Übungsblatt 12 / Tutorium 05
Marc Gehring 358302 & Yannick Eschweiler 434446

```
% find the working directory with pwd.
% change the working directory in Prolog with working_directory(CWD,
'/Users/Marc/Desktop/RWTH/RWTH Courses/Programmierung/Blatt12').
% then use [blatt12].
% reload with called [blatt12]. again => important to stay lower case

% A3
% a)
userDefinedList(nil).
userDefinedList(cons(_, X)) :- userDefinedList(X).

% b)
% prologDefinedList([]).
% prologDefinedList([_|X]) :- prologDefinedList(X).

asPrologList([], nil).
asPrologList([A|X], cons(A, Y)) :- asPrologList(X, Y).

% asPrologList([1,2,3], cons(1, cons(2, cons(3, nil)))).

% c)

% reduce both X and Y until one equals 0 => Z is reduced simultaneously and gets increased
back once either X or Y equal 0
maximum(X, 0, X).
maximum(0, Y, Y).
maximum(s(X), s(Y), s(Z)) :- maximum(X, Y, Z).

% compare X to every element in XS with maximum => X should be at least as large as every
element in XS
maximumList([Y|[]], X) :- maximum(X, Y, X).
maximumList([Y|XS], X) :- maximum(X, Y, X), maximumList(XS, X).

/*
maximumList([s(0), s(s(0)), s(0), 0, s(s(s(0)))], X).
-> maximumList([s(0)|[s(s(0)), s(0), 0, s(s(s(0)))]], X).
-> maximum(X, s(0), X), maximumList([s(s(0))|[s(0), 0, s(s(s(0)))]], X).
-> maximum(X, s(0), X), maximum(X, s(s(0)), X), maximumList([s(0))|[0, s(s(s(0)))]], X).
-> maximum(X, s(0), X), maximum(X, s(s(0)), X), maximum(X, s(0), X),
maximumList([0|[s(s(s(0)))]], X).
-> maximum(X, s(0), X), maximum(X, s(s(0)), X), maximum(X, s(0), X), maximum(X, 0, X),
maximumList([s(s(s(0)))|[]], X).
-> maximum(X, s(0), X), maximum(X, s(s(0)), X), maximum(X, s(0), X), maximum(X, 0, X),
maximum(X, s(s(s(0)))), X).
```

```
-> X = s(s(s(0)))
*/

% d)

remove([X|XS], X, XS).
remove([Y|YS], X, [Y|XS]) :- remove(YS, X, XS).

/*
remove([s(0), s(s(0)), s(0), 0, s(s(s(0)))], s(0), XS).
-> remove([s(0), [s(s(0)), s(0), 0, s(s(s(0)))]], s(0), [s(s(0)), s(0), 0, s(s(s(0)))]]).
remove([s(s(0)), s(0), 0, s(s(s(0)))], s(0), XS).
-> remove([s(s(0)), [s(0), 0, s(s(s(0)))]], s(0), [s(s(0))|XS]).
-> remove([s(0), 0, s(s(s(0)))], s(0), XS).
-> remove([s(0), [0, s(s(s(0)))]], s(0), XS).
-> remove([s(0), [0, s(s(s(0)))]], s(0), [0, s(s(s(0)))]).
-> remove([s(0), [0, s(s(s(0)))]], s(0), [s(s(0)), 0, s(s(s(0)))]).
*/

% e)

% YS is XS in descending order
% append Y in front of YS => Y is the maximum of XS that gets removed from XS
sortList([X|[]], [X]).
sortList(XS, [Y|YS]) :- maximumList(XS, Y), remove(XS, Y, ZS), sortList(ZS, YS).

/*
sortList([s(0), s(s(0)), s(0), 0, s(s(s(0)))], YS).
-> sortList([s(0), s(s(0)), s(0), 0, s(s(s(0)))], [s(s(s(0)))|YS]).
-> sortList([s(0), s(s(0)), s(0), 0], [s(s(0))|YS]).
-> sortList([s(0), s(0), 0], [s(0)|YS]).
-> sortList([s(0), 0], [s(0)|YS]).
-> YS = [0]
*/

% f)

% base case
flattenConsecutive([[A|[]]|[]], [A]).
% skipping empty lists
