

Tutoraufgabe 1 (Überblickswissen):

- Listen werden in Prolog intern mit der leeren Liste `[]` und dem Listenkonstruktor `.` repräsentiert. Inwiefern ist die Notation mit dem Listenkonstruktor `cons` und der leeren Liste `nil` dazu analog? Weil diese beiden Schreibweisen nicht sonderlich praktisch sind, schreibt man Listen häufig in der Schreibweise `[Kopf|Rest]`. Wie hängt diese Listenschreibweise mit den anderen zusammen?
- Sowohl Prädikate als auch Funktionen haben in Prolog Bezeichner, die mit einem Kleinbuchstaben beginnen. Obwohl sie also leicht zu verwechseln sind, haben sie ganz unterschiedliche Bedeutungen. Wozu dienen Funktionen? Was sind im Gegensatz dazu Prädikate?
- Was ist der Unterschied zwischen Unifikation und Pattern Matching?
- Spielt es eine Rolle, in welcher Reihenfolge die Klauseln eines Prolog-Programms stehen?
- Was sind Anwendungsgebiete der Logikprogrammierung?

Lösung: _____

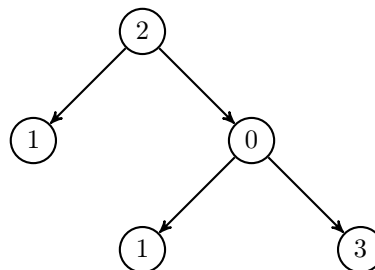
- Wenn man die Liste der Zahlen 1 bis 3 mit der Prolog-internen Schreibweise darstellen möchte, sieht das so aus: `.(1,.(2,.(3,[])))`. Statt des Konstruktors `.` wird gerne auch `cons` geschrieben und statt der leeren Liste `[]` gerne auch `nil`. Dann sähe die Liste der Zahlen 1 bis 3 so aus: `cons(1,cons(2,cons(3,nil)))`. Obwohl diese beiden Listen nicht syntaktisch gleich sind, weil sie andere Datenkonstruktoren nutzen, ist doch die Art und Weise der Listenbildung analog. Außerdem gibt es noch die Listenschreibweise, in der alle Elemente hintereinander angegeben werden: `[1,2,3]`. Man kann hierbei zu Beginn beliebig viele Elemente einzeln aufzählen (den Kopf), bevor man die Restliste angibt. Es gilt: `[1,2,3] = [1|[2,3]] = [1,2|[3]] = [1,2,3|[]]`. In der Praxis wird meist nur die zuletzt erläuterte Form verwendet.
- Funktionen werden in Prolog genutzt, um Datenstrukturen zu realisieren. Funktionen in Prolog sind vom mathematischen Begriff der Funktion abzugrenzen: Während wir bei letzteren gewohnt sind, dass sich bei Angabe konkreter Werte für die Argumente ein Funktionswert ergibt, findet in Prolog *keine* Auswertung von Funktionen statt. Wir geben die Argumente hier an, weil sich so verschiedene Terme mit demselben Funktionssymbol unterscheiden lassen. Für Konstanten, also Funktionen von Stelligkeit 0, gilt das natürlich nicht, da diese keine Argumente entgegennehmen und so immer denselben Term ergeben. Die Funktionen in Prolog entsprechen also den Datenkonstruktoren in Haskell.
Dem gegenüber stehen die Prädikate. Diese werden ausgewertet, daher geben wir für Prädikate beim Programmieren auch in Klauseln an, wie diese weiter verarbeitet werden können. Natürlich können dabei auch Funktionen eine Rolle spielen, aber diese werden immer nur im Zusammenspiel mit Prädikaten ausgewertet, niemals alleine. Ein Prädikat mit konkreten Argumenten hat also immer einen Wahrheitswert, eine Funktion nicht. Wenn wir eine mathematische Funktion f der Stelligkeit n in Prolog nachbilden wollen, benötigen wir dazu ein Prädikat p der Stelligkeit $n + 1$: Die ersten n Argumente sind die Argumente der nachzubildenden Funktion, das letzte Argument ist der Funktionswert. Es gilt dann $p(a_1, \dots, a_n, b)$ gdw. $f(a_1, \dots, a_n) = b$.
- Einfach ausgedrückt funktioniert Pattern Matching nur in eine Richtung, während Unifikation bidirektional funktioniert. Beim Matching werden nur die Variablen in den Patterns instanziiert, um 2 Ausdrücke gleich zu machen (s matcht t , falls es eine Substitution σ gibt, mit $\sigma(s) = t$). Bei der Unifikation werden die Variablen in beiden Ausdrücken instanziiert (s und t sind unifizierbar, falls es eine Substitution σ gibt mit $\sigma(s) = \sigma(t)$).
- Ja, es spielt eine Rolle. Während in einer "reinen" Logiksprache die Reihenfolge keinen Unterschied bei den gefundenen Lösungen macht, kann bei Prolog die Reihenfolge der Klauseln die Terminierung des Programms beeinflussen und damit auch dafür sorgen, dass spätere Lösungen nicht mehr gefunden werden.

