Fallunterscheidung

```
case(...)  :-  condition-1(...),declaration-1(...).
case(...)  :-  condition-2(...),declaration-2(...).
...
case(...)  :-  condition-n(...),declaration-n(...).
```

- Mit Backtracking, ggf. weitere Regel bearbeiten

```
case(Geld,Essen) :- Geld>500,adlon(Essen).
case(Geld,Essen) :- Geld>50,steakhouse(Essen).
case(Geld,Essen) :- Geld>5,doener(Essen).
case(Geld,Essen) :- selberkochen(Essen)
```


Fallunterscheidung mit cut

```
case(...) :- condition-1(...), !, declaration-1(...).
case(...) :- condition-2(...), !, declaration-2(...).
...
case(...) :- condition-n(...), !, declaration-n(...).
```

- Ohne Backtracking, höchstens eine Regel bearbeiten

```
case(Geld,Einladung) :- Geld>500, !, adlon(Einladung).
case(Geld,Einladung) :- Geld>50,  !, steakhouse(Einladung).
case(Geld,Einladung) :- Geld>5,   !, doener(Einladung).
case(Geld,Einladung) :-           !, selberkochen(Einladung)
```

„If then else“

CWA (2)

Wann soll Interpreter Antwort „no“ auf Anfrage Q liefern?

Logische Varianten:
Wenn ¬Q bewiesen wurde.
Wenn Q nachweisbar nicht bewiesen werden kann.
Wenn alle Beweisversuche für Q fehlgeschlagen sind.
Dabei jeweils implizite Annahmen bzgl. Negation.

Nicht-logische Varianten:
Wenn die verfügbaren Argumente für „nicht Q“ sprechen.
Wenn die verfügbaren Argumente gegen Q sprechen.
Im Zweifelsfall für den Angeklagten ...

CWA (2)

In Prolog:
Variante 3 **„Negation by (finite) failure“**
Wenn alle Beweisversuche für Q fehlgeschlagen sind.

Bedeutung der Antwort „no“:
Alle Beweisversuche sind fehlgeschlagen.

Probleme:
In PK1: Kein allgemeines Verfahren
(Nicht-Allgemeingültigkeit ist nicht aufzählbar)

Unterschiede zur logischen Negation

Negation by failure

Was würde bedeuten:
Wenn H nicht aus Programm P beweisbar ist,
wird angenommen, dass ¬H beweisbar ist.

Unterstellung von
„Genau eins von beiden, H oder ¬H , ist aus P beweisbar.“

bedeutet in der Logik
Vollständigkeit (im Sinne der klassischen Logik!)
– für alle Aussagen H gilt: H oder ¬H ist aus P beweisbar
Korrektheit
– für alle Aussagen H gilt:
H und ¬H sind nicht beide aus P beweisbar

Negation by failure

„Genau eins von beiden, H oder ¬H , ist aus P beweisbar.“

Trifft in Prolog i.a. nicht zu:
Weder male(fritz)
noch ¬male(fritz)
folgen logisch aus
unserem Programm

```
parent(gaea,
parent(gaea,
parent(rhea,
parent(cronus
parent(rhea,
parent (cronu
parent (cronu
parent(rhea,
parent(cro
parent(rhe
```

```
female(gaea).
female(rhea).
female(hera).
female(hestia).
female(demeter).
female(athena).
female(metis).
```

```
male(uranus).
male(cronus).
male(zeus).
male(hades).
male(hermes).
male(apollo).
male(dionysius).
male(hephaestus).
male(poseidon).
```

Inhaltlich ist Verfahren von Prolog oft sinnvoll

Operator **not** : Negation by failure

Ein subgoal not(p1(X1...,Xn)) gelingt in Prolog,
falls p1(X1...,Xn) nicht bewiesen werden kann.

```
?- not(male(fritz)).
    yes
```

```
male(X):- not(female(X)).
```

```
?- male(rhea).
   no
```

Nicht erlaubt: negierter Klauselkopf not (p) :- ...

not mittels cut implementierbar

Kombination von cut und fail:

```
verschieden(X,Y) :- X=Y, !, fail.
verschieden(X,Y) .

male(X) :- female(X), !, fail.
male(X) .
```

not/1 wäre implementierbar durch

```
not(P) :- P, ! , fail.
not(P) .
```


Negation by failure: Diskussion

Vollständiger Kalkül müsste negative Prädikate in gleicher Weise wie positive Prädikate behandeln können:

- In Klausel g :- g_1,...,g_n dürften als subgoals g_i
  positive Literale p(X1,...,Xn) oder
  negative Literale ¬p(X1,...,Xn) auftreten.
- Beide Formen mit der Interpretation dass Belegungen der Variablen gesucht werden, die g_i erfüllen

Tatsächlich aber nur für positive Literale realisiert.

not (p(X1,...,Xn)) bedeutet dagegen,
dass es keine Belegung gibt, die p(X1,...,Xn) erfüllt.

