Logikorientierte Programmierung

Teil I
2006/2007
Ulrich Neumerkel
Technische Universität Wien

Inhaltsverzeichnis

1 Einführung

Diese Lehrveranstaltung behandelt das Programmieren in der logikorientierten Programmiersprache Prolog. Die Übungsumgebung dazu wird in Anhang A und B beschrieben. Weitere Informationen zu dieser Lehrveranstaltung finden sich unter der Adresse http://www.complang.tuwien.ac.at/ulrich/lp und direkt im Labor.

1.1 Geschichte

Die Programmiersprache Prolog wurde 1972 von Alain Colmerauer und Phillipe Roussel an der Universität Marseille entwickelt. Anfänglich wurde Prolog für natürlichsprachige Dialogsysteme und andere Anwendungen im Bereich der künstlichen Intelligenz eingesetzt. Weite Bekanntheit erlangte Prolog, als es Anfang der achtziger Jahre vom japanischen Handels- und Industrieministerium (MITI) für die Entwicklung von Systemen der 5. Generation (wissensverarbeitende Systeme) auserkoren wurde. In weiterer Folge verbreitete sich Prolog auch in Europa und den Vereinigten Staaten.

Ausgehend von Prolog gibt es viele Weiterentwicklungen. Die Erweiterungen von Prolog um *constraints* eröffneten viele neue Anwendungsbereiche. Der Kern der Sprache aber hat sich bewährt und ist unverändert geblieben.

1.2 Anwendungsbeispiele

Heute wird Prolog für kommerzielle Anwendungen auch außerhalb der künstlichen Intelligenz eingesetzt. Die folgenden Beispiele zeigen typische Anwendungsbereiche für Prolog.

Stimmgesteuerte Systeme. Auf der Internationalen Raumstation ISS wird das prologbasierte stimmgesteuerte System Clarissa eingesetzt. Bei Wartungsaufgaben und Experimenten muss die Besatzung genaue bisher nur schriftlich festgehaltene Arbeitsabläufe durchführen. Clarissa assistiert beim rein stimmgesteuerten Nachschlagen und führt durch die Arbeitsabläufe. Hände und Sicht bleiben frei für die Arbeiten in Schwerelosigkeit.

Automatische Netzwerkkonfiguration. Microsoft Windows NT verwendet Prolog eingebettet in C++ zur automatischen Netzwerkkonfiguration (*applet NCPA.CPL*). Aus den Beschreibungen über vorhandene Netzwerktreiber und Karten (*NDIS — network driver interface specification*) ermittelt Prolog eine lauffähige Konfiguration. Entscheidend für Prolog war die rasche Verfügbarkeit eines lauffähigen Prototypen, der innerhalb weniger Tage fertiggestellt wurde.

Intelligente Datenbankschnittstelle. Die Wirtschaftsprüfungs- und Unternehmensberatungsgesellschaft PricewaterhouseCoopers (vormals Price Waterhouse und Coopers & Lybrand) entwickelte das in C, Prolog und Java geschriebene EDGARSCAN (http://edgarscan.tc.pw.com), um auf die von der U.S.-amerikanischen Securities and Exchange Commission (SEC) verwalteten Unternehmensdatenbank EDGAR, die

Quartalsberichte börsennotierter Firmen erfasst, zugreifen zu können. Prolog wird hier zur Analyse der frei strukturierten Geschäftsberichte verwendet.

Produktionsplanung. Der französische Flugzeughersteller Dassault Aviation verwendet zur Produktionsplanung ein in Prolog (um *constraints* erweitert) geschriebenes *decision support system* (PLANE), das Lagerkosten und Stehzeiten minimiert.

Juristisches Expertensystem. Zur Vereinfachung der Genehmigung der Ein- und Ausfuhren landwirtschaftlicher Produkte verwendet das belgische Wirtschaftsministerium ein in Prolog geschriebenes Expertensystem (EUREX). EUREX begründet Entscheidungen in französischer und flämischer Sprache. Durch EUREX werden ca. 500 Seiten EU-Recht sowie nationale Regelungen erfasst, die laufend erweitert werden.

Umwelterklärungen. U.S.-amerikanische Betriebe müssen jährlich der Umweltbehörde EPA (*environmental protection agency*) eine Erklärung über die Verwendung und Entsorgung von Chemikalien vorlegen. Pro Chemikalie ist ein Formular mit 300 Einträgen auszufüllen, wofür bei manueller Bearbeitung drei Arbeitstage anfallen. Größere Betriebe reichen 100 bis 1000 derartige Formulare ein. Durch das in Prolog geschriebene Experten-Datenbanksystem Syllog reduzierte IBM die Bearbeitungszeit eines Formulars auf eine halbe Stunde. Bei Betriebsprüfungen begründet Syllog jeden Formulareintrag.

Echtzeitexpertensystem. Die japanische Baufirma Toda verwendete für Bohrungen des Tokyo Bay Tunnels ein Expertensystem in Prolog, das alle 10 Sekunden über den Fortgang der Bohrarbeiten berichtet und Korrekturen vorschlägt und begründet.

Datenbanken — *legacy system mediation services.* Firmen, in denen vielen Jahren EDV eingesetzt wird, verfügen eine Unzahl unterschiedlicher und voneinander unabhängiger Datenbestände, die auf verschiedensten Rechnern existieren (sog. *legacy systems*). Der Datenaustausch in einer derartigen Umgebung ist äußerst aufwendig, da jeweils eigene Programme zur Datenextrahierung und -konvertierung benötigt werden. Der Flugzeughersteller Boëing entwickelte einen in Prolog realisierten *mediator* zum transparenten Zugriff auf sämtliche Datenbestände.

Multimedia-CAM. Die japanische Optikerkette Paris Miki verwendet ein prologbasiertes System für Entwurf, Fertigung, Vertrieb und Verkauf maßgeschneiderter Brillen. Prolog entwirft nach dem über eine Videokamera aufgenommenen Kundenbild, ärztlichen Befunden und Kundenwünschen ein Brillenmodell, das probeweise über das Bild des Kunden gelegt wird. Die Maßbrille wird innerhalb einer Stunde gefertigt.

So verschieden die Anwendungen sind, haben sie und viele andere erfolgreiche Prologanwendungen einige gemeinsame Eigenschaften:

- Die Anwendungen bauen auf bestehende Systeme auf. Sie ersetzen nicht bestehende Systeme traditioneller Technologien, sondern erweitern sie um Fähigkeiten, die mittels herkömmlicher Technologien nur schwer zu realisieren sind. Prolog wird gemeinsam mit anderen Programmiersprachen verwendet. Die graphische Benutzeroberfläche, die Anbindung an Betriebssysteme und Datenbanksysteme wurde in anderen Sprachen realisiert; etwa in C, C++, Wafe, TCL, SQL.

- Die genaue Problemstellung musste erst ausformuliert werden. Zur Validierung von Konzepten mussten Prototypen rasch erstellt werden. Ein exploratives Vorgehen war erforderlich.

- Das Endprodukt musste in sehr kurzer Zeit fertiggestellt werden.

- Während der Systementwicklung veränderten sich die Anforderungen. Die Anwendungen werden ständig erweitert. Die Eingabe neuen Wissens erfolgt durch Personen ohne Programmierkenntnisse (Chemiker, Juristen etc.). Das Wissen wird in spezialisierten Sprachen (also nicht direkt in Prolog) repräsentiert, die ihrerseits in Prolog geschrieben sind.

- Es wurden moderne Prologimplementierungen eingesetzt —wie das in dieser Laborübung verwendete SICStus Prolog, siehe http://www.sics.se/ps/sicstus.html— die punkto Effizienz und Zuverlässigkeit industriellen Anforderungen gerecht werden.

1.3 Programmieren in Logik

Der Name Prolog ist ein Kürzel für *programmation en logique*, Programmieren in Logik. Die grundlegenden Konstrukte von Prolog stammen aus der Logik. Aus diesem Grund wird Prolog als **logikorientierte Programmiersprache** bezeichnet. In Prolog geschriebene Programme können meist als logische Formeln gelesen werden. Logische Formeln ihrerseits können als einfache deutsche Sätze, die Sachverhalte beschreiben, gelesen werden. Deshalb können auch Prolog-Programme als deutsche Sätze gelesen werden.

Beim Lesen eines Prologprogramms kann man so auf das eigene Sprachgefühl zurückgreifen, um ein Programm zu verstehen oder die Richtigkeit einer Definition zu überprüfen. Die meisten fehlerhaften Programme klingen bereits falsch, wenn man sie laut liest.

Umgekehrt geht man beim Schreiben eines Programms oft von einer informellen deutschen Beschreibung aus. Durch das Ausformulieren impliziter Annahmen und das Entfernen sprachlicher Zweideutigkeiten und vager Formulierungen gelangt man schrittweise zu einer in Prolog formulierbaren Beschreibung. Im Satz *Die Umstrukturierung verhindert die Gewinnmaximierung.* ist etwa nicht eindeutig, was wodurch verhindert wird.

Prolog hilft also, wie auch Logik allgemein, Dinge genauer zu analysieren und zu formulieren.

1.4 Imperative und deklarative Programmiersprachen

Prolog wird auch als **deklarative Programmiersprache** bezeichnet, um es **imperativen Programmiersprachen** gegenüberzustellen. Programme in herkömmlichen Programmiersprachen bestehen aus einer Folge von auszuführenden Befehlen (z.B. *Erhöhe x um eins „x := x+1“, Sende eine Nachricht an ein Objekt*). Herkömmliche Programmiersprachen werden daher **imperativ**, **befehlsorientiert** oder auch **präskriptiv** genannt. In einem imperativen Programm muss man genau angeben, durch welche Schritte man zu dem gewünschten Resultat gelangt. Ein typisches imperatives Programm stellt eine Rechenvorschrift —einen Algorithmus— dar.

In einer **deklarativen** Programmiersprache beschreibt man, *was* das Resultat sein soll. Man vernachlässigt dabei, welche Schritte durchgeführt werden müssen, um zum gewünschten Resultat zu gelangen. Die Details der Ausführung überlässt man der Programmiersprache. Üblicherweise besteht ein Programm in einer deklarativen Programmiersprache nicht aus einer Rechenvorschrift, sondern aus einer Beschreibung, die besagt, was gilt.

Durch den Umgang mit imperativen Programmiersprachen ist man gewohnt, vor allem den Ablauf eines Programms (das Wie) statt die Bedeutung (das Was) zu betrachten. Solange man zu sehr auf das Wie fixiert ist, fällt es schwer, Prologprogramme zu schreiben und zu verstehen. Es ist daher anfangs besonders wichtig, sich auf die Bedeutung eines Programms zu konzentrieren.

1.5 Programmieren in Prolog

Prolog ist nicht nur ein an die Logik angelehnter Formalismus, sondern auch eine universelle Programmiersprache, die zur Lösung von konkreten Problemen eingesetzt wird. Dadurch mussten bei der Entwicklung von Prolog Kompromisse eingegangen werden. Im Gegensatz zu automatischen Beweisern verwendet Prolog etwa nur einen sehr einfachen, dafür aber sehr schnellen Beweismechanismus. Beweise in Prolog sind oft vergleichbar schnell mit der Ausführung eines Programms, das in einer herkömmlichen (imperativen) Programmiersprache geschrieben ist. Die Geschwindigkeit der Ausführung wird aber mit der Unvollständigkeit von Prologs Beweisverfahren bezahlt. Es ist daher bis zu einem gewissen Grad erforderlich, beim Programmieren auf das spezielle Beweisverfahren Rücksicht zu nehmen.

Im nächsten Kapitel werden die Grundelemente von Prolog vorgestellt. Danach wird gezeigt, wie man allgemein Prologprogramme lesen, verstehen und schreiben kann.

2 Grundelemente

Prolog weist verglichen mit herkömmlichen Programmiersprachen eine sehr geringe Anzahl von Konstrukten auf. Ein Prolog-Programm besteht aus einer Folge von Horn-Klauseln. Eine **Klausel** ist entweder ein Faktum, eine Regel oder eine Anfrage. Fakten und Regeln beschreiben geltende Sachverhalte. Sie stellen die **Wissensbasis** des Programms dar. Anfragen ermöglichen, dieses Wissen zu verwenden. Im Programmtext wird jedes Faktum, jede Regel und jede Anfrage durch einen Punkt abgeschlossen.

2.1 Fakten

Fakten beschreiben einfache Beziehungen (Relationen) zwischen Termen.

```
kind_von(joseph_II, maria_theresia).
```

Dieses Faktum kann gelesen werden: *Joseph II ist ein Kind von Maria Theresia.* Der Name der Relation ist in diesem Fall kind_von. Die **Stelligkeit** der Relation (**Arität**, engl. ***arity***) ist zwei. Es handelt sich also eine Beziehung zwischen zwei Termen. Man bezeichnet diese Relation auch kurz mit dem **Prädikatsindikator** kind_von/2. Dabei ist die Reihenfolge der Argumente signifikant. Hier ist das erste Argument das Kind, das zweite Argument ein Elternteil des Kindes. Ein Name für ein konkretes Ding wird als **Konstante** (nullstelliges **Funktionssymbol**, auch **Atom**[1] genannt) dargestellt. Hier wurde Joseph II mit dem Atom joseph_II/0 und Maria Theresia mit maria_theresia/0 bezeichnet.

Namen von Relationen sowie Atome beginnen mit einem Kleinbuchstaben, gefolgt von einer beliebigen Anzahl von Buchstaben, Ziffern und Unterstrichen. Namen, die nicht von dieser Form sind, müssen mit einfachen Apostrophzeichen angeschrieben werden. Man hätte statt kind_von auch 'Kind von' verwenden können. Dabei sind für Prolog das untenstehende 'Kind von' und kind_von zwei völlig verschiedene Namen.

```
'Kind von'('Joseph II', 'Maria Theresia').
```

Fakten, die wie obiges Faktum keine Variable enthalten, werden **Grund-Fakten** genannt.

Sprechende Prädikatsnamen. Bei einfachen Beziehungen ist die Deutung als deutscher Satz meist sehr einfach. Gibt man den Relationen sprechende Namen, so sind die Beziehungen umso leichter verständlich. Die folgenden Schritte zeigen, wie man einen sprechenden Prädikatsnamen findet.

1. **Beschreibung der Argument-Typen.** Finden Sie für jedes einzelne Argument einen Begriff und setzen Sie diese Begriffe in der Reihenfolge der Argumente mit Unterstrichen zusammen. Wählen Sie anfangs ruhig zu weit gefasste Begriffe.

[1] Innerhalb der Vorlesung Mathematische Logik bezeichnete das Wort Atom eine Atomformel. Unter Atomen versteht man in Prolog Konstanten.

typ1_typ2_typ3_typ4(Arg1, Arg2, Arg3, Arg4)

Für die Relation „Kind einer Person“ könnte man mit person_person/2 (= eine Relation zwischen zwei Personen) beginnen.

2. **Verfeinerung der Namen.** Der in Schritt 1 gefundene Name wird meist zu weit gefasst sein. Die Relation person_person/2 könnte ebensogut Ehepartner wie Kinder und ihre Eltern beschreiben. Versuchen Sie daher, die Namen so zu verfeinern, dass alle nicht beabsichtigten Beziehungen ausgeschlossen werden. Statt person_person/2 ist kind_person/2 schon genauer. Weiters wird kind_person/2 zu kind_elternteil/2 verfeinert.

3. **Beziehung zwischen Argumenten.** Durch die bisherigen Schritte wurden die einzelnen Argumente nur dadurch beschrieben, was sie für sich allein genommen sind; nicht jedoch, wie sie zusammenhängen. Bei Namen wie person_geburtsdatum/2 ist es auch nicht unbedingt erforderlich, die Beziehung zwischen den Argumenten noch mehr zu betonen. Offensichtlich ist person_geburtsdatum/2 eine Relation zwischen einer Person und *deren* Geburtsdatum (und nicht einem anderen Geburtsdatum). Für den Fall, dass die Beziehung zwischen Argumenten betont werden soll, gehe man wie folgt vor.

   - **Präpositionen.** Der Name kind_elternteil/2 sagt noch nicht ganz genau, wie das Kind mit dem Elternteil zusammenhängt. Mit Präpositionen wie von, mit, durch beschreibt man die Beziehung genauer und verkürzt gelegentlich sogar den Namen. Statt kind_elternteil/2 verwende man kind_vonelternteil/2 oder einfacher kind_von/2.
   - **Partizipien.** Mit dem Partizip II wird beschrieben, wie ein Argument mit einem anderen zusammenhängt. Eine Relation zwischen einem (arithmetischen) Ausdruck und seiner Vereinfachung: ausdruck_vereinfachterausdruck/2 oder kurz: ausdruck_vereinfacht/2

     programm_optimiertesprogramm/2 oder kurz: programm_optimiert/2
   - **Verb in 3. Person.** Z.B. grenzt_an/2, besteht_aus/2

   Namen können aber auch zu stark verkürzt werden, sodass die Bedeutung der Relation nicht klar ersichtlich ist. Der Name länge_von/2 könnte die zeitliche Länge oder die räumliche Länge meinen. Auch ist hier nicht klar, von welchen Dingen die Länge beschrieben werden soll. Es ist vorteilhaft, dies schon durch den Namen klarzustellen. Also statt länge_von/2 verwende man länge_vongegenstand/2 oder länge_vonereignis/2.

   Weitere Beispiele: verheiratet_mit/2, verlegt_durch/2, zugelassen_bis/2, abhängig_von/2, inkompatibel_mit/2, entsorgt_durch/2, ungültig_ab/2.

4. **Abkürzungen.** Ist der Name schlussendlich zu lang, was vor allem bei Prädikaten mit hoher Stelligkeit vorkommt, kürzt man den Namen am Ende ab. Man beachte jedoch, dass Prädikate mit hoher Stelligkeit selten sinnvoll sind.

Um Tatsachen über Länder zu beschreiben, könnte man etwa folgende Relation verwenden:

country_region_latitude_longitude_area_population_capital_currency/8

Der Name beschreibt zwar die Bedeutung der einzelnen Argumente, ist allerdings sehr lang. Man kürze einen solchen Namen an geeigneter Stelle. Etwa country_region_/8 oder einfach country_/8. Zur Erinnerung, dass diese Namen nur Abkürzungen sind, lasse man am Ende des Namens den Unterstrich stehen.

