Grundlagen der Informatik

1. Semester, 1999

Kapitel 3: Logische Programmierung

3.1 Modell der logischen Programmierung

3.2 Wissensbasis

- 3.2.1 Terme
- 3.2.2 Horn Klauseln

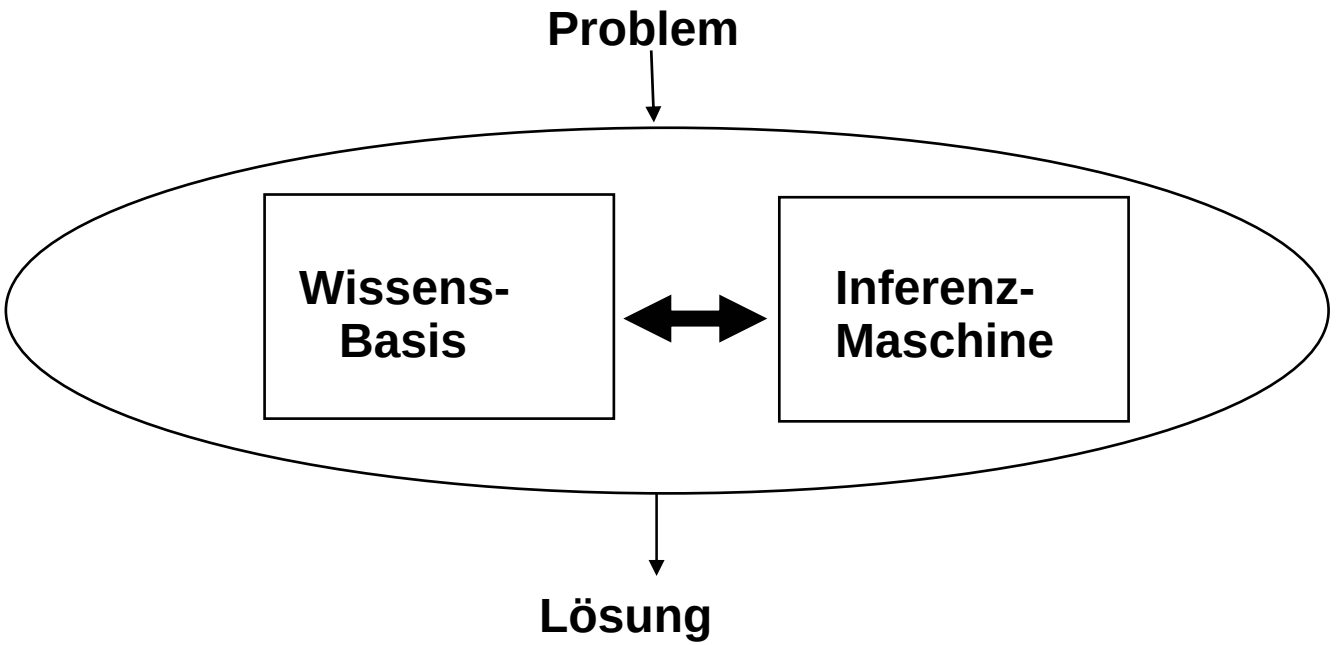
3.3. Inferenz-Maschine

- 3.3.1 Resolution
- 3.3.2 Unifikation
- 3.3.3 Nichtdeterminismen

3.4. Prolog

3.1 Modell der logischen Programmierung

Idee:



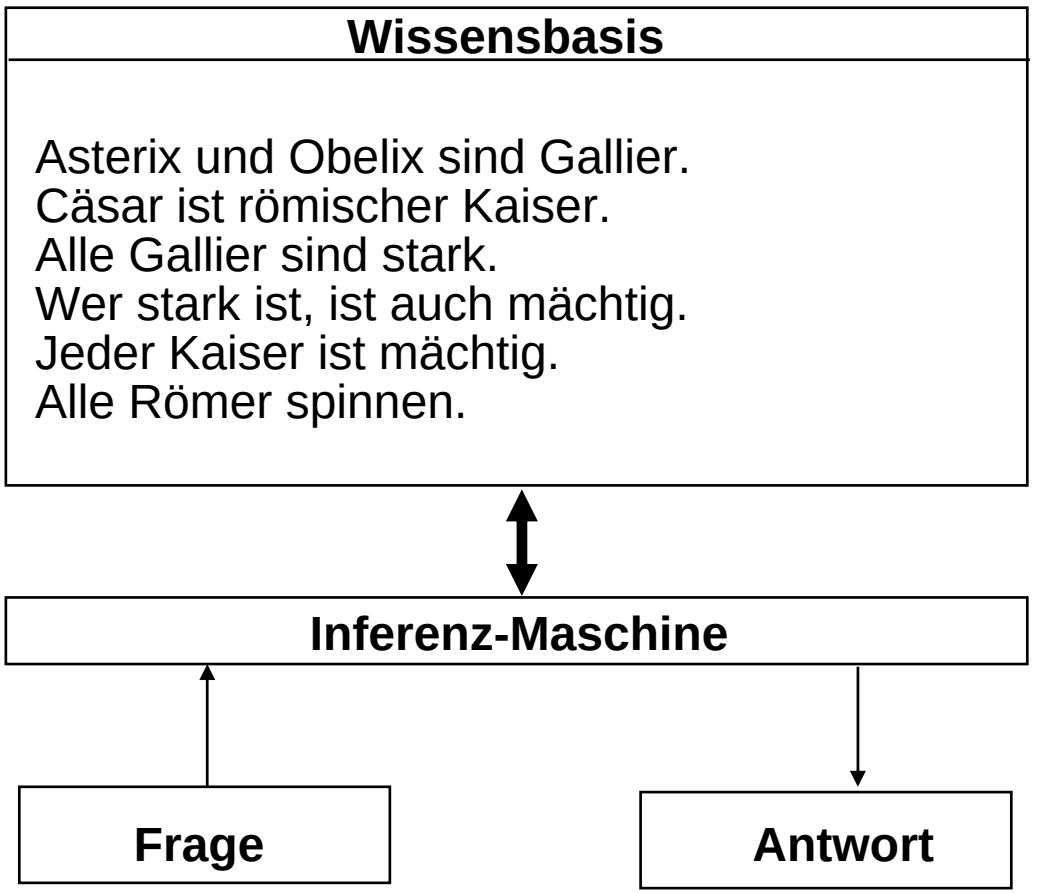
Programmierer: (muß)

- Wissen formalisieren
- die Wissensbasis mit Formeln füllen
- die Inferenz-Maschine anstoßen (durch eine Anfrage)

Inferenz-Maschine:

- ist vorgegeben
- leitet logische Folgerungen aus der Wissensbasis her

Programmier-Modell: Beispiel



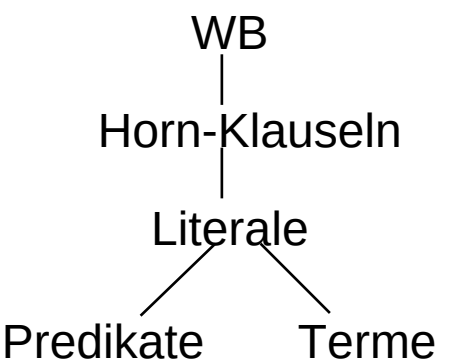
Welcher Mächtige spinnt? Cäsar

Logische Programmierung

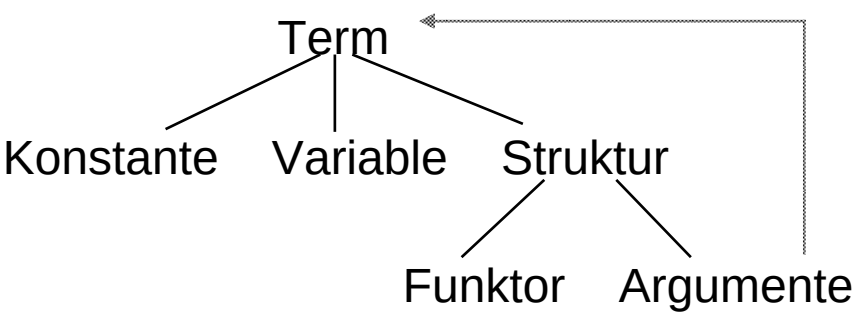
Wissensbasis: logische Formel = Konjunktion von Horn-Klauseln

Inferenz-Maschine: automatischer Beweiser : LF_{WB} => LF_{Frage}

Elemente der Wissensbasis:



Klassifikation von Termen:



3.2 Wissensbasis: Terme

Konstante: Jede Konstante repräsentiert ein konkretes Objekt

Beispiele: asterix
steffi_graf
1999
'Herr Kohl'

Konvention: Konstante beginnen mit einem Kleinbuchstaben

Variablen: Eine Variable repräsentiert eine Menge von Objekten

Beispiele: X
Liste_1
Resultat

Konvention: Varieblennamen beginnen mit einem Großbuchstaben

Strukturen: Eine Struktur repräsentiert ein Objekt, das aus mehreren Komponenten bestehen kann.

Form: f (..... , ... , ...)

Funktor Komponenten

Beispiele: kaiser (römisch)
datum (11 , januar , 2000)
* (+(5,2) , -(4,2))

Wissensbasis: Terme

Listen: Listen sind spezielle Strukturen . (Head , Tail)

Functor Liste

leere Liste: [ ]

Beispiel: . (maria , . (backt , . (pizza , [ ])))

vereinfachte Schreibweise [Element , ... , Element]

z.B. [maria , backt , pizza]

Zugriff auf Anfangselement und Restliste: [Elem1 , Elem2 , ... | Tail_List]

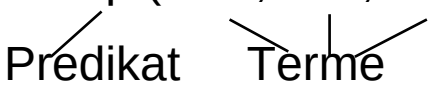
z.B. [Wer | [backt , pizza]]

[maria , backt | Was]

Wissensbasis: Literale, Horn-Klauseln

Literale:

- atomare Formel p (..... , ... , ...)



z.B. gallier (asterix)

mächtig (cäsar)

member (7 , [7 | Rest])

- negierte atomare Formel not p (.....)

Horn-Klauseln: Disjunktion von Literalen

Höchstens ein Literal darf positiv sein

p *or not* q1 *or not* q2 ..... *or not* qn

=> p *or not* (q1 *and* q2 ..... *and* qn)

=> p <- q1 , q2 , ... , qn

Wissensbasis: Horn-Klauseln

Klassifikation von Horn-Klauseln:

- Regeln: 1 positives Literal, >= 1 negierte Literale

p <- q1 , q2 , ... , qn.

Beispiele: stark(Wer) <- gallier(Wer).

sortiert([A,B|Rest]) <- <=(A,B) , sortiert([B|Rest]).

mag(walter,X) <- gut(X) , teuer(X).

- Fakten: 1 positives Literal, kein negiertes Literal

p <- true

Schreibweise: p.

Beispiele: gallier(asterix).

ist_liste([]).

ist_liste([X|Rest]).

Wissensbasis: Horn-Klauseln

- Fragen: kein positives Literal, >= 1 negierte Literale

false <- q1 , q2 , ... , qn.

Notation: <- q1, q2, ... qn.

Beispiele: <- römer(X), spinnt(X).

<- ist_liste([Head|[3,4]]).

- Leere Klausel: kein positives Literal, kein negiertes Literal

false <- true

Hinweis Die leere Klausel ist wichtig für Widerspruchsbeweise.

Beispiel: Element einer Liste: X ist in einer Liste L enthalten, falls

a) X der Head von L ist, oder

b) X im Tail von L enthalten ist

```
member(X,[X|Tail]).
member(X,[Head|Tail]) <- member(X,Tail).
```

Beantwortung von Fragen

Ausgangspunkt:

- Wissensbasis WB als logische Formel = Konjunktion von Horn-Klauseln

  A <- B, C, D, ...

  A.

- Frage, d.h. eine logische Formel F: <- G, H, I, ...

Ziel: Beantworte die Frage, d.h. leite F aus WB her

Strategien: - Vorwärtsschließen

WB => FORM1 => FORM2 => ..... => F

Nachteil: nicht zielgerichtet!

- Rückwärtsschließen

F <= FORM1 <= FORM2 <= ..... <= WB

Vorteil: zielgerichtet!

3.3 Inferenz-Maschine: Resolution

Wie sollen einzelne Beweisschritte aussehen?

- Frage habe die Form <- A.

  Suche in der Wissensbasis nach Klauseln der Form

  A. oder A <- B, C, D, ...

- Frage habe die Form <- A, ... , **B**, ... , C, ...

