Prolog

Contents

Prolog Basics

Terme

Atome

Atome beginnen mit Kleinbuchstaben:
hans, inge, fritz, fisch

Komplexere Atome: in Hochkommata:
’Hallo Fritz!’

Zahlen, Variablen und Fakten sind keine Atome! Atome sind: links, fritz, panzer, brot

Variablen

Platzhalter für unbekannte Terme.
Variablen beginnen mit Großbuchstaben oder Unterstrich:
X, Y, _X, X1, Fisch

Beispiele:		
	`liebt(franz,X).`	Franz liebt alles
	`liebt(X,fussball).`	Alle lieben Fußball
	`liebt(X,X).`	Alles liebt sich selbst
	`liebt(X,Y).`	jeder liebt alles
	`bekannt((X,3)).`	Alle Terme der Gestalt `(...,3)` sind bekannt

Anonyme Variablen

Anonyme Variablen werden mit _ ausgedrückt.
Genau wie bei Haskell: test(_, X) -> egal was auf der 1.Stelle steht, interessiert nicht, wird nicht verwendet, muss nicht unifiziert werden.

```
test(N,X,L) :- HD is N-X,
               ND is N+X,  N1 is N-1, test1(HD,ND,N1,L).

test1(_,_,0,[]).
test1(HD,ND,N,[P|L]) :- N > 0,
                        ND =\= N+P,
                        HD =\= N-P, N1 is N-1,
                        test1(HD,ND,N1,L).
```

- anonyme Variable _: Variable interessiert nicht
- Alle Vorkommen von _ unterschiedlich (werden nicht unifiziert)

Fakten

Zusammengesetzte Terme:
liebt(fritz,fisch), liebt(fritz,X)

Prolog-Programm: Bisher bekannte, variablenfreie Terme

```
bekannt(hans).                bekannt(liebt).
bekannt(inge).                bekannt(3).
bekannt(fritz).               bekannt(4.5).
bekannt(heinz).               bekannt(liebt(fritz,fisch)).
bekannt(fritz).               bekannt((3,4)).
bekannt('Hallo␣Fritz!').      bekannt(plus).
                              bekannt(<).
```

- Terme `hans`, `fritz` sind bekannte Terme
- Aber auch der zusammengesetze Term: `liebt(fritz,fisch)`

Zahlen

`3, 4.5`

Abfragen

Alle Fakten werden zu Laufzeit in einer Datenbank gehalten. Eine ”Abfrage“ wird durch ein Fragezeichen **?** eingeleitet und mit einem Punkt .
beendet:

```
?liebt(fritz,fisch).
```

Konjunktion von Abfragen

Konjunktion von Teilzielen getrennt durch Komma “,”:

```
?liebt(X,inge), liebt(inge,Y) .
```

Beispiel

```
liebt(hans,inge).

liebt(heinz,inge).

liebt(inge,fritz).

liebt(fritz,fisch).


?liebt(X,inge), liebt(inge,Y) .


-> X = hans, Y = fritz

-> X = heinz, Y = fritz

-> false
```

Regeln

Syntax

```
term :- termlist .
```

Wobei :- als ”Wenn“ zu lesen ist und Kommata in termlist als ”Und“, wie in Abfragen.

Beispiel

Wenn Inge X liebt und wenn X Fisch liebt, dann liebt Hugo X:
```
liebt(hugo,X) :- liebt(inge,X),liebt(X,fisch).
```

Prädikate

Eine Gruppe von Fakten/Regeln mit gleichem Funktor und gleicher Argumentzahl im Regelkopf heißt "Prozedur" oder "Prädikat"

```
testA(X) . - einstelliger Prädikat
testB(X, Z) . - zweistelliger Prädikat
test((A,B,C)) . - Einstelliger Prädikat, Tupel als Argument
```

Beispiel

```
grandparent(X,Y) :- parent(X,Z),parent(Z,Y).
parent(X,Y) :- mother(X,Y).
parent(X,Y) :- father(X,Y).
mother(inge,emil).
mother(inge,petra).
mother(petra,willi).
father(fritz,emil).
father(emil,kunibert).
```

father, mother, parent, grandparent sind Prädikate

Listen

[X|Y] ≡ '[|]'(X,Y) $[Z_1, Z_2, \ldots, Z_n] \equiv [Z_1 | [Z_2 | [\ldots[Z_n | []]\ldots]]]$

- '[|]' ist der Cons-Operator [2]
- X ist das erste Element der Liste (head)
- Y ist der Rest der Liste (tail)
- [] ist die leere Liste, (ein vordefiniertes Atom)
- Y muss nicht instanziiert sein

⇒ Listen können von vorn aufgebaut werden (anders als in Haskell)

- Achtung: [X|Y] trennt bei Unifikation Listenkopf- und -rest, nicht: die Liste irgendwo in der Mitte!

```
  ?[X|Y] = [1,2,3].
⇒ X = 1, Y = [2, 3].
```

[2] '.' in älteren Prolog-Versionen

Plain: [X, Y, Z, A, B, C]
Head: [X | Rest]

Two heads: [X | **[Y | Rest]**]

Unendliche Listen

Die `rev` Funktion lässt sich in beiden Implementierungen rückwärts ausführen. Die Abfrage

```
?rev(X,[1|R])
```

erzeugt alle Listen, deren Umkehrung mit 1 beginnt

```
X=[1], R=[];
X=[_137,1], R=[_137];
X=[_138,_137,1], R=[_137,_138];
...
```

Für unbekannte Listenelemente werden nichtinstanziierte (intern durchnummerierte) Variablen verwendet. Das Ziel der Abfrage ist unendlich oft re-erfüllbar.