flattenConsecutive([X, []|XS], Z) :- flattenConsecutive([X|XS], Z).
flattenConsecutive([[], X|XS], Z) :- flattenConsecutive([X|XS], Z).
% when the first list has at least 2 elements
flattenConsecutive([[A, A|X]|XS], Z) :- flattenConsecutive([[A|X]|XS], Z).
flattenConsecutive([[A, B|X]|XS], [A|Z]) :- flattenConsecutive([[B|X]|XS], Z).
% at the intersection of two consecutive lists => when the first list has one element left
flattenConsecutive([[A|[]], [A|X]|XS], Z) :- flattenConsecutive([[A|X]|XS], Z).
```

```
flattenConsecutive([[A|[]], [B|X]|XS], [A|Z]) :- flattenConsecutive([[B|X]|XS], Z).

% flattenConsecutive([[1, 1, 3, 2], [], [2, 4]], X).
% solution depends on the order of the predicates
% => the pattern of the second flattenConsecutive matches, as well as for some later
flattenConsecutives

% g)

% tree/2 => hat Konstante v/0 und Funktion children/n als Argumente => children/n ist
beispielsweise eine Liste von trees

% h)

flattenConsecutiveTree(tree(V, []), V).
flattenConsecutiveTree(tree(V1, [tree(V2, Children2)|Children1]), Res) :- append(Res1, Res2,
Res), flattenConsecutiveTree(tree(V2, Children2), Res1), flattenConsecutiveTree(tree(V1,
Children1), Res2).

/*
flattenConsecutiveTree(tree([1, 2, 3], [tree([], [tree([3], []), tree([3, 7], [])]), tree([], []),
tree([7, 15], [tree([7, 7], [])])]), X).
flattenConsecutiveTree(tree([1, 2, 3], [tree([], [tree([3], []), tree([3, 7], [])])|tree([], []),
tree([7, 15], [tree([7, 7], [])])], X)
V1 = [1, 2, 3]; V2 = []
-> 1.
-> flattenConsecutiveTree(tree([], [tree([3], []), tree([3, 7], [])]), Res1)
-> flattenConsecutiveTree(tree([], [tree([3], [])|[tree([3, 7], [])]]), Res1)
V1 = []; V2 = [3]; Children1 = [tree([3, 7], [])]; Children2 = []
-> 1.1
-> flattenConsecutiveTree(tree([3], [], Res1') => Res1' = [3]
-> 1.2
-> flattenConsecutiveTree(tree([], [tree([3, 7], [])]), Res2').
-> flattenConsecutiveTree(tree([], [tree([3, 7], [])|[]]), Res2').
V1 = []; V2 = [3, 7]; Children1 = []; Children2 = []
-> 1.2.1 => Res1'' = [3, 7], Res2'' = [] => Res'' = [3, 7] = Res2' => Res' = [3, 7] = Res1
-> ...
*/

append([], [], []).
append([X|[]], [], [X]).
append([], [Y|[]], [Y]).
append([X, X|XS], YS, Res) :- append([X|XS], YS, Res).
append([X1, X2|XS], YS, [X1|Res]) :- append([X2|XS], YS, Res).
append([X|[]], [X|YS], Res) :- append([], [X|YS], Res).
append([X|[]], [Y|YS], [X|Res]) :- append([], [Y|YS], Res).
append([], [Y, Y|YS], Res) :- append([], [Y|YS], Res).
```

```
append([], [Y1, Y2|YS], [Y1|Res]) :- append([], [Y2|YS], Res).

/*
append([1, 2, 2, 3], [3, 4, 4, 4], X).
-> append([1, 2|[2, 3]], [3, 4, 4, 4], [1|X]). -> append([2, 2, 3], [3, 4, 4, 4], X).
-> append([2, 2|[3]], [3, 4, 4, 4], X). -> append([2, 3], [3, 4, 4, 4], X).
-> append([2, 3|[]], [3, 4, 4, 4], [2|X]). -> append([3], [3, 4, 4, 4], X).
-> append([3|[]], [3|[4, 4, 4]], X). -> append([], [3, 4, 4, 4], X).
-> ...
*/
```

