Programmiersprachen

Alexandru Berlea

Institut für Informatik
TU München

Wintersemester 2006/2007

Negation in Prolog vs. NaF

```
p(X) :- p(X).
q(a).
```

- $not(q(b), p(a))$ ist nicht definiert in der LP-Semantik.
- `not((q(b),p(a)))` ist **erfolgreich** in Prolog.

Negation und Ziele mit Variablen

Bsp.:

```
braucht_schein(X) :- not(diplom(X)), student(X).
student(peter).
student(martina).
diplom(martina).
```

Die Anfrage `braucht_schein(X)` scheitert, obwohl die Antwort `{X=peter}` unter einer Interpretierung von `not` als logische Negation erwartet wird.

$\Longrightarrow$ Man muss sicherstellen, dass die Variablen in einem negierten Ziel bei seiner Auswertung belegt sind.

Negation und Ziele mit Variablen

Bsp.:

```
braucht_schein(X) :- student(X), not(diplom(X)).
student(peter).
student(martina).
diplom(martina).
```

$\Longrightarrow$ Antwort {X=peter}.

Red Cuts

Bsp.:

```
min(X,Y,X) :- X =< Y.
min(X,Y,Y) :-  X > Y.
```

Cuts deren Auftritt in einem Programm die Semantik des Programmes ändern heißen red cuts.

```
min(X,Y,X) :- X =< Y, !.
min(X,Y,Y).
```

Die Anfrage `min(1,2,2)` liefert (unerwünschterweise) *yes*.

Red Cuts

Bsp.: Eine Prozedur zur Entfernung von Elementen aus einer Liste:

```
delete([X|Xs],X,Ys) :- delete(Xs,X,Ys).
delete([X|Xs],Z,[X|Ys]) :- Z \== X, delete(Xs,Z,Ys).
delete([],X,[]).
```

Red Cuts

- Delete mit green Cuts:

```
delete([X|Xs],X,Ys) :- !, delete(Xs,X,Ys).
delete([X|Xs],Z,[X|Ys]) :- Z\==X,!,delete(Xs,Z,Ys).
delete([],X,[]).
```

- Delete mit red Cuts:

```
delete([X|Xs],X,Ys) :- !, delete(Xs,X,Ys).
delete([X|Xs],Z,[X|Ys]) :- !, delete(Xs,Z,Ys).
delete([],X,[]).
```

Das Programm mit red Cuts funktioniert richtig, ist dafür bei minimal besserer Effizienz viel unleserlicher geworden.

Selbst-modifizierende Programme

- Prolog erlaubt laufende Programme bei der Laufzeit zu analysieren und zu ändern.
- Das Ziel `clause(Kopf,Rumpf)` bietet Zugang zu den Klauseln des laufenden Programms.
- `Kopf` darf keine unbelegte Variable sein.
- Die erste Klausel, deren Kopf mit `Kopf` unifiziert wird gefunden und `Rumpf` wird mit dessen Rumpf unifiziert.
- Alle Klauseln deren Kopf mit `Kopf` unifizieren werden via Backtracking gefunden.
- Fakten haben `true` als ihren Rumpf.

Selbst-modifizierende Programme

Bsp.:

```
member(X,[X|Xs]).
member(X,[Y|Ys]) :- member(X,Ys).
```

Anfrage `clause(member(X,Ys),Rumpf)` liefert die Antworten:

- `{Ys=[X|Xs], Rumpf=true};`
- `{Ys=[Y|Ys1], Rumpf=member(X,Ys1)}.`

Selbst-modifizierende Programme

Systemprädikate zum Hinzufügen und Entfernen von Klauseln zum (laufenden) Programm:

- `assertz(Klausel)`: fügt `Klausel` als letzte Klausel der entsprechenden Prozedur.

  Bsp.: `assertz((mammal(X) :- whale(X))).`

- `asserta(Klausel)`: fügt `Klausel` als die erste Klausel der entsprechenden Prozedur.