- e) Es gibt viele Anwendungsbereiche, wobei Anwendungen in der künstlichen Intelligenz (insbesondere sogenannte Expertensysteme und Robotik) und dem Arbeiten mit Datenbanken hervorstechen.

Tutoraufgabe 2 (BinTree):

Natürliche Zahlen lassen sich in Prolog (oder anderen deklarativen Sprachen) durch die Peano-Notation als Terme darstellen. Dabei stellt die Konstante 0 die Zahl 0 dar und für eine Zahl n dargestellt durch den Term N stellt $s(N)$ die Zahl $n + 1$ dar. So wird z. B. die Zahl 3 durch den Term $s(s(s(0)))$ dargestellt.

Binäre Bäume können in Prolog folgendermaßen als Terme dargestellt werden. Sei n eine natürliche Zahl dargestellt durch den Term N . Dann repräsentiert der Term $\text{leaf}(N)$ einen Baum mit nur einem Blatt, welches den Wert n enthält. Für zwei Bäume x und y dargestellt durch die Terme X und Y repräsentiert der Term $\text{node}(X,N,Y)$ einen binären Baum mit einem Wurzelknoten, der den Wert n enthält und die Teilbäume x und y hat. Als Beispiel ist nachfolgend ein binärer Baum und seine Darstellung als Term angegeben.



`node(leaf(s(0)),s(s(0)),node(leaf(s(0)),0,leaf(s(s(s(0)))))`

- a) Schreiben Sie ein Prädikat `increment/2` in Prolog, wobei `increment(B,IncB)` genau dann wahr sein soll, wenn sich der Baum `IncB` aus dem Baum `B` ergibt, indem jede Zahl, die in einem inneren Knoten oder Blatt steht, um eins erhöht wird. Beispielsweise soll der Aufruf

`increment(node(leaf(s(0)),s(s(0)),leaf(0)),Res)`

das Ergebnis `Res = node(leaf(s(s(0))),s(s(s(0))),leaf(s(0)))` liefern.

- b) Schreiben Sie ein Prädikat `append(XS,YS,Res)`, das die Listen `XS` und `YS` hintereinanderhängt. Ein Aufruf von `append([a,b,c],[d,e],Res)` würde das Ergebnis `Res = [a,b,c,d,e]` liefern.
- c) Es gibt verschiedene Verfahren, um einen Baum systematisch zu durchsuchen. Man unterscheidet zwischen Pre-, In- und Postorder-Traversierung. Bei einer Preorder-Traversierung wird zuerst die Wurzel (W) eines Baums durchsucht, dann der linke Teilbaum (L) und anschließend der rechte Teilbaum (R). Bei Inorder ist die Reihenfolge LWR und bei Postorder LRW. Für den Beispielbaum aus der Abbildung ergeben sich folgende Ausgaben:

- Preorder: 2, 1, 0, 1, 3
- Postorder: 1, 1, 3, 0, 2
- Inorder: 1, 2, 1, 0, 3

Sie sollen nun ein Prädikat `inorder(B,Res)` schreiben, das alle Knoten eines Baumes `B` in **Inorder**-Reihenfolge in die Liste `Res` einträgt. Wenn `B` der Baum aus der Abbildung ist, so würde `inorder(B,Res)` das Ergebnis `Res = [s(0),s(s(0)),s(0),0,s(s(s(0)))]` liefern.

Hinweise:

- Sie dürfen die Funktion `append` aus Teilaufgabe b) verwenden.

Lösung: _____

```
% Teilaufgabe a)
increment(leaf(X),leaf(s(X))).
increment(node(L, N, R), node(IncL, s(N), IncR)) :- increment(L, IncL),
                                                    increment(R, IncR).

% Teilaufgabe b)
append([], YS, YS).
append([X|XS], YS, [X|Res]) :- append(XS, YS, Res).

% Teilaufgabe c)
inorder(leaf(X),[X]).
inorder(node(L, N, R), Res) :-
    inorder(L,ResL),
    inorder(R,ResR),
    append(ResL, [N|ResR], Res).
```

Aufgabe 3 (Listen): (3 + 3 + 6 + 3 + 6 + 15 + 4 + 10 = 50 Punkte)

Verwenden Sie in dieser Aufgabe **keine** vordefinierten Prädikate. Nutzen Sie Prädikate, deren Implementierung in früheren Teilaufgaben gefordert wurde, falls dies sinnvoll ist.

