# Intelligentes Spielen, Inkaufnahme von Unentschieden

Um auszudrücken, dass für A bestenfalls, und tatsächlich, ein Unentschieden erzwingbar ist:

```
to_draw(As,Bs,Xs,X) :- not(to_win(As,Bs,Xs,_)),
                       move(Xs,X,Ys), A1=[X|As],
                       (Ys=[]; to_draw(Bs,A1,Ys,_)).
```

```
?- to_draw([],[],[1,2,3,4,5,6,7,8,9],A).
A = 1 ;
…

?- move([1,2,3,4,5,6,7,8,9],A,Xs),not(to_draw([],[A],Xs,B)).
false.

?- to_draw([4,8],[3,5,9],[1,2,6,7],X).
false.

?- to_draw([8,9],[1,4,7],[2,3,5,6],X).
X = 5 ;
X = 6 ;
X = 6 ;
X = 6 ;
X = 6 ;
false.
```

# Echte Interaktivität

Unterscheidung zwischen Spieler und Gegenspieler:

```
play(As,Bs,Xs) :- read(A), !, move(Xs,A,Ys), A1=[A|As],
                  show(Bs,A1,Ys),
                  (sum15(A1); Ys=[]; play(Bs,A1,Ys)).
```

```
play(As,Bs,Xs) :- read(A), !, move(Xs,A,Ys), A1=[A|As],
                  %show(Bs,A1,Ys),
                  (sum15(A1); Ys=[]; reply(Bs,A1,Ys)).

reply(Bs,As,Ys) :- (to_win(Bs,As,Ys,B); to_draw(Bs,As,Ys,B)),
                   !, move(Ys,B,Zs), B1=[B|Bs],
                   show(As,B1,Zs),
                   (sum15(B1); Zs=[]; play(As,B1,Zs)).
```

## Code-Massage

Etwas Optimierung:

```
to_draw(As,Bs,Xs,X) :- not(to_win(As,Bs,Xs,_)),
                       move(Xs,X,Ys), A1=[X|As],
                       (Ys=[]; to_draw(Bs,A1,Ys,_)).
```

```
to_draw(As,Bs,Xs,X) :- move(Xs,X,Ys), A1=[X|As],
                       (Ys=[]; not(to_win(Bs,A1,Ys,_)),
                               to_draw(Bs,A1,Ys,_)).
```

```
to_draw(As,Bs,Xs,X) :- move(Xs,X,Ys), A1=[X|As],
                       (Ys=[]; not(to_win(Bs,A1,Ys,_))).
```

Live-Demonstration …

„Hausaufgabe“: Implementieren Sie Tic-Tac-Toe (mit „intelligentem“ Gegenspieler). (?)

# Deskriptive Programmierung

## Prolog-Spracherweiterung: DCGs

# Symbolische Sprachverarbeitung/-repräsentation (1)

- Angenommen, wir wollen Sätze der englischen Sprache modellieren.
- Wir brauchen verschiedene Kategorien von Worten und Satzteilen:

  verb, noun, verb phrase, …

  sowie Regeln zur grammatikalisch richtigen Kombination derselben:

| | | |
|---|---|---|
| sentence | → | noun phrase, verb phrase |
| noun phrase | → | determiner, noun |
| verb phrase | → | verb, noun phrase |
| | … | |

- Und natürlich einen Mechanismus, eine solche Grammatik „auszuwerten“.

# Symbolische Sprachverarbeitung/-repräsentation (2)

Einfache Umsetzung in Prolog:

- Wortkategorien + Regeln:

```
det([the]).
det([a]).

n([woman]).
n([man]).

v([knows]).
```

```
np(Z) :- det(X), n(Y), append(X,Y,Z).

vp(Z) :- v(X), np(Y), append(X,Y,Z).
vp(Z) :- v(Z).

s(Z) :- np(X), vp(Y), append(X,Y,Z).
```

- Verwendung:

```
?- s([a,woman,knows,a,man]).
true.

?- s([the,woman,knows]).
true.

?- s(Z).
Z = [the, woman, knows, the, woman] ;
...
Z = [a, man, knows].
```

Schön, aber potentiell ineffizient wegen der Art der Verwendung von **append**!