  Suche in der Wissensbasis nach Klauseln der Form

  B. oder B <- D, E, F, ...

In allen Fällen werden folgende Prinzipien zur Kombination von 2 Horn-Klauseln verwendet:

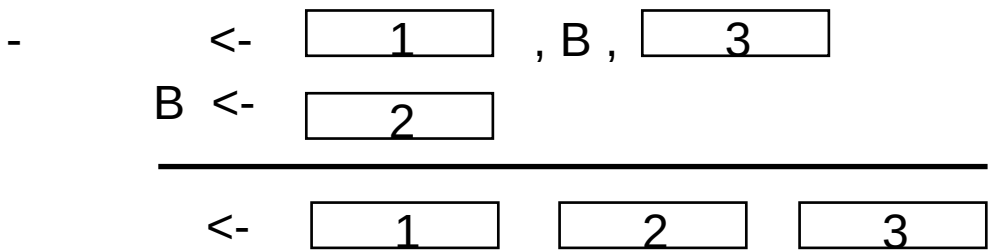
Resolution: A <- 1 , B , 3

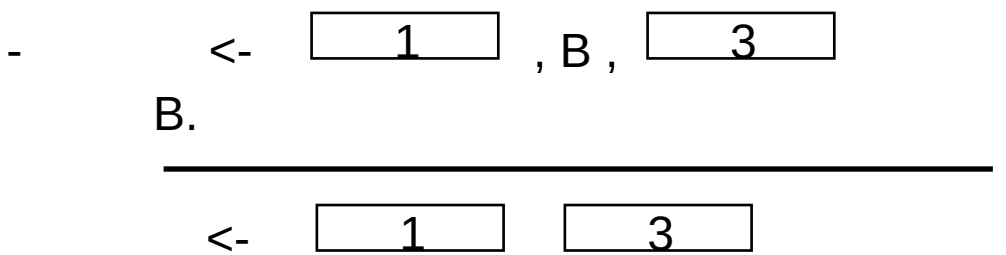
B <- 2

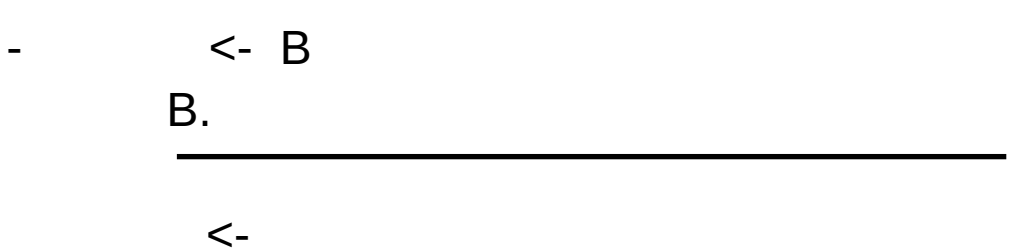
A <- 1 2 3

Inferenz-Maschine: Resolution

Spezialfälle:







Inferenz-Maschine: Unifikation

Problem: In vielen Fällen sind Fragen <- A. und Horn-Klauseln A. oder A <- ..... nicht identisch. Um sie bei der Resolution benutzen zu können, müssen sie angeglichen werden.

Example: Die Frage <- flug(stuttgart, Wohin).
paßt auf flug(stuttgart,berlin).
oder auf flug(Start,Ziel) <- ..... .
falls die Variablen wie folgt gebunden werden: Wohin = berlin
Start = stuttgart
Ziel = Wohin

Unifikation von Literalen: L1

Prä1 |	Term11	Term12	.....	Term1n

L2

Prä2 |	Term21	Term22	.....	Term2m

Dabei müssen Prä1 und Prä2 identisch sein, die Anzahl der Terme muß gleich sein (n=m) und die Terme müssen paarweise unifizierbar sein

Unifikation von Termen

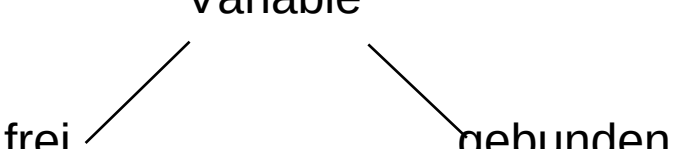
Es gibt 3 Arten von Termen: Konstante
Variablen
Strukturen

