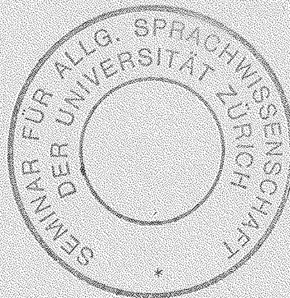
ARBEITEN DES SEMINARS FÜR ALLGEMEINE SPRACHWISSENSCHAFT DER UNIVERSITÄT ZÜRICH

NO. 4

PROGRAMMIEREN IN LOGIK
EINE ELEMENTARE EINFÜHRUNG IN DIE PROGRAMMIERSPRACHE PROLOG

Michael Hess



1986

Programmieren in Logik
Eine elementare Einführung in Prolog

Michael Hess
Seminar für allgemeine Sprachwissenschaft der Universität Zürich
April 1986

ZUSAMMENFASSUNG

Prolog ist eine auf der Logik basierte Programmiersprache, die in zunehmendem Mass für alle Probleme aus dem Bereich der Künstlichen Intelligenz verwendet wird, die aber ganz besonders geeignet ist für die Verarbeitung Natürlicher Sprache (wie z.B. Übersetzung von Texten durch Computer). Anhand einfacher Beispiele u.a. aus dem Bereich der Sprachverarbeitung werden die wichtigsten Eigenarten von Prolog erläutert und mit den Charakteristiken traditioneller Programmiersprachen verglichen.

Leicht überarbeitete Fassung einees in "Computer persönlich", Nr. 26/1985 bis Nr. 3/1986, Verlag Markt & Technik, erschienenen Artikels "Programmieren in Logik".

1. DER ÜBERGANG VOM "ZAHLENKNACKEN" ZUM "STRUKTURENKNACKEN" IN DER INFORMATIK

Computer sind als Werkzeuge zum Erledigen ganz bestimmter Arbeiten entstanden. So ist es auch kein Wunder, dass sowohl die Architektur der Computer wie auch die Struktur der Programmiersprachen ein direktes Resultat jener Aufgaben sind, zu deren Lösung man Computer überhaupt entwickelte. Diese Aufgaben hatten alle mit "Zahlenknacken" zu tun: Berechnung von aerodynamischen Korrekturen an der V1 (Zuses "Z 3"), Knacken von deutschen Codes (der "Colossus" von Bletchley Park), Auswertung von Volkszählungsdaten und Berechnung ballistischer Werte für die Raumfahrt und für strategische Raketen (in den USA). Deshalb sind auch die allermeisten Programmiersprachen noch heutzutage Zahlenknack-Sprachen.

In den letzten Jahren hat sich aber das Schwergewicht der Anwendungen langsam vom reinen Zahlenknacken wegzubewegen begonnen: Nur schon die Textverarbeitung erfordert eine erhebliche Anzahl von nicht-numerischen, strukturorientierten Operationen (z.B. "string-matching"), während Datenbankprogramme sogar vorwiegend mit strukturorientierten Operationen beschäftigt sind, ganz zu schweigen von jenen Interfaces für Datenbanken, welche Abfragen in (vereinfachter) menschlicher Sprache erlauben, wie man sie ja in einfacher Form auch schon kommerziell erhalten kann.

Insbesondere bei Forschungsarbeiten auf dem Gebiet der Künstlichen Intelligenz, aus denen z.B. die zuletzt erwähnten Programme zur Datenbankabfrage in natürlicher Sprache stammen, erwiesen sich die traditionellen zahlenorientierten Sprachen immer mehr als Hindernis. Das wird auch der Hobby-Programmierer sehr schnell merken, wenn er ein

einfaches Programm dieser Art z.B. in Pascal schreiben will: Er wird seine Tage mit dem Deklarieren von Arrays und dem Initialisieren von Varablen verbringen, aber eigentlich nie zu den wirklich zentralen und interessanten Fragen der Sprachanalyse vordringen. Schon in den Sechzigerjahren wurden aus diesen Gründen spezielle strukturorientierte Sprachen entwickelt, deren erfolgreichste LISP ist (was für "List Processing language" steht, ein Name, der die grundsätzlich auf Strukturen, eben Listen, ausgerichtete Konzeption illustriert). LISP ist durchaus auch für den Hobby-Programmierer zugänglich, sowohl was die Erhältlichkeit der Sprache für Mikrocomputer betrifft wie auch, was den Aufwand angeht, den man zum Erlernen der Sprache betreiben muss. Es ist aber auch nicht zu übersehen, dass die Sprache unübersichtlich (geworden) ist und zum "Hacken" statt zum Programmieren ermuntert. Es ist ebenso schwer, ein von einer anderen Person geschriebenes LISP-Programm lesend zu verstehen, wie ein von einem selbst geschriebenes Programm zu entwanzen - beides schwerwiegende Nachteile (die LISP natürlich mit den traditionellen Sprachen gemeinsam hat). LISP war der erste und bahnbrechende Schritt zu einem neuen Typ von Sprache, aber eben nur der erste. Der zweite Schritt wurde vor etwa zehn Jahren eingeleitet und ist nun soweit vollzogen, dass auch der Mikrocomputer-Benutzer ihn nachvollziehen kann. Der zweite Schritt weg von den zahlenknackenden Sprachen und hin zu den strukturknackenden ist eben Prolog. Wer immer sich den neuen und sehr interessanten Problemstellungen der Künstlichen Intelligenz zuwenden will, tut gut daran, Prolog als Programmiersprache in Betracht zu ziehen.

1.1 Imperative Sprachen und deklarative Sprachen

Der grösste Teil der Leistungsfähigkeit von Prolog geht auf die Tatsache zurück, dass es keine imperative Programmiersprache ist, sondern eine deklarative Sprache. Was soll das heissen?

In "traditionellen" Programmiersprachen muss man ein Programm so schreiben, dass man dem Computer im einzelnen befiehlt, auf welchem Weg er zur Lösung eines bestimmten Problems kommen muss: Einen Schritt nach vorn, wenndort ein Hindernis hat, eine Drehung nach links, eine nach rechts, sonst geradeaus, usw. Der Programmverfasser muss dementsprechend an jeder Stelle des vorausgeplanten Weges alle möglicherweise auftauchenden Hindernisse vorhersehen und entsprechend berücksichtigen. Es ist dabei aber buchstäblich unvermeidbar, dass er das eine oder andere Hindernis nicht vorhersieht, was dann zu Laufzeitfehlern führt. Dennoch scheint dieses Vorgehen derart normal und natürlich, dass man sich anfänglich gar nicht vorstellen kann, dass es auch anders ginge. Aber es geht anders: Man gibt dem Computer bloss noch Ausgangspunkt und Ziel an und versieht ihn mit Rezepten, wie er sich angesichts von Hindernissen, wo immer sie auch auftauchen mögen, verhalten kann. Wie im einzelnen der Computer dann ans Ziel kommt, ob er ein bestimmtes Hindernis rechts oder links umgeht, ist dem Programmverfasser egal. Er akzeptiert der Möglichkeit, dass der Computer ein Hindernis rechts umgeht, obwohl es links herum kürzer gewesen wäre: Er spart enorm Zeit beim Programmieren, und da ist er gerne bereit, dafür etwas mehr Zeit fürs Rechnen zu bezahlen.

Wie aus dem vorstehenden Vergleich klargeworden ist, besteht diese Art von Programm aus Angaben, auf welche Weise sich bestimmte Teilziele

erreichen lassen, eben z.B., wie man trotz einem Hindernis in der allgemeinen Richtung vorwärts kommen kann. Man muss also immer noch vorhersehen, was für Hindernisse da zu erwarten sind, und wie man jedes einzelne umgehen kann, aber man muss eben nicht mehr vorhersehen, wo und wann welche Art von Hindernis auftauchen könnte. Deshalb muss man diese allgemeinen Angaben über Teilproblem-Lösungen auch nicht in einer bestimmten Reihenfolge ins Programm einfügen - man kann sie im Prinzip in irgendeiner Reihenfolge, wie sie einem gerade in den Sinn kommen, hinschreiben: Sie lassen sich völlig unabhängig voneinander festlegen. Das Programm hat dann die Aufgabe, wann immer es auf ein Hindernis aufläuft, das Programm auf Regeln hin zu durchsuchen, welche für die Lösung dieses eben aufgetauchten Problems nützlich sein könnten. Es wendet die erste an, die es antrifft, und wenn die funktioniert, ist gut, sonst versucht es die zweite, dann die dritte usw. Da man deshalb im Programm selbst gar keinen festen Ablauf der Abarbeitung einplanen muss, sondern bloss sozusagen statisch festlegt, was der Computer unter welchen Umständen tun kann, spricht man von "deklarativem Programmieren" und setzt es vom "imperativen Programmieren" in traditionellen Sprachen ab.

1.2 Ein sehr einfaches Beispiel: Eine Datenbank mit elementaren Fakten

Wir wollen nunmehr versuchen, eine Variante des verwendeten einfachen Beispiels konkret in Prolog, der bekanntesten und weitestverbreiteten rein deklarativen Sprache, zu programmieren, um damit die soeben erwähnten Grundprinzipien dieser Programmiersprachen zu illustrieren. Da die Grundbegriffe von Prolog erfahrungsgemäss genau jenen Leuten am meisten Schwierigkeiten bereiten, die am meisten Erfahrung mit imperativen Sprachen haben, wird die folgende Schilderung sehr elementar sein. Wer findet, es werde zu viel zu oft wiederholt, kann einfach zu den späteren Abschnitten springen.

Wir wollen also ein Prolog-Programm schreiben, um von einem Punkt zu einem andern zu gelangen, wobei bestimmte Probleme auftauchen können, die gelöst werden müssen. Die Probleme sind, wie oft, mit Geld verbunden: Wir beginnen unsere Reise mit einem bestimmten Geldbetrag in der Tasche und müssen damit die erste Teilstrecke der Reise bewältigen. An verschiedenen Orten auf der Welt haben wir Bankkonten, wo wir uns neu mit Geld eindecken können. Da aber nicht alle Konten sehr gut dotiert sind, werden wir nicht immer die komfortabelste Reiseart wählen können.

Wir müssen natürlich zuallererst die Angaben über die möglichen Verbindungen, über die Art der Transportmittel und über die Kosten in der Datenbank festhalten. Die dazu verwendete Darstellungsweise mag auf den ersten Blick ein wenig befremden, aber bald wird man ihre Vorteile zu schätzen beginnen: Einheitlichkeit und Klarheit. Diese Darstellungsweise ist dieselbe, die man auch in der Logik verwendet: Ein Prädikat, das eine Eigenschaft oder eine Beziehung nennt, sowie mehrere Argumentstellen, in die man die konkreten Werte eintragen kann, zwischen denen die Beziehung besteht. (Aus der Tatsache, dass man in Prolog die Notation der Logik verwendet, erklärt sich auch der Name "Prolog" er steht für "Programmieren in Logik"). Die Tatsache, dass man von Zürich nach London mit der Air Lanka fliegen kann, und dass das 300 Fr. kostet, kann man in folgender Form ausdrücken:

```
verbindung(zürich,london,air_lanka,300).
```

Man könnte auch eine andere Reihenfolge der Argumente wählen, z.B.

```
verbindung(300,air_lanka,zürich,london)
```

(oder irgend eine andere der möglichen Kommutationen). Wesentlich ist nicht, wie die Reihenfolge festgelegt ist - wesentlich ist bloss, dass man sich an die Konvention hält, wenn man sie einmal festgelegt hat: Die <u>Position</u> eines Argumentwerts definiert seinen <u>"Typ"</u> (wie eben z.B. den Typ "Kosten"). Bemerkenswert ist, dass man diese Konvention aber nirgendwo im Programm niederschreiben (deklarieren) muss. Wir schreiben alle Ausdrücke mit kleinen Anfangsbuchstaben. Auf die Gründe dafür werden wir sehr bald zu sprechen kommen.