- a) Eine Möglichkeit, Listen in Prolog darzustellen, ist die Verwendung eines nullstelligen Funktionssymbols `nil` zur Repräsentation der leeren Liste und eines zweistelligen Funktionssymbols `cons` zur Repräsentation nicht-leerer Listen, wobei das erste Argument von `cons` der in dem aktuellen Listenelement gespeicherte Wert und das zweite Argument von `cons` die Restliste ist. Auf diese Art und Weise kann z.B. die Liste `[1,2,3]` durch den Term `cons(1, cons(2, cons(3, nil)))` dargestellt werden, vgl. Aufgabe 1a).

Implementieren Sie ein Prädikat `userDefinedList(X)`, das genau dann wahr ist, wenn `X` eine derartige mit Hilfe der Funktionssymbole `nil` und `cons` beschriebene Liste ist.

- b) Implementieren Sie ein Prädikat `asPrologList(X,Y)`, das genau dann wahr ist, wenn
- `X` eine Liste ist, die die vordefinierte Prolog-Schreibweise für Listen nutzt und
 - `Y` eine mit Hilfe der Funktionssymbole `nil` und `cons` (siehe vorheriger Aufgabenteil) beschriebene Liste ist und die gleiche Liste wie `X` beschreibt.

Es soll also beispielsweise `asPrologList([1,2,3], cons(1, cons(2, cons(3, nil))))` gelten.

- c) Sie haben bereits die Darstellung der natürlichen Zahlen mittels der Repräsentation `0`, `s(0)`, `s(s(0))`, ... kennengelernt. Schreiben Sie ein dreistelliges Prädikat `maximum(X,Y,Z)`, das auf solchen Zahlen operiert, sodass das Prädikat genau dann erfüllt ist, wenn `Z` das Maximum von `X` und `Y` ist. Beispielsweise gilt also `maximum(s(0),0,s(0))`. Schreiben Sie ein weiteres Prädikat `maximumList(XS,X)`, welches genau dann erfüllt ist, wenn `X` das Maximum der Liste `XS` ist. Beispielsweise gilt `maximumList([s(0),s(s(0)),s(0),0,s(s(s(0)))], s(s(s(0))))`.
- d) In dieser Teilaufgabe sollen Sie ein Prädikat `remove(XS, X, YS)` schreiben. Falls `X` in der Liste `XS` mindestens einmal vorkommt, soll `remove(XS, X, YS)` genau dann erfüllt sein, wenn `YS` aus der Liste `XS` entsteht, indem ein Vorkommen von `X` entfernt wird. Kommt `X` nicht in `XS` vor, dann kann sich Ihr Prädikat beliebig verhalten. Zum Beispiel gilt `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)))])` und `remove([s(0),s(s(0)),s(0),0,s(s(s(0)))], s(0), [s(0), s(s(0)),0,s(s(s(0)))])`.
- e) Schreiben Sie nun ein zweistelliges Prädikat `sortList(XS,YS)`, welches genau dann erfüllt ist, wenn `YS` die Liste ist, welche aus den Elementen der Liste `XS` besteht und die Elemente dieser Liste absteigend nach Größe sortiert sind. Verwenden Sie hierzu das Prädikat `maximumList`, um zunächst das Maximum von `XS` zu bestimmen und `remove`, um das Maximum aus `XS` zu löschen. Sortieren Sie dann zunächst die dadurch entstehende Liste und fügen Sie schließlich das Maximum wieder vorne

ein. Es würde zum Beispiel `sortList([s(0),s(s(0)),s(0),0,s(s(s(0)))], Res)` das Ergebnis `Res = [s(s(s(0))),s(s(0)),s(0),s(0),0]` liefern.

f) Implementieren Sie ein Prädikat `flattenConsecutive(X,Y)`, das genau dann wahr ist, wenn

- `X` eine Liste von Listen ist und
- `Y` eine Liste ist, die entsteht, wenn man alle Elemente von `X` konkateniert. Hierbei dürfen mehrere gleiche, aufeinander folgende Elemente entfernt werden, sodass mindestens eines dieser gleichen Elemente überbleibt. Die Reihenfolge der Elemente der Liste darf nicht weiter verändert werden.