Das folgende etwas umfangreichere Faktum country_/8 beschreibt einige Daten über Österreich. (In der Übungsumgebung finden Sie country_/8 bereits vordefiniert. Es beschreibt sämtliche Länder der Erde um 1980.) Das erste Argument ist der Name des Landes, die weiteren Argumente enthalten verschiedene Angaben über das Land. Die Bedeutung der weiteren Argumente ist hier nicht direkt ersichtlich. Man gibt die Bedeutung der Argumente wie folgt als **Kommentar** an:

```
% country_(Country, Region, Latitude,Longitude, Area, Population, Capital, Currency)
country_(austria, western_europe, 47, -14, 83803, 7520000, vienna, schilling).
```

Dieses Faktum kann gelesen werden: *Österreich ist ein Teil Westeuropas, befindet sich auf 47 Grad geographischer Breite und -14 Grad geographischer Länge (also östlich), ist 83803 km^2 groß, hat 7,52 Millionen Einwohner, als Hauptstadt Wien und als Währung den Schilling.*

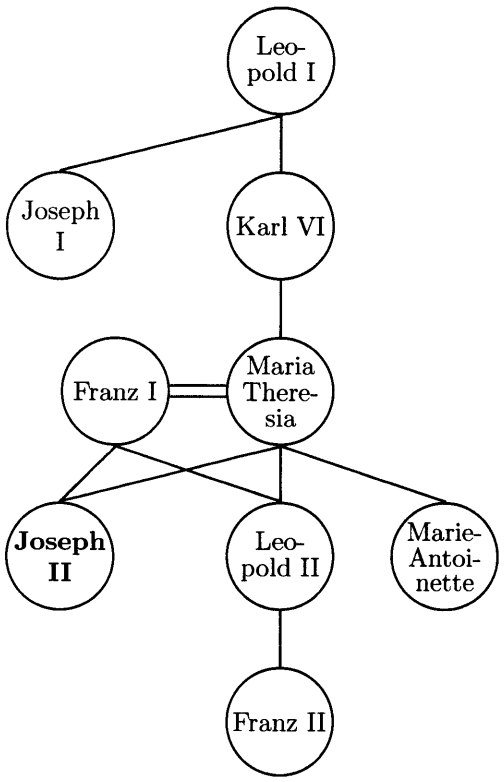
Mehrere Fakten gleichen Namens und gleicher Arität bilden ein einfaches Prädikat. Ein einfaches Prolog-Programm besteht aus einer Menge solcher Prädikatsdefinitionen.

Das Bild auf Seite 10 zeigt Verwandtschaftsverhältnisse des österreichischen Kaiserhauses im 17. und 18. Jahrhundert. Das Programm besteht aus den Prädikaten kind_von/2, gatte_gattin/2, männlich/1 und weiblich/1. Der Graph wird mit den Fakten kind_von/2 und gatte_gattin/2 dargestellt. Alle vier Prädikate bestehen ausschließlich aus Grund-Fakten. Ein derartiges Programm nennt man auch **Datenbasis**.

2.2 Anfragen, Zusicherungen

Einfache Anfragen. Um zu erfahren, ob ein Sachverhalt gilt, wird eine **Anfrage** der Form ← Ziel. in das Programm geschrieben. (Der Pfeil ← wird wie :- geschrieben.) Eine Anfrage ← Ziel. kann auf deutsch als *Ist es wahr/ableitbar, dass* Ziel *gilt?* gelesen werden. Bei jedem Absichern des Programms wird überprüft, ob das **Ziel** der Anfrage ableitbar ist. Ist es nicht ableitbar, wird dies als Fehler gemeldet. Wir nennen deshalb Anfragen auch **Zusicherungen**. Mit Zusicherungen wird also ein Programm getestet.

Es können auch mehrere unabhängige Anfragen zugleich angegeben werden. Folgend etwa werden vier verschiedene Anfragen angegeben. Alle vier Anfragen werden auf ihre Ableitbarkeit hin überprüft.




```
kind_von(joseph_I, leopold_I).
kind_von(karl_VI, leopold_I).
kind_von(maria_theresia, karl_VI).
kind_von(joseph_II, maria_theresia).
kind_von(joseph_II, franz_I).
kind_von(leopold_II, maria_theresia).
kind_von(leopold_II, franz_I).
kind_von(marie_antoinette, maria_theresia).
kind_von(franz_II, leopold_II).

männlich(franz_I).
männlich(franz_II).
männlich(joseph_I).
männlich(joseph_II).
männlich(karl_VI).
männlich(leopold_I).
männlich(leopold_II).

weiblich(maria_theresia).
weiblich(marie_antoinette).

gatte_gattin(franz_I, maria_theresia).
```

Abbildung 1: Das Haus Habsburg im 17. u. 18. Jhdt.

```
← kind_von(joseph_II, maria_theresia).
← kind_von(leopold_I, ferdinand_III). % Fehler
← kind_von(joseph_II, friedrich_II). % Fehler
← kind_von(maria_theresia, joseph_II). % Fehler
```

Das Ziel der ersten Anfrage (*Ist Joseph II ein Kind von Maria Theresia?*) ist beweisbar. Die Anfrage ist somit **erfolgreich**. Wie man aus der Definition des Prädikats kind_von/2 im Bild auf Seite 10 ersieht, wurde dies ausdrücklich so definiert. Die drei anderen Ziele können jedoch nicht bewiesen werden. Es finden sich keine passenden Fakten in der Datenbasis. Diese Ziele **scheitern**. Es kann also aus dem gegebenen Programm weder hergeleitet werden, dass Leopold I Kind Ferdinands III ist, noch, dass Joseph II Kind von Friedrich II ist, noch, dass Maria Theresia ein Kind von Joseph II ist.

Inhaltlich sind die Gründe für das Scheitern dieser Anfragen völlig verschieden. Während Leopold I in Wirklichkeit ein Kind Ferdinands ist, das Programm also nur einen **unvollständigen** Ausschnitt der Wirklichkeit darstellt, so sind die beiden anderen Anfragen sicher nicht der Fall. Joseph II ist sicher kein Kind Friedrichs. Wie aus kind_von/2 zu ersehen ist, ist der Vater Josephs II Franz I. Maria Theresia ist kein Kind von Joseph II, weil sie ja die Mutter Josephs ist.

Scheitert eine Anfrage, kann ein Ziel nicht bewiesen werden, so kann dies bedeuten:

1. Das Programm beschreibt die Wirklichkeit nur unvollständig. Das Gegenteil der gescheiterten Anfrage ist deshalb nicht notwendigerweise wahr. Das Programm „weiß" nicht, ob die Anfrage gilt oder nicht.

2. Das Programm beschreibt den Teil der Wirklichkeit, der für diese Anfrage von Interesse ist, vollständig. Das Gegenteil der gescheiterten Anfrage gilt. Die Anfrage stimmt nicht. (***negation as finite failure***).

Mittels Horn-Klauseln kann man die beiden Fälle nicht voneinander unterscheiden. Zudem ist es nicht möglich, negatives Wissen direkt mit Horn-Klauseln darzustellen. Bei der Behandlung von negativem Wissen muss daher mit besonderer Vorsicht vorgegangen werden. Eine nicht ableitbare Aussage bedeutet nicht unbedingt eine falsche Aussage.

Um zuzusichern, dass eine Anfrage scheitert, wird statt $\leftarrow$ das Symbol $\not\leftarrow$ (`:/-` „ist nicht ableitbar") der Anfrage vorangestellt. Die folgende **negative Zusicherung** überprüft also, ob Joseph II aufgrund des momentanen Wissens kein Kind Friedrichs ist.

$\not\leftarrow$ kind_von(joseph_II, friedrich_II).

Eine negative Zusicherung überprüft nur, ob das gegebene Ziel im bestehenden Programm tatsächlich nicht ableitbar ist. Durch sie wird das Programm jedoch nicht durch negatives Wissen erweitert. Die Bedeutung des Programms wird in keiner Weise verändert.

Allgemeine Anfragen mit Variablen. **Variablen** ermöglichen, allgemeinere Anfragen zu stellen. Statt die Argumente eines Ziels mit konkreten Namen zu belegen, „lässt man Argumente frei". Variablen, die in einer Anfrage vorkommen, sind existentiell quantifiziert. Man fragt, *Gibt es eine* ***Antwortsubstitution*** *für diese Variablen?* Als Antwort erhält man nun **Variablenbindungen**. Variablennamen beginnen mit einem Großbuchstaben gefolgt von einer beliebigen Anzahl von Buchstaben, Ziffern und Unterstrichen.

$\leftarrow$ kind_von(Kind, maria_theresia).

a) *Ist es wahr, dass* $\leftarrow$ kind_von(Kind, maria_theresia). *der Fall ist?*

Hat Maria Theresia (zumindest) ein Kind?

b) *Welches Kind/Welche Kinder hat Maria Theresia?*

Wer ist Kind von Maria Theresia?

Interaktive Anfragen. In der Übungsumgebung werden Anfragen üblicherweise getestet (a). Will man die konkreten Bindungen einer Anfrage sehen (b), so positioniert man den Cursor auf den : (Doppelpunkt) des Pfeils und drückt die mittlere Maustaste. Als Antwort auf diese Fragen erhalten wir eine Folge von Bindungen der Variable Kind.

← kind_von(Kind, maria_theresia).
@@ % Kind = joseph_II.
@@ % Kind = leopold_II.
@@ % Kind = marie_antoinette.
@@ % 3 Lösungen gefunden

Die Antwort lautet also: *Sowohl Joseph II, Leopold II als auch Marie-Antoinette sind Kinder Maria Theresiens.*

Anonyme Variablen. Argumente, die nicht von Interesse sind, werden durch **anonyme Variablen** dargestellt. Die Bindungen an anonyme Variablen scheinen nicht in der Antwort einer Anfrage auf. Die Variablen können also *irgendwelche* Werte annehmen.

← kind_von(_, _).

Gibt es irgendein Kind von irgend jemandem?

Mehrere Vorkommnisse einer anonymen Variable stellen jeweils verschiedene Variablen dar, während das mehrfache Vorkommen einer gewöhnlichen Variable die Argumente gleichsetzt. Das Ziel der folgenden negativen Zusicherung gilt etwa nicht.

⊬ kind_von(Kind, Kind).

Es ist nicht ableitbar: *Jemand ist Kind von sich selbst.*

Negative Anfragen werden verwendet, um **inkonsistente Daten** zu erkennen (***integrity constraints***). So ist es z.B. sicher nicht sinnvoll, dass eine Person ihr eigenes Kind ist.

Zur besseren Lesbarkeit von Programmen hebt man Variablen, die nur ein einziges Mal in einer Klausel vorkommen und daher einer anonymen Variable entsprechen, mit einem beginnenden Unterstrich hervor. Man kann dadurch auch solchen Variablen einen Namen geben und die Bedeutung des Arguments dokumentieren.

Zusammengesetzte Anfragen. Mehrere Ziele können in einer Anfrage konjunktiv verknüpft werden. Damit eine Anfrage, die aus einer **Konjunktion** von zwei Zielen besteht, erfüllt ist, müssen beide Ziele ableitbar sein.

← kind_von(joseph_II, maria_theresia), weiblich(maria_theresia).

Ist Joseph II ein Kind von Maria Theresia und ist Maria Theresia weiblich?

Gemeinsame Variablen. Wenn Variablen zugleich in mehreren Zielen einer Konjunktion vorkommen, werden die Argumente, in denen diese Variablen vorkommen, gleichgesetzt. Die obige Anfrage lässt sich etwa verallgemeinern, indem das Atom maria_theresia/0 durch die Variable Mutter ersetzt wird. Gemeinsamer Variablen (engl. ***shared variables***) können im Deutschen oft mit Relativsätzen umschrieben werden. Mittels eines Genitivs lassen sich solche Sätze gelegentlich noch weiter verkürzen.

← kind_von(joseph_II, Mutter), weiblich(Mutter).

Wer ist die Frau, deren Kind Joseph II ist? Wer ist die Mutter Josephs II?

Gelegentlich werden gemeinsame Variablen auch mit Wörtern wie *beide*, *alle drei* oder *gemeinsam* im Deutschen umschrieben.

← kind_von(joseph_II, Elternteil), kind_von(leopold_II, Elternteil).

Wer sind die gemeinsamen Eltern von Joseph II und Leopold II?

2.3 Regeln

Um die Relation zwischen Müttern und ihren Kindern zu beschreiben, könnten wir die folgende zusammengesetzte Anfrage stellen.

← kind_von(Kind, Mutter), weiblich(Mutter).

Wer ist Mutter welchen Kindes?

Welches Kind hat welche Mutter?

Um diese zusammengesetzte Anfrage leichter verwenden zu können, fasst man sie in einer **Regel** zusammen. Dabei werden die einzelnen Ziele —kind_von/2 und weiblich/1— eingerückt direkt untereinander angeschrieben:

```
..... ← % Schlußfolgerung wird später eingesetzt
   kind_von(Kind, Person), % ein Kind einer Person
   weiblich(Person). % eine weibliche Person
```

Das, was aus diesen beiden Zielen folgen soll, wird über den Zielen angegeben. So können wir aus der Tatsache, dass eine weibliche Person ein Kind hat, folgern, dass diese weibliche Person die Mutter des Kindes ist. Diese Schlussfolgerung gilt nur unter der Voraussetzung, dass die angegebenen Ziele gelten. Informell können wir eine Regel daher wie folgt lesen.

```
mutter_von(Person, Kind) ←
   kind_von(Kind, Person),
   weiblich(Person).
```

Wenn eine Person weiblich ist und diese Person ein Kind hat, (dann, so) ist diese Person die Mutter des Kindes.

Eine weibliche Person hat ein Kind → Diese Person ist die Mutter des Kindes.

Der Aufbau einer Regel sieht also wie folgt aus: Eine Regel beginnt mit dem Schluss im **Regelkopf**, der durch einen Folgerungspfeil, das **Regelatom** (←, engl. ***rule atom***) gekennzeichnet ist. Danach folgt der **Regelkörper**, (engl. ***body***). Der Regelkörper besteht aus einer Folge von Zielen.

```
mutter_von(Mutter, Kind) ← % Regelkopf = Schlußfolgerung
   kind_von(Kind, Mutter), % Regelrumpf = Voraussetzungen
   weiblich(Mutter).
```

Eine Regel ist dann ableitbar, wenn sämtliche Ziele ableitbar sind. Regeln werden verwendet, um allgemeinere Beziehungen zu definieren. Konkrete Namen wie franz_I kommen in ihnen selten vor. Während Prädikate, die nur aus Grund-Fakten bestehen, eher spezielle Beziehungen, etwa über konkrete Personen, beschreiben, definieren Prädikate, die Regeln enthalten, allgemeine von konkreten Dingen unabhängige Beziehungen. So hängt das nachfolgende Prädikat mutter_von/2 in keiner Weise von konkreten Daten des Prädikats kind_von/2 ab.

```
mutter_von(Mutter, Kind) ←
   kind_von(Kind, Mutter),
   weiblich(Mutter).
```

Man ist Mutter eines Kindes, wenn man weiblich und ein Elternteil des Kindes ist.

Lösungsmenge/Lösungssequenz. Im Gegensatz zu dem in der Logik-Programmierung und im automatischen Beweisen definierten Begriff der **Lösungsmenge**, produziert Prolog nur eine **Lösungssequenz**. Das heißt, dass gelegentlich **redundante Lösungen** von Prolog generiert werden. Das System zeigt sich wiederholende Lösungen eigens an. Redundante Lösungen sind aber nur dann störend, wenn sie unendlich oft in der Lösungssequenz auftreten und dadurch andere Lösungen verdecken. Redundante Lösungen entstehen meist durch Variablen, die nur im Regelrumpf vorkommen. Diese Variablen nennt man auch **existentielle** oder **interne Variable**.

```
vater(Vater) ←                    ← vater(Vater).
   männlich(Vater),               @@ % Vater = franz_I.
   kind_von(_Kind, Vater).        @@ % Vater = franz_I. % Redundant
                                  @@ % Vater = karl_VI.
                                  @@ % Vater = leopold_I.
                                  @@ % Vater = leopold_I. % Redundant
                                  @@ % Vater = leopold_II.
```

Redundante Lösungen können weiters entstehen, wenn in einem Prädikat verschiedene Regeln gleiche Lösungen besitzen. Das einfachste Beispiel ist das Prädikat wahr2/0.

```
wahr2.
wahr2.
```

Rekursive Regeln. Definitionen von Prädikaten können auch rekursiv sein. Gerade in diesem Fall empfiehlt es sich *in der Richtung des Regelpfeils* vorzugehen. Man betrachte also zuerst den Regelkörper, und betrachte erst dann den Regelkopf. Bei rekursiven Definitionen achte man weiters, ob nicht auch eine einfache, nicht rekursive Definition möglich ist. Ein Beispiel für eine unnötigerweise rekursive Definition ist etwa folgender Versuch, das Prädikat verheiratet_mit/2 mittels des Prädikats gatte_gattin/2 zu definieren. Während beim Prädikat gatte_gattin/2 das erste Argument stets der Mann ist, ist bei verheiratet_mit/2 die Reihenfolge nicht signifikant. Gilt etwa ← gatte_gattin(adam, eva). , so soll für das neue Prädikat sowohl ← verheiratet_mit(adam, eva). , als auch ← verheiratet_mit(eva, adam). gelten.

```
verheiratet_mit(Gatte, Gattin) ←
   gatte_gattin(Gatte, Gattin).
verheiratet_mit(PersonA, PersonB) ← % rekursive Regel
   verheiratet_mit(PersonB, PersonA).
```

Alternativ dazu ist in diesem Fall eine nichtrekursive Definition möglich:

```
verheiratet_mit(Gatte, Gattin) ←
   gatte_gattin(Gatte, Gattin).
verheiratet_mit(Gattin, Gatte) ←
   gatte_gattin(Gatte, Gattin).
```

Die folgenden rekursiven Prädikate vorfahre_von/2 und vorfahre_von_2/2 beschreiben jeweils alle Vorfahren einer Person. Sie werden im Kapitel über die Termination von Programmen genauer besprochen.

```
vorfahre_von(Vorfahre, Nachfahre) ←
   kind_von(Nachfahre, Vorfahre).
vorfahre_von(Vorfahre, Nachfahre) ←
   kind_von(Person, Vorfahre),
   vorfahre_von(Person, Nachfahre).
```

```
vorfahre_von_2(Vorfahre, Nachfahre) ←
   kind_von(Nachfahre, Vorfahre).
vorfahre_von_2(Vorfahre, Nachfahre) ←
   vorfahre_von_2(Person, Nachfahre),
   kind_von(Person, Vorfahre).
```

3 Lesen von Programmen

Grenzen des informellen Lesens. Um die Bedeutung eines Prädikats zu verstehen, kann man es als deutschen Satz lesen. Bei komplexeren Prädikaten wird das jedoch sehr umständlich. Das gesamte Prädikat vorfahre_von/2 als deutscher Satz:

> *Jemand ist Vorfahre eines Nachfahren, wenn der Nachfahre sein Kind ist,* **oder** *wenn er eine Person als Kind hat, die Vorfahre des Nachfahren ist.*
> (Einfacher, aber weniger eindeutig: *Eltern* **und** *ihre Vorfahren sind Vorfahren.*)

Der erste Satz ist hier bereits länger als das entsprechende Prologprädikat, weil man im Deutschen nur umständlich und nicht völlig eindeutig auf mehrere Referenten bezugnehmen kann. Während im Prologprädikat Variablen verwendet werden, deren Gültigkeit sich auf genau eine Klausel erstrecken, muss man im Deutschen verschiedenste Konstrukte einsetzen (z.B. er, der, dieser, jener, erstgenannter, letzterer, obiger), um sich auf bereits erwähnte Dinge oder Personen zu beziehen.