Wir können unsere elementare Datenbank für die Reiseplanung mit der Information über unsere zwei Konten und deren gegenwärtigen Stand vorläufig einmal folgendermassen darstellen:

```
konto(zürich,1800).
konto(hong_kong,150).
verbindung(zürich,london,air_lanka,300).
verbindung(zürich,london,bahn,250).
verbindung(zürich,hong_kong,swissair,3000).
verbindung(london,hong_kong,cathay,1200).
verbindung(hong_kong,beijing,bahn,150).
verbindung(hong_kong,beijing,caac,500).
```

Man könnte glauben, damit hätten wir erst die Daten festgelegt, über dem unser Datenbankprogramm, wenn wir es einmal geschrieben haben, operieren kann - aber in Tat und Wahrheit haben wir damit schon ein vollständiges Prolog-Datenbank-Programm geschrieben! Man braucht nämlich nur mehr den obenstehenden Programmtext einzulesen und kann loslegen mit dem Abfragen der Datenbank: kein einziges weiteres Wort muss dazu ins Programm eingefügt werden. Allerdings muss man sich beim Formulieren der Abfrage schon noch einige Gedanken machen.

1.3 Die einfachste Art von Datenbankabfrage

Wenn wir vorerst eine sehr einfache Auskunft haben wollen, z.B. was die verschiedenen Reisemöglichkeiten von Zürich nach London sind, so werden wir diese Abfrage formulieren als

```
verbindung(zürich,london,Verkehrsmittel,Kosten).
```

Die letzten zwei Argumentpositionen haben wir diesmal mit Ausdrücken belegt, die mit einem Grossbuchstaben beginnen. Das ist die Prolog-Konvention, Varablen darzustellen. Man kann beliebige Namen wählen, auch ganz unsinnige Ausdrücke wie ZtpTTbn oder NNN_oPP, sofern sie bloss mit einem Grossbuchstaben beginnen. Dass man vernünftigerweise sinnvolle Varablennamen wählt, liegt auf der Hand.

Wenn man nun also den obigen Programmtext, die Datenbankangaben, eingelesen hat und den Systemprompt von Prolog sieht (in der Regel

"?-"), tippt man die soeben formulierte Anfrage ein. Da Prolog (meist) als interpretierte Sprache eingesetzt vorliegt, wird die Anfrage sogleich beantwortet, wenn man nur noch auf die Return-Taste drückt. Was der Interpretierer mit der Anfrage tun muss, ist eigentlich schon vom Beispiel her klar: Er nimmt den Frage-Ausdruck, und vergleicht ihn mit den Ausdrücken im Programm, d.h. in der Datenbank. Er beginnt dabei zuoberst, und legt gleichsam die Anfrage auf einen Ausdruck des Programms nach dem andern und schaut, ob sie sich zur Deckung bringen lassen. Dass sich die Anfrage mit den Ausdrücken der zwei ersten Programm-Zeilen (den Angaben über unsere Konten) nicht zur Deckung bringen lassen, ist offensichtlich: Allein schon der Name der Beziehung, das Prädikat, ist ja verschieden. Schwieriger wird es bei den folgenden Ausdrücken des Programms: Hier stimmt zumindest das Prädikat der Frage mit den Prädikaten der Programmeinträge überein. In einem solchen Fall musse auch noch die Anzahl der Argumente übereinstimmen: "reise(A,B)" z.B. lässt sich nicht mit "reise(zürich,london,lufthansa,300)" zur Deckung bringen, weil die beiden Ausdrücke nicht gleich viele Argumente haben. Diese Bedingung ist aber beim betrachteten Beispiel erfüllt. Nun müssen aber auch noch die Werte der Argumente von Anfrage und Datenbankeintrag verglichen werden. Beim ersten Eintrag mit dem Prädikat "verbindung" stimmen nun tatsächlich die erten zwei Argumentwerte von Anfrage und Eintrag überein: in beiden Fällen sind sie "zürich" und "london". Was geschieht aber mit den zwei weiteren Argumenten, die wir ja in der Anfrage mit Variablen belegt hatten, wo aber Konstanten im Datenbankeintrag stehen? Hier nun nimmt einfach jede Variable einer Anfrage den Wert jener Konstante im Datenbankeintrag an, auf die sie zu liegen kommt; der Konstantenwert "scheint" einfach "durch die Variable durch". Die Variable "Verkehrsmittel" erhält dadurch den Wert "air_lanka", und die Variable "Kosten" den Wert "300". Der Prolog Interpreter ist nun so konzipiert, dass er beim ersten "Treffer" (d.h. bei der ersten Gelegenheit, wo er eine Anfrage mit einem Datenbankeintrag zur Deckung bringen kann), innehält, "yes" schreibt, und zusätzlich die Werte aller Variablenbindungen ausschreibt. Wenn wir also den ganzen Mini-Dialog aufzeichnen, sieht das so aus:

```
Computer:   "?-"
Benützer:   "verbindung(zürich,london,Verkehrsmittel,Kosten). <return>"
Computer:   "yes
             Verkehrsmittel=air_lanka
             Kosten=300
             ?-"
```

Wir erhalten die gewünschten Angaben, sowie die Bereitschaftsmeldung des Systems "?-", welche uns sagt, dass wir weitere Fragen eingeben können. Wenn uns die Antwort etwas zu lakonisch ist, können wir die Frage so stellen, dass eine etwas ausführlichere Antwort erzeugt wird. Wenn wir z.B. eintippen

```
"verbindung(zürich,london,Vm,K), write(Verkehrsmittel ist ,
 write(Vm), write( und die Kosten sind , write(K),
 write(Fr.. <return>"
```

werden wir zusätzlich zu der oben angeführten Antwort noch den Satz ausgeschrieben erhalten

"Das Verkehrsmittel ist air_lanka, und die Kosten sind 300 Fr."

Dieser ganze lange Ausdruck war eine Mehrfachfrage, bestehend aus sechs Teilfragen ("write(X)" ist für Prolog auch eine "Frage"). Man tippt einfach alle Teilfragen nacheinander ein, mit Kommas dazwischen, und drückt erst am Schluss auf die Return-Taste. Prolog geht dann daran, eine Teilfrage nach der andern zu beantworten. Die erste dieser Teilfragen ist offensichtlich dasselbe wie die ursprüngliche Frage, bloss mit abgekürzten Variablennamen (wie wir uns erinnern, kommt es auf die Form des Namens nicht an - allein ihre Position innerhalb eines Prädikats gibt einer Variablen ihre Bedeutung). Wenn Prolog diese Teilfrage beantwortet hat, sind die Variablen Vm und K an die Werte "air_lanka" und "300" gebunden, wie gehabt. Wichtig ist nun aber, dass sich der Wert einer Variablenbindung, sobald er im Lauf einer Beantwortung ermittelt worden ist, sogleich ausbreitet innerhalb der Klausel, in der die Variable steht. Eine "Klausel" ist alles, was vom Anfang eines Ausdrucks bis zum ersten Punkt steht, sei der Ausdruck nun eine Anfrage oder ein Datenbankeintrag. Was heisst es aber, dass der Wert einer Variable sich innerhalb einer Klausel ausbreitet? Sobald die Variable Vm in der ersten Teilfrage einen Wert zugewiesen erhält (also "air_lanka" in unserem Beispiel), sucht der Prolog-Interpreter, bevor er irgend etwas anderes tut, weitere Vorkommen dieser Variable in der Klausel. Und in der Tat findet er ein weiteres Vorkommen von "Vm", und zwar in der dritten Teilfrage: "write(Vm)". Hier wird nun auch "air_lanka" in die Lücke mit Namen "Vm" eingefüllt.

Sobald alle Vorkommen dieser bestimmten Variablen gefunden und aufgefüllt worden sind, macht sich der Prolog-Interpreter an die Beantwortung der zweiten Teilfrage, "write(Verkehrsmittel ist ".
Das Prädikat "write" ist nun aber ein besonderes Prädikat, ein sogenanntes Systemprädikat. Wenn der Interpreter einen Wert für ein Systemprädikat berechnen soll, versucht er gar nicht erst, in der Datenbank (dem Programm) einen entsprechenden Eintrag zu finden. Er hat statt dessen eine besondere Liste, in der verzeichnet ist, welche besondere Aktion für jedes Systemprädikat auszuführen ist. Das Systemprädikat "write(_)" schreibt den Wert seines (einzigen) Arguments (auf den Bildschirm oder in einen File - das kann man separat festlegen), und "read(_)" liest Dinge ein. Alle arithmetischen Operationen sind als Systemprädikate definiert (welche auch die übliche Infix-Schreibweise zulassen); z.B. wird das Teilziel "Sum is 5+3" so beantwortet, dass fünf und drei addiert werden und das Resultat in die Variable "Sum" gesteckt wird, ohne dass irgend etwas in der Datenbank nachgeschaut würde. Im Falle unserer Beispielabfrage würde nunmehr also einfach "Das Verkehrsmittel ist " auf den Bildschirm geschrieben.

Die Beantwortung der dritten Teilfrage ist ebenso einfach: In die Position von "Vm" im Ausdruck "write(Vm)" ist ja schon vorher der Wert "air_lanka" eingefügt worden, so dass der Interpreter jetzt bloss noch diesen Wert ausschreiben muss. Wesentlich ist hier nur, dass man sich daran erinnert, wie sich dieser Wert vorher "von links" her ausgebreitet hatte, vom ersten Vorkommen der Variable "Vm" her, wo eine Bindung ermittelt werden konnte.

Die verbleibenden Teilfragen werden in genau derselben Weise beantwortet.

1.4 Eine etwas kompliziertere Abfrage

Nun möchte man aber sicher auch andere, etwas schwierigere Fragen beantwortet haben. Statt nur direkter Verbindungen (wie eben zwischen Zürich und London) könnte man sich für Routen interessieren, welche man aus einzelnen Teilstrecken zusammensetzen kann, wie z.B. von von Zürich nach Beijing. Der erste Gedanke, wie man diese Frage wohl formulieren könnte, ist einfach (und falsch): "verbindung(zürich,beijing,Vm,K).". Wie man sogleich sieht, wird der Interpreter keinen Eintrag in der Datenbank finden und daher mit "no" antworten. Schliesslich haben wir im Programm ja auch nur direkte Verbindungen angegeben, da kann man nichts mehr erwarten. Es gibt aber eine Ausweg: Man muss die Anfrage umformulieren. Wenn wir fragen

"verbindung(zürich,A,Vm1,K1),
verbindung(A,B,Vm2,K2),verbindung(B,beijing,Vm3,K3)."

werden wir die Antwort bekommen

"yes
A=london
Vm1=air_lanka
K1=300
B=hong_kong
Vm2=cathay
K2=1200
Vm3=bahn
K3=150
?-"

Diese Antwort enthält sicher alles, was wir erwarteten (obwohl wir eine etwas benützerfreundlichere Darstellung wünschen möchten), und es ist auch leicht nachzuvollziehen, wie die Antwort zustandekam - genau gleich wie zuvor: Es wurde immer der erste Eintrag in der Datenbank, mit dem sich ein Anfrage-Prädikat zur Deckung bringen liess, benützt, um Werte für die Variablen zu finden. Diese Werte breiteten sich sodann in der ganzen Frageklausel aus. So fand die erste Teilfrage "verbindung(zürich,A,Vm1,K1)" als ersten Eintrag in der Datenbank, mit dem die Anfrage zur Deckung gebracht werden konnte, "verbindung(zürich,london,air_lanka,300)", und die Variable A bekam dadurch den Wert "london" zugewiesen. Dieser Wert breitete sich auch auf das "A" in der zweiten Teilfrage "verbindung(A,B,Vm2,K2)" aus. Da für die anderen Variablen in dieser Teilfrage aber noch keine Werte hatten ermittelt werden können, wurde die Frage in der nur teilweise "instantiierten" Form "verbindung(london,B,Vm2,K2)" auf die Datenbank losgelassen. Erneut wurde der erste passende Eintrag verwendet, nämlich "verbindung(london,hong_kong,cathay,1200)", und der Wert "hong_kong" für die Variable "B" breitete sich in analoger Weise auf den dritten Term aus.