Verwendung von Akkumulatoren/„Differenzlisten“:

```
det([the]).
det([a]).

n([woman]).
n([man]).

v([knows]).
```

```
det([the|U],U).
det([a|U],U).

n([woman|U],U).
n([man|U],U).

v([knows|U],U).
```

```
np(Z) :- det(X), n(Y), append(X,Y,Z).

vp(Z) :- v(X), np(Y), append(X,Y,Z).
vp(Z) :- v(Z).

s(Z) :- np(X), vp(Y), append(X,Y,Z).
```

```
np(ZU,U) :- det(ZU,YU), n(YU,U).

vp(ZU,U) :- v(ZU,YU), np(YU,U).
vp(ZU,U) :- v(ZU,U).

s(ZU,U) :- np(ZU,YU), vp(YU,U).
```

# Symbolische Sprachverarbeitung/-repräsentation (4)

Neue Version:

```
det([the|U],U).
det([a|U],U).

n([woman|U],U).
n([man|U],U).

v([knows|U],U).
```

```
np(ZU,U) :- det(ZU,YU), n(YU,U).

vp(ZU,U) :- v(ZU,YU), np(YU,U).
vp(ZU,U) :- v(ZU,U).

s(ZU,U) :- np(ZU,YU), vp(YU,U).
```

Tests:

```
?- s([a,woman,knows,a,man],[]).
true.

?- s([the,woman,knows],[]).
true.

?- s(Z,[]).
Z = [the, woman, knows, the, woman] ;
...
Z = [a, man, knows].
```

# Symbolische Sprachverarbeitung/-repräsentation (5)

Spezielles Prolog-Feature: „Definite Clause Grammars“

```
det --> [the].
det --> [a].

n --> [woman].
n --> [man].

v --> [knows].
```

```
np --> det, n.

vp --> v, np.
vp --> v.

s --> np, vp.
```

Automatische Umsetzung:

```
?- listing.
v([knows|A], A).
np(A, C) :- det(A, B), n(B, C).
det([the|A], A).
det([a|A], A).
n([woman|A], A).
n([man|A], A).
s(A, C) :- np(A, B), vp(B, C).
vp(A, C) :- v(A, B), np(B, C).
vp(A, B) :- v(A, B).
```

# Symbolische Sprachverarbeitung/-repräsentation (6)

Bisher können wir nur testen oder generieren:

```
?- s([a,woman,knows,a,man],[]).
true.

?- s(Z,[]).
Z = [the, woman, knows, the, woman] ;
…
Z = [a, man, knows].
```

Zusätzlich würden wir gerne echt „parsen“, also mit Ausgabe der Satzstruktur.

Durch Hinzufügen eines Syntaxbaum-Arguments:

```
det(td) --> [the].
det(td) --> [a].

n(tn) --> [woman].
n(tn) --> [man].

v(tv) --> [knows].
```

```
np(tnp(T,S)) --> det(T), n(S).

vp(tvp(T,S)) --> v(T), np(S).
vp(tvp(T))   --> v(T).

s(ts(T,S)) --> np(T), vp(S).
```

# Symbolische Sprachverarbeitung/-repräsentation (7)

```
det(td) --> [the].
det(td) --> [a].

n(tn) --> [woman].
n(tn) --> [man].

v(tv) --> [knows].
```

```
np(tnp(T,S)) --> det(T), n(S).

vp(tvp(T,S)) --> v(T), np(S).
vp(tvp(T))   --> v(T).

s(ts(T,S)) --> np(T), vp(S).
```

```
?- s(T,[a,woman,knows,a,man],[]).
T = ts(tnp(td,tn),tvp(tv,tnp(td,tn))).

?- s(T,Z,[]).
T = ts(tnp(td,tn),tvp(tv,tnp(td,tn))),
Z = [the, woman, knows, the, woman] ;
...
T = ts(tnp(td,tn),tvp(tv)),
Z = [a, man, knows].

?- listing(s).
s(ts(A, C), B, E) :- np(A, B, D), vp(C, D, E).
```