Um die Bezüge in unserem Beispiel genauer zu sehen, unterstreichen wir sie wie folgt. Wörter, die eine Person zum erstenmal erwähnen, sind unterstrichen. Wörter, die sich auf eine bereits erwähnte Person beziehen, sind doppelt unterstrichen. Man sieht, dass das Wort „Vorfahre“ zwei verschiedene Personen bezeichnet. (Den „Jemand“ und die „Person“. Um den Satz eindeutiger zu machen, könnte man noch weitere Bemerkungen (in Klammern) einfügen:

> *Jemand$_1$ ist Vorfahre$_1$ eines Nachfahren$_2$,*
>
> 1. *wenn der Nachfahre$_2$ sein$_1$ (des Vorfahren) Kind ist,*
> 2. **oder** *wenn er$_1$ (der Vorfahre) eine Person$_3$ als Kind hat, die Vorfahre$_3$ des (anfänglich erwähnten) Nachfahren$_2$ ist.*

Deutsche Texte, die eindeutig formuliert sein müssen (Vertragstexte oder mathematische Texte), enthalten daher Vorbemerkungen, die die Bedeutung von Wörtern einschränken. („Herr Müller, im weiteren Verkäufer genannt, ...“, „Seien x und y die Koordinaten ...“.) Derartige Texte sind meist sehr holprig zu lesen und nur für Spezialisten gedacht.

Lesen größerer Programmteile. Da man ein Programm oder Prädikat nur schwer auf einmal lesen und verstehen kann, werden wir uns darauf beschränken, nur einen Teil zu betrachten. Die restlichen Teile werden wir vorerst vernachlässigen. Sie werden durchgestrichen (mit ≡). Nun ist es möglich, ein Prädikat Schritt für Schritt zu lesen, ohne umständliche und kaum verständliche Formulierungen in Kauf nehmen zu müssen.

Prologs Anfragen werden von einer so genannten **abstrakten (Prolog-) Maschine** ausgeführt. Um Prolog als Programmiersprache gut zu beherrschen, sind detaillierte

Kenntnisse des Ablaufs dieser Maschine jedoch *eher hinderlich.* Nur besonders einfache Programme können auf diese Weise verstanden werden. Versucht man, ein Prolog-Programm auf der Ebene der Prolog-Maschine zu verstehen, versucht man sich also den genauen operationalen Ablauf eines Programms zu vergegenwärtigen, verliert man die Übersicht über das Programm aufgrund der Fülle von Details. Beim Verbessern eines Programms achtet man so kaum auf die Bedeutung des Programms, sondern versucht ausschließlich den Programmablauf zu verstehen und zu verbessern. Der Programmablauf für eine konkrete Anfrage gibt jedoch nicht Aufschluss über die Richtigkeit eines Programms für *alle möglichen* Anfragen. Mittels Anfragen (Zusicherungen) wird zwar beispielhaft überprüft, ob eine Definition eines Prädikats bzw. einer Relation auch mit der tatsächlichen Intention übereinstimmt, allgemein kann man aber nur durch die genaue Analyse des Programms Gewissheit erlangen, ob das Programm mit der eigenen Intention übereinstimmt.

Aus diesen Gründen werden wir vermeiden, genau nachzuvollziehen, wie Prolog eine Anfrage beweist. Viel zielführender ist es, sich die Bedeutung eines Programms zu vergegenwärtigen. Gewisse kleinere operationale Details (Endlosableitungen) werden wir in ähnlicher Weise analysieren.

Die folgenden Fragen werden beim Lesen eines Programms berücksichtigt werden:

1. Was beschreibt das Programm? (Deklarative Lesart)
2. Wie wird das Programm ausgeführt? (Prozedurale Lesart)
3. Terminiert das Programm?
4. Wie effizient läuft das Programm ab?

3.1 Deklarative Lesart von Prädikaten

Da ein Prolog-Programm logischen Formeln entspricht, können wir das Programm wie logische Formeln analysieren. Wir können uns beim Lesen eines Prädikats von der Richtigkeit des Prädikats überzeugen und allfällige Fehler in der Definition des Prädikats auffinden. Stimmt die Definition eines Prädikat nicht mit unserer Intention überein, verwende man die folgenden Methoden, um den Fehler zu finden. Wir werden nicht versuchen, die gesamte Definition eines Prädikats „auf einmal" zu verstehen, sondern werden Teile kurzfristig vernachlässigen (mit $\equiv$ gekennzeichnet), indem wir die Definition des Prädikats spezialisieren oder verallgemeinern. So reduzieren sich auf den ersten Blick unübersichtlich erscheinende Prädikate auf verständlichere Einheiten.

3.1.1 Analyse eines Prädikats

Spezialisierungen. Ein Prädikat besteht aus einer Folge von Klauseln. Jede einzelne Klausel beschreibt einen Teil der Lösungsmenge des Prädikats. Wir beschränken uns vorerst darauf, jede Klausel für sich allein zu verstehen, indem wir in den restlichen Klauseln das Ziel false/0 hinzufügen und sie dadurch vernachlässigen. Die verbleibende Definition beschreibt nun nur einen Teil der Lösungsmenge, ist also eine Spezialisierung.

```
vorfahre_von(Vorfahre, Nachfahre) ←
   kind_von(Nachfahre, Vorfahre).
vorfahre_von(Vorfahre, Nachfahre) ←
   kind_von(Person, Vorfahre),
   vorfahre_von(Person, Nachfahre).
```

Vernachlässigt man die zweite Klausel, so gilt: *Derjenige ist auf alle Fälle ein Vorfahre eines Nachfahren, der den Nachfahren als Kind hat. (Aber nicht unbedingt nur solche sind Nachfahren, weil es ja noch eine andere Klausel gibt.)* Oder einfacher: *Jedenfalls Eltern sind Vorfahren.*

```
vorfahre_von(Vorfahre, Nachfahre) ←
   kind_von(Nachfahre, Vorfahre).
~~vorfahre_von(Vorfahre, Nachfahre) ←~~ false,% durch false wird Klausel vernachlässigbar
   ~~kind_von(Person, Vorfahre),~~
   ~~vorfahre_von(Person, Nachfahre).~~
```

Vernachlässigt man die erste Klausel, so gilt: *Vorfahre eines Nachfahren ist auf alle Fälle jener, dessen Kind auch Vorfahre des Nachfahren ist. (Was auch immer sonst ...)* Oder wieder etwas verständlicher: *Jedenfalls Eltern von Vorfahren sind Vorfahren.*

```
~~vorfahre_von(Vorfahre, Nachfahre) ←~~ false,
   ~~kind_von(Nachfahre, Vorfahre).~~
vorfahre_von(Vorfahre, Nachfahre) ←
   kind_von(Person, Vorfahre),
   vorfahre_von(Person, Nachfahre).
```

In der folgenden fehlerhaften Definition vorfahre_von_zu_allgemein/2, wurden die Variablen der ersten Klausel irrtümlicherweise vertauscht. Die erste Regel allein sagt nun aus: *Kinder sind Vorfahren ihrer Eltern.* Unabhängig von der zweiten Regel ist die erste Regel bereits falsch, da durch sie bereits Dinge beschrieben werden, die nicht der Fall sein sollen. Die eigenen Kinder sind ja nicht die Vorfahren einer Person. Um einzusehen, dass vorfahre_von_zu_allgemein/2 falsch definiert ist, genügt es demnach vollauf, nur die erste Regel zu betrachten. Die zweite Regel kann niemals den Fehler der ersten Regel „wiedergutmachen".

```
vorfahre_von_zu_allgemein(Vorfahre, Nachfahre) ←
   kind_von(Vorfahre, Nachfahre).
~~vorfahre_von_zu_allgemein(Vorfahre, Nachfahre) ←~~ false,
   ~~kind_von(Person, Vorfahre),~~
   ~~vorfahre_von_zu_allgemein(Person, Nachfahre).~~
```

3.1.2 Analyse einer Regel

Verallgemeinerungen. Jedes einzelne Ziel einer Klausel schränkt die Lösungsmenge des Prädikats ein. Durch Vernachlässigung eines Ziels (indem man * vor das Ziel setzt) erhält man eine verallgemeinerte Definition. Wenn diese verallgemeinerte Definition bereits zu definierende Dinge *nicht* umfasst, ist in dem noch sichtbaren Teil ein Fehler.

vater(Vater) ←
  * ~~männlich(Vater)~~,
  * ~~kind_von(_Kind, Vater)~~.

Die Regel des Prädikats vater/1 enthält im Kopf nur die freie Variable Vater. Die obige Variante ist also immer erfüllt. Hingegen ist folgende Definition von vater_zu_eingeschränkt/1 zu speziell, um Väter allgemein zu definieren. Man erkennt bereits durch das Zudecken sämtlicher Ziele, dass vater_zu_eingeschränkt/1 etwa nicht für franz_I gilt. Es ist also nicht nötig, die Definition genauer zu betrachten, weil der Kopf der Klausel bereits falsch definiert ist.

vater_zu_eingeschränkt(leopold_I) ←
  * ~~männlich(leopold_I)~~,
  * ~~kind_von(_Kind, leopold_I)~~.

vater(Vater) ←
  männlich(Vater),
  * ~~kind_von(_Kind, Vater)~~.

Die obige Variante von vater/1 kann gelesen werden: *Ein Vater ist auf alle Fälle männlich. (Aber nicht alle Männer sind unbedingt Väter.)* Oder: *Ein Vater ist zumindest männlich.* Die obige Definition ist eine Verallgemeinerung, weil alle männlichen Wesen und nicht nur Väter beschrieben werden.

vater(Vater) ←
  * ~~männlich(Vater)~~,
  kind_von(_Kind, Vater).

Um Vater zu sein, muss man auf alle Fälle ein Kind haben. In dieser Definition sind Eltern allgemein enthalten, nicht nur Väter.

Die folgende fehlerhafte Definition großmutter_von/2 ist zu speziell, da bereits die folgende Verallgemeinerung zu speziell (fehlerhaft) ist. Der Fehler lässt sich also durch Verallgemeinerung auf zwei Ziele eingrenzen. Welches der beiden Ziele nun fehlerhaft ist, kann nur aufgrund inhaltlicher Überlegungen entschieden werden.

großmutter_von(GM, E) ←
  weiblich(GM),
  * ~~kind_von(P, GM)~~,
  * ~~kind_von(E, P)~~,
  männlich(GM).

Schließendes Lesen. Man verdecke den Kopf einer Regel und frage sich, was alles aus den sichtbaren Zielen folgt. Statt den Regelkopf zuerst zu lesen, lese man also zuerst die Ziele des Regelkörpers. Man stelle sich nun einige Relationen vor, die aus diesen Zielen folgerbar sind. Oft kann man so erkennen, dass eine Relation vielleicht zu eingeschränkt gefasst ist, man also besser eine allgemeinere Relation verwenden sollte. So ist vater_von/2 eventuell vater/1 vorzuziehen.

```
% kindmitvater(Kind) ←
% vater_von(Vater,Kind) ←
% weltmitvätern ←
~~vater(Vater)~~ ←
    männlich(Vater),
    kind_von(_Kind, Vater).
```

3.1.3 Fehlersuche

Stimmt ein Prädikat nun nicht mit unserer Intention überein, ist die Definition entweder zu allgemein oder zu speziell. Man kann den fehlerhaften Teil mit folgenden Methoden eingrenzen. In der Übungsumgebung wird diese Eingrenzung automatisch durchgeführt.

1. Das Prädikat ist für eine Anfrage erfüllt, die scheitern sollte. Die Definition ist also zu allgemein. In diesem Fall ist zumindest eine Klausel zu allgemein definiert. Durch Einfügen von false/0-Zielen kann die Definition spezialisiert werden. Wenn nun die spezialisierte Definition noch immer zu allgemein ist, muss der Fehler sich in den verbleibenden (noch sichtbaren) Klauseln befinden.

2. Das Prädikat scheitert für eine Anfrage, die erfüllt sein sollte. Die Definition ist zu speziell. Durch das Zudecken von Zielen mit */1 wird die Definition verallgemeinert. Ist die verallgemeinerte Definition noch immer zu speziell, befindet sich der Fehler im sichtbaren Teil.

Eine ähnliche Vorgangsweise empfiehlt sich andererseits auch zum **Schreiben** von Prolog-Programmen: Man beginnt mit zu allgemeinen Definitionen von Regeln und verfeinert diese durch Hinzufügen weiterer Ziele.

3.2 Prozedurale Lesart eines Programms

Da, wie erwähnt, die Veranschaulichung des tatsächlichen Ablaufs eines Programms nur wenig zum Verständnis von Prolog beiträgt, betrachten wir eine an die deklarative Lesart angelehnte Methode, um prozedurale (operationale) Aspekte eines Programms zu analysieren. Um ein Ziel zu beweisen, selektiert Prolog eine Klausel desselben Prädikats und ersetzt das Ziel durch den Regelkörper der Klausel. Diese Ersetzung heißt **Inferenz**.

verschwistert_mit(Kind1, Kind2) ←
kind_von(Kind1, Elternteil),
kind_von(Kind2, Elternteil).

← verschwistert_mit(joseph_II, Person).

Das Ziel der Anfrage wird durch den Regelkörper der Klausel verschwistert_mit/2 ersetzt. Dabei werden entsprechende Variablen gleichgesetzt also gebunden.

← kind_von(joseph_II, Elternteil), kind_von(Person, Elternteil).

In der Folge wird dieses Verfahren mit dem am linken Rand stehenden Ziel fortgesetzt. Ein Ziel kann nur ersetzt werden, wenn es eine entsprechende Definition gibt, sodass die Argumente passen. Es werden also Variable gebunden, bzw. gleiche Atome verglichen. Bei längeren Ableitungen ist eine derartige Betrachtungsweise sehr aufwendig. Es ist daher sinnvoller, statt dieses Ersetzungsprozesses nur die zu ersetzenden Regeln zu betrachten. Wir betrachten also das Prädikat verschwistert_mit/2 wie folgt:

1. verschwistert_mit(Kind1, Kind2) ← % ⟸
   ~~kind_von(Kind1, Elternteil)~~,
   ~~kind_von(Kind2, Elternteil)~~.

2. verschwistert_mit(Kind1, Kind2) ←
   kind_von(Kind1, Elternteil), % ⟸
   ~~kind_von(Kind2, Elternteil)~~.

3. verschwistert_mit(Kind1, Kind2) ←
   kind_von(Kind1, Elternteil),
   kind_von(Kind2, Elternteil). % ⟸

Zuerst werden sämtliche Ziele verdeckt, und dann die Ziele schrittweise von oben nach unten freigelegt. Dabei ist vor allem das gerade freiwerdende Ziel von Interesse. Wir können für dieses Ziel die Variablenbindungen ablesen, mit denen es bewiesen wird. Anfangs ist nur der Kopf der Regel sichtbar. Im zweiten Schritt wird die Variable Kind1 mit dem ersten Ziel verbunden. Eine neue Variable Elternteil tritt auf. Die Variable Elternteil ist also zu diesem Zeitpunkt stets frei. Das erste Ziel kind_von/2 wird also *immer* eine freie Variable als zweites Argument haben. Im dritten Schritt wird Kind2 und Elternteil mit dem zweiten Ziel verbunden.

Durch eine derartige Betrachtungsweise des Prädikats ersehen wir schon einiges über die möglichen Bindungen der Variablen. Die Variable Kind2 ist etwa nur für das letzte Ziel interessant. Die Variable Elternteil kann „von außen“ —also seitens der Anfrage— nicht gesehen werden. Eine derartige Variable nennt man auch **existentielle** oder **interne Variable**.