Obwohl wir also eine Methode gefunden haben, wie man die gewünschte Antwort aus der Datenbank herausholen kann, bleibt ein sehr ungutes Gefühl: Man hat zwar ein extrem einfaches Programm, aber um eine auch nur einigermassen komplizierte Frage zu beantworten, musste man beim Formulieren der Frage soviel Vorkenntnisse über die Struktur der Datenbank besitzen, dass man sich die Antwort ebensoleicht hätte von Hand ausrechnen können. Man musste insbesondere wissen, dass man von

Zürich nach Beijing zweimal umsteigen muss, dass man also (genau) drei Teilfragen stellen muss. Aber genau diese Art von Information möchte man ja aus einer Datenbank herausbekommen, nicht in sie hineinstecken.

Wir möchten dem Programm eine Frage stellen können wie "reise(zürich,beijing,Vm,K)", und als Antwort die ganze Reiseroute inklusive Umsteigestationen (in Variable Vm) und die Kosten für die einzelnen Teilstrecken oder vielleicht auch den Gesamtpreis der Reise (in Variable K) erhalten. Dazu muss das Programm den Computer dazu bringen, alle jene Überlegungen anzustellen, die wir vorhin selbst beim Formulieren der Frage anstellen mussten. Wir wollen zuerst eine sehr einfache, eine allzu einfache, "Lösung" dieses Problems anschauen, um eine wesentliche Erweiterung der bisher vorgestellten Ausdrucksmöglichkeiten von Prolog zu illustrieren.

Man könnte nämlich davon ausgehen, dass in unserer Datenbank gar keine längeren Reisenrouten als die soeben dargestellte möglich sind, dass man also höchstens zweimal umsteigen muss. Das Programm könnte daher einfach zuerst schauen, ob eine gewünschte Verbindung direkt möglich ist, und wenn das nicht der Fall ist, prüfen, ob es mit einmaligem Umsteigen geht, und es sonst mit zweimaligem Umsteigen versuchen. Wenn auch das nicht zum Ziel führen sollte, dürfte das Programm guten Gewissens mit "no" antworten. Diese (wie erwähnt nicht eben elegante Strategie) lässt sich folgendermassen in Prolog ausdrücken:

```
reise(Start,Ziel,[Vm],[K])  :-
               verbindung(Start,Ziel,Vm,K).
reise(Start,Ziel,[Vm1,Vm2],[K1,K2])  :-
               verbindung(Start,Zwischenziel,Vm1,K1),
               verbindung(Zwischenziel,Ziel,Vm2,K2).
reise(Start,Ziel,[Vm1,Vm2,Vm3],[K1,K2,K3])  :-
               verbindung(Start,Zwischenziel1,Vm1,K1),
               verbindung(Zwischenziel1,Zwischenziel2,Vm2,K2),
               verbindung(Zwischenziel2,Ziel,Vm3,K3).
```

Dieses neue Programmstück fügt man irgendwo ins bestehende Programm ein. Nehmen wir einmal an, wir fügten es oben an. Betrachten wir nun diese Programmerweiterung etwas näher. Ein neues Zeichen ist eingeführt worden: ":-". Es trennt den "Kopf" einer sog. Regel vom "Körper". Der Kopf darf nur einen einzigen Term umfassen, aber der Körper darf aus mehreren, durch Kommas voneinander getrennten Termen bestehen. Der Sinn dieser Regeln ist der: Wenn der Interpreter mit einer Frage (eben z.B. "reise(zürich,beijing,Vm,K)") in die Datenbank hineingeht, versucht er nicht nur, "direkte Treffer" zu finden, wie wir das bisher immer beschrieben haben, sondern er schaut auch, ob er vielleicht eine Regel antrifft, deren Kopf mit dem Suchbergriff zur Deckung gebracht werden kann. Wenn das der Fall ist, wird aber nicht gleich ein Erfolg angenommen, sondern zuerst müssen noch alle Terme des Körpers dieser Regel abgearbeitet werden. Die "Abarbeitung" besteht darin, alle diese rechtsseitigen Terme als Unterziele zu behandeln, die man lösen muss. Das heisst, dass der Interpreter alle diese Terme als neue Abfragen behandelt, die er in genau der gleichen Weise zu beantworten sucht, in der er die vom Benützer gestellten Fragen behandelt.

Betrachten wir, wie sich das bei unserer Beispielabfrage "reise(zürich,beijing,Vm,K)" abspielt. Der Interpreter sucht einen

einfachen Term, auch als "Fakt" bezeichnet, oder aber den Kopf einer Regel, der sich mit dieser Abfrage zur Deckung bringen lässt. Schon die erste Regel im neueingefügten Programmstück,

```
"reise(Start,Ziel,Vm,K) :- verbindung(Start,Ziel,Vm,K).",
```

hat einen Kopf, der sich mit der Abfrage deckt. Da sich im Kopf der Regel die Variable "Start" mit der Konstante "zürich" deckt und die Variable "Ziel" mit "beijing", breiten sich diese Werte in den Körper der Regel aus; auch dort werden diese beiden Variablen nunmehr diese Konstanten als Wert haben. Der Interpreter muss nun aber noch alle rechtsseitigen Ausdrücke der Regel abarbeiten, was in diesem Fall besonders einfach ist, da es nur einen einzigen rechtsseitigen Ausdruck gibt:

```
"verbindung(Start,Ziel,Vm,K)",  respektive
"verbindung(zürich,beijing,Vm,K)",
```

wenn man die zurzeit schon bekannten Variablenwerte einsetzt. Diese variablengebundene Form ist es auch, die der Interpreter nunmehr als Abfrage benutzt, um erneut in die Datenbank hineinzugehen. Wiederum von oben ausgehend sucht er den ersten Treffer. Wir wissen aber aus dem allerersten Beispiel, dass es keine Antwort auf diese Abfrage gibt - die Datenbank kennt keine Direktverbindungen von Zürich nach Beijing. Im Unterschied zu vorhin muss der Interpreter nun aber noch nicht aufgeben. Er kann jetzt nämlich versuchen, weitere Fakten oder Regeln zu finden, die zur Abfrage passen. Zu diesem Zweck muss sich der Interpreter merken, welche Fakten und Regeln er schon mit einer bestimmten Abfrage ausprobiert hatte, aber der Benützer muss sich darum nicht kümmern - das geschieht alles vollautomatisch. In unserem Beispiel wird der Interpreter sofort einen weiteren auf den ersten Blick aussichtsreich aussehenden Datenbankeintrag finden, nämlich die zweite Regel,

```
"reise(Start,Ziel,[Vm1,Vm2],[K1,K2])  :-
              verbindung(Start,Zwischenziel,Vm1,K1),
              verbindung(Zwischenziel,Ziel,Vm2,K2)."
```

Wir sind versucht, schon jetzt zu sagen, dass auch die von dieser Regel erzeugte Abfrage (genauer: die zwei erzeugten Abfragen) nicht zum Erfolg führen werden. Wir wissen ja (glauben zu wissen), dass wir zweimal umsteigen müssen nach Beijing. Falsch: Das Programm findet eine unvermutete, bessere Lösung: Von Zürich direkt nach Hong Kong, und von dort weiter nach Beijing. Wie kam es auf diese Lösung? Sehr einfach: Der Interpreter merkt sich ja, wie erwähnt, welche Regeln (und Fakten) er mit welcher Abfrage schon ausprobiert hat. Wann immer er eine der Teilfragen einer Regel nicht beantworten kann, geht er zum letzten dieser "Entscheidungspunkte" zurück und schaut, ob er vielleicht eine andere passende Regel finden kann. Anhand unseres Beispiels: Als erste Antwort auf die erste Teilfrage wird der Interpreter, wie schon immer, den Eintrag "verbindung(zürich,london,air_lanka,300)" finden. Daraus ergibt sich der zweite (und letzte) Aufruf "verbindung(london,beijing,Vm2,K2)". Diese Frage kann aber nicht positiv beantwortet werden. Und jetzt geht der Interpreter zurück zur ersten Teilfrage "verbindung(Start,Zwischenziel,Vm1,K1)", wo erst der Wert der Variablen "Start" gebunden war (und zwar an die Konstante "zürich"), und schaut,

ob er auf diese Teilfrage eine weitere Antwort finden kann. Und ja, er kann: "verbindung(zürich,london,bahn,250)". Daraus ergibt sich ein neuer Versuch, die zweite Teilfrage zu beantworten, der Aufruf "verbindung(london,beijing,Vm2,K2)". Das ist zufälligerweise die selbe, erfolglose, Form der zweiten Teilfrage wie zuvor. Tut nichts, der Interpreter gibt immer noch nicht auf und geht wieder zurück zur gleichen Stelle und schaut, ob er nicht vielleicht eine dritte Möglichkeit hat, die erste Teilfrage zu beantworten. Ja, er hat: Das Faktum "verbindung(zürich,hong_kong,swissair,3000)" ist eine dritte mögliche Antwort auf Teilfrage Nummer eins. Daraus ergibt sich auch eine dritte Form von Teilfrage Nummer zwei, nämlich "verbindung(hong_kong,beijing,Vm2,K2)". Und dieser Aufruf wird von Erfolg gekrönt sein, mit den Variablenwerten Vm2=bahn und K2=150. Womit die vom Benützer eingegebene Frage beantwortet ist.

Noch ist allerdings nicht gesagt worden, wozu die bisher unerwähnten Konstruktionen von der Form "[Vm1,...]" im Kopf der Regeln "reise(_,_,[Vm1,...],_) :- ... ." dienen. Zwei Fragen stellen sich: Erstens, was die eckigen Klammern bedeuten, und zweitens, was mit den Variablen in den eckigen Klammern während der Problemlösung geschieht. Zuerst die eckigen Klammern: Sie bezeichnen in Prolog sog. Listen.

Eine Liste ist eine besondere Datenstruktur, die einem erlaubt, zwei Ausdrücke miteinander zur Deckung zu bringen, auch wenn sie (von "aussen" gesehen) nicht dieselbe Anzahl von Argumentspositionen haben. Bei "normalen" Prädikaten ist das bekanntlich nicht so: Ein Prädikat "pred(a,b,c)" kann man wegen der verschiedenen Anzahl von Argumenten nicht mit einem Prädikat "pred(X,Y)" zur Deckung bringen. Eine Liste [a,b,c] kann aber mit einer Liste [X¦Y] zur Deckung gebracht werden, obwohl man im zweiten Fall scheinbar nur zwei Argumente hat. Der Grund liegt darin, dass das besondere Zeichen "¦" nicht zwei Argumente voneinander trennt, sondern das erste Element einer Liste vom Rest der Liste - wie viele Elemente auch in diesem Rest stehen mögen. Man kann daher [a,b,c] mit [X¦Y] zur Deckung bringen und erhält die Variablenbindungen X=a sowie Y=[b,c], aber man könnte ebensogut [a,b,c,d,e] mit [X¦Y] zur Deckung bringen, und dann wären die Variablenbindungen X=a und Y=[b,c,d,e]. Der Operator "¦" greift also das erste Element aus einer beliebig langen (nichtleeren) Liste heraus und hinterlässt eine Liste mit den verbleibenden Elementen. Diese besonderen Eigenschaften der Listen-Datenstruktur werden zwar im vorliegenden Beispiel vorläufig noch nicht verwendet, aber später wird klar werden, warum man sie an dieser Stelle verwendet.