# Symbolische Sprachverarbeitung/-repräsentation (8)

Eine weitere sinnvolle Verwendung von zusätzlichen Argumenten: grammatikalische Features.

- Angenommen, wir wollen Pronomen einführen:

```
det --> [the].
det --> [a].

n --> [woman].
n --> [man].

v --> [knows].

pro --> [he].
pro --> [she].
pro --> [him].
pro --> [her].
```

```
np --> pro.
np --> det, n.

vp --> v, np.
vp --> v.

s --> np, vp.
```

- Hmm:

```
?- s(Z,[]).
Z = [he, knows, he] ;
Z = [he, knows, she] ; ...
```

# Symbolische Sprachverarbeitung/-repräsentation (9)

- Korrektur mittels zusätzlicher Argumente:

```
det --> [the].
det --> [a].

n --> [woman].
n --> [man].

v --> [knows].

pro(subject) --> [he].
pro(subject) --> [she].
pro(object) --> [him].
pro(object) --> [her].
```

```
np(X) --> pro(X).
np(_) --> det, n.

vp --> v, np(object).
vp --> v.

s --> np(subject), vp.
```

- Nun:

```
?- s(Z,[]).
Z = [he, knows, him] ;
Z = [he, knows, her] ;
Z = [he, knows, the, woman] ;
Z = [he, knows, the, man] ;
Z = [he, knows, a, woman] ; …
```

# Fallstudie: Parsen arithmetischer Ausdrücke

- Zur Erinnerung:

```
expr    ::=  term + expr | term
term    ::=  factor * term | factor
factor  ::=  nat | (expr)
```

- Umsetzung in Haskell:

```
expr :: Parser Expr
expr   =  ( Add  <$>  term  <*  char '+'  <*>  expr )  |||  term

term :: Parser Expr
term   =  ( Mul  <$>  factor  <*  char '*'  <*>  term )  |||  factor

factor :: Parser Expr
factor  =  ( Lit  <$>  nat )  |||  ( char '('  *>  expr  <*  char ')' )
```

# Fallstudie: Parsen arithmetischer Ausdrücke

- Nun in Prolog:

```
expr(+(T,E)) --> term(T),"+",expr(E).
expr(T)      --> term(T).

term(*(F,T)) --> factor(F),"*",term(T).
term(F)      --> factor(F).

factor(N) --> nat(N).
factor(E) --> "(",expr(E),")".

nat(0) --> "0".
…
nat(9) --> "9".
```

(So, mit expliziten Strings, werden wir DCGs auch in der Übung machen.)

- Tests:

```
?- expr(E,"1+2*3",""), R is E.
E = 1+2*3, R = 7.

?- expr((1+2)*3,S,"").
S = [40, 49, 43, 50, 41, 42, 51] ;

?- expr((1+2)*3,S,""), writef("%s",[S]).
(1+2)*3
```

# Fallstudie: Parsen arithmetischer Ausdrücke

- Ausnutzung verschiedener Aufrufmodi:

```
parse(S,E) :- expr(E,S,"").

pretty_print(E,S) :- expr(E,S,"").

normalize(S,T) :- parse(S,E),pretty_print(E,T).
```