Negation by failure: Diskussion

Unterschied:
¬Q(X1,...,Xn) beweisen
Q(X1,...,Xn) nicht beweisen können

Insbesondere auch:

Misserfolg von „not(Q)“ ohne Backtracking in Q .

Negation by failure: Diskussion

Veränderte Semantik:

```
a:-not(b).
b.
```

```
?-not (a).
yes
```

Aber ¬ a folgt nicht aus {¬b →a, b }

```
a:-not(b).
```

```
?-a.
yes
```

Aber a folgt nicht aus {¬b →a }

Negation by failure: Diskussion

Man kann positive/negative Fakten separat benennen:
Beispiel: `female` und `male` (= nicht `female`)

Aber kein logischer Zusammenhang zwischen beiden:
Anfragen

```
?- female(fritz).
?- male(fritz).
```

führen beide zu „`no`“ , falls `fritz` nicht in Datenbasis.
Anfrage

```
?- not(female(fritz)).
?- not(male(fritz)).
```

führen beide zu „`yes`“ , falls `fritz` nicht in Datenbasis

Negation by failure: Diskussion

Falls `fritz` nicht in Datenbasis:

```
?- male(fritz).
no
?- not(male(fritz)).
yes
```

Wäre gleichzeitig definiert

```
male(X) :- not(female(X)).
```

wären alle nicht bekannten Objekte „männlich“:

```
? - male(fritz).
yes
```


„not“ führt keine Bindungen aus.

Keine Variablenbindung durch not .
Nicht im Sinne existentieller Anfragen verwendbar.

Problem bei Verwendung für nicht gebundene Variable

```
male(X):-not(female(X)).
?-male(X).
no.
```

Solange `female(X)` beweisbar.

Es würde funktionieren bei vorheriger Bindung, z.B.

```
..., human(X),male(X),...
```


Negation by failure: Diskussion

```
male(X) :- not(female(X)).
```

Abarbeitung von `?-male(X)` als $\forall X\ (\neg female(X))$
d.h. $\neg\ \exists X\ (female(X))$
statt $\exists X\ (male(X))$

Unterschiedliche Semantik bei logischer Äquivalenz :

```
r1:-male(X),human(X).
r2:-human(X), male(X).
female(anna).
human(fritz).
```

r1 und r2 sind logisch äquivalent.

```
?-r1.
no
```

```
?-r2.
yes
```


CWA (2): Negation by **finite** Failure

not(Q) gelingt (Antwort „yes“) , falls Q misslingt (Antwort „no“)
not(Q) misslingt, (Antwort „no“), falls Q gelingt (Antwort „yes“)

Wann kann Interpreter Antwort „not(Q)“ bestätigen?

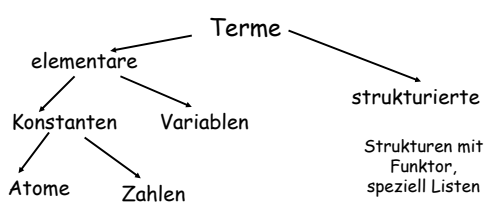
Gemäß *Variante 3*:
Wenn alle Beweisversuche für Q fehlgeschlagen sind.

Alle Beweisversuche müssen nach endlicher Zeit geprüft sein. Nicht möglich bei unendlichem Und-oder-Baum.

Finite failure

Bedeutung der Antwort „no“:
Alle Beweisversuche sind fehlgeschlagen.

