Michael Arndt
Fritz Hamm
Peter Schroeder-Heister

Sommersemester 2003
Übungsblatt 1
7.5.2003

Übungen zur Vorlesung *Logikprogrammierung*

Die Übungen basiert zu einem großen Teil auf einem Praktikum von Prof. R. Stärk an der Universität Freiburg/Schweiz.

PROPLOG — Aussagenlogisches PROLOG

PROPLOG ist eine sehr kleine Teilmenge von PROLOG.

Syntax von PROPOLOG:

PROGRAM	→	CLAUSE PROGRAM
PROGRAM	→	ε
CLAUSE	→	PROPSYMB.
CLAUSE	→	PROPSYMB :- PROPLIST
PROPLIST	→	PROPSYMB.
PROPLIST	→	PROPSYMB, PROPLIST
GOAL	→	?- PROPLIST
PROPSYMB	→	[a–z]([a–z,A–Z,0–9,_])*

Ein PROPLOG-Programm (auch Datenbank genannt) besteht aus einer endlichen Folge von Klauseln. Eine Klausel ist ein Faktum oder eine Regel. Ein *Faktum* hat die folgende Form:

$$A.$$

Die Bedeutung ist: A ist wahr. Eine *Regel* hat die folgende Form:

$$A \text{ :- } B_1, \ldots, B_n.$$

Die Bedeutung ist: Falls B_1 und ... und B_n wahr sind, so ist A wahr. A ist der Kopf der Klausel und $B_1, \ldots, B_n$ ist der Körper der Klausel. Man kann Fakten auffassen als Regeln mit leerem Körper. Ein *Ziel* hat die Form

$$?\text{- } B_1, \ldots, B_n.$$

Die Bedeutung ist: Sind B_1 und ... und B_n wahr?

Bsp. eines PROPLOG-Programms:

```
p   :-  q, r.
p   :-  q, s.
q.
s.
```

Der PROPLOG-Interpreter ermittelt, ob ein Ziel aus einem Programm folgt oder nicht.

Wie funktioniert ein PROPLOG Interpreter?

Wir beschreiben einen abstrakten Interpreter, der Zeichenketten und Listen von Zeichenketten manipulieren kann. Wir nehmen an, daß die Klauseln des Programms in einem Array

$$\text{clause}[1], \ldots, \text{clause}[\text{MAXCLAUSES}]$$

gespeichert sind und zwar in derselben Reihenfolge wie sie im Programm stehen. Weiter nehmen wir an, daß wir folgende Funktionen zur Verfügung haben.

first(L): gibt das erste Element einer Liste zurück, also $\text{first}([A_1, \ldots, A_n]) = A_1$.

rest(L): schneidet das erste Element einer Liste weg und gibt den Rest zurück, also:
$\text{rest}([A_1, \ldots, A_n]) = [A_2, \ldots, A_n]$.

head(C): gibt den Kopf einer Klausel zurück, also $\text{head}(A \text{ :- } B_1, \ldots, B_n) = A$.

body(C): gibt den Körper einer Klausel als Liste zurück, d.h. $\text{body}(A\text{:-}B_1, \ldots, B_n) = [B_1, \ldots, B_n]$ und $\text{body}(A) = []$.

concatenate(L_1, L_2): fügt zwei Listen aneinander und erzeugt dabei eine neue Kopie, d.h.
$\text{concatenate}([A_1, \ldots, A_m], [B_1, \ldots, B_n]) = [A_1, \ldots, A_m, B_1, \ldots, B_n]$.

Wir denken uns das Ziel ?- $B_1, \ldots, B_n$ als Liste der Form $[B_1, \ldots, B_n]$.

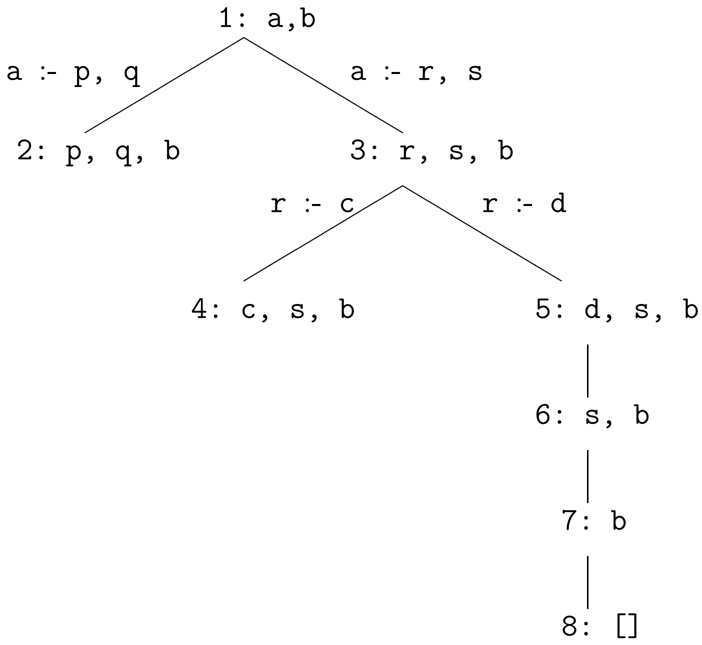
Die Funktion *provable* versucht ein Ziel durch eine *Top-Down-Evaluation* zu beweisen. Wenn es beim ersten Versuch nicht geht, beginnt sie mit *Back-Tracking.*

```
function provable(goal): boolean;
begin
  if goal = [] then return(true)
  else
    for i := 1 to MAXCLAUSES do
      if head(clause[i]) = first(goal) then
        if provable(concatenate(body(clause[i]), rest(goal))) then
          return(true);
        end
      end
    end;
    return(false)
  end
end.
```

Bsp. Gegeben sei das Ziel ?- `a, b` und das Programm

```
a  :-  p, q.
a  :-  r, s.
b.
d.
r  :-  c.
r  :-  d.
s.
```

Dann sieht der Suchbaum bei der Evaluation des Zieles folgendermaßen aus:



Der PROPLOG Interpreter durchläuft den Baum *depth-first* von links nach rechts. Dabei erhält er der Reihe nach die Subziele 1, . . . , 8.

Aufgabe 1 (1 Punkt)

Ein *vollständiger* Suchbaum umfaßt alle überhaupt durch das Programm erreichbaren Äste, d.h. er endet nicht mit dem ersten Ast, der im leeren Ziel endet. Schreiben Sie den vollständigen Suchbaum auf für das Ziel `p` und folgendes PROPLOG-Programm.

```
p :- q, fail.
p :- q, s.
q :- t, u.
s :- t.
t.
t :- v.
u.
v.
```

Aufgabe 2 (1 Punkt)

Was geschieht, wenn in einem Programm die Klausel `p :- p` steht und man die Frage `?- p` stellt?

Aufgabe 3 (1 Punkt)

Wie kann man die folgende Aussage als PROPLOG-Programm (um)schreiben?
„Falls `p` wahr ist und `q` oder `r` wahr ist, so ist `s` wahr.“

Aufgabe 4 (1 Punkt)

Geben Sie ein endliches PROPLOG-Programm P und ein Ziel G, so daß der vollständige Suchbaum von G unendlich viele verschiedene Pfade hat, die in das leere Ziel enden.