Die zweite und im gegenwärtigen Zeitpunkt wichtigere Frage betrifft den Zustand der Variablen in diesen Listen nach der Beantwortung der Ausgangsfrage. Bis anhin haben wir nämlich immer gesagt, dass sich der Wert einer Variablen innerhalb einer Klausel "nach rechts" ausbreitet; mit andern Worten: wann immer zu einem bestimmten Zeitpunkt der Antwort-Berechnung eine Variable gebunden wird, wird in allen späteren Schritten anstatt dieser Variablen nunmehr der soeben ermittelte Wert verwendet. Wenn dies die ganze Wahrheit wäre, würden somit die genannten Listen-Variablen im Kopf der Regel "reise(_,_,[...],_) :- .... " ungebunden bleiben, da ihr Wert ja erst ermittelt wurde, nachdem der Interpreter schon weit rechts vom Kopf angekommen war. Ein guter Teil der erstaunlichen Mächtigkeit von Prolog stammt nun daher, dass man eine Variable, die mehrmals vorkommt in einer Klausel, lange

Zeit ungebunden lassen kann, dass aber alle Vorkommen der Variable auf einen Schlag "vorwirkend" und "rückwirkend" an denselben Wert gebunden werden, wenn mit einem einzigen Vorkommen der Variable irgendwo in der Klausel ein "Treffer" erzielt wird. Für unser Beispiel heisst das, dass die gefundenen Werte sowohl für die Verkehrsmittel der Reise wie auch für die Kosten jeder einzelnen Teilstrecke "nach oben" zurückgeboten werden, in den Kopf der Regel "reise(_,_[Vm,...],[Km,...]) :- ..." und weiter in die entsprechenden Argumentspositionen der vom Benützer eingegebenen ursprünglichen Frage "reise(zürich,beijing,Vm,K)". Wenn diese Frage beantwortet ist, wird sie daher folgende Form angenommen haben: "reise(zürich,beijing,[bahn,swissair],[150,3000])".

Nun haben wir zwar eine Antwort auf unsere Frage erhalten, wie man von Zürich nach Beijing reisen könnte, und zudem eine von uns selbst nicht vorhergesehene Antwort, aber vielleicht sind wir damit noch nicht zufrieden. Vielleicht möchten wir alle Routenvorschläge vorgelegt bekommen, nicht bloss den ersten, auf den das Programm gestossen ist. Diese Forderung ist wiederum mit minimalem Zusatzaufwand zu erfüllen. Wir müssen uns dazu nur daran erinnern, was der Prolog-Interpreter tut, wenn er irgendwo eine Teilantwort nicht finden kann: Er geht zurück zur letzten Stelle, wo er eine weitere Antwort auf eine Frage finden könnte, und versucht diesen neuen Weg. Diesen Mechanismus (automatisches Backtracking) können wir nun dazu benützen, alle möglichen Antworten auf eine Frage (statt bloss der erstbesten) zu erhalten. Wir stellen zu diesem Zweck einfach folgende Frage:

"reise(zürich,beijing,Vm,K), write(Vm), nl, write(K), nl, fail."

Wie wird diese Mehrfachfrage beantwortet? Zuerst wird, wie soeben gezeigt, die erstbeste Antwort gefunden. Dann werden die ermittelten Variablenwerte ausgeschrieben, je auf eine neue Zeile ("nl" heisst "new line"):

```
"[swissair,bahn]
 [3000,150]"
```

Dann kommt der Interpreter zum Systemprädikat "fail". Wie der Name des Prädikats sagt, wird es immer fehlgehen. Und nun, da wir den Interpreter dazu gebracht haben zu glauben, er habe keine Lösung gefunden, wird er in den "automatischen Rückwärtsgang" gehen und sich zum ersten Punkt zurückziehen, wo er einen weiteren Beantwortungsversuch unternehmen kann. Im betrachteten Beispiel ist dies die zweite Möglichkeit, von Hong Kong nach Beijing zu reisen: Statt der Bahn kann man das Fugzeug (CAAC) benützen. Der Interpreter beschreitet diesen zweiten Weg und druckt die so gefundene zweite Lösung aus:

```
"[swissair,caac]
 [3000,500]"
```

Erneut trifft er auf das Systemprädikat "fail", erneut geht er zurück und sucht nach neuen Lösungen. Man könnte bei der Betrachtung des Programms meinen, damit seien nun alle Möglichkeiten ausgeschöpft, aber das trifft nicht zu: Zwar gibt es keine weiteren Möglichkeiten mehr, mit nur einmaligem Wechseln des Verkehrsmittel von Ausgangspunkt zum Ziel zu kommen, aber nunmehr wird der Interpreter eben auf der obersten

Ebene des Programms (dem neu eingefügten Teil) die dritte Variante der Regel "reise" versuchen, eben jene, bei der man in drei Teilstrecken vorgeht. Und hier wird nun, wie wohlbekannt, der Umweg über London gefunden werden, und zwar das erste Mal mit der Air-Lanka, und das zweite Mal mit der Bahn. Diese Lösungen werden ebenfalls ausgeschrieben, so dass wir am Schluss folgende Gesamtantwort auf dem Schirm stehen haben:

```
"[swissair,bahn]
 [3000,150]
 [swissair,caac]
 [3000,500]
 [air_lanka,cathay,caac]
 [300,1200,500]
 [air_lanka,cathay,bahn]
 [300,1200,150]
 [bahn,cathay,bahn]
 [250,1200,150]
 [bahn,cathay,caac]
 [250,1200,500]
no."
```

Das sind in der Tat alle möglichen Kombinationen von Verkehrsmitteln. Das etwas unmotiviert erscheinende "no" am Schluss erklärt sich daraus, dass der Interpreter ja schlussendlich nach dem letzten durch "fail" erzwungenen Misserfolg keine weiteren Lösungen mehr fand und daher annahm, für die Gesamtfrage gebe es keine Antwort.

Nun könnte man durchaus einwenden, mit diesem Wust an Rohinformation sei einem Benützer wenig gedient. Man möchte z.B. die billigste Kombination von Verkehrsmitteln errechnen lassen. Das ist aber aus einem besonderen Grund nicht ganz trivial: Wann immer der Interpreter in den "automatischen Rückwärtsgang" schaltet, vergisst er auf dem Weg zurück alle unterwegs gefundenen Variablenbindungen. Das ist auch richtig so, denn dieser Weg war ja erfolglos gewesen - schliesslich hatte er zu einem Misserfolg geführt. So ist es zumindest normalerweise, aber im letzten Beispiel hatten wir einen "künstlichen Misserfolg" erzwungen, mittels "fail". Die in den Variablenbindungen ausgedrückte Information war in diesem Fall gar nicht falsch gewesen, sie war bloss nicht die ganze Information gewesen, die wir bekommen wollten. Wir hatten sie zwar scheinbar retten können, bevor der Interpreter sie rückwärtsschreitend vergass, indem wir sie auf den Bildschirm schreiben liessen, aber für die weitere Verarbeitung steht sie nunmehr nicht mehr zur Verfügung - sie sitzt nur noch im Bildschirmspeicher, ist aber unzugänglich für das Programm. Wenn man sie weiterverarbeiten will (eben, um z.B. den billigsten Weg zu errechnen), muss man ein besonderes Systemprädikat benützen, das intern zwar genau so funktioniert wie der Trick mit dem "fail", das aber die gefundenen Variablenwerte akkumuliert und am Schluss zur Weiterverwendung bereitstellt: Das Prädikat "setof(V,E,SV)". In der Variablen E muss man eine Frage (oder deren mehrere) einschreiben, auf welche man alle möglichen Antworten haben will. In V muss man eine oder mehrere Variablen jener Frage(n) erwähnen, deren Werte man akkumuliert haben will. SV schliesslich lässt man ungebunden, denn hier wird die Liste mit allen akkumulierten Werten erscheinen. In unserem Beispiel wäre das:

"setof((Vm,K),reise(zürich,beijing,Vm,K),Result)."

und die Variable "Result" wird demnach am Schluss den folgenden Wert haben:

```
[([swissair,bahn],[3000,150]),([swissair,caac],[3000,500]),
([air_lanka,cathay,bahn],[300,1200,150]),([air_lanka,cathay,caac],
  [300,1200,500]),([bahn,cathay,bahn],[250,1200,150]),
 ([bahn,cathay,caac],[250,1200,500])]
```

Es ist eine Liste aus Paaren der Form (Vm,K), und jede der Variablen Vm und K ist ihrerseits eine Liste, nämlich aus Verkehrsmitteln und Kosten:

```
[ ([Vm1,Vm2,...],[K1,K2,...]) , ([...],[...]) , ...  ]
```

Das ist zwar für einen menschlichen Betrachter noch weniger übersichtlich als die ursprüngliche Form der Darstellung, aber das war ja auch nicht der Grund für die Einführung dieses neuen Prädikats: Das Resultat sollte für das Programm verdaulich sein, nicht für den Menschen. Der soll diese unhandliche Liste gar nicht sehen. Wenn wir jetzt die billigste Route errechnen lassen wollen, müssen wir folgende neue Regel ins Programm einfügen (irgendwo):

```
billigste_reise(Start,Ziel,BVm,BK)  :-
         setof((Vm,K),reise(zürich,beijing,Vm,K),Result),
         billigst(Result,BVm,BK).
```

Natürlich müssten wir nunmehr noch das Prädikat "billigst" definieren, das die Einzelkosten aller Teilstrecken aufaddieren und danach den kleinsten Gesamtbetrag ermitteln muss. Diese Aufgabe wollen wir aber noch verschieben, da unser ursprüngliches Programm in verschiedener Beziehung gravierende Mängel aufweist, deren Behebung weit dringender ist als das Einfügen dieser Verfeinerung.

1.5 Eine viel leistungsfähigere Datenbank: Rekursion

Der offensichtlichste dieser Mängel besteht sicher darin, dass wir einfach dekretierten, eine Reise könne aus nicht mehr als drei Teilstrecken bestehen. Das mochte ja beim gegenwärtigen Stand unserer Datenbank über Verkehrsmittel sinnvoll sein (wenn man einmal Rundreisen als nicht sinnvoll ausschliesst), aber was, wenn neue Daten eingefügt würden? Plötzlich wären durchaus sinnvolle Routen ausgeschlossen, und wir könnten gewisse Reiserouten nicht mehr berechnen, obwohl sie offensichtlich möglich wären. Dazu kommt das ästhetische Argument, dass es ganz einfach plump ist, für eine aus einer einzigen Teilstrecke bestehende Route eine erste Regel, für eine aus zwei Teilstrecken bestehende Route eine zweite Regel, für eine aus drei Teilstrecken bestehende Route eine dritte Regel, usw. usf., zu schaffen. Es ist ja offensichtlich, dass diese Regeln sich gesetzmässig generieren lassen, und das ist genau das, was der Computer besser und schneller kann als der Mensch. Wir müssen unser Programm so umschreiben, dass es selbständig die Gesamtroute zusammenstellt, und zwar sinnvollerweise zuerst jene aus der geringsten Anzahl von Teilstrecken, danach in aufsteigender Reihenfolge alle andern, wenn vom Benutzer gewünscht.

Wie kann das gemacht werden?