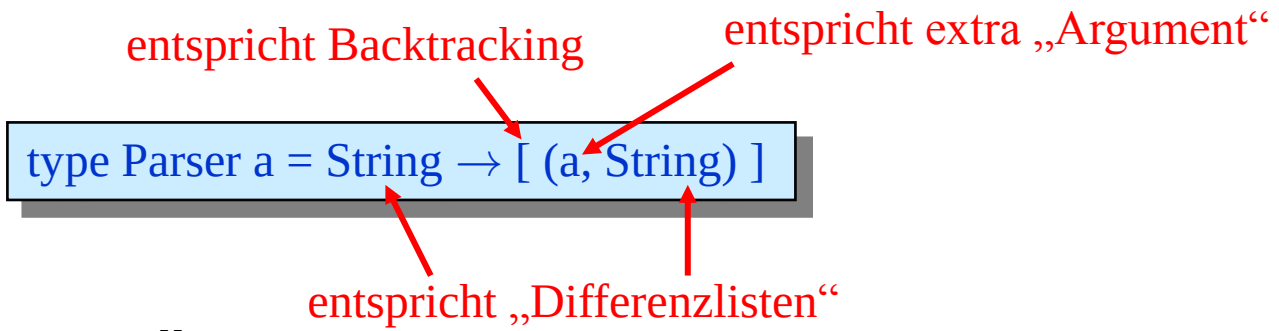
- Tests:

```
?- parse("1+(2*3)",E), R is E.
E = 1+2*3, R = 7.

?- pretty_print(1+2*3,S), !, writef("%s",[S]).
1+2*3

?- normalize("1+(2*3)",S), !, writef("%s",[S]).
1+2*3

?- normalize("(1+2)*3",S), !, writef("%s",[S]).
(1+2)*3
```

# Fallstudie: Parsen arithmetischer Ausdrücke

Etwas Reflexion zur Prolog- vs. Haskell-Lösung:

- konzeptionell:




- pragmatisch, notationell:

```
term(*(F,T)) --> factor(F),"*",term(T).
term(F)      --> factor(F).
```

vs.

```
term = (factor  ++>  \f → char  '*' +++
                            term  ++>
                            \t → yield  (Mul f t))
       |||  factor
```

oder

```
term = do f  ←  factor
          char '*'
          t  ←  term
          return (Mul f t)
       |||  factor
```

oder

```
term = ( Mul  <$>  factor  <*  char '*'  <*>  term )  |||  factor
```

# Deskriptive Programmierung

## Diverse andere Spracherweiterungen von Prolog

# Zur Erinnerung: Transitive Hülle, aber jetzt mal mit Zyklen

```
direct(frankfurt,san_francisco).
direct(frankfurt,chicago).
direct(san_francisco,honolulu).
direct(honolulu,maui).
direct(honolulu,san_francisco).

connection(X, Y) :- direct(X, Y).
connection(X, Y) :- direct(X, Z), connection(Z, Y).
```

```
?- connection(san_francisco,Y).
Y = honolulu ;
Y = maui ;
Y = san_francisco ;
Y = honolulu ;
Y = maui ;
Y = san_francisco ;
Y = honolulu ;
Y = maui ; …
```

Ziel sollte sein: Endlossuche vermeiden

## Zur Erinnerung: Transitive Hülle, aber jetzt mal mit Zyklen

Idee: schon bereiste Zwischenstationen merken, zum Beispiel als Liste:

```
direct(frankfurt,san_francisco).
…
direct(honolulu,san_francisco).

connection(X, Y) :- connection1(X, Y, [X]).

connection1(X, Y, _) :- direct(X, Y).
connection1(X, Y, L) :- direct(X, Z), not(member(Z,L)),
                        connection1(Z, Y, [Z|L]).
```

```
?- connection(san_francisco,Y).
Y = honolulu ;
Y = maui ;
Y = san_francisco ;
false.
```

Eventuell problematisch: lineare Suche in der Zwischenstationsliste.