Prolog Syntax: Terme



Prolog Syntax: Terme

Atome: beginnen mit kleinen Buchstaben
`thomas`, `lisa`, `robert`

Zeichenketten: in Hochkommata eingeschlossen
`` `Thomas` ``, `` `Lisa` ``, `` `Robert` ``

Zahlen: natürliche und reelle Zahlen
`26`, `5.7E-3`, `-1.25`

Variable:
beginnen mit großem Buchstaben oder Unterstrich
`X`, `Thomas`, `_21`,
`_` (anonyme Variable)

Prolog Syntax: Terme

zusammengesetzte Terme (Strukturen):

kleingeschriebener Funktor, Argumente

– *Funktor* charakterisiert durch Name + Stelligkeit:
`name/i`
Unterschied: `punkt(3,4)`, `punkt(12,5,7)`

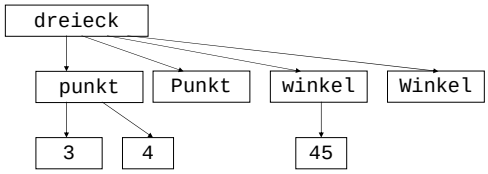
– Argumente sind Terme (rekursiv!)

`punkt(3,4)`, `dreieck(X,Y,Z)`,
`dreieck(punkt(3,4),Punkt,winkel(45),Winkel)`

Prolog Syntax: Terme

Darstellung von Strukturen als Bäume.

`dreieck(punkt(3,4),Punkt,winkel(45),Winkel)`

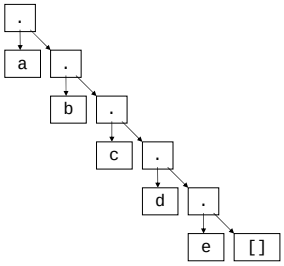


Prolog Syntax: Terme

Darstellung von Strukturen als Bäume.

[a,b,c,d,e] = [a,[b,[c,[d,[e,[]]]]]] = .(a, .(b, .(c, .(d, .(e, []))))

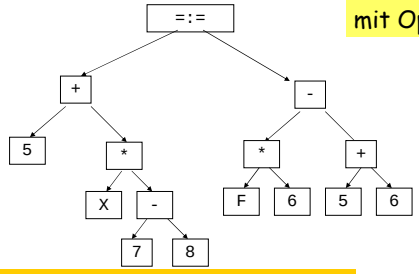
Spezielle Notationen für Listen



Prolog Syntax: Terme

Spezielle Notationen mit Operatoren

5 + X * (7 – 8) =:= F * 6 - (5 + 6)



=:=(+(5, *(X, -(7,8))), -(*(F,6), +(5,6)))

Prolog-Syntax: Programm

Programm: Menge von Prozeduren
Prozedur: Folge von Klauseln mit gleichem Kopf-Funktor
Klauseln: Fakten oder Regeln
Fakt: Prädikat
Regel: Kopf :- Körper
Kopf: Prädikat
Körper: Folge von Teilzielen
Teilziel: Prädikat
Prädikat: Funktor-Name und Liste von Argumenten
Argumente: Terme

Parameter unterscheiden in
- Eingabe-Parameter
- Ausgabe-Parameter
- beliebig

Fakten und Regeln sind auch Strukturen

Prolog-Syntax: Programm

Die Antwort auf alle Fragen erhält man mit

```
? - X .
```

Listen

Endliche geordnete Menge von Elementen
(endliche Folge, Sequenz, Wort)

Schreibweise: [a,b,c,d,e]
[*2, 34, 7, 13, 4, 2, 3, 13*]
[*S*, *t*, *u*, *d*, *e*, *n*, *t*, *i*, *n*, *n*, *e*, *n*]

Mathematisch definierbar als
Abbildung L von {1,...,n } in Elemente-Menge

L = [*L*(1), *L*(2), ..., *L*(*n*)]

oder als ineinander verschachtelte Mengen:
{ *L*(1), {*L*(2), { ..., {*L*(*n*), { } } ... }}}

Listen

Operationen (Methoden) mit Listen:
- Verketten von Listen
- Zugriff auf Elemente
  - einfügen
  - suchen
  - löschen
- Reorganisation (z.B. Sortieren)
- Anzahl der Elemente

Listen

Implementation z.B. als
- Feld von Elementen
- Einfach verkettete Liste: Referenz auf Nachfolger
- Doppelt verkettete Liste: vorwärts-/rückwärts-Referenz

in JAVA z.B.:
- Klasse `String` für Zeichenketten:
  Liste implementiert als Feld von Zeichen: `char[]`
- Klasse `Vector` für Listen allgemein

