Rheinisch-Westfälische Technische Hochschule Aachen
Viktor Engelmann, 15. Dezember 2008
Erstellt mit LaTeX2e

Prolog in der Praxis

Inhaltsverzeichnis

1 Einleitung

1.1 Vorwort, Motivation

Am Anfang des Informatik-Studiums haben viele Studenten Erfahrung mit imperativen Programmiersprachen wie C/C++, Pascal/Delphi, Python oder Java. Mit logischen Programmiersprachen hat im Gegensatz dazu kaum einer davon Erfahrung und erfahrungsgemäß haben viele Studenten große Probleme damit, diese Programmiersprache zu verstehen. Der erste Eindruck bleibt bei den meisten Studenten hängen und sie freunden sich nie mit Prolog an.

Ich habe Prolog schon in der Schule kennengelernt - auch ich fand es auf den ersten Blick „doof“, weil für jemanden, der imperative Programmierung gewohnt ist, nicht ersichtlich ist, wann eigentlich was passiert. Das kommt hauptsächlich daher, dass man nicht selbst beschreibt, **wie** ein Ergebnis berechnet wird (also wann was zu tun ist) sondern eher beschreibt, **was** zu berechnen ist, also wie eine Lösung für ein Problem aussieht - den Rest erledigt Prolog selbstständig, was Einsteiger eben nicht gewohnt sind. Man muss nicht nur eine neue Syntax lernen, sondern ein ganz anderes Grundkonzept.

Da ich Prolog schon in der Schule kennengelernt habe, hatte ich weit mehr Praxis damit, als die 2-3 Wochen während des ersten Semesters und dadurch mehr Zeit, es zu verstehen. Im Rahmen eines HiWi-Jobs am Lehrstuhl für Wissensbasierte Systeme, bei Prof. Gerhard Lakemeyer an der RWTH Aachen habe ich ausserdem 12 Monate lang intensiv praktisch mit Prolog gearbeitet (an einer Roboter-Programmiersprache namens Golog). Seitdem ich verstanden habe, wie Prolog funktioniert und noch einige Tricks gelernt habe, die ich hier auch vorstellen will, erkenne ich häufig Situationen, in denen man ein Programm in Prolog schneller entwickeln kann, als man es in imperativen Sprachen könnte, weil man dort die Informationen in oft umständlich zu entwickelnden Datenstrukturen ablegen muss und die Art, wie Prolog arbeitet, selbst schreiben muss. Als Beispiele sind besonders kombinatorische Rätsel, Sprachverarbeitung und künstliche Intelligenz zu nennen.

Dieses Verständnis möchte ich in diesem Tutorial weitergeben, um eine eigentlich sehr nützliche Programmiersprache zu de-mystifizieren und für mehr Menschen zugänglich zu machen. Änderungsvorschläge bitte an

Viktor.Engelmann@RWTH-Aachen.de

oder selbst vornehmen, der LaTeX 2_ε-Quellcode ist im Archiv enthalten. Bitte beachten: das Dokument steht unter der **Creative Commons Lizenz** BY-NC-SA

siehe http://creativecommons.org/licenses/by-nc-sa/3.0/de/

wobei ich den Verkauf zum Selbstkostenpreis (was nach EU-Recht komerzielle Nutzung ist) ausdrücklich erlaube.

1.2 Zielgruppe

Ich richte mich hier vor allem an Menschen, die schon etwas Erfahrung mit imperativen Sprachen haben, so werde ich z.B. ohne viele Erklärungen über den

Quicksort sprechen und erwarte, dass der Leser ihn kennt oder werde Vergleiche zu imperativen Sprachen ziehen, die Lesern, die Erfahrung mit imperativen Sprachen haben, helfen werden. Nichtsdestotrotz hoffe ich, dass auch unerfahrene Leser viel aus dieser Anleitung lernen können und hoffe auf Feedback.

1.3 Das Grundkonzept logischer Programmiersprachen

Man kann sich eine logische Programmiersprache, als künstliche Intelligenz vorstellen und die Programme als die Datenbank oder das Wissen, das man diesen künstlichen Intelligenzen zur Verfügung stellt. Man bezeichnet es auch als „***Wissensbank***“ oder „***Wissensbasis***“. Man „startet“ die „Programme“, indem man in der Eingabezeile, die Prolog anzeigt, ***Anfragen*** (auch „***Goal***“ genannt) stellt. Prolog versucht dann, diese Anfragen anhand seines Wissens zu beantworten.

1.4 Installation und erste Schritte

Ich erkläre Prolog hier unter Verwendung des open source Interpreters „SWI Prolog“. Die Aussagen stimmen z.T. nicht für andere Interpreter/Compiler - SWI Prolog erfüllt aber weitgehend den ISO Standard d.h. andere ISO-konforme Interpreter/Compiler sind vielleicht anders zu bedienen, sollten aber die gleichen Quellcodes akzeptieren. Da auf den Computern im CIP Pool der RWTH Aachen GNU Prolog installiert ist, werde ich möglichst auch auf GNU Prolog eingehen. Wenn man die Wahl zwischen Prolog Interpretern hat, rate ich persönlich zu SWI, weil es einerseits (im Gegensatz zu GNU Prolog und Borland Turbo Prolog) einen großen Teil des ISO Standards erfüllt und andererseits (im Gegensatz zu ECLiPSe Prolog) die Eingabe der Anfragen sehr benutzerfreundlich ist (z.B. hat SWI „code-completion“, eine History und man kann mit den Pfeiltasten den Cursor durch die Eingabezeile bewegen). Ausserdem hat SWI Prolog eine Abstraktionsschicht vor dem Dateisystem, wodurch Programme sehr leicht plattform-unabhängig programmiert werden können.

1.4.1 Installation

Unter Linux Systemen wird SWI Prolog (sofern man Paketquellen hat) installiert, indem man es in der grafischen Paketverwaltung sucht oder in der Shell je nach Distribution einen der folgenden Befehle gibt:

- apt-get install swi-prolog
- yum install swi-prolog
- smart install swi-prolog
- urpmi swi-prolog
- emerge swi-prolog

Unter Windows Systemen läd man sich das Setup Programm von

http://www.swi-prolog.org/dl-stable.html

herunter und führt es aus. (Dort ist auch ein RPM Paket für Linux Systeme zu finden).

Um GNU Prolog unter Linux zu installieren, installiert man wie oben das Paket *gprolog* anstatt $swi - prolog$. Für Windows findet man das Setup Programm unter

http://www.gprolog.org/#download

1.4.2 Dateien Laden

SWI Prolog ist nur ein Compiler und Interpreter für Prolog, keine IDE, d.h. man muss Quellcodes in einem Texteditor (z.B. Notepad, Notepad++, Gedit, Kate, KWrite, ...) schreiben und in Prolog laden. Unter Windows wird die Dateiendung ***.pl*** mit Prolog assoziiert, ein Doppelklick auf eine Prolog Quellcode Datei im Explorer/Arbeitsplatz startet also Prolog und läd die Datei. Unter Linux wird das Programm *pl* installiert. Der Shellbefehl *pl -f dateiname* startet Prolog und läd die Datei. Unter GNU Prolog heisst das Programm nicht „pl“, sondern „prolog“ bzw „gprolog“. Dort gibt man im laufenden Programm die Anfrage ***[dateiname].*** ein (einschließlich der eckigen Klammern und dem Punkt am Ende - die Dateiendung muss nur angefügt werden, wenn sie von „.pl“ abweicht). Auf diese Weise kann man auch unter SWI Prolog Dateien innerhalb des bereits laufenden Systems laden.

1.4.3 Code Completion

Ein sehr nützliches Feature von SWI Prolog ist die Code Completion: während man eine Anfrage eingibt kann man die Tabulator-Taste ⇆ drücken. Prolog listet dann Möglichkeiten auf, wie die Eingabe fortgesetzt werden kann. Gibt es hierzu nur eine Möglichkeit, wird sie direkt in die Eingabezeile eingefügt.

1.4.4 Programme unterbrechen

Manchmal kommt es vor, dass Prolog aufgrund irgendeines Fehlers in der Wissensbasis unendlich lange versucht, eine Anfrage zu beantworten. In so einer Situation kann man die Tastenkombination $Strg + C$ benutzen, um Prolog zu unterbrechen. Danach kann man mit verschiedenen Tasten Befehle geben. Z.B. kann man mit e Prolog beenden, mit a kann man die aktuelle Frage abbrechen (Prolog geht dann zurück zur Eingabezeile). Mit der Taste h kann man eine Liste der möglichen Befehle anzeigen lassen.

1.4.5 Prolog Beenden

Mit Strg+D oder der Anfrage ***halt.*** (einschließlich des Punktes) wird Prolog beendet.

2 Grundlegende Techniken

Ein Prolog Programm besteht aus sogenannten Prädikaten. Ein Prädikat ist so etwas ähnliches wie ein Funktions**name** in imperativen Sprachen - die „Funktionen“, die diesen Namen tragen nennt man „Klauseln“. Sie sehen so ähnlich aus wie Funktionen/Methoden in imperativen Sprachen (vgl. Beispiel unten) - sie haben einen Namen und Parameter - die Parameter stehen zwischen Klammern und werden durch Kommata getrennt, aber es kommen keine Typdefinitionen vor. Hinter der schließenden Klammer steht (vorerst) nur ein Punkt. Kommentare setzt man zwischen /* und */ wie in C und C++ bzw. einzeilige Kommentare schreibt man hinter einem % wie in LaTeX 2_ε.

2.1 Ein erstes Beispiel

Der folgende Code enthält lediglich Informationen über ein paar (fiktive) Menschen: Vaterschaft, Mutterschaft und Ehe. Dieser Quellcode ist in dem Archiv, in dem auch dieses Dokument enthalten ist, im Unterordner *Sources\Kapitel 2* enthalten - es heisst *family1.pl*

```
verheiratet(wilhelm,margarethe).
verheiratet(fridolin,irmtraut).

vater(hans,wilhelm).
vater(wilhelm,fridolin).
vater(wilhelm,hermann).

mutter(margarethe,fridolin).
mutter(irmtraut,hermann).
```

„verheiratet“, „vater“ und „mutter“ sind die ***Prädikate***. vater(hans,wilhelm) ist eine ***Klausel*** des vater-Prädikats, vater(wilhelm,fridolin) ist eine weitere Klausel des vater-Prädikats. Für Prolog bedeutet dies nur, dass eine Beziehung namens „vater“ zwischen *hans* und *wilhelm*, zwischen *wilhelm* und *fridolin* und zwischen *wilhelm* und *hermann* besteht - dass wir den ersten Parameter als den Vater und den zweiten Parameter als das Kind interpretieren, weiss Prolog nicht. Es weiss auch nicht, was wir unter dem Wort „Vater“ verstehen - es weiss jetzt nur, dass zwei Dinge eine Beziehung namens „Vater“ haben können und in den Klauseln legen wir fest, zwischen welchen „Dingen“ diese Beziehung besteht.
Ein Prädikat, das n Parameter hat, nennt man auch ein ***n*-stelliges Prädikat** (in englisch: ***n*-ary predicate**) wobei man bei n =1,2 und 3 eigene Begriffe benutzt: ein 1-stelliges Prädikat ist *unär*, ein 2-stelliges Prädikat *binär* und ein 3-stelliges Prädikat ist *ternär* (in englisch: *unary*, *binary* und *ternary*) z.B. ist *mutter* ein 2-stelliges oder binäres Prädikat. n nennt man auch die ***Stelligkeit*** oder Arität (in englisch: ***arity***) des Prädikats.
Wie man sieht, liegen die verschiedenen Klauseln der Prädikate alle beieinander, es kommt z.B. keine Klausel des *verheiratet* Prädikats zwischen zwei Klauseln des *mutter* Prädikats - Es wird einerseits als guter Stil betrachtet, die Klauseln der Prädikate beieinander liegend zu schreiben, andererseits ist es eine Fehlerquelle, wenn man dies nicht tut (man könnte z.B. ausversehen zwei Prädikaten den gleichen Namen geben, wodurch Prolog dann falsche Ergebnisse berechnen könnte) Aus diesem Grund gibt SWI Prolog eine Warnung aus, wenn die

Klauseln eines Prädikats nicht beieinander liegen. GNU Prolog ist sogar noch strenger und verwendet sogar die Klauseln nicht mehr, die hinter Klauseln eines anderen Prädikats folgen. Es gibt dann die Meldung „warning: discontiguous predicate <*Name des Prädikats*> - clause ignored“ aus. Manchmal ist es trotzdem sinnvoll, die Klauseln eines Prädikats verteilt oder sogar auf mehrere Quellcode-Dateien verteilt zu programmieren. Wie man Prolog anweist, das zu erlauben, erkläre ich in Kapitel 8.3.

2.2 Eine erste, einfache Anfrage stellen

Nun wollen wir diesen Quellcode (wie in 1.4 beschrieben) in den Prolog Interpreter laden. Wir sehen eine Eingabezeile, an deren Anfang ***?-*** steht. Hier geben wir das Goal ein, zum Beispiel *vater(wilhelm,fridolin).* Der Punkt am Ende der Eingabe ist wichtig![1]
Hat man keinen Fehler gemacht, wird Prolog „true.“ antworten. Wir haben hier gefragt, ob *wilhelm* der Vater von *fridolin* ist und dies konnte Prolog wegen der zweiten Klausel des *vater*-Prädikats bestätigen. Hätte Prolog keinen Beweis für die Anfrage gefunden, hätte es $fail$ anstatt *true* geantwortet.

Konvention Da Prologs Eingabezeile mit ?- beginnt, werde ich in Beispielen (wie es in Prolog-Anleitungen üblich ist) signalisieren, dass eine Zeile eine Anfrage darstellt, indem ich die Zeile ebenfalls mit ?- beginne. An eine solche Beispiel-Anfrage werde ich immer Prologs Antwort anfügen. Z.B. hieße

```
?- vater(wilhelm,fridolin).
true.
```

dass Prolog die Anfrage $vater(wilhelm, fridolin)$. mit *true* beantwortet.

2.3 Bezeichner, Atome

Die Namen der Prädikate und die Namen der Menschen beginnen mit Kleinbuchstaben. Das ist kein Zufall, sondern notwendig, weil Großbuchstaben für Variablen reserviert sind. Wenn man unbedingt Namen mit Großbuchstaben verwenden will, muss man mit Strings arbeiten z.B.

```
mutter('Margarethe','Fridolin').
```

Ich habe mich dafür entschieden, hier nicht mit Strings zu arbeiten, sondern mit diesen sogenannten „Atomen“.
Die Prädikate müssen **immer** mit Kleinbuchstaben beginnen!

2.4 Anfragen kombinieren

Man kann in der Eingabezeile, in der man das Goal eingibt, auch mehrere Anfragen stellen, wobei man sie mit den Operatoren , (Komma für AND) und ; (Semikolon für OR) verknüpfen muss.

[1] Vergisst man den Punkt, erhält man eine weitere Eingabezeile, an deren Anfang eine vertikale Linie steht - dort kann man z.B. den Punkt nachtragen oder eine mehrzeilige Eingabe fortsetzen

```
?- vater(wilhelm,fridolin) , mutter(margarethe,fridolin).
true.
?- vater(wilhelm,hans) ; mutter(margarethe,hermann).
fail.
```

Es gibt auch den Implikationspfeil ->. In der mathematischen Logik gilt eigentlich $A \rightarrow B \Longleftrightarrow \neg A \vee B$, d.h. falls A wahr ist, muss auch B wahr sein, was auch heisst, dass falls A falsch ist, auch B falsch sein darf. In Prolog funktioniert ein einzelnes -> aber genau wie ein AND. Im Beispiel

```
?- vater(wilhelm,hans) -> mutter(margarethe,fridolin).
fail.
?- vater(wilhelm,hans) -> mutter(margarethe,hermann).
fail.
```

ergeben beide Anfragen $fail$, obwohl $vater(wilhelm, hans)$ falsch ist. In Kombination mit einem OR verhält sich das -> wie ein if-then-else. Ist bei der Anfrage A -> $B; C$ das A wahr, dann muss B auch wahr sein, damit der Ausdruck insgesamt als wahr angesehen wird. Ist A falsch, dann muss C wahr sein, damit der Ausdruck insgesamt als wahr angesehen wird.

```
?- 5<3  ->  7>2  ;  7<2.
true.
?- 5>3  ->  7>2  ;  7<2.
fail.
?- 5>3  ->  7<2  ;  7>2.
true.
```

Will man also eine Implikation A -> B im mathematisch-logischen Sinn erreichen, muss man dafür sorgen, dass der Ausdruck A -> $B; C$ wahr wird, wenn A falsch ist - dazu muss C wahr sein, man muss also dafür sorgen, dass C immer wahr ist. Hierzu kann man für C das Systemprädikat $true$ benutzen.

$$A- > B; true$$

Man sollte beachten, dass diese Kombination sich unter Umständen über Klammerungen hinwegsetzt. Nach den Erklärungen von oben sollte $5 < 3$ -> $(7 < 2; 7 > 2)$. eigentlich $fail$ ergeben, weil $5 < 3$ falsch ist und das -> nach dieser Klammerung kein zugehöriges OR hat. Trotzdem ergibt diese Anfrage $true$, weil Prolog entgegen der Klammerung das OR dem -> zuordnet.

Das Kombinieren von Anfragen erscheint zu diesem Zeitpunkt etwas überflüssig, wird aber ab dem nächsten Kapitel unersetzlich.

2.5 Anfragen mit Variablen

Jetzt kommen wir zum interessanten Teil. Bisher haben wir Prolog nur Fragen gestellt, deren Antworten wir auch direkt aus der Datenbank hätten ablesen können. Prolog kann aber weit mehr, als solche primitiven Datenbank-Abfragen. Geben Sie als Goal doch einmal *vater(wilhelm,X).* ein[2], Prolog wird $X =$ *fridolin*

[2] Wie gesagt: Großbuchstaben sind für Variablen

antworten! Drücken Sie nun die Taste *n*, *r* oder *;* und Prolog sucht die nächste Lösung: $X = wilhelm$ (unter GNU Prolog, auf den Computern im CIP Pool der RWTH Aachen, erreicht man dies nur mit der ; Taste!) . Nochmaliges drücken einer dieser Tasten lässt Prolog *No* antworten, weil es keine weitere Lösung gefunden hat.
Auch diese Informationen hätten wir noch leicht aus der Datenbank lesen können, aber man kann ja auch Anfragen kombinieren (siehe 2.4) - damit wollen wir einmal ermitteln, ob es eine Person gibt, die einen Großvater väterlicherseits hat: *vater(Opa,Vater), vater(Vater,Enkel).* Wir haben Prolog damit gefragt, ob es eine Belegung der Variablen Opa,$Vater$,$Enkel$ gibt, sodass die *vater* Beziehung zwischen Opa und $Vater$ besteht UND die *vater* Beziehung zwischen $Vater$ und $Enkel$ besteht. Prologs Antwort lautet: *Opa = hans, Vater = wilhelm, Enkel = fridolin.* nach drücken von *n*,*r* oder ; kommt eine weitere Lösung *Opa = hans, Vater = wilhelm, Enkel = hermann.*
Übung:

- Formulieren Sie eine Anfrage, die sowohl Großväter väterlicherseits, als auch mütterlicherseits bestimmt. Denken sie an das OR (das Semikolon)
- Formulieren Sie eine Anfrage, die alle Großeltern bestimmt.

2.6 Prädikate mit Bedingungen

Es wäre recht sinnlos, wenn man nur auf obige Weise eine Datenbank festlegen könnte und abenteuerliche Goals formulieren müsste, um interessante Daten aus der Datenbank entnehmen zu können - dann könnte man ja kein Programm schreiben, das man einem Kunden zumuten könnte, weil die Bedienungsanleitung damit sehr kompliziert würde - ausserdem werden auch zum Teil Daten ausgegeben, an denen man gar nicht interessiert ist - z.B. war der *Vater* im Beispiel in 2.5 belanglos, wir wollten nur wissen wer *Enkel* und *Opa* sind.
Man kann Variablen auch bei den Prädikaten verwenden. Das macht aber offensichtlich nur Sinn, wenn durch irgendwelche Regeln festgelegt wird, dass nicht einfach alles für die Variablen akzeptiert wird. Man signalisiert, dass das Prädikat ***Regeln*** hat, indem nach dem Kopf des Prädikats anstatt einem Punkt ein *:-* folgt.
Die Regeln formuliert man genauso, wie Goals (Aufrufe anderer Prädikate verknüpft durch Komma, Semikolon, Implikationspfeil, Klammerung, Belegung von Variablen, Vergleiche etc.). Die Regeln werden (wie Goals) durch einen Punkt abgeschlossen. Eine Klausel ohne Regeln bezeichnet man übrigens auch als ***Fakt***.

```
elternteil(E,K):-
   vater(E,K);
   mutter(E,K).
```

Dieses Prädikat lesen wir „E ist Elternteil von K genau dann, wenn E Vater von K ist ODER E Mutter von K ist.“

```
geschwister(A,B):-
   ( mutter(M,A), mutter(M,B) )   /* muetter stimmen ueberein */
   ,                              /* und */
   ( vater(V,A), vater(V,B) ).    /* vaeter stimmen ueberein */
```

Dieses Prädikat lesen wir „A und B sind Geschwister genau dann, wenn die Mutter M von A auch Die Mutter von B ist UND Der Vater V von A auch der Vater von B ist

Übung:

- Schreiben Sie ein Prädikat, das Halbgeschwister bestimmt (der Einfachheit halber dürfen zwei Menschen auch als Halbgeschwister angesehen werden, wenn ihre **beiden** Eltern übereinstimmen)
- Schreiben Sie ein Prädikat, das alle Großeltern bestimmt (Sie dürfen das *elternteil*-Prädikat verwenden)

2.7 Klauseln mit Variablen

Variablen kann man einfach benutzen, ohne sie zu deklarieren (dass es sich bei einem Bezeichner um eine Variable handelt, erkennt Prolog daran, dass sie mit einem Großbuchstaben beginnen). Wenn eine Variable zum ersten mal verwendet wird, wird sie angelegt.

```
nullstellen(P,Q,NST):-
   A is -P/2,
   B is sqrt(A*A-Q),
   NST is A-B.
nullstellen(P,Q,NST):-
   A is -P/2,
   B is sqrt(A*A-Q),
   NST is A+B.
```

Hier werden in beiden Klauseln die Variablen A und B angelegt und sofort belegt - in Prolog sagt man dazu, dass die Variablen „gebunden“ werden - wenn man ihnen keinen Wert zuweist, bezeichnet man die Variablen als „ungebunden“ oder als „frei“. Man kann Variablen auch im Aufruf eines anderen Prädikats zum ersten mal verwenden - d.h. wenn A so zum ersten mal auftritt:

```
beispiel:-
   nullstellen(0,-1,A).
```

dann wird es ebenfalls angelegt und bekommt dann den Wert, mit dem NST im anderen Prädikat gebunden wird (in diesem Beispiel -1 oder 1).

2.8 OR vs. mehrere Klauseln

Viele Bücher über Prolog wollen einem verbieten, das OR innerhalb der Bedingungen eines Prädikats zu verwenden, weil es genügt, mehrere Klauseln zu benutzen

```
a :- b ; c.
```

entspricht

```
a :- b.
a :- c.
```

Es ist wahr, dass man jedes OR auf diese Weise entfernen kann und die formale Definition von Prolog-Programmen kein OR in den Bedingungen erlaubt (weil es sich bei den Klauseln dann formal nicht mehr um sog. ***Horn-Formeln*** handelt). Aber das Problem ist, dass man beim Entfernen eines ORs die Klausel, in der es vorkam, verdoppeln muss

```
a :- (b ; c) , (d ; e).
```

⇓

```
a :- b , (d ; e).
a :- c , (d ; e).
```

⇓

```
a :- b , d.
a :- b , e.
a :- c , d.
a :- c , e.
```

Damit lässt sich ohne Mühe eine Klausel mit ORs konstruieren, die zu sehr vielen Klauseln ohne ORs zerlegt würde:

```
a :- (b1 ; c1) , (b2 ; c2) , ... , (bn ; cn).
```

entspräche 2^n Klauseln ohne ORs. Wenn $c1, c2, ..., cn$ fehlschlagen (weil sie z.B. nur die Bedingung $false$ haben), wäre die Laufzeit der Anfrage $a.$ im Fall mit ORs linear und im Fall ohne ORs exponentiell. Von daher halte ich die Verwendung des OR innerhalb der Klauseln für legitim. Man könnte sogar alle Klauseln eines Prädikats zu einer einzigen zusammenfassen, doch auch das ist nicht unbedingt sinnvoll, wenn z.B. die verschiedenen Klauseln sehr verschiedene Parameter haben. Ich würde zwei Klauseln am ehesten dann zusammenfassen, wenn sie weitgehend die gleichen Bedingungen haben, aber das ist Geschmackssache.

2.9 Zuweisung und (Un-)Gleichheit

2.9.1 Der = Operator

Der = Operator hat eine sehr kuriose Rolle, weil er weder für Vergleiche, noch für Zuweisungen im alltäglichen Sinn wirklich gut zu gebrauchen ist:
$7 = 7$ ist true, aber $7 = 4 + 3$ ist false, für Vergleiche ist das = also nicht gut geeignet. Stellt man die Anfrage $A = 5 + 3$, dann antwortet Prolog $A = 5 + 3$, das = hat A also mit dem Ausdruck $5 + 3$ gebunden, anstatt den Ausdruck zuerst auszuwerten. Hingegen liefert die Anfrage $A + B = 5 + 3$ die Belegung $A = 5$, $B = 3$. Der = Operator bindet also Teilausdrücke an Teilausdrücke!
Genau gesagt (auch wenn ich hier nicht weiter auf das Thema eingehen will: es könnte Ihnen in Vorlesungen begegnen) ***unifiziert*** das = die beiden Seiten, d.h. es versucht die rechts und links vorkommenden Variablen so zu belegen, dass die rechte und linke Seite gleich werden - hierzu berechnet Prolog den ***Most General Unifier*** (MGU) und wendet ihn an. Interessierte Leser verweise ich an

http://verify.rwth-aachen.de/programmierung/folien/IV3_Rechnen.pdf

Übung:

- Wie sind C und D gebunden? C + D = 5 + 4 * 3.
- Wie sind C und D gebunden? C * D = 5 + 4 * 3.
- Warum schlägt die zweite Bindung fehl?

2.9.2 Der *is* Operator

Nachdem der = Operator nicht für die „alltäglichen“ Zwecke geeignet ist, stellt sich die Frage, wie man nun Ergebnisse von arithmetischen Ausdrücken vergleichen oder an Variablen binden soll. Hierzu gibt es den infix Operator *is*. *7 is 4 + 3* ist true. Das *is* kann also vergleichen. *A is 5+3* liefert *A=8*, das *is* kann also auch Variablen Werte zuweisen. Wann es zuweist und wann es vergleicht ist einfach: wenn links eine ungebundene Variable steht, wird sie mit dem Ergebnis des arithmetischen Ausdrucks rechts gebunden. Steht links eine Konstante oder eine gebundene Variable, wird sie mit dem Ergebnis des arithmetischen Ausdrucks rechts verglichen.

WICHTIG:

- Das *is* wertet nur den Ausdruck auf der rechten Seite aus! *5+3 is 5+3* ist falsch, weil dies ausgewertet wird zu *5+3 is 8*, also rechts eine Zahl, links ein Ausdruck
- Damit das *is* den Ausdruck auf der rechten Seite auswerten kann, dürfen darin keine ungebundenen Variablen vorkommen! In dem Fall wäre Prolog auch nicht so nett, einfach „No“ zu liefern, sondern es bricht gleich die komplette Berechnung der Anfrage ab mit der Meldung „ERROR: is/2: Arguments are not sufficiently instantiated“.

2.9.3 der =:= Operator

Will man zwei arithmetische Ausdrücke auf Gleichheit testen, kann man den =:= Operator anwenden $5 + 3 =:= 2 * 4$ ist true, allerdings kann der =:= Operator keine Variablen binden. Bei diesem Operator dürfen weder links noch rechts ungebundene Variablen vorkommen.

2.9.4 Gleiche Variablennamen

Eine weitere Form, wie man Dinge[3] auf Gleichheit prüfen oder belegen kann ist, ihnen gleiche Variablennamen zu geben. Damit kann man auf sehr elegante Weise Parameter schon im Kopf eines Prädikats zuweisen oder vergleichen. Zum Beispiel

```
equal(A,B):-          entspricht          equal(A,A).
   B=A,                                       ... .
   ... .
```

Diese Methode wird besonders in Kapitel 4 über Listen einige sehr elegante Programme ermöglichen.

[3] Integers, Strings,..., Alles, was auch mit = verglichen oder gebunden werden kann.

Ein anderes Beispiel ist, dass man nicht unbedingt zwei Ergebnisse aus Prädikaten vergleichen muss, nachdem beide Prädikate fertig berechnet wurden, sondern man kann das Ergebnis des ersten Prädikats als Parameter des zweiten Prädikats benutzen - dann akzeptiert das zweite Prädikat nur, falls sein Ergebnis mit dem übergebenen Ergebnis übereinstimmt.

```
test:-              entspricht        test:-
   foo(A),                               foo(A),
   bar(B),                               bar(A).
   A=B.
```

2.9.5 Der \= Operator

Der „ungleich“ Operator hat die Form \ =, was ein durchgestrichenes Gleichheitszeichen darstellt. Er wertet arithmetische Ausdrücke links und rechts genausowenig aus, wie der = Operator.