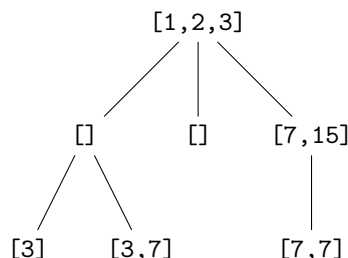
Es gilt also beispielsweise `flattenConsecutive([1,1,3,2],[],[2,4]),[1,3,2,4])`, `flattenConsecutive([1,1,3,2],[],[2,4]),[1,1,3,2,4])` und `flattenConsecutive([1,1,3,2],[],[2,4]),[1,1,3,2,2,4])`.

g) Entwerfen Sie eine Datenstruktur, mit deren Hilfe sich Mehrwegbäume in Prolog darstellen lassen. Ein Mehrwegbaum ist ein Baum, dessen Knoten *beliebig viele Kindknoten* haben können. Darüber hinaus muss jeder Knoten einen beliebigen Wert speichern können. Beschreiben Sie kurz die Bedeutung der von Ihnen zu diesem Zweck verwendeten Funktionssymbole und ihrer Argumente.

h) Implementieren Sie ein Prädikat `flattenConsecutiveTree(X,Y)`, das genau dann wahr ist, wenn

- `X` ein Baum von Listen ist, wobei die von Ihnen im letzten Aufgabenteil entworfene Datenstruktur zur Repräsentation von Bäumen und die vordefinierten Prolog-Listen verwendet werden, und
- `Y` eine Liste ist, die entsteht, indem alle in `X` enthaltenen Listen konkateniert werden. Hierbei dürfen mehrere gleiche, aufeinander folgende Elemente entfernt werden, sodass mindestens eines dieser Elemente überbleibt.

Die Reihenfolge, in der die Listen aus `X` konkateniert werden sollen, ist eine Preorder-Traversierung, vgl. Aufgabe 2c): Am Anfang steht jene Liste, die in der Wurzel von `X` gespeichert ist. Es folgen alle Listen, die in dem durch den ersten Kindknoten definierten Teilbaum gespeichert sind, gefolgt von allen Listen, die in dem durch den zweiten Kindknoten definierten Teilbaum gespeichert sind, usw. Es gilt also beispielsweise `flattenConsecutiveTree(X, [1,2,3,7,15,7])`, wenn `X` der folgende Mehrwegbaum ist:



Hinweise:

- Verwenden Sie das Prädikat `append` aus der Tutoraufgabe 2b).

Lösung: _____

```
% a)
userDefinedList(nil).
userDefinedList(cons(_,XS)) :- userDefinedList(XS).
```

```
% b)
asPrologList([], nil).
asPrologList([X|XS], cons(X,YS)) :- asPrologList(XS, YS).
```

```
% c)
maximum(0, X, X).
maximum(X, 0, X).
```

```

maximum(s(X),s(Y), s(Z)) :- maximum(X,Y,Z).

maximumList([], 0).
maximumList([X|XS], Y) :- maximumList(XS, Z), maximum(X,Z,Y).

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

% e)
sortList([], []).
sortList(XS, [Y|YS]) :- maximumList(XS, Y), remove(XS, Y, ZS), sortList(ZS, YS).

% f)
removeConsecutive([], []).
removeConsecutive([X], [X]).
removeConsecutive([X,X|XS], [X|YS]) :- removeConsecutive([X|XS], [X|YS]).
removeConsecutive([X,Y|XS], [X,Y|YS]) :- removeConsecutive([Y|XS], [Y|YS]).

flatten([], []).
flatten([[]|XS], YS) :- flatten(XS, YS).
flatten([X|XS] | XSS, [X|YS]) :- flatten([XS|XSS], YS).

flattenConsecutive(XS,YS) :- flatten(XS,ZS), removeConsecutive(ZS, YS).

% g)
% node(X,Children)
% Das Funktionssymbol node dient zur Repraesentation eines Knotens,
% wobei X den gespeicherten Wert und Children die Liste der Kindknoten
% darstellt.

% h)
%X = node([1,2,3],[node([], [node([3],[]), node([3,7],[])]), node([],[]),
% node([7,15],[node([7,7],[])])])
append([],YS,YS).
append([X|XS],YS,[X|ZS]) :- append(XS,YS,ZS).

treeToList(node(X,[]), [X]).
treeToList(node(X,[Child|Children]), [X|XS]) :-
    treeToList(Child,YS), treeToList(node(X,Children), [X|ZS]), append(YS,ZS,XS).

flattenConsecutiveTree(X,Y) :- treeToList(X,Z), flattenConsecutive(Z,Y).

