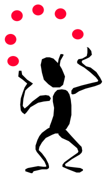
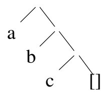
Praktische Programmierung



Listen



[a, b, c]

(a (b (c)))

Listennotation list ::= [ ] | [<head>, <tail>]
Listenkonstruktor list ::= [ ] | .(<head>,<tail>)
Objektnotation list ::= [ ] | [(<term>(,<term>)*)]

Bsp. komm_herein dann sitze dann esse dann tue_nichts

Tue_nichts : leere Liste
Infixoperator: op(500, xfy, dann)

Sortieren von Listen

Ordnungsrelation:
gt(X,Y) :- X > Y. % X größer als Y

sortiert(Liste, Sortiert). % Sortiert entsprechend gt

Bsp. 1

```
bubblesort(Liste,Sortiert) :-
vertausche(Liste, Liste1), !,      % suche zwei benachbarte Elemente X,Y
                                   % in Liste, so dass gt(X,Y) gilt und vertausche
bubblesort(Liste1, Sortiert).      % sortiere die erhaltene Liste1
bubblesort(Sortiert,Sortiert).     % ansonsten ist die Liste bereits sortiert.

vertausche([X,Y|Rest],[Y,X|Rest]) :- gt(X,Y).
vertausche([Z|Rest], [Z|Rest1]) :- vertausche(Rest,Rest1).
```

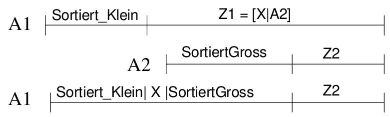
Quicksort

```
quicksort([],[]).
quicksort([X | Schwanz], Sortiert) :-
      teile(X, Schwanz, Klein, Gross),          %Loesche beliebiges X der Liste und
                                                %teile den Rest in Klein und Gross.
      quicksort(Klein, Sortiert_Klein),         %Sortiere Klein,
      quicksort(Gross, Sortiert_Gross),                 %sortiere Gross und
      append(Sortiert_Klein, [X|Sortiert_Gross], Sortiert). %füge sie zusammen.

teile(X, [], [], []).
teile(X, [Y|Schwanz], [Y|Klein], Gross) :- %Füge Erstes Klein hinzu,
      gt(X, Y), !,                              %wenn Erstes < X, und
      teile(X, Schwanz, Klein, Gross).          %teile weiter,

teile(X, [Y|Schwanz], Klein, [Y|Gross]) :-              %sonst füge es Gross hinzu
      teile(X, Schwanz, Klein, Gross).                  %und teile weiter.
```

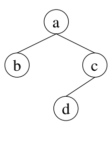

Quicksort mit Differenzlisten




```
quicksort([],[]).
quicksort(L,S) :- quicksort2(L,S,[]).

quicksort2([],Z,Z).
quicksort2([X|Schwanz],A1,Z2) :-          %Loesche beliebiges X der Liste und
      teile(X, Schwanz, Klein, Gross),    %teile den Rest in Klein und Gross.
      quicksort2(Klein, A1,[X|A2]),       %Sortiere Klein,
      quicksort2(Gross, A2,Z2).                   %sortiere Gross
                                          %A1\Z2 ist die sortierte Liste
```


Binärbäume



Gebräuchlichste Darstellung:

- Das Atom nil repräsentiere den leeren Baum
- b sei Funktor des Prädikats b(L,X,R), wobei X die Wurzel und L bzw. R die linken bzw. rechten Teilbäume sind.

Bsp. b(b(nil, **b**, nil), **a**, b(b(nil, **d**, nil), **c**, nil))

Suchen in Binärbäumen

enthalten(X,Baum).