2.9.6 der = \ = Operator

Der zweite „ungleich“ Operator hat die Form = \ = und wertet analog zum =:= Operator arithmetische Ausdrücke links und rechts aus, bevor er sie auf Ungleichheit testet.

2.10 Operatoren und Relatoren

Wie in praktisch jeder Programmiersprache gibt es die Operatoren +, −, ∗ und /. Das / berechnet Divisionen mit Nachkommastellen (auch wenn man damit Integer durch Integer teilt). Integer-Divisionen (Nachkommastellen werden abgeschnitten) erreicht man durch den Operator // z.B. 5//3 ergibt 1. Das Modulo (Divisionsrest) ist ebenfalls ein infix Operator und trägt die Bezeichnung *mod*, wie in Pascal/Delphi. z.B. 5 *mod* 3 ergibt 2. Es gibt auch den ^ Operator für Potenzierung (Potenzrechnung vor Punktrechnung!)
Ebenfalls nicht ungewöhnlich sind die Relatoren <, > und >=. Das „Kleinergleich“ hat die Form =<, was in den meisten anderen Programmiersprachen anders ist - Eine leichte Merkregel hierfür ist, dass das Gleichheitszeichen zusammen mit der spitzen Klammer **keinen** Pfeil ergibt.

Weitere mathematische Funktionen (z.B. Trigonometrie, Runden, Random) sind unter

http://gollem.science.uva.nl/SWI-Prolog/Manual/arith.html

dokumentiert.

2.11 Rückgabewerte

Ein Prolog Prädikat hat in dem Sinne keinen Rückgabewert (ausser technisch gesehen Yes oder No) Soll ein Prädikat ein Ergebnis berechnen, muss man einen Parameter dafür definieren, für den beim Aufruf eine ungebundene Variable übergeben wird und die innerhalb des Prädikats gebunden wird. Es wird als guter Stil betrachtet, die „Rückgabewerte“ als letzte Parameter zu definieren. Gut, jemand könnte für andere Parameter ungebundene Variablen übergeben,

um das Prädikat zu anderen Zwecken zu benutzen (das werden wir in Kapitel 4 über Listen exzessiv tun) - aber darum kümmert man sich nicht, sondern bezeichnet einfach die Parameter als Ergebnisse, die das Prädikat seinem Namen nach berechnen soll - z.B. plus(A,B,C) hieße C *is* $A+B$, auch wenn das Prädikat so programmiert worden ist, dass man es auch verwenden kann, um Differenzen zu bilden. Um auf das Beispiel aus 2.7 zurückzukommen:

```
nullstellen(P,Q,NST):-
   A is -P/2,             /*  PQ Formel zur Bestimmung der  */
   B is sqrt(A*A-Q),      /*  Nullstellen von x^2+P*x+Q=0   */
   NST is A-B.
nullstellen(P,Q,NST):-
   A is -P/2,
   B is sqrt(A*A-Q),
   NST is A+B.
```

hier sollen P und Q übergeben werden um die Nullstellen NST zu berechnen - diese werden in der jeweils dritten Zeile gebunden. Die Anfrage $nullstellen(0, -1, NST)$. würde $NST = 1$ und $NST = -1$ liefern.

Aufmerksame Leser werden verwundert sein, wie man *B is sqrt(A*A-Q)* verwenden kann, wenn Prädikate keinen Rückgabewert haben - der Grund hierfür ist, dass *sqrt* und ähnliche arithmetische Ausdrücke keine Aufrufe von Prädikaten sind, sondern Teil eines Terms sind, den der *is* Operator Schrittweise auswertet. Wie genau der *is* Operator dies tut, werden wir in Kapitel 6.2 über Termverarbeitung betrachten.

2.12 Negation

Den Wahrheitswert, den ein Prädikat zurückliefert, kann man auch negieren. Hierzu schreibt man *not* und den zu negierenden Teil in Klammern dahinter - dieser kann wieder mehrere Prädikate enthalten, verknüpft durch Komma, Semikolon, Implikationspfeil, weitere „not“ s, Klammerungen.
Nach dem ISO Standard heisst das Systemprädikat nicht *not*, sondern \+ und der zu negierende Teil muss in doppelte Klammern, falls er Verknüpfungen wie AND oder OR enthält. Obwohl SWI Prolog auch *not* kennt und darin freundlicherweise die doppelte Klammerung nicht verlangt, werde ich den ISO Standard verwenden.

```
unehelichesKind(Kind):-
   mutter(Mutter,Kind),              /* Mutter bestimmen */
   vater(Vater,Kind),                /* Vater bestimmen */
   \+((  verheiratet(Vater,Mutter);  /* verheiratet? */
         verheiratet(Mutter,Vater)  )).
```

Bei der Negation gilt: der Teil in den Klammern ist *wahr*, wenn es eine Lösung dafür gibt, **die Prolog anhand seiner Daten auch berechnen kann** (Was Prolog schlussfolgern kann, dazu kommen wir in Kapitel 3.4) d.h. \+((*Aussage*)) ist wahr, wenn keine Lösung für *Aussage* gefunden wird.

Diese Definition weicht von der mathematischen Logik ab, wo $\neg\varphi$ gilt, falls bewiesen werden kann, dass φ falsch ist. Die abweichende Definition der Negation, die Prolog anwendet, nennt man ***Negation by Failure***, und geht davon aus, dass jede wahre Aussage beweisbar ist (diese Annahme nennt man die ***Closed World Assumption***). Wenn Prolog diese Annahme nicht machen würde, müsste man sehr viel mehr und sehr viel vorsichtiger programmieren. Denken Sie an einen Zug-Fahrplan: Wenn dort kein Zug um 8:45 von Aachen nach Köln verzeichnet ist, nehmen Sie an, dass um 8:45 kein Zug von Aachen nach Köln fährt, obwohl das keine mathematisch-logische Schlussfolgerung ist, solange man keine zusätzliche Information hat, die besagt, dass jeder Zug auch verzeichnet ist - und genau solche Informationen müssten Sie programmieren, wenn Prolog keine geschlossene Welt annehmen würde. Gegen diesen Luxus tauscht man aber einige Eigenschaften der mathematisch-logischen Negation, so gilt im Allgemeinen nicht mehr $\varphi = \backslash + \backslash + \varphi$ und man verliert die Korrektheit der Lösungen - z.B. wird mit der Datenbank $a(3).b(3).b(4).$ die Anfrage $(\backslash + a(X)), b(X)$ mit $fail$ beantwortet, obwohl $X = 4$ eine gültige Lösung wäre, die bei umgekehrten Anfragen $b(X), (\backslash + a(X))$ auch gefunden würde.

Übung:

- Warum wird bei obiger Anfrage $(\backslash + a(X)), b(X)$ die Lösung $X = 4$ nicht gefunden?
- Schreiben Sie ein Prädikat, das Halbgeschwister bestimmt (wobei dieses mal zwei Menschen nicht mehr als Halbgeschwister betrachtet werden sollen, deren Eltern beide übereinstimmen)

Man spricht in dem Zusammenhang auch von Nicht-Monotonem Schlussfolgern, worüber es ganze Bücher gibt.

2.13 Unbedeutende („Anonyme“) Parameter

Wenn bei einem Aufruf eines Prädikats ein Parameter völlig unwichtig ist, kann man ihn als „beliebig“ deklarieren, indem man dafür den Unterstrich _ übergibt[4]. Will man z.B. nur wissen, ob *margarethe* Kinder hat, kann man $mutter(margarethe, \_)$. fragen und erhält Yes, ohne den Namen ihres Kindes übergeben zu haben und ohne den Namen ihres Kindes zurück zu bekommen. Man hätte damit allerdings nicht in 2.5 die Ausgabe des Vaters unterdrücken können, weil diese Information nicht unwichtig ist, denn der Sohn des Opas muss ja mit dem Vater des Enkels übereinstimmen!

Auch Parameter in Klausel-Köpfen kann man als „beliebig“ deklarieren - denn manchmal ist ein Parameter nicht in allen Klauseln von Bedeutung

```
vater(gott,_).
```

Wobei diese Deklaration in diesem Beispiel viele Berechnungen merkwürdige Lösungen produzieren lassen würde (z.B. ist jeder mit jedem verwandt).

Wenn man eine Variable in einer Klausel nur ein mal benutzt (z.B. ihr nur im Kopf einen Namen gibt, sie dann im Rumpf aber nicht benutzt o.Ä.), dann

[4] Auf den Unterstrich dürfen noch beliebige Zeichenketten (auch mit Zahlen) folgen

meldet Prolog beim Laden des Quellcodes „Singleton variables“ was bedeutet, dass man sie auch als beliebig deklarieren könnte.

2.14 Ein- und Ausgabe

Als „Bedingungen“ kann man auch verschiedene Systemprädikate benutzen, durch die Prolog sich annähernd wie eine imperative Sprache benutzen lässt. Zum Beispiel kann man die Ausgabe eines Textes oder von Variablen als „Bedingungen“ nehmen - diese werden als *wahr* betrachtet, wenn Prolog den übergebenen Teil erfolgreich ausgeben konnte.

2.14.1 Ausgabe

Ausgaben geschehen durch das Prädikat *write*(...) oder *writeln*(...), welches hinter dem ausgegebenen Text einen Zeilenumbruch anfügt. Das Prädikat *nl* (ohne Parameter) bewirkt nur einen Zeilenumbruch.

```
?- write('test'),write('test').
testtest
true.

?- write('test'), nl, write('test'), nl.
test
test
true.

?- writeln('test'),writeln('test').
test
test
true.
```

Zeilenumbrüche kann man, wie in den meisten anderen Programmiersprachen, auch mit einem $\backslash n$ in einem String erreichen

```
?- write('test\ntest\n').
test
test
true.
```

Ausgegebener Text muss in einzelne Anführungszeichen ' gesetzt werden, weil Strings mit doppelten Anführungszeichen intern als Listen von Integern behandelt werden.

```
?- writeln("test").
[116, 101, 115, 116]
true.
```

Ähnlich wie der = Operator werten die Ausgabe-Prädikate arithmetische Ausdrücke nicht aus, bevor sie sie ausgeben

```
?- writeln(5+4*3).
5+4*3
true.
```

Ebenso werden nicht-arithmetische Ausdrücke nicht ausgewertet, bevor sie ausgegeben werden, aber ungebundene Variablen werden (schon beim Aufruf) mit internen Namen ersetzt, damit lange Variablennamen nicht die Laufzeit verschlechtern.

```
?- writeln(vorfahre(hermann,hermann)).
vorfahre(hermann,hermann)
true.

?- writeln(vorfahre(hermann,X)).
vorfahre(hermann,_G235)
true.
```

Es gibt noch weitere Prädikate zur Ausgabe, die *printf* in C oder *System.out.printf* in Java recht ähnlich sind, sie heissen *writef*, *swritef* und *format*, sie sind dokumentiert unter

$$http://gollem.science.uva.nl/SWI-Prolog/Manual/format.html$$

Ich werde sie hier nicht weiter benutzen, ich will nur darauf Aufmerksam machen, dass sie existieren.

```
?- writef('%15L%w', ['Hello','World']).
Hello          World
true.

?- swritef(S, '%15L%w', ['Hello','World']).
S = "Hello          World".
```

Der erste Parameter des writef bestimmt das Format, der zweite Parameter ist eine Liste, die die Dinge enthält, die in den Format-String eingesetzt werden. *swritef* gibt nichts auf dem Bildschirm aus, sondern es hat noch einen Parameter vor dem Format-String, für den man eine ungebundene Variable übergibt. Diese Variable wird gebunden mit dem String, den *writef* auf dem Bildschirm ausgeben würde.

2.14.2 Eingabe

Eingaben kann man in Prolog durch das Prädikat *read* einlesen - dieses Prädikat ist aber mit sehr großer Vorsicht zu benutzen! Zum Ersten müssen die Eingaben durch einen Punkt abgeschlossen werden, zum Anderen kann hier ein ganzer **Term** eingegeben werden - das heisst zum Beispiel, dass der Benutzer keine zwei Worte hintereinander schreiben kann, die durch ein Leerzeichen getrennt sind, weil das kein gültiger **Term** ist! Wenn der Benutzer einen String eingeben soll, muss er Anführungszeichen darum setzen, ausserdem kann der Benutzer keine Wörter schreiben, die mit Großbuchstaben beginnen, weil Prolog sie sonst als Variablen interpretiert:

```
?- read(X).
|: Y.         /*  Eingabe  */
X = _G180 ;   /* Prologs Interpretation der Eingabe */
No            /* (eine freie Speicheradresse) */
```

Nach sehr langem suchen im Internet und in der Dokumentation glaube ich, dass SWI Prolog keine eigenen Prädikate mitbringt, durch die man den Benutzer auf einfachere Weise Eingaben machen lassen kann, daher habe ich eigene Prädikate geschrieben, die hierzu gut sind.
Im Ordner *source* (im Archiv, in der auch dieses Dokument war) liegt eine Datei namens *io.pl*, die in eigene Projekte eingebunden werden kann, wie in Kapitel 8.1 beschrieben. Es beinhaltet die Prädikate *readString*, *readInt* und *readFloat* - diese brauchen jeweils einen ungebundenen Variablenparameter, an den die Eingabe des Benutzers gebunden wird. Diese Eingabe muss nun nicht mehr durch einen Punkt abgeschlossen werden und Strings müssen nicht mehr in Anführungszeichen gesetzt werden. Die Datei enthält auch ein Prädikat *writeString*, welches Strings, die in doppelte anführungszeichen gesetzt sind, nicht mehr als Liste ausgibt, sondern als String.

2.14.3 Dateien Schreiben

Um in eine Datei zu schreiben, muss man sie erst einmal öffnen. Dazu benutzt man das Prädikat *open*, welches als Parameter einen Dateinamen bekommt und einen Modus (zum Schreiben benötigt man den Modus *write*). Als dritten Parameter übergibt man eine freie Variable, an die ein sogenannter ***Handle*** gebunden wird, über welchen die Datei identifiziert wird. Schreiben kann man in die Dateien mit den gleichen Prädikaten wie auf den Bildschirm - jedes dieser Prädikate existiert ein weiteres mal, mit einem zusätzlichen ersten Parameter, für den man den Handle der Datei übergibt. Zuletzt muss man die Dateien noch schließen

```
?- open('test1.txt',write,D1), open('test2.txt',write,D2),
|   write(D2, 'foo'), write(D1, 'bar'),
|   close(D1), close(D2).
```

Diese Anfrage generiert die Dateien $test1.txt$ und $test2.txt$, schreibt in $test1.txt$ den Text „bar“ und in $test2.txt$ den Text „foo“.

Wenn eine der Dateien schon existiert, wird sie überschrieben. Wenn man aber statt *write* den Modus *append* verwendet, wird der ausgegebene Teil an die Datei hinten angefügt, sofern sie schon existiert. Nach der weiteren Anfrage

```
?- open('test1.txt',append,D1), write(D1, 'ney'), close(D1).
```

enthält $test1.txt$ den Text „barney“. Es ist übrigens problemlos möglich, bereits benutzte Variablen für Handles (wie oben $D1$) wiederzuverwenden, nachdem die zuerst zugeordnete Datei geschlossen wurde. Hingegen würde das zweite *open* in

```
?- open('test1.txt',write,D1), open('test2.txt',write,D1).
```

einfach fehlschlagen, sodass diese Anfrage $fail$ liefert.

2.14.4 Im Dateisystem navigieren (TODO)

Es stellt sich die Frage, in welchem Verzeichnis die Dateien $test1.txt$ und $test2.txt$ im vorigen Kapitel abgelegt würden. Genau wie eine Unix-Shell oder die DOS-Eingabeaufforderung hat Prolog ein aktuelles Arbeitsverzeichnis „in dem es sich

befindet“. Beim Start von Prolog unter Linux wird als Arbeitsverzeichnis das Verzeichnis benutzt, in dem sich die Shell beim Start befunden hat.

Leider gibt es anscheinend keinen ISO-Standard für die Prädikate zur Navigation durch das Dateisystem. Jeder Prolog-Interpreter hat seine eigenen Prädikate, die untereinander völlig inkompatibel sind. SWI Prolog hat (im Gegensatz zu den anderen mir bekannten Interpretern) zusätzlich eine Abstraktionsschicht, durch die man unter Windows und Unix-artigen Betriebssystemen einheitliche Pfadnamen verwenden kann.

Das Prädikat *working_directory* kann dazu benutzt werden, das Arbeitsverzeichnis zu ändern, übergibt man aber für seine beiden Parameter die gleiche freie Variable, erfährt man das aktuelle Arbeitsverzeichnis.

```
?- working_directory(WorkDir, WorkDir).
WorkDir = '/home/viktor/'.
```

Die Dateien *test*1.*txt* und *test*2.*txt* befinden sich also im Verzeichnis „/home/viktor“. Um das Arbeitsverzeichnis zu ändern, kann man wie gesagt *working_directory* benutzen. Das neue Arbeitsverzeichnis wird als zweiter Parameter übergeben, der erste Parameter wird an das alte Arbeitsverzeichnis gebunden.

```
?- working_directory(OldDir, '/home/viktor/Downloads/').
OldDir = '/home/viktor/'.
?- working_directory(WorkDir, WorkDir).
WorkDir = '/home/viktor/Downloads/'.
```

würde man jetzt erneut wie im letzten Kapitel die Dateien *test*1.*txt* und *test*2.*txt* erstellen, würden diese also ins Verzeichnis „/home/viktor/Downloads“ geschrieben werden.

Mit dem Prädikat $expand\_file\_name$ erhält man eine Liste von Dateien im aktuellen Verzeichnis, wobei der erste Parameter die ***Wildcards*** festlegt - hierfür wird man in aller Regel '*' benutzen, man kann aber z.B. auch die Dateien mit einer bestimmten Endung filtern

```
?- expand_file_name('*.pl', Dateien).
Dateien = ['for.pl', ].
```

wie man mit solchen Listen umgeht, werden wir aber erst in Kapitel 4 betrachten.

2.14.5 Dateien Lesen (TODO)

2.15 Rekursion

Mathematische Funktionen nennt man **rekursiv**, wenn sie Teil ihrer eigenen Definition sind, zum Beispiel die Fibonacci-Zahlen:

$$fibo(n) = fibo(n-1) + fibo(n-2)$$

Auch Prolog Prädikate können rekursiv sein - da es keine richtigen Schleifen gibt, sind sie es sogar fast immer. Rekursive Funktionen oder Prädikate brauchen immer eine Abbruchbedingung, denn sonst kommt es zu einer sogenannten „Endlosrekursion“ , wie

$$fibo(2) = fibo(1) + fibo(0) = fibo(0) + fibo(-1) + fibo(-1) + fibo(-2)$$
$$= fibo(-1) + fibo(-2) + fibo(-2) + fibo(-3)$$
$$+fibo(-2) + fibo(-3) + fibo(-3) + fibo(-4) = etc.$$

irgendwo muss das Ergebnis für ein oder mehrere n explizit angeben werden (häufig bei $n = 0$). Im Fall der Fibonacci Zahlen ist $fibo(0) = 0, fibo(1) = 1$. Hier die ersten 20 Fibonacci Zahlen:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181$$

Wie man sieht ist jede Fibonacci Zahl gleich der Summe der beiden vorigen Fibonacci Zahlen.

```
fibo(0,0).
fibo(1,1).
fibo(N,Erg):-
   N1 is N-1,         /* Parameter fuer die Rekusrsionen */
   N2 is N-2,
   fibo(N1,Erg1),     /* Rekursionen ausfuehren */
   fibo(N2,Erg2),
   Erg is Erg1+Erg2.  /* Ergebnisse summieren */
```

Nachdem Prolog allerdings z.B. fibo(1,Erg) zu Erg=1 ausgewertet hätte, würde es ja auch die nächste Klausel ausprobieren, was (nur in Prolog - nicht in Haskell oder imperativen Sprachen!) immernoch zu einer Endlosrekursion führen würde, obwohl die Fälle $N = 0$ und $N = 1$ explizit angegeben sind. Um diese Endlosrekursion zu verhindern, verlangen wir, dass die letzte Klausel nicht verwendet wird, wenn schon eine der anderen Fälle ($N \leq 1$) eingetreten war.

```
fibo(0,0).
fibo(1,1).
fibo(N,Erg):-
   N > 1,
   N1 is N-1,
   N2 is N-2,
   fibo(N1,Erg1),
   fibo(N2,Erg2),
   Erg is Erg1+Erg2.
```

Wenn man mit logischen Programmiersprachen arbeitet, sollte man sich angewöhnen, rekursiv zu denken d.h. zu versuchen, Probleme zu zerlegen in einen trivialen Fall (der Rekursionsabbruch) und einen Fall, den man rekursiv beschreiben kann - d.h. auf einen Fall zurückzuführen, der „näher am trivialen Fall liegt“.

Als Beispiel wollen wir ein Prädikat betrachten, das Vorfahren bestimmt. Für die Einfachheit erlauben wir es, dass ein Mensch sein eigener Vorfahre ist.

```
vorfahre(Person,Person).
vorfahre(Vorf,Person):-
   elternteil(Temp,Person),
   vorfahre(Vorf,Temp).
```

Ein Testlauf $vorfahre(X, hermann)$. liefert: X=*hermann; X = irmtraut ; X = wilhelm ; X = hans ;*

Dieses Prädikat sagt aus, dass ein Mensch V genau dann ein Vorfahre eines Menschen P ist, wenn $V = P$ ist oder wenn V ein Vorfahre eines Elternteils von P ist. (dadurch, dass in diesem Beispiel jeder Mensch sein eigener Vorfahre ist, werden so auch die Elternteile eines Menschen als seine Vorfahren erkannt).

Ein weiteres Beispiel für rekursive Prädikate ist die Fakultät einer natürlichen Zahl:

$$n! = \prod_{i=1}^{n} i = n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1$$

Rekursiv lässt sich diese Funktion folgendermaßen beschreiben:

$$1! = 1, n! = n * (n-1)!$$

```
fakultaet(1,1).
fakultaet(N,Ergebnis):-
   N > 1,
   N1 is N-1,
   fakultaet(N1,N1fakultaet),
   Ergebnis is N*N1fakultaet.
```

Einige weitere Beispiele (die zwar Trivial sind, aber Rekursionen gut veranschaulichen):

- $a^b = \underbrace{a^{b-1}}_{Rekursion} \cdot a$ (Rekursionsabbruch: $a^0 = 1$)
- $a \cdot b = \underbrace{a \cdot (b-1)}_{Rekursion} + a$ (Rekursionsabbruch: $a \cdot 0 = 0$)
- $a + b = \underbrace{a + (b-1)}_{Rekursion} + 1$ (Rekursionsabbruch: $a + 0 = a$)

In diesen Beispielen wird die Operation ausgeführt, indem die Operation mit einem kleineren b ausgeführt wird (die Rekursion) und das Ergebnis weiter bearbeitet wird (1 wird dazu addiert, a wird dazu addiert, a wird dazu multipliziert)
Übung:

- Programmieren Sie die drei trivialen Beispiele (für a^b, $a \cdot b$ und $a + b$)
- Das Problem der *Türme von Hanoi* ist folgendes: man hat drei Stäbe, wobei auf dem ersten n Scheiben mit einem Loch in der Mitte liegen, deren Größe von oben nach unten zunimmt. Man soll nun alle Scheiben auf den zweiten Stab bewegen, allerdings darf man immer nur eine Scheibe auf einmal bewegen und es darf niemals eine Scheibe über einer kleineren Scheibe liegen. (Nein, man darf sie auch nicht neben den Stäben ablegen!)

- Überlegen Sie, wie man dieses Problem rekursiv lösen kann (überlegen Sie, was es bringt, $(n-1)$ Scheiben von A nach C zu bewegen, wenn man n Scheiben von A nach B bewegen will)
- Schreiben Sie ein Programm, das der Reihe nach angibt, von welchem Stab zu welchem Stab eine Scheibe verlegt werden muss. *Hinweis:* Das Prädikat soll die Parameter *hanoi(N,Von,Nach,Ueber)* besitzen.
- Rekursive Programme legen häufig einen Beweis (durch vollständige Induktion) für ihre Korrektheit nahe. Beweisen sie, dass das Problem der *Türme von Hanoi* für jedes n lösbar ist.
- Wie viele Verlege-Operationen sind für 3,4 und 5 Scheiben nötig? (Zählen Sie die Zeilen, die ihr Programm ausgegeben hat)
- Wie viele Verlege-Operationen sind für n Scheiben nötig? Geben Sie eine rekursive und eine explizite Gleichung an (Hinweis für die explizite Gleichung: Betrachten Sie für 3,4 und 5 die nötigen Anzahlen + 1)

2.16 Endrekursion

Man unterscheidet zwei Typen von Rekursionen: normale Rekursion und Endrekursion. Der Unterschied ist, dass eine normal rekursive Funktion sich selbst aufruft und das Ergebnis weiterverwendet. Eine endrekursive Funktion hingegen berechnet zuerst etwas, das sie beim Rekursionsaufruf als Parameter übergibt und danach nurnoch das Ergebnis des Rekursionsaufrufs zurückgibt. Zum Beispiel die Fakultät:

```
fakultaet2(0,GesamtProdukt,GesamtProdukt).
fakultaet2(N,Teilprodukt,GesamtProdukt):-
   N > 0,
   NaechstesTeilprodukt is N*Teilprodukt,
   N1 is N-1,
   fakultaet2(N1,NaechstesTeilprodukt,GesamtProdukt).
              /* parameter uebergeben, Ergebnis weiterleiten */
```

Teilprodukt ist das Produkt der Zahlen von $N+1$ bis n (dem vom Benutzer übergebenen N). sobald $N=0$ erreicht ist, enthält *Teilprodukt* also das Produkt $1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n$ was gleich $n!$ ist, daher wird Teilprodukt nun als *GesamtProdukt* zurückgeliefert und in den höheren Rekursionsstufen nurnoch weitergeleitet.
Übung:

- Schreiben sie ein Prädikat, das die Fibonaccizahlen endrekursiv berechnet! Hinweis: Es soll die Parameter $fibo2(N, A, B, Ergebnis)$ haben und aufgerufen werden z.B. mit *fibo2(10,1,1,Ergebnis).* um die zehnte Fibonacci Zahl zu berechnen. A und B sind in der i-ten Rekursionstiefe Zwischenergebnisse, die $fibo(i-2)$ und $fibo(i-1)$ enthalten.
- Testen Sie fibo und fibo2 für größere Zahlen (Größenordnung 40-60)! was fällt Ihnen auf?

Als letztes Beispiel wollen wir den größten gemeinsamen Teiler (ggT) von zwei Zahlen A und B ermitteln. Die naive Methode wäre, alle Zahlen von

$min\{A, B\}$ bis 1 durchzugehen und die erste Zahl G, die beide Zahlen teilt ($A \; mod \; G = 0$ und $B \; mod \; G = 0$), auszugeben. Es gibt aber auch den Euklidischen Algorithmus, der auf Euklids Erkenntnis basiert, dass

$$ggt(a, b) = ggt(b, a \; mod \; b) \text{ und } ggt(a, 0) = a$$

gilt. Als Prolog-Code:

```
ggt(A,0,A).
ggt(A,B,C):-
   B > 0,
   AMODB is A mod B, /* Parameter bestimmen */
   ggt(B,AMODB,C).
            /* Ergebnis weiterleiten */
```

Das kleinste gemeinsame Vielfache (kgV) von a und b ist übrigens $= \frac{a \cdot b}{ggT(a,b)}$. Interessanterweise sind der *worst-case*[5] für diesen Algorithmus zwei aufeinanderfolgende Fibonacci Zahlen (in jedem Schritt ist der Divisionsrest die nächst kleinere Fibonacci Zahl)

Fachchinesisch: im worst-case hat der euklidische Algorithmus eine Laufzeit, die proportional zu $log(A \cdot B)^2$ ist. $log(A \cdot B)$ ist proportional zur Anzahl der Ziffern der größeren Zahl.

2.17 Arithmetische Ausdrücke als Parameter

Auf den ersten Blick scheint es so, als könnte man keine arithmetischen Ausdrücke als Parameter an Prädikate übergeben, was so aussehen könnte, als läge es daran, dass die Parameter per Call-by-Reference übergeben werden (auch Compiler von imperativen Sprachen geben Warnungen oder Fehler aus, wenn man arithmetische Ausdrücke für Call-by-Reference Parameter übergibt). Oft liest man in Anleitungen, dass man immer erst eine temporäre Variable an den arithmetischen Ausdruck binden und diesen dann per Call by Reference übergeben **muss**

```
/*
angeblich           angeblich einzige
unmoeglich          Moeglichkeit
*/
fibo(0,0).          fibo(0,0).
fibo(1,1).          fibo(1,1).
fibo(N,F):-         fibo(N,F):-
   N > 1,              N > 1,
   fibo(N-1,F1),       N1 is N-1,
   fibo(N-2,F2),       N2 is N-2,
   F is F1+F2.         fibo(N1,F1),
                       fibo(N2,F2),
                       F is F1+F2.
```

Das Problem an arithmetischen Ausdrücken ist aber eigentlich die Tatsache, dass die Parameter mittels = an die übergebenen Werte gebunden werden d.h.

