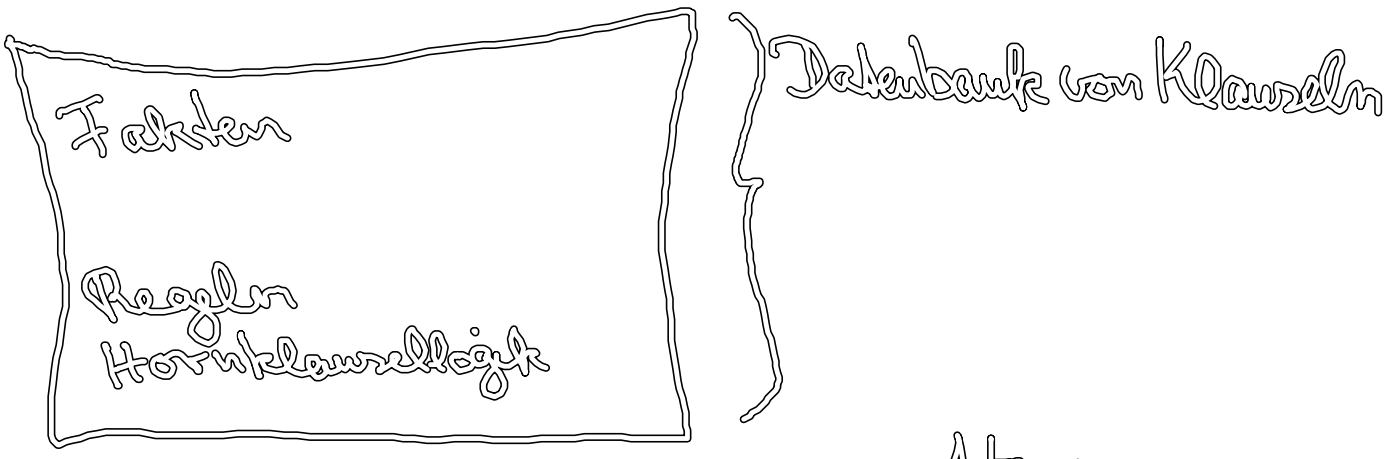
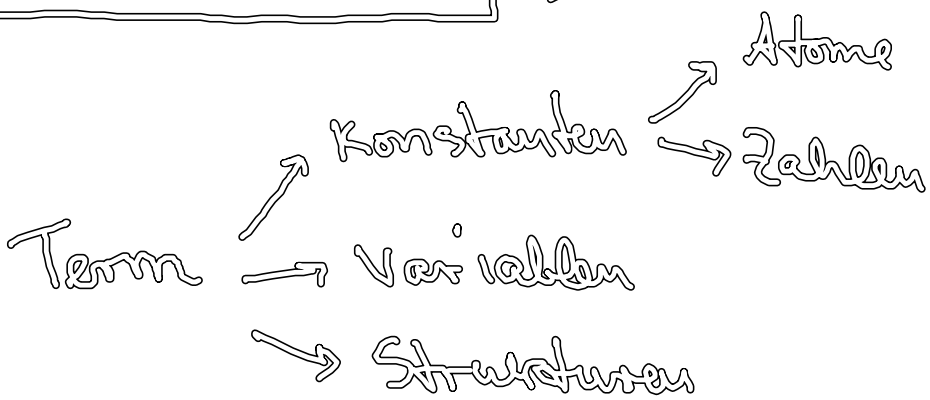
Prolog





deutsch	Prädikaten-logik	Prolog
und	$\wedge$	,
oder	$\vee$	;
wenn, dann	$\leftarrow$	:-
nicht	$\neg$	not

Annahme der Weltabgeschlossenheit
(World Closed Assumption) oder
negation as failure.

Prolog: Alles ist falsch, dessen Gegenteil
nicht als wahr bewiesen
werden kann.

Das gesamte Wissen über die Welt
(Domäne) befindet sich in der DB.

Bsp:

```
mag(peter, susanne):-.
not(mag(karin, peter)).
```

Arbeiten mit Prolog

? assert(P).

Prädikat P zur Menge
der Spezifikationen
hinzufügen

? asserta(P)

P an den Anfang

?- assertz(P) P an das Ende

?- retract(P) P wird entfernt

sehr mühsam, daher Prologskript

Listen

geordnete Menge von Elementen (die auch wieder Listen sein können)

[a,b,c,d]

[[1,2],[3,4]]

[a,[1,2],b,c]

[]

head & tail - Prinzip wie bei Haskell

[H|T]

[X, Y | T]

[a, b, c]

[X | Y]

X = a

Y = [b, c]

member - Prädikat

? member(a, [a, b, c]).

yes

? member(X, [b, c]).