Listen

Rekursive Definition
- Die leere Liste (NIL oder []) ist eine Liste.
- L ist eine Liste, wenn sie ein erstes Element (**head**) besitzt und der Rest (**tail**) wieder eine Liste ist.

```
liste([ ]).
liste(L)  falls  liste(tail(L)).
```

head([a,b,c,d,e]) = a tail([a,b,c,d,e]) = [b,c,d,e]

[a,b,c,d,e] = [a,[b,c,d,e]] = [a,[b,[c,d,e]]] = [a,[b,[c,[d,e]]]]
= [a,[b,[c,[d,[e]]]]] = [a,[b,[c,[d,[e,[]]]]]]

Listen in Prolog: Funktor „ ./2 “

Struktur . (Element, Liste)
dient zur rekursiven Beschreibung von Listen:

. (Kopf, Restliste) beschreibt die Liste [Kopf, ...(Rest)...]
.(a,.(b, .(c, .(d, .(e, [])))))) = [a,b,c,d,e]

Prolog erlaubt weitere Schreibweisen:
[Kopf | Restliste]
[Element_1, ..., Element_n]
[Element_1, ..., Element_i | Restliste_ab_i+1]

```
[a,b,c,d,e]
= [a | [b,c,d,e] ]
=[a,b | [c,d,e] ]
= ...
=[a,b,c,d,e | [] ]
```


member-Prädikat

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

?- member(a, liste). Ist a Element von liste?

?- member(X, liste). Welche Elemente hat liste?

?- member(a, L). Welche Listen besitzen a als Element?

Variante für einmalige Antwort:

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

append-Prädikat

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

```
?- append(liste1, liste2, L3).
?- append(liste1, L2, liste3).
?- append(L1, liste2, liste3).
?- append(L1, L2, liste3).
```


Anwendungen für append-Prädikat

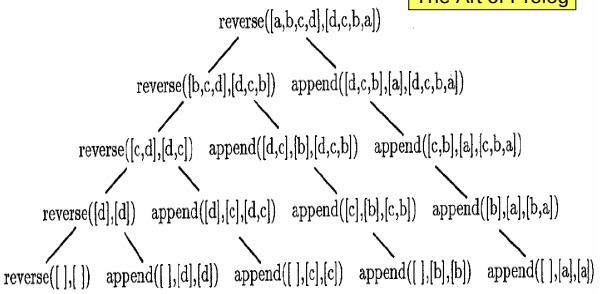
```
prefix(P, L ) :- append(P,L2,L).
suffix(S, L ) :- append(L1,S,L).
sublist(SL,L ) :- prefix(P,L), suffix(SL,P).

member(X,L) :- append(P, [X | S], L).

naive_reverse([ ],[ ]) .
naive_reverse( [H | T], R) :- naive_reverse(T,R1),
                              append(R1,[H], R).
```


Naive Reverse

Aus Shapiro: The Art of Prolog



Anwendungen für append-Prädikat

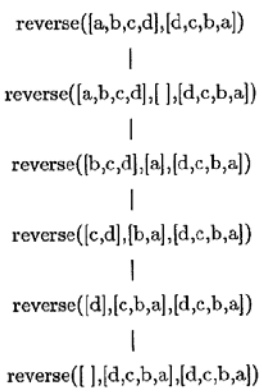
Effizientere Implementation für reverse
Mit Hilfe eines „Akkumulators“ (Zwischenspeicher):

```
reverse( L,R ) :- reverse_acc( L, [ ], R ).

reverse_acc( [ H| T], Acc, R ) :- reverse_acc( T, [H | Acc] , R ) .
reverse_acc( [ ], R, R ).
```


Reverse mit Akkumulator

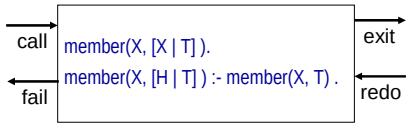
Aus Shapiro: The Art of Prolog



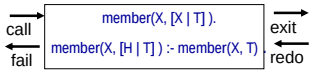
Box-Modell, trace/0

trace/0
Verfolgen des nächsten Beweisversuchs
in der Darstellung des Boxmodells.