Fall 1: Beide Terme sind Konstante

Unifikation ist nur möglich, falls beide identisch sind

d.h. stuttgart ist unifizierbar mit stuttgart
stuttgart ist nicht unifizierbar mit berlin

Fall 2: Ein Term ist eine Variable



- Freie Variablen passen auf jeden Term (*)

d.h. Start ist unifizierbar mit stuttgart
Spieler ist unifizierbar mit [boris, becker]
Sieben ist unifizierbar mit 7
Ziel ist unifizierbar mit Wohin

(*) Unifikation nicht möglich bei Strukturen, welche die Variable wieder enthalten

Unifikation von Termen

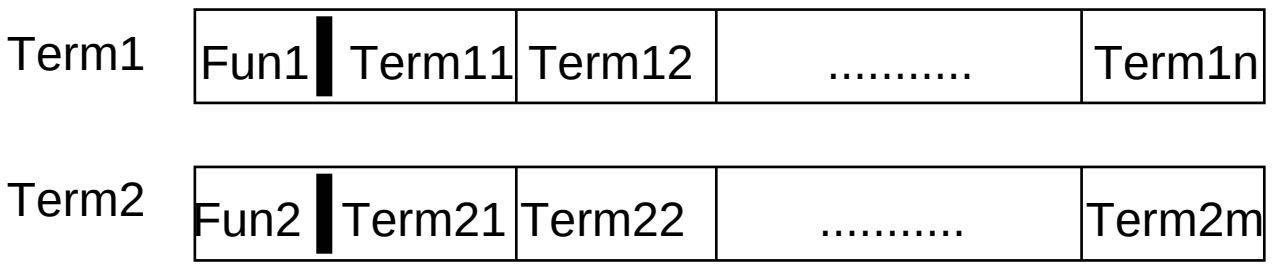
- Gebundene Variablen werden behandelt wie die Terme, an die sie gebunden sind

  d.h. wie - Konstante

  - Variablen

  - Strukturen

Case 3: Beide Terme sind Strukturen



Unifikation ist möglich, falls - Fun1 und Fun2 identisch sind

- die Anzahl der Terme gleich ist (n=m)
- die Terme paarweise unifizierbar sind

Beispiel: [Eins, Zwei | Rest] ist unifizierbar mit [1, 2 | Zahlen]

Bindungen: Eins <- 1

Zwei <- 2

Rest <-> Zahlen

Unification of Terms

[Eins , Zwei , Drei] ist nicht unifizierbar mit list(1,2,3)
weil die Funktoren . und list nicht identisch sind!

Fall 4: Ein Term ist Konstante, der andere ist Struktur

keine Unifikation möglich!

Eigenschaft der Unifikation

- Unifikation ist eine symmetrische Operation, d.h. gleiche Termpositionen können Werte importieren oder exportieren
- Der Gültigkeitsbereich einer Variablen ist die Hornklausel, in der sie vorkommt. Bei mehrfachem Auftreten innerhalb einer Klausel muß eine Variable überall die gleiche Bindung erhalten (unabhängig von der Importposition!)

Folge: Die Strukturen twins(X,X) und twins(max,moritz) sind nicht unifizierbar wegen der inkonsistenten Bindungen

X <- max

X <- moritz

ODER Nichtdeterminismus

Was ist in der folgenden Situation zu tun?

<- A. (Frage)

Wissensbasis: .....

A <- .....

A.

A <- .....

.....

Antwort:

- sequentielle Suche in der Wissensbasis (Suchregel bestimmt die anzuwendende Klausel)
- parallele Suche nach alternativen Lösungen (ODER Parallelität)

UND Nichtdeterminismus

In welcher Reihenfolge sollen Teilanfragen beantwortet werden?