Arithmetik und Vergleich

Unifikation

Unifikation gilt auch für **uninstanziierte** Variablen

```
Unifikation: = (wertet die beiden Seiten NICHT aus)
?- 3 = 3.  -> true
?- 1 + 2 = 3.  -> false


Unifiziert nicht: \=
?- 2 \= 3.  -> true
```

Vergleichsoperatoren

Gilt nur für **vollständig instanziierte** Variablen!

```
Equal: =:=   (wertet die beiden Seiten aus)
?- 1 + 2 =:= 3.  -> true

Not equal: =\= (vertet die beiden Seiten aus)
?- 2 + 4 =\= 1.  -> true
?- 1 =\= 1 -> false

<, =<, >, >= arithmetischer Vergleich
?- 1+2 <= 1 -> false
```

Auswertung mit is

Variablen im **rechten** Term müssen **instanziiert** sein!

(Der linke Term kann uninstanziierte Variablen enthalten)

```
Auswertung und Unifikation: is
?X is 3+3.  -> true, X=6
?1 is 3*3.  -> false
?X is 0*Y   -> ERROR: Arguments not sufficiently instantiated
```

“Zuweisungs”-Teilziel X **is** X + 1 nie erfolgreich!

Methoden

Listenfunktionen

Member

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

X kommt in einer Liste vor, wenn es mit dem ersten Element unifizierbar ist oder wenn X im Listenrest R vorkommt.

Member2

member2(X,L) ist erfüllt, wenn L zwei aufeinanderfolgende Vorkommen von X enthält.

```
member2(X,[X|[X|Rest]]).
member2(X,[Y|Rest]) :- member2(X, Rest).
```

Append

```
append([],L,L).
append([X|R],L,[X|T]) :- append(R,L,T).
```

Die Konkatenation von [] und L ist L. Wenn die Konkatenation von R und L die Liste T ergibt, dann ergibt die Konkatenation von [X|R] und L die Liste [X|T].

```
?append([1,2,3,4],[2,3,4,5],X).
-> (einzige) Ausgabe: X = [1,2,3,4,2,3,4,5]
```

Reverse

Naive, aber reicht

```
rev([],[]).
rev([X|R],Y) :- rev(R,Y1),append(Y1,[X],Y).
```

Effizienter

```
rev(X,Y) :- rev1(X,[],Y).
rev1([],Y,Y).
rev1([X|R],A,Y) :- rev1(R,[X|A],Y).
```

Quicksort

```
qsort([],[]).
qsort([X|R],Y) :-   split(X,R,R1,R2),
                    qsort(R1,Y1),
                    qsort(R2,Y2),
                    append(Y1,[X|Y2],Y).
split(X,[],[],[]).
split(X,[H|T],[H|R],Y) :- X>H, split(X,T,R,Y).
split(X,[H|T],R,[H|Y]) :- X=<H, split(X,T,R,Y).
```

Listenpermutation

```
permute([],[]).
permute([X|R],P) :- permute(R,P1),append(A,B,P1),append(A,[X|B],P).
```

Beispielabfrage: `?permute([1,2,3],Y).`

- Unifikation mit `permute([X|R],P)`

⇒ `X=1, R=[2,3], P=Y`

permute(R,P1)	append(A,B,P1)	append(A,[X|B],P)
permute([2,3],P1) ⇒P1=[2,3]	append(A,B,[2,3]) ⇒A=[], B=[2,3]	append([],[1,2,3],P) ⇒P=[1,2,3]
	append(A,B,[2,3]) ⇒A=[2], B=[3]	append([2],[1,3],P) ⇒P=[2,1,3]
	append(A,B,[2,3]) ⇒A=[2,3], B=[]	append([2,3],[1],P) ⇒P=[2,3,1]
permute([2,3],P1) ⇒P1=[3,2]	append(A,B,[3,2]) ⇒A=[], B=[3,2]	append([],[1,3,2],P) ⇒P=[1,3,2]

even und odd

```
RICHTIG:
even(0).
even(X) :- X>0, X1 is X-1, odd(X1).

odd(1).
odd(X) :- X>1, X1 is X-1, even(X1).

?even(2) -> Yes
```

Aber Achtung: Arithmetische Ausdrücke **unifizieren nicht** mit Konstanten!

FALSCH:

~~even(0)~~.