[5] Die „schlimmste“ Eingabe. Eine Eingabe bei der der Algorithmus seine maximale Laufzeit benötigt

dass der Parameter dann nicht mit einer Zahl gebunden wird, sondern mit einem Ausdruck (vgl 2.9.1). Wenn man diesen Ausdruck nachträglich mittels *is* oder =:= auswertet, kann man doch arithmetische Ausdrücke übergeben:

```
fibo(A,0):- A =:= 0.
fibo(A,1):- A =:= 1.
fibo(N,F):-
   M is N,
   M > 1,
   fibo(M-1,F1),
   fibo(M-2,F2),
   F is F1+F2.
```

Man kann sogar im Kopf des Prädikats die Summe zusammensetzen:

```
fibo(A,0):- 0 is A.
fibo(A,1):- 1 is A.
fibo(N,F1+F2):-
   M is N,
   M > 1,
   fibo(M-1,F1),
   fibo(M-2,F2).
```

Allerdings ist das Ergebnis dann selbst ein arithmetischer Ausdruck

$$fibo(6, F). \Rightarrow F = 1 + 0 + 1 + (1 + 0) + (1 + 0 + 1) + (1 + 0 + 1 + (1 + 0))$$

der im Fall der Fibonaccizahlen sogar für recht kleine Parameter sehr lang würde, was viel RAM verbraucht und Laufzeit kostet.

3 Wie Prolog „denkt“

Wir haben bisher logische Regeln formuliert und Prolog Fragen gestellt, die es uns meistens beantworten konnte. Selbstverständlich „zaubert“ Prolog die Ergebnisse nicht herbei, sondern berechnet sie nach einem Schema (das „SLD Resolution“ genannt wird). Dieses Schema wollen wir nun näher beleuchten.

Intern funktioniert ein Prädikat fast genauso wie eine Funktion/Methode mit Boo'lschem Rückgabetyp (*bool*, *boolean*) in imperativen Programmiersprachen. Die mit AND verknüpften Bedingungen werden nach und nach Ausgewertet - so, wie die Anweisungen in imperativen Sprachen nach und nach aufgerufen werden, ABER sobald eine der Bedingungen No ergibt, wird das Prädikat beendet und No zurückgegeben (ausser wenn das No in einem Negierten Block vorkommt o.Ä.)
Der **wesentliche** Unterschied zu imperativen Programmiersprachen ist: falls ein Aufruf auf mehrere Klauseln passt oder ein OR (Semikolon) in den Bedingungen vorkommt - sprich wenn auf mehrere verschiedene Wege weiter gerechnet werden könnte, dann meldet der Compiler weder einen Fehler (wie es bei imperativen Sprachen gemacht würde), noch wählt es einfach eine bestimmte Klausel (wie z.B. Haskell den ersten passenden Funktionskopf wählt), sondern

Es probiert alle Möglichkeiten nach und nach aus

Dieses Verhalten lässt sich sehr anschaulich durch sog. SLD Bäume darstellen:

3.1 SLD Bäume

SLD Bäume sind so etwas ähnliches wie ein Programmfluss in imperativen Programmiersprachen - Bei einem Programmfluss in einer imperativen Sprache ist ein Name einer Prozedur/Funktion/Methode unterhalb einer anderen, wenn sie direkt nach ihr ausgewertet/ausgeführt wird. Da Prolog für Prädikat-Aufrufe alle passenden Klauseln ausprobiert, könnte die nächste „Zeile“ nicht eindeutig angegeben werden - statt dessen verzweigt sich ein SLD Baum, wenn zu einem Prädikat-Aufruf mehrere Klauseln passen (wobei in jedem der Zweige eine andere Klausel verwendet wird - die Klausel, die im Quellcode als oberste zu dem Aufruf passt, entspricht im SLD Baum dem linkesten Zweig; die Klausel, die im Quellcode als zweit-oberste passt, entspricht im SLD Baum dem zweiten Zweig von links usw.)

Abbildung 1 zeigt den SLD Baum zur Anfrage $vorfahre(X, hermann)$. (wobei die Namen der Personen und der Prädikate abgekürzt sind).

An den Blättern von SLD Bäumen stehen entweder gültige Lösungen für das angegebene Goal oder aber $fail$, was signalisiert, dass die Berechnung zu diesem Zeitpunkt fehlschlägt (normalerweise deshalb, weil für einen Aufruf keine passende Klausel gefunden wurde). Wenn man die Blätter (an denen nicht $fail$ steht) von Links nach Rechts durchgeht, erhält man die Lösungen, die Prolog liefert und zwar **in der gleichen Reihenfolge**!

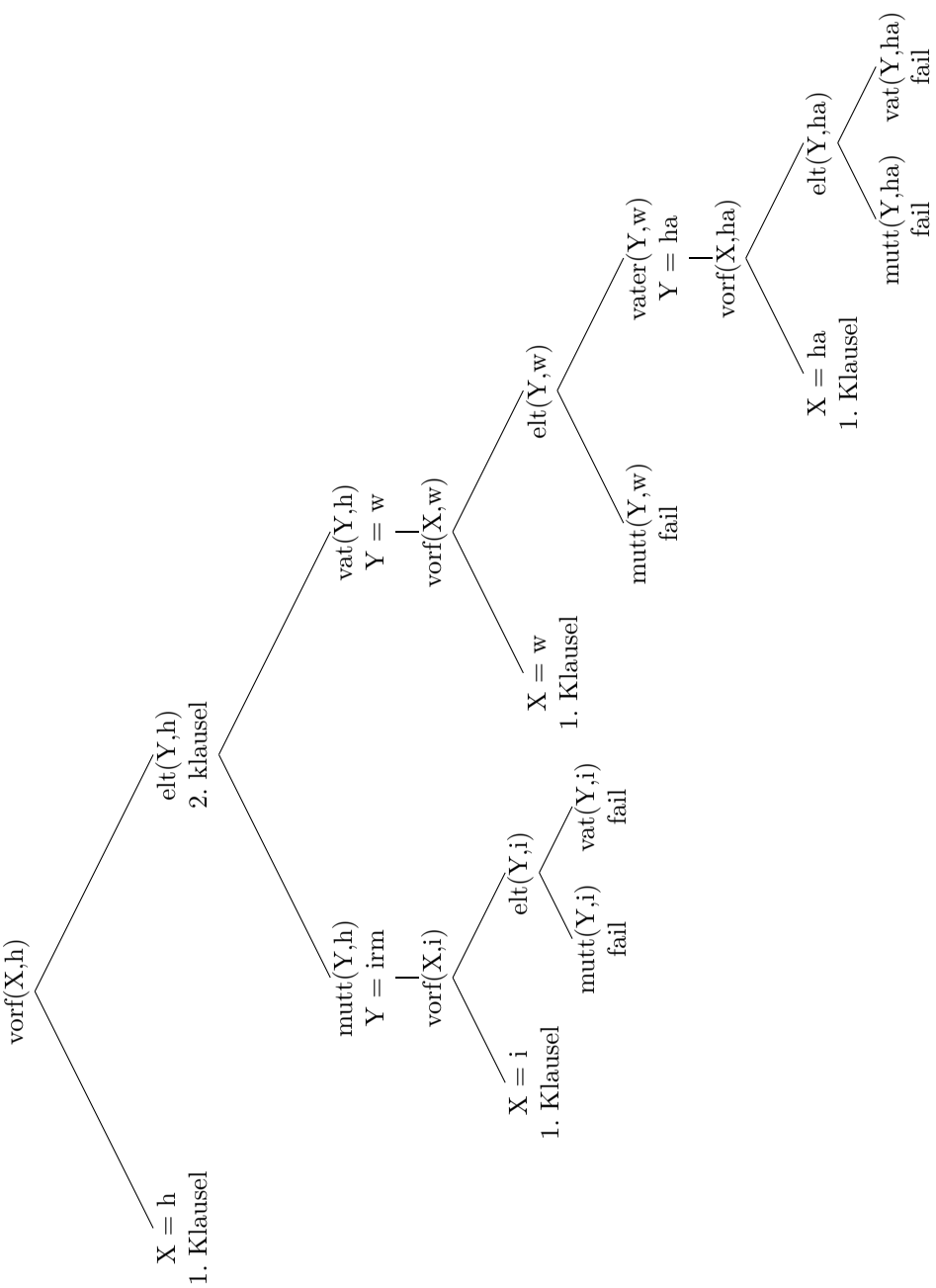


Abbildung 1: SLD Baum zur Anfrage *vorfahre(X,hermann)*.

Hintergrundwissen in Fachchinesisch:
Die Bezeichnung „SLD“ steht für ***L****inear resolution with* ***S****election function on* ***D****efinite clauses* (Sie können sich selbst überlegen, warum dieser Name nicht mit „LSD-Resolution“ abgekürzt wird). SLD Resolution ist eine spezielle Form der ***prädikatenlogischen Resolution***, die Sie in Vorlesungen über mathematische Logik kennen lernen werden. Die SLD Resolution ist deutlich schneller als die prädikatenlogische Resolution, hat aber den Nachteil, dass sie nur mit ***Horn-Formeln*** funktioniert. Horn-Formeln haben die Form

$$a(p_1, ..., p_n) \vee \neg b_1 \vee \neg b_2 \vee ... \vee \neg b_m$$

was logisch das gleiche ist, wie

$$a(p_1, ..., p_n) \Leftarrow b_1 \wedge b_2 \wedge ... \wedge b_m$$

oder in Prolog-Syntax:

$$a(p_1,...,p_n) \text{ :- } b_1 \text{ , } b_2 \text{ , ... , } b_m \text{ .}$$

solche Horn-Formeln nennt man ***Definite Klauseln***, der zweite Typ von Horn-Formeln wird als ***Negative Klauseln*** bezeichnet und entspricht Anfragen in Prolog.
zwei Links zu dem Thema:

$$http://de.wikipedia.org/wiki/Horn-Formel$$

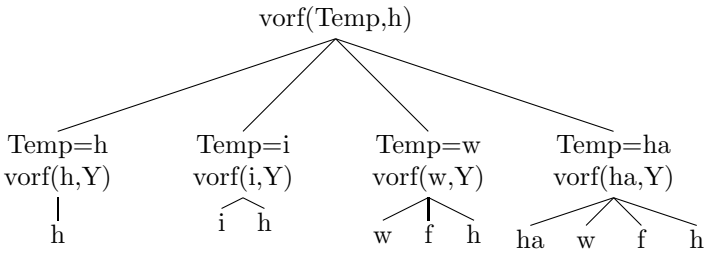
$$http://de.wikipedia.org/wiki/Resolution_(Logik)$$

3.2 SLD Bäume vereinfachen

Es ist meistens nicht sinnvoll, komplette SLD Bäume zu betrachten, weil man meistens nur das Verhalten eines einzelnen Prädikats analysieren will. Man kann in einem SLD Baum auch Aufrufe von anderen Prädikaten weglassen und sie durch einen **einzelnen Knoten** ersetzen, aus dem **nur die Lösungen** des weggelassenen Prädikats als Kanten herausführen - Betrachten wir z.B. das Prädikat

```
verwandt(X,Y):-
   vorfahre(Temp,X),
   vorfahre(Temp,Y).
```

mit dem Aufruf $verwandt(hermann, X)$. Hier werden nach und nach Vorfahren von *hermann* bestimmt ($Temp$) und dann werden die Nachfahren von $Temp$ gesucht (wobei die Ergebnisse bei dieser Implementierung häufiger vorkommen werden, weil Nachfahren einer Person auch Nachfahren seiner Vorfahren sind). Hier interessiert es uns z.B. nicht, wie die Vorfahren berechnet werden (Abbildung 1 sollte uns davon abschrecken), also können wir diese Berechnungen zu je einem Knoten zusammenfassen:



3.3 Wann Variablen gebunden sind (TODO)

3.4 Was Prolog berechnen kann (TODO)

Für erfahrene Informatiker ist die Frage, was Prolog berechnen kann, sicher unsinnig, denn Prolog ist selbstverständlich (wie jede praktisch anwendbare Programmiersprache) Turing-Vollständig, man kann damit also alles Berechnen, was (Turing-)Berechenbar ist - so ist die Frage allerdings nicht gemeint, sondern in diesem Abschnitt wollen wir beleuchten, welche Parameter in einem Prädikat gebunden werden können

3.5 Prolog Programme designen (logisch) (TODO)

Wenn man den logischen Ansatz wählt, um Prolog Prädikate zu designen, dann ist es sinnvoll, davon auszugehen, dass der Benutzer alle Parameter übergibt und man nurnoch prüfen muss, ob die Belegungen eine gültige Lösung darstellen - danach kann man analysieren, welche Parameter gebunden sein müssen, um andere Parameter bestimmen zu können (siehe 3.4)
Beispiel:

3.6 Prolog Programme designen (quasi-imperativ) (TODO)

Wenn man quasi-imperativ in Prolog programmieren will, dann schreibt man seine Klauseln meist so, dass die Bedingungen ausschließlich durch AND verknüpft sind, sodass die „Anweisungen“ (die eigentlich immernoch Bedingungen sind) in der Reihenfolge abgearbeitet werden, in der man sie hinschreibt - man sollte dann dafür sorgen, dass seine Prädikate immer *true* liefern, damit die imperativ designeten Prädikate, die dieses Prädikat verwenden, nicht einfach terminieren.
Man sollte bei jedem Aufruf eines anderen Prädikats daran denken, dass Prolog über alle Lösungen dieses Prädikates verzweigt, daher kann man nach einem Aufruf eines Prädikats davon ausgehen, dass die Parameter des aufgerufenen Prädikats, die man ungebunden gelassen hatte, nun gebunden sind und zwar mit IRGENDEINER Lösung.

3.7 Den Programmfluss manipulieren

3.7.1 Cuts

Ein ***Cut*** erzwingt die Beendigung der Suche nach alternativen Beweiswegen - anschaulich heisst das, dass Äste vom SLD Baum abgeschnitten werden. Einen

Cut führt man durch ein Ausrufezeichen ! aus. Zum Beispiel

```
cuttest(1).
cuttest(2):-!.
cuttest(3).
```

Der Aufruf *cuttest*(X). liefert $X = 1$ und $X = 2$ - die dritte Klausel wurde wegen dem Cut nicht mehr ausprobiert.

Weiterführende Information:
Man bezeichnet einen Cut als „roten Cut“, „grünen Cut“ , oder „blauen Cut“ jenachdem, wo er ausgeführt wird und welchen Einfluss er dadurch auf den Programmfluss hat. Nähere Informationen dazu:

http://www.ling.uni-potsdam.de/kurse/Prolog/11_Cut.pdf

Die Frage, die sich nun stellt ist, welche Äste genau abgeschnitten werden. Äste die im (nicht vereinfachten!) SLD Baum Nachfolger des Cuts sind, bleiben unberührt:

```
cuttest(A):- !, foo(A).
foo(1).
foo(2).

?- cuttest(A).
A=1;
A=2;
fail.
```

Auch Äste, die oberhalb des Aufrufs des aktuellen Prädikats liegen, bleiben unberührt (andernfalls käme ein Cut in den meisten Fällen ja einem Abbruch des gesamten Goals gleich!)

```
cuttest2(1,A):-
   foo(A).
cuttest2(2,A):-
   foo(A).
foo(1):-!.
foo(2).

?- cuttest2(X,Y).
X=1, Y=1;
X=2, Y=1;
fail.
```

Der Aufruf *cuttest*$2(X, Y)$ liefert also auch $X = 2, Y = 1$ d.h. die zweite Klausel des *cuttest* Prädikats ist noch ausprobiert worden, obwohl bei der Berechnung von $X = 1, Y = 1$ im foo Prädikat ein Cut ausgeführt wurde. Die Suche nach Alternativen wurde also nur für das foo Prädikat beendet.

3.7.2 Cuts in rekursiven Prädikaten

Die Frage, welche Auswirkung ein Cut bei rekursiven Prädikaten hat ist noch offen. Der Folgende Code entspricht im Wesentlichen dem letzten Code, nur dass

das foo Prädikat den selben Namen wie das andere Prädikat bekommen hat - der erste Parameter sorgt hier dafür, dass in der Rekursion nur die hinteren beiden Klauseln ausprobiert werden (sodass $cuttest3(3, X)$ praktisch das selbe ist, wie zuvor $foo(X)$)

```
cuttest3(1,A):-
   cuttest3(3,A).
cuttest(2,A):-
   cuttest3(3,A).
cuttest3(3,1):-!.
cuttest3(3,2).
```

Der Aufruf $cuttest3(A, B)$ liefert sowohl $A = 1, B = 1$, als auch $A = 2, B = 1$, also ist die zweite Klausel des $cuttest3$ Prädikats ausprobiert worden, obwohl in einem $cuttest3$ Aufruf zuvor ein Cut ausgeführt wurde. Es wurde also nur die Rekursion abgebrochen. Der Aufruf liefert auch noch $A = 3, B = 1$ - die Lösung mit der dritten Klausel, aber da dort wiederum ein Cut ausgeführt wird, wird die Lösung $A = 3, B = 2$ nicht berechnet.

Merke: Wenn in einer Klausel K ein Cut ausgeführt wird, dann werden im SLD Baum **die Geschwister** des Knotens K, die weiter rechts als K stehen, abgeschnitten.

3.7.3 Cuts zwischen anderen Bedingungen

Ein Cut muss nicht die einzige Bedingung einer Klausel sein

```
praedikat(A):-
   a(A,B),
   b(B,C),
   !,
   c(A,B),
   d(C,D).
praedikat(0).
```

Prolog verhält sich bei einem solchen Cut folgendermaßen: es versucht, Belegungen von A,B und C zu finden, die die Bedingungen a und b erfüllen. Wird **EINE** solche Belegung gefunden, dann erreicht Prolog die Zeile mit dem Cut (bei allen vorherigen Versuchen kam es nicht so weit, weil a oder b nicht erfüllt waren) - dann wird also der Cut ausgeführt, d.h. es werden keine weiteren Belegungen für A,B und C gesucht, sondern c und d werden mit diesen festen A,B und C behandelt. Wenn keine passende Belegung von A,B,C gefunden wurde, wird noch die Klausel $praedikat(A, A, A, A)$ ausprobiert, weil Prolog dann in der ersten Klausel nicht bis zum Cut gekommen ist.
Übung:

- Was passiert beim Beispiel $praedikat(A, B, C, D)$ wenn mehrere Lösungen für $d(C, D)$, möglich sind? verzweigt Prolog wieder, oder nicht? Was erwarten Sie? Schreiben Sie ein Programm, um ihre Vermutung zu überprüfen!

3.7.4 Gefahren von Cuts (TODO)

3.7.5 fail

Schreibt man in den Rumpf einer Klausel die Bedingung „fail“, dann bricht Prolog (sofern es an an diese Stelle kommt) die Berechnung der Klausel ab und betrachtet sie als nicht erfüllt. Das klingt nutzlos, denn was nützt es, das Scheitern einer Anfrage zu erzwingen, wenn die nachfolgenden Klauseln doch ohnehin noch ausprobiert werden und man die Klausel, die das $fail$ enthält auch einfach weglassen könnte, damit es nicht als erfüllt betrachtet wird? Tatsächlich wird das $fail$ in der Praxis meistens im Zusammenhang mit einem cut ausgeführt

```
natuerliche_zahlen(X,X).
natuerliche_zahlen(Y,X):-
   Y2 is Y+1,
   natuerliche_zahlen(Y2,X).
```

liefert beim Aufruf $natuerliche\_zahlen(1, X)$ die Ergebnisse $X = 1$, $X = 2$, $X = 3$, $X = 4$,... wollen wir maximal $X = Z$, können wir dies z.B. durch ein $cut - and - fail$ erreichen

```
natuerliche_zahlen(X,_,X).
natuerliche_zahlen(X,Z,_):-
   X > Z,
   !,
   fail.
natuerliche_zahlen(Y,Z,X):-
   Y2 is Y+1,
   natuerliche_zahlen(Y2,Z,X).
```

3.7.6 fail-getriebene Fortsetzung

Man kann das fail auch dazu benutzen, eine Art Schleife nachzubilden.

```
t(a).
t(b).
t(c).
t_ausgabe :-
   t(Symbol),
   write(Symbol),
   write(' '),
   fail.
```

Die Anfrage $t\_ausgabe.$ lässt Prolog $a\ b\ c\ No.$ Antworten.[6]
Diese Methode ist aber nur für Ausgaben geeignet, denn das $fail$ kann nicht die bereits ausggegebenen Zeichen wieder vom Bildschirm löschen, daher werden alle Zeichen ausgegeben, obwohl das Prädikat niemals erfüllt wird - daher funktioniert die Ausgabe, aber jede Berechnung eines Ergebnisses mit dieser Methode würde wieder abgebrochen. z.B. würde

[6] Will man lieber ein Yes als Antwort, kann man noch die Klausel $t\_ausgabe.$ hinzufügen - dadurch wird $t\_ausgabe$ nachträglich akzeptiert.

```
elternteil(Elternteil,Kind):-
   (   vater(Elternteil,Kind);
       mutter(Elternteil,Kind)   ),
   write(Elternteil),write(','),write(Kind),nl,
   fail.
```

alle Kinder und ihre Eltern ausgeben, aber damit würde das $vorfahre$ Prädikat

```
vorfahre(Vorfahre, Von):-
   elternteil(Vorfahre,Von).
vorfahre(Vorfahre, Von):-
   elternteil(Elternteil,Von),
   vorfahre(Vorfahre,Elternteil).
```

nurnoch (je zwei mal) ausgeben, wer wessen Elternteil ist, diese Information aber nicht weiterverwenden um rekursiv Vorfahren der Eltern zu bestimmen, denn das $elternteil(Elternteil, Von)$, Prädikat wird nicht als erfüllt betrachtet, daher wird $vorfahre(Vorfahre, Elternteil)$. nicht mehr aufgerufen.

Insgesamt kann man sagen, dass diese Methode zur Ausgabe von Ergebnissen geeignet ist, aber nur in Prädikaten angewendet werden sollte, bei denen man davon ausgehen kann, dass keine anderen Prädikate sie jemals verwenden werden, zum Beispiel in Goals.

4 Listen

Listen sind in Prolog dazu gut, auch weniger statisches Wissen zu repräsentieren (weil sie eine variable Länge haben können). Wenn man eine komplette Liste angeben will (z.B. als Parameter für ein Goal), dann schreibt man die Elemente nacheinander auf, getrennt durch Kommata und umgeben von eckigen Klammern, z.B. *[1,2,3,4,5]* die leere Liste beschreibt man folgerichtig mit *[ ]*
Wenn man Prädikate schreibt, die mit beliebigen Listen arbeiten, kann man selbstverständlich nicht für jede Liste oder für jede Länge einer Liste (durch Verwendung von Variablen) vorher eine passende Klausel schreiben. Prädikate, die mit Listen arbeiten, tun dies meist rekursiv, indem sie das erste Element (oder mehrere Elemente am Anfang der Liste) abkoppeln, die Restliste rekursiv bearbeiten und dann die Elemente vom Anfang in irgend einer Weise mit dem Ergebnis des Rekursionsaufrufs kombinieren. Dieses „Abkoppeln“ funktioniert mit dem Symbol | (vertikaler Strich). Wenn man z.B. im Kopf eines Prädikats [Element1,Element2| Rest] angibt und dann im Goal [1,2,3,4,5] übergibt, dann gilt im Rumpf des Prädikats: Element1=1, Element2=2, Rest=[3,4,5]. Der Rest ist also selbst wieder eine Liste.

4.1 Elemente suchen

Wir wollen prüfen ob ein gegebenes Element in einer Liste vorhanden ist - dieses Prädikat nennen wir *member*. Der Gedanke dahinter ist: ein Element ist genau dann in einer Liste vorhanden, wenn es am Anfang steht ODER vorhanden ist in der Liste, die entsteht, wenn man das erste Element abkoppelt:

```
member(Element, [Anfang|Rest]):-
   Anfang = Element.        /* Element ist gleich dem Anfang */
member(Element, Liste):-
   [Anfang|Rest] = Liste,
   member(Element,Rest).        /* Element ist in der Restliste vorhanden */
```

dadurch, dass Variablen ja schon auf Gleichheit überpruft werden, wenn man ihnen den gleichen Namen gibt, und dadurch, dass man Listen schon in den Köpfen der Prädikate zerlegen kann, können wir uns direkt 2 Zeilen sparen:

```
member(Element, [Element|Rest]).   /* Element ist gleich dem Anfang */
member(Element, [Anfang|Rest]):-
   member(Element,Rest).            /* Element ist in der Restliste vorhanden */
```

Nun ist in der ersten Klausel ja egal, wie die Restliste aussieht, denn die entscheidende Information (dass das Element in der Liste vorhanden ist) ist ja jetzt schon bekannt. Ebenso ist in der zweiten Klausel egal, was der Anfang der Liste ist, denn es wird ja nur der Rest weiter betrachtet

```
member(Element, [Element|_]).    /* Element ist gleich dem Anfang */
member(Element, [_|Rest]):-
   member(Element,Rest).           /* Element ist in der Restliste vorhanden */
```

Übrigens: wenn wir den *Element* Parameter ungebunden lassen, dann verzweigt Prolog an dieser Stelle und in jedem der Berechnungs-Zweige ist eins der Elemente der Liste an *Element* gebunden (wobei im ersten Zweig *Element* das

erste Element der Liste ist, im zweiten Zweig ist *Element* das zweite Element der Liste etc.
Damit bewirkt das Goal $member(Element, [1,2,3,4,5]), write(Element), write(','), fail$[7] die Ausgabe $1,2,3,4,5$, *No*.

Dieses *member* ist in allen mir bekannten Prolog Interpretern vordefiniert, eine sinnvolle Erweiterung ist aber, einen Integer mitzuführen, der 0 ist, wenn das Element gefunden wird und in den höheren Rekursionsstufen jeweils um 1 erhöht wird

```
member(Element, 0, [Element|_]).
member(Element, N, [_|Rest]):-
   member(Element, N1, Rest),
   N is N1+1.
```

Damit erfährt man nun auch, an welcher Position in der Liste das gesuchte Element steht, bzw. wenn man den ersten Parameter ungebunden lässt und für den zweiten Parameter einen Index übergibt, erfährt man, welches Element an diesem Index in der Liste steht

```
?- member(5, Pos, [7,5,3,4]).
Pos = 1. % Das erste Element hat den Index 0

?- member(X, 2, [7,5,3,4]).
X = 3.
```

Übung:

- Schreiben Sie ein Prädikat, das die Länge einer Liste berechnet.
- Warum ist die folgende Lösung schlecht?

  ```
  length([],0).
  length([_|Rest],N):-
     N1 is N-1,
     length(Rest,N1).
  ```

- Schreiben Sie ein Prädikat, das die Summe der Elemente einer Liste berechnet. Hinweis: die erste Klausel lautet *summe([ ],0).*

4.2 Elemente löschen und einfügen

Wir wollen ein Element aus einer Liste löschen. Das tun wir aber nicht, indem wir irgendwelche Pointer verändern, wie in imperativen Sprachen, sondern wir schreiben ein Prädikat, das zu einer gegebenen Liste und einem zu löschenden Element eine neue Liste berechnet, die der gegebenen Liste mit entferntem gegebenen Element entspricht. Das funktioniert so, dass wir (nach Kochrezept) das erste Element der Liste abspalten. Entspricht es dem gesuchten Element, geben wir die Restliste zurück, ansonsten löschen wir das gesuchte Element aus der Restliste und hängen an das Ergebnis der Rekursion das erste Element der ursprünglichen Liste wieder an.

[7] vgl 3.7.6

```
delete(Element, [Element|Rest], Rest).
delete(Element, [Anfang|Rest], Ergebnis):-
   delete(Element, Rest, Liste2),
   Ergebnis = [Anfang|Liste2].
```

Interessanterweise kann man im Kopf eines Prädikats nicht nur Elemente von Listen abspalten, sondern sie sogar anhängen (selbst, wenn die Restliste noch gar nicht gebunden ist!)

```
delete(Element, [Element|Rest], Rest).
delete(Element, [Anfang|Rest], [Anfang|Liste2]):-
   delete(Element, Rest, Liste2).
```

Wenn wir noch einen Integer in diesem Prädikat mitführen, der beim Auffinden des Elements auf 0 gesetzt wird und in den höheren Rekursionsstufen jeweils erhöht wird

```
delete(Element, 0, [Element|Rest], Rest).
delete(Element, N, [Anfang|Rest], [Anfang|Liste2]):-
   delete(Element, N1, Rest, Liste2),
   N is N1+1.
```

dann können wir dieses Prädikat für die verschiedensten Aufgaben zweckentfremden (je nachdem, welche Parameter wir ungebunden lassen). Lassen wir z.B. N ungebunden, erfahren wir, an welcher Position das gelöschte Element stand. Binden wir N und lassen statt dessen *Element* ungebunden, wird das Element an der N-ten Position gelöscht[8] und *Element* mit dem gelöschten Element gebunden. Das ist vielleicht noch nicht spektakulär, aber wenn wir die die Ursprungsliste ungebunden lassen und dafür die Ergebnisliste und sowohl N als auch *Element* binden, dann wird an die Variable, die wir für die Ursprungsliste übergeben haben, diejenige Liste gebunden, die entsteht, wenn man in die (als Ergebnisliste gebundene) Liste das *Element* an der Position N **einfügt**!

```
?- delete(Elem, 2, [1,2,3,4,5], Erg).
Elem=3,
Erg=[1,2,4,5].

?- delete(7,3,Liste,[1,2,3,4,5,6,7,8]).
Liste=[1,2,3,7,4,5,6,7,8].
```

Übung:

- Wie werden K und Erg gebunden beim Aufruf $delete(9, K, Erg, [1, 2, 3, 4])$.? Hinweis: es gibt mehrere Lösungen, Prolog verzweigt hier also, wobei in jedem Zweig eine andere Lösung angenommen wird. Wenn Sie die Lösung nicht selbst herausfinden, lesen Sie die Musterlösung, denn es handelt sich hier um eine sehr fundamentale und sehr nützliche Zweckentfremdung des *delete* Prädikats!