<- 1 , B , 2 , C , 3

B <- 4

<- 1 4 2 , C , 3

oder

<- 1 , B , 2 , C , 3

C <- 5

<- 1 , B , 2 5 3

Antwort:

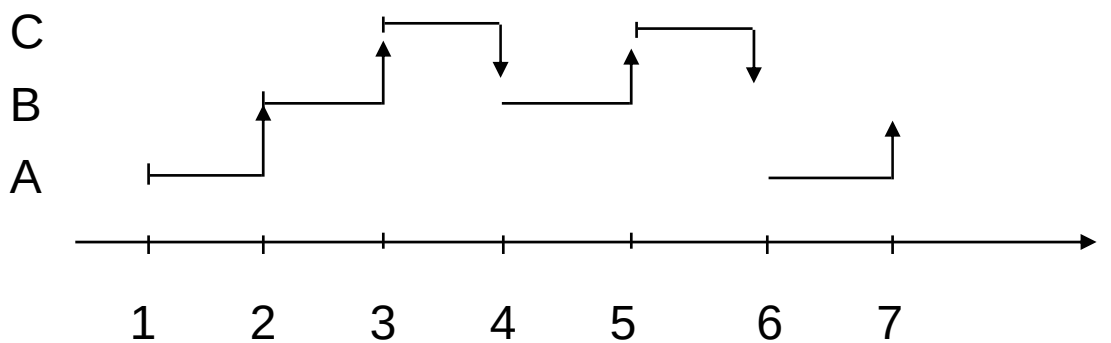
- sequentielles Herleiten von Teilproblemen von links nach rechts (Berechnungsregel wählt das zu lösende Teilproblem aus)
- parallele Herleitung von Teilproblemen (UND Parallelität)

Backtracking

Falls die Inferenzmaschine beim Lösen eines Teilproblems scheitert

- Rücknahme des zuletzt abgeschlossenen Beweisschrittes
- Test, ob noch nicht untersuchte Klauseln ein erneutes Vorrücken ermöglichen
- falls nötig, Rücknahme weiterer Schritte

Beispiel: Frage <- A, B, C



1: Ziel A erreicht, Marke für A setzen

2: Ziel B erreicht, Marke für B setzen

3: Ziel C verfehlt, Rückkehr zur Marke für B

4: Ziel B erneut erreicht, Marke für B setzen

5: Ziel C erneut verfehlt, keine weitere alternative Klausel für B

6: Ziel A erneut erreicht, Marke für A setzen

Backtracking, Prolog-Wissensbasis

Frage: Welcher Teilproblem soll beim Erreichen einer Sackgasse neu gelöst werden?

Answer: - sequentielles Rücksetzen bereits gelöster Teilprobleme von rechts nach links

- Neuberechnung solcher Teilprobleme, die eine im gescheiterten Teilproblem beteiligte Variablenbindung erzeugt haben(intelligentes Backtracking)

Kontrollfluß in Prolog-Wissensbasen

Suchregel: von oben nach unten

Berechnungsregel: von links nach rechts

Rücksetzregel: von rechts nach links

Prolog-Wissensbasen

Probleme:

* Die Reihenfolge, in der Fakten und Regeln angeordnet werden, beeinflußt die Laufzeit von Programmen. Es gibt i.a. keine optimale Anordnung.

Beispiel: (1) member(E,[E|R]).

(2) member(E,[A|R]) :- member(E,R).

- Anordnung (1), (2) ist optimal für die Frage

  ?- member(17,[17,34,51,68,85]).

- Anordnung (2), (1) ist optimal für die Frage

  ?- member(85,[17,34,51,68,85]).

* Die Reihenfolge der Teilprobleme innerhalb einer Regel beeinflußt die Laufzeit.

  P :- Q, R, S, ... sollte so gewählt sein, daß die Prädikate von links nach rechts immer schwächer werden.

* Die Reihenfolge der Teilprobleme innerhalb einer Regel beeinflußt die Anzahl unnötiger Rücksetzschritte. Variablenbindungen sollten möglichst direkt nach ihrer Erzeugung weiterverwendet werden.

Allgemein: Es gibt keine optimale Anordnung!

Prolog

Entwicklung: - 1972 erste Implementierung in Marseille Colmerauer,.....
- 1980 Programmiersprache für das Fifth Generation Project

Eigenschaften: Suchregel - von oben nach unten
Berechnungsregel - von links nach rechts
Rücksetzregel - von rechts nach links

***** Jede Hornklausel in der Wissensbasis endet mit einem '.'

A. (Fakt)

A :- B, C, D. (Regel)

***** Darstellung von Fragen

?- A.

***** Laden der Wissensbasis

consult(...Datei...). oder [...Datei...].