- `retract(K)`: entfernt die erste Klausel, die mit `K` unifiziert.

  Bsp.: Eine Klausel `a :- b,c` kann mit `retract((a :- X))` entfernt werden.

Selbst-modifizierende Programme

Das dynamische Hinzufugen und Entfernen von Klauseln macht einen Unterschied zwischen **dynamischen** und **statischen** benutzerdefinierten **Prädikaten**.

- Statische Prädikate können kompiliert werden $\Longrightarrow$ können effizienter ausgewertet werden.
- Dynamische Prädikate müssen als solche deklariert werden.
- Dynamische Prädikate machen den Code von Seiteneffekten abhängig $\Longrightarrow$ weniger deklarativ und leserlich.
- Allerdings können dynamische Prädikate u.U. die Effizienz unterstützen, z.B. für *dynamische Programmierung*.

Beispiel: Dynamische Programmierung

- **Dynamische Programmierung**
  - Idee: speichere partielle Ergebnisse während einer Berechnung, die später wieder benutzt werden können.
  - Bsp.: Berechnung der Binomialkoeffizienten (☞ Übung)
- **Mögliche Umsetzung in Prolog**

  Versuche ein Ziel abzuleiten, und wenn dies möglich ist, speichere dieses Ergebnis als Fakt und vermeide, dass später alternative Ableitungen betrachtet werden.

```
lemma(Z) :- Z, asserta((Z :- !)).
```

Beispiel: Dynamische Programmierung

Anwendung: Die Türme aus Hanoi

```
:- dynamic hanoi/5.

hanoi(1,A,B,C,[move(A,B)]).
hanoi(N,A,B,C,Moves) :-
        N > 1, N1 is N -1,
        lemma(hanoi(N1,A,C,B,Ms1)),
        hanoi(N1,C,B,A,Ms2),
        append(Ms1,[move(A,B)|Ms2],Moves).

lemma(P):- P, asserta((P :- !)).

test_hanoi(N,Pegs,Moves) :-
        hanoi(N,A,B,C,Moves), Pegs = [A,B,C].
```

Beispiel: Dynamische Programmierung

Anwendung: Die Türme aus Hanoi

```
:- dynamic hanoi/5.

hanoi(1,A,B,C,[move(A,B)]).
hanoi(N,A,B,C,Moves) :-
        N > 1, N1 is N -1,
        lemma(hanoi(N1,A,C,B,Ms1)),
        hanoi(N1,C,B,A,Ms2),
        append(Ms1,[move(A,B)|Ms2],Moves).

lemma(P):- P, asserta((P :- !)).

test_hanoi(N,Pegs,Moves) :-
        hanoi(N,A,B,C,Moves), Pegs = [A,B,C].


?- test_hanoi(3,[a,b,c],X).
X = [move(a, b),move(a, c),move(b, c),move(a, b),move(c, a),move(c, b),move(a, b)]
```

Mehr Prolog

Prolog bietet mehr an, z.B.:

- Prädikate zum **Testen und Manipulieren der Struktur der Terme**;
- Mehr meta-logische Prädikate z.B. zum **Testen des Zustands der Ableitung**;
- Mehr extra-logische Prädikate, die Seiteneffekte bei ihrer "Ableitung" haben, z.B für **Ein- und Ausgabe** oder für die **Schnittstelle zum Betriebssystem**.
- Es gibt Prolog-Erweiterungen, z.B. für Constraint-Programmierung...

Part II

Constraint-Programmierung

Constraint-Programmierung (CP)

Einschränkungen der logischen Programmierung:

- Alle in einem puren logischen Programm manipulierten Objekte (die **Terme**) **sind rein syntaktische Konstruktionen**, denen keine Semantik zugewiesen wird.

- D.h. die **Funktor-Symbolen sind nicht-interpretiert**.