```

Tutoraufgabe 4 (Unifikation):

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 auch 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 $\sigma_1, \sigma_2, \dots, \sigma_n$ 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) $f(X, X, g(X, X))$ und $f(Y, a, g(b, Y))$

- (ii) $f(X, g(a, Y))$ und $f(h(Y), g(Z, h(Z)))$
- (iii) $f(g(X, a), X)$ und $f(g(Y, X), g(Y, X))$
- (iv) $f(W, g(Y), Y, W)$ und $f(g(X), X, g(Z), Z)$
- (v) $f(g(X, Z), Z, g(Z, X))$ und $f(Y, a, g(Z, g(W, X)))$

Beispiel:

Für $f(A, g(c), h(Y, Y))$ und $f(c, X, h(A, X))$ ist folgende Lösung anzugeben:

$$\sigma_1 = \{A = c\}$$

$$\sigma_2 = \{X = g(c)\}$$

$\sigma_3 = \text{mgu}(h(Y, Y), h(c, g(c)))$ existiert nicht, da c und $g(c)$ nicht mit dem gleichen Symbol beginnen. Folglich liegt ein *clash failure* vor.

Für $f(A, g(c), h(Y, g(c)))$ und $f(c, X, h(A, X))$ ist folgende Lösung anzugeben:

$$\sigma_1 = \{A = c\}$$

$$\sigma_2 = \{X = g(c)\}$$

$$\sigma_3 = \{Y = c\}$$

$$\sigma = \sigma_3 \circ \sigma_2 \circ \sigma_1 = \{A = c, X = g(c), Y = c\}$$

Lösung: _____

- (i) $\sigma_1 = \{X = Y\}$
 $\sigma_2 = \{Y = a\}$
 $\sigma_3 = \text{mgu}(g(a, a), g(b, a))$ existiert nicht. *clash failure*.
- (ii) $\sigma_1 = \{X = h(Y)\}$
 $\sigma_2 = \{Z = a, Y = h(a)\}$
 $\sigma = \{X = h(h(a)), Z = a, Y = h(a)\}$
- (iii) $\sigma_1 = \{X = a, Y = a\}$
 $\sigma_2 = \text{mgu}(a, g(a, a))$ existiert nicht. *clash failure*.
- (iv) $\sigma_1 = \{W = g(X)\}$
 $\sigma_2 = \{X = g(Y)\}$
 $\sigma_3 = \{Y = g(Z)\}$
 $\sigma_4 = \text{mgu}(g(g(g(Z))), Z)$ existiert nicht. *occur failure*.
- (v) $\sigma_1 = \{Y = g(X, Z)\}$
 $\sigma_2 = \{Z = a\}$
 $\sigma_3 = \text{mgu}(g(a, X), g(a, g(W, X)))$ existiert nicht. *occur failure*.

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 Termopaare auf Unifizierbarkeit zu testen.

Geben Sie neben dem Endergebnis σ auch die Unifikatoren $\sigma_1, \sigma_2, \dots, \sigma_n$ 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 $f(g(h(Y), h(a)), X, g(h(X), h(a)))$

Lösung: _____

- (i) $\sigma_1 = \{Z = g(a, a)\}$
 $\sigma_2 = \{Y = g(X, g(a, a))\}$
 $\sigma_3 = \text{mgu}(X, g(a, g(X, g(a, a))))$ existiert nicht. *occur failure*.
- (ii) $\sigma_1 = \{W = g(X)\}$
 $\sigma_2 = \{Z = h(Y, Y)\}$
 $\sigma_3 = \{Y = a\}$
 $\sigma_4 = \{X = b\}$
 $\sigma = \{X = b, Y = a, Z = h(a, a), W = g(b)\}$.
- (iii) $\sigma_1 = \{Z = a\}$
 $\sigma_2 = \text{mgu}(h(a, f(b)), h(a, f(a)))$ existiert nicht. *clash failure*.
- (iv) $\sigma_1 = \{X = a, Y = a\}$
 $\sigma_2 = \emptyset$ (Identitätsabbildung)
 $\sigma = \{X = a, Y = a\}$
- (v) $\sigma_1 = \{X = a, Z = h(Y)\}$
 $\sigma_2 = \emptyset$ (Identitätsabbildung)
 $\sigma_3 = \text{mgu}(g(Y, h(Y)), g(h(a), h(a)))$ existiert nicht. *clash failure*.