[8] wobei das erste Element die Position 0 hat

4.3 Listen aneinanderhängen und zerlegen, Teil-Listen suchen

Wir wollen eine Liste an eine andere anhängen (*append*). Wir gehen wieder nach dem Kochrezept vor, erstes Element abkoppeln, Reste rekursiv bearbeiten, Ergebnis weiterverwerten: Wenn die erste Liste leer ist, ist das Ergebnis des „Anhängens“ gleich der zweiten Liste. Ansonsten wird das erste Element der ersten Liste abgekoppelt und vorne an das Ergebnis der Rekursion angehangen, wobei das Ergebnis der Rekursion die Liste ist, die durch Zusammenfügen der Restliste mit der zweiten Liste entsteht.

```
append([], Liste2, Liste2).
append([Anfang|Rest], Liste2, [Anfang|Rest2]):-
   append(Rest,Liste2,Rest2).
```

Dieses Prädikat lässt sich zweckentfremden, um zu prüfen, ob der Anfang einer Liste A gleich einer Liste B ist, denn B ist der Anfang von A, wenn eine Liste existiert, die an B angehängt A ergibt.

```
?- append([1,2,3],_ ,[1,2,3,4,5,6]).
true.
```

Dieses *append* ist in allen mir bekannten Prolog Interpretern vordefiniert. Wenn wir aber auch hier einen Integer mitführen, wie beim erweiterten *member*-Prädikat, dann eröffnen sich noch viele weitere Möglichkeiten, das *append* seinem Zweck zu entfremden:

```
append([], 0, Liste2, Liste2).
append([Anfang|Rest], N, Liste2, [Anfang|Rest2]):-
   append(Rest,N2,Liste2,Rest2),
   N is N2+1.
```

dieses Prädikat können wir nun z.B. dazu benutzen, um eine Liste A hinter dem K-ten Element zu spalten, sodass vorne V und hinten H entsteht:

```
?- append(V,3,H,[0,1,2,3,4,5,6,7,8]).
V=[0,1,2],
H=[3,4,5,6,7,8].
```

wir können prüfen, ob eine Liste A (am Stück) in einer anderen Liste B vorkommt:

```
append(V,K,H,B),   /* spalte B an unbestimmter Stelle K in zwei Listen V und H */
append(A,_,_,H).   /* pruefe, ob H mit A beginnt */
```

und mehr noch: K wird die **Position**, an der A in B anfängt (wenn sie existiert)
Übung:

- Schreiben Sie ein Prädikat, das eine Liste umdreht (sodass $reverse([1,2,3],X)$. die Lösung $X = [3,2,1]$ liefert) Sie dürfen hierzu das *append* Prädikat verwenden (Hinweis: aus einem Element X kann man eine Liste machen, die nur X enthält indem man eckige Klammern $[X]$ darum setzt)

- Versuchen Sie, ein Endrekursives *reverse* zu schreiben (Sie werden einen weiteren Parameter benötigen, für den beim Aufruf die leere Liste übergeben werden wird)
- Schreiben Sie (mit Hilfe des *reverse* Prädikats) ein Prädikat, das Palindrom-artige Listen akzeptiert
- Schreiben Sie ein Prädikat *concat*, das nicht nur zwei Listen zusammenfügen kann, sondern das alle Listen aus einer Liste von Listen zusammenfügt. Sie dürfen das *append* Prädikat verwenden
- Schreiben Sie ein Prädikat *concat*, das nicht nur zwei Listen zusammenfügen kann, sondern das alle Listen aus einer Liste von Listen zusammenfügt. Benutzen sie keine zusätzlichen Prädikate

4.4 Mehr über Teil-Listen

Mit zwei kombinierten *append*s hat man zwar ein extrem mächtiges Werkzeug, um Listen sehr flexibel zu zerlegen, aber mit dieser Methode ist es oft kompliziert, sich passende Anfragen auszudenken. Das folgende Prädikat ist zwar weniger mächtig, dafür aber deutlich simpler zu benutzen und es reicht für die meisten Anwendungen einfach aus: Das *sub_list*-Prädikat bekommt eine Liste und extrahiert daraus eine Teil-Liste. Sein erster Parameter ist die Liste, aus der ein Teil extrahiert werden soll, der zweite Parameter gibt an, ab welchem Index die Teil-Liste anfangen soll, der dritte Parameter gibt an, wie lang die Teil-Liste sein soll, der vierte Parameter wird normalerweise nur zurückgeben, wie viele Elemente hinter der Teil-Liste noch übrig sind (das ist normalerweise nutzlos, ist aber für einige Zweckentfremdungen nützlich) und der 5. Parameter gibt die Teil-Liste zurück. Um diese vielen Informationen zu veranschaulichen, bevor wir zu der Implementierung kommen, zuerst ein Beispiel:

```
% Teil-Liste mit Laenge 3, ab Index 2
?- sub_list([1,2,5,3,42,23,9,4711], 2, 3, RestLaenge, TeilListe).
RestLaenge = 3,
TeilListe = [5, 3, 42].
```

Dieses Prädikat realisieren wir, indem wir ganz normal vorne Elemente weg nehmen und den zweiten Parameter runter zählen, bis er 0 erreicht hat (dritte Klausel), dann nehmen wir vorne weitere Elemente weg, bauen daraus das Ergebnis auf und zählen dabei den dritten Parameter runter, bis er 0 erreicht hat (zweite Klausel). Hat auch er 0 erreicht, dann zählen wir noch die restlichen Elemente mittels *length* (erste Klausel).

```
sub_list(Rest, 0, 0, RestLaenge, []):-
   length(Rest, RestLaenge).

sub_list([X|Rest], 0, Laenge, RestLaenge, [X|RestTeil]):-
   sub_list(Rest, 0, Laenge1, RestLaenge, RestTeil),
   Laenge is Laenge1 + 1.

sub_list([_|Rest], Index, Laenge, RestLaenge, RestTeil):-
   sub_list(Rest, Index1, Laenge, RestLaenge, RestTeil),
   Index is Index1 + 1.
```

Wie man sieht, zählen wir den zweiten und dritten Parameter nicht wirklich **runter**, sondern setzen sie in der zweiten bzw. ersten Klausel auf 0 und zählen sie in der dritten bzw. zweiten Klausel **hoch** - dadurch bleibt die Funktionalität erhalten, aber wir müssen den zweiten und dritten Parameter nicht unbedingt binden, wodurch interessante Zweckentfremdungen möglich werden:

```
% Teil-Liste mit Laenge 3, ab dem 2. Index VON HINTEN!
?- sub_list([1,2,5,3,42,23,9,4711], RestLaengeVorne, 3, 2, TListe).
RestLaengeVorne = 3,
TListe = [3, 42, 23]

% Entferne vorne 3 und hinten 2 Elemente
?- sub_list([1,2,5,3,42,23,9,4711,11880], 3, _, 2, TeilListe).
TeilListe = [3, 42, 23, 9]

% Enthaelt die Liste die Teil-Liste?
?- sub_list([1,2,5,3,42,23,9], StartetBeiIndex, _, _, [5,3,42]).
StartetBeiIndex = 2

% Was sind die Teil-Listen der Laenge 6?
?- sub_list([1,2,5,3,42,23,9,4711,11880],_,6,_,X).
X = [1, 2, 5, 3, 42, 23] ;
X = [2, 5, 3, 42, 23, 9] ;
X = [5, 3, 42, 23, 9, 4711] ;
X = [3, 42, 23, 9, 4711, 11880]

% Wie ist die Laenge der Liste?
?- sub_list([1,2,5,3,42,23,9,4711,11880],0,0,Laenge,_).
Laenge = 9.
```

4.5 Kombinatorik

Ich denke wir sind so weit, dass ich in diesem Kapitel nur die Grundgedanken nennen brauche und dann direkt Prädikate angeben kann, die die gewünschten Aufgaben erledigen

4.5.1 Permutationen

Eine Permutation einer Liste ist einfach eine beliebige Anordnung der Elemente der Liste (ohne eins mehrfach zu verwenden). Eine Permutation einer Liste $[Anfang|Rest]$ ist eine Permutation vom $Rest$, in den der $Anfang$ irgendwo, an einer beliebigen Stelle, eingefügt wird (wir erinnern uns an die möglichen Zweckentfremdungen des *delete* Prädikats)

```
permutationen([],[]).
permutationen([Anfang|Rest],Permutation):-
   permutationen(Rest, Restpermutation),
   /* Anfang irgendwo einfuegen */
   delete(_,Anfang,Permutation,Restpermutation).
```

Von einer Liste mit n Elementen gibt es $n! = 1 \cdot 2 \cdot 3 \cdot ... \cdot n$ Permutationen.

4.5.2 Teilmengen

Eine Teilmenge einer Liste $[Anfang|Rest]$ ist eine Teilmenge vom $Rest$, an den der $Anfang$ entweder angehangen wird, oder nicht

```
teilmengen([],[]).
/*Anfang nicht in Teilmenge enthalten*/
teilmengen([_|Rest],Teilmenge):-
   teilmengen(Rest,Teilmenge).
/* Anfang enthalten */
teilmengen([Element|Rest],[Element|Teilmenge]):-
   teilmengen(Rest,Teilmenge).
```

Von einer Liste mit n Elementen gibt es 2^n Teilmengen.

4.5.3 Folgen / Kombinationen

Bei den nächsten beiden Prädikaten muss man eine Länge der resultierenden Liste vorgeben, da sie die Elemente der Grundmenge mehrfach enthalten können.

Eine Folge der Länge N von Elementen einer $Grundmenge$ ist eine Folge der Länge $N-1$, an die vorne irgend ein Element der Grundmenge angehangen wird.

```
folgen(0,_,[]).
folgen(N,Grundmenge,[Anfang|Rest]):-
   N > 0,
   N1 is N-1,
   folgen(N1,Grundmenge,Rest),
   /* hole beliebigen Anfang aus der Grundmenge */
   member(Anfang,Grundmenge).
```

Von einer Grundmenge mit n Elementen gibt es n^k Folgen der Länge k.

4.5.4 Sortierte Folgen

Sortierte Folgen sind rekursiv etwas schlechter zu beschreiben. Der Grundgedanke ist hier, dass nur das erste Element der Grundmenge an die rekursiv erhaltene Liste angehangen werden darf (aber nicht muss). ENTWEDER das erste Element der Grundmenge wird als erstes Element der Ergebnisliste benutzt - dann darf es danach am Anfang der ***N-1***-elementigen, Restliste erneut auftreten - ODER ABER das erste Element der Grundmenge wird nicht als erstes Element der Ergebnisliste benutzt, dann darf es später aber auch nicht mehr in der Ergebnisliste auftreten. In diesem Fall entfernen wir also das erste Element der Grundmenge und berechnen eine ***N***-elementige sortierte Folge über der kleineren Grundmenge.

```
sortiertefolgen(0,_,[]).
/* Anfang der Grundmenge verwenden */
sortiertefolgen(N,[Anfang|GrundmengenRest],[Anfang|Rest]):-
   N > 0,
   N1 is N-1,
   sortiertefolgen(N1,[Anfang|GrundmengenRest],Rest).
```

```
/* Anfang der Grundmenge nicht verwenden */
sortiertefolgen(N,[_|GrundmengenRest],Folge):-
   N > 0,
   sortiertefolgen(N,GrundmengenRest,Folge).
```

Von einer Grundmenge mit n Elementen gibt es $\binom{n+k-1}{k} = \frac{(n+k-1)!}{k!\cdot(n-1)!}$ sortierte Folgen der Länge k.

4.6 Kombinatorische Rätsel

Ein Student kommt mit dem Geld, das seine Eltern ihm zur Verfügung stellen nicht aus und schreibt seinem Vater einen Brief mit dem Inhalt „SEND MORE MONEY“ . Der Vater will vorher wissen, ob sein Sohn an der Universität auch etwas lernt und nicht nur Partys[1] feiert. Er schickt dem Sohn die Antwort „SEND+MORE=MONEY“ und will die Zahlungen (falls der Sohn diese Aufgabe lösen kann) um den Betrag MONEY erhöhen. Konkret formuliert sollen den Buchstaben S, E, N, D, M, O, R und Y (verschiedene) Ziffern zugeordnet werden, sodass gilt

$$\begin{array}{rr} & 1000*S+100*E+10*N+D \\ + & 1000*M+100*O+10*R+E \\ = & 10000*M+1000*O+100*N+10*E+Y \end{array}$$

In Prolog ist kaum etwas leichter als das:

```
sendmoremoney(S,E,N,D,M,O,R,Y):-
   permutationen([0,1,2,3,4,5,6,7,8,9],[S,E,N,D,M,O,R,Y|_]),
                  1000*S + 100*E + 10*N + D
                + 1000*M + 100*O + 10*R + E
   =:=  10000*M + 1000*O + 100*N + 10*E + Y.
```

Was hier passiert ist, dass jede Permutation der Ziffern $0-9$ erzeugt wird, wobei die Variablen S, E, N, D, M, O, R, Y mit den ersten 8 Ziffern der aktuellen Permutation gebunden werden. Danach wird noch geprüft, ob diese Belegung die Vorgabe $SEND + MORE = MONEY$ erfüllt.
Da es viele Lösungen gibt, benutzen wir die Methode, aus 3.7.6 um das Programm benutzerfreundlicher zu machen (der Benutzer muss nicht mehr ständig Tasten drücken um weitere Ausgaben zu erhalten) ein netter Nebeneffekt ist, dass die Parameter überflüssig werden, als Goal reicht also *sendmoremoney.* anstatt (wie vorher) $sendmoremoney(S, E, N, D, M, O, R, Y)$.

```
sendmoremoney:-
   permutationen([0,1,2,3,4,5,6,7,8,9],[S,E,N,D,M,O,R,Y|_]),
                   1000*S + 100*E + 10*N + D
                 + 1000*M + 100*O + 10*R + E
    =:=  10000*M + 1000*O + 100*N + 10*E + Y,
   write(S),write(E),write(N),write(D),write('+'),
   write(M),write(O),write(R),write(E),write('='),
   write(M),write(O),write(N),write(E),write(Y), nl,
   fail.
```

[1] Die Rechtschreibung ist nach deutscher Grammatik korrekt, „Parties“ hieße es nach englischer Grammatik, die hierzulande nicht gilt.

Probleme:

- Die Lösungen werden je zwei mal auftreten, weil die gleiche Belegung der ersten 8 Stellen mit vertauschten hinteren beiden Stellen ebenfalls erzeugt wird - dies lässt sich beheben, indem die hinteren beiden Ziffern ebenfalls gebunden werden und dann gefordert wird, dass die erste davon größer ist, als die zweite.
- Das Programm könnte schneller laufen, wenn die Variablen einzeln, nach und nach gebunden würden (bei den hinteren Stellen angefangen) wobei nach jeder Bindung geprüft wird, ob die Summe bis zu diesem Punkt gültig ist (zur Veranschaulichung dieser Methode ein Ausschnitt aus dem - deutlich längeren - Quellcode)

```
sendmoremoney_quick:-
   Ziffern = [0,1,2,3,4,5,6,7,8,9],
   delete(_,D,Ziffern,DZiffern),  % waehle D
   delete(_,E,DZiffern,EZiffern), % waehle E =\= D
   delete(_,Y,EZiffern,YZiffern), % waehle Y =\= D, E
   Y is (D+E) mod 10,
   Uebertrag1 is (D+E) // 10,

   delete(_,N,YZiffern,NZiffern), % waehle N =\= D,E,Y
   delete(_,R,NZiffern,RZiffern), % waehle R =\= D,E,Y,N
   E is (N+R+Uebertrag1) mod 10,
   Uebertrag2 is (N+R+Uebertrag1) //\left[ \left\lbrace  10,
   ...
```

Dennoch ist dies meiner Meinung nach ein eindrucksvolles Beispiel dafür, wie leicht und elegant man Prolog komplizierte Aufgaben lösen lassen kann.

Eine weitere Möglichkeit, um den Code noch leichter schreiben zu können ist, die Ziffern in einem anderen Prädikat zu einer Zahl zusammen zu setzen - dieses Prädikat nenne ich mal listenZahl:

```
listenZahl([],1,0).
listenZahl([Anfang|Rest], NextPotenz, Erg):-
   listenZahl(Rest, EigenePotenz, RestErg),
   Erg is RestErg+EigenePotenz*Anfang,
   NextPotenz is 10*EigenePotenz.
```

Dieses Prädikat kann man nun benutzen, um die Ziffern eines einzelnen Worts zu einer Zahl zusammenzufassen, indem die Ziffern zu einer Liste gemacht werden (z.B. [S,E,N,D]) zu der die Listenzahl berechnet wird, der wir z.B. das Wort selbst als Namen geben. z.B. $listenZahl([S,E,N,D],\_,SEND)$ sodass SEND =:= 1000*S+100*E+10*N+D gilt. (über den zweiten Parameter werden nur interne Informationen über die Länge der Restliste nach oben weiter gereicht - an ihm sind wir aussen nicht interessiert, daher deklarieren wir ihn aussen als beliebig)
Nun lässt sich das Programm für das SEND MORE MONEY Problem folgendermaßen formulieren:

```
sendmoremoney:-
   permutationen([0,1,2,3,4,5,6,7,8,9], [S,E,N,D,M,O,R,Y|_]),
   listenZahl([S,E,N,D],_,SEND),
   listenZahl([M,O,R,E],_,MORE),
   listenZahl([M,O,N,E,Y],_,MONEY),
   SEND+MORE=:=MONEY,
   write(SEND),write('+'),write(MORE),write('='),writeln(MONEY),
   fail.
```

Übung:

- Lösen Sie die folgende Aufgabe (es existiert nur eine Lösung, die aber mehrmals genannt werden könnte):

```
ABB -  CD = EED
 -     -     *
 FD +  EF =  CE
 =     =     =
EGD *  FH = ???
```

- Vervollständigen Sie den Ausschnitt aus dem Quellcode für die schnellere Lösung des SEND MORE MONEY Problems. Vergleichen Sie die Laufzeiten - hat sich die Entwicklung des schnelleren Programms rentiert?
- Lösen Sie das kombinatorische Rätsel GAUSS + RIESE = EUKLID (es gibt 2 verschiedene Lösungen, die den gleichen Wert für EUKLID haben)

4.7 Strings

Für Strings gibt es zwei verschiedene Repräsentationen - einerseits Strings, die von normalen Anführungszeichen umgeben sind - diese sind intern nichts anderes als eine Liste von ASCII-Codes[9] und man kann sie genau wie normale Listen von Integern behandeln. Strings, die von einfachen Hochkommata umgeben sind, werden intern als Array von Bytes behandelt, die Prolog schneller verarbeiten kann als Listen - Arrays gibt es aber gar nicht in Prolog, also kann man mit derartigen Strings gar nicht arbeiten - man kann sie eingeben und zwischen Prädikaten herumreichen, aber man kann sie nicht wie Listen zerlegen, verändern etc.

```
 ?- writeln('Hallo'), write("Hallo").
Hallo
[72, 97, 108, 108, 111]

?- [A,B|Rest] = "Hallo".
A = 72,
B = 97,
Rest=[108, 108, 111].

?- [A,B|Rest] = 'Hallo'.
fail.
```

[9] siehe http://de.wikipedia.org/wiki/Ascii-code

Ich bin schon in Situationen gewesen, in denen ich von Prädikaten Strings in Array-Form bekommen habe, sie aber dann als Liste von Integern benötigte (z.B. um Großbuchstaben in Kleinbuchstaben umzuwandeln). Hierzu kann man das Systemprädikat *string_to_list* benutzen. Übergibt man als ersten Parameter einen String in Array-Form und als zweiten Parameter eine Variable, dann wird die Integer-Liste, die dem String entspricht, an die Variable gebunden. Übergibt man eine Liste von Integern als zweiten Parameter und eine Variable als ersten Parameter, dann wird der String an die Variable gebunden, der der Liste entspricht

```
 ?- string_to_list('Hallo', X), write(X).
[72, 97, 108, 108, 111]

?- string_to_list(X, "Hallo"), write(X).
Hallo

?- string_to_list(X, [72, 97, 108, 108, 111]), write(X).
Hallo
```

Damit könnten wir jetzt z.B. die Länge eines Strings bestimmen, indem wir ihn in eine Liste umwandeln und die Länge der Liste bestimmen - aber Prolog hat auch ein paar interne Prädikate für Strings. Man kann die Länge eines Strings mit *string_length* bestimmen, Strings (wie mit dem *append* für Listen) zusammenfügen oder spalten mittels *string_concat* und mit *sub_string* lassen sich Teil-Strings extrahieren. *sub_string* bekommt als Parameter einen String, den Index, ab dem der Teilstring bestimmt werden soll, die Länge des Teilstrings, der vierte Parameter gibt die Anzahl der Zeichen hinter dem Teil-String zurück und der fünfte Parameter gibt den Teil-String selbst zurück. Dieses Prädikat lässt sich genau so wie das *sub_list* Prädikat (siehe 4.4) zweckentfremden - das *sub_list* Prädikat habe ich ehrlich gesagt sogar gezielt so entwickelt, dass es sich genau wie das *sub_string* Prädikat verhält.

```
 ?- string_length('Hallo', X).
X = 5

?- string_concat('Hallo ', 'Welt!', X).
X = 'Hallo Welt!'

?- string_concat(X, 'Welt!', 'Hallo Welt!').
X = 'Hallo '
```

Beispiele für das *sub_string* Prädikat:

```
% Teil-String mit Laenge 3, ab dem 5. Zeichen
?- sub_string('Dies ist ein Test', 5, 3, X, Y).
X = 9,
Y = "ist".

% Teil-String mit Laenge 3, ab dem 5. Zeichen von hinten
?- sub_string('Dies ist ein Test', X, 3, 5, Y).
X = 9,
Y = "ein".
```

```
% Entferne vorne 5 und hinten 2 Zeichen
?- sub_string('Dies ist ein Test', 5, _, 2, Y).
Y = "ist ein Te".

% Enthaelt der String den Teil-String?
?- sub_string('Dies ist ein Test', X, Y, Z, 'ein').
X = 9,
Y = 3,
Z = 5.

% Was sind die Teil-Strings der Laenge 14?
?- sub_string('Dies ist ein Test',_,14,_,X).
X = "Dies ist ein T" ;
X = "ies ist ein Te" ;
X = "es ist ein Tes" ;
X = "s ist ein Test".

% Wie ist die Laenge des Strings?
?- sub_string('Dies ist ein Test',0,0,Laenge,_).
Laenge = 17.
```

Man kann auch Strings in Atome umwandeln und umgekehrt, mittels *string_to_atom* - Ich will es hier nur erwähnt haben, dieses Prädikat ist wohl eher selten von Nutzen, ich habe es aber schon einmal benötigt.

Übung:

- Schreiben Sie Prädikate lowercase und uppercase, welche Großbuchstaben (in Strings!) durch Kleinbuchstaben ersetzen bzw. umgekehrt! Hinweis: der ASCII-Code des großen A ist 65, der des kleinen a ist 97, der des großen Z ist 90, der des kleinen z ist 122

5 Prädikate höherer Ordnung, Meta-Programmierung

In funktionalen Sprachen wie Haskell kann man Funktionen an andere Funktionen als Parameter übergeben, ein ähnliches Konzept sind Callback-Funktionen, die viele imperative Programmiersprachen bieten.
Genauso kann man in Prolog einfach Ausdrücke als Parameter an Prädikate übergeben. Ein simples Beispiel wäre

```
do(G) :- G.
```

übergeben wir diesem *do*-Prädikat einen beliebigen Prolog Code, wird er ausgeführt:

```
?- do(writeln('Hallo Welt!')).
Hallo Welt!
true.
```

Dabei ist aber zu beachten, dass falls der übergebene Code aus mehreren (durch AND, OR,... verknüpften) Teilen besteht, der Parameter in Klammern zu setzen ist, da Prolog sonst nicht wüsste, ob nun ein AND gemeint ist oder ein weiterer Parameter des Prädikats. Im folgenden Beispiel nimmt Prolog an, dass *write('Hallo ')* und *writeln('Welt!')* zwei voneinander unabhängige Parameter für das Prädikat wären.

```
?- do(   (   write('Hallo '),writeln('Welt!')   )   ).
Hallo Welt!
true.

?- do(       write('Hallo '),writeln('Welt!')       ).
ERROR: Undefined procedure: do/2
ERROR:     However, there are definitions for:
ERROR:         do/1
fail.
```

5.1 Negation as Failure

Ein gern verwendetes Beispiel für Prädikate höherer Ordnung ist das *not*. Intern funktioniert es folgendermaßen:

```
not(G) :-
   G,
   !,
   fail.
not(_).
```

also es bekommt ein Goal G, das auf seine Richtigkeit geprüft wird, indem es einfach als Bedingung verwendet wird. Ist das Goal erfüllbar, dann wird das $cut - and - fail$ erreicht, wodurch $not(G)$ zu $false$ ausgewertet wird und die nächste Klausel des *not* nicht mehr ausprobiert wird. Ist das Goal unerfüllbar, dann wird der cut nicht erreicht und die zweite Klausel akzeptiert das (unerfüllbare) Goal unmittelbar.

5.2 Pseudo-Verzweigungen

Ähnlich wie die Negation by Failure kann man ein if-then-else programmieren. Das Prädikat nenne ich if und es ist 3-stellig. Der erste Parameter ist die Bedingung, die geprüft wird, indem es als Bedingung verwendet wird. Ist es erfüllbar, dann wird ein Cut ausgeführt, der verhindert, dass in der 2. Klausel des Prädikats der $ElseCode$ ausgeführt wird. Anschließend wird der $ThenCode$ ausgeführt. Ist die Bedingung unerfüllbar, dann wird der Cut und der $ThenCode$ nicht erreicht und in der 2. Klausel wird der $ElseCode$ ausgeführt.

```
if(Cond, ThenCode, _) :-
   Cond,
   !,
   ThenCode.
if(_, _, ElseCode) :-
   ElseCode.
```

Beispiel-Anfragen:

```
?-if(5<3, writeln('Ja'), writeln('Nein')).
Nein
true.

?-if(5>3, writeln('Ja'), writeln('Nein')).
Ja
true.
```

Mit diesem if (und ähnlichen Prädikaten zur Nachbildung von Schleifen, die wir später behandeln werden), zu arbeiten, kann sehr unübersichtlich werden, deshalb empfehle ich, in Quellcodes (ähnlich wie in imperativen Sprachen) Einrückung zu benutzen:

```
if((5<3),(
   writeln('Ja')
),(
   writeln('Nein')
)
```

5.3 Pseudo-Schleifen

Eine einfache Schleife ist

```
repeat.
repeat :- repeat.
```

ruft man dieses *repeat* auf, dann verzweigt Prolog unendlich oft und der darauf folgende Teil kann unendlich oft ausgeführt werden. Benutzt man hier eine $fail$-getriebene Fortsetzung, dann wird der Teil zwischen dem repeat und dem fail unendlich oft ausgeführt, es sei denn irgendwann wird ein cut ausgeführt.

```
?- repeat, write('X'), fail.
XXXXXXXXXXXXXXX...
```

Man kann natürlich nicht nur eine Pseudo-Schleife entwickeln, die unendlich oft verzweigt, sondern auch welche, deren Verzweigung man mit den Parametern steuern kann, z.B.

```
for(Y,Y,Z) :-
   Y =< Z.
for(X,Y,Z) :-
   Y < Z,
   Y1 is Y+1,
   for(X,Y1,Z).

?- for(I,1,10), write(I), write(' '), fail.
1 2 3 4 5 6 7 8 9 10
```

Ein solches *for* ist in GNU Prolog vordefiniert, in SWI Prolog nicht. In der Praxis wird es aber sehr unbequem, solche Schleifen zu schreiben und die Cuts richtig zu setzen. Leichter wird es, wenn man Prädikate schreibt, die einen übergebenen Prolog-Code mehrfach ausführen:

```
repeat(Code) :-
   Code,
   fail.
repeat(Code) :-
   repeat(Code).

?- repeat(write('X')).
XXXXXXXXXXXXXXX...

for(Y,Y,Z, Code) :-
   Y =< Z,
   Code,
   fail.
for(X,Y,Z, Code) :-
   Y < Z,
   Y1 is Y+1,
   for(X,Y1,Z, Code).

?- for(I,1,10,( write(I), write(' ') )).
1 2 3 4 5 6 7 8 9 10
```

Benutzt man solche Schleifen in den Bedingungen anderer Prädikate, empfehle ich (wie bei den Pseudo-Verzweigungen) Einrückung zu benutzen

```
repeat(
   write('X')
)

for(I,1,10,(
   write(I),
   write(' ')
)).
```

5.4 Liste aller Lösungen

Manchmal benötigt man nicht eine Lösung für eine Anfrage, sondern alle Lösungen auf einmal. Man kan mit dem Systemprädikat $findall$ eine Liste von Lösungen verlangen. $findall$ braucht 3 Parameter - der zweite Parameter ist ein Goal ($findall$ ist also ein Prädikat höherer Ordnung), der erste Parameter ist eine freie Variable (oder auch ein beliebiger Term) und der dritte Parameter ist für die Rückgabe der resultierenden Liste.
Als Beispiel erst mal nur eine Anfrage: $?-findall(X, member(X, [1,2,3]), Ergebnis)$
Ausgabe: Ergebnis=[1,2,3]. Dieser Aufruf bindet lediglich die an member übergebene Liste an $Ergebnis$ - Es gibt auch bessere Wege, eine Liste zu kopieren, aber wir können ja auch komplexere Goals verwenden. Z.B. können wir aus zwei Listen die Vereinigungsmenge (set union) , Schnittmenge (set intersection) und Differenzmenge (difference set) bilden:

```
set_union(A,B,C):-
   findall(X, (member(X,A) ; member(X,B)), C).
set_intersection(A,B,C):-
   findall(X, (member(X,A) , member(X,B)), C).
set_difference(A,B,C):-   % C = A \ B
   findall(X, (member(X,A) , \+ member(X,B)), C).
```

Ebenfalls interessant ist, dass wir mit dem $findall$ eine ähnliche Funktionalität haben, wie mit der $filter$-Funktion in Haskell - z.B. können wir damit aus einer Liste alle Elemente auswählen, die kleiner bzw. grösergleich einem bestimmten Element sind:

```
?- findall(X,   (member(X,[1,9,4,7,3,6]),X>5),  Liste).
Liste = [1,4,3]
```

womit wir alles Nötige zusammen haben, um den Quicksort zu schreiben.