3.3 Termination von Programmen

Da mit Prolog allgemeine Programme geschrieben werden können, ist es schon theoretisch unmöglich, dass sämtliche Programme terminieren. Prologs Beweisstrategie ist zudem (verglichen mit automatischen Beweisern) besonders einfach. Das folgende Prädikat infinit/0 ist die einfachste Definition, die zu einer **Endlosableitung** führt. Um ← infinit. zu beweisen, wird ← infinit. durch ← infinit. ersetzt. Es ist zwar direkt ersichtlich, dass es keine endliche Ableitung für das Ziel ← infinit. gibt, dennoch wird dies von Prolog nicht erkannt. Das Ziel ← infinit. führt zu einer **Endlosschleife**. In der Übungsumgebung wird die Ableitung dieses Ziels nach einiger Zeit unterbrochen. Eine Erklärung für die Fehlerursache zeigt in diesem Fall, dass das Ziel tatsächlich nicht terminiert. Es kann durch die Übungsumgebung allerdings im allgemeinen nicht erkannt werden, ob es sich um eine echte Endlosschleife oder nur um eine sehr lange Berechnung handelt. Endlosableitung können eigens gekennzeichnet werden. Mit $\overset{\infty}{\leftarrow}$ (`:-&`) wird ein Anfrage gekennzeichnet, die erfüllt sein sollte; mit $\not\overset{\infty}{\leftarrow}$ (`:/-&`) eine, die nicht erfüllt sein sollte, aber zu einer Endlosableitung führt. Für das Prädikat infinit/0 gibt es keine mögliche Ableitung, daher wurde $\not\overset{\infty}{\leftarrow}$ verwendet, während es für winfinit/0 eine Lösung gibt (das Faktum), die aber von Prolog nicht gefunden wird. Durch Zudecken der rekursiven Regel kann man ersehen, dass winfinit/0 eine Lösung besitzt.

```
infinit ←
   infinit.
← infinit.
@@ ! Ausführung dauert zu lang, Erklärung mit [Do] am Pfeil
∤∞← infinit. % nicht ableitbar, Endlosableitung

winfinit ←
   winfinit.
winfinit.
←∞ winfinit. % wahr, aber endlos
```

In den meisten Fällen treten Endlosableitungen nicht derart offensichtlich zutage. Oft finden sich Lösungen, obwohl das Ziel letztlich nicht terminiert. Man hat dadurch, dass Lösungen gefunden werden, den Eindruck, dass das Prädikat richtig definiert ist. Erst später, bei der Verwendung des Prädikats, kommt es zu Fehlern. Für das Prädikat hinfinit/0 kann eine Lösung gefunden werden, dennoch enthält es so wie winfinit/0 die rekursive Regel, die letztlich zu einer Endlosableitung führt. Diese Endlosableitung wird hier durch die (unendlich vielen und redundanten) Lösungen »verdeckt«. Um die Terminationseigenschaft eines Ziels deutlicher zu sehen, füge man eine Zusicherung ∤← Ziel, false. hinzu. Deklarativ betrachtet, muss diese Zusicherung falsch sein. Die einzige noch beobachtbare Eigenschaft ist die Termination bzw. Nichttermination des Ziels. Im Falle von hinfinit/0 terminiert diese Zusicherung nicht, daher wird der Pfeil $\not\overset{\infty}{\leftarrow}$ verwendet.

```
hinfinit.                                  ⁄∞← hinfinit, false. % sichert Nichttermination zu
hinfinit ←                                 ⁄← false, hinfinit. % nicht sinnvoll
   hinfinit.
← hinfinit. % Lösung
@@ % true.
@@ % true. % ↑↑Redundant
@@ % true. % ↑↑Redundant
...
```

Man beachte, dass die Reihenfolge der Ziele für Prologs Ausführungsmechanismus signifikant ist. Die Zusicherung ⁄← false, hinfinit. ist zwar logisch äquivalent mit ⁄∞← hinfinit, false. — sie ist genauso falsch — sie besagt aber nichts über die Terminationseigenschaft von hinfinit/0.

Zusicherung		deklarativ	prozedural
← Ziel.	`:-`	wahr	es findet sich (zumindest) eine Lösung
⁄← Ziel.	`:/-`	falsch	endliches Scheitern, Termination
⁄← Ziel, false.		(falsch)	Termination
∞← Ziel.	`:-&`	wahr	wegen Endlosableitung keine Lösung
⁄∞← Ziel.	`:/-&`	falsch	wegen Endlosableitung kein Scheitern
⁄∞← Ziel, false.		(falsch)	Ziel terminiert nicht
$← Ziel.	`:-$`	wahr	aufwendige Lösung
⁄$← Ziel.	`:/-$`	falsch	aufwendiges (aber endliches) Scheitern
⁄$← Ziel, false.		(falsch)	aufwendige Termination

Endlosableitungen werden durch (direkt oder indirekt) rekursive Regeln verursacht. Statt das gesamte Programm zu betrachten, beschränkt man sich nur auf jene Teile des Programms, die Rekursionen enthalten. Klauseln, die keine Ziele von rekursiven Prädikaten, bzw. von Prädikaten, die andere rekursive Prädikate verwenden, enthalten, werden nicht betrachtet. Beim Prädikat vorfahre_von_2/2 vernachlässigt man also die erste Regel, weil sie sicher nicht Teil einer Endlosableitung ist. In ihr wird ausschließlich das Prädikat kind_von/2, das ja nur aus Fakten besteht, verwendet.

```
~~vorfahre_von_2(Vorfahre, Nachfahre) ←~~ false,
   ~~kind_von(Nachfahre, Vorfahre)~~.
vorfahre_von_2(Vorfahre, Nachfahre) ←
   vorfahre_von_2(Person, Nachfahre),
   kind_von(Person, Vorfahre).
```

Ziele für das verbleibende Prädikat können niemals erfüllt sein. Uns interessiert jedoch nur, ob dieser übriggebliebene Teil stets terminiert (also scheitert), oder, ob Endlosrekursionen möglich sind. Dazu untersuchen wir den Regelkörper der verbleibenden Regel.

Beim Prädikat vorfahre_von_2/2 betrachtet man also die folgende Variante des Programms. Hier sieht man, dass die Variable Vorfahre nicht im ersten Ziel verwendet wird.

Die Variable Vorfahre kann daher keinen Einfluss darauf haben, ob Endlosableitungen auftreten oder nicht. Das zweite Argument von vorfahre_von_2/2, die Variable Nachfahre wird zwar vom Kopf zum ersten Ziel „durchgereicht". Aber auch hier treten keinerlei Bindungen, die die Endlosrekursion durch Scheitern aufhalten könnten, auf. Wir können daraus ersehen, dass unabhängig von den konkreten Argumenten eines Ziels für vorfahre_von_2/2 eine Endlosableitung auftreten wird.

vorfahre_von_2(Vorfahre, Nachfahre) ←
  vorfahre_von_2(Person, Nachfahre), **false,**
  ~~kind_von(Person, Vorfahre)~~.

Stellt man die Anfrage ← vorfahre_von_2(Vorfahre, Nachfahre)., so sieht man zwar anfangs sämtliche Lösungen, die auch ← vorfahre_von(Vorfahre, Nachfahre). liefert, bleibt dann jedoch in einer Endlosschleife stecken. Dieses Verhalten ist sehr heimtückisch, weil man oft geneigt ist, beim Erstellen eines Prädikats nur die ersten Lösungen zu betrachten. Man hat dadurch den Eindruck, dass das Prädikat „funktioniert". Die Endlosschleifen treten erst dann zutage, wenn man das Prädikat in einem anderen Programm verwendet. Aus diesem Grunde ist es sehr empfehlenswert, Programme nicht nur durch Beispiele zu testen, sondern wie vorgestellt auch analytisch zu betrachten. Das exemplarische **Testen von Programmen** kann nur Programmierfehler zutage bringen, nicht jedoch sicherstellen, dass keine Fehler im Programm vorhanden sind.

Zusätzlich zu den Überlegungen zur Termination am Papier sollen Sie auch Anfragen zum Testen der Termination eines Programmes angeben. Wenn ein Ziel niemals zu Endlosableitungen führt, terminiert (und scheitert) das Ziel ← Ziel, false. Das Prädikat false/0 ist ein vordefiniertes Prädikat, das falsch ist — also immer scheitert. Es erzwingt, dass die Anfrage nie erfüllt sein kann. Durch die negative Zusicherung ⊬ Ziel, false. wird also zugesichert, dass das Ziel stets terminiert.

Allgemein kann bei vielen Prädikaten nicht zugesichert werden, dass sie stets terminieren. Vor allem bei Prädikaten, die eine unendlich große Lösungsmenge besitzen, können wir gar keine Definition angeben, sodass die Anfrage ← Ziel, false. stets scheitert. In diesen Fällen sind Überlegungen erforderlich, wie die Argumente des Ziels aussehen müssen, damit es terminiert. Wir überlegen uns dies anhand der Definition von vorfahre_von/2.

~~vorfahre_von(Vorfahre, Nachfahre) ←~~ **false,**
  ~~kind_von(Nachfahre, Vorfahre)~~.
vorfahre_von(Vorfahre, Nachfahre) ←
  kind_von(Person, Vorfahre),
  vorfahre_von(Person, Nachfahre).

Hier sind, verglichen mit dem Prädikat vorfahre_von_2/2, die beiden Ziele der zweiten Regel vertauscht. Bevor die Rekursion betrachtet wird, wird das Ziel kind_von/2 bewiesen. Da dieses Ziel stets terminiert, terminiert sicher auch das entsprechende Fragment.

vorfahre_von(Vorfahre, Nachfahre) ←
  kind_von(Person, Vorfahre), **false,**
  ~~vorfahre_von(Person, Nachfahre)~~.

Diese Regel terminiert genau dann, wenn kind_von/2 einmal (nicht notwendigerweise bei der ersten Inferenz) scheitert. Ob vorfahre_von/2 terminiert, hängt damit von den Lösungen von kind_von/2 und der Variable Vorfahre ab. Die Termination hängt aber nicht von der Variable Nachfahre ab. Damit keine Endlosableitung auftritt, darf in der Regel Vorfahre niemals gleich Person sein. Das heißt, dass die untenstehende Klausel und damit das Ziel kind_von/2 stets scheitern muss. Andernfalls würde diese Regel nicht terminieren, weil damit ähnlich dem Prädikat infinit/0 stets das gleiche Ziel bewiesen werden würde. Im nachstehenden Programm wurde durch **Vorfahre = Person** ein Fall konstruiert, für den kind_von/2 scheitern muss, um Endlosableitungen in vorfahre_von/2 zu verhindern.

```
vorfahre_von(Vorfahre, Nachfahre) ←
    Vorfahre = Person,
    kind_von(Person, Vorfahre), % muss scheitern, sonst endlos
    vorfahre_von(Person, Nachfahre).
```

Bei der Relation kind_von/2 wären derartige zyklische Abhängigkeiten auch gar nicht sinnvoll. Weder kann man sein eigenes Kind sein, noch sein eigener Enkel etc. Durch negative Zusicherungen sichern wir zu, dass für kind_von/2 keine derartigen Zyklen erlaubt sind. Etwa: $\not\leftarrow$ kind_von(Kind, Kind).

Programme, die ausschließlich direkte Rekursionen (keine indirekten Rekursionen) enthalten, nennt man **stratifiziert**. Die meisten Prolog-Programme sind stratifiziert. In diesen Programmen sind Überlegungen zur Termination von Programmen besonders einfach, weil man jedes Prädikat für sich auf Endlosableitungen hin untersuchen kann.

Sinnvolle Terminationsannotationen. Betrachten wir eine Terminationsannotation der Form $\not\leftarrow$ Ziel, false. Wenn hier bereits Ziel alleine scheitert, ist das false/0 am Ende an sich überflüssig und evtl. irreführend. Die meisten interessanten Fälle von Termination treten auf, wenn Ziel allein (zumindest) eine Lösung findet. Für sinnvolle Terminationsannotationen muss ein Paar Zusicherungen für ein und dasselbe Ziel verwendet werden. Es finden sich also für Ziel Lösungen und das Ziel terminiert auch.

← Ziel.	← Ziel.
$\not\leftarrow$ Ziel, false.	$\not\not\leftarrow$ Ziel, false.

Ähnliche Überlegungen gelten für $\not\not\leftarrow$: Hier finden sich Lösungen, insgesamt terminiert das Ziel jedoch nicht. Die Ursache kann dafür entweder eine unendlich große Menge von erforderlichen Antwortsubstitutionen sein oder Probleme mit Prologs Terminationsverhalten. Im ersten Falle darf das Ziel gar nicht terminieren, da ja sonst nicht alle Lösungen beschrieben werden. Derartige Fälle lassen sich bereits vor der eigentlichen Definition des Prädikats abschätzen. Der zweite Fall tritt auf, wenn die Antwortsubstitutionen zwar endlich sind, Prolog aber aufgrund seines einfachen Beweismechanismus dies nicht in endlicher Zeit zeigen kann. Derartige Fälle können meist erst anhand einer konkreten Prädikatsdefinition erkannt werden.

3.4 Effizienzüberlegungen

Eine genaue Abschätzung des Ressourcenverbrauchs (Speicherplatz und Zeit) eines Beweises ist sehr aufwendig, da Prologs Ausführungsmechanismus verglichen mit anderen Programmiersprachen sehr komplex ist. Meist genügt es aber völlig, den Ressourcenverbrauch eines Ziels grob abzuschätzen, oder für konkrete Beispiele Messungen durchzuführen. Der entscheidende Faktor, der die Effizienz eines Prologprogramms beeinflusst, ist ohnehin die Repräsentation des Wissens.

Bevor Sie aber Überlegungen über die Effizienz eines Programms anstellen, sollten Sie auf alle Fälle sicherstellen, dass Ihr Programm überhaupt terminiert! Die Terminationsüberlegungen aus dem vorigen Abschnitt sollten in jedem Fall angestellt werden. Anfänglich macht man häufig den Fehler, nichtterminierende Programme zu optimieren.

Um den Aufwand für den Beweis eines Zieles abzuschätzen, verwende man folgende Maße: Größe der Lösungsmenge, Anzahl der Inferenzen und Termgröße.

3.4.1 Größe der Lösungsmenge

Durch jedes Ziel wird eine Lösungsmenge beschrieben. Je nach dem, welche Argumente eines Ziels bereits bekannt sind, wird die Lösungsmenge eingeschränkt. Je weniger Argumente eines Ziels bekannt sind, umso größer wird diese Lösungsmenge sein, umso aufwendiger ist ein Beweis.

3.4.2 Anzahl der Inferenzen

Um die Anzahl der erforderlichen Inferenzen zum Beweis eines Zieles abzuschätzen, gehen wir ähnlich wie bei Terminationsüberlegungen vor. Zusätzlich werden wir uns bei jedem Ziel überlegen, wie groß die Anzahl der möglichen Lösungen ist. Wir betrachten erneut das Prädikat vorfahre_von/2 und zwei Ziele.

```
← vorfahre_von(joseph_II, Nachfahre).
← vorfahre_von(Vorfahre, joseph_II).

vorfahre_von(Vorfahre, Nachfahre) ←
   kind_von(Nachfahre, Vorfahre).
vorfahre_von(Vorfahre, Nachfahre) ←
   kind_von(Person, Vorfahre),
   vorfahre_von(Person, Nachfahre).
```

Als grobe Annäherung vernachlässigen wir wiederum nichtrekursive Regeln. Das so erhaltene Prädikat wird natürlich scheitern, aber in etwa gleich viele Inferenzen benötigen, wie das ursprüngliche Programm.

```
~~vorfahre_von(Vorfahre, Nachfahre) ←~~ false,
   ~~kind_von(Nachfahre, Vorfahre).~~
vorfahre_von(Vorfahre, Nachfahre) ←
   kind_von(Person, Vorfahre),
   vorfahre_von(Person, Nachfahre).
```

Weiters können wir die Variable Nachfahre vernachlässigen. Sie kommt in der Regel nur im Kopf als Argument und im letzten rekursiven Ziel vor. Deshalb hat diese Variable keinen Einfluss auf die Anzahl der durchzuführenden Inferenzen.

~~vorfahre_von(Vorfahre, Nachfahre) ←~~ **false,**
~~kind_von(Nachfahre, Vorfahre).~~
vorfahre_von(Vorfahre, ~~Nachfahre~~) ←
kind_von(Person, Vorfahre), % ⇐
vorfahre_von(Person, ~~Nachfahre~~).

Beim Ziel ← vorfahre_von(joseph_II, Nachfahre). ist das erste Argument bekannt, dadurch wird auch das erste Ziel kind_von(Person, Vorfahre) in seiner Lösungsmenge eingeschränkt werden. Für kind_von/2 wird es so viele Lösungen geben, wie es Kinder von Joseph II gibt. Nach Beweis von kind_von/2 ist Person bekannt. Damit ist für die nächste Inferenz wiederum das erste Argument gebunden. Die Anzahl der benötigten Inferenzen hängt also davon ab, wieviele Nachfahren Joseph II hat.

Im Ziel ← vorfahre_von(Vorfahre, joseph_II). ist das zweite Argument bekannt. Dieses Argument wird in der Regel jedoch immer nur „durchgereicht". Es hat keinen Einfluss auf die Anzahl der Inferenzen von vorfahre_von/2. Die Argumente des Ziels kind_von/2 werden stets freie Variablen sein. Das bedeutet, dass die Lösungsmenge vom Ziel kind_von/2 so groß wie die gesamte Datenbasis sein wird. Ganz unabhängig davon, wer die Person in der Anfrage überhaupt ist. Nach Beweis des Ziels kind_von/2 ist Person bekannt. Die weiteren Inferenzen werden ähnlich dem Ziel ← vorfahre_von(joseph_II, Nachfahre). ablaufen.

Indirekt werden durch das Ziel ← vorfahre_von(Vorfahre, joseph_II). sämtliche Vorfahren sämtlicher Personen der Datenbasis berechnet werden. Dies ist bei einer größeren Datenbasis natürlich ein enormer Aufwand. In diesem Falle wäre es günstiger, die Regel vorfahre_von_3/2 zu verwenden. Im zweiten Teil des Skriptums werden einige Methoden vorgestellt, um derartige Probleme allgemein zu lösen.

vorfahre_von_3(Vorfahre, Nachfahre) ←
kind_von(Nachfahre, Person),
vorfahre_von_3(Vorfahre, Person).

3.4.3 Termgröße

Wir haben bisher gesehen, dass als Argumente von Prädikaten Konstanten (Atome) verwendet werden können. Komplexere Prädikate verwenden strukturierte Terme (siehe Kapitel 6, Seite 31). Die Ausführungszeit eines Ziels verhält sich oft in etwa proportional zur Größe der verwendeten Terme (unter der Voraussetzung, dass alles andere gleich bleibt).

4 Schreiben von Programmen

Im letzten Kapitel haben wir gesehen, wie man Prologprogramme lesen kann. Beim Schreiben eines Programms gehe man wie folgt vor, um zu verhindern, dass man ein Prologprogramm „wie ein C-Programm“ zu schreiben beginnt. Vor dem eigentlichen Kodieren sollten die folgenden Überlegungen angestellt werden.

4.1 Bestehende Beschreibungen

In vielen Bereichen gibt es bereits Notationen zur Beschreibung des nötigen Wissens. Oft wird bestehendes Wissen in Form von Tabellen (Fahrpläne, Börsenkurse) oder graphisch dargestellt. Familiäre Beziehungen werden seit langem in der im Bild auf Seite 10 gezeigten graphischen Notation beschrieben. Tabellarisch scheinen familiäre Beziehung in Taufregistern und Matrikeln auf. Komplexere Beschreibungen sind etwa Taxonomien zur Klassifikation von Lebewesen, Patenten, Büchern und Bauteilen. Häufig kann man sich darauf beschränken, die bereits bestehenden Notationen auf Prolog abzubilden.