- **Bsp.:**
  - Das Ziel `X=2+3` bewirkt nur die Bindung von `X` an den Term `2+3`, weil das Funktionssymbol `+` nicht interpretiert wird.

  - Das Ziel `1+4=2+3` scheitert.

Constraints

- Zwei Objekte sind in LP nur dann gleich, wenn sie syntaktisch gleich sind.
- Idee: erweitere die rein syntaktische Gleichheit in LP zur Gleichung, die zu lösen ist:
  - Das Ziel `X+Y=8,X-Y=2` erhält in einer CP-Sprache die Antwort `X=5,Y=3`.
- Allgemeiner: spezifiziere (implizite) Relationen zwischen semantischen Objekten. I.A. heißen Relationen zwischen semantischen Objekten **Constraints**.

CP als Erweiterung der logischen Programmierung

- Die Constraints werden als syntaktisch ausgezeichnete Prädikate dargestellt, und statt mittels Resolution durch spezielle Algorithmen über bestimmte Wertebereiche mit Hilfe eines **Constraint-Löser** gelöst.

- LP ist CP, wobei das einzige Constraint die syntaktische Gleichheit zwischen Termen ist, und der Unifikationsalgorithm zur Lösung solcher Constraints benutzt wird.

Constraint-Löser

- Programmierung mit Constraints ist in beliebigen Programmiersprachen möglich, vorausgesetzt dass ein Constraint-Löser, evt. als eine Erweiterungs-Bibliothek zur Verfügung gestellt wird, z.B. für Java, C++, Prolog, Lisp.

- Meist werden LP-Sprachen um Constraints erweitert (☞ Constraint-Logikprogrammierung), z.B. Eclipse-, Sicstus-, SWI-Prolog, Mozart/Oz, etc.

- Effektive Constraint-Löser gibt für verschiedene Constraint-Arten, z.B.:
  - Logische Formeln über boolsche Variablen
  - Intervall-Constraints über endliche Bereiche
  - Lineare Gleichungssysteme über reale Zahlen

Constraint-Löser

Von einem Constraint-Löser zu erfüllenden Berechnungsdienste:

- **Konsistenztest**(Erfüllbarktest) (*consistency/satisfiability test*): sind die Constraints erfüllbar? Ein Constraint-Löser ist **vollständig**, wenn er die Erfüllbarkeit jeder beliebigen Menge von Constraints entscheiden kann.

- **Vereinfachung**: Die Constraints in eine einfachere Normalform darstellen können. Zur effizienten Vereinfachung soll der Löser **inkrementell** sein: Vereinfachung der Constraints zusammen mit einem neu hinzukommendes Constraint ohne die Vereinfachung der bisherigen Constraints.

- **Determination**: Erkennen, wenn eine Variable nur noch einen bestimmten Wert haben kann. (z.B. $X \geq 1$, $X \leq 1 \Rightarrow X = 1$.)

Vorteile der Constraint-Logikprogrammierung

- Zusätzlich zur erhöhten Deklarativität kann CP die Effizienz der LP-Sprachen erhöhen.

- Constraints können benutzt werden, um den *Nicht-Determinismus* der LP-Suche nach Lösungen einzuschränken, indem man Teilbäume von der Suche ausschliesst, die keine Lösung der Constraints enthalten können (durch Konsistenzteste).
  $\Longrightarrow$ CP wird eingesetzt, um kombinatorische Probleme zu lösen, die meist exponentielle Komplexität haben.