Aufgabe 5 (Unifikation): (5 · 4 = 20 Punkte)

In dieser Aufgabe sollen allgemeinste Unifikatoren bestimmt werden. Sie sollten diese Aufgabe ohne Hilfe eines Rechners lösen, da Sie zur Lösung von Aufgaben dieses Typs in der Klausur keinen Rechner zur Verfügung haben.
Nutzen Sie den Algorithmus zur Berechnung des allgemeinsten Unifikators (MGU) aus der Vorlesung, um die folgenden Termpaare auf Unifizierbarkeit zu testen.
Geben Sie neben dem Endergebnis σ auch die Unifikatoren $\sigma_1, \sigma_2, \ldots, \sigma_n$ für die direkten Teilterme der beiden Terme an. Sollte ein σ_i nicht existieren, so begründen Sie kurz, warum die Unifikation fehlschlägt. Geben Sie in diesem Fall an, ob es sich um einen *clash failure* oder einen *occur failure* handelt.

(i) **f(Z,Y,X)** und **f(g(a,a),g(X,Z),g(a,Y))**

(ii) **f(g(X),Z,a,W)** und **f(W,h(Y,Y),Y,g(b))**

(iii) **h(h(Z,a),h(Z,f(b)))** und **h(h(Z,Z),h(Z,f(Z)))**

(iv) **f(g(X,Y),g(Y,Y))** und **f(g(Y,a), g(a,X))**

(v) **f(g(Z,h(X)),X,g(Y,h(Y)))** und **f(g(h(Y),h(a)),X,g(h(X),h(a)))**

(i)
σ(f(Z, Y, X)) = σ(f(g(a, a), g(X, Z), g(a, Y)))
σ1 = MGU(Z, g(a,a)) = {Z = g(a, a)}
σ2 = MGU(σ1(Y), σ1(g(X, Z))) = MGU(Y, g(X, g(a, a))) = {Y = g(X, g(a, a))}
σ3 = MGU(σ2(σ1(X)), σ2(σ1(g(a, Y))) = MGU(X, g(a, g(X, g(a, a)))) = {X = g(a, g(X, g(a, a)))}
=> OCCUR FAILURE

(ii)
σ(f(f(X), Z, a, W)) = σ(f(W, h(Y, Y), Y, g(b)))
σ1 = MGU(f(X), W) = {W = f(X)}
σ2 = MGU(σ1(Z), σ1(h(Y, Y))) = MGU(Z, h(Y, Y)) = {Z = h(Y, Y)}
σ3 = MGU(σ2(σ1(a)), σ2(σ1(Y))) = MGU(a, Y) = {Y = a}
σ4 = MGU(σ3(σ2(σ1(W))), σ3(σ2(σ1(g(b))))), MGU(f(X), g(b)) = {f(X) = g(b)}
=> σ = {f(x) = g(b), Y = a, Z = h(a, a), W = g(b)}

(iii)
σ(h(h(Z, a), h(Z, f(b)))) = σ(h(h(Z, Z), h(Z, f(Z))))
σ1 = MGU(h(Z, a), h(Z, Z)) = {Z = a}
σ2 = MGU(σ1(h(Z, f(b))), σ1(h(Z, f(Z)))) = MGU(f(b), f(a)) = {a = b}
=> σ = {a = b, Z = b}

(iv)
σ(f(g(X, Y), g(Y, Y))) = σ(f(g(Y, a), g(a, X)))
σ1 = MGU(g(X, Y), g(Y, a)) = {X = Y, Y = a}
σ2 = MGU(σ1(g(Y, Y)), σ1(g(a, X))) = MGU(g(a, a), g(a, a)) = {}
=> σ = {X = Y, Y = a}