Wir ersetzen die drei Regeln "reise" durch die folgenden zwei Regeln:

```
reise(Start,Ziel,Vm,K)                    :-  verbindung(Start,Ziel,Vm,K).
reise(Start,Ziel,[Vm1|Vm2],[K1|K2])   :-
               verbindung(Start,Zwischenziel,Vm1,K1),
               reise(Zwischenziel,Ziel,Vm2,K2).
```

Was leisten nun diese zwei kurzen Regeln? Alles, und noch viel mehr, als die drei plumpen Regeln, die von ihnen ersetzt werden. Die erste Regel oben prüft, ob eine Reise zufälligerweise in einem einzigen Schritt, ohne Umsteigen, gemacht werden kann. Wenn dieser Glücksfall eintritt, ist das Problem natürlich gleich gelöst: Die Werte von "Start" und "Ziel" werden nach unten (ins einzige Teilziel, d.h. "verbindung") gereicht, die Werte von (einzigem) Verkehrsmittel und Kosten werden in der Datenbank nachgeschaut und als Antwort zurückgereicht. Wenn es sich aber erweist, dass es keine direkte Verbindung zwischen Start und Ziel gibt, wird der Versuch, die erste Regel zu benützen, mit einem Misserfolg enden. Dann wird, wie wir das schon kennen, eben einfach die nachfolgende Regel versucht. Das erste Teilziel dieser Regel ("verbindung") hat als zweites Argument eine Variable, die im Kopf der Regel nicht auftaucht ("Zwischenziel"), und daher ungebunden sein wird, wenn das zweite Zwischenziel in Angriff genommen wird. Hierin unterscheidet sich diese neue Definition des Prädikates "reise" nicht von der ersten, plumpen, Form. Dieses erste Zwischenziel versucht bekanntlich, eine Teilstrecke vom Startpunkt zu irgendeinem Ort zu finden; in unserem Beispiel ist dies London. Das zweite Teilziel hingegen ist nicht mehr "verbindung", wie in der plumpen Variante, sondern erneut "reise". Die Definition des Prädikates "reise" benützt also ebendieses Prädikat selbst: Die Definition ist zirkulär (rekursiv). Was geschieht, wenn "reise" sich selbst mit dem Wert "reise(london,beijing,Vm2,K2)" aufruft? Der Interpreter sucht nicht mehr eine direkte Verbindung von London nach (im Beispiel) Beijing, sondern eine (möglicherweise sehr indirekte) Reiseroute. Allerdings wird der Interpreter gleich wieder versuchen, die einfachstmögliche Lösung zu finden und probiert, sich selbst aufrufend, die erste Regel aus, also jene, wo man testet, ob eine direkte Verbindung in der Datenbank zu finden ist. Das wird, wie wir wissen, nicht gehen: Es gibt keine direkte Verbindung von London nach Beijing. Nun wird erneut die zweite Regel verwendet, d.h. es wird eine direkte Verbindung von London nach irgendwohin gesucht, um dann zu versuchen, eine (direkte oder eben indirekte) Verbindung von diesem Irgendwo nach Beijing zu ermitteln. Aus der Reihenfolge der Einträge ergibt sich (wie bekannt) eine direkte Verbindung von London nach Hong Kong. "reise" ruft sich nunmehr ein zweites Mal selbst auf, aber diesmal mit dem Wert "reise(hong_kong,beijing,Vm2,K2)". Und wiederum wird zuerst die einfache Lösung versucht, und diesmal klappt es auch: Es gibt eine direkte Verbindung, die zweite Regel "reise" muss nicht mehr benützt werden, und die Lösung ist gefunden.

Aber wie bekommen wir die Resultate aus der Lösung heraus? Erneut ist die Antwort: Sie ist schon da. Zwar sieht man davon nichts, während man die "Feder" der Rekursion spannt, indem man eine Regel sich dauernd selbst aufrufen lässt, aber sobald man dann die Feder "loslässt"

(sobald man einen direkten Treffer gefunden hat, nämlich die direkte Verbindung von Hong Kong nach Beijing), werden beim "Entspannen" der Feder alle Teilwerte von einer "Federwindung" in die nächste, höhere, nach oben geboten und dort in den zwei Listen akkumuliert, welches Resultat seinerseits nach oben geboten und an die dort wartenden Teilresultate angefügt wird, bis sich am Schluss, im obersten Term, das Gesamtresultat vollautomatisch akkumuliert hat. Dies tönt ein bisschen wie Zauberei oder Schummelei (je nach Weltauffassung), aber es funktioniert genau so. Der "Zauberer" ist natürlich der Interpreter, der im versteckten eifrig Buch führt über die verschiedenen Ebenen der rekursiven Aufrufe und was für Variablenbindungen wo gefunden wurden, so dass er am Schluss alle diese wartenden Werte beim Abrollen des rekursiven Jo-Jos an der richtigen Stelle an das sich schrittweise zusammensetzende Resultat ankleben kann.

Rekursion ist eine jener Angelegenheiten, die man abstrakt nur recht schwer verstehen kann, wenn man sie nie in Aktion gesehen hat. Wenn man sie genügend oft gesehen hat (z.B. beim Beobachten eines Tracers), erscheint sie einem bald als recht natürlich. Wer das Phänomen der Rekursion kennt, wird daher die obige Schilderung als selbstverständlich bis primitiv empfinden, und wer sie nicht kennt, wird daraus wohl bloss eine Art Ahnung ableiten können, was da so etwa vor sich gehen könnte. Wir werden deshalb keine weiteren Versuche unternehmen, sie noch plastischer zu schildern. Es muss genügen, darauf hinzuweisen, dass man mit den zwei obigen Regeln nicht nur die ursprünglichen drei Regeln ersetzt hat, sondern eine (im Prinzip) unbeschränkte Anzahl derartiger Regeln, für Reiserouten mit vier, fünf, sechs, etc. Teilstrecken. "Im Prinzip" deshalb, weil der Speicher eines jeden Computers beschränkt ist, und daher auch die Anzahl von Ebenen rekursiver Einbettung, über die ein Prolog-Interpreter Buch führen kann. Aber auch mit dieser selbstverständlichen Beschränkung ist die Rekursion eine ungemeine Arbeitserleichterung beim Programmieren.

Je mächtiger ein Instrument ist, desto mehr Unheil kann man anrichten, wenn man es unsachgemäss anwendet. Im Fall der Rekursion heisst das, dass man in ein eigentliches Schwarzes Loch fallen kann, wenn man es ungeschickt genug anstellt. Wenn man z.B. die zweite der oben eingeführten Regeln bloss ein bisschen umordnet, indem man die zwei Ausdrücke auf der rechten Seite vertauscht (und dabei die Variablennamen sinnentsprechend abändert), wenn man daraus also folgende Regel macht

```
reise(Start,Ziel,[Vm1¦Vm2],[K1¦K2])  :-
                reise(Start,Zwischenziel,Vm1,K1),
                verbindung(Zwischenziel,Ziel,Vm2,K2),
```

so wird man unvermeidlicherweise in eine unendliche Schleife geraten: Das Prädikat "reise" wird sich immer wieder selbst aufrufen, bevor es die Gelegenheit erhält, zu testen, ob nicht vielleicht eine direkte Verbindung zwischen dem jeweiligen Start und dem Ziel besteht. Der Test, ob die sog. Abbruchbedingung erfüllt ist, muss daher immer vor einem rekursiven Aufruf gemacht werden. Wenn man das vergisst, wird man beim ersten Versuch, das Programm laufen zu lassen, einige Zeit vor dem Bildschirm sitzen und sich wundern, was denn bloss passiert ist, bis dann die Meldung vom Betriebssystem kommt, man habe soeben den gesamten Speicherraum aufgefüllt (mit den Buchhaltungsangaben über

einige Dutzend bis Tausend leere rekursive Aufrufe - je nach Speichergrösse).

Aber auch mit dem korrekt geschriebenen Programm kann man seine Überraschungen erleben, wenn man die Frage entsprechend stellt. Während wir vorher bei der Frage

"?- setof((Vm,K),reise(zürich,beijing,Vm,K),Result)."

mit der ursprünglichen, plumpen Version des Programms immer Erfolg haben werden, kann uns diese Frage mit der neuen Version des Programms (mit rekursiven Aufrufen) u.U. in einem ähnlichen Schwarzen Loch verschwinden lassen. Es genügt, dass wir die Datenbasis unseres Programms durch die Angabe

"verbindung(beijing,zürich,swissair,4500)."

ergänzen (und von der Sache her ist das eine sehr vernünftige Ergänzung), und schon landen wir in einem weiteren Schwarzen Loch. Warum? Über dieser neuen Datenbasis sind nunmehr Rundreisen Zürich-Beijing-Zürich möglich. Unsere Definition einer Reise (d.h. die Definition des Prädikats "reise") hat nicht ausgeschlossen, dass man von Zürich nach Beijing reist, indem man zuerst nach Beijing reist, dann nochmals zurück nach Zürich, und erst dann endgültig nach Beijing. Natürlich lassen sich unbegrenzt viele derartige kleine Rundreisen einschieben, und man kommt schlussendlich immer noch an Ziel an. Bloss: Mit dem "setof"-Aufruf hat man ja nach <u>allen</u> möglichen Routen gefragt. Es gibt aber unbegrenzt viele Routen zwischen beliebigen zwei Punkten, wenn man derartige sinnlose Zwischen-Rundreisen zulässt. Also wird der Prolog-Interpreter eifrig wie ein Biber "leere" Rundreise-Varianten errechnen und nie zur "letzten" Route kommen; er wird nicht einmal ein Zwischenresultat ausgeben, sondern stumm weitermachen, bis ihm die Luft ausgeht.

1.6 Eine reichhaltigere Datenbank: Unterziele und Unter-Unterziele

Bis jetzt haben wir bloss Fragen der Reiseroutenberechnung betrachtet. Angenommen, wir schreiben das Programm korrekt und wir stellen keine unbeantwortbaren Fragen, können wir immerhin schon mit einem Vier-Zeilen-Programm alle möglichen Routen zwischen zwei Punkten errechnen (zwar gehören die Daten über die bestehenden direkten Verbindungen funktional gesehen auch zum Prolog-Programm, aber fairerweise darf man sie bei Grössenvergleichen nicht einberechnen). Aber unser Projekt war ja ursprünglich, vom Programm auch andersartige Schwierigkeiten, die im Laufe einer Reise auftreten können, lösen zu lassen. Dazu gehörte z.B. das Problem, wie man sich das Geld für die Reise beschaffen kann. Wir müssen an jedem Ort das Billett für die folgende Teilstrecke kaufen, und zu diesem Zweck müssen wir sicherstellen, dass wir genügend Geld haben. Das Geld können wir entweder (noch) auf uns tragen, oder wir können es von einem unserer Bankkonten abheben (sofern wir an Ort eins haben). Wie können wir dieses zusätzliche Wissen in ein Programm kleiden? Nun, fast genau so, wie wir es soeben in Worte gefasst haben:

```
reise(Start,Ziel,Taschengeld,Rest,Vm,K)                :-
              verbindung(Start,Ziel,Vm,K),
              kaufe_billett(Start,K,Taschengeld,Rest).

reise(Start,Ziel,Taschengeld,Rest,[Vm1|Vm2],[K1|K2])  :-
              verbindung(Start,Zwischenziel,Vm1,K1),
              kaufe_billett(Start,K1,Taschengeld,Taschengeld1),
              reise(Zwischenziel,Ziel,Taschengeld1,Rest,Vm2,K2).

kaufe_billett(Start,Kosten,Taschengeld,Rest)  :-
              Kosten < Taschengeld,
              Rest is Taschengeld - Kosten.
kaufe_billett(Start,Kosten,Taschengeld,Rest)  :-
              konto(Start,Stand),
              Geld is Taschengeld + Stand,
              Kosten < Geld,
              Rest is Geld - Kosten.
```

Wann immer wir eine Teilstrecke in Betracht ziehen, stellen wir sicher, dass wir sie uns auch leisten können. Sonst ziehen wir sie für unsere weitere Planung an diesem Punkt gar nicht mehr in Betracht und gehen zurück, eine andere Möglichkeit suchen. "Sich leisten können" heisst im Idealfall, dass unser Taschengeld dafür reicht, und genau dieser Test ist in der ersten Regel "kaufe_billett" codiert ("Kosten < Taschengeld"). Wenn dies der Fall ist, errechnen wir, was uns übrigbleibt: Der "Rest" ist ("is") unser ursprüngliches "Taschengeld" "minus" die "Kosten"; "is" ist der arithmetische Operator in Prolog, wenn man so will, der einzige Fall, wo man eine Zuweisung benützt in Prolog: "Rest := Geld - Kosten". Wenn unser Taschengeld nicht ausreicht, testen wir, ob wir am Ausgangsort ein Konto haben, und was sein Stand ist (in der zweiten "kaufe_billett"-Regel: "konto(Start,Stand)"), addieren es zu unserem Taschengeld ("Geld is Taschengeld + Stand") und testen, ob es nunmehr reicht. Wenn es reicht, ist natürlich alles gut. Was aber, wenn es nicht reicht? Nun, offensichtlich wird der Interpreter den Rückzug antreten - aber nur bis zur ersten Möglichkeit, eine andere Routenwahl durchzutesten. Dabei mag er auf eine billigere Variante in dieser früheren Phase der Routenplanung kommen, die ihm jetzt mehr Geld für jene Phase übriglässt, wo Geldprobleme ihn zum Abbruch der Planung gezwungen hatten. Mit dem so gesparten Geld kann er sich diese Phase jetzt vielleicht leisten.