X = b ;

X = c ;

no

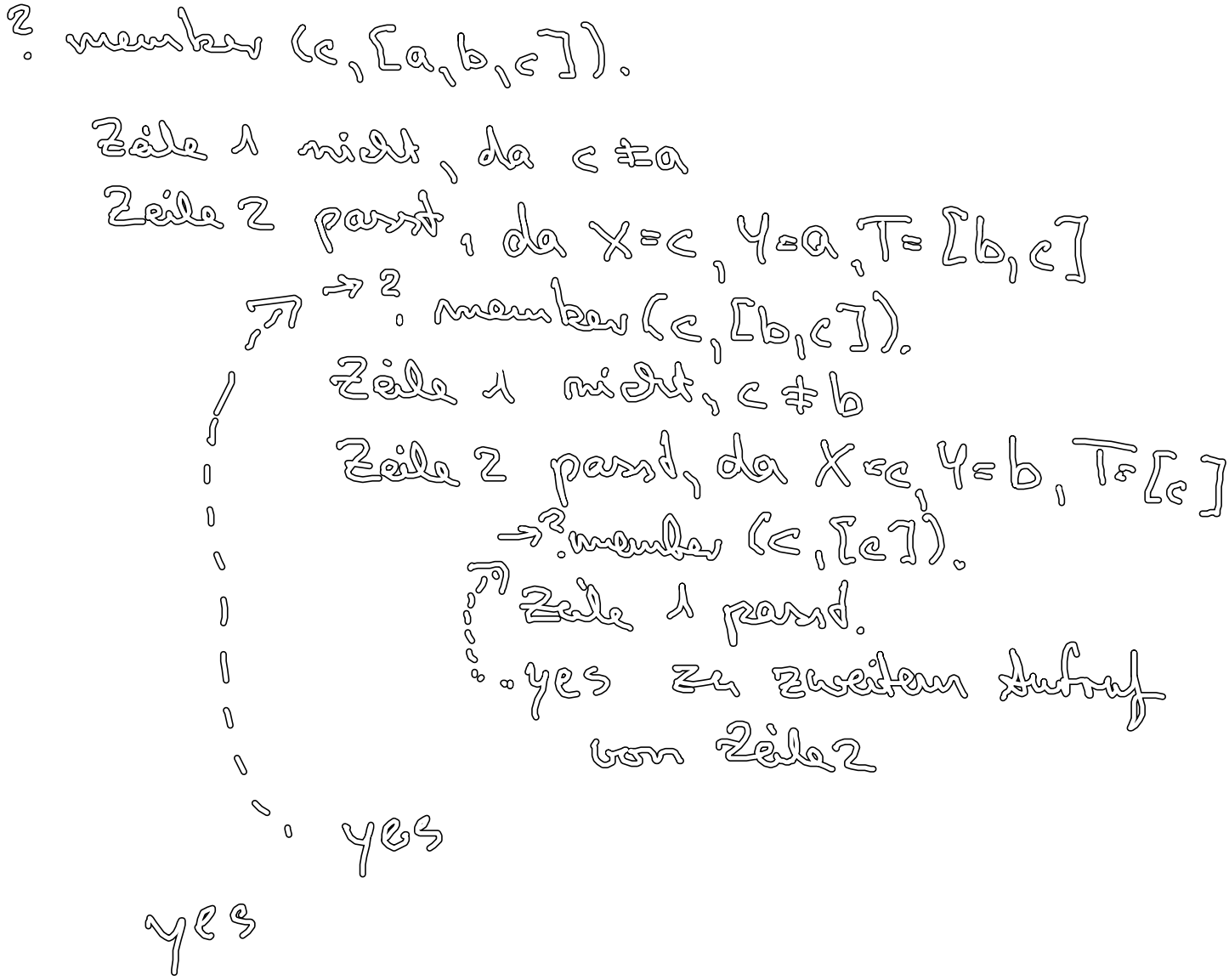
? member(X, []).

no

rekursive Definition von member

1) member(X, [X|T]).

2) member(X, [Y|T]) :- member(X, T).



anonyme Variablen

```
member(X, [X|_]).
member(X, [_|T]) :- member(X, T).
```

Länge einer Liste

laenge([],0).
laenge([_,T],N)
:- laenge(T,N1), N is 1+N1.

?- X=1+2.
X = 1+2

?- X is 1+2
X=3

?- laenge([a,b,c],N).
N=3 ;
no

Arithmetische Operatoren

+, -, *, /, **, //, mod

a**b = a^b

Integer div

5 // 2 = 2

Vergleichsoperatoren

>, <, >=, =<, =:=, =\=

= ≠

Ausgaben in Prolog

Bsp: Ausgabe der Listenelemente unter einander

```
output_list([]).
output_list([H|T]) :- write(H),
                      nl, % Zeilenumbruch
                      output_list(T).
```

tab(4)

```
wg_ol([]).
wg_ol([H|T]) :- wg_ol(T),
                wink(H).
```

Konkatenation von Listen

```
concat([1,2],[3,4],[1,2,3,4]).

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

?- concat([g,o],[o,g,l,e],L).
L = [g,o,o,g,l,e] ;
no

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

```
L2 = [b,c],
 :
 :
```

so

```
member(X,L) :- concat(L1,[X|L2],L).

