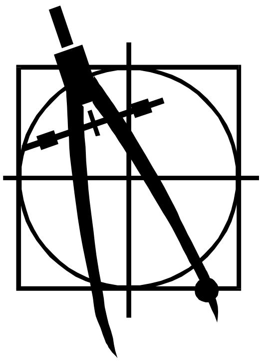
Elementare Programmiertechniken

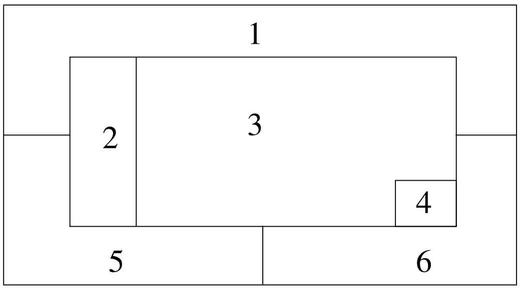


Suchbasierte Berechnungen

```
find(L) :- generate(L), test(L).
```

Lösungsvorschläge aus dem gesamten Lösungsraum des Problems werden fortlaufend erzeugt und jeweils auf ihre Gültigkeit überprüft.

Bsp. Färben einer Landkarte



geg. Farben: rot, gelb, grün, blau

1. Definition der Farben

```
farbe(rot).
farbe(gelb).
farbe(gruen).
farbe(blau).
```

2. Generateprädikat

```
erzeugeFaerbung(L1,L2,L3,L4,L5,L6) :- farbe(L1),farbe(L2), farbe(L3),
farbe(L4), farbe(L5), farbe(L6).
```

3. Testprädikat

```
testFaerbung(L1,L2,L3,L4,L5,L6) :- L1\=L2,L1\=L3,L1\=L5, ...
```

```
?- erzeugeFaerbung(L1,L2,L3,L4,L5,L6),testFaerbung (L1,L2,L3,L4,L5,L6).
```

Bsp. Sortieren von Zahlenlisten

Testprädikat:

```
sortiert([]).                                   % leere Liste ist sortiert
sortiert([E]).                                  % 1-elem. Liste ist sortiert
sortiert([E1,E2|L) :- E1 =< E2, sortiert([E2|L]).
```

Generateprädikat:

```
permutation([],[]).
permutation(L1,[E2|R2]) :- streiche(E2,L1,R1), permutation(R1,R2).
streiche(E,[E|R],R).                            % zu Streichendes ist Erstes
streiche(E,[A|R],[A|RohneE]) :- streiche(E,R,RohneE).
```

Generate-and-Test-Prädikat

```
sortiere(UL,SL) :- permutation(UL,SL), sortiert(SL).
```


Musterorientierte Wissenrepräsentation

„Jede Klausel steht für einen anderen Typ“

Bsp.
Def. in Prolog

```
append([ ], Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :- append (Xs,Ys,Zs).
```

Bsp. Lösungssuche für ?- append([1,2],[3],X).

sub_1= {[1]=X‘, [2]=Xs‘, [3]=Ys‘, X=[X‘|Zs‘]=[[1]|Zs‘]}

append([2],[3],Zs‘).

sub_2= {[2]=X“, []=Xs“, [3]=Ys“, Zs‘=[X“|Zs“]=[[2]|Zs“]} o sub_1

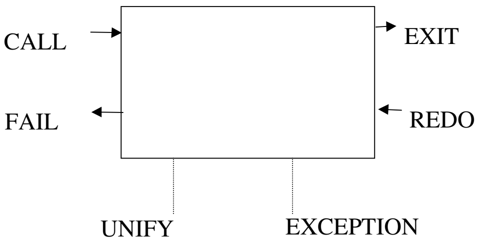
append([],[3],Zs“).

sub_3= {[3]=Ys“=Zs“} o sub_2

□

X = [[1]|Zs‘] = [[1]| [[2]|Zs“]] = [[1]| [[2]|[3]]] = ... = [1,2,3]

Exkurs: Fehlersuche




```
?- visible(+all), trace, app([1,2],[3],X).
   Call:   ( 8)  app([1, 2], [3], _G248) ?
   Unify: ( 8)  app([1, 2], [3], [1|_G359])
   Call:   ( 9)  app([2], [3], _G359) ?
   Unify: ( 9)  app([2], [3], [2|_G362])
   Call:   ( 10) app([], [3], _G362) ?
   Unify: ( 10) app([], [3], [3])
   Exit:   ( 10) app([], [3], [3]) ?
   Exit:   ( 9)  app([2], [3], [2, 3]) ?
   Exit:   ( 8)  app([1, 2], [3], [1, 2, 3]) ?
X = [1, 2, 3]
Yes

[debug]  ?- nodebug.
Yes
?-
```


Verwenden von Relationen

```
append([], L,L).
append([E|RestL1],L2,[E|RestL2]) :- append(RestL1,L2,RestL2).

?- append([1],[2,3],[1,2,3]).        % Relation erfüllt?
?- append([1],[2,3],L).              % Funktion
?- append([3],L1,[1,2,3]).           % Umkehrfunktion

last(Liste,Element)  :- append(ListeErg,Element,Liste).

enthaelt(L,E) :- append(L1,[E|L2],L).
teilliste(T,L)   :- append(L1,L2,L), append(T,L2ohneT,L2).
```

- Nicht funktional, sondern in Relationen denken
- Prädikate habe keine Ein- / Ausgabeparameter
- Zur Definition neuer Prädikate vorhandene Prädikate nutzen und
- neue Prädikate möglichst allgemein definieren