Natürlich müsste man noch einige Verfeinerungen an diesem Programm anbringen (z.B. müsste man das "kleiner als", d.h. "<", ersetzen durch ein "kleiner als oder gleich", in Prolog geschrieben als "=<", auch müsste man verhindern, dass sinnlose Rundreisen eingebaut werden können), aber das Programm wird schon in dieser Form zeigen, wie einfach man in Prolog vermeintlich aufwendige Operationen programmieren kann (und zwar genau deswegen, weil man sie nicht als Operationen programmiert, sondern als Beziehungen - aber davon mehr später).

2. PROLOG IST KEINE DATENBANK-ABFRAGESPRACHE: SPRACHANALYSE

In der Zwischenzeit muss beim Leser der Eindruck entstanden sein, Prolog sei einfach eine Art Datenbank-Abfragesprache, und wer sich

nicht für Datenbanken interessiere, könne damit nichts anfangen. Falsch, ganz falsch. Datenbank-Anwendungen sind zwar das beste Beispiel, das Verhalten eines Prolog-Interpreters zu illustrieren, aber die eigentliche Leistungsfähigkeit von Prolog kommt erst bei andern Anwendungen voll zum Tragen. So ist z.B. der beste Compiler für ein Mainframe-Prolog (D.H.D. Warrens Dec-10-Prolog) selbst in Prolog geschrieben, und es ist vor allem die syntaktische Analyse von (künstlichen und natürlichen) Sprachen, für die Prolog nun tatsächlich weit geeigneter ist als jede andere Programmiersprache. Daher wollen wir ein solches Problem als nächstes Beispiel betrachten.

2.1 Die syntaktische Analyse von Sprachen

Zuerst allerdings sei kurz diskutiert, warum man sich überhaupt für die syntaktische Analyse, für das sog. Parsen, von Sprachen durch den Computer interessieren kann. Ein erster Grund ist schon genannt worden: Compiler sind bekanntlich sehr nützliche Dinge, und sie sind nichts anderes als Übersetzungsprogramme für Programmiersprachen. Und dazu muss man eben unausweichlicherweise die syntaktische Struktur eines "Satzes" der Ausgangssprache, der Programmiersprache, analysieren - wie sonst sollte man den Satz in die syntaktische Struktur einer andern Sprache, der Maschinensprache, überführen? Einen Schritt weiter könnte man zu gehen versuchen, wenn man auch natürliche Sprachen (also normale, menschliche Sprachen) statt bloss Computersprachen zu übersetzen versuchte. Auf diesem Gebiet wird seit den früheren Fünfzigerjahren gearbeitet, und in den letzten Jahren sind Programme entwickelt wurden, die nunmehr kommerziell angeboten werden (die dazu erforderliche sehr grosse Computerleistung musste erst billig genug werden). Dass derartige Programme enorm nützlich sind, muss wohl nicht näher begründet werden - und auch sie beruhen auf dem Übersetzen von Strukturen in andere Strukturen. Schliesslich sei noch auf die Möglichkeit hingewiesen, natürlichsprachliche "Frontends" für Datenbanken zu entwerfen: Programme, welche Anfragen an Datenbanken aus natürlichen Sprachen in die von der jeweiligen Datenbank verlangte interne Repräsentation übersetzen; normalerweise muss man ja Datenbankabfragen in einer bestimmten, von Datenbank zu Datenbank verschiedenen, besonderen Abfragesprache formulieren, die der normale Benützer kaum zu lernen gewillt ist. Mit Hilfe von natürlichsprachlichen "Frontends" kann sich auch der ungeübte Benützer endlich die Vorteile von Datenbanken zunutze machen. Auch hier muss man Strukturen übersetzen. Wie man sieht, ist diese Fragestellung sehr wichtig für alle Anwendungen, wo Sprachen betroffen sind.

Wie nun kann man eine (scheinbar) datenbankorientierte Sprache wie Prolog auf diese Problemstellungen ansetzen? Wir wollen einen sehr kleinen Ausschnitt der englischen Grammatik betrachten (Englisch deshalb, weil Deutsch eine ganze Reihe von zusätzlichen Problemen stellt, die am Anfang bloss den Blick für das Wesentliche verstellen könnten). Zuerst wollen wir diesen Ausschnitt der Grammatik definieren und dann sehen, wie Prolog ihn verwenden könnte, um englische Sätze syntaktisch zu analysieren. Wir wollen also letztlich ein Programm schreiben, das englische Sätze verschluckt und dann ihre syntaktische Struktur auswirft.

2.2 Ein Mini-Parser für das Englische

Als Beispielgrammatik verwenden wir den folgenden Satz von Regeln (Erklärungen dazu folgen gleich anschliessend):

2.2.1 Die Grammatik

```
1)  s   -->  np,  vp.
2)  np   -->  det, adj,  n.
3)  vp   -->  v.
4)  vp   -->  v, np.

5)  np --> [meat].

6)  det --> [the].
7)  det --> [a].
8)  det  --> [an].
9)  det --> [].

10) adj --> [].
11) adj --> [brown].

12) n --> [dog].
13) n --> [man].

12a) n --> [dogs].
13a) n --> [men].

14) v --> [eats].
15) v --> [barks].
16) v --> [eat].
17) v --> [bark].
18) v --> [is,barking].
19) v --> [is,eating].
```

Diese Mini-Grammatik sagt folgendes: Ein Satz ("s") besteht aus einer "Nominal-Phrase" ("np"), gefolgt von einer "Verbal-Phrase" ("vp"). Dies ist, was in Zeile 1 der Grammatik ausgedrückt wird. Eine Nominal-Phrase ist jener Teil eines Satzes, der um ein Substantiv (Nomen: "n") herum gebaut ist. Vor dem Substantiv steht in der Regel ein Artikel (ein Determiner: "det") und oft ein Adjektiv ("adj"); diese Angaben sind in Zeile 2 enthalten. Zeilen 3 und 4 sagen aus, dass eine Verbalphrase aus einer Verbform und sonst nichts bestehen kann (wie im Satz "Dogs bark"), oder aber aus einem Verb, gefolgt von einer weiteren Nominalphrase ("A dog is eating a sausage"). Zeilen 6 bis 8 sagen offensichtlich aus, dass die verschiedenen möglichen Formen des Artikels im Englischen "the", "a" und "an" sind. Was aber sagt die merkwürdige Regel 9 aus? Wie bis jetzt sicher klar geworden ist, bedeuten die eckigen Klammern, dass wir ein Wort vor uns haben, wie es im Satz erscheint, während die ungeklammerten Ausdrücke abstrakte grammatikalische Kategorien sind, die wir nie als solche in einem Satz sehen werden. Regel 9 sagt mithin, dass ein Artikel aus einer "Lücke" im Satz bestehen kann, also einfach: dass ein Artikel fehlen kann. Und tatsächlich: Im Satz "Dogs bark" finden wir keinen Artikel. Regel

10 sagt dasselbe über Adjektive aus: Die meisten Substantive in einem beliebigen Satz haben kein Adjektiv vorangestellt. Regel 5 sagt aus, dass "meat" ausschliesslich ohne Artikel verwendet wird, d.h. dass dieses eine Wort eine komplette Nominal-Phrase darstellt. Die restlichen Regeln (12 bis 19) führen an, welche Formen von Verben und Substantiven wir erwarten können (wie erwähnt, dies ist eine Mini-Grammatik ohne jeden Anspruch auf Vollständigkeit).

Diese Grammatik definiert hiemit folgende Sätze als korrekt:
- Brown dogs eat meat.
- A dog is barking.
- The brown man is eating a dog.

Folgende Sätze hingegen sind im Sinne dieser Grammatik nicht korrekt (weil sie nicht ausdrücklich als korrekt definiert sind):

- Dogs brown eat meat.
- The meat is brown.
- The red man is eating a dog.

Der zweite und der dritte Satz in dieser Serie sind nach allgemeinem Sprachverständnis natürlich korrekt, aber im Sinne der verwendeten Mini-Grammatik sind sie falsch. Hier müsste man nun die Grammatik erweitern und differenzieren. Schiesslich sind aber auch die folgenden Sätze im Sinne dieser Grammatik "korrekt", obwohl sie offensichtlich völlig falsch sind nach den normalen Regeln des Englischen:

- A dogs eat meat.
- Dog eat meat.
- Dogs bark meat.

Erneut gilt aber, dass wir hier einfach die (sehr engen) Grenzen unserer sehr kleinen Beispiel-Grammatik erreicht haben und nicht irgendwelche Beschränkungen des verwendeten Formalismus. Schliesslich wollen wir hier auch nicht eine umfassende Grammatik des Englischen vorlegen, sondern zeigen, wie man ein Prolog-Programm schreiben kann, das Sätze daraufhin analysiert, ob sie korrekt <u>im Sinne der zugrundegelegten Grammatik</u> sind. Die Grammatik selbst kann man dann später noch erweitern - das ist weniger ein Problem für den Programmierer als für den Linguisten.

2.2.2 Der "Parser"

Das Verblüffende ist, dass man die obige Grammatik nur ein klein wenig abändern muss, um daraus ein Prolog-Programm zu gewinnen, das die dargestellte Grammatik verwendet, um eingegebene Sätze syntaktisch zu analysieren. Dazu muss man bloss jeden Ausdruck (links und rechts der Pfeile) durch je zwei Argumente ergänzen, und die Pfeile selbst durch das übliche Zeichen für eine Inferenzregel (also ":-") ersetzen, in der folgenden Weise: Aus

```
s  -->  np,  vp.
```

machen wir

```
s(S0,S2)  :-  np(S0,S1), vp(S1,S2).
```

aus

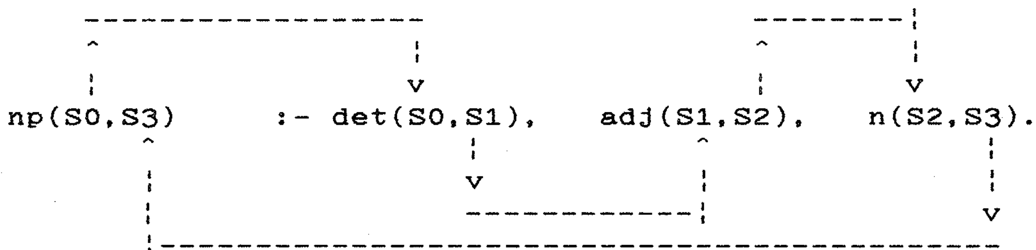
```
np  -->  det, adj,  n.
```

machen wir

```
np(S0,S3)  :-  det(S0,S1), adj(S1,S2), n(S2,S3).
```

usw.

Wie man sieht, werden die zwei zusätzlichen Argumente in systematischer Weise nach einem sehr einfachen Prinzip benannt: Das erste Argument im Kopf der Regel stimmt mit dem ersten Argument des ersten Ausdrucks auf der rechten Seite der Regel überein, das zweite Argument des Kopf-Ausdrucks stimmt mit dem zweiten Argument des letzten Ausdrucks auf der rechten Seite überein, und die übrigen Argumente in den Ausdrücken auf der rechten Seite sind immer so benannt, dass das zweite Argument eines Ausdrucks mit dem ersten Argument des folgenden Ausdrucks übereinstimmt. Das tönt wesentlich komplizierter als es ist, und alles wird viel einfacher, wenn man erklärt, was diese Art der Argumente-Benennung bezweckt: Ein Wert, der ins Argument S0 eines Regel-Kopfes (also z.B. oben "np(S0,S3)") eingefüllt wird, breitet sich in der bekannten Weise automatisch in den ersten rechtsseitigen Ausdruck "det(S0,S1)" aus, wird dort zu einem neuen Wert S1 verrechnet (auf eine Weise, die wir gleich erklären werden), dieses Resultat S1 wird ebenso automatisch nach rechts zum Ausdruck "adj(S1,S2)" geboten, weiter verrechnet, als S2 zur weiteren Bearbeitung nach rechts zu "n(S2,S3)" weitergereicht, von wo das Endresultat S3 schliesslich in den Kopf der Regel zurückgegegeben wird.



