Übungsblatt 8

Abgabe der Lösungen: Tutorium in der Woche 24.06-28.06

Aufgabe 1 Listenunifikation (8 Punkte)

Verwenden Sie die in der Vorlesung eingeführte Kodierung von Listen durch die Operationen $[]/0$ (leere Liste) und $./2$ (Element vorn anhängen), um mittels des Algorithmus aus der Vorlesung für die folgenden Paare von Ausdrücken jeweils einen allgemeinsten Unifikator zu berechnen bzw. zu zeigen, dass sie nicht unifizierbar sind. Der Operator $./2$ ist die Präfixversion von $[\_|\_]/2$, d.h.

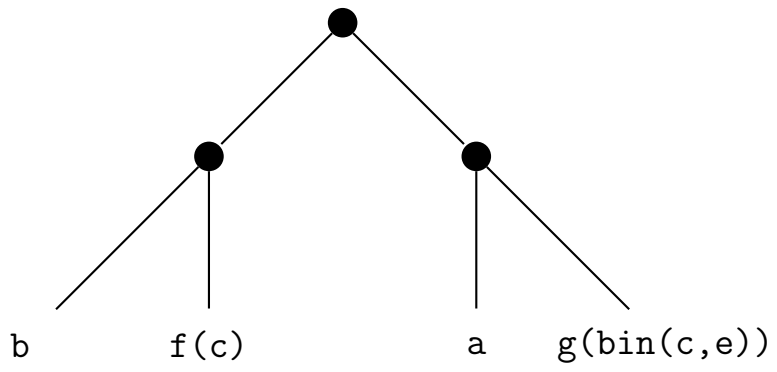
$$.(X, Y) = [X \mid Y]$$

1. $[X, b, Z]$ und $[a, Y]$,
2. $[[a, b], Y \mid Z]$ und $[X \mid X]$,
3. $[X \mid Y]$ und $[a, X \mid Y]$,
4. $[a, X, Y \mid []]$ und $[X, Y \mid Z]$,
5. $[X, [Y \mid Z] \mid Z]$ und $[[a, b \mid Z], X \mid Z]$.

Vergleichen Sie das Ergebnis mit der in Prolog eingebauten Unifikation. Welche Schlüsse kann man ziehen hinsichtlich des Occurs Check in Prolog?

Aufgabe 2 Präsenzaufgabe: Binäre Bäume (0 Punkte)

Binäre Baüme werden in Prolog einfach als Terme mit einer speziellen binären Operation—wir nehmen beispielsweise `bin/2`—präsentiert. Somit entspricht beispielsweise der folgende Baum dem Term `bin(bin(b,f(c)),bin(a,g(bin(c,e))))`.



Implementieren Sie folgende Operationen über Binäre Bäume in Prolog.

1. `mirror/2` — Spiegelung des Baums von links nach rechts.

2. `bin_to_list/3` und `bin_to_list/2` — Übersetzung des Baums (erstes Argument) in eine Liste (zweites Argument), die alle Blätter des Baums enthält. Das dritte Argument der ersten Funktion ist eine Liste, die an das Ergebnis am Ende angehängt werden soll (Akkumulator). Das zweite Prädikat darf das erste Prädikat verwenden (aber nicht umgekehrt).

3. `rem_first/3` — Löschen des erste Vorkommens des gegebenen Elements aus dem Baum.

4. `rem_all/3` — Löschen aller Vorkommen des gegebenen Elements aus dem Baum; wenn dadurch keine Elemente übrig bleiben, einfach dieses Element als Ergebnis ausgeben.

Beispiel des beabsichtigten Programmablaufs:

```
?- mirror(bin(bin(a,b),c), X).
X = bin(c, bin(b, a)) ;
false.

?- bin_to_list(bin(bin(a,b),c), X, [d]).
X = [a, b, c, d] .
false.

?- bin_to_list(bin(bin(a,b),c), X).
X = [a, b, c] ;
false.

?- rem_first(bin(bin(a,b),a), a, X).
X = bin(b, a) ;
false.

?- rem_all(bin(bin(a,b),a), a, X).
X = b ;
false.
?

?- rem_all(bin(bin(a,a),a), a, X).
X = a ;
X = a ;
X = a ;
X = a.
```

Aufgabe 3 Suche in Listen (6 Punkte)

Programmieren Sie ein Prädikat `occurs`/3, so dass die Anfrage ← `occurs(X,L,N)` als N eine Position in L (d.h. einen Index) liefert, an der das Element X vorkommt, in zwei Varianten: (a) es wird nur das erste Vorkommen von X in L zurückgegeben (natürlich nur dann, wenn X überhaupt in L vorkommt) und (b) es werden alle Vorkommen von X in L zurückgegeben.

Beispiel des beabsichtigten Programmablaufs:

```
?- occurs_version1(a,[a,b,a],X).
X = 1 ;
false.

?- occurs_version2(a,[a,b,a],X).
X = 1 ;
X = 3 ;
false.
```

Aufgabe 4 Listen umdrehen (6 Punkte)

Programmieren Sie in Prolog ein Prädikat `reverse/2`, das eine Liste umdreht, d.h. es soll gelten

$$\texttt{reverse}([a_1, \ldots, a_n], l) \iff l = [a_n, \ldots, a_1].$$

Es soll hierbei natürlich kein „eingebautes“ reverse-Prädikat verwendet werden, sondern Sie sollen die Funktion selbst programmieren. Sie können aber das Prädikat `append/3` aus der Vorlesung verwenden, das zwei Listen aneinanderhängt, d.h.

$$\texttt{append}([a_1, \ldots, a_n], [b_1, \ldots, b_k], l) \iff l = [a_1, \ldots, a_n, b_1, \ldots, b_k].$$

Programmieren Sie nun *ohne Verwendung von* `reverse` (und selbstverständlich auch ohne `reverse` noch einmal nachzuimplementieren, etwa per gesonderten Klauseln für $\texttt{reverse2}(k, [])$) eine Version `reverse2` von `reverse` mit *Akkumulator*, d.h. es soll gelten

$$\texttt{reverse2}([a_1, \ldots, a_n], [b_1, \ldots, b_k], l) \iff l = [a_n, \ldots, a_1, b_1, \ldots, b_k]$$

(man beachte die umgedrehte Reihenfolge der a_i). Wie erhält man eine Implementierung von `reverse` aus `reverse2`?