4.2 Typen bestimmen

Durch einfache einstellige Prädikate beschreibe man die für den Bereich interessanten Daten. Diese Prädikate dienen meist nur der Dokumentation, sie werden in den eigentlichen Programmen gar nicht verwendet. Dennoch ist es empfehlenswert, derartige Dokumentationsprädikate zu schreiben. Um diese Prädikate von anderen zu unterscheiden, lässt man oft den eigentlichen Prädikatsnamen mit is_ oder ist_ beginnen. Später, wenn wir auch strukturierte Terme verwenden werden, werden derartige is_-Definitionen besonders hilfreich sein.

```
himmelsrichtung(nord).
himmelsrichtung(süd).
himmelsrichtung(west).
himmelsrichtung(ost).

ist_zeit(vergangenheit).
ist_zeit(gegenwart).
ist_zeit(zukunft).
```

```
is_operationtype(import).
is_operationtype(export).

is_caroption(air_conditioning).
is_caroption(power_steer).
is_caroption(airbag).
is_caroption(abs).
```

4.3 Auffinden von Relationen/Prädikaten

Versuchen Sie, Relationen zwischen konkreten Elementen der Typen, die im letzten Schritt gesammelt wurden, zu finden. Falls es keine bestehenden Notationen (wie Tabellen etc.) gibt, ist es oft schwer, solche Relationen zu finden. Ist der Bereich informell beschrieben, kann man versuchen, deutsche Sätze (oder Teile davon) auf Relationen abzubilden. So wie wir im vorigen Kapitel Prädikate als deutsche Texte gelesen haben, versuche man nun umgekehrt vorzugehen. Notfalls greife man willkürlich Typen heraus und versuche

zu überlegen, wie diese miteinander zusammenhängen könnten. (z.B. *Wie können zwei Personen miteinander zusammenhängen?*)

4.4 Prädikatsnamen bestimmen

Der Name eines Prädikats sollte Auskunft darüber geben, was beschrieben wird. Wie man einen sinnvollen Namen für ein Prädikat findet, steht auf Seite 7.

Anfangs verwendet man oft Namen wie in imperativen Programmiersprachen. Solche Namen sind aber irreführend. Programme in imperativen Programmiersprachen sind eine Folge von Befehlen. Prozedurnamen sind daher meist Befehle. Sie bestehen aus einem Verb im Imperativ. (z.B. redraw, load, find, open, copy, duplicate, merge, sort). In Prolog sind imperative Namen irreführend, weil wir mit einem einzigen Prädikat verschiedene Aufgaben bewältigen können. Das, was allen Verwendungsarten eines Prädikats gemeinsam ist, ist die Relation, die durch das Prädikat beschrieben wird. So kann Prädikat kind_von/2 für folgende Aufgaben verwenden.

```
← kind_von(Kind, maria_theresia). % Suche die Kinder Maria Theresiens.
← kind_von(joseph_II, Elternteil). % Suche die Eltern Josephs II.
← kind_von(joseph_II, maria_theresia). % Teste, ob JII Kind Maria Theresiens ist.
← kind_von(Kind, Elternteil). % Liste alle Kinder und ihre Eltern auf.
```

Das Prädikat kind_von/2 verwenden wir also zum Suchen, Testen und Auflisten. Um in einer imperativen Programmiersprache alle diese Aufgaben zu lösen, bräuchten wir jeweils eine eigene Prozedur.

4.5 Zusicherungen anschreiben

Schreiben Sie sowohl positive ← als auch negative ⊬ Zusicherungen an. Noch *bevor* Sie das Prädikat implementieren! Sie haben dadurch gleich einige Testfälle parat.

Die Sinnhaftigkeit Ihrer Zusicherungen kann zum Teil bereits zu diesem Zeitpunkt (also noch bevor das Prädikat definiert wird) beim Absichern überprüft werden. Derartige Fehlermeldungen werden durch != gekennzeichnet.

```
← kind_von(A,A).
!= Dies sollte eine negative Zusicherung sein
← kind_von(karl_VI,joseph_II).
```

Die erste Zusicherung besagt, dass es ein A geben soll, das Kind von sich selbst ist — eine derartige Person sollte es sicher nicht geben; was auch erkannt wird. Die zweite Zusicherung ist wahrscheinlich ebenso unsinnig, dies kann aber nicht erkannt werden, da die Relation hier unterspezifiziert ist.

4.6 Prädikat definieren

Erst nach all diesen Schritten ist es sinnvoll, ein Prädikat auch wirklich zu implementieren. Man hat nun bereits Testfälle zur Hand, die bei jedem Absichern des Programms automatisch getestet werden. Schreiben Sie also stets Zusicherungen (und sichern Sie diese), bevor Sie das Prädikat implementieren!

5 Negative Definitionen I

Häufig versucht man, statt einer positiven Definition eines Prädikats eine negative anzugeben. Statt Dinge direkt zu definieren, ist man versucht, Dinge als Negation anderer Dinge zu definieren. Eine positive Definition erscheint auf den ersten Blick aufwendiger als eine negative. Nehmen wir an, wir möchten Prädikate zur Klassifikation von Erdoberflächen definieren. Ein Prädikat meer/1, das alle Meere beschreibt, sei schon definiert. Da — umgangssprachlich— das Gegenteil vom Meer das Land ist, könnte man nun versucht sein, land/1 als „Nicht-Meer“ zu definieren. Häufig ist eine derartige Definition allerdings falsch, weil man den Begriffsumfang von Meer und Land nicht richtig eingeschätzt hat. Meist sind negative Definitionen zu weit gefasst. Es kann z.B. neben Meeren auch andere Oberflächen geben, die ebenfalls „Nicht-Länder“ sind. Weiters müssen sämtliche negative Definitionen bei einer Erweiterung des Programms auf ihre Richtigkeit überprüft werden. Anfangs könnten z.B. nur Meere und Länder definiert sein. Die Definition von land/1 als Nicht-Meer scheint korrekt zu sein. Fügt man jedoch zu einer derartigen Datenbasis eine neue Art von Erdoberfläche hinzu (z.B. See, Niemandsland etc.), müssen alle negativen Definitionen durchgesehen werden. Negative Definitionen bewirken also zumindest einen höheren Wartungsaufwand für das Programm als positive; Programme mit negativen Definitionen sind schlechter erweiterbar. Die folgenden einfachen Faustregeln helfen zu bestimmen, ob eine Definition, die negative Teile enthält, sinnvoll ist.

Formulieren Sie die Definition auf deutsch, wie:
Alles, wirklich alles, was kein Meer ist.

Versuchen Sie zu überprüfen, ob auch absurde, nicht im Kontext des Programms sinnvolle Namen, für die negative Definition erfüllt sind. (Das **nicht** im folgenden Beispiel ist kein in Prolog definiertes Prädikat, es dient nur zur Veranschaulichung).

```
land(Land) ←
   nicht meer(Land).
```

Die folgenden Ziele sind bei einer derartigen Definition von Land erfüllt.

← land(27). % 27 ist ein Land (???)
← land(sessel). % ...
← land(almwiese).
← land(prolog).
← land(donau).

Das Prädikat land/1 ist also bei der obigen informellen negativen Definition für eine viel zu große Menge erfüllbar.

6 Allgemeine Terme

6.1 Strukturen

Um das Geburtsdatum von Personen zu beschreiben, könnte man die folgenden Fakten definieren:

geboren_jahr_monat_tag(joseph_II, 1741, 3, 13).
geboren_jahr_monat_tag(maria_theresia, 1717, 5, 13).

Es ist übersichtlicher, zusammengehörende Argumente in einer **Struktur** zu gruppieren. Statt der drei zusätzlichen Argumente wird dadurch nur ein einziges Argument benötigt. Die folgenden Fakten verwenden die Struktur datum/3. datum/3 wird **Funktor** der Struktur datum(_,_,_) genannt.

geboren_am(joseph_II, datum(1741,3,13)).
geboren_am(maria_theresia, datum(1717,5,13)).

Die Verwendung der Fakten wird dadurch einfacher, weil sich die Stelligkeit eines Prädikats reduziert, sowie weniger Argumente benötigt werden und dadurch Ziele dieses Prädikats kompakter sind. Die Repräsentation des Datums mittels der Struktur datum/3 kann auch leichter verändert werden, als eine Repräsentation ohne Strukturen. Das folgende Prädikat zugleichgeboren_mit/2 verwendet zwar das Datum, um die Geburtsdaten zweier Personen gleichzusetzen, dennoch wird die Struktur datum/3 in diesem Prädikat nicht direkt verwendet. Ändert man im Nachhinein datum/3 auf datum/4, um etwa auch die Geburtsstunde zu erfassen, müsste die Definition dieses Prädikats nicht verändert werden.

```
zugleichgeboren_mit(PersonA, PersonB) ←
    dif(PersonA, PersonB),
    geboren_am(PersonA, Datum),
    geboren_am(PersonB, Datum).
```

Funktionssymbole und Prädikatensymbole sehen in Prolog gleich aus. Sie unterscheiden sich nur durch ihr Vorkommen. Während der Kopf einer Klausel oder ein Ziel ein **Prädikatensymbol** ist, enthalten die Argumente eines Prädikatensymbols Terme. Im Prädikat geboren_am/2 ist etwa geboren_am/2 ein Prädikatensymbol, während datum/3 ein Funktionssymbol ist.

6.2 Unifikation

Um zwei allgemeine Terme „gleichzusetzen“ werden diese Terme miteinander unifiziert. Allgemeine Unifikation in Prolog lässt sich durch ein einfaches Faktum beschreiben:

```
gleich_mit(X, X).
```

Alle Terme, die gleich sind.

Das Prädikat gleich_mit/2 beschreibt also alle gleichen, unifizierbaren Terme. In Prolog gibt es dieses Prädikat als vordefiniertes Prädikat unter dem Namen =/2. Statt des Ziels ← gleich_mit(X, Y). wird das Ziel ← X = Y. verwendet. Dieses Prädikat wird im Gegensatz zu den bisher verwendeten Prädikaten in **Operatorschreibweise** angeschrieben. Dies ist jedoch nur eine syntaktische Konvention, die der besseren Lesbarkeit dient.

Unifikation übernimmt in Prolog die Aufgabe, Terme zu vergleichen, zu erzeugen und zu selektieren. Das folgende Faktum erlaubt es, aus einem Datum die Jahreszahl zu selektieren.

```
datum_jahr(datum(Jahr,_Monat,_Tag),Jahr).

← datum_jahr(datum(1741,3,13),Jahr).
@@ % Jahr = 1741.
```

Umgekehrt ist es ebenso möglich, mittels eines bekannten Jahres ein neues Datum zu erzeugen. Hier sind _A und _B freie Variablen. Das Datum ist also nicht vollständig beschrieben.

```
← datum_jahr(Datum,1741).
@@ % Datum = datum(1741,_A,_B).
```

In vielen Fällen entspricht Unifikation dem sogenannten ***matching***. Das heißt, dass nur die Variablen einer Seite der Gleichung X = Y gebunden werden. Unifikation ist aber viel allgemeiner, wie die folgenden Fälle zeigen.

- Dieselben Variablen kommen auf beiden Seiten vor.

  ```
  ← f(X, Y) = f(Y, Z).
  ```

- Eine Variable wird an verschiedene Terme gebunden. (Vorausgesetzt, diese sind wiederum unifizierbar.)

  ```
  ← f(g(A), g(1), G) = f(Z, Z, Z).
  ```

- Mehr als zwei Variablen werden aneinander gebunden.

  ```
  ← f(A, B, C, D) = f(B, C, D, A).
  ```

Prolog geht beim Unifizieren folgendermaßen vor: Die gleichzusetzenden Terme werden anfangs in mehrere Gleichungen zerlegt. Dann werden diese Gleichungen schrittweise aufgelöst. Gleichungen, die im nächsten Schritt entfernt werden, sind unterstrichen. Neu hinzugekommene Gleichungen sind **fett** gedruckt.

← f(U, g(V)) = f(g(a), W).
% Hilfsvariable X1, X2 für die linke Seite
% und Y1, Y2, Y3 für jeden nichtvariablen Subterm der rechten Seite
← X1 = f(U, X2), X2 = g(V), Y1 = f(Y2, W), Y2 = g(Y3), Y3 = a, X1 = Y1.
% Auflösen von X1 = Y1
← X2 = g(V), Y2 = g(Y3), Y3 = a, **U = Y2, X2 = W**.
% Entfernung von X2
← Y2 = g(Y3), Y3 = a, U = Y2, **W = g(V)**.
% Entfernung von Y2
← Y3 = a, W = g(V), U = g(Y3).
% Entfernung von Y3
← W = g(V), **U = g(a)**.

6.3 Occur-Check

Nach der Definition der Unifikation müsste das Ziel ← X = f(X). scheitern. Hier wird eine Variable an eine Struktur gebunden, in der die Variable selbst vorkommt. Die Struktur f/1 würde also sich selbst als Subterm besitzen. Wie in den meisten Prologsystemen so auch in dem in der Übungsumgebung verwendeten sind solche Terme zulässig; der ***occur-check*** wird nicht durchgeführt. Statt dessen wird ein sog. **infiniter Term** erzeugt. Der Grund für diesen Kompromiss sind —wie schon bei den Kompromissen im Beweisverfahren— Effizienzüberlegungen. Wenn infinite Terme erzeugt werden, handelt es sich fast immer um fehlerhafte Programme. Durch &@(Ebene,_) wird der infinite Term textlich dargestellt. Diese Notation bedeutet: anstelle des &@(N,_) sollte an dieser Stelle der Term, der N Ebenen weiter oben vorkommt, eingesetzt werden.

← X = f(X).
@@ % A = f(&@(1,'↑↑Infiniter_Term'(f/1))). % entspricht A = f(f(f(f(...)))).

6.4 Ungleichheit

Zusätzlich zur Gleichheit von Termen erlauben viele Prologsysteme auch die explizite Verwendung von **Ungleichheit**. Das vordefinierte Prädikat dif/2 ist wahr, wenn die beiden Argumente ungleich sind. Die durch dif/2 definierten Ungleichungen über Terme können auch bei freien Variablen zusichern, dass Terme stets ungleich bleiben.

← dif(X, Y), X = a, Y = b.
↚ dif(X, Y), X = a, Y = a.

7 Termarithmetik

Eine besonders einfache Darstellung von natürlichen Zahlen wird mit Strukturen realisiert. Derartige Zahlen werden auch **s(X)-Zahlen** genannt. Sie sind wie folgt definiert:

1. Die Konstante 0 ist eine natürliche Zahl.
2. Wenn X eine natürliche Zahl ist, so ist auch s(X) ist eine natürliche Zahl.

Die informelle Definition lässt sich direkt in Prolog übertragen:

```
natürlichezahlsx(0).
natürlichezahlsx(s(SN)) ←
   natürlichezahlsx(SN).
```

Um die obige Regel zu verstehen, verwende man die deklarative Lesart zur Analyse des Regelkörpers. Durch das Zudecken des Regelkopfes betrachten wir nun ausschließlich das Ziel natürlichezahlsx(SN):

```
natürlichezahlsx(s(SN)) ←  % Kopf = Schluß verdecken
   natürlichezahlsx(SN).
```

Wenn SN eine natürliche Zahl ist, ...

```
natürlichezahlsx(s(SN)) ←
   natürlichezahlsx(SN).
```

..., dann ist auch s(SN) *eine natürliche Zahl.*

Statt der Struktur s/1 hätte man ebensogut eine beliebige andere einstellige Struktur verwenden können. Die Anfrage ← natürlichezahlsx(SN). *Welche natürlichen Zahlen gibt es?* besitzt eine unendlich große Lösungsmenge:

```
← natürlichezahlsx(SN).
← natürlichezahlsx(SN).
@@ % SN = 0.
@@ % SN = s(0).
@@ % SN = s(s(0)).
@@ % SN = s(s(s(0))).
@@ % SN = s(s(s(s(0)))).
@@ ? Weitere Lösungen mit SPACE
```

Die s(X)-Repräsentation von Zahlen unterscheidet sich völlig von jener der Ganzzahlen, wie sie in früheren Beispielen verwendet worden ist und in Prolog vorhanden ist. s(X)-Zahlen sind nicht mit Ganzzahlen unifizierbar.

```
⊬ s(0) = 1. % Ganzzahlen und s(X)-Zahlen verschieden.
```

Aufbauend auf dieser Repräsentation natürlicher Zahlen können komplexere Relationen (wie etwa die Relation nat_nat_summe/3) dargestellt werden. Das Prädikat nat_nat_summe(I, J, K) ist wahr, wenn K die Summe aus I und J ist.

```
← nat_nat_summe(s(0), s(s(0)), s(s(s(0)))).

nat_nat_summe(0, I, I).
nat_nat_summe(s(I), J, s(K)) ←
    nat_nat_summe(I, J, K).
```

Dabei ähneln diese Definitionen der Struktur von natürlichezahlsx/1:

```
nat_nat_summe(0, I, I).
nat_nat_summe(s(I), J, s(K)) ←
    nat_nat_summe(I, J, K).

natürlichezahlsx(0).
natürlichezahlsx(s(SN)) ←
    natürlichezahlsx(SN).
```

Terminationsüberlegungen. Das Prädikat nat_nat_summe/3 terminiert, wenn das folgende Fragment terminiert (und daher scheitert). Da das zweite Argument J eine freie Variable ist und auch im Ziel vorkommt, kann das zweite Argument die Termination des Prädikats nicht beeinflussen. Die verbleibende Definition zeigt die Symmetrie des ersten und dritten Arguments. nat_nat_summe/3 terminiert nur dann, wenn das erste oder dritte Argument eine bekannte Zahl ist.

```
nat_nat_summe(0, I, I) ← false.
nat_nat_summe(s(I), J, s(K)) ←
    nat_nat_summe(I, J, K).

nat_nat_summe(0, I, I) ← false.
nat_nat_summe(s(I), J, s(K)) ←
    nat_nat_summe(I, J, K).
```

8 Listen

Listen sind spezielle Strukturen mit einer äußerlich etwas unterschiedlichen Syntax. Listen werden zur Darstellung von Sequenzen von Elementen verwendet.

8.1 Syntax

← [ ] = Leere_Liste.
← [a] = Einelementige_Liste.
← [a,b] = Zweielementige_Liste.
← [a,b,c] = Dreielementige_Liste.
... .

Die eckigen Klammern sind hier eine spezielle kompakte Notation für die folgenden Strukturen '.'/2. '.'/2 wird **Listenkonstruktor** genannt. Der Listenkonstruktor ist eine Struktur der Arität zwei. Der einzige Unterschied zwischen einem Listenkonstruktor und einer anderen Struktur der Arität zwei liegt in der unterschiedlichen syntaktischen Darstellung.