Box modelliert Bearbeitung einer Prozedur
durch 4 Ports (Ereignisse) für Betreten/Verlassen



Box-Modell, trace/0



Call: Start des Beweises einer Prozedur
(erster Beweisversuch: erste Klausel).
Exit: Erfolgreicher Beweis.
Redo: Alternativer Beweisversuch (Backtracking).
- Backtracking im Beweis der Klausel
- bzw. bei deren endgültigem Fehlschlag:
  Beweisversuch mit nächster alternativer Klausel.

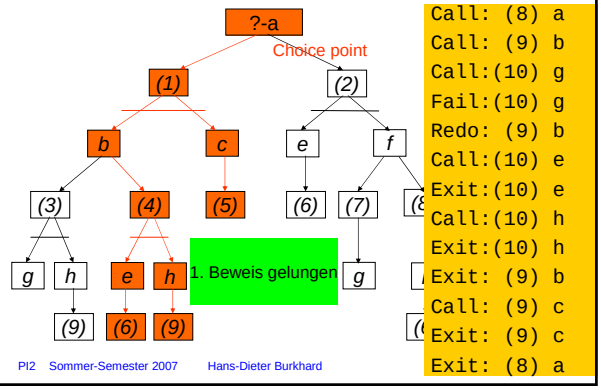
Fail: Keine weiteren Beweismöglichkeiten für Prozedur.

Box-Modell, trace/0

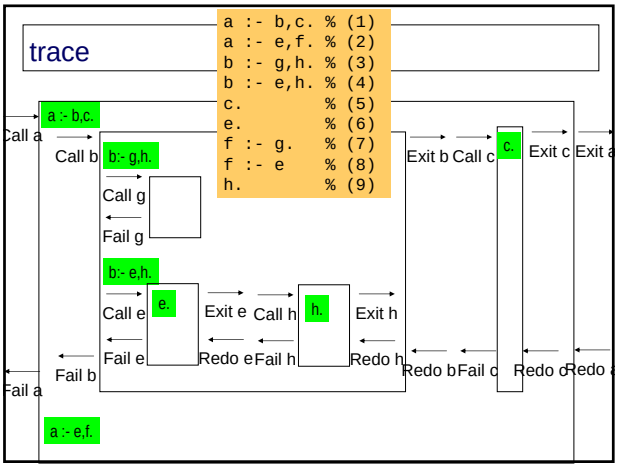
Reihenfolge der Boxen graphisch:
- Subgoals einer Klausel sequentiell
- Klauselaufrufe: ineinander verschachtelt

Reihenfolge der Boxen im trace:
- sequentiell gemäß Abarbeitung im Und-oder-Baum
- mit Nummerierung gemäß Ebenen im Und-oder-Baum

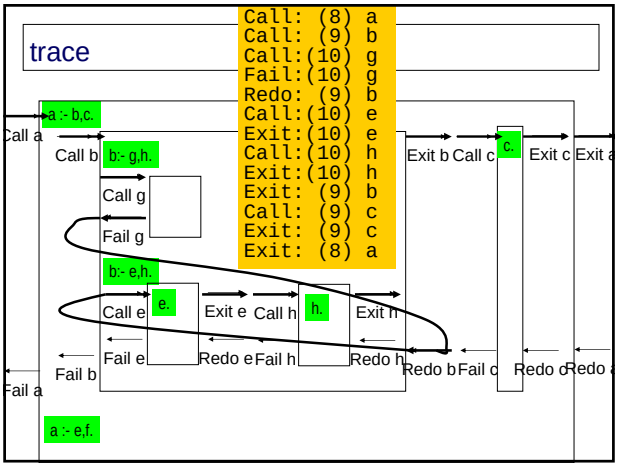
trace



trace



trace



Debugger

Mit debug/0 wird debug-Modus gestartet:
- Anzeigen von Trace-Punkten
- Unterbrechung an Spy-Punkten
  mit Möglichkeit zur interaktiven Arbeit
- Setzt ggf. Optimierungen außer Kraft
- Impliziter Aufruf durch spy bzw. trace
- Abschalten mit nodebug/0