~~even(X)~~ :- X>0, odd(X-1).

~~odd(1)~~.

~~odd(X)~~ :- X>1, even(X-1).

?even(2) -> No

atom

atom(Term) -> True if *Term* is bound to an atom.

```
?- atom(pizza). -> true.
?- atom(likes(mary, pizza)). -> false.
?- atom(235). -> false.
```

Zahlen, Variablen und Fakten sind keine Atome! Atome sind: links, fritz, panzer, brot

Fibonacci

Einfach:

- Berechnung `X`-ter Fibonacci-Zahl: Suche passende Instanziierung von zusätzlichem Parameter `Y`

```
fib(0,0).
fib(1,1).
fib(X,Y) :- X>1,
   X1 is X-1,  X2 is X-2,
   fib(X1,Y1), fib(X2,Y2),
   Y is Y1+Y2.

 ?fib(3,Y).        ?fib(3,2).        ?fib(3,42)
⇒Yes, Y=2        ⇒ Yes             ⇒ No
```

- Formal ähnlich zum pattern matching
- Der Test `X>1` (ein Wächter) verhindert eine Endlosresolution für einen Fall wie `?fib(0,X),1<0.`

Mit Cuts:

```
fib(0,Y) :- !,Y=0.
fib(1,Y) :- !,Y=1.
fib(X,Y) :-  X1 is X-1, X2 is X-2,
             fib(X1,Y1), fib(X2,Y2),
             Y is Y1+Y2.


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

Ein Negationsprädikat ist in Prolog ohne (roten) Cut nicht ausdrückbar.

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

call ruft das Argument als Teilziel auf und schlägt fehl, wenn dieses fehlschlägt

fail schlägt immer fehl

Anwendung: `?not(5<6).` ⇒ `no` und `?not(6<5).` ⇒ `yes`

Allerdings ist das `not` Prädikat nicht zu verwechseln mit der klassischen Logik! So ergibt `?not(liebt(X,inge)).` die Antwort `no`, anstatt einer Menge von `X` Instanziierungen.

Prolog Expert

Generate and Test

Prolog ist besonders gut

- für systematisches Durchprobieren (z.B. Branch and Bound)
- mittels mehrfach reerfüllbarer Prädikate
- erzeugen Lösungskandidaten, welche danach getestet werden

Funktion :- Generator-Teil, Tester-Teil

Vermeidung kombinatorischer Explosion: **Generator** möglichst intelligent machen

Generate

Beispiel für unendlich oft reerfüllbares Prädikat:

```
nat1(0).
nat2(X) :- nat(Y), X is Y+1.
```

Test

Verwendung zum systematischen Durchprobieren natürlicher Zahlen:

```
sqrt(X,Y) :- nat(Y),
             Y2 is Y*Y, Y3 is (Y+1)*(Y+1),
             Y2 =< X, X < Y3.
?sqrt(27,X). -> liefert X=5
```

Ausführungsbaum

Rückwartsausführung

```
?append(X,Y,[1,2,3]).
```

```
-> X = [], Y = [1,2,3];
-> X = [1], Y = [2,3];
-> X = [1,2], Y = [3];
-> X = [1,2,3], Y = []
```

Der Cut

Determinismus

Definition

Ein Prädikat heißt deterministisch, wenn es stets auf höchstens eine Weise erfüllt werden kann; hat es möglicherweise mehrere Lösungen, so heißt es nichtdeterministisch.

- „Cut" ermöglicht die Beeinflussung des Backtrackings und das Abschneiden von Teilen des Ausführungsbaums
- Syntax: `!`
- Beispiel: `p(X) :- a(X), !, b(X).`
- als Teilziel immer erfüllbar, aber Reerfüllungsversuch lässt alle Teilziele links vom Cut sofort fehlschlagen
  Im Beispiel: Sollte `b(X)` fehlschlagen, dann auch sofort `a(X)` und `p(X)`
- Implementierung: Cut löscht beim ersten Aufruf alle Choice-Points in allen Boxen für Teilziele links von der Cut-Box
  Im Beispiel: evtl. vorhandene Choice-Points für `p(X)` und `a(X)`

Sehr wichtig: **als Teilziel immer erfüllbar**, aber **Reerfüllungsversuch** lässt alle Teilziele **links vom Cut sofort fehlschlagen**.
Im Beispiel: Sollte b(X) fehlschlagen, dann auch sofort a(X) und p(X)

Beispiel