Was für Werte werden nun aber in die Argumente der Regel-Köpfe eingegeben, und was geschieht mit ihnen in den rechtsseitigen Ausdrücken? Am besten erklärt man das wiederum anhand eines Beispiels: Wenn man überprüfen will, ob der Satz "A dog is barking" korrekt (im Sinne der oben gegebenen Grammatik) ist, tippt man einfach folgende Frage ein

```
?- s([a,dog,is,barking],[]).
```

Wie man sieht, hat man den Satz in eine Liste von Einzelwörtern umgewandelt (das könnte man natürlich auch vom Progamm machen lassen, aber der Einfachheit halber lassen wir das mal weg) und fragt, ob das ein Satz ("s(...)") sei. Wir verschieben die Frage, weshalb man das zweite Argument als leere Liste "[]" eingibt. In gewohnter Prolog-Manier sucht der Interpreter Fakten oder Regeln, welche sich mit der Frage decken, und er findet offensichtlich sehr bald die Regel

```
s(S0,S2)  :-  np(S0,S1), vp(S1,S2).
```

Der Satz (in Listenform) breitet sich in den Ausdruck "np(...)" aus, welcher daher die folgende Form annimmt:

```
?- np([a,dog,is,barking],S1).
```

Das ist das erste Teil-Problem, das vom Interpreter in Angriff genommen wird. Im nächsten Schritt wird die Regel

```
2)  np  -->  det, adj,  n.
```

(resp. die in beschriebener Art durch Argumente S0 etc. erweiterte Version dieser Regel) aufgerufen, und die nächste Teilfrage

```
?- det([a,dog,is,barking],S1).
```

erzeugt. Diese Teilfrage wird die Regeln

```
6)  det --> [the].
7)  det --> [a].
8)  det  --> [an].
9)  det --> [].
```

benützen müssen. Diese Regeln mit terminalen Elementen (Wörtern) müssen von Prolog etwas anders übersetzt werden als die bisherigen Regeln. Die Regel 6 wird z.B. in die Regel

```
det(S0,S1)  :-  c(S0,the,S1).
```

übersetzt, wobei "c" ein Systemprädikat ist, das in der folgenden Weise definiert ist

```
c([Word|Rest],Word,Rest).
```

Da im ersten Argument ("S0") ein Satz (oder Satzteil) eingefüllt wird, spaltet dieses Systemprädikat "c" das erste Wort dieses Satzes ab, kontrolliert, ob es mit dem Wert des zweiten Arguments übereinstimmt, und gibt, wenn das der Fall ist, den Rest des Satzes im dritten Argument zurück. Im Falle von Regel 6 heisst das, dass überprüft wird, ob das zur Zeit vorderste Wort des Satzes (oder Satzteils) "the" ist. Da dies beim verwendeten Beispielsatz nicht der Fall ist, wird Regel 7 versucht, welche prüft, ob das Wort vielleicht "a" ist. Da dies nunmehr stimmt, wird der Rest des Satzes ("dog is barking") zur weiteren Verarbeitung zurückgeboten.

Regel 2 verlangt nun, dass als nächstes geprüft werde, ob ein Adjektiv vorliegt. Wir wissen, dass dies nicht der Fall ist. Es ist schon erwähnt worden, wie sog. fakultative Elemente in der Grammatik dargestellt werden, nämlich durch "Lücken" im Eingabesatz. Wie werden diese Regeln aber vom Präprozessor aus Grammatik-Regeln in Prolog-Regeln übersetzt?

Von den Regeln

```
10) adj --> [].
11) adj --> [brown].
```

wird 11 vom Präprozessor in der normalen Weise übersetzt, Regel 10 hingegen wird zu

10a) adj(S,S) :- true.

Wie wird diese Regel von Prolog interpretiert? "true" ist ein Systemprädikat, das immer wahr ist; daher wird der Aufruf

"?- adj([dog,is,barking],S2)."

unter Benützung von Regel 10a sofort gelingen, und S (in Regel 10a, das mit S2 im Aufruf zur Deckung kommt) wird die unveränderte Wortliste [dog,is,barking] zurückgeben. Das ist aber genau das, was wir brauchen, um das Fehlen eines Adjektivs als grammatikalisch akzeptabel erscheinen zu lassen. Da Regel 10 vor Regel 11 steht (und daher zuerst abgearbeitet wird), wird der Interpreter zuerst davon ausgehen, ein Adjektiv fehle, und nur, wenn sich das später als Irrtum herausstellt, wird er backtrackenderweise auch noch Regel 11 versuchen (und in einer grösseren Grammatik alle anderen möglichen Adjektive durchprobieren). Wenn man Regel 10 hingegen nach Regel 11 anführte, würde der Interpreter zuerst von der Annahme ausgehen, eine vorgelegte Nominalphrase enthalte tatsächlich ein Adjektiv, und nur im Falle eines Misserfolgs würde er schauen, ob vielleicht keins da ist. Hier kommen also empirische Werte zum Tragen: Je nachdem, wie man die Häufigkeitsverteilung gewisser Phänomene einschätzt, wird man die Reihenfolge von solchen Regeln verändern, um ein möglichst effizientes Programm zu erhalten. Auf die Korrektheit des Programmes aber hat die Reihenfolge von Regeln keinen Einfluss.

Das nächste zu lösende Teilziel der Anfrage np(...) wird prüfen, ob das nunmehr folgende Wort ein Substantiv ist, und ein Blick auf die Grammatik wird zeigen, dass Regel 12 zur Bejahung dieser Frage führen wird. Damit ist Regel 2 insgesamt abgearbeitet worden. Da Regel 2 von Regel 1 aufgerufen worden war, und zwar vom ersten rechtsseitigen Term ("np"), muss nunmehr dort weitergefahren werden, nämlich mit dem zweiten Term von Regel 1, also "vp". Regel 3 sagt dem Interpreter, dass eine Verbalphrase aus einem Verb allein bestehen kann, worauf die Liste aller möglichen Verbalformen (Regeln 14 bis 19) durchsucht wird. Regel 18 gelingt. Nunmehr ist der Eingabesatz restlos konsumiert worden: Der Wert der Variable S2 in "vp(S1,S2)" hat deshalb den Wert "[]" angenommen und wird nach oben zurückgeboten. Auf der obersten Ebene, in Regel 1, wird dieser Wert dabei mit dem "[]" in der Ausgangsfrage übereinstimmen. Diese leere Liste in der Ausgangsfrage hatte also die Bedeutung von "analysiere den folgenden Satz bis zum Ende ". Wenn man dort nicht eine leere Liste als Variablenwert eingegeben hätte, sondern die Variable ungebunden gelassen hätte, hätte dies der Frage entsprochen "teste, ob die folgende Folge von Wörtern mit einem korrekten Satz beginnt ".

Hier sei noch besonders darauf hingewiesen, dass durchaus mehrere terminale Symbole (also Wörter im Eingabesatz) aufs Mal "aufgefressen" werden können, wie im vorliegenden Fall "is barking". Das ist besonders praktisch bei Nominalfügungen wie "Logic Programming", die zu einem einzigen festen Begriff geworden sind und als solcher analysiert werden müssen.

Anhand eines (allerdings sehr einfachen) Beispiels haben wir nun

gesehen, wie eine Grammatik von einem Prolog-Interpreter sozusagen unverändert als Syntaxanalyseprogramm verwendet werden kann, weil der in Prolog eingebaute Präprozessor die Grammatik vollautomatisch in ein lauffähiges Prolog-Programm übersetzt. Allerdings haben wir damit erst einen sog. Akzeptor geschrieben, d.h. ein Syntaxanalyseprogramm, das mit "ja" und "nein" antwortet, je nachdem, ob ein Satz "syntaktisch wohlgeformt" ist oder nicht. Wenn wir aber z.B. einen Compiler schreiben wollen, wird uns das nicht genügen; in diesem Fall wollen wir, dass das Analyseprogramm das Resultat seiner Bemühungen, also die Syntaxstruktur des Eingabesatzes, ausgibt, damit sie weiter verarbeitet werden kann.

2.2.3 Das Errechnen der Sytaxstruktur

Erneut dürfen wir überrascht feststellen, dass eine minime Abänderung der ursprünglichen Grammatik genügt, um aus einem Akzeptor einen richtiggehenden Parser zu machen. Dazu fügen wir in der Grammatik in jeden Term (links und rechts der Pfeile) ein zusätzliches Argument ein. Die Grammatik sieht dann so aus:

```
1)  s(s(Np,Vp))  -->        np(Np),  vp(Vp).
2)  np(np(Det,Adj,N))  -->  det(Det), adj(Adj), n(N).
3)  vp(vp(V))  --> v(V).
4)  vp(vp(N,Np))  -->       v(V), np(Np).

5)  np(np(meat)) -->        [meat].

6)  det(det(the)) -->     [the].
7)  det(det(a)) -->    [a].
8)  det(det(a))  -->   [an].
9)  det(det([])) -->   [].

10) adj([]) -->             [].
11) adj(adj(brown)) -->     [brown].

12)  n(n(dog)) -->     [dog].
13)  n(n(man)) -->     [man].
12a) n(n(dogs)) -->     [dogs].
13b) n(n(mans)) -->     [men].

14) v(v(eats)) -->    [eats].
15) v(v(barks)) -->   [barks].
16) v(v(eat)) -->         [eat].
17) v(v(bark)) -->        [bark].
18) v(v(is_barking)) -->   [is,barking].
19) v(v(is_eating)) -->    [is,eating].
```

In den Termen links der Pfeile "skizzieren" wir in dem einen zusätzlichen Argument die Syntaxstruktur der ganzen Regel "im Umriss". So sagt z.B. der Ausdruck "s(Np,Vp)" im Kopf der ersten Regel

```
1)  s(s(Np,Vp))  -->        np(Np),  vp(Vp).
```

aus, dass die Struktur eines Satzes ("s") in zwei gleichrangige Hälften zerfällt, die wir hier "Np" und "Vp" genannt haben (obwohl man sie bekanntlich irgendwie sonst nennen könnte: "Eins" und "Zwei", oder ähnlich). Die innere Struktur dieser zwei Hälften ist der Regel 1 noch nicht bekannt; deshalb gibt sie ihnen auch nur Variablen als Namen. Das ist, was wir unter "im Umriss skizzieren" verstanden haben: Die <u>äussere Form</u> der Syntaxstruktur ("s(_,_)") können wir schon jetzt festlegen, aber den <u>Inhalt</u> ("Np","Vp") können wir noch nicht einfüllen. Dies kann erst geschehen, wenn die Variablen "Np" und "Vp" ermittelt worden sind, und das ist möglich, sobald die Ausdrücke rechts des Pfeiles evaluiert worden sind.

Beim Evaluieren dieser rechtsseitigen Ausdrücke passiert natürlich

genau dasselbe: Auf jeder Ebene der Evaluation kann man jeweilen die äussere Form festlegen, muss aber das Auffüllen des Inhalts auf später veschieben. Sobald man das erste terminale Element (Wort im Eingabesatz) antrifft, findet dieses dauernede "Abschieben der Verantwortung" aber ein Ende: Die Regel 7 z.B.

```
7) det(det(a)) -->   [a].
```

gibt sowohl Form wie Inhalt der Syntaxstruktur an: Die Form ist "det(_)", und der Inhalt der Wert "a" (der unbestimmte Artikel selbst). Beides, Form und Inhalt, werden nunmehr in der bekannten Weise nach oben zurückgeboten, wo sie sich Schritt um Schritt in die schon vorbereiteten "Strukturskelette" einfügen und, mit den Werten anderer terminaler Elemente kombiniert, schliesslich die Gesamtstruktur des Satzes ergeben. Auf diese Art würde sich für den Satz "Brown dogs bark" die Syntaxstruktur "s(np(det([],adj(brown),n(dogs)),vp(v(bark)))" ergeben.

Eine andere Art, den Aufbau der Syntaxstruktur zu beschreiben, wäre die: Anhand der beschriebenen Strukturterme lässt man das Programm intern festhalten, welchen Weg es beim Ausführen der Analyse genommen hatte, und die "Spur" dieses Marsches durch die Grammatik ist dann genau die Syntaxstruktur des analysierten Satzes.