Syntax einer CP-Sprache

Atome:	A, B	$::=$	$p(t_1, \ldots, t_n)$
Constraints:	C, D	$::=$	$c(t_1, \ldots, t_n) \mid C \wedge D$
Ziele:	G, H	$::=$	$\top \mid \bot \mid A \mid C \mid G \wedge H$
Klauseln:	K	$::=$	$A \leftarrow G$
Programme:	P	$::=$	$\{K_1, \ldots, K_m\}$

Berechnungszustände

- Ein **Zustand** ist ein Paar $< G, C >$, wobei G ein Ziel und C ein Constraint.
- G heißt **Zielspeicher** (die noch zu lösende Ziele), C heißt **Constraintspeicher** (die bereits aufgetretene Constraints).
- Ein **Anfangszustand** ist ein Zustand der Form $< G, \mathit{true} >$.
- Ein Zustand heißt **erfolgreicher Endzustand**, falls er von der Form $< \top, C >$ ist.
- Ein Zustand heißt **erfolgloser Endzustand**, falls er von der Form $< G, \mathit{false} >$ ist.

CP-Kalkül

Entfalten (*unfold*)	$$\frac{(B \leftarrow H) \in P \quad (B \overset{*}{=} A) \wedge C \text{ ist erfüllbar}}{< A \wedge G, C > \mapsto_{Entfalten} < H \wedge G, (B \overset{*}{=} A) \wedge C >}$$
Scheitern (*failure*)	$$\frac{\text{Es gibt keine Klausel } (B \leftarrow H) \in P, \text{ so dass } (B \overset{*}{=} A) \wedge C \text{ erfüllbar ist}}{< A \wedge G, C > \mapsto_{Scheitern} < \bot, false >}$$
Vereinfachen (*solve*)	$$\frac{C \wedge D_1 \equiv D_2}{< C \wedge G, D_1 > \mapsto_{Vereinfachen} < G, D_2 >}$$

wobei $B \overset{*}{=} A$ gilt gdw. $A = p(t_1, \ldots, t_n), B = p(s_1, \ldots, s_n)$ und $t_1 = s_1 \wedge \ldots \wedge t_n = s_n$.

Die Antwort einer CP-Berechnung

- Die Antwort einer Berechnung, die einen erfolgreichen Endzustand $< \top, C >$ erreicht, ist C.
- Eine Antwort heißt **bestimmt** (*definite*), wenn er eine Gleichung `X=Konstante` für jede Variable in der Anfrage enthält.
  - Z.B. `X+Y=10,X-Y=6` liefert die Antwort `X=8,Y=2`.
- I.A. kann eine Antwort **unbestimmt** (*indefinite*) sein, d.h. eine unendliche Menge von Lösungen repräsentieren.
  - Z.B. liefert `X≤Y,Y≤Z,Z≤X` die Antwort `X=Y=Z`.

Lineare arithmetische Constraints

- **Arithmetische Ausdrücke:**

  t ::= Zahl | Variable | $t_1 \odot t_2$ mit $\odot \in \{+, -, *, /\}$

  Linearität:
  - Höchstens ein Term einer Multiplikation enthält eine Variable.
  - Der Teiler in einer Division enthält keine Variable.

- **Arithmetische Constraints:**

  C ::= true | false | $C \wedge C$ | $t_1 \ \mathcal{R} \ t_2$ mit $\mathcal{R} \in \{<, \leq, =, >, \geq, \neq\}$

Beispiel: Menu-Berechner

```
appetiser(radishes,1). appetiser(pasta,6).
meat(beef,5). meat(pork,7).
fish(sole,2). fish(tuna,4).
dessert(fruit,2). dessert(icecream,6).

main(M,I) :- meat(M,I).
main(M,I) :- fish(M,I).

lightmeal(A,M,D) :-
   I>0, J>0, K>0,
   I+J+K <= 10,
   appetiser(A,I), main(M,J), dessert(D,K).
```

Beispiel: erfolgreiche Berechnung

$< \texttt{lightmeal(A,M,D)};\texttt{true} >$

Beispiel: erfolgreiche Berechnung

```
    < lightmeal(A,M,D);true >
↦   < I>0,J>0,K>0,I+J+K<=10,appetiser(A,I),main(M,J),dessert(D,K);true >
```