```
a(x) :- b(x),c(x),d(x).
a(x) :- e(x),f(x),!,g(x).
a(x) :- h(x),i(x),j(x).
```

- Die erste Regel wird normal ausgeführt
- Für die zweite Regel werden die ersten beiden Teilziele normal ausgeführt (was bedeuten kann, dass `e` mehrfach erfüllt wird, solange bis `f` erfüllt wird)
- Sobald der Cut ausgeführt wird, werden alle Choice-Points von `e`, `f` und `a` gelöscht.
- `g` kann beliebig viele Lösungen produzieren, was jeweils zu Lösungen von `a` führt.
- Schlägt `g` einmal fehl, dann auch `e`, `f` und `a`, was bedeutet, dass die dritte Regel nie ausgeführt wird.
- Schlägt `e` fehl bevor der Cut aufgerufen wurde, kann die dritte Regel zur Resolution verwendet werden.

Blaue, grüne und rote Cuts

Blauer Cut: beeinflusst weder Programmlaufzeit, noch -verhalten
Grüner Cut: beeinflusst Programmlaufzeit, aber nicht -verhalten
Roter Cut: beeinflusst das Programmverhalten

Beispiel: Grüner Cut

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

Als Programmierer weiß man im Gegensatz zum Prologsystem, dass `max` deterministisch ist, weshalb wir einen grünen Cut einfügen:

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

$\Rightarrow$ schnellere Ausführung und weniger Speicherbedarf

Beispiel: Roter Cut

Version mit grünem Cut:

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

Rote Cuts werden verwendet um Wächter zu ersetzen. Ist der erste Wächter erfolgreich, wird der zweite nie angewendet:

```
max(X,Y,X) :- X>Y,!.
max(X,Y,Y).
```

Doch nun ist das Programm fehlerhaft. Es ist z.B. `max(3,2,2)` nun erfüllbar, dank der zweiten Regel. Das dritte Argument muss uninstanziiert sein:

```
max(X,Y,Z) :- X>Y,!,Z=X.
max(X,Y,Y).
```

Vorteil: Effizienzgewinn

Rote Cuts können zu erheblichem Effizienzgewinn führen, da sie u.U. sehr komplexe und teuere Wächter ersetzen.
Beispiel: Sei `B'` ein Prädikat, das genau dann erfüllbar ist, wenn `B` nicht erfüllbar ist. Dann kann man

```
A(X) :- B(X), C(X).
A(X) :- B'(X),D(X).
```

ersetzen durch

```
A(X) :- B(X),!,C(X).
A(X) :- D(X).
```

vorausgesetzt, `B` ist deterministisch und wird (unabhängig von der Instanziierung von der Termliste X) immer ausgeführt.

Faustregel

Faustregel: Der Cut darf erst kommen, wenn man weiß, dass **man in der richtigen Regel ist**, aber muss **vor der Instanziierung der Ausgabevariablen** stehen.

Dictionaries

Dictionary: Instanziierungen `(N,A)` von Wortsequenz-„Variablen"

- Verwendung von partiell instantiierten Variablen, z.B.:

```
D=[(1,[ich,bin])|_],   D=[(1,[ich,denke]),(2,[ich,bin])|_]
```

Zugriff auf Dictionaries:

```
lookup(N,[(N,A)|_],A1) :- !,A=A1.
lookup(N,[_|T],A) :- lookup(N,T,A).
```

- Nachsehen von Wert A von N in D: `lookup(N,D,A)`
  N instanziiert, D partiell instanziiert, A uninstanziiert

  `? lookup(1,[(1,[ich,bin])|_],A).` ⇒ `Yes, A=[ich,bin].`
- Eintragen von Wert A für N in D: `lookup(N,D,A)`
  N instanziiert, D partiell instanziiert, A instanziiert

  `? D=[(1,[ich,denke])|_], lookup(2,D,[ich,bin]).`

  ⇒ `Yes, D=[(1,[ich,denke]),(2,[ich,bin])|_].`

⇒ `lookup` vorwärts und rückwärts verwendbar

Exotische Programme

Differentialrechnung

Symbolisches Differenzieren

Erinnerung: Terme A+B, A∗B, . . . stehen für sich selbst

```
diff(X,X,1) :- atom(X),!.
diff(X,Y,0) :- atomic(X),!.
diff(A+B,X,DA+DB) :- diff(A,X,DA), diff(B,X,DB),!.
diff(A-B,X,DA-DB) :- diff(A,X,DA), diff(B,X,DB),!.
diff(A*B,X,DA*B+A*DB) :- diff(A,X,DA), diff(B,X,DB),!.
diff(A/B,X,(DA*B-A*DB)/(B*B)) :- diff(A,X,DA), diff(B,X,DB),!.
diff(A**N,X,N*(A**N1)*DA) :- integer(N),N1 is N-1,diff(A,X,DA),
diff(sin(A),X,cos(A)*DA) :- diff(A,X,DA),!.
diff(ln(A),X,DA/A) :- diff(A,X,DA),!.
```

Weitere Differenzierungsregeln: einfach hinzuzufügen

atom ist x mit einem Atom instanziiert?

atomic ist x mit einem Atom oder einer Zahl instanziiert?

