
?- L1 = [a,b,z,z,c,z,z,z,d,e], conc( L2, [z,z,z | _], L1).
L1 = [a,b,z,z,c,z,z,z,d,e]
L2 = [a,b,z,z,c]
```

Membership vs. Konkatenation

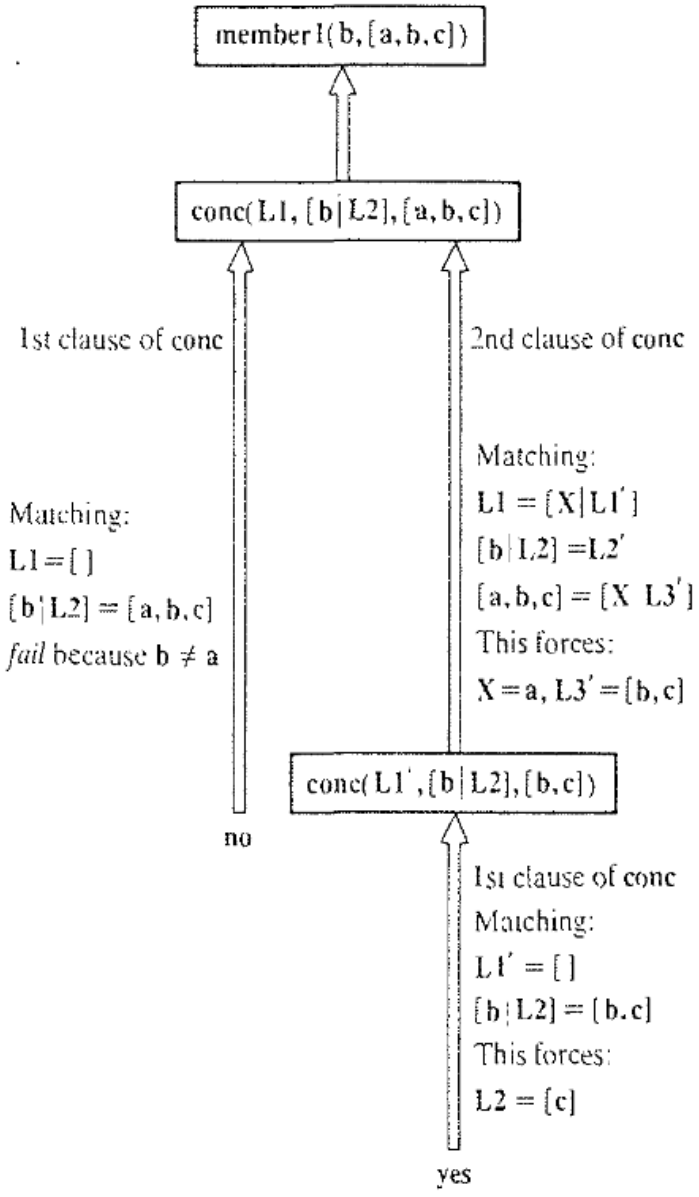
```
member1 ( X, L) :- conc( L1, [X | L2], L).
```

X ist ein ELement der Liste L, falls L in zwei Listen zerlegbar ist, so dass die zweite Liste X als Kopf besitzt.

Mit anonymen Variablen:

```
member1( X, L) :- conc(_, [X |_], L).
```

MEMBER1


member1(b,[a,b,c])
conc(L1,[b|L2],[a,b,c])
1st clause of conc
Matching:
L1 = [ ]
[b|L2] = [a,b,c]
fail because b ≠ a
2nd clause of conc
Matching:
L1 = [X|L1']
[b|L2] =L2'
[a,b,c] = [X L3']
This forces:
X = a, L3' = [b,c]
conc(L1',[b|L2],[b,c])
no
1st clause of conc
Matching:
L1' = [ ]
[b|L2] = [b,c]
This forces:
L2 = [c]
yes

Zugriff auf einzelne Elemente einer Liste

Gesucht ist ein Prädikat das im Regelrumpf auf einzelne Elemente zugreifen kann:
Zugriff auf das erste Element und die zugehörige Definition des Prädikats `get` mit Hilfe des Restlistenoperators | :

```
get1([X|L]) :-
write(X).
```

Zugriff auf das zweite Element die zugehörige Definition des Prädikats `get`:

```
get2([X,Y|L]):-
write(X),write(Y).
```

Beim Zugriff auf das zweite Element sieht man eine wichtige Technik zum Aufspalten einer Liste:
Die Schreibweise: [X1,X2,...,Xn| L] zerlegt eine Liste in die Elemente X1,X2,...,Xn und die Restliste L.

Zugriff auf einzelne Elemente einer Liste

Bei dieser Technik ist zu bemerken, dass nur dann eine Liste zerlegt wird, wenn die Liste mindestens n Elemente besitzt, da der Restlistenoperator | für die Liste L mindestens die leere Liste fordert.

Beispiel

- `get2([1,2,3,4])`
- matched. Zerlegt in X=1, Y=2 und L = [3,4], Ausgabe von 1 und 2.
- `get2([1,2])`
- matched. Zerlegt in X=1, Y=2 und L = [], Ausgabe von 1 und 2.
- `get2([1])`
- matched nicht, da die Liste zu kurz ist.

Zugriff auf alle Elemente einer Liste

Möchte man nicht nur auf die ersten Elemente einer Liste zugreifen, sondern der Reihe nach alle Elemente bearbeiten, muß man einen Weg finden die Liste durchzugehen.
In Prolog verwendet man dazu die Rekursion, was bedeutet, dass der Kopf der Regel im Rumpf wieder aufgerufen wird.
Wenn man nun den Restlistenoperator verwendet um jeweils den Kopf der Liste abzutrennen und die Regel danach mit dem Rest der Liste aufruft, geht Prolog durch die Liste.
Damit die Rekursion nicht zu einer unendlichen Schleife führt, benötigt man Abbruchkriterium das festlegt, wann die Rekursion endet.

Zugriff auf alle Elemente einer Liste

Ausdrucken aller Elemente einer Liste:

```
drucke([]).
drucke([X|L]):-
write(X),
drucke(L).
```

Bei diesem Beispiel sieht man, wie der Kopf einer Liste extrahiert und bearbeitet wird. Der Rest der Liste wird über den rekursiven Aufruf weiter verarbeitet.
Da die Rekursion terminieren muss, ist es notwendig, die wiederholte Zerlegung einer Liste zu terminieren. Dies geschieht mit dem Definieren einer Abbruchregel, die meistens als Argument die leere Liste hat.

Suche nach Elementen einer Liste

Sucht man nach bestimmten Elementen in einer Liste, dann muss man das gesuchte Element als Argument übergeben. Eine erste Lösung könnte wie folgt aussehen:

```
suche(E,[X|L]):-
E=X,
write('gefunden \n').
suche(E,[X|L]):-
suche(E,L).
```

Suche nach Elementen einer Liste

Die Suche nach einem Element geht viel einfacher, wenn der Vergleich auf Gleichheit zweier Terme die Tatsache der Unifizierung auf der Ebene der Argumente auszunutzt.
Man erhält folgende Lösung:

```
member(X,[X|_]):-
write('gefunden \n'),
write(X),
write('\n').
member(X,[_|Y]):-
member(X,Y).
```

Bemerkung:

Die erste Regel wird nur dann aufgerufen, wenn das gesuchte Element X als Kopf der Liste auftaucht.

Suche nach Elementen einer Liste

Die Eigenschaft des Restlistenoperators nur dann mit einer Liste zu matchen, wenn die Liste ausreichend viele Elemente hat wird auch verwendet um zu testen, ob eine Liste eine gerade Anzahl von Argumenten hat:

```
odd([]).
odd([_,_|L]):-
odd(L).
```

Elemente an einer Liste anfügen

Die einfachste Methode eine Element an eine Liste anzufügen ist, das Element an den Anfang einer Liste zu stellen.
Dabei hilft auch hier der Restlistenoperator, aber diesmal nicht als Operator, der eine Liste zerlegt, sondern als direkte Listenkonstruktion:

Anhängen

von X an L führt zu `[X|L]`

Als Prolog Prädikat :

Anhängen von X an L : `add(X,L,[X|L]).`

Elemente an einer Liste anfügen

Sollen die Elemente einer Liste an eine andere Liste angehängt werden, dann muss man auch hier die zu übertragende Liste elementweise zerlegen und jedes Element an die neue Liste anhängen.

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

Elemente aus einer Liste entfernen

`del(X, L, L1)`

$L1$ ist die Liste L ohne das Element X.

1. X ist Kopf der Liste, dann erhalten wir den `tail` der Liste.
2. X ist im `tail`, dann wird X aus dem `tail` entfernt.

```
del( X, [X | Tail], Tail).
del(X, [Y | Tail], [Y | Tail1]) :-
del( X, Tail, Tail1).
```

Teillisten

```
sublist([c,d,e], [a,b,c,d,e,f]) true
```

```
sublist([c,e], [a,b,c,d,e,f]) false
```

Allgemein:

```
sublist(S, L) :-
conc(L1, L2, L),
conc( S, L3, L2).
```

Bestimme alle Teillisten

```
?- sublist(S, [a,b,c]).
S = [ ];
S = [a];
S = [a,b];
S = [a,b,c]; S =[ ];
S = [b];
...
```

PERMUTATIONEN

```
?- permutation([a,b,c], P).
P = [a,b,c];
P = [a,c,b];
p = [b,a,c];
...
```

PERMUTATIONEN

```
?- permutation([a,b,c], P).
P = [a,b,c];
P = [a,c,b];
p = [b,a,c];
...
```

```
permutation( [ ],[ ]).
permutation([X | L], P) :-
permutation(L, L1),
insert(X, L1, P).
```

PERMUTATIONEN

```
permutation2( [ ],[ ]).
permutation2( L, [X | P]) :-
del(X, L, L1),
permutation2( L1, P).
```

Diskussion

```
?- permutation([red,blue,green], P).
P = [ red, blue, green];
P = [ red, green, blue];
P = [ blue, red, green];
P = [ blue, green, red];
P = [ green, red, blue];
P = [ green, blue, red];
no
```

Eine weitere Möglichkeit ist

```
?- permutation(L, [a,b,c] ).
```