Tutoraufgabe 6 (Beweisbäume):

Betrachten Sie die Anfrage $?- t(c, Z)$ auf folgendem Prolog-Programm:

```

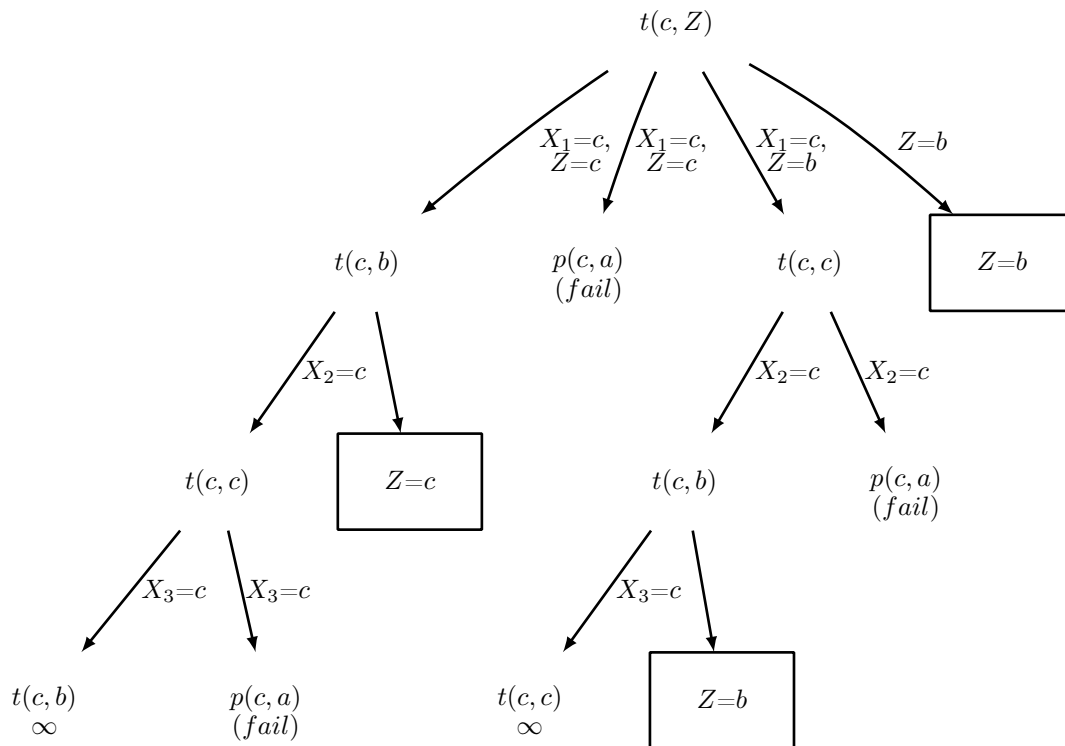
t(X, c) :- t(X, b).
t(X, X) :- p(X, a).
t(X, b) :- t(c, X).
t(c, b).

```

- 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 in ein logisch äquivalentes Programm um, sodass Prolog mit seiner Auswertungsstrategie **mindestens zwei** Lösungen zur gegebenen Anfrage findet. Der Beweisbaum (SLD-Baum) muss nicht endlich sein! Bei dieser Umstrukturierung dürfen Sie nur die Reihenfolge der Prolog-Klauseln verändern.

Lösung: _____

a)