X ist in einem Baum enthalten, wenn
- X die Wurzel des Baums ist, oder
- X im linken Unterbaum von B enthalten ist, oder
- X im rechten Unterbaum von B enthalten ist.

```
enthalten(X, b( _, X, _)).
enthalten(X, b( L, _, _)) :- enthalten(X, L).
enthalten(X, b( _, _, R)) :- enthalten(X, R).
```


Suchbäume

Ist X in einem Suchbaum S(Binärwörterbuch) enthalten?
- Wenn X die Wurzel von S ist, dann ist X gefunden, sonst
- wenn X größer der Wurzel, suche im rechten Unterbaum, sonst
- suche im linken Unterbaum.
- Ist S leer, scheitert die Suche.

```
enthalten_s(X,nil).
enthalten_s(X, b( _, X, _)).
enthalten_s(X, b( _, Y, R)) :- gt(X,Y), enthalten(X,R).
enthalten_s(X, b( L, _, _)) :- enthalten(X,L).

?- enthalten_s(3,b( b(nil,4,nil),6,b(b(nil,7,nil),9,nil))).
```


Operationen auf Suchbäumen

enthalten(X,S)
fuege_hinzu(S, X, S1)
loesche(S, X, S1)

Hinzufügen von X:
- füge X entweder an der Wurzel von S ein, oder
- wenn die Wurzel von S größer als X, füge X im linken Unterbaum von S ein,
- ansonsten füge X im rechten Unterbaum von S ein.

Die **Löschoperation** kann als Umkehrfunktion mit dieser Definition der Hinzufüge-Operation implementiert werden.

Graphen

graph(Knoten, Kanten, Gewicht).

Repräsentationsvarianten:

1) jede Kante als getrennte Klausel
kante(Start,Ziel,Gewicht)

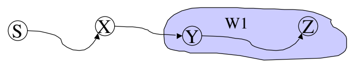
2) den ganzen Graphen als Datenstruktur
G = graph([a,b,c], [kante(a,b), kante(b,d),...])

3) Jeder Knoten wird mit seinen Nachbarn assoziiert.
G = [a->[b], b->[a,c,d], ...]

Suche Weg in einem Graphen

sucheWeg(Start, Ziel, Graph, Weg)

sucheWeg W von S nach Z im Graphen G:
- Wenn S = Z dann W = [S], sonst
- sucheWeg W1 von beliebigem Y nach Z und
  suche Weg von S nach Y unter Vermeidung der Knoten in W1.




```
sucheWeg(S, Z, G, W) :- sucheWeg1(S, [Z], G, W).

sucheWeg1(S, [S|W1], _, [S,W1]).
sucheWeg1(S, [Y|W1], G, W) :-
  benachbart(X, Y, G),                %in G existiert eine kante(X,Y)
  not member(X, W1),                  %Bedingung für Zyklenfreiheit
  sucheWeg1(S, [X,Y|W1], G, W).
```


Finden eines Spannbaums

spannbaum(Graph, Baum).

Rein deklarativer Ansatz:
Ein Baum B spannt den Graphen G auf, wenn
- B eine Teilmenge von G ist, und
- B ein Baum ist, und
- B den Graphen G „überdeckt", d. h. jeder Knoten von G ist auch in B.

Eine Menge von Kanten von B ist ein Baum, wenn
- B zusammenhängend ist, und
- B keine Zyklen enthält.

Bsp. Spannbaum-Suche

```
% Graphen und Bäume sind als Listen von Kanten repräsentiert
spannbaum(G, B) :- teilmenge(G, B),
                   baum(B),
                   ueberdeckt(B, G).
baum(B) :- verbunden(B),
           not hat_zyklus(B).

verbunden(B) :-
        not( (knoten(K1, B), knoten(K2,B),
               not( weg(K1, K2, B)) ).
hat_zyklus(B) :- benachbart(K1, K2, B),
           weg(K1, K2, B, [K1, X, Y|_]).
ueberdeckt(B,G) :- not( (knoten(K1,G), not knoten(K1, B)) ).
teilmenge([], []).
teilmenge([X|L], T) :- teilmenge(L, L1), (T = L1; T=[X|L1]).
```