Debugger

trace/2 setzt/löscht Trace-Punkte
- für Prädikate (1.Argument)
- an angebenen Ports (2.Argument):
  +portname bzw. –portname bzw. Liste

spy/1 bzw. nospy/1 setzt/löscht Spy-Punkte
- für angegebene Prädikate

trace/0 setzt überall Trace-Punkte für folgenden Aufruf

debugging/0 zeigt Status

Variable/Parameter in Prolog

Imperative Programmiersprachen
- Überschreiben von Variablen
- Parameterübergabe bei Prozedur-Aufruf über
  - Wert (value-Parameter)
  - Referenz (reference-Parameter)

Prolog
- Dauerhafte Bindung (Instantiierung) von Variablen
- Parameter-Übergabe durch Unifikation
  - Referenzierung an Variable bzw. Wert

Variable/Parameter: Verzögerte Berechnung

```
[trace]  ?- append1([a,b,c],[1,2],L).
   Call: (6) append1([a, b, c], [1, 2], _G294)
   Call: (7) append1([b, c], [1, 2], _G366)
   Call: (8) append1([c], [1, 2], _G369)
   Call: (9) append1([], [1, 2], _G372)
   Exit: (9) append1([], [1, 2], [1, 2])
   Exit: (8) append1([c], [1, 2], [c, 1, 2])
   Exit: (7) append1([b, c], [1, 2], [b, c, 1, 2])
   Exit: (6) append1([a, b, c], [1, 2], [a, b, c, 1, 2])

L = [a, b, c, 1, 2]
```


Variable/Parameter: Verzögerte Berechnung

```
[trace]  ?- naive_reverse([a,b,c],L).
   Call: (6) naive_reverse([a, b, c], _G287)
   Call: (7) naive_reverse([b, c], _G356)
   Call: (8) naive_reverse([c], _G356)
   Call: (9) naive_reverse([], _G356)
   Exit: (9) naive_reverse([], [])
   Call: (9) append1([], [c], _G360)
   Exit: (9) append1([], [c], [c])
   Exit: (8) naive_reverse([c], [c])
   Call: (8) append1([c], [b], _G363)
   Call: (9) append1([], [b], _G358)
   Exit: (9) append1([], [b], [b])
   Exit: (8) append1([c], [b], [c, b])
   Exit: (7) naive_reverse([b, c], [c, b])
   Call: (7) append1([c, b], [a], _G287)
   Call: (8) append1([b], [a], _G364)
   Call: (9) append1([], [a], _G367)
   Exit: (9) append1([], [a], [a])
   Exit: (8) append1([b], [a], [b, a])
   Exit: (7) append1([c, b], [a], [c, b, a])
   Exit: (6) naive_reverse([a, b, c], [c, b, a])
L = [c, b, a]
```

Variable/Parameter: Verzögerte Berechnung

```
[trace]  ?- reverse1([a,b,c],L).
   Call: (6) reverse1([a, b, c], _G287)
   Call: (7) reverse_acc([a, b, c], [], _G287)
   Call: (8) reverse_acc([b, c], [a], _G287)
   Call: (9) reverse_acc([c], [b, a], _G287)
   Call: (10) reverse_acc([], [c, b, a], _G287)
   Exit: (10) reverse_acc([], [c, b, a], [c, b, a])
   Exit: (9) reverse_acc([c], [b, a], [c, b, a])
   Exit: (8) reverse_acc([b, c], [a], [c, b, a])
   Exit: (7) reverse_acc([a, b, c], [], [c, b, a])
   Exit: (6) reverse1([a, b, c], [c, b, a])

L = [c, b, a]
```


Prädikate

Prädikate (Relationen) für „gültige Sachverhalte“
Beweise für Folgerungen/Ableitungen
Prädikate werden vom Nutzer definiert
Prolog-System realisiert (spezielles) Beweisverfahren
Prolog-System besitzt eingebaute (built-in) Prädikate für
- Programmierumgebung
- Eingabe/Ausgabe
- Steuerung der Abarbeitung
- Programm-Manipulation
- Term-Operationen
- Häufig benutzte Prädikate
- Arithmetik

Eingabe/Ausgabe

Eröffnen von files:

tell/1 see/1

Schließen von files:

told/0 seen/0

see(datei1),lese_von_datei(X),see(user), ...

tell(datei3),schreibe_auf_datei(Z),tell(user),...