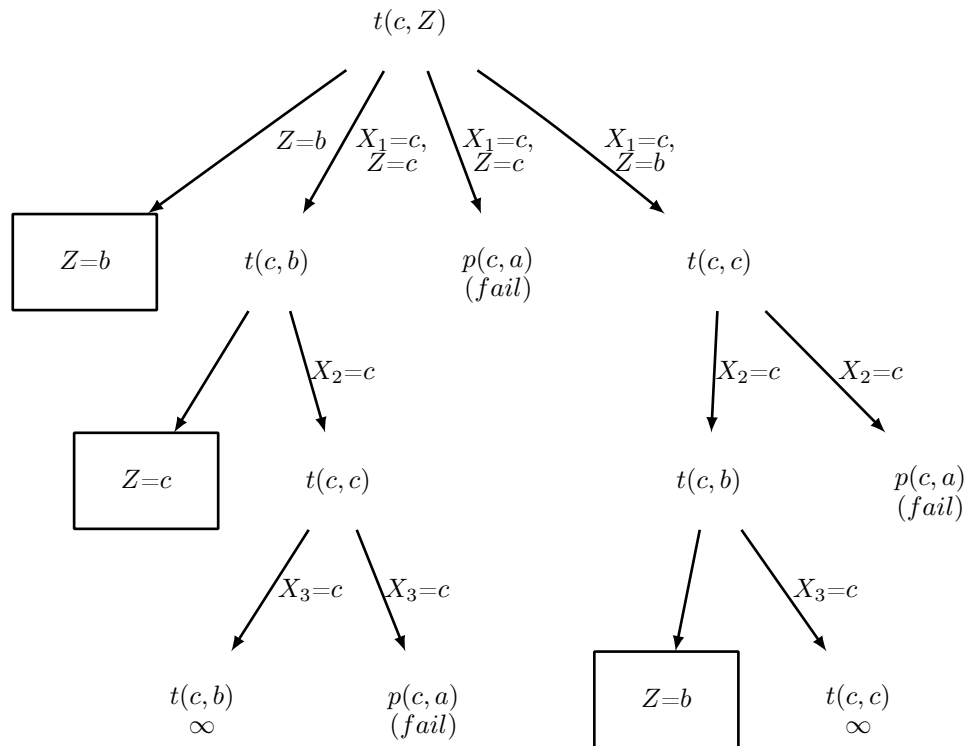
Es gibt (in diesem Teilbaum) drei Antwortsubstitutionen: Z/c und zweimal Z/b . Allerdings werden diese alle nicht erreicht, da der am weitesten links stehende Ast unendlich ist.

- b) Das Programm sollte wie folgt umstrukturiert werden:

```

t(c, b).
t(X, c) :- t(X, b).
t(X, X) :- p(X, a).
t(X, b) :- t(c, X).
  
```

Es ergibt sich dann der folgende unendliche Beweisbaum für die Anfrage $?- t(c, Z) :$



Der Vorteil gegenüber dem ursprünglichen Programm ist hier, dass der am weitesten links stehende Ast zu einer Lösung führt.

Aufgabe 7 (Beweisbäume):

(18 + 2 = 20 Punkte)

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

```

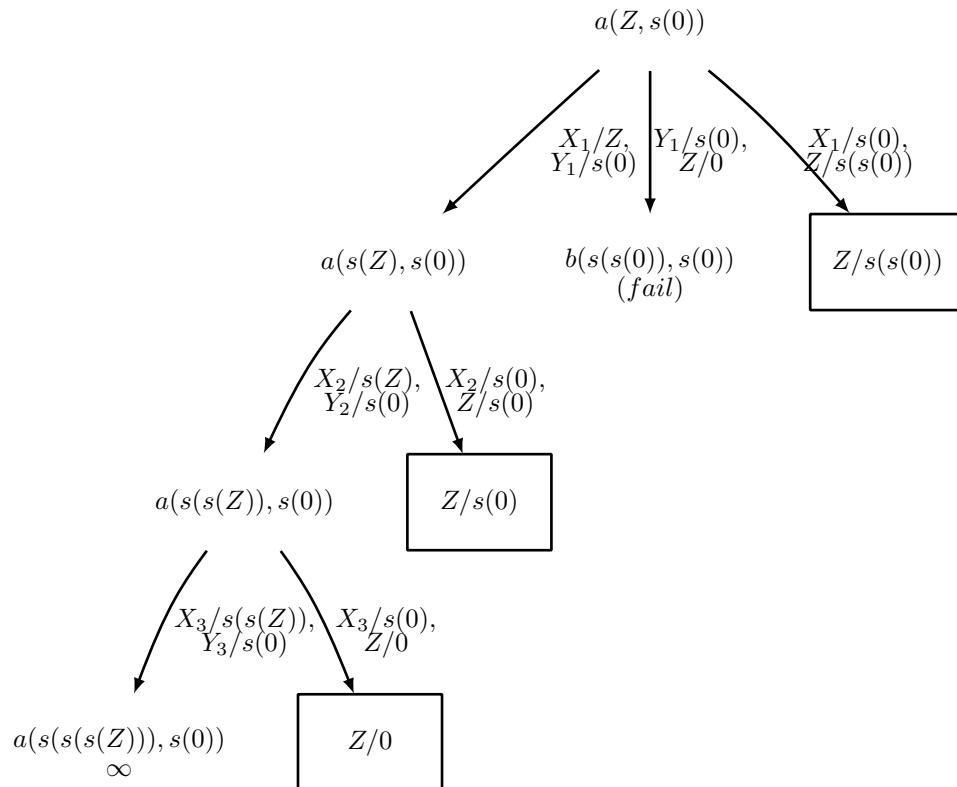
a(X, Y) :- a(s(X), Y).
a(0, Y) :- b(s(Y), Y).
a(s(X), X).
b(X, X) :- a(s(X), X).
  