Beispiel: erfolgreiche Berechnung

$$< \texttt{lightmeal(A,M,D);true} >$$

$$\mapsto \quad < \texttt{I>0,J>0,K>0,I+J+K<=10,appetiser(A,I),main(M,J),dessert(D,K);true} >$$

$$\mapsto^* \quad < \texttt{appetiser(A,I),main(M,J),dessert(D,K);I>0,J>0,K>0,I+J+K<=10} >$$

Beispiel: erfolgreiche Berechnung

$$
\begin{array}{ll}
 & \langle\, \texttt{lightmeal(A,M,D)};\texttt{true} \,\rangle \\
\mapsto & \langle\, \texttt{I>0,J>0,K>0,I+J+K<=10,appetiser(A,I),main(M,J),dessert(D,K)};\texttt{true} \,\rangle \\
\mapsto^{*} & \langle\, \texttt{appetiser(A,I),main(M,J),dessert(D,K)};\texttt{I>0,J>0,K>0,I+J+K<=10} \,\rangle \\
\mapsto & \langle\, \texttt{main(M,J),dessert(D,K)};\texttt{A=radishes,I=1,J>0,K>0,1+J+K<=10} \,\rangle
\end{array}
$$

Beispiel: erfolgreiche Berechnung

$< \texttt{lightmeal(A,M,D);true} >$

$\mapsto \quad < \texttt{I>0,J>0,K>0,I+J+K<=10,appetiser(A,I),main(M,J),dessert(D,K);true} >$

$\mapsto^* \quad < \texttt{appetiser(A,I),main(M,J),dessert(D,K);I>0,J>0,K>0,I+J+K<=10} >$

$\mapsto \quad < \texttt{main(M,J),dessert(D,K);A=radishes,I=1,J>0,K>0,1+J+K<=10} >$

$\mapsto \quad < \texttt{meat(M1,I1),dessert(D,K);}$
$\texttt{M=M1,J=I1,A=radishes,I=1,J>0,K>0,1+J+K<=10} >$

Beispiel: erfolgreiche Berechnung

$< \texttt{lightmeal(A,M,D);true} >$

$\mapsto \quad < \texttt{I>0,J>0,K>0,I+J+K<=10,appetiser(A,I),main(M,J),dessert(D,K);true} >$

$\mapsto^* \quad < \texttt{appetiser(A,I),main(M,J),dessert(D,K);I>0,J>0,K>0,I+J+K<=10} >$

$\mapsto \quad < \texttt{main(M,J),dessert(D,K);A=radishes,I=1,J>0,K>0,1+J+K<=10} >$

$\mapsto \quad < \texttt{meat(M1,I1),dessert(D,K);}$
$\texttt{M=M1,J=I1,A=radishes,I=1,J>0,K>0,1+J+K<=10} >$

$\mapsto \quad < \texttt{dessert(D,K);M1=beef,I1=5,M=beef,J=5,A=radishes,I=1,K>0,1+5+K<=10} >$

Beispiel: erfolgreiche Berechnung

$< \texttt{lightmeal(A,M,D)};\texttt{true} >$

$\mapsto \quad < \texttt{I>0,J>0,K>0,I+J+K<=10,appetiser(A,I),main(M,J),dessert(D,K)};\texttt{true} >$

$\mapsto^* \quad < \texttt{appetiser(A,I),main(M,J),dessert(D,K)};\texttt{I>0,J>0,K>0,I+J+K<=10} >$

$\mapsto \quad < \texttt{main(M,J),dessert(D,K)};\texttt{A=radishes,I=1,J>0,K>0,1+J+K<=10} >$

$\mapsto \quad < \texttt{meat(M1,I1),dessert(D,K)};$
$\texttt{M=M1,J=I1,A=radishes,I=1,J>0,K>0,1+J+K<=10} >$

$\mapsto \quad < \texttt{dessert(D,K)};\texttt{M1=beef,I1=5,M=beef,J=5,A=radishes,I=1,K>0,1+5+K<=10} >$