← [ ] = Leere_Liste.
← [a] = Einelementige_Liste, '.'(a,[ ]) = Einelementige_Liste.
← [a,b] = Zweielementige_Liste, '.'(a,'.'(b, [ ])) = Zweielementige_Liste.
← [a,b,c] = Dreielementige_Liste, '.'(a,'.'(b, '.'(c, [ ]))) = Dreielementige_Liste.
... .

Statt '.' wird meist folgende Notation verwendet:

← [ ] = Leere_Liste.
← [a] = Einelementige_Liste, [a| [ ]] = Einelementige_Liste.
← [a,b] = Zweielementige_Liste, [a| [b | [ ]]] = Zweielementige_Liste.
← [a,b,c] = Dreielementige_Liste, [a| [b | [c | [ ]]]] = Dreielementige_Liste.
... .

Allgemein werden Listen wie das folgende is_list/1 definiert: 1) [] ist eine Liste, 2) Wenn Xs eine Liste ist, dann ist auch [X|Xs] eine Liste. Prädikate über Listen sind von ähnlicher Gestalt wie dieses Prädikat.

```
is_list([ ]). % Die leere Liste, das Atom [ ] (nil).
is_list([_X | Xs]) ←% Der Listenkonstruktor [_X|Xs] — also '.'(_X,Xs)
%_X ist ein Listenelement, auch Listenkopf genannt
   is_list(Xs). % Xs ist der Listenrest. Xs muss wiederum eine Liste sein.

← is_list(Xs).
@@ % Xs = [ ].
@@ % Xs = [_A].
@@ % Xs = [_A,_B].
@@ % Xs = [_A,_B,_C].
@@ % etc.
```

Eine **unvollständige Liste** hat als Rest eine freie Variable. So besteht die Liste Unvollständige_Liste aus zumindest drei Elementen. Sie beginnt mit den Elementen a, b und c. Durch Bindung von Xs kann die Liste „noch größer werden".

```
← Unvollständige_Liste = [a,b,c|Xs].
```

Beachten Sie bitte, dass nicht jede Instanz des Listenkonstruktors eine gültige Liste ist.

```
← Nichtgültige_Liste = [X|Xs], X = e, Xs = b.
⊬ is_list([e|a]).
```

Terme wie etwa [e|a] oder [1,2|b] sind keine Listen. Im ersten Fall ist das Restargument des Listenkonstruktors das Atom a/0, im zweiten Fall ist das Restargument des zweiten Listenkonstruktors das Atom b/0.

8.2 Listenvariablennamen

Wenn Listen verwendet werden, ist es sinnvoll, Variablennamen für die Listenelemente und die Restliste einheitlich zu vergeben. Dies verbessert die Lesbarkeit der Programme. Da Listen meist gleichartige Elemente enthalten, ist es sinnvoll, den Rest der Liste mit dem Namen des Elements —jedoch in den Plural gesetzt— zu verwenden. Wenn ein Element einer Liste eine Zahl ist, so nenne man die Liste Zahlen. Als Elementname Baum, als Name der Liste Bäume. Wenn die Verwendung des Plurals keinen Sinn ergibt, ist die Verwendung einer Liste möglicherweise gar nicht angebracht. Eine Struktur könnte u.U. angemessener sein. Bei kurzen nicht sprechenden Variablennamen, wie X für ein Element der Liste, bezeichne man den Rest der Liste mit Xs entsprechend dem englischen Plural-s.

Um zu entscheiden, ob eine Liste oder eine Struktur als Datenstruktur verwendet werden soll, stelle man die folgenden Überlegungen an.

- Handelt es sich bei den Elementen um gleichartige Objekte?
- Ist auch die leere Liste [ ] sinnvoll?
- Ist die Anzahl der Elemente variabel?

Zur Darstellung eines Paars ist es z.B. besser, eine Struktur paar/2 zu verwenden, weil schon die leere Liste nicht sinnvoll ist.

8.3 Stringnotation, Zeichencodes

Zeichencode Texte (**Strings**) werden in Prolog als Listen von Latin1-Zeichen dargestellt. Um Texte einfacher anschreiben zu können, gibt es eine eigene Notation. Die Zeichen des Strings werden unter doppelte Hochkommas gesetzt. Sie stehen für eine Liste der entsprechenden Zeichencodes.

```
← [Zeichencode_von_a] = "a".
@@ % Zeichencode_von_a = 97.
← [Zeichencode_von_a, Zeichencode_von_b] = "ab".
@@ % Zeichencode_von_a = 97, Zeichencode_von_b = 98.
← [233,103,97,108,32,224,32,49] = "égal à 1".
```

Beachten Sie, dass Strings weder Atome noch eine Liste von Atomen sind.

```
⊬ "abcdef" = [a,b,c,d,e,f].
⊬ "abcdef" = 'abcdef'.
```

Einzelne **Latin1-Zeichen** werden wie folgt dargestellt:

```
← 0'é = Z.
@@ % Z = 233.
@@ % Eine Lösung gefunden
← "é" = [0'é].
```

9 Grammatiken

Zur kompakten Beschreibung komplex strukturierter Listen gibt es in Prolog einen eigenen Formalismus. Durch eine **Grammatik**, auch **DCG** genannt (engl. ***definite clause grammar***), werden Listen von Termen beschrieben.

9.1 Syntax

Eine **Grammatikregel** wird mittels des Pfeils ⟶ (im Programmtext wie `-->`) angegeben. Sie beschreibt, wie eine Liste aufgebaut ist. Wir beschreiben nun einfache Sätze, die als Listen von Atomen dargestellt werden. So soll etwa folgende Liste ein gültiger Satz sein.

```
[der,kleine,löwe,ist,glücklich]
```

Ein einfacher Aussagesatz besteht aus Subjekt und Prädikat. Als Grammatikregel schreiben wir: *Ein Aussagesatz besteht aus einem Subjekt gefolgt von einem Prädikat.*

```
aussagesatz ⟶
   subjekt,
   prädikat.
```

aussagesatz//0, subjekt//0, prädikat//0 werden **Nichtterminalsymbole** genannt. (Um sie von gewöhnlichen Prädikaten zu unterscheiden, schreibt man Name//Stelligkeit.) Sie stellen noch keine konkreten Elemente der zu beschreibenden Liste dar. Konkrete Listenelemente werden mit der folgenden Listennotation angeschrieben. Solche Listen, wie [der], werden **Terminalsymbole** genannt, weil sie bereits Teile des zu beschreibenden Satzes sind. Folgen von Terminalsymbolen kann man auch, wie bei der zweiten Regel für subjekt//0, zu einer mehrelementigen Liste zusammenfassen.

```
subjekt ⟶
   [der],
   [große],
   [bär].
subjekt ⟶
   [der, kleine, löwe].% drei Terminalsymbole
```

```
prädikat ⟶
   [brüllt].
prädikat ⟶
   [ist, glücklich].
prädikat ⟶
   [wohnt, in, der, goldenen, stadt].
```

9.2 phrase/2

Durch das Prädikat phrase(NichtTerminal,Liste) werden Grammatikregeln von Prolog aus verwendet. phrase/2 ist wahr, wenn Liste von der Form NichtTerminal ist. Grammatiken können zum Testen oder zum Generieren von Listen verwendet werden.

```
← phrase(aussagesatz, Aussagesatz).
@@ % Aussagesatz = [der,große,bär,brüllt].
@@ % Aussagesatz = [der,große,bär,ist,glücklich].
@@ % Aussagesatz = [der,große,bär,wohnt,in,der,goldenen,stadt].
@@ % Aussagesatz = [der,kleine,löwe,brüllt].
@@ % Aussagesatz = [der,kleine,löwe,ist,glücklich].
@@ % Aussagesatz = [der,kleine,löwe,wohnt,in,der,goldenen,stadt].
```

Um allgemeinere Sätze darzustellen, verfeinern wir die grammatische Struktur der Nichtterminale subjekt//0 und prädikat//0. Die folgende Regel besagt: *Ein Subjekt besteht aus einem Artikel, einem Adjektiv und einem Substantiv.*

```
subjekt ⟶          artikel ⟶
   artikel,           [der].
   adjektiv,       artikel ⟶
   substantiv.        [die].
                   artikel ⟶
                      [das].
```

Durch eine solche Beschreibung werden aber auch grammatisch nicht richtige Sätze beschrieben. Etwa „Das große Bär ...“. In grammatisch richtigen Sätzen müssen Satzteile nach grammatischen Kategorien wie Person (1., 2., 3.), Zahl (Einzahl/Mehrzahl) oder Fall (1.-4.) übereinstimmen. Um derartige Eigenschaften zu beschreiben, werden einem Nichtterminal Argumente beigefügt. Diese Argumente bestehen wie bei gewöhnlichen Prolog-Prädikaten aus Prolog-Termen. Die Kategorien Person und Zahl werden durch die Variablen gleichgesetzt.

```
aussagesatz ⟶
   subjekt(Person, Zahl),
   prädikat(Person, Zahl).

subjekt(3, Zahl) ⟶ % Ein Subjekt in der dritten Person besteht aus
   artikel(Geschlecht, Zahl, 1), % einem übereinstimmenden Artikel im 1. Fall
   adjektiv(Geschlecht, Zahl, 1), % —"— —"— Adjektiv —"—
   substantiv(Geschlecht,Zahl,1). % —"— —"— Substantiv —"—

adjektiv(_Geschlecht, _Zahl, _Fall) ⟶      substantiv(männlich, einzahl, 1) ⟶
   [ ]. % Ein Adjektiv kann entfallen          [löwe].
adjektiv(Geschlecht, Zahl, Fall) ⟶         substantiv(männlich, einzahl, 2) ⟶
   ... .                                       [löwen].
                                               ... .
```

Aus der neuen Definition eines Aussagesatzes kann man direkt ersehen, dass das Subjekt mit dem Prädikat in Person und Zahl übereinstimmen muss, weil beide Grammatikziele die gemeinsamen Variablen Person und Zahl aufweisen.

Viele Nichtterminale einer Grammatik bestehen aus einer großen Anzahl von sehr einfachen Regeln. Das Nichtterminal substantiv//3 sollte schließlich aus sämtlichen Hauptwörtern bestehen. Es ist übersichtlicher, für derartige sehr einfach strukturierte, aber sehr umfangreiche Daten, ein direkt in Prolog geschriebenes Prädikat zu definieren. Statt der obigen Regeln substantiv//3 definieren wir ein Prädikat substantiv_/4. Ziele für dieses Prädikat werden in die Grammatik mit geschwungenen Klammern eingefügt. Wie man aus der Deklination von Löwe ersieht, sind viele Fälle sehr ähnlich. Es ist naheliegend, statt einer umfangreichen Sammlung von Fakten, wiederum Regeln zu verwenden.

```
substantiv(Geschlecht, Zahl, Fall) ⟶
   [Substantiv],
   {substantiv_(Substantiv,Geschlecht, Zahl, Fall)}. % gewöhnliches Ziel

artikel(Geschlecht, Zahl, Fall) ⟶
   [Artikel],
   {artikel_(Artikel,Geschlecht, Zahl, Fall)}.
```

```
substantiv_(löwe, männlich, einzahl, 1).
substantiv_(löwen, männlich, einzahl, 2).
substantiv_(löwen, männlich, einzahl, 3).
substantiv_(löwen, männlich, einzahl, 4).
substantiv_(löwen, männlich, mehrzahl, 1).
... .
```

```
artikel_(der, männlich, einzahl, 1).
artikel_(des, männlich, einzahl, 2).
artikel_(dem, männlich, einzahl, 3).
artikel_(den, männlich, einzahl, 4).
artikel_(die, männlich, mehrzahl, 1).
... .
```

9.3 Deklarative Lesart einer Grammatik

Grammatikregeln beschreiben eine Folge (Liste) von Termen. Das Komma wird *und danach* gelesen. Zur Spezialisierung einer Grammatik füge man {false} ein. Zur Verallgemeinerung setze man * vor ein Grammatikziel. Innerhalb einer Regel halte man sich beim Wegstreichen von Grammatikzielen vor Augen, dass die sich so ergebende Regel die Liste nicht mehr vollständig beschreibt: *Ein Subjekt endet mit einem Substantiv im ersten Fall.*

```
subjekt(3, Zahl) ⟶
   * ~~artikel(Geschlecht, Zahl, 1)~~,
   * ~~adjektiv(Geschlecht, Zahl, 1)~~,
   substantiv(Geschlecht,Zahl,1).
```

9.4 Allgemeine Listen

DCGs und ähnliche Formalismen werden in Prolog zur Realisierung von Grammatiken für natürliche Sprachen und Computersprachen verwendet. Man kann aber DCGs auch einfach verwenden, um irgendwelche Listen von Termen darzustellen. So kann eine Liste, in der jedes Element zweimal hintereinander vorkommt, mit paare//0 wie folgt beschrieben werden.

```
← phrase(paare,"112233").
↚ phrase(paare,"11223").
```

```
paare ⟶
   [ ].
paare ⟶
   [E,E], % zwei gleiche Elemente
   paare.
```

Ein besonders nützliches Nichtterminal ist liste//1. Es beschreibt eine (beliebige) Folge von Elementen. Das Argument von liste//1 ist die Liste der Elemente, die liste//1 beschreibt.

```
liste([ ]) ⟶
   [ ].
liste([E|Es]) ⟶
   [E],
   liste(Es).
```

```
← phrase(liste("abcd"), Xs).
@@ % Xs = "abcd".
← phrase(liste(Xs), "abcd").
@@ % Xs = "abcd".
← phrase(liste(Es), Xs).
@@ % Es = [ ], Xs = [ ].
@@ % Es = [_A], Xs = [_A].
@@ % Es = [_A,_B], Xs = [_A,_B].
@@ % ...
```

Mit dem Nichtterminal liste//1 werden komplexere Relationen beschrieben. Das Nichtterminal liste_liste//2 beschreibt zwei aufeinanderfolgende Sequenzen. Prozedural betrachtet kann liste_liste//2 verwendet werden, um zwei Listen zusammenzuhängen, oder eine bestehende Liste in zwei Teile aufzuteilen.

```
liste_liste(As,Bs) ⟶
   liste(As),
   liste(Bs).
```

```
← phrase(liste_liste("a","b"), Xs). % Zusammenhängen
@@ % Xs = "ab".
← phrase(liste_liste(As,Bs), "abc"). % Aufteilen
@@ % As = [ ], Bs = "abc".
@@ % As = "a", Bs = "bc".
@@ % As = "ab", Bs = "c".
@@ % As = "abc", Bs = [ ].
@@ % 4 Lösungen gefunden
```

liste_liste_liste//3 beschreibt drei aufeinanderfolgende Listen/Sequenzen.

```
liste_liste_liste(As,Bs,Cs) ⟶ % Drei Listen
   liste(As),
   liste(Bs),
   liste(Cs).

← phrase(liste_liste_liste("IV","+","XII"), Ds).
@@ % Ds = "IV+XII".
← phrase(liste_liste_liste(_,"er",_), "Dieser Dichter"). % Ist "er" ein Substring?
← phrase(liste_liste_liste(As,Bs,Cs), "abc").
@@ % As = [ ], Bs = [ ], Cs = "abc".
@@ % As = [ ], Bs = "a", Cs = "bc".
@@ % As = [ ], Bs = "ab", Cs = "c".
@@ % ... .
```

9.5 Termination

Da Grammatikregeln zumeist rekursiv sind, können in ihnen, wie in rekursiven Prologregeln, Endlosschleifen auftreten. Wird eine Grammatik verwendet, um eine Liste zu erzeugen, so stelle man Überlegungen wie für gewöhnliche Prolog-Programme an. Man betrachtet also die Argumente des Nichtterminals.

Ist die Liste aber bekannt, wird die Grammatik also zum Analysieren eines Programms verwendet, ist die folgende Betrachtungsweise hilfreich. Ein Ziel einer Grammatik terminiert für eine gegebene Liste, falls *vor* einem rekursiven Ziel ein Terminalsymbol vorkommt. Im folgenden Beispiel wird versucht, eine Liste von Atomen a auf zweierlei Arten zu beschreiben. Beide Beschreibungen as1//0 und as2//0 sind deklarativ gesehen äquivalent.

```
as1 -->                 as2 -->
   [ ].                    [ ].
as1 -->                 as2 -->
   [a],                    as2,
   as1.                    [a].
```

Um mögliche Endlosschleifen zu ersehen, betrachten wir nur die zweite Regel und in dieser vorerst das erste Ziel. In as1//0 muss die Liste ein Atom a enthalten. Ist dies nicht der Fall, scheitert die Regel. Für as2//0 gilt dies jedoch nicht. Die Regel as2//0 ist immer anwendbar, auch wenn eine leere Liste oder eine Liste mit von a verschiedenen Termen zu analysieren ist. Die Regel as2//0 terminiert also nicht. as1//0 hingegen terminiert, falls die Größe der Liste bekannt ist: Durch jede Rekursion wird die um ein Element kleinere Liste betrachtet. Die in as2//0 auftretende stets zu einer Endlosschleife führende Rekursion wird im Übersetzerbau **Linksrekursion** genannt.

```
~~as1~~ --> false,          ~~as2~~ --> false,
   ~~[]~~.                     ~~[]~~.
as1 -->                     as2 -->
   [a], false,                 as2, false,% Linksrekursion
   ~~as1~~.                    ~~[a]~~.
```

9.6 Text„ausgabe“

Durch Grammatiken lassen sich insbesondere Texte (Listen von Latin1-Zeichen) beschreiben, die bei interaktiven Anfragen als Antwortsubstitutionen zu sehen sind. Bei größeren Texten ist diese Darstellung unübersichtlich. Als Beispiel diene das Nichtterminal lineal//1.

```
lineal(6) --> % Ein Lineal der Länge 6.      zeilenumbruch -->
   "0 1 2 3 4 5 6",                             [10].
   zeilenumbruch,
   "|-|-|-|-|-|-|".
← phrase(lineal(6),Xs).
@@ % X = [48,32,49,32,50,32,51,32,52,32,53,32,54,10,124,95 ...].
@@ % Eine Lösung gefunden.
```

Der Text, der das Lineal darstellt, wird also als String beschrieben, welcher auch einen zeilenumbruch//0 enthält. Bei einer interaktiven Anfrage erhalten wir als Antwortsubstitution nun lediglich die Liste der Zeichencodes, weil sich diese Liste ja nicht in einer Zeile als Textstring darstellen ließe. Durch Voransetzen der Annotation text(Xs) ⋘ geben wir an, dass Antwortsubstitutionen der Variable Xs als Text dargestellt werden sollen.