```

- 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.
- 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.

Lösung: _____

a)



Die Belegungen für die Variablen X_i und Y_j aus den Programmklauseln müssen nicht mit angegeben werden.

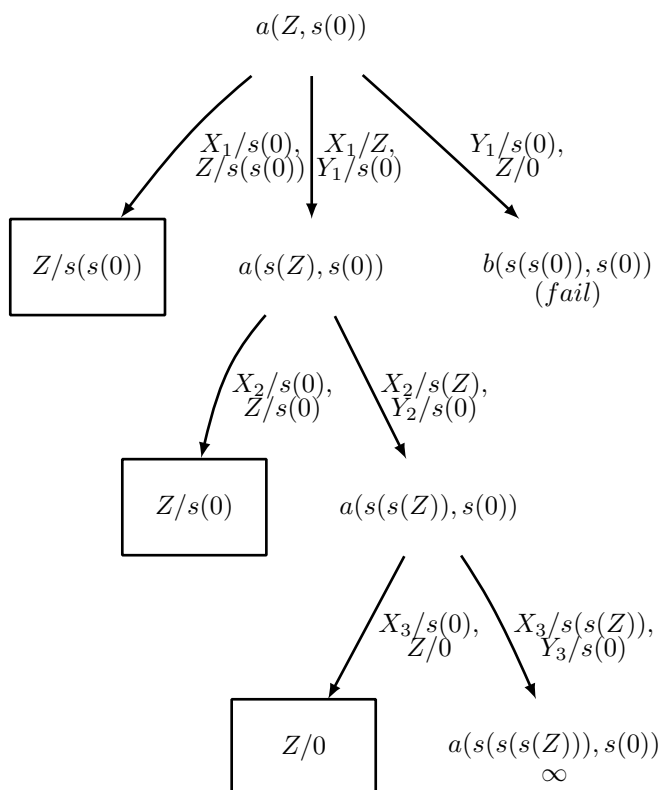
Es gibt drei Antwortsubstitutionen: $Z/0$, $Z/s(0)$ und $Z/s(s(0))$. Allerdings werden diese alle nicht erreicht, da der am weitesten links stehende Ast unendlich ist.

b) Das Program kann wie folgt umstrukturiert werden:

```

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).
  
```

Es ergibt sich dann der folgende, unendliche Beweisbaum für die Anfrage $?- a(Z, s(0))$:



Der Vorteil gegenüber dem ursprünglichen Programm ist hier, dass Prolog die Antworten findet, bevor es in den nichtterminierenden Ast der Berechnung läuft.

Tutoraufgabe 8 (Arithmetik mit Prolog):

Formulieren Sie ein Prolog-Programm mit einem Prädikat `squares(N, R)`, das wahr ist, wenn $N \geq 1$ gilt und `R` die absteigende Liste der Quadratzahlen von N^2 bis 1 ist. Beispielsweise soll `squares(5, [25, 16, 9, 4, 1])` wahr sein. Verwenden Sie dafür die Gleichung $k^2 = (k-1)^2 + 2*(k-1) + 1$ und nicht direktes Quadrieren. Benutzen Sie das vordefinierte Prädikat `is/2`.

Lösung: _____

```

squares(1, [1]).
squares(N, [NN,T2|R]) :- N > 1,
                        T is N - 1,
                        squares(T, [T2|R]),
                        NN is T2 + 2*T + 1.
  
```

Aufgabe 9 (Arithmetik mit Prolog):

(10 Punkte)

Formulieren Sie ein Prolog-Programm mit einem Prädikat `factors(X,Y)`, das wahr ist, wenn $X \geq 1$ und `Y` die aufsteigende Liste der positiven Teiler von `X` ist. Beispielsweise soll `factors(6, [1,2,3,6])` wahr sein. Benutzen Sie nur die vordefinierten Prädikate `is/2`, `>/2` und `</2` sowie die Funktion `mod` und die Addition.

Lösung: _____

```

factors(X,Y) :- X > 0, factorsHelp(X,1,Y).
  
```

```
factorsHelp(X,X,[X]).  
factorsHelp(X,N,[N|T]) :- N < X, 0 is X mod N,  
                           Y is N + 1, factorsHelp(X,Y,T).  
factorsHelp(X,N,T) :- N < X, A is X mod N, A > 0, Y is N + 1, factorsHelp(X,Y,T).
```