$\mapsto \quad < \top;\texttt{D=fruit,K=2,M1=beef,I1=5,M=beef,J=5,A=radishes,I=1} >$

Beispiel: erfolgreiche Berechnung

$$
\begin{array}{ll}
 & \langle \texttt{lightmeal(A,M,D)};\texttt{true} \rangle \\
\mapsto & \langle \texttt{I>0,J>0,K>0,I+J+K<=10,appetiser(A,I),main(M,J),dessert(D,K)};\texttt{true} \rangle \\
\mapsto^{*} & \langle \texttt{appetiser(A,I),main(M,J),dessert(D,K)};\texttt{I>0,J>0,K>0,I+J+K<=10} \rangle \\
\mapsto & \langle \texttt{main(M,J),dessert(D,K)};\texttt{A=radishes,I=1,J>0,K>0,1+J+K<=10} \rangle \\
\mapsto & \langle \texttt{meat(M1,I1),dessert(D,K)}; \\
 & \qquad \texttt{M=M1,J=I1,A=radishes,I=1,J>0,K>0,1+J+K<=10} \rangle \\
\mapsto & \langle \texttt{dessert(D,K)};\texttt{M1=beef,I1=5,M=beef,J=5,A=radishes,I=1,K>0,1+5+K<=10} \rangle \\
\mapsto & \langle \top;\texttt{D=fruit,K=2,M1=beef,I1=5,M=beef,J=5,A=radishes,I=1} \rangle
\end{array}
$$

Antwort: *A=radishes,M=beef,D=fruit*.

Beispiel: gescheiterte Berechnung

$< \texttt{lightmeal(A,M,D);true} >$

Beispiel: gescheiterte Berechnung

$< \texttt{lightmeal(A,M,D);true} >$

$\mapsto \quad < \texttt{I>0,J>0,K>0,I+J+K<=10,appetiser(A,I),main(M,J),dessert(D,K);true} >$

Beispiel: gescheiterte Berechnung

```
    < lightmeal(A,M,D);true >
↦   < I>0,J>0,K>0,I+J+K<=10,appetiser(A,I),main(M,J),dessert(D,K);true >
↦   < appetiser(A,I),main(M,J),dessert(D,K);I>0,J>0,K>0,I+J+K<=10 >
```

Beispiel: gescheiterte Berechnung

```
    < lightmeal(A,M,D);true >
↦   < I>0,J>0,K>0,I+J+K<=10,appetiser(A,I),main(M,J),dessert(D,K);true >
↦   < appetiser(A,I),main(M,J),dessert(D,K);I>0,J>0,K>0,I+J+K<=10 >
↦   < main(M,J),dessert(D,K);A=pasta,I=6,J>0,K>0,6+J+K<=10 >,
```

Beispiel: gescheiterte Berechnung

```
    < lightmeal(A,M,D);true >
↦   < I>0,J>0,K>0,I+J+K<=10,appetiser(A,I),main(M,J),dessert(D,K);true >
↦   < appetiser(A,I),main(M,J),dessert(D,K);I>0,J>0,K>0,I+J+K<=10 >
↦   < main(M,J),dessert(D,K);A=pasta,I=6,J>0,K>0,6+J+K<=10 >,
↦   < meat(M1,I1),dessert(D,K);
                              M=M1,J=I1,A=pasta,I=6,J>0,K>0,6+J+K<=10 >
```

Beispiel: gescheiterte Berechnung

```
    < lightmeal(A,M,D);true >
↦   < I>0,J>0,K>0,I+J+K<=10,appetiser(A,I),main(M,J),dessert(D,K);true >
↦   < appetiser(A,I),main(M,J),dessert(D,K);I>0,J>0,K>0,I+J+K<=10 >
↦   < main(M,J),dessert(D,K);A=pasta,I=6,J>0,K>0,6+J+K<=10 >,
↦   < meat(M1,I1),dessert(D,K);
                              M=M1,J=I1,A=pasta,I=6,J>0,K>0,6+J+K<=10 >
↦   < dessert(D,K);M1=beef,I1=5,M=beef,J=5,A=pasta,I=6,K>0,6+5+K<=10 >
```