Akkumulatortechnik

Bsp. Spiegelung einer Liste

Standarddefinition (Top-Down):

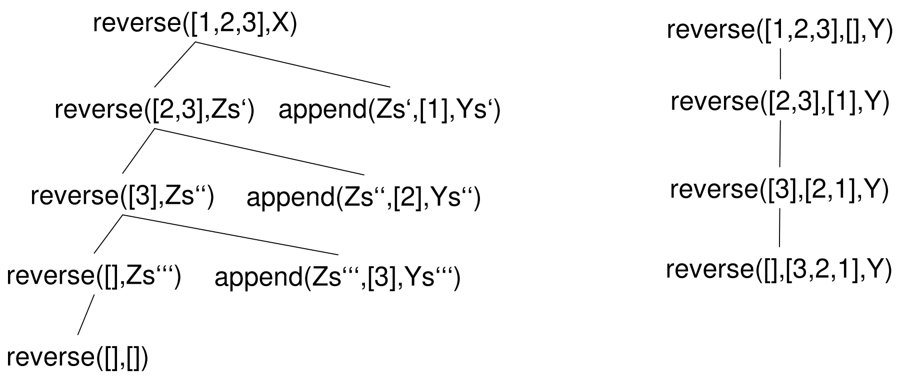
```
reverse([],[]).
reverse([X|Xs],Ys) :- reverse(Xs,Zs), append(Zs,[X],Ys).
```

mit Akkumulatortechnik (Bottom-Up):

```
reverse(Xs,Ys) :- rev(Xs,[],Ys).
rev ([X|Xs],R,Ys) :- rev(Xs,[X|R],Ys).
rev ([],R,R).
```

Akkumulator R

Vergleich der Definitionen:



Differenzlisten

(Einsatz Partieller Strukturen)

[X | LV] \ LV

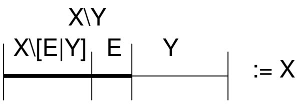
Lückenvariable

Bsp. [1,2,3] ≡ [1,2,3,4,5] \ [4,5]

Anwendungsbeispiel: Anhängen einer Liste L1 an eine Liste L2

bisher:
```
append([], L,L).
append([E|RestL1],L2,[E|RestL2]) :- append(RestL1,L2,RestL2).
```

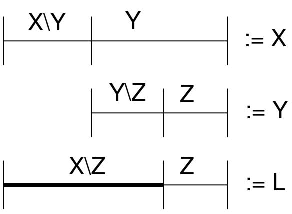
1. Überlegung: Anhängen <u>eines</u> Elementes an eine Liste L1 ≡ [L1,E|Y]\[E,Y] ≡ X\[E|Y]



$$\text{insertAsLast\_dl}(E,\ \underbrace{X,\ [E|Y]}_{X\backslash[E|Y]},\ \underbrace{X,\ Y}_{X\backslash Y}).$$

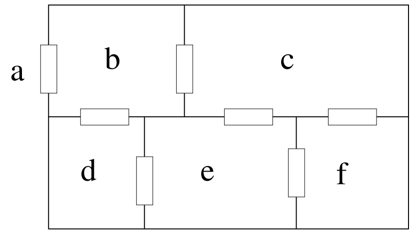
Forts. Differenzlisten

2. Schritt: Erweiterung der Überlegung auf beliebige Listen



$$\text{append\_dl}(\underbrace{X,Y}_{X\backslash Y},\underbrace{Y,Z}_{Y\backslash Z},\underbrace{X,Z}_{X\backslash Z}).$$

Datenstrukturen als Fakten



Wissensrepräsentation in der Datenstruktur Liste:

[tuer(a,b), tuer(b,c), tuer(b,d), ...]

• aufwendige Listenoperationen notwendig

Wissensrepräsentation als Fakten:

tuer(a,b).
tuer(b,c). usw.

• Problemdarstellung einfacher
• Nachteil: Modifikation der Daten nur mit nicht-log. Konstrukten möglich

zum Bsp. Labyrinth

!! tuer(X,Y) :- tuer(Y,X). Problem: unendliche Beweisketten möglich

Entweder die andere Varianten als Fakten angeben oder bei der Lösungsfindung berücksichtigen:

```
gehe(X,X).
gehe(X,Y) :- tuer(X,Z), gehe(Z,Y).
gehe(X,Y) :- tuer(Z,X), gehe(Z,Y).
```

Auch hier droht die Gefahr von Endlosschleifen.
Daher: notiere die besuchten Räume in einer Liste.

```
gehe(X,X,Besucht).
gehe(X,Y,Besucht) :- tuer(X,Z), member(Z,Besucht), gehe(Z,Y).
gehe(X,Y,Besucht) :- tuer(Z,X), member(Z,Besucht), gehe(Z,Y).

?- gehe(a,f,[]).
yes
```

Anmerkungen zum Programmierstil

Kriterien für ein gutes Programm (Ziele):

Korrektheit, Effizienz, Transparenz, Lesbarkeit, Modifizierbarkeit, Robustheit, Dokumentation

Methoden:

- erst formulieren, dann codieren
- schrittweise Verfeinern
- Identfiziere den Grenz- und den Allgemeinfall einer rekursiven Definition
- Allgemeinere Probleme sind evtl. leichter zu lösen, als spezielle.
- Anwendung graphischer Darstellungen zur Problemlösung

Stil:

- Programmklauseln kurz halten
- Verwendung aussagekräftiger Bezeichner für Objekte
- Strukturierung des Programms mit Leerzeilen und -zeichen
- ausreichende Kommentierung