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

Permutation von Listen

zunächst del

```
del(X,[X|Tail],Tail).
del(X,[Y|Tail1],[Y|Tail2])
    :- del(X,Tail1,Tail2).
```

```
insert(X,List,List_mit_X)
    :- del(X,List_mit_X,List).
```

```
perm([],[]).
```

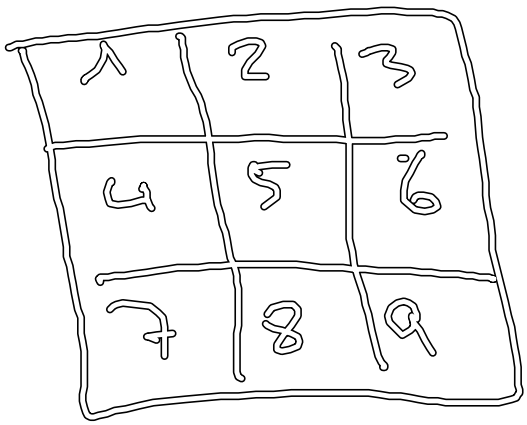
```
perm([X|L],P)
    :- perm(L,L1), insert(X,L1,P).
```

?- perm([a,b,c], Permlist).

Permlist = [a,b,c] ;
⋮

} n! Perm für Liste der Länge n

Springerproblem 3×3



zug(1,6)
zug(1,8)

prädikatenlogisch

„Weg der Länge 2"

$\forall x,y\ [weg(x,y) \leftarrow \exists z [zug(x,z) \wedge zug(z,y)]]$

allg.: weg

weg

$\forall x\ weg(x,x)$

$\forall x,y\ [weg(x,y) \leftarrow \exists z [zug(x,z) \wedge weg(z,y)]]$

Prolog:

```
zug(1,3).
zug(1,6).
  :
zug(9,4).

weg(Z,Z).
weg(X,Y) :- zug(X,W), not(member(W,[X,Y])), weg(W,Y).
```

```
weg(Z,Z,L).
weg(X,Y,L) :- zug(X,W),
              not(member(W,L)),
              weg(W,Y,[W|L]).
```

```
weg(X,Y) :- weg(X,Y,[X]).
```

Der Prädikat Cut

Cut wird durch "!" - Operator repräsentiert.
Beim erstenmal erfolgreich ausgeführt und bei Backtracking mißlingt das gesamte Ziel.

Sprung beispiel

```
weg(Z,Z,_).
weg(X,Y,L) :- zug(X,Z),
              not(member(Z,L)),
              weg(Z,Y,[Z|L]),
```

1

bestenfalls eine Lösung wird produziert

Cut hat Nebeneffekte

Choicepoints werden eingespart

Ausführung beschleunigt

Bsp:

```
p :- a,b.
p :- c.
```

$(a \wedge b) \vee c \Leftrightarrow p$

```
p :- a,!,b.
p :- c.
```

$(a \wedge b) \vee (\neg a \wedge c) \Rightarrow p$

```
p :- c.
p :- a,!,b.
```

$c \vee (a \wedge b) \Rightarrow p$

max-Beispiel

```
max(X,Y,X) :- X >= Y.
```

```
max(X,Y,Y) :- X<Y.
```

mit cut

```
max(X,Y,X) :- X>=Y, !.
max(_,Y,Y).
```

```
?- max(5,7,M).
?- max(7,5,M).
?- max(8,1,1).
```

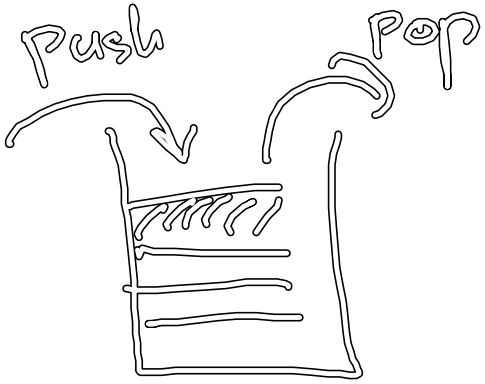
```
max(X,Y,Max) :- X>=Y, !, Max=X;
                Max=Y.
```

```
if (X>=Y) then Max=X else Max=Y
```

```
max(X,Y,Max) :- X>=Y, !, Max=X.
max(X,Y,Max) :- Max=Y.
```

Abstrakte Datentypen in Prolog

Stack



Methodes

isempty

push

pop

peek

member stack(x)

```
empty-stack([]).

member_stack(E,S) :- member(E,S).

stack(E,S,[E|S]).

push
?- stack(a,[b,c],S).
S = [a,b,c];
no

pop
```

?- stack(Top, Rest, [a,b,c]).

Top = a,

Rest = [b,c]

Peek

?- stack(Top, _, [a,b,c]),

Top = a