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 $\neg$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:

- $\neg$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 $\neg$ a folgt nicht aus $\{\neg b \rightarrow a,\ b\}$

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

```
?-a.
yes
```

Aber a folgt nicht aus $\{\neg b \rightarrow a\}$

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.        ?-r2.
no           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*:

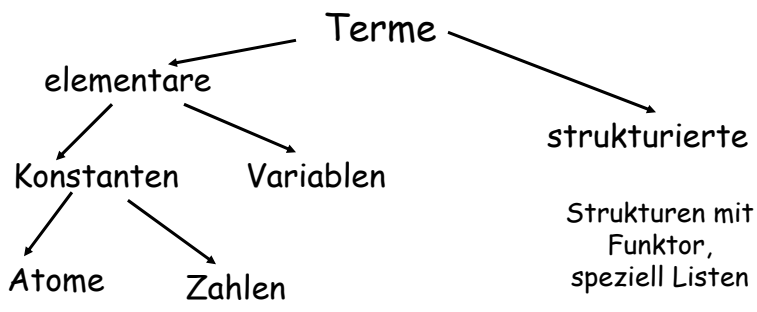
Wenn alle Beweisversuche für Q fehlgeschlagen sind.

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)`

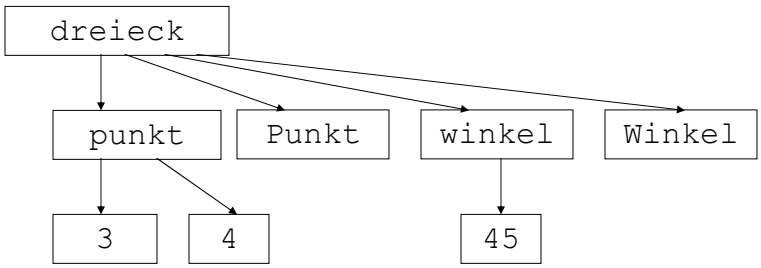
  – Argumente sind Terme (rekursiv!)

```
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)
```

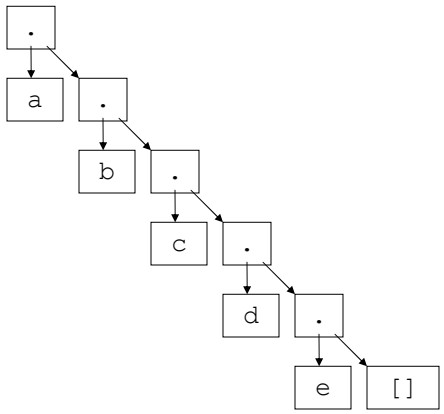


Prolog Syntax: Terme

Darstellung von Strukturen als Bäume.

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

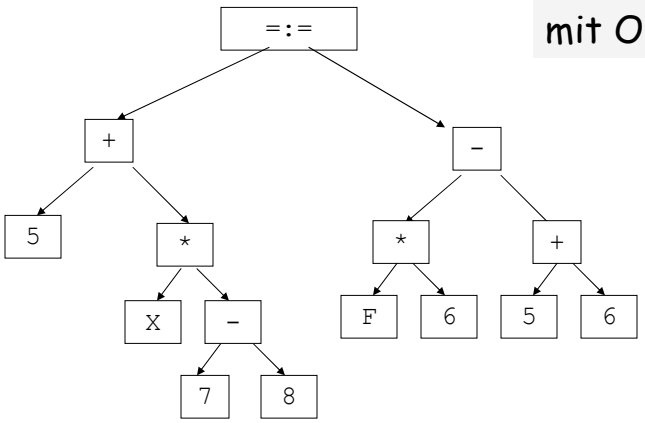
Spezielle Notationen für Listen



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).
```

Anwendungen für append-Prädikat

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

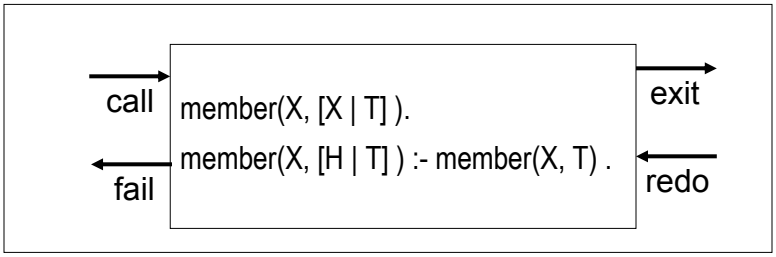
```
reverse( L,R ) :- reverse_acc( L, [ ], R ).