(v)
σ(f(g(Z, h(X)), X, g(Y, h(Y)))) = σ(f(g(h(Y), h(a)), X, g(h(X), h(a))))
σ1 = MGU(g(Z, h(X)), g(h(Y), h(a))) = {Z = h(Y), h(X) = h(a)} = {Z = h(Y), X = a}
σ2 = MGU(σ1(X), σ1(X)) = {}
σ3 = MGU(σ1(g(Y, h(Y))), σ1(g(h(X), h(a)))) = MGU(g(Y, h(Y)), g(h(a), h(a)) = {Y = a, Y = h(a)}
=> OCCUR FAILURE

Aufgabe 7 (Beweisbäume): (18 + 2 = 20 Punkte)

Betrachten Sie die Anfrage `?- a(Z,s(0)).` zu folgendem Prolog-Programm:

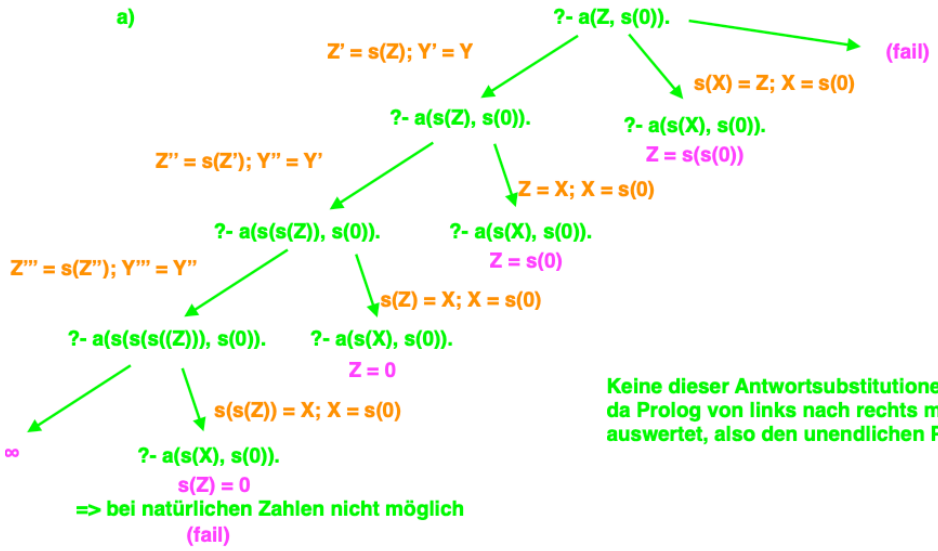
```
a(X, Y) :- a(s(X), Y).
a(0, Y) :- b(s(Y), Y).
a(s(X), X).
b(X, X) :- a(s(X), X).
```

a) Geben Sie den zugehörigen Beweisbaum (SLD-Baum) bis einschließlich Höhe 3 an. Die Höhe eines Baums ist der längste Pfad von der Wurzel bis zu einem Blatt. Ein Baum, welcher nur aus einem Blatt besteht, hat also die Höhe 0. Markieren Sie unendliche Pfade mit ∞ und Fehlschläge mit $(fail)$. Geben Sie alle Antwortsubstitutionen zur obigen Anfrage an und geben Sie an, welche dieser Antwortsubstitutionen von Prolog gefunden werden.

b) Strukturieren Sie das gegebene Programm so in ein logisch äquivalentes Programm um, dass Prolog mit seiner Auswertungsstrategie **mindestens eine** Lösung zur gegebenen Anfrage findet. Der Beweisbaum (SLD-Baum) muss nicht endlich sein! Sie brauchen den SLD Baum nicht angeben.

Hinweis: Bei dieser Umstrukturierung dürfen Sie nur die Reihenfolge der Prolog-Klauseln verändern.

a)



Keine dieser Antwortsubstitutionen wird gefunden, da Prolog von links nach rechts mit depth-first search auswertet, also den unendlichen Pfad nimmt.

b)

```
a(s(X), X).
a(X, Y) :- a(s(X), Y).
a(0, Y) :- b(s(Y), Y).
b(X, X) :- a(s(X), X).
```