Beispiel: gescheiterte Berechnung

$< \texttt{lightmeal(A,M,D);true} >$

$\mapsto \quad < \texttt{I>0,J>0,K>0,I+J+K<=10,appetiser(A,I),main(M,J),dessert(D,K);true} >$

$\mapsto \quad < \texttt{appetiser(A,I),main(M,J),dessert(D,K);I>0,J>0,K>0,I+J+K<=10} >$

$\mapsto \quad < \texttt{main(M,J),dessert(D,K);A=pasta,I=6,J>0,K>0,6+J+K<=10} >,$

$\mapsto \quad < \texttt{meat(M1,I1),dessert(D,K);}$
$\texttt{M=M1,J=I1,A=pasta,I=6,J>0,K>0,6+J+K<=10} >$

$\mapsto \quad < \texttt{dessert(D,K);M1=beef,I1=5,M=beef,J=5,A=pasta,I=6,K>0,6+5+K<=10} >$

$\mapsto \quad < \bot, \mathit{false} >$

Beispiel: Finanzberater

```
darlehen(Darlehenshoehe, Monate, Zinssatz, Rate, Restschuld)
 :- Monate=0, Darlehenshoehe=Restschuld

darlehen(Darlehen, Monate, Zinssatz, Rate, Restschuld) :-
    Monate>0,
    Monate1=Monate-1,
    Darlehen1=Darlehen+Darlehen*Zinssatz-Rate,
    darlehen(Darlehen1, Monate1, Zinssatz, Rate, Restschuld).
```

Beispiel: Finanzberater

SWI-/Sicstus-Prolog-Syntax:

```
:- use_module(library(clpr)).


darlehen(Darlehenshoehe,Monate,Zinssatz,Rate,Restschuld)
 :- {Monate=0},Darlehenshoehe=Restschuld.

darlehen(Darlehens,Monate,Zinssatz,Rate,Restschuld) :-
   {Monate>0,
   Monate1=Monate-1,
   Darlehens1=Darlehens+Darlehens*Zinssatz-Rate},
   darlehen(Darlehens1,Monate1,Zinssatz,Rate,Restschuld).
```

Beispiel: Finanzberater...

- Welcher Restschuld bleibt nach 30 Jahren für ein Darlehen von €200000 bei einer Rate von €1000 und einem monatlichen Zinssatz von 0.4%.

```
?- darlehen(200000,360,0.004,1000,S).
S = 39570.5
```

- Wie hoch ist die Rate, um das Darlehen in 30 Jahren vollständig zu zahlen?

```
?- darlehen(200000,360,0.004,X,0.0).
X = 1049.33
```

- Wieviele Monate muss man zahlen, um die Schulden zu bezahlen bei einer monatlicher Rate von €1000?

```
?- darlehen(200000,X,0.004,1000,S),{S<0.0}.
X = 404.0, S = -836.065
```

Beispiel: Finanzberater

- Welches ist das Verhältnis zwischen Darlehenshöhe und Rate, wenn man in 30 Jahren die Schulden komplett Zahlen möchte?

```
?- darlehen(D,360,0.004,R,0.0).
{R=0.00524665*D}
```

- Bei welchem Zinssatz bezahlt man die Schuld in genau 30 Jahren?

```
darlehen(200000,360,Z,1000,0.0).
```

...kann nicht von einem linearer Gleichungslöser behandelt werden.
Grund: die im zweiten Schritt aufgestellte Gleichung ist nicht mehr linear:

$$D_1 = D + D \cdot Z - R$$
$$D_2 = D_1 + D_1 \cdot Z - R$$
$$\vdots$$