5.4.1 Quicksort

Der Quicksort ist wohl der bekannteste Sortieralgorithmus, den es gibt. Er wurde von **Charles Antony Richard Hoare** entwickelt, von dem auch das Hoare-Kalkül stammt, mit dem sich die Korrektheit von (imperativen) Algorithmen beweisen lässt. Hoare ist heute leitender Forscher bei Microsoft Research in Cambridge.[10]
Ich beschreibe den Quicksort immer gerne folgendermaßen: Man stelle sich vor, man will ein Zimmer aufräumen. Dazu sammelt man alle Dinge, die in der linken Hälfte des Zimmers liegen, aber in die rechte Hälfte gehören und legt sie in die rechte Hälfte, und umgekehrt. In jeder der beiden Hälften sucht man sich nun jeweils alle Dinge, die im vorderen Viertel des Zimmers liegen, aber ins hintere Viertel gehören und bringt sie dort hin und umgekehrt. So fährt man fort, auf immer kleiner werdenden Bereichen des Zimmers, bis jedes Ding „nah genug“ an dem Platz ist, wo es hingehört.
Das Problem bei dieser Betrachtungsweise ist, dass man bei Listen (oder auch Arrays in imperativen Sprachen) nicht genau weiss, welche Elemente nach links bzw. rechts gehören - man kann sich nur irgend ein Element auswählen (dieses

[10] vgl. http://de.wikipedia.org/wiki/Tony_Hoare

Element nennt man den **Median**) und kann nun alle Elemente sammeln, die weiter nach links bzw. weiter nach rechts als der Median gehören. Es gibt viele Strategien, wie man den Median sinnvoll wählen kann, in Prolog ist es am einfachsten, zu implementieren, das erste Element der Liste zu nehmen.

```
quicksort([],[]).
quicksort([Median|Rest], Sortiert):-
   /* sammele alles, was weiter nach links als der Median gehört */
   findall(X, (member(X,Liste),X<Median), Links),
   /* sammele alles, was weiter nach rechts als der Median gehört */
   findall(Y, (member(Y,Liste),Y>=Median), Rechts),
   quicksort(Links, LinksSortiert),   /* sortiere links */
   quicksort(Rechts, RechtsSortiert),   /* sortiere rechts */
   append(LinksSortiert, [Median|RechtsSortiert], Sortiert).
```

Wer den Quicksort schon in imperativen Sprachen gesehen hat (Implementierungen in nahezu jeder Programmiersprache findet man im Internet wie Sand am Meer), wird erstaunt sein, wie viel kürzer und verständlicher die Implementierung in Prolog ist.

Zusätzliche Information:
Der Quicksort trägt seinen Namen eigentlich zu Unrecht - es gibt Fälle, in denen er auf Arrays/Listen der Länge n eine Laufzeit von n^2 hat, also wo der er genauso schlecht ist, wie der Selectionsort (einer der schlechtesten Sortieralgorithmen, die noch Praxistauglich sind). Das passiert dann, wenn man als Median in jedem Rekursionsaufruf ein Element wählt, das nach ganz links bzw. ganz rechts gehört - dann kann man nämlich „das Zimmer“ nicht halbieren, sondern findet nur für ein „Ding“ (für den Median) den richtigen Platz und muss immernoch fast das ganze Zimmer mit einem kaum reduzierten Aufwand weiter aufräumen. In unserer Implementierung ist das schon der Fall, wenn die übergebene Liste aufsteigend oder absteigend sortiert ist.
Im Average-Case (also im „Regelfall“) hat der Quicksort aber eine Laufzeit von $n \cdot log(n)$, was Der Geschwindigkeit der besten Sortieralgorithmen entspricht. Die anderen guten Sortieralgorithmen, deren Laufzeit garantiert höchstens $n \cdot log(n)$ ist (wie dem Mergesort), brauchen dazu aber viel zusätzlichen Arbeitsspeicher, weswegen man in der Praxis häufig den Quicksort findet.

5.4.2 Stolperstein

Ein Problem des findall ist, dass Prolog verzweigt, wenn ungebundene Variablen im Goal vorkommen. Verwenden wir z.B. das *member* Prädikat von oben, welches die Position eines Elements ebenfalls bestimmt, um die rechte und linke Hälfte einer Liste zu bekommen

```
...
length(Liste, Laenge),
findall(X, (member(X,Pos,Liste), Pos<Laenge//2), Links),
findall(X, (member(X,Pos,Liste), Pos>=Laenge//2), Rechts),
...
```

dann verzweigt Prolog über *Pos* und wir bekommen mehrere Ergebnislisten (je eine für jeden Wert, den *Pos* annimmt). Um dieses Verhalten abzustellen,

müssen wir $Variable\hat{}$ vor das übergebene Goal schreiben für jede Variable, über die nicht verzweigt werden soll, also z.B.

```
...
length(Liste, Laenge),
findall(X, Pos^(member(X,Pos,Liste),Pos<Laenge//2), Links),
findall(X, Pos^(member(X,Pos,Liste),Pos>=Laenge//2), Rechts),
...
```

Dieses Verzweigen über freien Variablen verursacht häufig Fehler, die oft auch für Profis schwer zu finden sind. Nichtsdestotrotz haben wir mit der Spaltung von Listen in der Mitte alles Nötige zusammen, um den Mergesort zu schreiben.
Übung:

- Schreiben Sie den Mergesort

5.4.3 Weitere Prädikate für Listen von Lösungen

Das $findall$ liefert eine Liste, die jede gefundene Lösung enthält. Wenn eine Lösung mehrfach vorkommt, erscheint sie auch mehrmals in der Liste und wenn keine Lösung gefunden wurde, gibt $findall$ die leere Liste zurück. Dieses Verhalten ist meistens vernünftig, aber es kann auch vorkommen, dass man nur fortfahren will, wenn es überhaupt eine Lösung gegeben hat (also die Liste von Lösungen nicht leer ist). Man kann nun entweder nach dem $findall$ verlangen, dass Ergebnis ungleich der leeren Liste ist, oder man benutzt das Prädikat $bagof$, das die gleichen Parameter wie $findall$ bekommt. Benötigt man die Ergebnisse nur je ein mal, dann kann man das Prädikat $setof$ benutzen, welches ebenfalls nur dann akzeptiert, wenn die Liste der Lösungen nicht leer ist. Ein Prädikat, welches jede Lösung nur ein mal in die Liste aufnimmt, aber auch die leere Liste zurückgibt, wenn keine Lösung existiert, ist der Dokumentation von SWI Prolog zufolge nicht vordefiniert. Da dies aber auch nützlich sein könnte, wollen wir ein solches Prädikat selbst entwickeln:

```
findall_set(A,B,C):-
   setof(A,B,C) -> true ; C=[].
```

5.5 Prädikatenlogik - Prädikate mit Quantoren

Wir machen jetzt einen kleinen Ausflug in die mathematische Logik, was für Schüler und Studien-Anfänger vielleicht zu kompliziert sein könnte - sollten Sie in den folgenden zwei Kapiteln nurnoch Bahnhof verstehen, lassen Sie sich nicht entmutigen, die späteren Kapitel trotzdem zu lesen. Lesen Sie auf jeden Fall auch das Kapitel „Anwendung des All-Quantors“, in dem der sehr theoretische Teil der kommenden zwei Kapitel sehr nützlich in die Praxis integriert wird.

In der sogenannten „First Order Prädikatenlogik“ (Kurz „FO-Logik“ oder noch kürzer „FOL“ genannt) hat man die gleichen Aussagen wie in der Aussagenlogik (Aussagen wie $a > b$ verknüpft mit AND, OR und NOT). Neben den Dingen, die es in der Aussagenlogik gibt, gibt es zwei neue Symbole: den Existenz Quantor $\exists$ und den All-Quantor $\forall$.

5.5.1 Existenz-Quantor ∃

Eine FOL Aussage mit Existenz-Quantor ist wahr, wenn es ein Element in der „Grundmenge“ gibt, mit dem die Formel hinter dem Quantor wahr wird - die Grundmenge wird normalerweise vorher vereinbart oder in der Formel mit angegeben. Einige Beispiele:

- $\exists x \in \mathbb{Z} : (5 + x = 3)$ ist Wahr, nämlich für x=-2
- $\exists x \in \mathbb{R} : (x^2 = -1)$ ist Falsch, denn es gibt keine reelle Wurzel aus -1
- $\exists x \in \mathbb{Q} : (x^2 = 2)$ ist Falsch
- $\exists x \in \mathbb{R} : (x^2 = 2)$ ist hingegen Wahr

Oft will man keine Aussagen über die gesamte Grundmenge treffen, sondern nur über eine Teilmenge davon, deren Elemente spezielle Eigenschaften φ haben. Soll eine Aussage ψ für ein Element zutreffen, das auch die Aussage φ erfüllt benutzt man eine Konjunktion

$$\exists x : \varphi(x) \wedge \psi(x)$$

Wenn ein Element φ nicht erfüllt, wird es auch nicht als Lösung von ψ akzeptiert, weil das $\wedge$ insgesamt nicht erfüllt ist. Also ist $\exists x : \varphi(x) \wedge \psi(x)$ wahr, wenn ein Element existiert, das φ erfüllt (d.h. ein Element der Teilmenge ist) und ψ erfüllt. $\exists x : \varphi(x) \wedge \psi(x)$ bezeichnet man auch als ***relativierten Existenz-Quantor***.
Beispiele:

- $\exists x \in \mathbb{N} : (x < 3) \wedge (x^2 = 9)$ ist Falsch
- $\exists x \in \mathbb{Z} : (x < 3) \wedge (x^2 = 9)$ ist Wahr, nämlich für $x = -3$
- $\exists x \in \mathbb{N} : (x \mod 2 = 0) \wedge (\neg \exists y \in \mathbb{N} \backslash \{1, x\} : (x \mod y = 0))$ es gibt eine gerade Primzahl (die 2)

Wie benutzt man nun Exitenz-Quantoren in Prolog? Ganz einfach: von alleine! den Existenzquantor haben wir im Grunde schon die ganze Zeit über benutzt - die Variablen in einem Goal bzw. in den Bedingungen einer Klausel sind implizit Existenz-quantifiziert.

5.5.2 All-Quantor ∀

Eine FOL Aussage mit All-Quantor ist wahr, wenn die Aussage hinter dem Quantor für alle Elemente der Grundmenge Wahr ist. Einige Beispiele:

- $\forall x \in \mathbb{Z} : (x * 1 = x)$ ist Wahr
- $\forall x \in \{1, 2, 3\} : (x < 4)$ ist Wahr
- $\forall x \in \mathbb{Q} : (x^2 \neq 2)$ ist Wahr
- $\forall x \in \mathbb{Z} : (x + 42 \neq 42)$ ist Falsch - nämlich für x=0 gilt die Aussage nicht

Oft will man keine Aussagen über die gesamte Grundmenge treffen, sondern nur über eine Teilmenge davon, deren Elemente spezielle Eigenschaften φ haben. Soll eine Aussage ψ nur für diejenigen Elemente gelten, die auch φ erfüllen, benutzt man eine Implikation

$$\forall x : \varphi(x) \rightarrow \psi(x)$$

Wenn ein Element φ nicht erfüllt, erfüllt es trotzdem $\varphi(x) \rightarrow \psi(x)$, weil $a \rightarrow b \Leftrightarrow \neg a \vee b$, also ist $\forall x : \varphi(x) \rightarrow \psi(x)$ wahr, wenn jedes Element, das φ erfüllt (d.h. jedes Element der Teilmenge), auch ψ erfüllt. $\forall x : \varphi(x) \rightarrow \psi(x)$ bezeichnet man auch als ***relativierten All-Quantor***.
Beispiele:

- $\forall x \in \mathbb{N} : x < 5 \rightarrow x^2 < 25$ ist Wahr
- $\forall x \in \mathbb{Z} : x < 5 \rightarrow x^2 < 25$ ist Falsch ($x = -6$ ist ein Gegenbeispiel)
- $\forall x \in \mathbb{N} : x \mod 4 = 0 \rightarrow x \mod 2 = 0$ ist Wahr (jedes Vielfache von 4 ist auch ein Vielfaches von 2)

Wie benutzen wir den All-Quantor in Prolog? Ich will zuerst einige „Rechenregeln“ der mathematischen Logik angeben, mit der wir gleich die Form des All-Quantors in Prolog herleiten können:

- $\neg\neg\varphi = \varphi$ doppelte Negation hebt sich auf
- $\varphi \rightarrow \psi = \neg\varphi \vee \psi$
- $\neg(\varphi \vee \psi) = \neg\varphi \wedge \neg\psi$ Demorgan'sche Regel
- $\neg(\varphi \wedge \psi) = \neg\varphi \vee \neg\psi$ Demorgan'sche Regel
- $\neg\exists x\varphi(x) = \forall x\neg\varphi(x)$ Es existiert keine Lösung x für $\varphi(x)$, genau dann, wenn $\neg\varphi(x)$ für alle x gilt
- $\neg\forall x\varphi(x) = \exists x\neg\varphi(x)$ Es gilt nicht $\varphi(x)$ für alle x, genau dann, wenn ein x existiert. sodass $\varphi(x)$ nicht gilt, also $\neg\varphi(x)$

Damit können wir jetzt zeigen:
$\forall x(\varphi(x) \rightarrow \psi(x))$ doppelt negieren
$= \neg\neg\forall x(\varphi(x) \rightarrow \psi(x))$ eine Negation in die Formel hineinziehen
$= \neg\exists x\neg(\varphi(x) \rightarrow \psi(x))$ Implikationspfeil auflösen
$= \neg\exists x\neg(\neg\varphi(x) \vee \psi(x))$ Demorgan'sche Regel
$= \neg\exists x(\neg\neg\varphi(x) \wedge \neg\psi(x))$ doppelte Negation entfernen
$= \neg\exists x(\varphi(x) \wedge \neg\psi(x))$.

5.5.3 Implementierung des All-Quantors

Wie wir im letzten Kapitel gesehen haben, gilt

$$\forall x(\varphi(x) \rightarrow \psi(x)) = \neg\exists x(\varphi(x) \wedge \neg\psi(x))$$

also können wir ein Prädikat $forall(Phi, Psi)$ schreiben, denn der Existenz-Quantor wird ja automatisch angewendet

```
forall( Phi, Psi ) :-
   \+(( Phi, \+((Psi)) )).
```

in SWI Prolog ist dieses *forall* genau so vordefiniert (nur mit anderen Variablennamen). In GNU Prolog existiert es nicht, aber es ist ja leicht selbst zu implementieren. In der Praxis bin ich mit dieser *forall* Implementierung aber auf Probleme gestoßen, weil sich dieses *forall* innerhalb eines negierten Blocks anscheinend mit der Negation by Failure in die Quere kommt, deshalb habe ich die folgende Implementierung entwickelt:

```
forall( Phi, Psi ) :-
   Phi,
   ( Psi -> true ; ! ),
   fail.
forall( _ , _ ).
```

In der ersten Klausel wird versucht, einen Widerspruch zu finden (also eine Lösung von Phi, die Psi nicht erfüllt). Wird einer gefunden, dann wird das Prädikat mittels *cut-and-fail* abgebrochen. Die erste Klausel ist wegen dem *fail* niemals erfolgreich, aber wenn kein Widerspruch gefunden wird, wird der Cut niemals ausgeführt, weswegen dann die zweite Klausel akzeptiert.
Was kann man nun mit diesem *forall* machen?

5.5.4 Anwendung des All-Quantors

Ein simples Beispiel für die Anwendung des All-Quantors ist die Ausgabe aller Elemente einer Liste

```
?- forall( member(X,[1,2,3]) , (write(X),write(' ')) ).
1 2 3
true.
```

interessantere Beispiele für die Benutzung des All-Quantors wären z.B. zu prüfen, ob die Menge der Elemente einer Liste eine Teilmenge der Menge der Elemente einer anderen Liste ist

```
subset(A,B):-
   forall( member(X,A) , member(X,B) ).

?- subset([1,2,3], [5,4,1,3,8,2]).
true.

?- subset([1,2,3], [1,2,5,6]).
fail.
```

woraus sich direkt ein Prädikat ergibt, das zwei Listen daraufhin prüft, ob sie als Mengen identisch sind

```
equalset(A,B):-
   subset(A,B),
   subset(B,A).

?- equalset([1,2,3], [1,3,2,1,3]).
```

```
true.

?- equalset([1,2,3], [1,2,3,4]).
fail.
```

Übung:

- Schreiben Sie ein Prädikat, das mit einer Datenbank über Krankheiten der Form $krankheit(Name, Symptome)$ z.B.

  ```
  krankheit(grippe, [husten, schnupfen, fieber]).
  krankheit(magen_darm_entzuendung, [erbrechen, durchfall]).
  ```

  Diagnosen stellt! Also es soll eine Liste von Symptomen übergeben werden, die ein Patient hat und das Prädikat soll ermitteln, welche Krankheiten der Patient haben könnte. (Wenn eine Krankheit Symptome hat, die der Patient nicht hat, kommt diese Krankheit nicht in Frage) Benutzen Sie nicht das *subset* Prädikat! Das Prädikat soll die Krankheiten direkt mit *write* Ausgeben, es soll also nicht verzweigen und in jedem Zweig je eine Krankheit zurückgeben.
  Tip: Sie können zwei $forall$ Schleifen verschachteln

5.6 Dynamische Wissensbanken (Klauseln hinzufügen und löschen)

Damit man Klauseln eines Prädikats f, welches n Parameter hat, hinzufügen oder entfernen kann, muss man Prolog anweisen, dieses Prädikat als dynamisches Prädikat zu behandeln. Dazu schreibt man

```
?- dynamic(f/n).
```

z.B.

```
?- dynamic(fibo/2).
```

dann kann man Klauseln hinzufügen mittels des Prädikats *assert* bzw *assertz*- die neue Klausel wird dann beim nächsten Aufruf des Prädikats als letztes ausprobiert. Will man es als erstes ausprobieren lassen, kann man statt *assert* bzw. *assertz* das Prädikat *asserta* benutzen.

```
?- dynamic(a/1), assert(a(1)), asserta(a(2)),
| assertz(a(3)), assert(a(5)).
true.

?- a(X).
X = 2 ;
X = 1 ;
X = 3 ;
X = 5 .
```

Mit dieser Technik speichern z.B. Robotersteuerungen neu gesammeltes Wissen. Soll die neue Klausel Bedingungen haben, kann man diese (genau wie im Quellcode) getrennt durch :− hinter den Kopf schreiben.

```
?- dynamic(a/0), dynamic(b/0), assert(a:-b),
| assert(b:-writeln('Hallo')), a.
Hallo
Yes
```

Löschen kann man Klauseln aus der Wissensbank mit dem Prädikat *retract* bzw. *retractall*, welche als Parameter eine Klausel in der gleichen Form bekommen, wie die *assert* Prädikate. *retract* löscht immer die erste Klausel, die mit dem übergebenen Teil zusammenpasst. *retractall* löscht alle passenden Klauseln. Besteht die Wissensbank also wie oben aus dem dynamischen Prädikat $a/1$, mit den Klauseln **a(2), a(1), a(3) und a(5)**, dann löscht die Anfrage $retract(a(\_))$ im folgenden Beispiel das Fakt $a(2)$, nachdem $retract(a(3))$ das Fakt $a(3)$ gelöscht hatte, weil dies jeweils die vordersten passenden Fakten waren.

```
?- retract(a(3)), retract(a(_)), a(X).
X = 1 ;
X = 5.
?- retractall(a(_)), a(X).
fail.
```

Es ist in dem Zusammenhang wichtig zu wissen, dass Prolog keine Fakten löscht, wenn man Klauseln löschen will, die beliebige Regeln haben, obwohl man für Fakten annehmen könnte, dass sie einfach die Regel *true* haben:

```
b(3).
?- dynamic(a/1), assert(a(4)), assertz(a(X):-b(X)), a(X).
X = 4 ;
X = 3.
?- retract(a(_):-_), a(X).
X = 4.
```

In dem Beispiel wurde $a(X) :- b(X)$ gelöscht, $a(4)$ passte also nicht auf $a(\_) :- \_$, obwohl $a(4)$ gleichbedeutend mit $a(4) :- true$ ist.

5.6.1 Globale Pseudo-Variablen

Man kann durch dynamisches Hinzufügen und Löschen von Klauseln ein System schreiben, das Prolog ermöglicht, sich zu verhalten, als gäbe es globale Variablen. Hierzu definiert man ein zweistelliges Prädikat (z.B. *pseudo_var*), zu dem man dynamisch Klauseln hinzufügt und entfernt, wobei jeweils der erste Parameter ein Atom für den Variablennamen ist und der zweite Parameter den Wert der Variablen darstellt:

```
?- dynamic(pseudo_var/2).
set_var(Name, Value):-
   retract_all(pseudo_var(Name,_)),
   assert(pseudo_var(Name,Value)).
```

Nun kann man z.B. mit dem Aufruf $set\_var(a, 5)$ die Klausel $pseudo\_var(a, 5)$ hinzufügen. Ruft man danach irgendwann z.B. $set\_var(a, 7)$ auf, dann wird $pseudo\_var(a, 5)$ entfernt und dafür $pseudo\_var(a, 7)$ hinzugefügt. Überall im Programm kann man dann mit $pseudo\_var(a, X)$ eine lokale Variable X an den „Wert“ der „Globalen Variable“ a binden.

5.6.2 Projekt: Roboter KI (TODO)

6 Datenstrukturen

Bevor wir zu den Datenstrukturen kommen, die viele Leser sicher schon aus imperativen Sprachen kennen werden, müssen wir erst einmal tiefer in zwei interne Mechanismen von Prolog sehen - die Term-Zerlegung und die Term-Verarbeitung. Ohne dieses Hintergrundwissen würde man die Konzepte, die hinter den Datenstrukturen stehen, nicht verstehen.

6.1 Term-Zerlegung

Da dieses Thema gerne für Verwirrung sorgt, will ich erst noch einmal darauf hinweisen, dass Prolog Terme nicht automatisch auswertet. Wenn wir einen Term wie $5+sin(3-cos(2))$ an ein Prädikat $a(X)$ übergeben, dann gilt innerhalb des Prädikates **nicht** $X = 4,72888$, sondern $X = 5 + sin(3 - cos(2))$. Wenn wir uns an das komische Verhalten des = Operators erinnern, dann bewirkte die Zuweisung $A + B = 5 + 3$ die Belegung $A = 5$, $B = 3$. Genauso können wir mit dem = Operator oder auch in Klausel-Köpfen beliebige Terme zerlegen: die Zuweisung $a(X) = a(5)$ bewirkt die Belegung $X = 5$. $a(3, X) = a(Y, 4)$ bewirkt $X = 4$, $Y = 3$. Der Fachausdruck hierfür ist **Unifikation**, das = unifiziert die Terme rechts und links, was nichts weiter heisst, als dass die Variablen rechts und links so belegt werden, dass die Terme identisch werden. Im letzten Beispiel mussten dazu eben $X = 4$ und $Y = 5$ gesetzt werden. Ebenso lassen sich Terme in Klausel-Köpfen zerlegen - Prolog unifiziert nämlich auch die Anfragen in Goals bzw. in Klausel-Bedingungen mit den Köpfen der Klauseln aus der Wissensbank, um zu entscheiden, welche der Klauseln bei Aufrufen ausprobiert werden.

```
a(c(X), d(7), b(3,e(9))):-
   write('Klausel 1: X='),
   writeln(X).
a(b(X), c(3), d(5,e(7))):-
   write('Klausel 2: X='),
   writeln(X).

?- a(b(5), c(Y), Z).
Klausel 2: X=5
Y=3, Z=d(5,e(7))

?- a(c(3), Y, b(3,Z)).
Klausel 1: X=3
Y=d(7), Z=e(9)
```

Im ersten Aufruf wurde der Code aus der ersten Klausel nicht benutzt, weil schon der erste Parameter $b(5)$ nicht mit $c(X)$ unifizierbar war.

6.1.1 Ableitungen berechnen (TODO)

```
derivation(X, C, 0):-
   integer(C) ; float(C).
derivation(X, X, 1).
derivation(X, X^M, M*X^M1):-
```

```
   (integer(M); float(M)),
   M1 is M-1.

derivation(X, A+B, DA+DB):-
   derivation(X, A,DA),
   derivation(X, B,DB).
derivation(X, -A, -DA):-
   derivation(X,A,DA).
derivation(X, A-B, DA-DB):-
   derivation(X, A,DA),
   derivation(X, B,DB).
derivation(X, A*B, A*DB+DA*B):-
   derivation(X, A, DA),
   derivation(X, B, DB).
derivation(X, A/B, (DA*B-A*DB)/(B*B)):-
   derivation(X, A,DA),
   derivation(X, B,DB).


derivation(X, sin(Y), cos(Y)*DY):-
   derivation(X,Y,DY).
derivation(X, cos(Y), -sin(Y)*DY):-
   derivation(X,Y,DY).
derivation(X, tan(Y), (1/cos(Y)^2)*DY):-
   derivation(X,Y,DY).
derivation(X, cot(Y), -(1/sin(Y)^2)*DY):-
   derivation(X,Y,DY).

derivation(X, e^Y, e^Y*DY):-
   derivation(X,Y,DY).
derivation(X, A^Y, A^Y*ln(A)*DY):-
   derivation(X,Y,DY).
derivation(X, ln(Y), DY/Y):-
   derivation(X,Y,DY).
derivation(X, log(A,Y), 1/(Y*ln(A))*DY):-
   derivation(X,Y,DY).

derivation(X, arcsin(Y), (1-Y^2)^-(1/2)*DY):-
   derivation(X,Y,DY).
derivation(X, arccos(Y), -((1-Y^2)^-(1/2))*DY):-
   derivation(X,Y,DY).
derivation(X, arctan(Y), DY/(1+Y^2)):-
   derivation(X,Y,DY).
derivation(X, arccot(Y), -(DY/(1+Y^2))):-
   derivation(X,Y,DY).
```

$5\frac{d}{dx} = 0$
$x^3\frac{d}{dx} = 3 \cdot x^2$
$\sqrt{x}\frac{d}{dx} = 0.5 \cdot x^{-0.5}$
$\sin(x)\frac{d}{dx} = \cos(x)$

$\cos(x)\frac{d}{dx} = -\sin(x)$
$\tan(x)\frac{d}{dx} = \frac{1}{\cos(x)^2}$
$\cot(x)\frac{d}{dx} = -\frac{1}{\sin(x)^2}$
$e^x\frac{d}{dx} = e^x$
$5^x\frac{d}{dx} = 5^x \cdot 1.60944$
$\ln(x)\frac{d}{dx} = \frac{1}{x}$
$\log_3(x)\frac{d}{dx} = \frac{1}{x \cdot 1.09861}$
$\arcsin(x)\frac{d}{dx} = \left(\left(1-x^2\right)\right)^{-0.5}$
$\arccos(x)\frac{d}{dx} = -\left(\left(1-x^2\right)\right)^{-0.5}$
$\arctan(x)\frac{d}{dx} = \frac{1}{1+x^2}$

6.2 Term-Verarbeitung (TODO)

6.2.1 LaTeX 2$_\varepsilon$ Code von Mathematischen Funktionen

```
latex(A+B):-
   write('\\left('),
   latex(A),
   write('+'),
   latex(B),
   write('\\right)').
latex(-A):-
   write('- \\left('),
   latex(A),
   write('\\right)').
latex(A-B):-
   write('\\left('),
   latex(A),
   write('+'),
   latex(B),
   write('\\right)').
latex(A*B):-
   write('\\left('),
   latex(A),
   write(' \\cdot '),
   latex(B),
   write('\\right)').
latex(A/B):-
   write('\\frac{'),
   latex(A),
   write('}{'),
   latex(B),
   write('}').
latex(A^-B):-
   latex(1/A^B).
latex(A^(B/C)):-
   write('\\sqrt'),
   (C \= 2
```

```
   -> write('['),
      latex(C),
      write(']')
   ; true),
   write('{'),
   latex(A),
   write('}'),
   (B \= 1
   -> write('^{'),
      latex(B),
      write('}')
   ; true).
latex(A^B)
   write('\\left('),
   latex(A),
   write('\\right)^{'),
   latex(B),
   write('}').
latex(log(A,Y)):-
   write('\\log_{'),
   latex(A),
   write('}\\left('),
   latex(Y),
   write('\\right)').
latex(X):-
   atomic(X),
   write(X).
latex(X):-
   X=..[Name|Params],
   write('\\'),
   write(Name),
   write('\\left('),
   ( Params=[P1|Rest]
   -> latex(P1)
      forall( member(Param,Rest),(
         write(','),
         latex(Param)
      ))
   ;  true
   ),
   write('\\right)').
```

6.3 Interne Funktionsweise des *is*-Operators (TODO)

Ausser den internen Behandlungen arithmetischer Funktionen (+, -, *, /, //, mod) kann man auch viele weitere Funktionen benutzen, wie dem Sinus und Cosinus: $Xissin(cos(5)) \Rightarrow X = 0.279873$. Für diese gibt es Prädikate, deren Ergebnis über den letzten Parameter zurück gegeben wird: $cos(5, A), sin(A, X) \Rightarrow X = 0.279873$. Das *is* könnte nun für jedes dieser nicht-arithmetischen Funktionen

```
is( X, Y) :-
   X=..[Name|Params],
   findall( EvParam, (member(Param, Params),is(Param,EvParam)), EvParams),
   append(EvParams, [Y], CallParams),
   Call =.. [Name|CallParams],
   Call.
```

6.4 Natürliche Zahlen

Wie wir in Kapitel 6.2 gesehen haben, kann man Terme in den Köpfen der Klauseln oder mit dem = Operator sehr einfach zerlegen. Dies wollen wir uns nun zu nutze machen, um neue Datenstrukturen zu definieren. Eine Instanz einer Datenstruktur kodieren wir einfach als Term, den wir dann in Prädikaten, die an den schematischen Aufbau der Instanzen angepasst sind, verarbeiten. Wir definieren also gar nicht (wie in imperativen oder funktionalen Sprachen) die Datenstruktur, sondern formulieren einfach Terme, die den Instanzen entsprechen und Prädikate, die Terme eines solchen Aufbaus verarbeiten können.
Als einfaches Beispiel werden häufig die natürlichen Zahlen $\mathbb{N}_0$ genommen. Die 0 wollen wir nun kodieren, zum Beispiel durch den Term, der nur aus dem Atom *zero* besteht. Jetzt könnten wir die 1 als *one* kodieren, aber sinvoller ist es, die Nachfolgerfunktion +1 zu kodieren, zum Beispiel durch $succ(X)$, was den Nachfolger der Zahl darstellt, die durch X kodiert ist.