2.2.4 Das Einfügen zusätzlicher Tests

Es kann nicht der Sinn dieses Überblicksartikels sein, in die Feinheiten der Syntaxanalysetechnik zu gehen. Anhand eines einzigen weiteren Beispiels möchten wir aber zeigen, wie ausserordentlich einfach es ist, in einem Prolog-Programm der geschilderten Art weitere Tests einzufügen, die in praktisch allen anderen Programmiersprachen zu grösseren Modifikationen zwingen würden.

So ist es offensichtlich notwendig, in der obigen Grammatik noch festzuhalten, dass Nominalphrase und Verbalphrase im Numerus übereinstimmen müssen: "Dog bark" ist ebenso falsch, wie "A dogs barks". Wir müssen, mit anderen Worten, sicherstellen, dass Artikel, Substantiv und Verb alle entweder im Singular oder im Plural stehen.

Wiederum können wir diese neue Leistung erbringen, indem wir in den entsprechenden Termen ein weiteres Argument einfügen, das wir mit der Variable "Number" besetzen. Das ergibt die folgende Version der Grammatik:

```
1)  s(s(Np,Vp))  -->        np(Np,Number),  vp(Vp,Number).
2)  np(np(Det,Adj,N),Number)  -->  det(Det,Number), adj(Adj),
                                   n(N,Number).
3)  vp(vp(V),Number)  --> v(V,Number).
4)  vp(vp(N,Np),Number)  -->       v(V,Number), np(Np,_).

5)  np(np(meat),sing) -->        [meat].

6)  det(det(the,_)) -->     [the].
7)  det(det(a),sing) -->    [a].
8)  det(det(a),sing)  -->   [an].
9)  det(det([]),plur) -->   [].

10) adj([]) -->             [].
11) adj(adj(brown)) -->     [brown].

12)  n(n(dog),sing) -->     [dog].
13)  n(n(man),sing) -->     [man].
12a) n(n(dog),plur) -->     [dogs].
13b) n(n(man),plur) -->     [men].

14) v(v(eat),sing) -->    [eats].
15) v(v(bark),sing) -->   [barks].
16) v(v(eat),_) -->         [eat].
17) v(v(bark),_) -->        [bark].
18) v(v(bark),sing) -->   [is,barking].
19) v(v(eat),sing) -->    [is,eating].
```

Das ist alles.

Wie aber soll diese minime Änderung die geforderte Leistung erbringen?! Nun, sobald der Parser den Artikel der Nominalphrase antrifft, wird der Wert der Variable "Number" dort gebunden an die Konstante "plur" oder "sing", je nachdem, ob der Artikel eben im Plural oder Singular ist. Bemerkenswert ist dabei, dass ein fehlender Artikel, also eine "Lücke" im Eingabesatz, ebenfalls einen Numerus hat, nämlich Plural (nur "dogs bark" is korrekt, "dog barks" hingegen nicht). Wenn der Artikel "the" ist, kann der Numerus noch nicht bestimmt werden, und die Variable bleibt ungebunden. Der soeben ermittelte Variablenwert breitet sich nach oben aus, in die Variable "Number" in "np(Np,Number)" und in "vp(Vp,Number)". Von dort wiederum wird der Wert nach unten gegeben, im Falle der Nominalphrase also in den Term "n(N,Number)", im Fall der Verbalphrase in den Term "v(V,Number)", und dort erzwingt der Wert der Variablen, "sing" oder "plur", dass nur eine Substantiv- oder Verbform im entsprechenden Numerus aus dem Eingabesatz akzeptiert wird. Allein durch die Wahl des gleichen Variablennamens konnten wir so die Übereinstimmunbg im Numerus von Nominal- und Verbalphrase erzwingen.

3. STRUKTURTRANSFORMATIONEN

Natürlich kann es kein Selbstzweck sein, einen englischen Satz syntaktisch zu analysieren. Man wird das Analyseresultat im Normalfall bloss als Zwischenresultat betrachten, um daraus z.B. eine Übersetzung des Ausgangssatzes in Prolog-Ausdrücke zu erreichen, die man dann über einer Datenbank evaluieren kann. Auf diese Art kann man ein eigentliches Fragenbeantwortungssystem schreiben, das auf englische Fragen entsprechende Antworten gibt (allerdings werden die Antworten nicht auf Englisch sein - das ist dann nochmals ein ganz eigenes Kapitel). Wie kann man also vom Syntaxanalyse-Resultat weitergehen? Wie kann man eine erste Struktur (eben die Syntaxstruktur) in eine zweite (die Prolog-Ausdrücke, die man über der Datenbank evaluieren kann) übersetzen? Hier nun wird noch offensichtlicher werden als bisher, dass Prolog ganz besonders geeignet ist, Strukturen zu verrechnen.

3.1 Übersetzung von Syntaxstrukturen in Semantische Strukturen

Wir könnten z.B. wünschen, die folgenden Sätze in Einträge einer kleinen Datenbank zu übersetzen:

A dog is a mammal.
Mammals are animals.
A warm-blooded animal is a mammal.
Dogs bark.

Alle Sätze dieses Typs drücken allgemeine Wahrheiten, d.h. Regeln aus: Alle Hunde sind Säugetiere, alle Hunde bellen usw. Es ist daher vernünftig, als Übersetzung dieser Sätze in eine Prolog-basierte Datenbank folgende Prolog-Klauseln anzunehmen:

```
mammal(X)      :-     dog(X).
animal(X)      :-     mammal(X).
mammal(X)      :-     animal(X), warm_blooded(X).
bark(X)        :-     dog(X).
```

Um die Syntaxstrukturen dieser Sätze in die gegebenen Prolog-Regeln zu überführen, müssen wir Regeln schreiben, die im Prinzip folgendermassen aussehen:

```
interpret_sentence(Syntax,Semantics).
```

Die Variable "Syntax" wird dabei als Input-Kanal verwendet, in den man die Syntaxstruktur des zu interpretierenden Satzes gibt, und "Semantics" dient als Output-Kanal. Der Aufruf würde also im Falle des ersten oben gegebenen Beispielsatzes folgendermassen aussehen:

```
interpret_sentence(s(np(det(a),[],n(dog)),
                     vp(v(be),np(det(a),[],n(animal)))),Semantics).
```

Die Definition dieser Regel muss

1. sicherstellen, dass beide Nominalphrasen indefinit sind (den Artiekl "a" verwenden, oder aber keinen Artikel verwenden),

2. die Substantive extrahieren, da sie zu Prädikaten in der Übersetzung gemacht werden müssen,

3. sicherstellen, dass diese Prädikate über derselben Variable ("X") definiert sind,

4. sicherstellen, dass das Verb "to be" ist,

5. sicherstellen, dass nur das Subjekt des Satzes ein Adjektiv enthält.

Das lässt sich in diesem einfachen Fall so machen, dass diese Bedingungen als instantiierte Strukturen im ersten Argument von "interpret_sentence" definiert werden, während die zu übersetzenden Werte (also die Substantive und Adjektive) dort als ungebundene Variablen erscheinen:

```
interpret_sentence(s(np(det(a),Adj,n(Noun1)),
                     vp(v(be),np(det(a),[],n(Noun2)))),
                   (Head :- Body1, Body2))
         :-   Head=..[Noun2,X],
              Body1=..[Noun1,X],
              interpret_adjective(X,Adj,Body2).
```

Die instantiierte Struktur "det(a)" z.B. lässt diese Regel nur auf indefinite Singularsätze ansprechen; die ungebundene Variable "Noun1" hingegen dient dazu, das Substantiv des Subjekt (was immer es auch sei) aufzunehmen und in den entsprechenden Term auf der rechten Seite der Regel (also "Body1=..[Noun1,X]") zu übertragen". Der hier zum ersten Mal verwendete Operator "=.." bewirkt, dass aus einer Liste ein Prädikat mit Argumenten gemacht wird: P=..[dog,X] resultiert in einer Instantiierung von P als "dog(X)"; aus P=..[gives,john,apple,mary] wird "gives(john,apple,mary)".

Zusätzlich müssen wir nun noch ein analoges Interpretations-Prädikat für Adjektive definieren:

```
interpret_adjective(X,[],true).
interpret_adjective(X,adj(Adj),IAdj) :-   IAdj=..[Adj,X].
```

Ein Adjektiv wird durch diese Regel in ein weiteres Prädikat übersetzt, das über derselben Variable definiert ist wie die aus den Substantiven abgeleiteten Prädikate (um das sicherzustellen, wird die Variable "X" im Prädikate "interpret_adjective" an erster Stelle genannt). Als Übersetzung für ein fehlendes Adjektiv hingegen (im Syntaxbaum als "[]" erscheinend) wird "true" geliefert, das vom Prolog-Interpreter als immer wahr behandelt, d.h. faktisch ignoriert wird.

Das zweite Argument des Prädikats "interpret_sentence", also "(Head :- Body1, Body2)", bewerkstelligt, dass die soeben generierten Prädikate in der korrekten Form einer Prolog-Regel zusammengestellt werden. Damit ist (für diesen einen Satztyp) die Übersetzung auch schon gemacht. Natürlich müsste man nun noch eine analoge Regel für

indefinite Pluralsätze schreiben, aber das ist ebenso einfach; die Regel für Adjektive müsste natürlich nicht neu geschrieben werden - sie könnte von allen Regeln mitverwendet werden.

3.2 Der Grund für die Einfachheit von Strukturoperationen in Prolog

Der Hauptzweck dieses einfachen Beispiels war zu zeigen, wie einfach man in Prolog mit Strukturen operieren kann, weil Prolog selbst alle Strukturvergleichs- und Strukturaufbauoperationen schon selbst macht: Wir müssen die Struktur

```
s(np(det(a),[],n(dog)), vp(v(be),np(det(a),[],n(animal))))
```

nicht mühsam aufsplittern, um die tief in der Struktur vergrabenen Substantive zu extrahieren, wenn wir sie in Prädikate übersetzen wollen - das tut Prolog in einem einzigen Schritt automatisch, wenn wir die teilweise instantiierte Struktur

```
s(np(det(a),Adj,n(Noun1)), vp(v(be),np(det(a),[],n(Noun2))))
```

"drauflegen" (matchen). Und in analoger Weise bauen wir Strukturen auf, indem wir einfach eine "Grobstruktur" mittels ungebundener Variablen, sozusagen als Beispiel, skizzieren, wie z.B.

```
(Head :- Body1, Body2)
```

in das wir dann schrittweise die "Feinstrukturen" einfüllen, welche ihrerseits zuerst als Grobstruktur definiert werden, usw. In dieser Weise lassen sich auch sehr komplexe Strukturtransformationen nicht nur sehr einfach, sondern auch sehr effizient programmieren: Der Zugriff zu einem tief unten in einer komplexen Struktur vergrabenen Term lässt sich in der Regel in einem einzigen Unifikationsschritt machen, was natürlich zu entsprechend kurzen Rechenzeiten führt. Der Aufbau von Strukturen kann sogar (wie im angegebenen Beispiel) im gleichen Schritt bewerkstelligt werden, was erneut Zeit spart. In sozusagen allen anderen Programmiersprachen müsste man diese Strukturoperationen explizit programmieren, was nicht nur den Programmierer, sondern nachher auch den Computer viel Zeit kosten wird.

4. SCHLUSS

Wir hoffen gezeigt zu haben, dass Prolog eine Sprache ist, welche besonders für den Nicht-Informatiker von grösster Bedeutung ist. Prolog erlaubt es, rasch zum Kern der wirklich interessanten Probleme vorzustossen. Prolog befreit den Programmierer von sehr viel uninspirierender Kleinarbeit, die in traditionellen Sprachen geleistet werden muss, um überhaupt einmal ein Programm zum Laufen zu bringen (Deklarieren von Datenstrukturen, Deklarieren und Initialisieren von Variablen, usw.). Die auf der Logik basierende Struktur von Prolog führt den Programmierer dazu, unwillkürlich und ohne bewusste Anstrengung strukturiert und logisch durchsichtig zu programmieren.