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

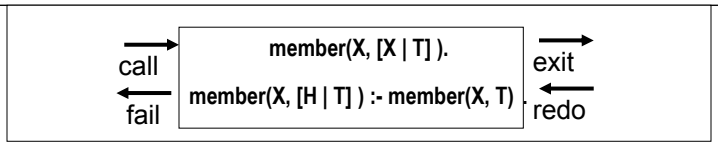

Box-Modell, trace/0

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:

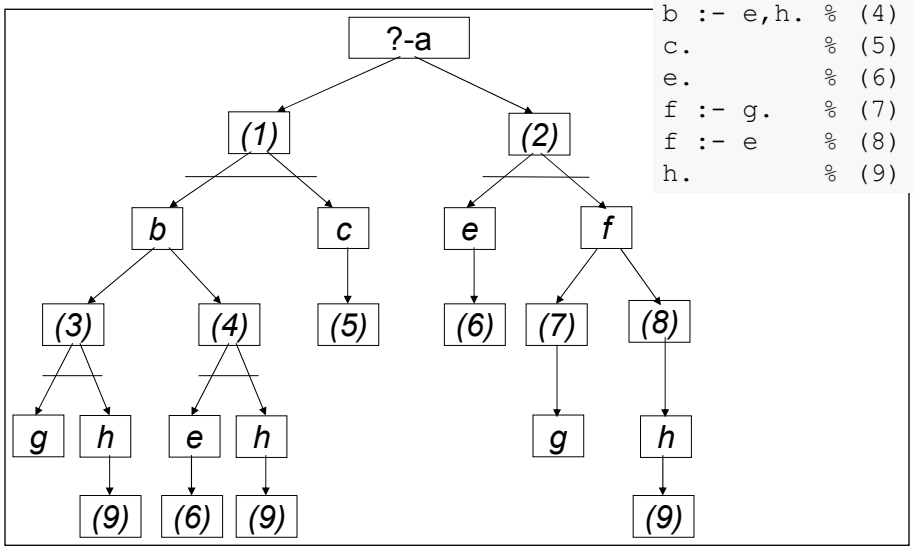
- Subgoals einer Klausel sequentiell
- Klauselaufrufe: ineinander verschachtelt

Reihenfolge der Boxen im trace:

- sequentiell gemäß Abarbeitung im Und-oder-Baum
- mit Nummerierung gemäß Ebenen im Und-oder-Baum

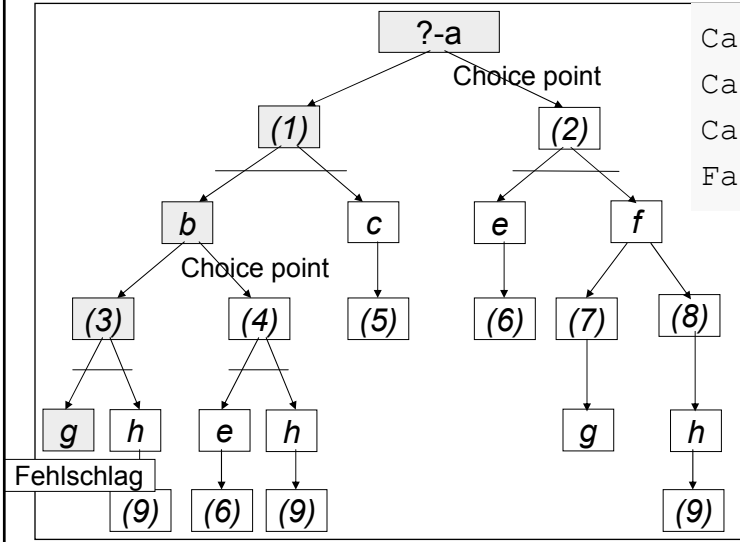
trace

```
a :- b,c.  % (1)
a :- e,f.  % (2)
b :- g,h.  % (3)
b :- e,h.  % (4)
c.         % (5)
e.         % (6)
f :- g.    % (7)
f :- e     % (8)
h.         % (9)
```

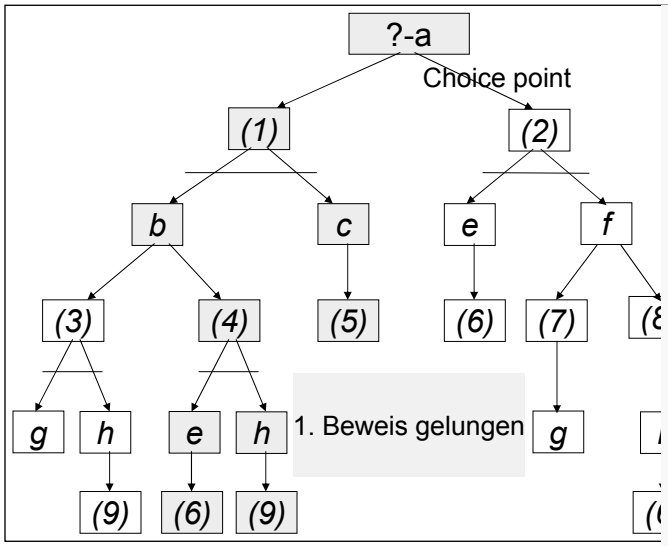


trace

```
Call: (8) a
Call: (9) b
Call:(10) g
Fail:(10) g
```



trace




```
Call: (8) a
Call: (9) b
Call:(10) g
Fail:(10) g
Redo: (9) b
Call:(10) e
Exit:(10) e
Call:(10) h
Exit:(10) h
Exit: (9) b
Call: (9) c
Exit: (9) c
Exit: (8) a
```