# Fakten als Datenstruktur

Alternative: Speichern der besuchten Stationen als Prolog-Fakten.

```
direct(frankfurt,san_francisco).
…
direct(honolulu,san_francisco).

connection(X, Y) :- assert(visited(X)), connection2(X, Y).

connection2(X, Y) :- direct(X, Y).
connection2(X, Y) :- direct(X, Z), not(visited(Z)),
                     assert(visited(Z)), connection2(Z, Y).
```

```
?- connection(san_francisco,Y).
Y = honolulu ;
Y = maui ;
Y = san_francisco ;
false.

?- connection(san_francisco,Y).
Y = honolulu ;
false.
```

**Oops!**

# Fakten als Datenstruktur

„Aufräumen“:

```
direct(frankfurt,san_francisco).
…
direct(honolulu,san_francisco).

connection(X, Y) :- retractall(visited(_)),
                    assert(visited(X)), connection2(X, Y).

connection2(X, Y) :- direct(X, Y).
connection2(X, Y) :- direct(X, Z), not(visited(Z)),
                     assert(visited(Z)), connection2(Z, Y).
```

```
?- connection(san_francisco,Y).
Y = honolulu ;
Y = maui ;
Y = san_francisco ;
false.

?- connection(san_francisco,Y).
Y = honolulu ;
Y = maui ;
Y = san_francisco ;
false.
```

Beispielverwendungen der Metaprädikate **`assert`** und **`retract`**:

```
1 ?- listing.
true.

2 ?- assert(p(1)).
true.

3 ?- assert(p(1)).
true.

4 ?- assert(p(2)).
true.

5 ?- listing.

:- dynamic p/1.
p(1).
p(1).
p(2).
true.
```

```
6 ?- p(X).
X = 1 ;
X = 1 ;
X = 2.

7 ?- retract(p(1)).
true.

8 ?- p(X).
X = 1 ;
X = 2.

9 ?- retract(p(X)).
X = 1 ;
X = 2.

10 ?- listing.

:- dynamic p/1.
true.
```

## Fakten als Datenstruktur

- Eine nützliche Verwendung von **`assert`** ist Memoisierung.

- Zur Erinnerung, in Haskell (unmemoisiert):

```
fib 0 = 1
fib 1 = 1
fib n = fib (n – 1) + fib (n – 2)
```

```
fib(N,1) :- N<2, !.
fib(N,M) :- N1 is N-1, fib(N1,M1), N2 is N-2, fib(N2,M2), M is M1+M2.
```

- Das Problem:

```
?- fib(10,X).
X = 89.

?- fib(30,X).
X = 1346269.

?- fib(50,X).
```

**hoffnungslos**

# Fakten als Datenstruktur

```
fib(N,1) :- N<2, !.
fib(N,M) :- N1 is N-1, fib(N1,M1), N2 is N-2, fib(N2,M2), M is M1+M2.
```

```
:- dynamic(memo/2).

fib(N,1) :- N<2, !.
fib(N,M) :- memo(N,M), !.
fib(N,M) :- N1 is N-1, fib(N1,M1), N2 is N-2, fib(N2,M2), M is M1+M2,
            assert(memo(N,M)).
```

- Nun:

```
?- fib(10,X).
X = 89.

?- fib(30,X).
X = 1346269.

?- fib(50,X).
X = 20365011074.
```