Erfragen des offenen files (Terminal: user):

telling/1 seeing/1

Eingabe/Ausgabe von Termen:

read/1 write/1

Eingabe/Ausgabe von Zeichen:

get/1 put/1

und weitere ...

read(-Term) („Eingabeparameter“)

write(+Term) („Ausgabeparameter“)

Eingabe/Ausgabe

```
kubik :- write('Nächste Zahl bitte: '),  read( X ), bearbeite( X ).

bearbeite( stop ):- !.
bearbeite( N )  :- K is N*N*N,
                   write('Dritte Potenz von '), write( N ),
                   write(' ist '), write( K ), nl,
                   kubik.
```


Steuerung von Programmen

`true`, `fail`, `!`, `not`, `;`, `,`, `call`, `repeat`

- Aufruf: call/1

```
call(Ziel) :-  Ziel .
```

- Endlose Schleifen: repeat/0

```
repeat.
repeat:-repeat.
```

Als System-Prädikat repeat/0 beliebig oft backtrackbar.

```
bearbeite_datei(D) :- see(D),
                      repeat, read(Term),
                      ( Term = end_of_file ; write(Term), fail ),  ! , seen.
```

end_of_file: CTRL d

Programmierumgebung

Laden eines PROLOG-Programms:

consult/1 oder [filename]

consult(user) bzw. [user] : Standard-Eingabe bis CTRL D

Beenden des PROLOG-Systems: halt/0

Debugging: trace/0, notrace usw.

Hilfesystem: help/0, help/1, apropos/1

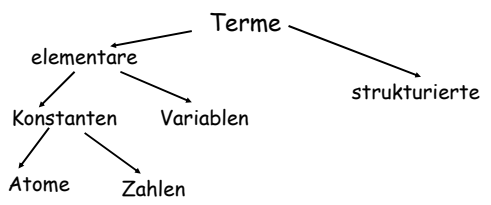
History-System: set_prolog_flag(history, N).

Auflisten von Programm-Klauseln: listing/1, listing/0

Analyse und Konstruktion von Termen

atom/1, integer/1, var/1, nonvar/1, ...

compound/1, ground/1, ...



Analyse und Konstruktion von Termen

- =.. *(„univ“)* , funktor/3, arg/3 , ...

```
?- funktor_name(arg1,arg2) =.. L .
L = [funktor_name,arg1,arg2]
```

```
..., Goal=..[Funktor|ArgListe], call(Goal), ...
```

```
functor(parent(X,zeus),parent,2)
```

```
arg(2, parent(X,zeus),zeus)
```

- Termvergleich = , == , \= , \==

Programmoperationen

Hinzufügen von Programm-Klauseln
`assert/1, asserta/1, assertz/1`

Löschen von Programm-Klauseln
`retract/1, retractall/1 , abolish/1 , abolish/2`

Suche nach Programm-Klauseln
`clause/2`

Beschränkung auf „dynamische" Klauseln
`dynamic/1`
z.B.
`dynamic(male/1)`

- Selbstmodifikation von Prolog-Programmen
- Selbstanalyse von Prolog-Programmen

Verwendung von assert

Doppelberechnungen vermeiden.
z.B. Wiederverwenden von Zwischenergebnissen:

```
:-dynamic(fibo/2).
fibo( 0, 0 ).
fibo( 1, 1 ).
fibo( N, M ):-  N1 is N-1, N2 is N-2,
                fibo( N1, M1 ),
                fibo( N2, M2 ),
                M is M1 + M2,
                asserta( fibo( N, M )).
```

Kritik am Programm: Kein Test auf negative Zahlen!

Verwendung von assert

Vorverarbeitung, z.B. Multiplikationstabelle:

```
berechne_tabelle   :-
L=[0,1,2,3,4,5,6,7,8,9],
      member(X,L),member(Y,L),
      Z is X * Y,
      assert(produkt(X,Y,Z)),
      fail;
      true.
```


Willkürliches Abbrechen von Beweisketten

Ersatz von

```
problemloesen
  :- problemloesen_1(ARGUMENTE,Z),
     problemloesen_2(Z).
```

durch

```
problemloesen
  :- problemloesen_1(ARGUMENTE) ,
     assert(zwischenergebnis(Z)), fail.
problemloesen
  :- retract(zwischenergebnis(Z))
     problemloesen_2(Z).
```

Idee:
Der durch problemloesen_1 belegte Laufzeitspeicher wird am Ende der 1.Klausel freigegeben.

Bei Bedarf Cut in problemloesen_1 um Backtracking zu vermeiden.