← **text(Xs)** ⋘ phrase(lineal(6),Xs). % Annotierte Anfrage

Bei einer interaktiven Anfrage wird nun zusätzlich zur (unleserlichen) Antwortsubstitution ein eigenes Fenster auf der rechten Seite geöffnet, das den Text übersichtlich darstellt:

```
0 1 2 3 4 5 6
|_|_|_|_|_|_|
```

10 Arithmetik mit ganzen Zahlen — *finite domains*

Ganze Zahlen werden in modernen Prologsystemen mittels sogenannter *finite domains* dargestellt. Die Prädikate zur Beschreibung von Beziehungen zwischen ganzen Zahlen beginnen mit einem #: #>/2, #>=/2, #=/2, #=</2, #</2 und für Ungleichheit #\=/2. Die Argumente dieser Relationen sind einfache ganze Zahlen (1, 2, -99,...), Variable und zusammengesetzte Ausdrücke, die mit den Operatoren +,−,*,/,mod,min(_,_),max(_,_) und abs(_) gebildet werden. Diese Prädikate nennt man auch *finite domain constraints.*

Das Prädikat sx_n0/2 beschreibt die Beziehung der s(X)-Repräsentation und der neuen Zahlendarstellung.

```
sx_n0(0, 0).
sx_n0(s(SN), N0) ←
   N0 #> 0,
   N0 #= N1+1,
   sx_n0(SN, N1).
```

```
← sx_n0(SN,N).
@@ % N = 0, SN = 0.
@@ % N = 1, SN = s(0).
@@ % N = 2, SN = s(s(0)).
... .
```

```
← 1+1 #= 2.
← 1+N #= 2.
@@ % N = 1.
← N+N #= 2.
@@ % N = 1.
```

```
← N+M #= 2.
@@ % N in inf..sup, M in inf..sup. % −∞.. + ∞
← N+M #= 2, N #>= 0, M #>= 0.
@@ % N in 0..2, M in 0..2.
← N+M #= 2, N #>= 0, M #>= 0, N #< M.
@@ % N in 0..1, M in 1..2.
```

Falls die Anfrage noch zu allgemein war, um als Lösung Antwortsubstitutionen bestimmen zu können, wird als Antwort der Wertebereich angegeben, in dem sich die Lösungen (falls sie existieren) befinden müssen. Statt einer konkreten Antwortsubstitution (z.B. N = 1) werden nun Ausdrücke der Form N in Min..Max verwendet. Durch weitere Ziele kann die Lösung immer stärker eingeschränkt (engl. ***constrained***) werden. Im obigen Beispiel konnte die Variable N ausgehend vom Intervall $-\infty.. + \infty$ auf das Intervall 0..1 eingeschränkt werden.

Die Intervalle approximieren jedoch nur den Wertebereich, in dem sich Lösungen finden können. Um Gewissheit zu erlangen, müssen sämtliche Variable durch konkrete Werte ersetzt werden. Das vordefinierte Prädikat indomain/1 ersetzt Variablen durch konkrete Werte.

```
← N+M #= 2, N #>= 0, M #>= 0, N #< M.
@@ % N in 0..1, M in 1..2. % zu allgemeine Approximation
← N+M #= 2, N #>= 0, M #>= 0, N #< M, indomain(N).
@@ % M = 2, N = 0. % nur eine Lösung, Wertebereich war größer
@@ % Eine Lösung gefunden
```

Gelegentlich kommt es sogar vor, dass es gar keine Antwortsubstitution geben kann. Dennoch bietet uns Prolog eine Scheinlösung an. Die Anfrage besagt: *Gibt es drei verschie-*

dene Werte im Wertebereich 1..2? Durch das Ziel indomain(N1) kann diese Scheinlösung entfernt werden.

```
← N1 #\= N2, N1 #\= N3, N2 #\= N3, N1 in 1..2, N2 in 1..2, N3 in 1..2.
@@ % N1 in 1..2, N2 in 1..2, N3 in 1..2. % Scheinlösung
↯N1 #\= N2, N1 #\= N3, N2 #\= N3, N1 in 1..2, N2 in 1..2, N3 in 1..2, indomain(N1).
```

Da meist sehr viele Variable in einem Programm vorkommen, verwendet man statt in/2 und indomain/1, zwei Verallgemeinerungen, die über Listen von Variablen definiert sind. Die Prädikate domain_zs/2 und labeling_zs/2 sind in etwa wie folgt implementiert.

```
domain_zs(_Min.._Max, []).
domain_zs(Min..Max, [Z|Zs]) ←
   Z in Min..Max,
   domain_zs(Min..Max,Zs).

labeling_zs([], []).
labeling_zs([], [Z|Zs]) ←
   indomain(Z),
   labeling_zs([], Zs).
```

Programmaufbau. Programme mit *finite domain constraints* folgen einem einfachen Schema. Die eigentliche Relation kernrelation/1 und das Labelingverfahren werden voneinander getrennt definiert. Durch diese Trennung kann ein und dieselbe kernrelation/1 mit mehreren labeling-Methoden kombiniert werden. Andererseits kann dadurch auch die Termination von relation/1 rasch bewiesen werden. Es genügt nun, die Termination von kernrelation/1 zu beweisen, da labeling_zs/2 ohnehin immer terminiert.

```
relation(Zs) ←
   kernrelation(Zs),
   labeling_zs([], Zs).

kernrelation(Zs) ←
   groberwertebereich(Zs), % z.B. domain_zs/2
   beziehung(Zs).

↯$ relation(Zs), false. % sehr aufwendig
↯ kernrelation(Zs), false. % terminiert schneller
```

Beispiel: Planung. Es soll ein Plan gefunden werden, um sechs verschiedene Aufgaben in möglichst kurzer Zeit zu erledigen. Bild 2 zeigt die Abhängigkeiten zwischen den sechs Aufgaben. Für jede Aufgabe benötige man eine Zeiteinheit. Aufgabe 2 und 3 dürfen nicht zum gleichen Zeitpunkt erledigt werden. Gesucht ist ein Plan, der mit den Einschränkungen in Bild 2 konsistent ist. Für jede Aufgabe wird also ein Zeitpunkt gesucht, zu dem sie erledigt werden soll.

Der Plan muss für jede Aufgabe den Zeitpunkt, zu dem sie erledigt werden soll, enthalten. Es ist naheliegend, den Plan als Liste von Zeitpunkten (= ganze Zahlen) darzustellen. Das erste Element der Liste beschreibt den Zeitpunkt für die erste Aufgabe u.s.f. Das vordefinierte Prädikat zs/1 beschreibt eine Liste ganzer Zahlen.

```
is_plan(Zs) ←
   Zs = [A1,A2,A3,A4,A5,A6], % A1 = Zeitpunkt für Aufgabe 1 ...
   zs(Zs). % eine Liste ganzer Zahlen

← plan([1,2,3,4,5,6]). % Ein gültiger aber nicht optimaler Plan
```

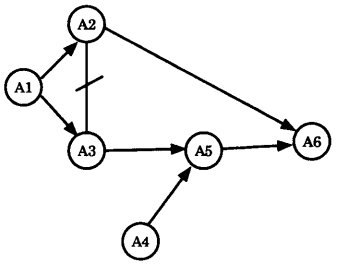


Abbildung 2: Abhängigkeitsgraph

Die Abhängigkeitspfeile aus dem Diagramm können direkt mit der Relation $\#<$ dargestellt werden. Dass Aufgabe 2 und 3 nicht zugleich stattfinden dürfen, drücken wir mittels A2 $\#\not=$ A3 aus (A2 ungleich A3).

```
plan([A1,A2,A3,A4,A5,A6]) ←
   A1 #< A2,
   A1 #< A3,
   A4 #< A5,
   A3 #< A5,
   A5 #< A6,
   A2 #< A6,
   A2 #\= A3.
← plan(As).
@@ % As = [_A,_B,_C,_D,_E,_F], _B in inf..sup, _A in inf..sup, _C in inf..sup, ...
@@ % Eine Lösung gefunden
```

Für die Anfrage ← plan(As). (*Welche Pläne gibt es?*) findet sich eine Lösung, die aber noch viel zu allgemein ist: Sämtliche Variablen liegen im Wertebereich $-\infty..+\infty$. Wir beschränken die möglichen Zeitpunkte auf einen möglichen Wertebereich (1..6) mit dem vordefinierten Prädikat domain_zs/2. Durch diese Einschränkung können die möglichen Wertebereiche bereits weiter konkretisiert werden: Der Zeitpunkt für die erste Aufgabe (Variable_A) kann nun nur mehr im Bereich 1..3 liegen.

```
← plan(As), domain_zs(1..6,As).
@@ % As = [_A,_B,_C,_D,_E,_F], _B in 2..5, _A in 1..3, _C in 2..4, _E in 3..5, _D in 1..4, ...
@@ % Eine Lösung gefunden
```

Um nun konkrete Lösungen zu sehen, entfernt labeling_zs/1 alle Variable. Die so erhaltene Lösungsmenge ist allerdings sehr groß. Meist kann sie gar nicht aufgezählt werden, auch wenn sie endlich ist. Wir müssen also den Wertebereich vor dem Ziel labeling_zs/1 noch weiter einschränken.

```
← plan(As), domain_zs(1..6,As), labeling_zs([], As).
@@ % As = [1,2,3,1,4,5].
... .
@@ % As = [3,5,4,4,5,6].
@@ % 107 Lösungen gefunden
```

← plan(As), domain_zs(1..4,As). % weitere Einschränkung auf 1..4
← plan(As), domain_zs(1..4,As).
@@ % As = [1,3,2,_A,3,4], _A in 1..2.
@@ %← plan(As), domain_zs(1..4,As), labeling_zs([],As).
@@ % As = [1,3,2,1,3,4].
@@ % As = [1,3,2,2,3,4].
@@ % 2 Lösungen gefunden
↚ plan(As), domain_zs(1..3,As). % keine Lösung für 1..3, also ist 1..4 ist das Minimum

Im allgemeinen ist es unmöglich, die Wertebereiche von Hand zu verkleinern. Statt dessen können wir uns darauf konzentrieren, die Pläne, die am frühsten enden, *zuerst* zu sehen. Das vordefinierte Prädikat max_ofzs/2 beschreibt das Maximum einer Liste von Werten. Mit max_ofzs/2 wird der späteste Zeitpunkt Max einer Aufgabe beschrieben werden.

← plan(As), domain_zs(1..6,As), max_ofzs(Max,As).
@@ % As = [_A,_B,_C,_D,_E,_F], Max in 4..6, _G in 4..6, _H in 4..6, _I in 4..6, ...
← plan(As), domain_zs(1..6,As), max_ofzs(Max,As), indomain(Max), labeling_zs([], As).
@@ % As = [1,3,2,1,3,4], Max = 4.
@@ % As = [1,3,2,2,3,4], Max = 4.
@@ % As = [1,2,3,1,4,5], Max = 5.
@@ % As = [1,2,3,2,4,5], Max = 5.
@@ % As = [1,2,3,3,4,5], Max = 5.
... .
@@ % 107 Lösungen gefunden

Glossar

Die meisten der folgenden Begriffe entsprechen dem ISO- bzw. DIN-Prolog-Standard (ISO/IEC DIS 13211-1).

abstrakte Maschine, die: *Prologmaschine.*

allgemeinster Unifikator, der: eine minimale Variablenbindung (Variablensubstitution), um zwei Terme gleichzusetzen. Jeder Unifikator ist *Instanz* eines allgemeinsten Unifikators.

Anfrage, die: Klausel mit leerem Kopf. Falls Anfrage erfüllt, werden Antwortsubstitutionen angezeigt. Siehe auch *interaktive Anfrage.*

anonyme Variable, die: Variable, die nur einmal in einer Klausel vorkommt. In Prädikaten müssen anonyme Variablen mit einem Unterstrich beginnen.

anonymous variable: *anonyme Variable.*

Antwortsubstitution, die: *Variablenbindung*, für die *Anfrage* erfüllt ist.

Arität, die: *Stelligkeit.*

arity: [engl.] *Stelligkeit.*

Atom, das: nullstelliges Funktionssymbol. Konstante. Z.B. `atom/0`. Andere Bedeutung (innerhalb der Mathematischen Logik): Atomformel.

atomic term: Ein Atom oder eine Zahl.

atom: *Atom.*

benannte Variable, die: Variable, deren Name mit einem Großbuchstaben beginnt.

BIP: Abkürzung für *built-in predicate.*

body: *Rumpf.*

built-in predicate: *vordefiniertes Prädikat.*

clause: *Klausel.*

compound term: *Struktur.*

Datenbasis: (Teil eines) Prologprogramm(s), das nur aus *Grundfakten* besteht. Gelegentlich auch synonym für Programm.

DCG: Abkürzung für definite clause grammar, *Grammatik.*

definite clause grammar: *Grammatik*

Definition, die: Gesamtheit der Fakten und Regeln eines Prädikats.

deklarativ: Gegenteil zu *imperativ.* Beschreibungsorientiert. Eine Programmiersprache ist deklarativ, wenn ihre Grundelemente zur Beschreibung der darzustellenden Dinge bestehen.

Differenzpaar, das: Paar von Termen zur Beschreibung einer Differenz.

Differenz, die: Das, was ein Differenzpaar beschreibt. Siehe auch Listendifferenz.

Element, das: *Listenelement.*

empty list: *nil.*

Emulator, der: Implementierung einer *Prologmaschine.* (auch: Zwischencodeinterpreter) Prologprogramme werden in einen eigenen Zwischencode eines Zwischencodeinterpreter (Emulators) übersetzt. Ungefähr eine Größenordnung schneller als Interpreter. Portabel, falls Emulator in C geschrieben. Manche Hersteller nennen *Emulatoren* auch *Compiler.*

Endlosableitung, die: Ein Ziel führt zu einer Endlosableitung, wenn $\not\leftarrow$ Ziel, false. nicht termininert. Ein Ziel kann durchaus Antwortsubstitutionen liefern und dennoch zu einer Endlosableitung führen.

enthalten, zu: Ein Term ist in einem anderen Term enthalten, wenn dieser ein Unterterm ist.

erfolgreich: *erfüllen.*

erfüllen, zu: Ein Ziel (erfolgreich) erfüllen heißt ein beweisbares Ziel beweisen. (satisfy, to)

existentielle Variable, die: Variable, die nur im Rumpf einer Regel vorkommt.

false/0: vordefiniertes Prädikat, das immer scheitert

fail, to: *scheitern.*

Faktum: eine *Klausel*, deren *Rumpf* leer ist, also deren *Rumpf* kein Ziel enthält.

Funktorname, der: Name eines Funktors. Auch Funktionssymbol genannt.

Funktor, der: Ein *Name* zusammen mit einer *Stelligkeit.* Der Funktor `f/3`. Ein *Prädikatsindikator* ist auch ein Funktor.

goal: [engl.] *Ziel.*

Grammatikregel, die: Eine Grammatikregel besteht aus einem Regelkopf einem Grammatikpfeil und einem Rumpf: `a` $\longrightarrow$ `b, c`. Sie besagt, dass eine Liste der Form `a//0` aus einer Liste der Form `b//0` gefolgt von einer Liste der Form `c//0` besteht.

Grammatik, die: Prolog-Grammatik: Formalismus bestehend aus Grammatikregel zur Beschreibung von Listen von Termen.

ground term: *Grundterm*

Grundfaktum, das: *Faktum* ohne Variable.

Grundterm, der: Ein variablenfreier Term.

Horn-Klausel, die: *Klausel.*

identifier: *Name.*

imperativ: Gegenteil zu *deklarativ.* Befehlsorientiert. Eine Programmiersprache ist i., wenn ihre Grundelemente Befehle sind.

Inferenz, die: Ersetzung eines zu beweisenden Ziels durch ein unifizierbares Faktum bzw. Regel.

infiniter Term, der: Term der einen sich selbst enthaltenden Subterm enthält. $\leftarrow$ `X = f(X), Y = g(X)`. Sowohl `X` als auch `Y` sind infinite Terme. Infinite Terme können nur auftreten, wenn während der Unifikation

kein occur-check erfolgt.

instantiieren, zu: (auch instanzieren) Variablenbindung durchführen.

Instanz, die von: Speziellerer Term, od. Ziel Terms. f(a,X) ist Instanz von f(Y,X).

integer: *Ganzzahl.*

interaktive Anfrage, die: Anfrage mit Antwortsubstitutionen. Im Programmtext ist eine Anfrage gleich einer Zusicherung. Antwortsubstitutionen werden nur gezeigt, wenn man danach fragt.

interne Variable, die: *existentielle Variable.*

Interpreter, der: Spezielle *Prologmaschine.* (auch: Meta-Interpreter) Einfachste Form der Implementierung. Prologprogramme werden als Prologterme dargestellt. Wird heute nur mehr zusammen mit besseren Techniken angeboten.

Klausel, die: Prologklauseln sind Hornklauseln. Sie enthalten genau ein Literal im Kopf und beliebig viele Literale im Rumpf.

klips, pl. die: 1 000 lips.

Konjunktion, die: Und-Verknüpfung von Zielen. Eine Konjunktion ist wahr, wenn alle Ziele der Konjunktion wahr sind. ← a, b, c. ist eine Konjunktion dreier Ziele.

Konstante, die: Ein Atom oder eine Zahl.

Kopf, der: *Regelkopf, Listenkopf.*

Latin1-Zeichen, das: ISO 8859/1-konformes Zeichen. Mit ISO 8859/1 werden westeuropäische Schriftzeichen dargestellt. Einzelne Zeichen werden in Prolog als Ganzzahlen dargestellt. ← 0'ê = 234.

leere Liste, die: *nil.*

Lesart, die: Es wird zwischen der informellen (Sätze auf Deutsch lesen), deklarativen (Verdecken beliebiger Programmteile) und prozeduralen (Aufdecken von Zielen von oben nach unten) unterschieden.