Natürliche Zahl	wird kodiert als
0	zero
1	succ(zero)
2	succ(succ(zero))
3	succ(succ(succ(zero)))
...	...

und nun wollen wir solche Terme verarbeiten. Als erstes Beispiel wollen wir ein Prädikat schreiben, das aus solchen Termen die natürliche Zahl bestimmt, die sie repräsentieren. *zero* entspricht wie gesagt der 0 und $succ(X)$ entspricht der Zahl, die durch X kodiert ist (was wir rekursiv bestimmen können) +1, also leistet der folgende Code das gewünschte:

```
succ_to_nat(zero, 0).
succ_to_nat(succ(X), Y) :-
   succ_to_nat(X, XNat),
   Y is XNat + 1.

?- succ_to_nat( succ(succ(succ(succ(succ(zero))))), X ).
X = 5.
```

Und umgekehrt

```
nat_to_succ(0, zero).
nat_to_succ(X, succ(X1Succ)) :-
   X > 0,
   X1 is X-1,
   nat_to_succ(X1, X1Succ).
```

```
?- nat_to_succ(7,X).
X = succ(succ(succ(succ(succ(succ(succ(zero)))))))).
```

Jetzt können wir die üblichen arithmetischen Funktionen für unsere Datenstruktur schreiben, wie die Addition. Wer nun an Code denkt, der folgendermaßen aussieht

```
succ_add(X, Y, Z) :-
   succ_to_nat(X,XN),
   succ_to_nat(Y,YN),
   ZN is XN + YN,
   nat_to_succ(ZN, Z).
```

der soll sich in die Ecke stellen und schämen! Der obige Code funktioniert zwar, ist aber trivial und kein Beispiel, aus dem man sonderlich viel lernen würde. Wir wollen lieber Code, der nativ auf unserer Datenstruktur arbeitet und das tun wir analog zu den trivialen Beispielen aus Kapitel 2.15

```
succ_add(zero, Y, Y).
succ_add(succ(X),Y, succ(Z)):-
   succ_add(X,Y,Z).
```

Da hier im Gegensatz zu Kapitel 2.15 kein *is* vorkommt, kann man dieses Prädikat unmittelbar umgekehrt zum subtrahieren benutzen:

```
?- succ_add(succ(succ(succ(zero))), succ(succ(zero)), Z).
Z = succ(succ(succ(succ(succ(zero)))))

?- succ_add(succ(zero), Y, succ(succ(succ(succ(zero))))) ).
Y = succ(succ(succ(zero))).

?- succ_add(X, succ(succ(zero)), succ(succ(succ(succ(zero))))) ).
X = succ(succ(zero)).
```

Übung:

- Schreiben Sie ein Prädikat, das zwei natürliche Zahlen multipliziert! Sie dürfen dabei das *succ_add* Prädikat verwenden

6.5 Listen

Ein Element einer Liste besteht wie in imperativen Sprachen aus einem Objekt des Inhaltstyps und einer Restliste (in imperativen Sprachen in Form eines Zeigers auf das erste Element der Restliste). Wenn wir also ein Element einer Liste modellieren wollen, brauchen wir Terme, die jeweils zwei Parameter haben - ein Objekt und eine Restliste. Die leere Liste wollen wir mit *nil* kodieren und mittels des **zwei**stelligen Terms $cons(Objekt, Restliste)$ ein Element einer Liste kodieren.

Liste	wird kodiert als
[ ]	nil
[1]	cons(1,nil)
$[2, 1]$	cons(2, **cons(1, nil)**)
$[a, b\|X]$	cons(a, cons(b, X))
...	...

Als erstes Beispiel wollen wir ein Prädikat betrachten, das zu einer so dargestellten Liste die Liste in nativer Prolog-Syntax bestimmt:

```
cons_to_list(nil, []).
cons_to_list(cons(Objekt,Rest), [Objekt | RestAlsListe]) :-
   cons_to_list(Rest,RestAlsListe).

?- cons_to_list( cons(a, cons(b, cons(c, nil))), X).
X = [a,b,c]
```

Da hier kein *is* vorkommt, funktioniert dieses Prädikat ebenfalls in beide Richtungen

```
?- cons_to_list( X, [a,b,c] ).
X = cons(a, cons(b, cons(c, nil)))
```

Nun können wir auf solchen Listen die gleichen Prädikate definieren, wie wir sie schon für Listen in Prolog-Syntax kennen, zum Beispiel die Länge einer Liste:

```
cons_length(nil, 0).
cons_length(cons(_,Rest), N) :-
   cons_length(Rest, N1),
   N is N1+1.

?- cons_length( cons(a, cons(b, cons(c, nil))), L).
L = 3
```

und das *member* Prädikat

```
cons_member(Objekt, 0, cons(Objekt,_)).
cons_member(Objekt, Pos, cons(_,Rest)) :-
   cons_member(Objekt, PosInRest, Rest),
   Pos is PosInRest+1.

?- cons_member( b, Pos, cons(a, cons(b, cons(c, nil)))).
Pos = 1
```

auch hier hat das erste Element die Position 0. Intern verwendet Prolog übrigens genau so eine Darstellung von Listen, nur dass die Terme nicht *cons* und *nil* heissen, sondern . und [] weswegen die Anfrage

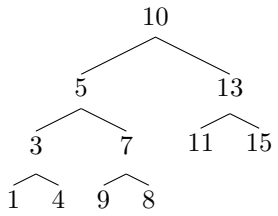
```
?- write( .(1, .(2, .(3, [] ))) ).
```

die Ausgabe $[1, 2, 3]$ bewirkt.
Übung:

- Schreiben Sie das *delete* Prädikat
- Schreiben Sie das *append* Prädikat

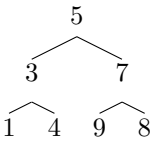
6.6 (Geordnete Binär-) Bäume

Die beiden vorigen Beispiele von Datenstrukturen waren eigentlich relativ unnötig, da Prolog für diese Strukturen ja schon interne Repräsentationen hat und die interne Handhabung natürlicher Zahlen deutlich effizienter ist. Als einfache Beispiele für Datenstrukturen waren sie aber eine gute Vorbereitung, um nun zu komplexeren Strukturen überzugehen. Bäume sind in der Informatik von sehr zentraler Bedeutung, denn sie ermöglichen es, (wie Listen) Daten abzuspeichern und (anders als bei Listen) in kurzer Zeit wieder abzurufen, weswegen sie häufig in Datenbanken verwendet werden. Ein simples Beispiel eines Baumes ist

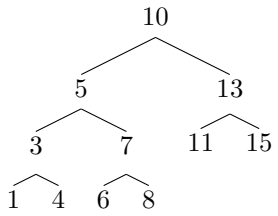


In diesem Beispiel steht an jeder Verzweigung eine Zahl - man könnte zwar auch ganz andere Objekte benutzen, aber vorerst wollen wir uns mit Zahlen begnügen. Die Verzweigungen nennt man **Knoten** (Englisch: **node**) oder auch **Ecken** (Englisch: **vertex**) und die Striche zu den anderen Verzweigungen nennt man **Kanten**[11] (Englisch: **edge**). In dem Beispiel ist 5 der **Vater** (Englisch: **parent**) von 7. 7 und 3 sind die **Kinder** von 5 (Englisch: **children**). 5 und 10 sind die **Vorgänger** (Englisch: **predecessors**) von 3 und 3,7,1,4,9 und 8 sind die **Nachfolger** (Englisch: **successors**) von 5. Den obersten Knoten nennt man **Wurzel** (Englisch: **root**), die Knoten, die keine Nachfolger haben, nennt man **Blätter** (Englisch: **leafs**) und alle anderen Knoten nennt man **innere Knoten** (Englisch: **inner nodes**).

ist ein **Teilbaum** (Englisch: **subtree**) von

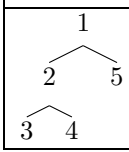


Einen Baum nennt man **geordnet**, wenn an jedem Knoten gilt, dass die Zahl die an dort steht, kleinergleich aller Zahlen im rechten Teilbaum und größergleich aller Zahlen im linken Teilbaum ist. Der Baum im ersten Beispiel ist nicht geordnet (weil 9 ein linker Nachfolger von 7 ist). Der Baum im folgenden Beispiel ist dagegen geordnet:



[11] Die Bezeichnungen „Ecke“ und „Kante“ stammen von Sir William Rowan Hamiltons Dodekaeder-Problem, bei dessen Lösung die Ecken und Kanten eines Dodekaeders mit solchen Kreisen und Strichen dargestellt wurden. Siehe
http://de.wikipedia.org/wiki/Hamiltonkreisproblem

Im Grunde ist ein Baum einer Liste sehr ähnlich, nur dass die Elemente nicht einen, sondern zwei Nachfolger haben. Daher kann man sie auch sehr ähnlich wie Listen implementieren, nur dass die Terme, die ein Element (einen Knoten) darstellen nun nicht mehr zweistellig, sondern dreistellig sind. Der erste Parameter ist der Inhalt (z.B. die Zahl), der an dem Knoten steht, der zweite Parameter ist der Term, der den linken Teilbaum darstellt und der dritte Parameter ist der Term, der den rechten Teilbaum darstellt. Diese Terme werden natürlich um einiges komplexer und unübersichtlicher als die Terme, die Listen darstellten.

Baum	**wird kodiert als**
	node(1, node(2,nil,nil), nil)
	node(1, node(2,nil,nil), node(3,nil,nil))
	node(1,node(2,node(3,nil,nil),node(4,nil,nil)),node(5,nil,nil))

Nun wollen wir erst einmal zum warm werden drei simple Prädikate für so dargestellte Bäume schreiben: Die Anzahl der Knoten eines Baumes ist die Anzahl der Knoten im linken Teilbaum + die Anzahl der Knoten im rechten Teilbaum + 1 (für die Wurzel)

```
nodes( nil, 0 ).
nodes( node( _, Links, Rechts), N ) :-
   nodes(Links, L),
   nodes(Rechts, R),
   N is L+R+1.

?- nodes( nil, X).
X = 0

?- nodes( node( 1, node(2,nil,nil), nil), X ).
X = 2

?- nodes( node( 1, node(2,nil,nil), node(3,nil,nil)), X ).
X = 3
```

die **Höhe** eines Baumes ist die Länge eines längsten Weges zwischen Wurzel und einem Blatt. Rekursiv gedacht ist das das Maximum der Höhen des rechten und linken Teilbaums + 1 (für die Wurzel). Die Höhe eines leeren Baums ist als 0 definiert.

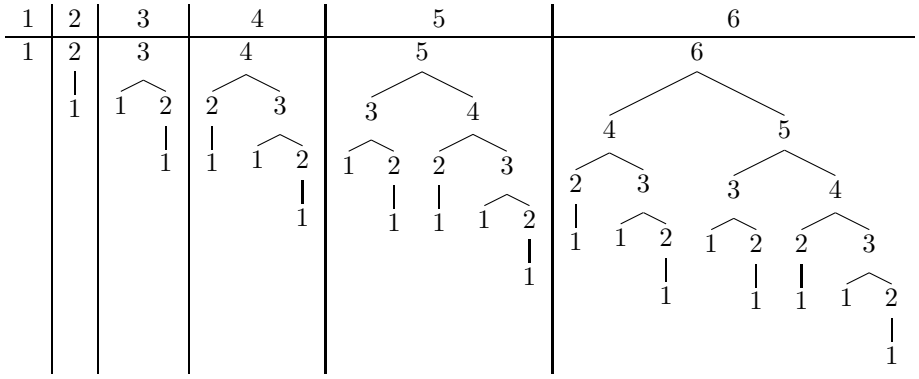
```
height( nil, 0 ).
height( node( _, Links, Rechts), N ) :-
   height(Links, L),
   height(Rechts, R),
   (R >= L -> N is R+1 ; N is L+1).
```

```
?- height( nil, X).
X = 0

?- height( node( 1, node(2,nil,nil), nil), X ).
X = 2

?- height( node( 1, node(2,nil,nil), node(3,nil,nil)), X ).
X = 2
```

die Anzahl der Blätter eines Baumes, der nur aus einer Wurzel besteht, ist 1, die Anzahl der Blätter eines leeren Baumes ist 0. Für jeden anderen Baum ist die Anzahl der Blätter gleich der Anzahl der Blätter des rechten Teilbaums + der Anzahl der Blätter des linken Teilbaums.

```
leafs( nil, 0 ).
leafs( node(_, nil, nil), 1 ).
leafs( node(_, Links, Rechts), N ) :-
   % wenn Links=nil und Rechts=nil waere, dann
   % handelte es sich um ein Blatt, was schon im
   % vorigen Fall behandelt wurde.
   (Links = nil -> Rechts \= nil ; true),
   leafs(Links, L),
   leafs(Rechts, R),
   N is L+R.

?- leafs( node( 1, node(2,nil,nil), nil), X ).
X = 1

?- leafs( node( 1, node(2,nil,nil), node(3,nil,nil)), X ).
X = 2
```

Nach diesen einfachen Aufwärmübungen wollen wir anfangen, solche Bäume zu generieren - ein gutes Beispiel sind die **Fibonacci-Bäume**. Der 0-te Fibonacci-Baum ist ein leerer Baum (nil), der 1-te Fibonacci-Baum besteht nur aus einer Wurzel mit Inhalt 1, ohne Nachfolger. Der n-te Fibonacci-Baum hat als Inhalt der Wurzel die Zahl n, als rechten Teilbaum den (n-1)-ten Fibonacci-Baum und den (n-2)-ten Fibonacci-Baum als linken Teilbaum. Die ersten 6 Fibonacci-Bäume sind



```
fibo_tree( 0, nil ).
fibo_tree( 1, node( 1, nil, nil) ).
fibo_tree( N, node( N, L, R) ):-
   N >= 2,
   N1 is N-1,
   N2 is N-2,
   fibo_tree( N1, R ),
   fibo_tree( N2, L ).

?- fibo_tree( 2, X ).
X = node( 2, node( 1, nil, nil), nil ).

?- fibo_tree( 3, X ).
X = node( 3, node( 2, node( 1, nil, nil), nil ), node(1,nil,nil) )
```

Übung:

- Welcher Zusammenhang besteht zwischen Fibonacci-Bäumen und den Fibonacci-Zahlen? (vgl. 2.15)
- Schreiben Sie ein Prädikat, das Elemente in ungeordneten Bäumen sucht (ähnlich dem *member* Prädikat für Listen)
- Schreiben Sie ein Prädikat, das Elemente in geordneten Bäumen sucht

6.6.1 Elemente Einfügen

Wie bei den Listen geht es hier nicht darum, Zeiger zu verbiegen, denn wir sind immernoch nicht in einer imperativen Sprache. Ein Element fügen wir in Bäume ein, indem wir aus dem Baum und dem Element einen neuen Baum berechnen, der dem Ursprungsbaum mit eingefügtem Element entspricht. Dabei wollen wir davon ausgehen, dass der Ursprungsbaum geordnet ist und wollen das neue Element so einfügen, dass der resultierende Baum wieder geordnet ist - das heisst, dass wenn man in einem Baum $node(X, Links, Rechts)$ das Element Y einfügen will, dann muss man Y im linken Teilbaum einfügen, falls $Y \leq X$ und erhält $LinksNeu$ als neuen linken Teilbaum, den man mit einer Kopie des rechten Teilbaums und X zum Ergebnisbaum zusammensetzt. Analog funktioniert das Einfügen im rechten Teilbaum, falls $Y > X$. Sobald man eine freie Position zum Einfügen findet (nil), gibt man den Baum zurück, der nur aus Y besteht.

```
tree_insert(Y, nil, node(Y,nil,nil)).
tree_insert(Y, node(X, Links, Rechts), node(X, Links, RechtsNeu)) :-
   Y > X,
   tree_insert(Y, Rechts, RechtsNeu).
tree_insert(Y, node(X, Links, Rechts), node(X, LinksNeu, Rechts)) :-
   Y =< X,
   tree_insert(Y, Links, LinksNeu).
```

Und nun wollen wir noch gleich ein (end-rekursives) Prädikat schreiben, das eine Liste von Elementen in einen Baum einfügt:

```
tree_insert_list([], Tree, Tree).
tree_insert_list([X|R], Tree, Result) :-
   tree_insert(X,Tree,NextTree),
   tree_insert_list(R,NextTree,Result).

?- tree_insert_list([5] ,nil, Tree).
Tree = node(5, nil, nil) .

?- tree_insert_list([5,10] ,nil, Tree).
Tree = node(5, nil, node(10, nil, nil)) .

?- tree_insert_list([5,10,2] ,nil, Tree).
Tree = node(5, node(2, nil, nil), node(10, nil, nil)) .

?- tree_insert_list([5,10,2,3] ,nil, Tree).
Tree = node(5, node(2, nil, node(3, nil, nil)), node(10, nil, nil)) .
```

6.6.2 LaTeX 2_ε Code generieren

Nicht ganz uneigennützig will ich nun ein Prädikat entwickeln, das aus einem Baum einen Output generiert, der den Baum in LaTeX2e Syntax beschreibt - Mit diesem Code werde ich die Bäume aus den Beispielen generieren. Die LaTeX 2_ε Syntax für Bäume ist unter

http : *//www.essex.ac.uk/linguistics/clmt/latex4ling/trees/qtree/*

beschrieben. Eine vollständigere (aber weniger verständliche) Dokumentation des *qtree* Paketes ist unter

http : *//www.ling.upenn.edu/advice/latex/qtree/qtreenotes*221.*pdf*

zu finden. Ich halte es für sinnvoll, ein allgemeines Prädikat *latex* zu definieren, das später nicht nur Bäume, sondern alle möglichen Datenstrukturen in LaTeX 2_ε Code umwandeln kann. Gibt man dann den Inhalt eines Knotens nicht mit *write*, sondern mit *latex* aus, dann kann Prolog hier auch Bäume ausgeben, deren Knoten keine Zahlen sind, sondern z.B. Listen, andere Bäume usw.

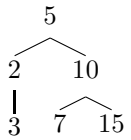
```
latex( X ) :-
   nonvar(X),
   integer(X),
   write(X).

tree_to_latex( nil ).
tree_to_latex( node(X,L,R) ) :-
   write('[.{'),
   latex(X),
   write('} '),
   tree_to_latex( L ),
   write(' '),
   tree_to_latex( R ),
   write(' ]').
```

```
latex( node(X,L,R) ) :-
   write('\\Tree '),
   tree_to_latex(node(X,L,R)).
```

Nun erhalte ich auf die Anfrage

```
?- tree_insert_list([5,10,2,3,15,7], nil, Tree), latex(Tree).
```

einen Code, der LaTeX 2ε den folgenden Baum zeichnen lässt:



und die Codes für die Fibonacci-Bäume habe ich generiert durch die Anfragen

```
?- fibo_tree(1, Tree), latex(Tree).
?- fibo_tree(2, Tree), latex(Tree).
...
```

6.6.3 Plätten von Bäumen

Man bezeichnet es als **Plätten** (Englisch: **flatten**), wenn man aus einem Baum eine Liste macht. Dazu pättet man den linken und rechten Teilbaum und fügt die daraus entstehenden Listen aneinander, wobei der Inhalt des aktuellen Knotens dazwischen eingefügt wird.

```
tree_flatten(nil, []).
tree_flatten(node(X, Links, Rechts), Flat):-
   tree_flatten(Links, L),
   tree_flatten(Rechts, R),
   append(Links, [X|Rechts], Flat).
```

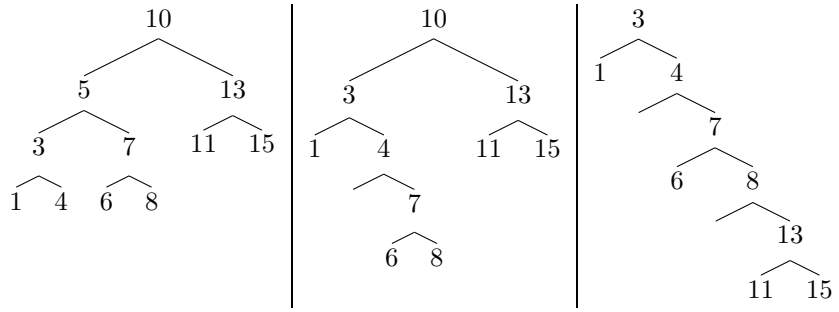
Wie man leicht sieht ist das Ergebnis des Plättens eines geordneten Baumes eine sortierte Liste. Daher kann man den folgenden Sortieralgorithmus programmieren:

```
treesort(Liste, Sortiert) :-
   tree_insert_list(Liste, nil, Tree),
   tree_flatten(Tree,Sortiert).
```

Wir haben den Baum hier in ***infix*-Ordnung** geplättet, weil X zwischen den geplätteten Teilbäumen eingefügt wurde. Man könnte es aber auch davor anfügen (***prefix*-Ordnung**) mit $append([X|Links], Rechts, Flat)$ oder dahinter (***postfix*-Ordnung**) mit $append(Links, Rechts, Temp), append(Temp, [X], Flat)$.

6.6.4 Löschen von Knoten, Balancierte Bäume

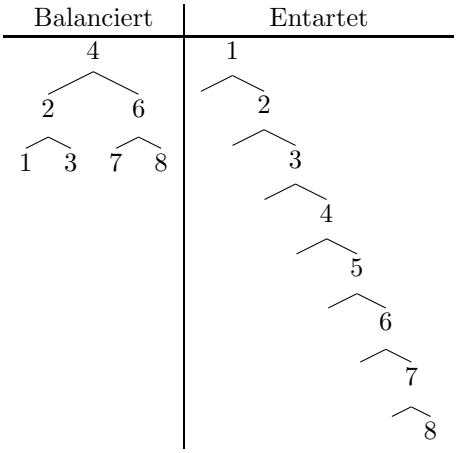
Wenn man ein Element X aus einem Baum löschen will, bleiben im Regelfall zwei Teilbäume übrig. Man will einerseits alle Elemente behalten und die Ordnung soll bestehen bleiben, andererseits soll das Löschen schnell gehen, weswegen man nicht einfach alle Elemente des einen Teilbaums nacheinander in den anderen einfügen kann. Man kann sich leicht überlegen, dass die Ordnung erhalten bleibt, wenn beim Löschen des Elements X, welches die Teilbäume L und R hat, das Ergebnis aus L besteht, mit R rechts an den rechtesten Knoten von L angehängt. Im folgenden Beispiel wird mit dieser Methode zuerst die 5 und dann die 10 gelöscht:



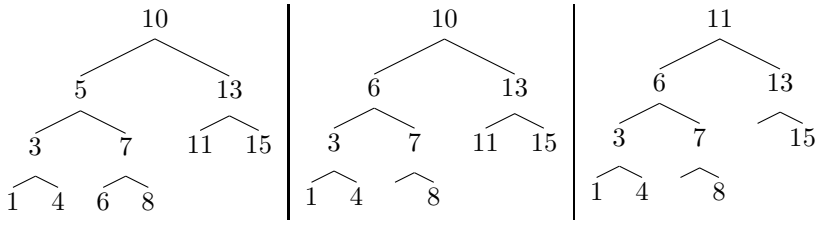
Beim Löschen der 5 rutscht die 3 hoch an die Position der gelöschten 5 und die 7 wird rechts unter der 4 angehangen. Beim Löschen der 10 rutscht die 3 hoch an die Position der gelöschten 10 und die 13 wird rechts unter der 8 angehangen.

```
put_rightmost(nil, R, R).
put_rightmost(node(X,L,R), Ins, node(X,L,NewR)):-
   put_rightmost(R,Ins,NewR).

tree_delete_bad(X, node(X, L, R), LUndR) :-
   put_rightmost(L,R, LUndR).
tree_delete_bad(X, node(Y, L, R), node(Y,L,NewR)):-
   Y < X,
   tree_delete_bad(X,R,NewR).
tree_delete_bad(X, node(Y, L, R), node(Y,NewL,R)):-
   Y > X,
   tree_delete_bad(X,L,NewL).
```

Das Problem an dieser Methode ist, dass der Baum hierdurch sehr schnell sehr unbalanciert wird. Im ersten Baum würde man 3 Rekursionen benötigen, um die 15 zu finden, im dritten Baum benötigt man dagegen schon 6 Rekursionen. Man möchte gerne möglichst **balancierte Bäume** haben - das sind Bäume, bei denen an jedem Knoten die Anzahl der linken Nachfolger mit der Anzahl der rechten Nachfolger nahezu überenstimmt. Das Gegenteil von balancierten Bäumen sind **entartete Bäume** - diese haben eine maximale Differenz zwischen der Anzahl der rechten und linken Nachfolger (d.h. jeder Knoten hat auf einer Seite 0 Nachfolger).



Logischerweise dauert es in entarteten Bäumen mit n Elementen etwa genauso lange, einen Wert zu finden, wie in einer normalen Liste (bis zu n Rekursionen), wogegen man in einem balancierten Baum mit n Elementen nur etwa $log(n)$ Rekursionen benötigt, daher sollte man z.B. in Datenbank-Anwendungen versuchen, möglichst balancierte Bäume zu erreichen. Mit der Lösch-Methode von oben macht man aber sehr schnell aus einem balancierten Baum einen (nahezu) entarteten Baum. In der Praxis ist daher die folgende Methode sinnvoller: Man lässt den Knoten, an dem X stand, bestehen (wodurch die Balance etwa erhalten bleibt), ersetzt X aber durch ein Element, das man löschen kann, ohne die Balance zu zerstören (z.B. ein Blatt) und löscht dafür das andere Element. Damit die Balance erhalten bleibt, genügt es, ein Element zu wählen, das nur einen Nachfolger hat und damit die Ordnung erhalten bleibt, kann man X durch das linkeste Element des rechten Teilbaums oder das rechteste Element des linken Teilbaums ersetzen (welche beide jeweils nur einen Nachfolger haben). Wir wählen jetzt einfach willkürlich, das linkeste Element des rechten Teilbaums.