**instantan**

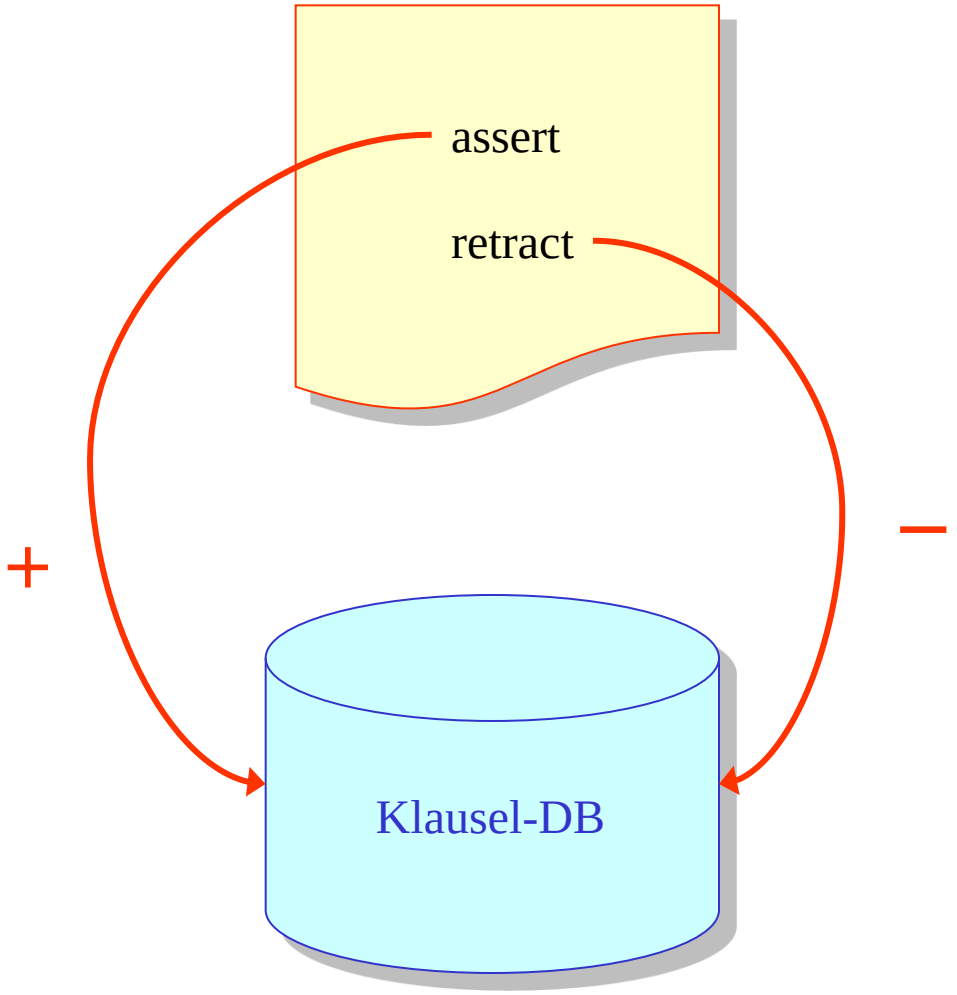
# Fakten als Datenstruktur

Seiteneffekte auf die
„Datenbank“ von Klauseln!

zwei Varianten üblich:

1) „DB“ als zusätzliche Datenstruktur (Fakten)
   ⇒ (fast schon) normal in LP

2) Selbstmodifikation des Programms
   („DB“ als Programm)
   ⇒ Meta-Programmierung



# Generierung aller Lösungen einer Anfrage (1)

- Oft existieren ja mehrere Lösungen zu einer Anfrage:

```
child(martha, charlotte).
child(charlotte, caroline).
child(caroline, laura).
child(laura, rose).

descend(X, Y) :- child(X, Y).
descend(X, Y) :- child(X, Z), descend(Z, Y).
```

Die Anfrage `?- descend(martha,X).` würde sukzessive die Antworten `X = charlotte`, `X = caroline`, `X = laura` sowie `X = rose` liefern.

- Prolog bietet drei verschiedene Meta-Prädikate, um alle Lösungen „auf einen Schlag“ zu generieren:

  `findall`, `bagof`, `setof`

  und sie auf jeweils eine bestimmte Art in einer Ergebnisliste aufzuführen.