```
   ?diff(5*sin(x-y),x,D).
⇒ Yes, D = 0*sin(x-y)+5*cos(x-y)*1
```

- Vereinfachung für ∗0, +0, ∗1 ⇒ Prädikat simplify

Vereinfachung von Termen

```
simplify(A+B,S) :- simplify(A,SA), simplify(B,SB), simp(SA+SB,S)
simplify(A*B,S) :- simplify(A,SA), simplify(B,SB), simp(SA*SB,S)
...
simp(X+0,X) :- !.
simp(0+X,X) :- !.
simp(X*0,0) :- !.
simp(X*1,X) :- !.
...
simp(X+Y,Z) :- integer(X),integer(Y),Z is X+Y,!.
simp(X*Y,Z) :- integer(X),integer(Y),Z is X*Y,!.
simp(X,X).
```

Die letzte Regel wird als “Catch-All-Regel” bezeichnet, die alle Fälle abdeckt, die übrig bleiben.

Acht Damen Problem

Einfach, naiv:

```
achtdamen(L) :- permute([1,2,3,4,5,6,7,8],L), test(L).
```

Effizienter:

```
achtdamen(L) :- solution(8,L).
solution(0, []).
solution(N,[X|L]) :- N > 0, N1 is N-1, solution(N1,L),
   member(X,[1,2,3,4,5,6,7,8]), test(N,X,L).
```

- Nutze Reerfüllbarkeit von `member`.
- Teste, ob neue Damen `X` bestehende Damen in `L` diagonal oder horizontal bedroht
- Notwendig, da Generator `member` Damen auch in Zeilen setzt, die in `L` schon belegt sind.

⇒ “push the tester into the generator”

More Money

More Money

Zahlenrätsel:

$$\begin{array}{ccccc} & S & E & N & D \\ + & M & O & R & E \\ \hline M & O & N & E & Y \end{array}$$

- Ziffern 0 . . . 9, S,M > 0, alle Ziffern verschieden

Naive Lösung: per Generate and Test

```
solve([S,E,N,D,M,O,R,Y]) :- digit(S), S>0, digit(M), M>0,
                            digit(E), digit(N), digit(D),
                            digit(O), digit(R), digit(Y),
                            ha(D,E,0,Y,C1), ha(N,R,C1,E,C2),
                            ha(E,O,C2,N,C3), ha(S,M,C3,O,M),
                            allDifferent([S,E,N,D,M,O,R,Y]).
digit(0). digit(1).  ... digit(9).

ha(X,Y,C,R,NC) :- R is (X+C+Y) mod 10, NC is (X+C+Y) // 10.

allDifferent([]) :- !.
allDifferent([H|T]) :- not(member(H,T)), allDifferent(T).
```

- Halbaddierer `ha(X,Y,C,R,NC)`
  `X,Y,C` instanziiert, Ergebnis `R`, neuer Übertrag `NC`

Aufgaben

Rot-Schwarz-Bäume

Aufgabe

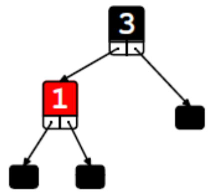
Aufgabe 3 (Prolog, Rot-Schwarz-Bäume) [20 Punkte]

Rot-Schwarz-Bäume sind gefärbte binäre Bäume mit den Eigenschaften

(i) sortiert

(ii) kein roter Knoten hat roten Kindknoten

(iii) alle vollständigen Pfade haben gleiche Anzahl schwarzer Knoten



Blätter zählen dabei als schwarze Knoten.

Gefärbte binäre Bäume lassen sich als Prolog-Terme darstellen, z.B., der abgebildete Baum `TExample` als `node(black, node(red, leaf, 1, leaf), 3, leaf).`

(a) Definieren Sie ein Prädikat `sorted(T)`, das genau dann erfüllt ist, wenn `T` Eigenschaft (i) erfüllt. [9 Punkte]

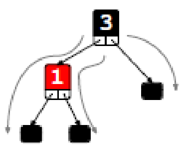
Hinweis: Es gilt: Blätter `leaf` sind sortiert, sowie: Knoten `node(Color, Left, X, Right)` sind sortiert, falls

- alle Knoten in `Left` Werte ≤ `X` haben,
- alle Knoten in `Right` Werte ≥ `X` haben, und
- die Bäume `Left` und `Right` sortiert sind.

Definieren Sie geeignete Hilfsprädikate.

(b) Definieren Sie ein Prädikat `colorPath(T,P)`, das bei Wiedererfüllung alle vollständigen Pfade durch `T` — repräsentiert als Liste von Farben — aufzählt. [5 Punkte]

```
? colorPath(TExample,P).
P = [black, red, black] ;
P = [black, red, black] ;
P = [black, black].
```



(c) Definieren Sie ein Prädikat `redRed(T)` das genau dann erfüllt ist, wenn `T` Eigenschaft (ii) *verletzt*. [6 Punkte]

Hinweis: Definieren Sie `redRed` direkt, *oder*: mittels `colorPath(T,P)` zusammen mit einem Hilfsprädikat `member2(X,L)` welches erfüllt ist, wenn `L` zwei aufeinanderfolgende Vorkommen von `X` enthält.

Lösung

Beispiellösung:

(a)
```
lt(_X, leaf).
lt(X,  node(_, Left, Y, Right)) :-
        X >= Y,
        lt(X,Left),
        lt(X,Right).
```