Wenn man ein Element löscht, das nur einen Nachfolger N hat (so, wie das Element, das wir wählen), dann ist das Ergebnis einfach genau N - daher funktioniert das $get\_leftmost$ Prädikat im folgenden Code zum Löschen des linkesten Knotens und Zurückgeben des Inhalts, der dort stand. das $get\_leftmost$ kann man im $tree\_delete$ Prädikat aber nur dann anwenden, wenn X überhaupt einen rechten Nachfolger hat (2. Klausel), anderenfalls können wir X aber eh direkt löschen, indem wir als Ergebnis des Löschens einfach den linken Nachfolger zurückgeben (1. Klausel). Mit der 3. und 4. Klausel durchlaufen wir den Baum auf der Suche nach dem X, das wir löschen wollen.

```
get_leftmost(node(X,nil,R), X, R).
get_leftmost(node(X,L,R), Leftmost, node(X,LRest,R)):-
```

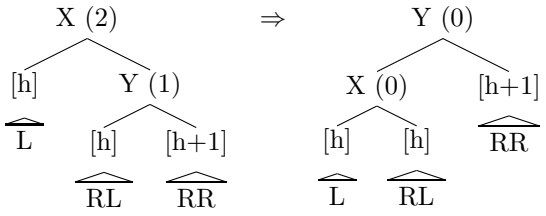
```
   get_leftmost(L, Leftmost, LRest).

tree_delete(X, node(X, L, nil), L).
tree_delete(X, node(X, L, R), node(Leftmost,L, RRest)) :-
   get_leftmost(R, Leftmost, RRest).

tree_delete(X, node(Y, L, R), node(Y,L,NewR)):-
   Y < X,
   tree_delete(X,R,NewR).
tree_delete(X, node(Y, L, R), node(Y,NewL,R)):-
   Y > X,
   tree_delete(X,L,NewL).
```

6.6.5 AVL-Bäume

Wie wir gesehen haben ist es wünschenswert, balancierte Bäume zu haben, aber wenn wir einfach die *tree_insert* Methode von oben benutzen, erhalten wir schon einen entarteten Baum, wenn wir nur eine sortierte Liste von Zahlen einfügen, was kein ungewöhnliches Szenario ist. Georgi Maximowitsch **A**delson-**V**elsky[12] und Jewgeni Michailowitsch **L**andis entwickelten eine andere Methode zum Einfügen von Elementen in einen Baum, die eine gewisse Balance erhält. Hierzu erhält jeder Knoten noch einen Integer, der die Zahlen -1, 0 oder 1 annehmen kann, was der **Differenz der Höhen** seiner Teilbäume entspricht. Wenn beim Einfügen eines Elements ein Teilbaum um 2 höher wird, als der andere (der zusätzliche Integer also den Wert 2 oder -2 annehmen würde), dann sortiert man einige Elemente um, wobei man die Ordnung erhält, aber die Differenz zwischen den Höhen ausgleicht. Dieses Umsortieren nennt man eine **AVL-Rotation**. Hierbei sind 4 Fälle zu unterscheiden, jenachdem, welcher Teilbaum das Ungleichgewicht der Höhen verursacht, also welcher Teilbaum eine Höhe hat, die um 2 größer ist, als die Höhe eines anderen Teilbaums. Wenn es der rechte Nachfolger des rechten Nachfolgers von X ist, dann wenden wir die **RR-Rotation** an. Ist es der linke Nachfolger des rechten Nachfolgers, dann wenden wir die **RL-Rotation** an. Analog gibt es die **LL-Rotation** falls das Ungleichgewicht vom linken Nachfolger des linken Nachfolgers verursacht wird und die **LR-Rotation**, falls es der rechte Nachfolger des linken Nachfolgers ist. Da die Fälle LL bzw. LR nur gespiegelte RR- bzw. RL-Rotationen sind, zeige ich nur die Fälle RR und RL.

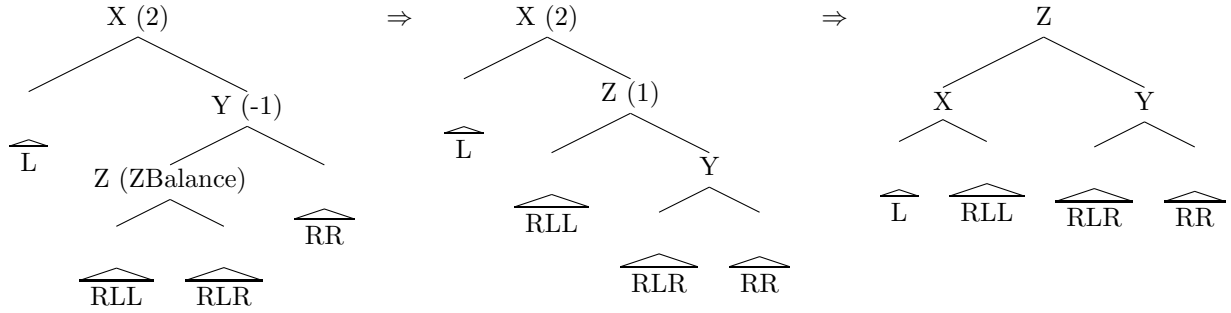
RR-Rotation: Ich muss jetzt erst einmal einigen Teilen des Baumes Namen geben, um die Schritte vernünftig beschreiben zu können. Das nächste Bild zeigt grafisch, was wie genannt wird. Die Balancen stehen dabei in den runden Klammern, die Höhen der Teilbäume in eckigen Klammern. Den linken Teilbaum von X wollen wir mit L bezeichnen, den rechten Nachfolger mit Y, den linken Teilbaum von Y nennen wir RL und den rechten Teilbaum RR. Die Höhe von L wollen wir mit h bezeichnen. Diese Situation erlaubt nur die Fälle, dass RL die Höhe h und RR die Höhe $h + 1$ hat, da sonst schon vor dem Einfügen des Elements ein zu großes Ungleichgewicht bestanden hätte, schon vorher rotiert worden wäre oder nicht die RR-Rotation angewendet würde.

[12] Man findet verschiedene Schreibweisen für den russischen Namen



Im resultierenden Baum haben X und Y Balance 0 (wie man an den oben genannten Höhen der Teilbäume L, RR und RL sieht) und die Ordnung blieb erhalten.

RL-Rotation Diese Rotation ist etwas komplizierter als die RR-Rotation, man kann sie sich folgendermaßen vorstellen: eine RR bzw LL Rotation sorgt dafür, dass auf der einen Seite ein Teilbaum schrumpft und auf der anderen Seite wächst. Benutzt man eine LL-Rotation an Y, dann verändert der Baum sich so, dass das Ungleichgewicht nun vom rechten Nachfolger von Y verursacht wird, weswegen man danach die RR-Rotation an X anwenden kann. Deshalb nennt man die RL- und LR-Rotation auch **Doppelrotationen**, obwohl man den Zwischenschritt auch umgehen und das Ergebnis direkt angeben kann:



Im folgenden Code erlaube ich in der LL- und RR-Rotation auch den Fall, dass der Nachfolger die Balance 0 hat. Beim Einfügen kann das zwar nicht passieren, aber beim Löschen, weswegen ich es hier schon erlaube.

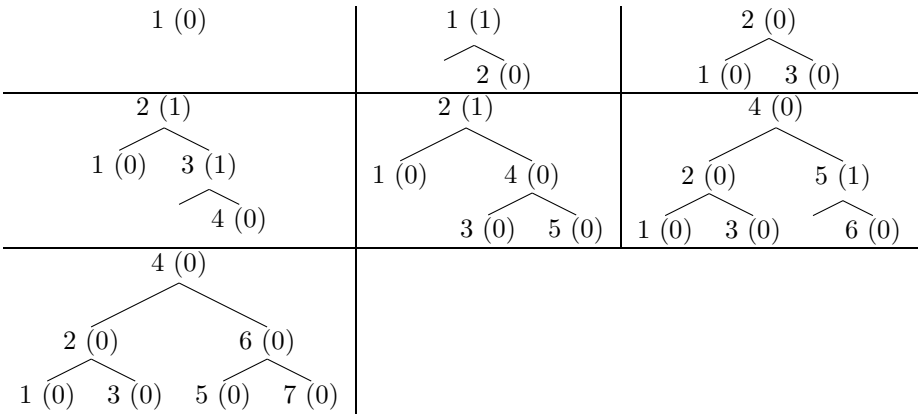
```
% RR-Rotation
avl_rotation(node(X,2,L,node(Y,YBalance,RL,RR)),
             node(Y,BR,node(X,BL,L,RL),RR) ) :-
   YBalance=  1  -> BL =  0, BR =  0 ;
   YBalance=  0  -> BL =  1, BR = -1.
% RL-Rotation
avl_rotation( node(X,2,L,node(Y,-1,node(Z,ZBalance,RLL,RLR),RR)),
             node(Z,0,node(X,BL,L,RLL),node(Y,BR,RLR,RR)) ):-
   ZBalance=  1  -> BL = -1, BR =  0 ;
   ZBalance=(-1) -> BL =  0, BR =  1.
% LL-Rotation
avl_rotation( node(X,-2,node(Y,YBalance,LL,LR),R),
             node(Y,BL,LL,node(X,BR,LR,R)) ):-
   YBalance=(-1) -> BL =  0, BR =  0 ;
   YBalance=  0  -> BL =  1, BR = -1.
```

```
% LR-Rotation
avl_rotation( node(X,-2,node(Y,1,LL,node(Z,ZBalance,LRL, LRR)), R),
              node(Z,0,node(Y,BL,LL,LRL), node(X,BR,LRR,R)) ):-
   ZBalance=  1  -> BL =  1, BR=  0 ;
   ZBalance=(-1) -> BL =  0, BR= -1.
```

Mit Hilfe dieser Rotationen können wir nun Elemente in einen Baum einfügen, indem wir sie wie gehabt Rekursiv nach unten wandern lassen, lassen den Rekursionsaufruf jetzt aber eine Information $RekHeightChanged$ darüber zurückliefern, ob die Höhe des Teilbaums gewachsen ist. Ist sie gewachsen, dann ändern wir den Integer für die Balance - erreicht er den Wert 2 oder -2, dann wird rotiert. Danach müssen wir noch über $HeightChanged$ die Information zurück liefern, ob der Baum nun selbst gewachsen ist. Er ist nicht gewachsen, wenn die AVL-Rotation das Wachsen ausgeglichen hat. Wenn der Teilbaum, in den wir X eingefügt haben, vorher kleiner war, als der andere Teilbaum, dann hat das Einfügen den Teilbaum $node(X, ..., L, R)$ ausbalanciert und seine Höhe ist insgesamt nicht gewachsen. Ist der Teilbaum, in den wir X eingefügt haben, nicht gewachsen (weil schon irgendwo weiter unten rotiert wurde oder ein Ungleichgewicht ausbalanciert wurde), dann kann der Teilbaum $node(X, ..., L, R)$ auch nicht gewachsen sein. Der einzige noch übrige Fall ist, dass $node(X, ..., L, R)$ vorher balanciert war und der Teilbaum, in den wir eingefügt haben, gewachsen ist, womit auch $node(X, ..., L, R)$ wächst. Übrigens kann es nicht vorkommen, dass mehr als 1 mal pro eingefügtem Element rotiert wird.

```
avl_insert(Q, nil, node(Q, 0, nil,nil), 1).
avl_insert(Q, node(X, Balance, L, R), Tree, HeightChanged):-
   (X < Q
   -> (avl_insert(Q, R, NewR, RekHeightChanged),
      NewL = L,
      NewBalance is Balance + RekHeightChanged)
   ;  (avl_insert(Q, L, NewL, RekHeightChanged),
      NewR = R,
      NewBalance is Balance - RekHeightChanged)
   ),
   (avl_rotation(node(X,NewBalance,NewL,NewR), Tree)
   -> HeightChanged = 0
   ;  (Tree=node(X,NewBalance,NewL,NewR),
      ((RekHeightChanged=0 ; NewBalance=0)
      -> HeightChanged = 0
      ;  HeightChanged = 1)
      )
   ).
```

Wer schon eine Implementierung von AVL-Bäumen in imperativen Sprachen gesehen hat, wird vermutlich kaum glauben können, dass dieser kurze Code in Prolog ausreicht. Fügen wir nun mit diesem *insert* die Elemente 1-7 in sortierter Reihenfolge in unseren Baum ein, dann erhalten wir nacheinander die folgenden Bäume:



6.6.6 Elemente aus AVL-Bäumen löschen

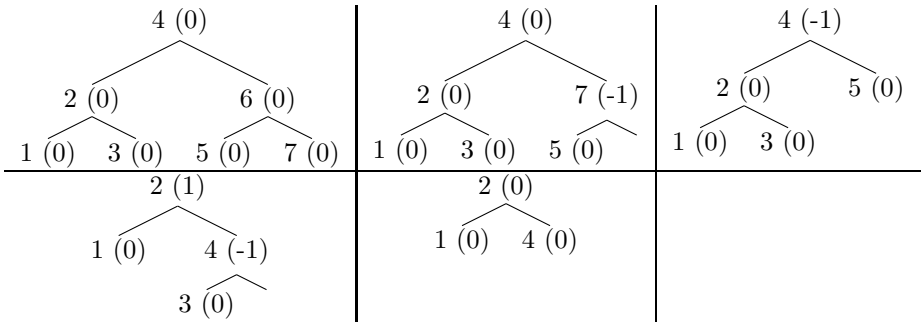
Nun können wir schon Bäume so generieren, dass sie relativ balanciert sind, aber natürlich wollen wir aus Bäumen auch Elemente löschen können, wobei wir wieder die AVL-Bedingung erhalten wollen, dass an jedem Knoten die Differenz der Höhen der Teilbäume höchstens 1 ist. Um dieses Thema drücken sich die meisten Fachbücher und sagen lediglich, dass es so ähnlich wie das Einfügen geht oder dass es offensichtlich ist. Überhaupt ist das Thema „Löschen aus Bäumen“ (nicht nur aus AVL-Bäumen) recht unbeliebt in der Literatur. Meine Vermutung ist, dass die meisten Autoren die Praxistaugliche Lösch-Methode nicht kennen - in den Fachbüchern findet man bestenfalls die schlechte Lösch-Methode, die die Balance drastisch zerstörte. Die „gute“ Lösch-Methode können wir aber tatsächlich relativ einfach erweitern, sodass sie die AVL-Bedingung erhält und das funktioniert tatsächlich so ähnlich, wie das Einfügen: wir löschen wie gehabt und falls die AVL-Bedingung verletzt wird, rotieren wir. Das kann allerdings nicht nur auf der Suche nach dem Element passieren, das gelöscht werden soll, sondern auch beim Entfernen des linkesten Elements des rechten Teilbaums, weswegen man die AVL-Rotationen und Weitergabe der $HeightChanged$ Information an mehreren Stellen einfügen muss. Um dabei nicht zu viele neue Zeilen zu erhalten, benutzen wir hierzu ein zusätzliches Prädikat avl_fix:

```
avl_fix(node(X,NewBalance,L,R),Fixed,RekHeightChanged,HeightChanged):-
   avl_rotation(node(X,NewBalance,L,R),Fixed)
   -> HeightChanged = 0
   ;  (Fixed=node(X,NewBalance,L,R),
      ((RekHeightChanged=0 ; NewBalance\=0)
      -> HeightChanged = 0
      ;  HeightChanged = 1)
   ).
```

Interessanterweise wird beim Löschen das $HeightChanged$ dann auf 1 gesetzt, wenn die Balance durch die Änderung $= 0$ wird, wogegen beim Einfügen $HeightChanged$ auf 1 gesetzt wurde, wenn die Balance durch das Einfügen $\neq 0$ wurde. Dennoch ist das logisch, denn beim Löschen ändert sich die Höhe, wenn der größere der beiden Teilbäume schrumpft, also dann, wenn die Balance 0 wird, wogegen die Höhe sich beim Einfügen ändert, wenn ein Teilbaum wächst und der andere nicht schon vorher größer war.

```
avl_get_leftmost(node(X,_,nil,R), X, R, 1).
avl_get_leftmost(node(X,Balance,L,R), Leftmost, Tree, HeightChanged):-
   avl_get_leftmost(L, Leftmost, LRest, RekHeightChanged),
   NewBalance is Balance + RekHeightChanged,
   avl_fix(node(X,NewBalance,LRest,R),Tree, RekHeightChanged, HeightChanged).

avl_delete(Q, node(Q, _, L, nil), L, 1).
avl_delete(Q, node(Q, Balance, L, R), Tree, HeightChanged) :-
   avl_get_leftmost(R, Leftmost, RRest, RekHeightChanged),
   NewBalance is Balance - RekHeightChanged,
   avl_fix(node(Leftmost, NewBalance, L, RRest), Tree, RekHeightChanged, HeightChanged).

avl_delete(Q, node(X, Balance, L, R), Tree, HeightChanged):-
   X < Q,
   avl_delete(Q,R,NewR,RekHeightChanged),
   NewBalance is Balance - RekHeightChanged,
   avl_fix(node(X,NewBalance,L,NewR), Tree, RekHeightChanged, HeightChanged).
avl_delete(Q, node(X, Balance, L, R), Tree, HeightChanged):-
   Q =< X,
   avl_delete(Q,L,NewL,RekHeightChanged),
   NewBalance is Balance + RekHeightChanged,
   avl_fix(node(X,NewBalance,NewL,R), Tree, RekHeightChanged, HeightChanged).
```

Löschen wir mit dieser *delete* Methode nacheinander die Elemente 6,7,5,3 aus dem Baum von oben, dann erhalten wir die folgenden Bäume:



Zum Abschluss dieses Kapitels will ich noch erwähnen, dass die Struktur der Fibonacci-Bäume dem Worst-Case für AVL-Bäume entsprechen - es gibt keine höheren Bäume, die das AVL-Kriterium erfüllen, ohne mehr Knoten zu haben, als ein Fibonacci-Baum. Eine weitere Methode zum balancieren von Bäumen sind die rot-schwarz Bäume, die in der Praxis eine etwas bessere Performance haben, auf die ich aber hier nicht eingehen will.

6.7 Graphen (TODO)

6.8 Automaten (TODO)

6.9 Turing-Maschinen (TODO)

7 Praxisrelevante Techniken

7.1 Multithreading (TODO)

Dieses Kapitel betrifft ausschließlich SWI Prolog, mir ist kein anderer Prolog Interpreter / Compiler bekannt, der Multithreading unterstützt.

WARNUNG
Bevor man anfängt, Programme mit mehreren Threads zu schreiben, sollte man sich unbedingt mit Deadlocks und Mutual Exclusion auseinandersetzen, da hier eine Fehlerquelle liegt, die häufig schwere Fehler verursacht, welche selbst von Experten kaum gefunden werden, daher sollte man gründlich planen, was man programmiert, wenn man mit mehreren Threads arbeitet!

$$http://de.wikipedia.org/wiki/Deadlock$$

$$http://de.wikipedia.org/wiki/Mutex$$

Multithreading ist ein wichtiges Fachgebiet der Informatik und in Zukunft wird es aller Voraussicht nach noch um einiges wichtiger werden, denn die Geschwindigkeit einzelner Prozessorkerne scheint bei einem Maximum von 3-4 GHz angekommen zu sein. Deshalb werden in den letzten Jahren Prozessoren mit immer mehr Kernen verkauft, deren jeweilige Geschwindigkeit aber etwa konstant bleibt. Jeder Thread kann nur auf einem Prozessorkern gleichzeitig laufen, also wird sich die Geschwindigkeit eines Single-Thread Programms zukünftig voraussichtlich nicht mehr signifikant verbessern - die einzige Möglichkeit, wie Programme zukünftig von der Weiterentwicklung von Prozessoren profitieren können, ist, mehrere Threads zu benutzen.
Ausserdem gibt es heute schon Supercomputer, die mehrere tausend Prozessoren haben, aber Single-Thread Programme würden auf diesen etwa so schnell laufen, wie auf einem Heim-PC. Extrem rechenintensive Anwendungen, die auf einem solchen Supercomputer ausgeführt werden könnten (z.B. Mathematische Beweiser, Datamining Auswertungen, Passwort-Cracker, etc.) sollten unbedingt möglichst viele Threads benutzen.

Einen neuen Tread erstellt man mit dem Prädikat $thread\_create$, welches ein Goal seinen Status kann man abfragen mit dem Prädikat $thread_property$, und man kann warten, bis er

7.2 Mit externen Programmen kommunizieren (TODO)

7.3 Netzwerk-Sockets (TODO)

7.4 Dateien Downloaden (TODO)

7.5 XML und HTML Dateien Lesen (TODO)

7.6 Windows-Registry (TODO)

8 Tips für die Praxis

8.1 Modularisierung (Programme aus mehreren Quellcodes)

Man kann natürlich auch Quellcodes in andere Quellcodes „importieren“ (wie *import* unter Java oder *#include* unter C) ansonsten müsste man ein Programm ja komplett in einer Quellcode Datei schreiben, was bei größeren Projekten absolut inakzeptabel wäre!
Um eine andere Datei zu importieren (Prolog Programmierer bezeichnen das gerne als „konsultieren“ - ein Programm wird betrachtet als ein Experte auf einem Gebiet, der jetzt zurate gezogen wird) schreibt man einfach am Anfang der Datei

$$:-include(dateiname).$$

wobei *dateiname* der Pfad zur anderen Datei ist - und zwar relativ zum Pfad, in dem sich die Datei befindet, in der das include benutzt wird. Der Dateiname wird als String in einzelnen Anführungszeichen übergeben.

8.2 Goals automatisch starten

Manchmal muss ein Programm irgendwelche Dinge erledigen, bevor es bereit ist, Anfragen zu beantworten - z.B. schalten Robotersteuerungen die einzelnen Geräte (Sensoren, Motoren) ein, manche Programme überprüfen, in welchem Prolog Interpreter sie gerade ausgeführt werden und passen sich an den jeweiligen Interpreter an, setzen globale pseudo-Variablen etc.
Wenn ein Programm ein Goal ausführen soll, bevor Prolog anfängt, Anfragen einzulesen, benutzt man

$$:-initialization(Goal).$$

Damit kann man auch eine andere Benutzerführung implementieren als Prologs standardmäßiges „Eingaben lesen und auswerten“. Mit dieser Methode habe ich schon einmal eine Robotersteuerung so modifiziert, dass man nicht mehr das Skript-Ausführungs-Prädikat aufrufen und ihm ein Skript übergeben musste, sondern man konnte gleich Roboter-Skripte eingeben, die Prolog dann selbstständig an das zuständige Prädikat übergeben hat. Dadurch hatte man viel mehr das Gefühl, mit einem Roboter zu kommunizieren, als vorher.

8.3 Verteilte Klauseln

Es ist nicht immer sinnvoll, die Klauseln eines Prädikats beieinander liegend zu programmieren. Zum Beispiel könnte man eine Datenbank mit Kundendaten haben, in der die Daten nach Kunden sortiert sein sollen

```
kunde(23).
name(23, 'Hans Meier').
adresse(23,'Musterweg 42').

kunde(24).
name(24, 'Hermann Mueller').
...
```

Will man die Klauseln eines Prädikats f, welches n Parameter hat, verteilen, dann muss man Prolog anweisen, dies zu erlauben. Das tut man, indem man irgendwo vor der ersten Klausel des Prädikats schreibt

```
:- discontiguous(f/n).
```

z.B.

```
:- discontiguous(adresse/2).
```

Normalerweise schreibt man alle diese Anweisungen ganz am Anfang des Quellcodes. Will man erlauben, dass die Klauseln eines Prädikats über mehrere Quellcode-Dateien verteilt sind, dann schreibt man

```
:- multifile(f/n).
```

z.B.

```
:- multifile(adresse/2).
```

8.4 Prüfen ob Variablen gebunden sind

Man hat i.A. keinen Einfluss darauf, welche Parameter gebunden sind, wenn das Programm zB von einem DAU benutzt wird oder auch von einem Hacker auf Schwachstellen untersucht wird. Man kann Fälle, in denen „unpassende“ Belegungen übergeben wurden aber auch im Quellcode **abfangen**. Hierzu benutzt man die System-Prädikate *var* und *nonvar*. *var* Akzeptiert einen Parameter, wenn er ungebunden ist, *nonvar* genau umgekehrt.
Z.B. habe ich in Kapitel 2.11 (Rückgabewerte) ein Prädikat $plus(A, B, C)$ erwähnt, dass $A + B$ an C bindet, oder aber $C - A$ an B bzw $C - B$ an A bindet - je nachdem, welche Variablen ungebunden sind. Dieses Prädikat kann man **nicht** so formulieren:

```
plus(A,B,C):-  /*  FALSCH!!!  */
   C is A+B.
plus(A,B,C):-  /*  FALSCH!!!  */
   A is C-B.
plus(A,B,C):-  /*  FALSCH!!!  */
   B is C-A.
```

denn wenn A oder B ungebunden sind, dann antwortet das Prädikat in der ersten Klausel nicht nur No, sondern es terminiert gleich das gesamte Goal wegen einem Instanziierungsfehler[13]. Wir müssen also sicherstellen, dass A und B gebunden sind, bevor wir zur Zeile $C\ is\ A + B$ kommen:

```
plus(A,B,C):- /* Fall in dem A und B gebunden sind */
   nonvar( A ),
   nonvar( B ),
   C is A+B.
plus(A,B,C):- /* Faelle in denen C gebunden ist */
   nonvar( C ),
```

[13]Sollte ich ein Kapitel über Exceptions schreiben, werde ich auch zeigen, wie man dieses Verhalten abstellen kann

```
   nonvar( B ),
   var( A ),
   A is C-B.
plus(A,B,C):-
   nonvar( C ),
   nonvar( A ),
   var( B ),
   B is C-A.
```

In der zweiten und dritten Klausel wird noch vorausgesetzt, dass der dritte Parameter frei ist, damit wird lediglich verhindert, dass der Fall, in dem **alle** Variablen gebunden sind, mehrfach behandelt wird. Wenn mehr als ein Parameter ungebunden ist, antwortet dieses Prädikat einfach *No*.

8.5 Typsicherheit

Wir sind bei allen Prädikaten davon ausgegangen, dass die Parameter, die der User übergibt, den richtigen Typ haben - diese Annahme ist natürlich für die Praxis ungeeignet, denn wenn man Programmiert (egal, ob logisch, imperativ oder funktional) sollte man grundsätzlich davon ausgehen, dass die User entweder Idioten sind - oder vorsätzlich Schwachstellen des Systems suchen (diese Annahme ist meistens sehr zutreffend...).

Man kann den Typen einer Variablen innerhalb einer Klausel prüfen - hierfür gibt es die Systemprädikate *integer*, *atom*, $float$, *atomic* - wobei *atomic* Integers, Atome und Floats akzeptiert.

Man könnte mit diesen Systemprädikaten natürlich am Anfang jeder Klausel prüfen, ob alle Parameter den richtigen Typ haben, aber da Variablen von diesen Prädikaten nicht akzeptiert werden, muss man für jeden Parameter prüfen, ob es eine Variable ist oder sie den richtigen Typ hat - wenn man das in jeder Klausel machen würde, dann würden die Programme furchtbar lang - statt dessen schreibt man nur **eine** zusätzliche Klausel, in der man Prüft, ob ein Parameter ungültig ist - wenn diese „Prüfung“ erfolgreich ist (d.h. wenn eine Variable ungültig ist) dann erreicht man einen Cut, gefolgt von einem $fail$ (denn in dem Fall soll das Prädikat ja eben nicht akzeptieren) - wenn die Prüfung fehlschlägt (d.h. alle Parameter haben richtigen Typ) dann wird der Cut nicht erreicht, sondern die Klausel als nicht erfüllt beendet - so entstehen keine zusätzlichen Lösungen.

```
plus(A,B,C):-
   (
      not(integer(A);var(A)) ;
      not(integer(B);var(B)) ;
      not(integer(C);var(C))
   ),
   !,
   writeln('plus/3: ein Parameter hat einen falschen Typ'),
   fail.
plus(A,B,C):-
   nonvar(A),
```

```
      nonvar(B),
      C is A+B.
...
```

die wenigen Systemprädikate *integer*, *atom*, *float* und *atomic* reichen aus, um den Typ jeder Variablen zu Prüfen, denn man kann sie zusammensetzen, um eigene Typ-Prüfungs Prädikate zu entwickeln - die folgenden Prädikate akzeptieren Listen, Integer-Listen (ich habbe in dem Beispiel Variablen erlaubt) oder Summen (die Summe wird nicht ausgerechnet, sondern es wird überprüft, ob das Argument ein Ausdruck ist, der nur Additionen von Zahlen enthält):

```
liste([]).
liste([_|Rest]):-
   liste(Rest).

integerListe([]).
integerListe([Anfang|Rest]):-
   (
      integer(Anfang);
      var(Anfang)
   ) ,
   integerListe(Rest).

summe(A):-
   integer(A);
   float(A),!.
summe(A+R):-
   (
      integer(A);
      float(A)
   ),
   summe(R).
```

8.6 Stand-Alone Programme (Executables) erstellen

Will man sein Programm nicht nur im Interpreter ausführen, sondern ein eigenständiges Programm erstellen, dann kann man SWI Prolog statt mit dem Parameter $-f$ *dateiname* mit den Parametern $--$stand_alone=true $-$c *dateiname* starten ($-c$ und der Dateiname müssen die letzten Parameter sein!). Der Zusätzliche Parameter $-$o *dateiname* lässt Prolog das fertige Programm unter dem angebenen Dateinamen speichern. Lässt man diesen Parameter weg, speichert SWI Prolog das Programm (zumindest unter Linux - unter Windows habe ich es nicht getestet) als *a.out* ab

Wenn man einen Quellcode auf diese Weise kompiliert, erhält man ein Programm, dass SWI Prolog sehr ähnlich sieht - es schreibt die gleiche Nachricht beim Start (Welcome to SWI Prolog [...]) und dann eine Eingabezeile, die mit ?$-$ beginnt. Will man lieber, dass das Programm direkt ein bestimmtes Prädikat startet, dann kann man beim Kompilieren den Parameter $--$goal=*praedikatname* übergeben, wobei *praedikatname* der Name des Prädikats ist, das automatisch gestartet werden soll - es kann auch ein komplexeres Goal sein, es sollte aber grundsätzlich in Anführungszeichen gesetzt

werden, damit Klammern u.Ä. nicht als Teil von verschachtelten Shell-Befehlen interpretiert werden. in der Praxis wird meist ein Prädikat namens *main* geschrieben, welches ein Haupt-Prädikat mit Parametern aufruft, die Rückgaben ausgibt und dann fail-getrieben fortsetzt.

Ein weiterer Parameter −O (Buchstabe, nicht Zahl) lässt SWI Prolog eine Code-Optimierung durchführen. die Code Optimierung fügt z.B. an einigen Stellen (im Programm, nicht im Quellcode!) Cuts ein. Diese Optimierung ist auf modernen Computern zwar sehr schnell, aber während der Entwicklung überflüssig - sie ist erst beim Kompilieren des endgültigen Programms sinnvoll.