Linksrekursion, die: Rekursion in einer Grammatikregel, wobei das rekursive Nichtterminal vor einem Terminal auftritt.

lips, pl. die: Logik-Inferenzen pro Sekunde. Maß für die Geschwindigkeit eines Prologsystems. Grundlage für die Messung ist die Zeit zur Ausführung des Ziels ← nr(List30,_). Dabei ist List30 eine 30-elementige Liste. Es werden für dieses Ziel $n = 30$ $((n+1).(n+2))/2 = 496$ Inferenzen benötigt. Eines der ersten Prologsysteme leistete 200 lips; heutige Prologs mehr als 2 000 000 lips.

```
nr([], []).              a([], L, L).
nr([X|Xs], R) ←          a([H|T], L, [H|R]) ←
  nr(Xs, R1),              a(T, L, R).
  a(R1, [X], R).
```

Listendifferenz, die: Differenz zwischen zwei Listen. Xs0 = [a,b,c], Xs = [c]. Die Differenz zwischen Xs0 und Xs beschreibt die Liste [a,b]. Achtung: Eine Listendifferenz ist selbst keine Liste!

Listenelement, das:

```
listenelement_von(E, [E|_Es]).
listenelement_von(E, [_F|Es]) ←
  listenelement_von(E, Es).
```

Listenkonstruktor, der: Struktur der Stelligkeit zwei mit Namen '.'.

Listenkopf, der: Erstes Argument eines Listenkonstruktors.

Listenrest, der: Zweites Argument eines Listenkonstruktors.

Liste, die: Datenstruktur zur Darstellung von Sequenzen.

Lösung, die: Eine oder alle Antwortsubstitutionen einer Anfrage.

Lösungsmenge, die: Die Menge aller Antwortsubstitutionen einer Anfrage, die Prolog finden *sollte.*

Lösungssequenz, die: Die von Prolog gefundenen Antwortsubstitutionen (mit Duplikaten, bei Nichttermination unvollständig).

Maschinencodeübersetzer, der: Implementierung einer *Prologmaschine*, um einen Faktor 3-30 schneller als *Emulatoren.* Nicht portabel.

matching, das: ein Spezialfall der Unifikation, wobei ein Term ein *Grundterm* ist.

MGU: most general unifier, *allgemeinster Unifikator.*

Mlips, pl. die: 1 000 000 lips

most general unifier: *allgemeinster Unifikator.*

named variable: Gegenteil zu anonymous variable.

Name, der: Namen bezeichnen Atome, Funktornamen und Prädikatsnamen. Namen werden als Atome geschrieben.

neck: *Regelatom.*

negation as finite failure, die: Sichtweise, die aus dem Scheitern eines Ziels folgert, dass das Ziel nicht wahr ist.

negative Zusicherung, die: sichert zu, dass ein Ziel nicht ableitbar ist.

Nichtterminalsymbol, das: Grammatiksymbol, das eine Liste von Termen beschreibt.

nil: Das Atom [ ]/0. Wird zur Darstellung der leeren Liste verwendet.

occur-check, der: Überprüfung während der *Unifikation*, um *infinite Terme* zu verhindern. Das Ziel X = f(X) sollte scheitern, weil durch einen occur-check festgestellt wird, dass die Variable X an einen Term gebunden wird, deren *Subterm* sie ist. unify_with_occurs_check/2.

partial list: *unvollständige Liste.*
Prädikatensymbol, das: Prädikatensymbol und Arität bilden den Prädikatsindikator.
Prädikation, die: Ein Prädikat der Stelligkeit n und eine Sequenz von n Argumenten. Der Regelkopf ist eine Prädikation genauso wie ein Ziel.
Prädikatsindikator, der: Schreibweise zur Bezeichnung eines Prädikats. Besteht aus Prädikatsname und der Stelligkeit des Prädikats verbunden mit /. Der Prädikatsindikator liste/1.
Prädikatsname, der: Prädikatensymbol.
Prädikat, das: Ein Prädikatsname gemeinsam mit einer Stelligkeit.
präskriptiv: Gegenteil zu *deskriptiv.*
Prinzipalfunktor, der: Funktor eines zusammengesetzten Terms.
Prologmaschine, die: Ausführungsmodell für Prolog. Prologsysteme werden als *Interpreter*, *Emulatoren*, Compiler nach C oder *Maschinencodeübersetzer* implementiert.
prozedural: *imperativ*
query: *Anfrage.*
redundante Lösung, die: Mehrfachlösung. Mehrfach vorkommende Antwortsubstitution. Redundante Lösungen treten in Lösungssequenzen auf.
Regelatom, das: :- bzw. ←. Verbindet *Kopf* und *Rumpf* einer *Regel.* Auch: rule atom, neck, Hals.
Regelkopf, der: linke Seite einer *Regel*, eine *Prädikation*, die auf der linken Seite einer Regel steht.
Regelrumpf, der: Rechte Seite einer Regel.
Regel, die: Spezielle Klausel mit nichtleerem Kopf und nichtleerem Rumpf.
Rest, der: *Listenrest.*
rule atom: *Regelatom.*
rule: *Regel*
Rumpf, der: *Regelrumpf.*
scheitern, zu: Ein *Ziel* scheitert, wenn es nicht *erfüllt* ist.
Seiteneffekt, der: Ergebnis der Ausführung eines Programms einer prozed. Programmiersprache.
shared variables: *gemeinsame Variablen.*
Stelligkeit, die: Anzahl der *Argumente* eines *zusammengesetzten Terms.* Bsp.: die Stelligkeit eines Prädikats, die Stelligkeit eines Funktors. (englische Bezeichnung: arity)
stratifiziertes Programm, das: Programm, das nur direkte Rekursionen enthält und keine Rekursionen über negative Ziele. (Negative Ziele werden erst im 2. Teil besprochen.)
String, der: Liste von Latin1-Zeichen.
Struktur, die: Eine Struktur besteht aus einem **Funktionssymbol** und einer Folge von Argumenten. Die Anzahl der Argumente, die Stelligkeit muss größer als Null sein. Eine Struktur wird meist durch ihren *Funktor* bezeichnet. (englische Bezeichnung: compound term)
Substitution, die: Variablensubstitution, *Variablenbindung.*
Subterm, der: @@@
succeed, to: *erfüllen.*
Terminalsymbol, das: Dient der Beschreibung eines konkreten Listenelements in einer Grammatik.
Term: Ein konstanter Term, ein *zusammengesetzter Term* oder eine Variable.
ungleich: Syntaktische Ungleichheit wird durch das Prädikat dif/2 ausgedrückt.
Unifikator: eine Variablenbindung, die zwei Terme gleich macht. Für ← f(X,A) = f(Y,b). ist A = b, X = j, Y = j ein Unifikator, jedoch nicht der *allgemeinste Unifikator.*
unifizierbar: Zwei Terme sind unifizierbar (können unifiziert werden), wenn es einen unifier für sie gibt.
unifizieren, zu: auch gleichsetzen. finden und anwenden eines allgemeinsten Unifikator für zwei Terme. Etwa durch das vordefinierte Prädikat =/2.
uninstanziert: Eine Variable ist uninstanziert, wenn sie nicht an einen Term gebunden ist.
unvollständige Liste, die: Liste, deren Rest noch eine freie Variable ist.
Variablenbindung, die:
Variablensubstitution, die: *Variablenbindung*
Variable, die: Ein Objekt, das an einen Term gebunden werden kann.
void variable: [engl.] *anonyme Variable.*
Wissensbasis, die: Datenbasis + Regeln.
Zeichenkette, die: String.
Ziel, das: zu beweisende Prädikation. Kommen in Anfragen und Regelkörper vor.
zusammengesetzter Term, der: Struktur.
zusammengesetzte Anfrage, die: Anfrage bestehend aus einer Konjunktion von Zielen.
Zusicherung, die: Anfrage. Sichert zu, dass ein Ziel ableitbar ist. Wird zum Testen von Programmen verwendet.
Zwischencodeinterpreter, der: *Emulator.*

Stichwortverzeichnis

A Die Übungsumgebung

Wenn Sie sich einloggen, sehen Sie zwei Fenster: Das linke enthält in einer Folge alle Beispiele, das rechte enthält Hilfetexte („Hinweise"). Wenn Sie ein Beispiel bearbeiten, fügen Sie nach der Beispielkennung (der Zeile `## x. Beispiel`) und den Angabezeilen (beginnen mit `#`) Ihre Lösung ein. In der Statuszeile erscheint die Nummer des Beispiels, das Sie gerade verändern und noch nicht abgesichert haben. Sie können natürlich auch andere Prädikate zusätzlich zu den Lösungen in ein beliebiges Beispiel schreiben.

A.1 Sichern und Laden

Um das Beispiel zu sichern, drücken Sie die Taste [Do] (rechts oben auf der Tastatur) oder M-Return. Dadurch wird das Beispiel nicht nur gesichert, sondern auch vom Compiler überprüft und eventuell geladen. Fehlermeldungen erscheinen als neu eingefügte mit einem Rufzeichen versehene Zeilen. Löschen Sie diese Zeilen nicht weg, sondern verbessern Sie nur das Programm und drücken Sie erneut [Do]. Die Fehlermeldungen werden bei jedem [Do] gelöscht und u.U. neu erzeugt. Wenn Sie Anfragen in das Beispiel schreiben, z.B. `:- kind_von(Kind,Elternteil).`, wird diese Anfrage bei jedem [Do] auf ihre Ableitbarkeit hin überprüft. Wenn die Anfrage scheitert, es also keine Antwort gibt, ist eine Fehlermeldung zu sehen. Auch wenn das System nach [Do] mit einer Fehlermeldung antwortet, ist das Beispiel gesichert.

A.2 Anfragen

Um eine Antwort, also die konkreten Antwortsubstitutionen einer Anfrage zu sehen, muss das Beispiel, in dem die Anfrage steht, mit [Do] geladen bzw. gesichert worden sein. Positionieren Sie den cursor auf den : des Pfeils und drücken Sie wiederum [Do]. Weitere Lösungen mit [Space]. Zur Unterbrechung der Anfrage, wenn Sie also nicht alle Lösungen sehen wollen, drücken Sie irgendeine andere Taste. Bedenken Sie bitte, dass Sie beim Unterbrechen einer Antwort die Folge von Lösungen nur unvollständig sehen. Bei unendlichen Lösungsmengen ist dies natürlich immer so. Die Antworten erscheinen in Zeilen beginnend mit @@. Diese Zeilen werden beim nächsten Absichern wieder gelöscht, es sei denn, sie entfernen die @@ händisch. Dadurch bleiben die Antworten im Text erhalten. Üblicherweise ist es aber nicht interessant, diese Zeilen zu behalten, da sie ja ohnehin jederzeit erzeugt werden können.

A.3 Fragen zur Übung, den Beispielen etc.

Wenn Sie sich bei einem Beispiel nicht auskennen, können Sie neben Ihren Kollegen und Ihrem Tutor auch innerhalb der Beispiele selbst (in der Übungsumgebung also) Fragen stellen. Dazu fügen Sie eine neue mit < beginnende Zeile ein. Schreiben Sie diese Zeile genau dort hin, wo Sie sich nicht auskennen und sichern Sie das Beispiel. Bitte bedenken Sie, dass die Fragen im System natürlich nicht sofort beantwortet werden können. Die

Fragen selbst werden *nicht* zur Benotung herangezogen. Sie gehen also kein Risiko ein, wenn Sie durch eine Frage „zugeben", dass Sie sich nicht auskennen.

A.4 Querverweise und Hinweise.

Ein **Hinweis** wird im Text mit zwei Pfeilen hervorgehoben. Do vor einem Verweis ↑↑Hinweis zeigt diesen Hinweistext im rechten Fenster an. Bitte beachten Sie, dass Hinweise auch gelegentlich in Fehlermeldungen zur genaueren Erklärung des Fehlers vorkommen. Sie können dann gleich mittels Do von der Fehlermeldung zur Erklärung des Fehlers gelangen. Wenn die zwei Pfeile vor einer Zahl stehen, dann bezieht sich der Verweis auf ein Beispiel. Do vor ↑↑27 geht etwa zum Beispiel 27.

A.5 Nachladen von Beispielen.

Wenn Sie sich einloggen, sehen Sie in Beispielen mit Programmtext eine Fehlermeldung, dass diese Beispiele nicht geladen sind. Bei jedem Einloggen müssen Sie sämtliche Beispiele nachladen, auf deren Definitionen Sie aufbauen.

Im folgenden verwenden wir in Beispiel 100 die Definition von wahr/0 aus Beispiel 90. Wenn wir nun nur das Beispiel 100 laden, so führt der Beweis der Zusicherung ← auchwahr. zu einem Fehler, weil auchwahr/0 das Prädikat wahr/0 verwendet.

```
## 90. Beispiel
! Dieses Beispiel wurde noch nicht geladen
wahr. % Definition des Prädikats wahr/0

## 100. Beispiel
auchwahr ←
   wahr. % Verwendung von wahr/0 aus Bsp. 90
← auchwahr.
! Prädikat wahr/0 nicht oder in nicht geladenem Beispiel definiert. ↑↑laden
```

Das Beispiel 90 muss hier also nachgeladen werden. Um zu dokumentieren, von welchen Beispielen das Beispiel 100 abhängt, verwende man das vordefinierte Prädikat geladen/1. In diesem Fall schreibe man die folgende Anfrage an den Anfang des Beispiels.

```
## 100. Beispiel
← geladen(↑↑90).
! Beispiel 90 nicht geladen. Mit DO DO C-DO laden!
```

Da Beispiel 90 noch nicht geladen ist, scheitert das Ziel geladen(↑↑90). Das Argument von geladen/1 ist in diesem Fall ein Querverweis. Sie gelangen nun, indem Sie Do vor ↑↑ drücken, direkt zum Beispiel 90 und können dieses mit Do nachladen. Mit C-DO (oder F17, C-x C-x) kehren Sie wieder zum Beispiel 100 zurück und können es nun fehlerfrei laden.

A.6 Prologsyntax

Die Übungsumgebung überprüft auch die Formatierung der Prädikate. Schreiben Sie jedes Faktum, jede Regel und jede Anfrage in eine neue Zeile. Die Klauseln eines Prädikats müssen in einer Folge definiert werden. Man darf also nicht Klauseln zweier Prädikate vermischen. Leerzeilen dienen zur Trennung von zwei Prädikaten. Die Ziele einer Regel müssen jeweils in eine eigene Zeile mit einem Tab eingerückt geschrieben werden. Durch diese Konvention ist es besonders einfach, einzelne Ziele in einer Regel zuzudecken. (Für Terminationsüberlegungen etc. wichtig). Zur Vereinfachung wird beim Schreiben einer Regel automatisch eingerückt.

A.7 Führung durch das System.

Über http://www.complang.tuwien.ac.at/ulrich/lp finden Sie neben allgemeinen Informationen zur Laborübung auch eine Führung durch das System.

B Tastaturbelegung

Die Programmierumgebung verwendet den Emacs-Editor. Standard-Emacs-Kommandos sind **fett gedruckt**.

C-a bedeutet Ctrl-a

M-a bedeutet Alt-a ebenso: ESC a, C-[a

Die 5 wichtigsten Kommandos

Do Sichert ein Beispiel;
Anfrage, wenn am : einer Zusicherung;
verfolgt Hinweis/Verweis, wenn vor/auf ↑↑.

Insert Verdoppelt aktuelle Zeile. Nützlich, um mehrere ähnliche Zusicherungen anzuschreiben. Statt Neueintippen, bestehende verdoppeln und modifizieren.

M-ä M-/ auf deutscher Tastatur

M-/ Erweitert aktuellen Wortanfang (Wort links vom Cursor) zu Wort mit gleichem Anfang im Text. M-/ mehrmals hintereinander zeigt alternative Erweiterungen. Mit C-/ zurücknehmen. Beschleunigt das Eintippen langer oder Umlaute enthaltender Prädikatsnamen. Oft kommen diese Namen bereits im Angabetext vor. Es genügt: die ersten Buchstaben und M-/.

C-ä C-/ auf deutscher Tastatur

C-/ Undo. Nimmt Textänderungen zurück.

C-g Abbruch.

Umlaute: Compose-a oder **C-x 8 " a**

C-x C-c Übungsumgebung verlassen

C-a Zeilenanfang, **C-e** -ende

M-< Anfang, **M->** Ende der Beispiele

F4 Suche Beispiel

M-. Suche Definition eines Prädikats

C-s Suche - inkrementell, Abbruch mit C-g

C-s C-s letzte Suche nochmals

C-s ret Wortsuche

M-C-s Suche regulären Ausdruck

C-r Suche zurück

C-k Löscht bis zum Zeilenende

C-y Schreibt Gelöschtes zurück

C-d Löscht Zeichen unter Cursor

M-% Ersetzen

M-? letzte Erweiterung durch M-/

C-c . Syntax der Zeile erklären

C-c s Syntax der Zeile erklären

TAB Suche nächsten Verweis in Hinweisen

C-x o Cursor in anderes Fenster

C-x 2 Schirm aufteilen, mit **C-x 1** zurücknehmen

C-c C-p Ausdrucken

C-c C-m Systembeschwerde (an Sysadmin)

C-l momentane Zeile in Mitte des Fensters

C-0 C-l momentane Zeile an oberen Fensterrand

Shift-up Fenster eine Zeile hinauf verschieben

Shift-6 ↑

F2, F3 sucht nach oben/unten Prädikatsfunktor

F1 fügt letzten Funktor bei Cursor ein

C-c 4 Sprache Deutsch/Französisch/Englisch

C-t, M-t Zeichen, Wörter vertauschen

C-x C-t Zeilen vertauschen

Häufige Tastenkombinationen

Fakten einer Datenbasis/längere Wörter.
Z.B. für Relation kind_von/2. Erstes Faktum schreiben. Für Funktor des zweiten Faktums nur den ersten Buchstaben tippen (k) und M-/. Danach für weiteren Funktor: M-? (Meta-Shift-/)

Mit C-/ u.U. wieder zurücknehmen. Bei mehrfachem M-/ werden alternative Expandierungen angezeigt.

Beispiel Nachladen. Beispiel 2 nachladen: F4 2 Do C-x C-x

Falls ein Verweis ↑↑2 im Text vorhanden, Cursor vor ↑↑ setzen und: Do Do C-Do

Weitere Anfrage stellen. Cursor in Zeile der Anfrage, Insert, Kopie modifizieren.

Löschen von Antwortsubstitutionen. Do (also einfach neu Laden), oder C-/ (Undo)

Rückkehr zur Stelle vor Sichern. Falls nach Do ein Fehler auftritt oder eine Antwort einlangt, springt der Cursor genau dorthin. In den meisten Fällen ist dies auch sinnvoll, um den Fehler oder die Antwort zu sehen. Zur Textstelle vor dem Sichern mit C-x C-x zurückkehren.