Prolog

***** Für Arithmetik in Termen verwendet man Infix-Notation

X + Y	anstatt	+(X,Y)
X - Y	anstatt	-(X,Y)
X * (Y - Z)	anstatt	*(X,-(Y,Z))

***** Arithmetische Operationen sind möglich

- innerhalb des **is**-Prädikats d.h. Summe **is** X + Y
  Rest **is** X mod Y
- in Relationen (als Prädikate in Infix-Notation) X + Y > A

(Bevor eine arithmetische Operation ausgeführt wird, müssen die Operanden an Werte gebunden sein)

***** Ein-/Ausgabe-Prädikate

- read(X)
- write(X)
- readln, writeln

Prolog

* Prolog erlaubt die dynamische Änderung der Wissensbasis während der Beantwortung einer Frage

retract(...Klausel...) löscht Klausel aus der Wissensbasis
asserta(...Klausel...) fügt Klausel am Anfang der Wissensbasis ein
assertz(...Klausel...) fügt Klausel am Ende der Wissensbasis ein

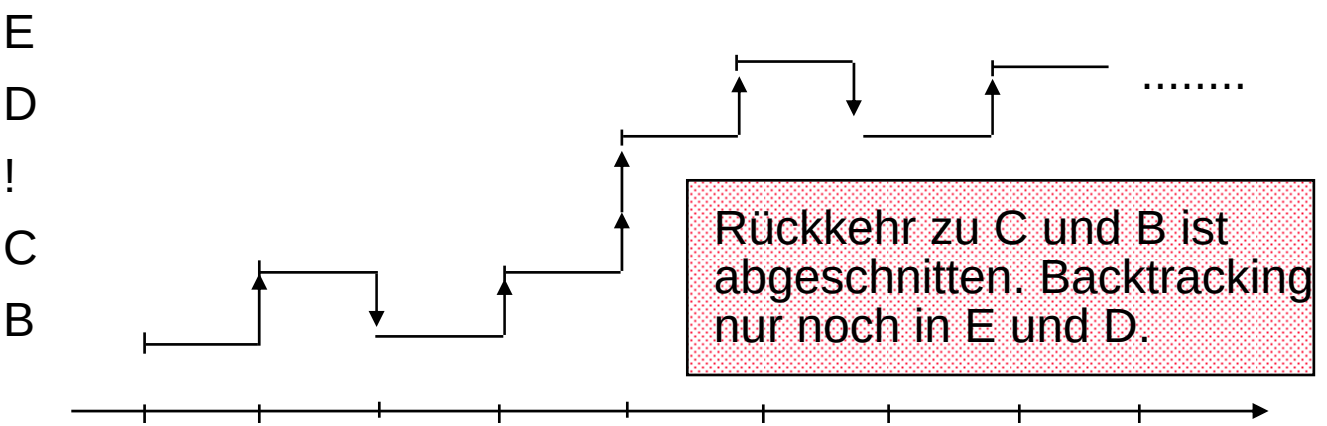
Darstellung von Klauseln innerhalb von retract und assert:
- Fakt: p(t1,.....) Prädikat p wird als Funktor behandelt
- Regel: ':-'(A,','(B,C)) repräsentiert die Regel A :- B, C.

Anwendung: solve(problem,Loesung),
asserta(solve(problem,Loesung)).
fügt die Lösung des gegebenen Problems am Anfang der Wissensbasis ein

Prolog

*** cut:** Beschneidung des Suchraums

Form: A :- B, C, !, D, E.



Falls das Backtracking über den ! zurücksetzt, werden alle Alternativen für A abgeschnitten.

- *Anwendung*: Fallunterscheidung

  case :- ..... test1..... , !, ... action1..... .

  case :- .....test2..... , !, ...action2..... .

  .....

Falls testi wahr ist, erfolgt bei eventuell späterem Backtracking kein weiterer Test.

Prolog

*** not** (Negation as failure)

implementiert durch:

```
not(p) :-  call(p), !, fail.
not(p).
```

- *Problem:* Falls es für p keine Lösung gibt, so kann dies nicht immer explizit festgestellt werden (bei Endlosschleifen!)

  => not(p) ist nicht immer entscheidbar!

*** call(x)**

Ein Term x kann mittels call(x) als Teilproblem ausgeführt werden.