Damit das erstellte Programm nach der vollständigen Abarbeitung beendet wird (und nicht wieder eine Eingabezeile anzeigt) kann man z.B. ein *halt* als letzte Bedingung des Goals benutzen *–goal=“ (praedikatname); halt.“*
In der Praxis benutzt man aber einfach eine zweite Klausel für das *main* Prädikat, welches nur die Bedingung *halt* hat. Bei beiden Möglichkeiten muss man aber sicherstellen, dass die anderen *main* Klauseln nicht akzeptieren, da sonst der Benutzer *n*,*r* oder ; drücken muss, um weiter bis zum halt zu kommen
Beispiel:

```
main:-
   foo(A,B),
   write('A='), write(A),
   write(', B='),write(B),
   nl,
   fail,
main:-
   halt.
```

Aufruf aus der Shell:

pl −O −o foo −−stand_alone=true −−goal=main −c foo.pl

8.7 Starten eigener Programme unter Linux

Als ich zum ersten mal unter Linux (im CIP Pool der RWTH Aachen) ein Programm geschrieben habe, habe ich es lange nicht geschafft, es zu starten - falls ein/e Leser/in mit seinen/ihren selbst geschriebenen Programmen vor dem gleichen Problem steht: Wenn man unter Linux in einer Shell in einem Ordner ist, in dem das auszuführende Programm liegt (Ich nenne es mal „foo“), kann man es nicht durch die Eingabe „foo“ starten (wie es z.B. unter DOS geht). Der Grund dafür ist, dass Linux eine Liste von Ordnern hat, die $PATH heisst - das sind die Ordner, in denen nach dem Programm gesucht wird, das der Benutzer eingegeben hat (der Befehl „echo $PATH“ gibt diese Liste übrigens aus) Will man ein Programm starten, das in einem anderen Ordner liegt, dann muss man den vollständigen Pfad angeben, aber es existiert auch eine Symbolische Verknüpfung zum aktuellen Ordner namens . (Punkt) - man kann das Programm foo also

nicht durch foo, sondern durch $./foo$ starten.

Man kann zwar . in die $PATH Liste eintragen, aber das ist ein Sicherheitsrisiko, weswegen Linux-Experten davon abraten.

8.8 Kommandozeilen Parameter auslesen

Häufig will man beim Start eines Programms Parameter übergeben (das macht z.B. auch das Betriebssystem automatisch: wenn man auf eine Datei doppelklickt, dann wird das Programm, welches dem Dateityp zugeordnet ist, gestartet und bekommt als Parameter den Pfad der doppelt angeklickten Datei übergeben.
In Prolog kann man diese Parameter z.B. in die Liste *Argumente* schreiben lassen, indem man das Systemprädikat

```
unix(argv(Argumente))
```

benutzt. **WICHTIG: das funktioniert auch unter Windows!** der Prädikatname *unix* klingt zwar danach, dass es nur in Unix-artigen Systemen funktionieren würde, aber das stimmt nicht!

Die Variable *Argumente* enthält danach eine Liste aller übergebenen Parameter. Diese Liste enthält leider auch die Parameter, die gar nicht für das Programm, sondern für das Prolog System bestimmt waren (wie z.B. -f dateiname, -G32m etc). Dieses Problem behebt man klassischerweise, indem man zuerst die Prolog Argumente an das System übergibt, dann ein $--$ gefolgt von den Argumenten für das Programm. Die Liste der Argumente kann man dann problemlos mit dem *append* Prädikat zerlegen

```
append(PrologArgumente,[--|ProgrammArgumente],Argumente)
```

und hat nun eine Liste von Programm-Argumenten.
Wenn man in diese Liste ein mehrteiliges Argument suchen will (z.B. ein *-f* gefolgt von einem Dateinamen, der in eine Variable *D* kopiert werden soll), kann man wiederum das *append* Prädikat zweckentfremden:

```
append(_,['-f',D|_ ],ProgrammArgumente)
```

8.9 Mit größerem Stack arbeiten

Wenn ein Programm abstürzt, obwohl man sicher ist, dass es korrekt ist (auch wenn man sich damit meistens irrt), kann es daran liegen, dass der Stack, den Prolog dem Programm zur Verfügung stellt (Standardmäßig 2 Megabyte), zu klein ist - z.B. kann es dann bestimmte Rekursionstiefen nicht erreichen, die es evtl erreichen können muss. Damit Prolog dem Programm einen größeren Stack zur Verfügung stellt, kann man Prolog mit dem Parameter

$$-Gsize$$

starten z.B. $pl\ -G32m$ bitte beachten: zwischen dem G und der Größe ist kein Leerzeichen! Die Größe wird in Kilobyte angegeben, es sei denn man hängt ein m dahinter, dann werden Megabyte benutzt.
Übergibt man

$$-G0$$

dann beschränkt Prolog den Stack gar nicht - dann passiert ein Stack Overflow erst, wenn der RAM und der Swap Speicher auf der Festplatte voll sind.

8.10 Dokumentationen

Wenn man Anleitungen für Prolog oder Dokumentationen liest, begegnen einem oft zwei Formulierungen, zum einen

$$praedikatname/zahl$$

z.B.

$$vater/2$$

die Zahl hinter dem / gibt einfach die Anzahl der Parameter an, die das angegebene Prädikat hat (alle Parameter - auch die, die ungebunden sein sollen). Solche Angaben findet man oft in kurzen Beschreibungen eines Prädikats - z.B. wenn man mittels der Anfrage $help(praedikatname)$. die Dokumentation anzeigen lässt, die bei SWI Prolog mit installiert wird (GNU Prolog hat das *help* Prädikat nicht)

Die andere Formulierung, die einem oft begegnet ist

$$praedikatname(+Parametertyp, ?Parametertyp, -Parametertyp)$$

z.B.

$$nullstellen(+Integer, +Integer, -Integer)$$

Das bedeutet, dass für die Parameter, vor denen ein + steht, nur gebundene Variablen übergeben werden DÜRFEN, es sind input Variablen - sie werden z.B. auf rechten Seiten von Zuweisungen benutzt, übergibt man hier ungebundene Variablen, dann wird das Prädikat entweder abstürzen oder No Antworten (jenachdem, wie es programmiert ist). Für die Parameter, vor denen ein − steht **sollten** ungebundene Variablen übergeben werden, es sind output Variablen. allerdings kann man hierfür gebundene Variablen oder Konstanten übergeben, um das Ergebnis direkt zu vergleichen (siehe 2.9.4). Z.B. der Aufruf $nullstellen(0, -1, 1)$. liefert true, weil 1 eine Nullstelle von $x^2 + 0x - 1$ ist. Variablen, vor denen ein ? steht können sowohl eingaben, als auch ausgaben sein - jenachdem, welche der anderen Parameter gebunden sind.
Die Parametertypen geben an, welcher Datentyp dort erwartet wird. Integer kann zB heissen, dass der Parameter auf der rechten Seite eines *is* vorkommt (würde man hier also z.B. einen String wie 'hallo' übergeben, würde Prolog die Berechnung des Goals abbrechen mit der Meldung „ERROR: is/2: Arithmetic: 'hallo/0' is not a function")

Es ist zu beachten, dass diese Formulierungen nur der Beschreibung dienen, man kann Prolog so **NICHT** dazu veranlassen, zu prüfen ob Variablen gebunden oder ungebunden sind und ob sie Typkorrekt sind! Um dies sicherzustellen muss man die (im Vergleich dazu) unbequemen Methoden aus Kapitel 8.4 und 8.5 benutzen. Obwohl Borland Turbo Prolog 2 (neuere Versionen kenne ich nicht) es ermöglicht, auf ähnliche Weise (in einem Block des Programms, in dem die Prädikate deklariert werden) Typsicherheit zu erreichen, rate ich von diesem Interpreter sehr ab, da er gegen viele Prolog-Standards verstößt.

8.11 Fehlersuche: Die Berechnung „tracen"

Will man wissen, welche Berechnungsschritte Prolog geht, kann man einen Aufruf *tracen*, was man mit *Debugging* in imperativen Sprachen vergleichen kann.

Wollen wir nun also das Goal $vorfahre(X, hermann)$. tracen: hierzu schreiben wir vor das Goal *trace*, damit Prolog in den trace-Modus geht. nun gibt Prolog jede intern aufgerufene Anfrage an andere Prädikate aus und macht Pause, bis wir durch die *Enter*- oder *Space*-Taste den Befehl[14] zum Fortsetzen geben.
Findet Prolog eine Lösung, müssen wir wie vorher *n*,*r* oder *;* drücken, damit Prolog weiter sucht - hier *Enter* oder *Space* zu drücken würde bedeuten, dass uns die gefundene Lösung genügt und Prolog die Berechnung nun abbrechen kann - (ja, bei größeren Berechnungen kann dieses Verhalten sehr nervtötend sein, weil man nicht einfach die *Enter* Taste gedrückt halten kann, denn damit riskiert man, die Berechnung abzubrechen, sobald ein Ergebnis gefunden wurde). Mit der Anfrage *notrace*. schaltet man den trace-Modus wieder aus.
Hier nun die Ausgabe beim tracen der Anfrage $trace, vorfahre(X, hermann)$.

```
?- trace,vorfahre(X,hermann).
   Call: (8) vorfahre(_G187, hermann) ? creep
   Exit: (8) vorfahre(hermann, hermann) ? creep

X = hermann ;
   Redo: (8) vorfahre(_G187, hermann) ? creep
   Call: (9) elternteil(_L201, hermann) ? creep
   Call: (10) mutter(_L201, hermann) ? creep
   Exit: (10) mutter(irmtraut, hermann) ? creep
   Exit: (9) elternteil(irmtraut, hermann) ? creep
   Call: (9) vorfahre(_G187, irmtraut) ? creep
   Exit: (9) vorfahre(irmtraut, irmtraut) ? creep
   Exit: (8) vorfahre(irmtraut, hermann) ? creep

X = irmtraut ;
   Redo: (9) vorfahre(_G187, irmtraut) ? creep
   Call: (10) elternteil(_L236, irmtraut) ? creep
   Call: (11) mutter(_L236, irmtraut) ? creep
   Fail: (11) mutter(_L236, irmtraut) ? creep
   Call: (11) vater(_L236, irmtraut) ? creep
   Fail: (11) vater(_L236, irmtraut) ? creep
   Fail: (10) elternteil(_L236, irmtraut) ? creep
   Call: (10) vater(_L201, hermann) ? creep
   Exit: (10) vater(wilhelm, hermann) ? creep
   Exit: (9) elternteil(wilhelm, hermann) ? creep
   Call: (9) vorfahre(_G187, wilhelm) ? creep
   Exit: (9) vorfahre(wilhelm, wilhelm) ? creep
   Exit: (8) vorfahre(wilhelm, hermann) ? creep

X = wilhelm ;
   Redo: (9) vorfahre(_G187, wilhelm) ? creep
   Call: (10) elternteil(_L212, wilhelm) ? creep
   Call: (11) mutter(_L212, wilhelm) ? creep
   Fail: (11) mutter(_L212, wilhelm) ? creep
   Call: (11) vater(_L212, wilhelm) ? creep
```

[14] Der Befehl heißt „creep“, das ist englisch für „kriechen“. Prolog kriecht dann also weiter zur nächsten internen Anfrage

```
   Exit: (11) vater(hans, wilhelm) ? creep
   Exit: (10) elternteil(hans, wilhelm) ? creep
   Call: (10) vorfahre(_G187, hans) ? creep
   Exit: (10) vorfahre(hans, hans) ? creep
   Exit: (9) vorfahre(hans, wilhelm) ? creep
   Exit: (8) vorfahre(hans, hermann) ? creep

X = hans ;
   Redo: (10) vorfahre(_G187, hans) ? creep
   Call: (11) elternteil(_L257, hans) ? creep
   Call: (12) mutter(_L257, hans) ? creep
   Fail: (12) mutter(_L257, hans) ? creep
   Call: (12) vater(_L257, hans) ? creep
   Fail: (12) vater(_L257, hans) ? creep
   Fail: (11) elternteil(_L257, hans) ? creep
   Redo: (11) vater(_L212, wilhelm) ? creep
   Fail: (11) vater(_L212, wilhelm) ? creep
   Fail: (10) elternteil(_L212, wilhelm) ? creep

No
```

Diese Ausgabe ist folgendermaßen zu verstehen: *Call* bedeutet, dass ein Prädikat aufgerufen wird (Prolog geht eine Ebene tiefer im SLD Baum). *Fail* bedeutet, dass ein Prädikat ohne Lösung beendet wird z.B. auch nachdem die letzte Lösung für das Prädikat gefunden wurde und nur keine **weitere** Lösung gefunden wurde (im SLD Baum wird zurück zur letzten Verzweigung, wo noch choice-points übrig sind, gesprungen). *Exit* bedeutet, dass eine Lösung gefunden wurde, mit der nun in einer höheren Rekursionsstufe weiter gearbeitet werden soll (Prolog geht im SLD Baum bleibt es stehen). *Redo* bedeutet, dass zur letzten Situation zurück gesprungen wird, in der eine Lösung gefunden wurde (so wird die Verzweigung realisiert). Die Zahl in den Klammern ist die Rekursionstiefe, auf der Prolog sich befindet, allerdings um 7 zu hoch (das hängt mit System-internen Prädikat-Aufrufen vor der Verarbeitung des Goals zusammen). Danach folgt der Name des Aufrufs, auf den die Zeile sich bezieht - nur dass Variablen durch eine interne Speicheradresse ersetzt wurden.
creep ist die Ausgabe des Befehls (Berechnung fortsetzen), den ich bei der Erstellung dieses Testlaufs jeweils gegeben habe.

Hinweis wenn man statt *enter* oder *space* das *h* drückt, gibt Prolog eine Liste der möglichen Befehle aus, die man mit verschiedenen Tasten geben kann - z.B. die Auswertung abbrechen mit *a*, Prolog beenden mit *e*, Details über die letzte Exception anzeigen mit *m*, Schrittweise weitergehen mit *c*, *Enter* oder *Space*, die folgende Bedingung ohne unterbrechung Prüfen mit *l* (sehr nützlich!) und den *trace*-Modus beenden mit *n*.

8.12 Fehlersuche in der Praxis

Wenn man in der Praxis Fehler sucht, dann wird man nicht das komplette Programm tracen, weil das unter Umständen SEHR lange dauern kann. Mittels *write* Meldungen auszugeben, wie es in imperativen Programmiersprachen üblich ist, kann durch das interne Backtracking, das Prolog durchführt, sehr

unverständliche Ausgaben produzieren. Eine weitere Methode, wie man Fehler suchen kann, ist, nur einzelne Prädikate zu tracen - hierzu setzt man sogenannte ***Spy Punkte***, indem man vor der Anfrage $spy(praedikat/stelligkeit)$ als Bedingung benutzt. Prolog geht dann in den *trace* Modus, sobald das Prädikat ausgewertet wird - das heisst auch, dass ab diesem Zeitpunkt auch die Schritte in höheren Rekursionsstufen ausgegeben werden.
Weiterhin kann man mit der Taste h anzeigen lassen, welche Befehle man geben kann - z.B. die Auswertung abbrechen mit a, Prolog beenden mit e, Details über die letzte Exception anzeigen mit m, Schrittweise weitergehen mit c, *Enter* oder *Space*, die folgende Bedingung ohne unterbrechung Prüfen mit l (sehr nützlich!) und den *trace*-Modus beenden mit n.

9 Anhang

9.1 Musterlösungen

9.1.1 Kapitel 2

```
?- vater(Opa,E) , (mutter(E,Enkel);vater(E,Enkel)).
?- (vater(G,E);mutter(G,E)) , (vater(E,Enkel);mutter(E,Enkel)).
```

oder

```
?- elternteil(G,E), elternteil(E,Enkel).
```

```
halbgeschwister(A,B):-
   (
      mutter(M,A),
      mutter(M,B)
   );
   (
      vater(V,A),
      vater(V,B)
   ).

halbgeschwister2(A,B):-
   (
      mutter(M,A),
      mutter(M,B)
   );
   (
      vater(V,A),
      vater(V,B)
   ),
   \+(  geschwister(A,B)  ). /* nicht beide Elternteile stimmen überein */


halbgeschwister3(A,B):-
   elternteil(E,A),
   elternteil(E,B),
   \+(   geschwister(A,B)   ).
```

C + D = 5 + 4 * 3 ⇒ C=5, D=4*3
C * D = 5 + 4 * 3 schlägt fehl wegen der Regel „Punktrechnung vor Strichrechnung“: 5+4*3=5+(4*3) müsste entweder C=5+(4 und D=3) gelten (was keine gültige Bindung ist) oder C=4, D=3 - dann bleibt links aber ein 5+ übrig.

```
potenz(A,0,1).
potenz(A,B,P):-
   B > 0,
   B1 is B-1,
   potenz(A,B1,P1),
```

```
   P is P1*A.

multiplikation(A,0,0).
multiplikation(A,B,M):-
   B > 0,
   B1 is B-1,
   multiplikation(A,B1,M1),
   M is M1+A.

addition(A,0,A).
addition(A,B,S):-
   B > 0,
   B1 is B-1,
   addition(A,B1,S1),
   S is S1+1.
```

Die Türme von Hanoi mit n Scheiben ist lösbar, indem man $n-1$ Scheiben von A nach C bewegt (Rekursion), dann eine Scheibe von A nach B bewegt und danach die $n-1$ Scheiben von C nach B bewegt (nochmal Rekursion)

```
hanoi(0,_,_,_).
hanoi(N,Von,Nach,Ueber):-
   N > 0,
   N1 is N-1,
   hanoi(N1,Von,Ueber,Nach),
   write('Verlege Scheibe von Stab Nr '), write(Von),
   write(' nach Stab Nr '), write(Nach), nl,
   hanoi(N1,Ueber,Nach,Von).
```

Induktions Anfang: Für $n = 1$ sind n Scheiben offensichtlich verlegbar.
Induktions Voraussetzung: Für $i = 1...(n-1)$ seien i Scheiben verlegbar.
Induktions Schritt: Verlege $n-1$ Scheiben zum $Ueber$-Stab (möglich nach Induktions Voraussetzung), Verlege eine Scheibe zum $Nach$-Stab, Verlege $n-1$ Scheiben vom $Ueber$-Stab zum $Nach$ Stab (möglich nach Induktions Voraussetzung). Damit sind auch n Scheiben verlegbar.
Induktions Schluss: Nach dem Prinzip der vollständigen Induktion ist das Problem der Türme von Hanoi für alle $n \in \mathbb{N}$ lösbar.
Verlegungen(3) = 7
Verlegungen(4) = 15
Verlegungen(5) = 31
Verlegungen(n) = Verlegungen(n-1) + 1 + Verlegungen(n-1) = $2^n - 1$ (leicht induktiv beweisbar)

```
fiboEndrek(1,A,_,A).
fiboEndrek(N,A,B,Erg):-
   N1 is N-1,
   AB is A+B,
   fiboEndrek(N1,B,AB,Erg).
```

Gegeben war der Quellcode

```
praedikat(A,B,C,D):-
   a(A,B),
   b(B,C),
   !,
   c(A,B),
   d(C,D).
praedikat(A,A,A,A).


a(1,1).
a(1,2).
b(1,3).
b(2,3).
c(1,1).
d(3,23).
d(3,42).
```

durch den Cut wird festgelegt, dass A=1, B=1 und C=3 gelten wird. Die Frage ist, ob D nun eine oder mehrere Lösungen annimmt.
Testlauf:

```
?- praedikat(A,B,C,D).

A = 1
B = 1
C = 3
D = 23 ;

A = 1
B = 1
C = 3
D = 42 ;

No
```

Also verzweigt Prolog bei $d(C, D)$ wieder trotz des Cuts.

9.1.2 Kapitel 4

Die angegebene Lösung ist schlecht, weil das „N1 is N-1“ nur funktioniert, wenn N gebunden ist. Das heisst, dass man eine Länge übergeben muss und Prolog mit diesem Prädikat nur testen kann, ob die Liste tatsächlich die angegebene Länge hat, kann sie aber nicht selbst berechnen. Besser ist die folgende Lösung:

```
length([],0).
length([_|Rest],N):-
   length(Rest,N1),
   N is N1+1.

summe([],0).
```

```
summe([Anfang|Rest], Summe):-
   summe(Rest, RestSumme),
   Summe is Anfang + RestSumme.
```

delete$(9, K, Erg, [1,2,3,4])$. liefert die Lösungen

- K=0, Erg=[**9**,1,2,3,4]
- K=1, Erg=[1,**9**,2,3,4]
- K=2, Erg=[1,2,**9**,3,4]
- K=3, Erg=[1,2,3,**9**,4]
- K=4, Erg=[1,2,3,4,**9**]

also wenn man das *delete* Prädikat zum Einfügen verwendet und die Position, an der das Element eingefügt werden soll, nicht vorgibt, dann verzweigt Prolog und fügt das Element überall mal ein!

```
reverse([],[]).
reverse([Anfang|Rest], Ergebnis):-
   reverse(Rest,RevRest),
   append(RevRest,[Anfang],Ergebnis).

reverse_endrek([], Stack, Stack).
reverse_endrek([X|Rest], Stack, Erg):-
   reverse_endrek(Rest, [X|Stack], Erg).

palindrom(X):-
   reverse(X,X).

concat( [], [] ).
concat( [X|Rest], Erg ):-
   concat(Rest, Temp),
   append(X, Temp, Erg).


concat2( [], [] ).
/* Vorderste Liste ist leer */
concat2( [[]|RestlicheListen], Erg):-
   concat2(RestlicheListen,Erg).
/* Vorderstes Element der vordersten Liste nehmen */
concat2( [[X|RestVordersteListe]|RestlicheListen], [X|Erg] ):-
   concat2([RestVordersteListe|RestlicheListen], Erg).
```

Das Rätsel lautete

```
ABB -  CD = EED
 -     -     *
 FD +  EF =  CE
 =     =     =
EGD *  FH = ???
```

Es ist lösbar mit diesem Code:

```
raetsel:-
   permutationen([0,1,2,3,4,5,6,7,8,9],[A,B,C,D,E,F,G,H|_]),
   listenzahl([A,B,B],_,ABB),
   listenzahl([C,D],_,CD),
   listenzahl([E,E,D],_,EED),
   listenzahl([F,D],_,FD),
   listenzahl([E,F],_,EF),
   listenzahl([C,E],_,CE),
   listenzahl([E,G,D],_,EGD),
   listenzahl([F,H],_,FH),

   ABB - CD =:= EED,
   FD + EF =:= CE,
   ABB - FD =:= EGD,
   CD - EF =:= FH,

   Erg is EED * CE,
   Erg is EGD * FH,

   write(Erg), nl,
   fail.
```

Die einzige Lösung lautet 9315 (A=2, B=0, C=8, D=5, E=1, F=6, G=3, H=9)

```
   sendmoremoney_quick:-
      Ziffern = [0,1,2,3,4,5,6,7,8,9],
      delete(_,D,Ziffern,DZiffern),
      delete(_,E,DZiffern,EZiffern),
      delete(_,Y,EZiffern,YZiffern),
      Y is (D+E) mod 10,
      Uebertrag1 is (D+E) // 10,

      delete(_,N,YZiffern,NZiffern),
      delete(_,R,NZiffern,RZiffern),
      E is (N+R+Uebertrag1) mod 10,
      Uebertrag2 is (N+R+Uebertrag1)//10,

      % E ist bereits gebunden
      delete(_,O,RZiffern,OZiffern),
      N is (E+O+Uebertrag2) mod 10,
      Uebertrag3 is (E+O+Uebertrag2)//10,

      delete(_,S,OZiffern,SZiffern),
      delete(_,M,SZiffern,_),
      O is (S+M+Uebertrag3) mod 10,
      M is (S+M+Uebertrag3)//10,
```

```
      write(S),write(E),write(N),write(D),write('+'),
      write(M),write(O),write(R),write(E),write('='),
      write(M),write(O),write(N),write(E),write(Y),nl,fail.
   sendmoremoney_quick.
```

Das schnellere Programm braucht kaum messbare Zeit, das langsamere Programm braucht etwa 20 Sekunden (auf einem P4 mit 2,4 GHz) aber zur Entwicklung des schnelleren Programms braucht man deutlich mehr als 20 Sekunden länger als zur Entwicklung des langsameren Programms, daher hat sich der zusätzliche Aufwand nicht rentiert - wenn man das langsamere Programm schreibt, hat man die Lösung früher, als wenn man das schnellere Programm schreibt.

```
permutationen([0,1,2,3,4,5,6,7,8,9],[G,A,U,S,R,I,E,K,L,D]),
listenzahl([G,A,U,S,S],_,GAUSS),
listenzahl([R,I,E,S,E],_,RIESE),
listenzahl([E,U,K,L,I,D],_,EUKLID),
GAUSS+RIESE=:=EUKLID,
write(GAUSS),write('+'),write(RIESE),write('='),writeln(EUKLID),
fail.
```

Lösungen: 57088+46181=103269 und 47088+56181=103269

```
list_lowercase([],[]).
list_lowercase([X|Rest],[LowerX|LowerRest]):-
   (
      65 =< X, X =< 90, % X ist ein Großbuchstabe
      -> LowerX is X-65+97
      ;  LowerX = X
   ),
   list_lowercase(Rest,LowerRest).
lowercase(Str, LowerStr):-
   string_to_list(Str, List),
   list_lowercase(List, LowerList),
   string_to_list(LowerStr, LowerList).

list_uppercase([],[]).
list_uppercase([X|Rest],[UpperX|UpperRest]):-
   (
      97 =< X, X =< 122, % X ist ein Kleinbuchstabe
      -> UpperX is X+65-97
      ;  UpperX = X
   ),
   list_uppercase(Rest,UpperRest).
uppercase(Str, UpperStr):-
   string_to_list(Str, List),
   list_uppercase(List, UpperList),
   string_to_list(UpperStr, UpperList).
```

9.1.3 Kapitel 5

```
mergesort([],[]).
mergesort([X],[X]).
mergesort(Liste, SortierteListe):-
   length(Liste, Laenge),
   findall(X, Pos^(member(X,Pos,Liste),Pos<Laenge//2), Links),
   findall(X, Pos^(member(X,Pos,Liste),Pos>=Laenge//2), Rechts),
   mergesort(Links,LinksSortiert),
   mergesort(Rechts,RechtsSortiert),
   merge(LinksSortiert,RechtsSortiert,SortierteListe).

merge([], Liste, Liste).
merge(Liste, [], Liste).
merge([A|ARest], [B|BRest], [A|GesamtRest]):-
   A < B,
   merge(ARest, [B|BRest], GesamtRest).
merge([A|ARest], [B|BRest], [B|GesamtRest]):-
   A >= B,
   merge([A|ARest], BRest, GesamtRest).

krankheit(grippe, [husten, schnupfen, fieber]).
krankheit(schuettelfrost, [schnupfen, fieber]).
krankheit(magen_darm_entzuendung, [erbrechen, durchfall]).
...

diagnose(Symptome):-
   forall( krankheit(Name, KSymptome), (
      forall(member(S, Symptome), member(S,KSymptome))
      -> writeln(Name)
      ;  true
   )).

?- diagnose([fieber, husten]).
grippe
schuettelfrost
```

9.1.4 Kapitel 6

```
succ_mult(zero, _, zero).
succ_mult(succ(X),Y, Z):-
   succ_mult(X,Y,Z1),
   succ_add(Z1,Y,Z).

cons_delete( Objekt, 0, cons(Objekt, Rest), Rest ).
cons_delete( Objekt, Pos, cons(X, Rest), cons(X,DelRest)):-
   cons_delete( Objekt, PosInRest, Rest, DelRest),
   Pos is PosInRest +1.

cons_append( nil, X, X ).
cons_append( cons( Objekt, Rest), X, cons(Objekt, Y)):-
   cons_append(Rest, X, Y).
```

9.2 Literatur

Sehr gute Anleitung für Einsteiger (pdf):
http://wwwtcs.inf.tu-dresden.de/~nauber/arbeitsbuch_prolog.pdf
Offizizelle Referenz:
http://hcs.science.uva.nl/projects/SWI-Prolog/Manual/index.html
Offline version der offiziellen Referenz (pdf):
http://gollem.science.uva.nl/cgi-bin/nph-download/SWI-Prolog/refman/refman.pdf

Webseite des SWI Prolog Projekts:
http://www.swi-prolog.org/
Eine Sammlung von Tricks für Profis:
http://www.pms.ifi.lmu.de/publikationen/projektarbeiten/Martin.Kluger_Kalliopi.Sidiropoulou/00-inhalt.html
http://www.ling.uni-potsdam.de/kurse/Prolog/11_Cut.pdf
http://de.wikipedia.org/wiki/Prolog_(Programmiersprache)
http://de.wikibooks.org/wiki/Prolog
Wilhelm Weisweber: „Prolog logische Programmierung in der Praxis“ (ISBN 3-8266-0174-2)

9.3 Ausblick

In zukünftigen Versionen dieses Dokuments plane ich

- Kontextfreie Grammatiken
- Kuenstliche Intelligenz

zu behandeln. Je nach Feedback denke ich daran

- Exceptions
- Module
- Constraints
- Parsing

zu behandeln.

9.4 Danksagungen

Ich danke als ersten Walter Bender, Oscar Dustmann und Nadine Bergner für's Korrekturgelesen und Verbesserungsvorschläge.
Ausserdem danke ich Herrn Helmut Schumacher für seinen sehr guten Schulunterricht, in dem ich in kurzer Zeit viele Aspekte von Prolog kennengelernt und verstanden habe, sowie Prof. Gerhard Lakemeyer, an dessen Lehrstuhl ich intensiv praktisch mit Prolog arbeiten durfte, wodurch ich sehr viel dazu gelernt habe.

Index