```
gt(_X, leaf).
gt(X,  node(_, Left, Y, Right)) :-
        Y >= X,
        gt(X,Left),
        gt(X,Right).
```

```
sorted(leaf).
sorted(node(_, Left, X, Right)) :-
        lt(X,Left), gt(X,Right),
        sorted(Left), sorted(Right).
```

(b)
```
colorPath(leaf, [black]).
colorPath(node(Color, Left, _,      _), [Color|LPath]) :-
        colorPath(Left,  LPath).
colorPath(node(Color,     _, _, Right), [Color|RPath]) :-
        colorPath(Right, RPath).
```

(c)
```
redRed(node(red, node(red, _, _, _), _, _)).
redRed(node(red, _, _, node(red, _, _, _))).
redRed(node(_, Left, _, _)) :- redRed(Left).
redRed(node(_, _, _, Right)) :- redRed(Right).
```

oder

```
member2(X,[X|[X|_Rest]]).
member2(X,[_Y|Rest]) :- member2(X, Rest).
redRed2(T) :- colorPath(T, Path), member2(red, Path).
```

Gewichtete Bäume

Aufgabe

Aufgabe 3 (Prolog, gewichtete Bäume) [18 Punkte]

Es seien in Prolog binäre Bäume rekursiv durch Terme `leaf(W)` und `node(T1, W, T2)` dargestellt, wobei `W` ein ganzzahliges Knotengewicht und `T1`, `T2` Unterbäume sind:

```
        2
       / \
      3   4
     / \
    1   1
```

`node(node(leaf(1), 3, leaf(1)), 2, leaf(4))`

Wir nennen einen solchen Baum *gewichtsbalanciert*, wenn die Summe der Gewichte auf jedem Pfad von einem Blatt zur Wurzel gleich ist. Der obige Baum ist gewichtsbalanciert, ebenso wie die folgenden Bäume mit den gleichen Gewichten 1, 1, 2, 3, 4:

```
    2              4              4
   / \            / \            / \
  4   3          2   3          3   2
     / \        / \                / \
    1   1      1   1              1   1
```

Ziel dieser Aufgabe ist es, zu einer Liste von Gewichten alle gewichtsbalancierten Bäume nach dem Prinzip *Generate and Test* zu finden.

Ziel dieser Aufgabe ist es, zu einer Liste von Gewichten alle gewichtsbalancierten Bäume nach dem Prinzip *Generate and Test* zu finden.

(a) Implementieren Sie ein Prolog-Prädikat `makeTree(Ws, T)`, das zu einer Liste von Gewichten `Ws` bei Reerfüllung `T` *jeden* binären Baum zuweist, dessen Inorder-Traversierung `Ws` entspricht (d.h. die Gewichte kommen in der Termdarstellung in der gleichen Reihenfolge wie in der Liste vor). [7 Punkte]

Beispiel:

```
?- makeTree([1,2,3,4,5], T).
T = node(leaf(1), 2, node(leaf(3), 4, leaf(5))) ;
T = node(node(leaf(1), 2, leaf(3)), 4, leaf(5)) ;
false.

?- makeTree([], T).
false.
```

Hinweis: Das Prädikat `append(Xs, Ys, XYs)` könnte nützlich sein!

(b) Implementieren Sie ein Prädikat `balanced(T, S)`, das `S` die Summe der Gewichte auf einem Pfad von einem Blatt zur Wurzel von `T` zuweist, wenn diese für jeden solchen Pfad übereinstimmt, und andernfalls fehlschlägt. [7 Punkte]

(c) Implementieren Sie schließlich ein Prädikat `makeBalanced(Ws, T)`, das zu einer Liste von Gewichten `Ws` bei Reerfüllung `T` jeden gewichtsbalancierten Baum zurückgibt, der genau die Gewichte aus `Ws` enthält. [4 Punkte]

Hinweis: Benutzen Sie das aus der Vorlesung bekannte Prädikat `permute(Xs, Ys)`. Sie müssen doppelte Lösungen nicht ausfiltern.

Lösung

```
makeTree([W], leaf(W)).
makeTree(Ws, node(T1, W, T2)) :-
    append(Ts1, [W|Ts2], Ws),
    makeTree(Ts1, T1),
    makeTree(Ts2, T2).
```

```
balanced(leaf(N), N).
balanced(node(T1, N, T2), S) :-
    balanced(T1, S1),
    balanced(T2, S1), S is S1 + N.
```