# Generierung aller Lösungen einer Anfrage (2)

```
findall(Template, Goal, List).
```

- Für jede Lösung der Anfrage `Goal` wird das instantiierte `Template` in die Ergebnisliste `List` aufgenommen.

```
?- findall(X, descend(martha, X), Z).
Z = [charlotte, caroline, laura, rose].
```

- Der Term `Template` kann auch eine ganze Struktur mit (oder ohne) Variablen sein, woraus dann die Einträge der Ergebnisliste „gebaut“ werden.

```
?- findall(fromMartha(X), descend(martha, X), Z).
Z = [fromMartha(charlotte), fromMartha(caroline),
                fromMartha(laura), fromMartha(rose)].
```

# Generierung aller Lösungen einer Anfrage (3)

Variante: `bagof(Template, Goal, List).`

Die nicht im `Template` vorkommenden freien Variablen werden getrennt gebunden:

```
?-  bagof(X, descend(Y, X), Z).
Y = caroline,
Z = [laura, rose] ;

Y = charlotte,
Z = [caroline, laura, rose] ;

Y = laura,
Z = [rose] ;

Y = martha,
Z = [charlotte, caroline, laura, rose].
```

Zum Vergleich:

```
?- findall(X, descend(Y, X), Z).
Z = [charlotte, caroline, laura, rose, caroline, laura, …].
```

# Generierung aller Lösungen einer Anfrage (4)

weitere Variante: `setof(Template, Goal, List).`

… verhält sich wie **`bagof`**, allerdings werden Duplikate aus der Ergebnisliste gelöscht, und die Liste sortiert.

Denkbare Verwendung der „Collection“-Prädikate: Simulation von list comprehensions.

Haskell: `[ e | x ← xs ]`

Prolog:

```
findall(E, member(X,Xs), List).
```

# Generierung aller Lösungen einer Anfrage (5)

Beispiele:

Prolog-Äquivalente zu folgenden Haskell-Definitionen?

1. `[ n .. m ]`
2. `[ n, m .. l ]`
3. `[ x * x | x ← [1 .. 100], x `mod` 2 == 0 ]`

Mögliche Lösungen zu 1.:

```
fromTo(N,M,L)      :- N > M, !, L = [].
fromTo(N,M,[N|L]) :- N1 is N+1, fromTo(N1,M,L).
```

oder

```
fromTo(N,M,L) :- findall(X,between(N,M,X),L).
```

# Generierung aller Lösungen einer Anfrage (6)

Beispiele:

Prolog-Äquivalente zu folgenden Haskell-Definitionen?

2. `[ n, m .. l ]`

3. `[ x * x | x ← [1 .. 100], x `mod` 2 == 0 ]`

Mögliche Lösungen zu 2.:

(Haskell „erlaubt“ übrigens auch [ 0, 0 .. 5 ] und [ 0, -2 .. -5 ].)

```
fromThenTo(N,M,L,Xs)     :- (N >= M; N > L), !, Xs = [].
fromThenTo(N,M,L,[N|R]) :- M1 is M+M-N, fromThenTo(M,M1,L,R).
```

oder

```
fromThenTo(N,M,L,Xs) :- (N >= M; N > L), !, Xs = [].
fromThenTo(N,M,L,Xs) :- D is M-N, fromTo(0,(L-N)/D,Is),
                        findall(X,(member(I,Is), X is N+I*D),Xs).
```

## Generierung aller Lösungen einer Anfrage (7)

Beispiele:

Prolog-Äquivalente zu folgenden Haskell-Definitionen?

3. `[ x * x | x ← [1 .. 100], x `mod` 2 == 0 ]`

Mögliche Lösungen zu 3.:

```
squares(L) :- fromTo(1,100,Xs), filter(Xs,Ys), map(Ys,L).

filter([],[]).
filter([X|Xs],[X|Ys]) :- X mod 2 =:= 0, !, filter(Xs,Ys).
filter([_|Xs],Ys)     :- filter(Xs,Ys).

map([],[]).
map([X|Xs],[Y|Ys]) :- Y is X*X, map(Xs,Ys).
```

oder

```
squares(L) :- fromTo(1,100,Xs),
              findall(Y,(member(X,Xs), X mod 2 =:= 0, Y is X*X),L).
```