```
makeBalanced(Ws, T) :-
    permute(Ws, Ws2),
    makeTree(Ws2, T),
    balanced(T, _).
```

Erklärung

makeTree(List, T) -> List = (Value), nur 1 Element -> T = leaf(Value)

makeTree(List, T) -> List = (LeftPart, Value, RightPart), LeftPart and RightPart not empty,
T = node(makeTree(LeftPart), Value, makeTree(RightPart))

Ableitungen von KFG

Aufgabe

Aufgabe 2 (Prolog, Nullableitungen) [18 Punkte]

Gegeben sei eine kontextfreie Grammatik $G = (\Sigma, V, P, S)$. Eine Sequenz $\alpha \in (\Sigma \cup V)^*$ von Terminalen und Nichtterminalen heißt *nullable*, wenn sich aus α die leere Sequenz ϵ ableiten lässt ($\alpha \Rightarrow^* \epsilon$).

In Prolog notieren wir: Nichtterminale A als Ausdrücke `nonterm(A)`, Terminale x als Ausdrücke `term(x)`, Sequenzen α als Liste solcher Ausdrücke, und Mengen P von Produktionsregeln $A \to \alpha$ als Liste von Paaren `(A,α)`. Z.B. notieren wir die Produktionsregeln P der Grammatik

$$\begin{array}{lcl} S & \to & T\ S' \\ S' & \to & \underline{+}\ S \mid \varepsilon \\ T & \to & \underline{\mathbf{value}} \mid \underline{(}\ S\ \underline{)} \end{array}$$

als

```
[  (nonterm(s),  [nonterm(t), nonterm(s_)]),
   (nonterm(s_), [term(+), nonterm(s)]),
   (nonterm(s_), []),
   (nonterm(t),  [term(value)]),
   (nonterm(t),  [term(lp), nonterm(s), term(rp)])
]
```

(a) Definieren Sie ein Prädikat `derive(P, α, β)` das genau dann erfüllt ist, wenn β in *einem* Schritt aus α ableitbar ist ($\alpha \Rightarrow \beta$). Insbesondere sollen alle solche β bei Reerfüllung generiert werden. [7 Punkte]

Hinweis: Verwenden Sie `member`, `append`.

(b) Definieren Sie ein Prädikat `nullable(P, α)` das genau dann erfüllt ist, wenn α *nullable* ist. [11 Punkte]

Hinweis: Die Aufgabe lässt sich *nicht* lösen, indem lediglich `derive` wiederholt angewandt wird. Um nachzuweisen, dass eine Sequenz *nullable* ist, merken wir uns (um nicht unendlich-lange Beweise zu suchen), welche Nonterminale nicht mehr untersucht werden dürfen. Es gilt:

- Die leere Sequenz ϵ ist *nullable*.
- Eine Sequenz $A\beta$ ist *nullable* falls
  - Nichtterminal A noch untersucht werden darf,
  - $A \to \alpha$ eine Produktion in P ist,
  - ohne nochmals A zu untersuchen nachweisbar ist, dass α *nullable* ist, und
  - nachweisbar ist, dass β *nullable* ist.

Definieren Sie also ein passendes drei-stelliges Hilfsprädikat. Verwenden Sie **not**, `member`.

Lösung

Beispiellösung:

(a)
```
derive(P, [nonterm(A)|Beta], AlphaBeta) :-
    member((nonterm(A),Alpha), P),
    append(Alpha, Beta, AlphaBeta).
derive(P, [Y|Alpha], [Y|AlphaNew]) :-
    derive(P, Alpha, AlphaNew).
```

(b)
```
nullable(P, Alpha) :- nullableIgnoring(P, Alpha, []).

nullableIgnoring(_, [], _).
nullableIgnoring(P, [nonterm(A)|Beta], N) :-
    not(member(A,N)),
    member((nonterm(A),Alpha), P),
    nullableIgnoring(P,Alpha, [A|N]),
    nullableIgnoring(P,Beta,    N ).
```

freie Variablen

Aufgabe

Aufgabe 3 (Prolog, freie Variablen) [7 Punkte]

In Prolog können Formeln der Sprache[1]

$$\begin{array}{rcl} \textit{Formula} & := & \mathsf{p}\,(\textit{Term}) \mid \mathsf{q}\,(\textit{Term}, \textit{Term}) \\ & | & \forall\ \textit{Variable}.\ \textit{Formula} \\ & | & \exists\ \textit{Variable}.\ \textit{Formula} \end{array} \qquad \begin{array}{rcl} \textit{Integer} & := & \ldots \mid \text{-1} \mid 0 \mid 1 \mid \ldots \\ \textit{Variable} & := & \mathsf{x} \mid \mathsf{y} \mid \ldots \\ \textit{Term} & := & \textit{Variable} \mid \textit{Integer} \end{array}$$

als Terme `42, x, p(T), q(T1,T2), exists(X,T), forall(X,T), ...` dargestellt werden, z.B.

$$\forall\ \mathsf{x}.\ \exists\ \mathsf{y}.\ \mathsf{q}\,(\mathsf{x},\mathsf{y}) \quad \text{als} \quad \texttt{forall(x, exists(y, q(x,y)))}$$

Definieren Sie ein Prädikat `hasfree(BoundVars, Term)` das feststellt, ob ein solcher Term freie Variablen enthält! Da `forall` und `exists` Bindungskonstrukte sind, muss dieses Prädikat eine Liste der „außerhalb" gebundenen Variablen bekommen. **Beispiel:**

```
? hasfree([], forall(x,q(42,x))).
  false.

? hasfree([], forall(x,p(y))).
  true.
```

Hinweis: Verwenden Sie `atom(X)`.

Lösung

(a)
```
hasfree(BoundVars,X)  :- atom(X), not(member(X,BoundVars)).
hasfree(BoundVars,forall(X,T)) :- hasfree([X|BoundVars],T).
hasfree(BoundVars,exists(X,T)) :- hasfree([X|BoundVars],T).
hasfree(BoundVars,p(T1))    :- hasfree(BoundVars, T1).
hasfree(BoundVars,q(T1,_)) :- hasfree(BoundVars, T1).
hasfree(BoundVars,q(_,T2)) :- hasfree(BoundVars, T2).
```

Ratsel (generieren mit append)

Aufgabe

Ein beliebtes Buchstabenrätsel funktioniert wie folgt: Ein Start- und ein Zielwort werden vorgegeben. Das Startwort soll nun schrittweise zum Zielwort transformiert werden. Dabei gibt es 3 mögliche Arten von Schritten:

- Ein einzelner Buchstabe des momentanen Worts wird ersetzt.
- Ein einzelner Buchstabe des momentanen Worts wird entfernt.
- Ein einzelner Buchstabe wird dem momentanen Wort hinzugefügt.

Nach jedem Schritt muss wieder ein gültiges Wort entstehen.

Beispiel:
Rast → Rat → Rad → Rand

In Prolog lassen sich Buchstaben als Atome und Wörter als Listen von Atomen darstellen (z.B. "Rad" als `[r,a,d]`). Zudem seien folgende Prolog-Prädikate bereits vordefiniert: Der Generator `buchstabe(X)`, der bei Reerfüllung für `X` alle gültigen Buchstaben generiert, sowie der Tester `erlaubt(W)`, der testet, ob das Wort `W` gültig ist.

(a) Definieren Sie einen zweistelligen *Generator* [8 Punkte]

`schritt(Wort1, Wort2),`

welcher für ein gegebenes `Wort1` bei Reerfüllung alle Wörter `Wort2` generiert, die aus `Wort1` durch einen Schritt entstehen. Sie müssen hier noch nicht prüfen, ob das Wort gültig ist oder ob sich `Wort2` von `Wort1` unterscheidet.

(b) Das Prädikat `lösung(Woerter)` generiert alle Lösungen des Problems: [8 Punkte]

```
lösung(Woerter) :- start(S), ziel(Z), erreichbar(S,[S],Woerter,Z).
start([r,a,s,t]).
ziel([r,a,n,d]).
```

Definieren Sie das hierzu benötigte vierstellige Prädikat

`erreichbar(S,Besucht,Woerter,Z)`

welches für Start-Wort `S` und Ziel-Wort `Z` erfüllt ist, falls `Z` von `S` durch eine Folge von Zwischen-**`Woerter`**n erreichbar ist. Dabei dürfen nur **`erlaubt`**e Zwischenwörter entstehen. Um Endlosschleifen zu vermeiden, darf dabei weiterhin keine der in der Liste `Besucht` enhaltenen Wörter

nochmals vorkommen. Die Liste `Woerter` soll bei Erfüllung die einzelnen Zwischen-**`Woerter`** in richtiger Reihenfolge enthalten.

Hinweis: Verwenden Sie die Prädikate `member` und `not` aus der Vorlesung.

Lösung (ähnlich zu Wolf-Ziege-Kohl)

```
schritt(Wort1, Wort2) :-
        append(Xs, [_|Ys], Wort1),
        buchstabe(Y),
        append(Xs, [Y|Ys], Wort2).
schritt(Wort1, Wort2) :-
        append(Xs, [_|Ys], Wort1),
        append(Xs, Ys, Wort2).
schritt(Wort1, Wort2) :-
        append(Xs, Ys, Wort1),
        buchstabe(Y),
        append(Xs, [Y|Ys], Wort2).

erreichbar(S,_,[S],S).
erreichbar(S,Besucht,[S|Woerter],Z) :-
        schritt(S,W), erlaubt(W), not(member(W,Besucht)),
        erreichbar(W,[W|Besucht],Woerter,Z).
```

Erklärung

Schritt:

1. Replace eine bel. Buchstabe mit einer bel. Buchstabe
2. Delete eine bel. Buchstabe in bel.Position
3. Add eine bel. Buchstabe in bel.Position

Erreichbar:
Genau, wie bei Wolf, Ziege, Kohl.
generiere ein Schritt W, prüfe es mit erlaubt und not member.
Dann benutze W als neue S. Erweitere Ausgabe-Wörter mit S Besucht von nächster Rekursion mit W.

Ausführungsbaum

Weiter: Übungsblatt 7

```
even(0).
even(X) :-  odd(Y), X is Y+1, X>0.

odd(1).
odd(X)  :- even(Y), X is Y+1, X>1.
```

-> ?even(X) ergibt: