PROLOG

Tutorium zur Vorlesung

„Datenbanken und Wissensrepräsentation“

(Prof. Dr. G. Büchel)

Stand: April 2010

Verfasser: Dipl.-Ing. (FH) Andreas W. Lockermann

Vorwort

Der Name PROLOG leitet sich aus den Begriffen PROGRAMMING und LOGIC ab. Prolog steht demnach für eine logische Programmiersprache, die sich an formaler Logik orientiert.

Prolog wurde Anfang der 70er Jahre von Alain Colmerauer an der Universität Aix-Marseille entwickelt. Logische Programmierung unterscheidet sich von prozeduraler Programmierung, die z.B. in C, C++ oder Java verwendet wird: Bei Prolog wird das Wissen anhand von Fakten und Regeln beschrieben. Mit welchen einzelnen Schritten dieses Wissen erlangt werden kann, wird von Prolog selbst festgelegt. Man spricht bei Prolog von einer nicht-prozeduralen Programmiersprache, die über eine Sammlung von Faktenwissen und Regeln verfügt. Fakten und Regeln werden auch Klauseln genannt. Prolog basiert auf formaler Logik.

(vgl. [PRO ACH 08, S. 1], [PRO BK 91, Vorwort] und [PRO GOL 08, S. 1])

Das Prinzip von Prolog ist es, Faktenwissen und Regeln zu erfassen, welche der Anwender durch Anfragen/Behauptungen abfragen kann. Zusammengefasst unterscheidet man zwischen drei Tätigkeiten:

- Fakten deklarieren
- Regeln definieren
- Fragen stellen [PRO ACH 08, S. 1]

Dieses Skript soll eine einfache und praxisgerechte Einführung in die Programmiersprache Prolog liefern. Es besteht aus 12 Kapiteln. Das erste Kapitel dient der Einführung in die Grundlagen der logischen Programmierung. Das zweite Kapitel befasst sich mit dem Aufbau von Prolog. Im darauf folgenden Kapitel werden Syntax, Strukturen und Operatoren von Prolog beschrieben. Das Arbeiten mit sowie die Arbeitsweise von Prolog werden in Kapitel 4 erläutert. Kapitel 5 und 6 beschäftigen sich mit der Rekursion und mit Listen in Prolog. Die Ein- und Ausgabe auf der Konsole, das dynamische Verändern der Wissensbasis und das Ein- und Auslesen von Dateien werden in Kapitel 7 vorgestellt. Es gibt Möglichkeiten, die Arbeitsweise von Prolog zu beeinflussen. Mit diesen Verfahren beschäftigt sich Kapitel 8. Logische Rätsel können in Prolog mittels geeigneter Modellierung gelöst werden: Kapitel 9 erläutert anhand eines Beispiels das Lösen von Alphametics.

Kapitel 10 behandelt die Syntaxprüfung von Satzphrasen, indem geprüft wird, ob eine Satzphrase die Wortfolge betreffend grammatikalisch korrekt gebildet ist. In Kapitel 11 wird die Schnittstelle JPL vorgestellt, mit der es möglich ist, ein Prolog-Programm aus einem Java-Programm heraus aufzurufen. Andersherum ist es auch möglich, Java-Klassen und Methoden, die während der Laufzeit gefunden werden können, von einem Prolog-Programm aus aufzurufen. Eine kurze Bedienungsanleitung des SWI-Prolog befindet sich in Kapitel 12.

Der Inhalt dieses Skripts orientiert sich hauptsächlich an folgenden Quellen: [PRO RÖH 07], [PRO KS 03], [PRO BK 91], [PRO CM 87]), [JPL SI 09], [SW WI 09] und [WI BHS 07]. Weitere verwendete Quellen: siehe Literaturverzeichnis.

Köln, den 16. April 2010

Andreas W. Lockermann

Inhaltsverzeichnis

1 Logik

Eine Definition des Gegenstandes der Logik aus Sicht von Informatikern lautet: „Gegenstand der Logik ist die Darstellung von Wissen durch Formeln eines geeigneten Logikkalküls und die Herleitung von neuem Wissen auf Basis geeigneter Schlussregeln“ [WI BOE 07, S. 93]. Die Programmiersprache Prolog ist eine logische Programmiersprache und basiert auf einem Logikkalkül. Aus diesem Grund werden im folgenden Abschnitt kurz die Grundlagen der logischen Programmierung erläutert, die für das Verständnis von Prolog eine wichtige Rolle spielen.

1.1 Aussagenlogik

Die Aussagenlogik befasst sich mit logischen Aussagen, die durch Junktoren (logische Operatoren) verknüpft werden. Diese Junktoren sind das UND, dargestellt als „$\wedge$“, das ODER, dargestellt als „$\vee$“ sowie das NICHT, dargestellt als „$\neg$“. Zudem gibt es noch die Implikation „$\Rightarrow$“.

Mit den genannten Junktoren können Aussagen verknüpft werden. Diese Aussagen können entweder wahr oder falsch sein. Oft erfolgt die Auswertung aussagenlogischer Formeln mit Hilfe von Wahrheitstafeln.

A	**$\neg$A**
0	1
1	0

Abbildung 1.1: Wahrheitstafel I

A	**B**	**A $\vee$ B**	**A $\wedge$ B**	**A $\Rightarrow$ B**
0	0	0	0	1
0	1	1	0	1
1	0	1	0	0
1	1	1	1	1

Abbildung 1.2: Wahrheitstafel II

Abbildung 1.1 und Abbildung 1.2 zeigen zwei Wahrheitstafeln:

- Ist die Aussage A wahr, ist die Aussage ¬A falsch (und umgekehrt).
- Die Aussage A ∨ B ist immer dann wahr, wenn A ODER B wahr ist oder wenn A UND B wahr ist.
- Die Aussage A ∧ B ist nur dann wahr, wenn A wahr UND B wahr ist.
- Die Implikation in Abbildung 1.2 ist wie folgt zu verstehen: Folgt B aus A?

  Wenn man z.B. A="Es regnet" und B="Die Straße ist nass" zuordnet, sind dies zwei Aussagen, die wahr oder falsch sein können:

  - Wenn es nicht regnet und die Straße nicht nass ist, lässt sich B aus A folgern: Es regnet nicht; daraus folgt, die Straße ist nicht nass.
  - Wenn es nicht regnet und die Straße nass ist, lässt sich auch B aus A folgern. Das liegt daran, dass „aus etwas Falschem Beliebiges gefolgert werden kann (lat. ex falso quodlibet)" [WI BHS 07, S. 95]. Die Straße muss nicht durch den Regen, sondern kann z.B. durch einen umgestürzten Wassereimer nass sein.

    Wenn es nicht regnet, kann also daraus gefolgert werden, dass die Straße entweder nass oder nicht nass ist – beide Folgerungen sind wahr.

  - Die Aussage „wenn es regnet, wird die Straße nicht nass", ist dagegen falsch. Denn wenn es regnet, muss die Straße nass sein, damit die Aussage wahr ist.

    Ist Aussage **A** wahr, Aussage **B** aber nicht, folgt nicht **A ⇒ B**.

  - „Es regnet" daraus folgt „Die Straße ist nass" (siehe Zeile 4 in Abbildung 1.2) ist wahr. Denn wenn es regnet, wird die Straße nass.

(vgl. [WI BHS 07, S. 94f])

Die Wahrheitstafel in Abbildung 1.3 zeigt die Äquivalenz zwischen **A ⇒ B** und **¬A ∨ B**:

A	**B**	**A ⇒ B**	**¬A ∨ B**
0	0	1	1
0	1	1	1
1	0	0	0
1	1	1	1

Abbildung 1.3: Wahrheitstafel III

Die Verknüpfung von **A** und **B** mit der Implikation oder mit dem NICHT und dem ODER ergibt in jeder Zeile das gleiche Ergebnis. Somit lässt sich abschließend die Äquivalenz zwischen beidem feststellen:

$$\mathbf{A \Rightarrow B \equiv \neg A \vee B}$$

(vgl. [WI BHS 07, S. 94f])

1.2 Prädikatenlogik

Die Prädikatenlogik ist eine formale Sprache und dient zur Darstellung von Wissen. Ein Prädikat ist eine Aussagenform für Aussagen, die den gleichen Aufbau haben; z.B. ist die Aussagenform *ist_ein_Mensch()* ein Prädikat. Durch Einsetzen eines Terms entsteht eine Aussage, *ist_ein_Mensch(*Müller*)* oder *ist_ein_Mensch(*Schmitz*)*. Zur Darstellung dienen atomare Formeln, die ebenso wie in der Aussagenlogik mit Junktoren verknüpft werden können. Atomare Formeln bestehen aus Prädikatensymbolen und Termen. Als Gliederungszeichen sind in der Prädikatenlogik eckige und runde Klammern zugelassen. Zudem ist die Verknüpfung atomarer Formeln mit Quantoren möglich. Man unterscheidet bei den Quantoren zwischen dem Allquantor, dargestellt als „∀“ und dem Existenzquantor, dargestellt als „∃“. Der Allquantor bedeutet „Für alle Individuen gilt...“, der Existenzquantor bedeutet „Es gibt ein Individuum, für das … gilt“ [WI BHS 07, S. 113].

Zunächst eine Definition für Terme:

Definition 1.01: Terme

Der Aufbau von Termen ist folgendermaßen definiert:

a) Jede Konstante ist ein Term

b) Jede Variable ist ein Term

c) Ist f ein n-stelliges Funktionssymbol und sind $t_1, \ldots t_n$ Terme,

   so ist auch $f(t_1, \ldots t_n)$ ein Term.

[WI BHS 07, S. 114]

Beispiel:

4

23

plus(x, 2*)*

mal(x, 5*)*

plus() und *mal()* stellen zweistellige Funktionssymbole da. „4“ und „23“ sind Konstanten. Alle Ausdrücke sind nach der Definition 1.01 Terme.

Definition 1.02: Formeln der Prädikatenlogik

Die Syntax von Formeln der Prädikatenlogik ist folgendermaßen definiert:

a) Ist P ein n-stelliges Prädikatensymbol und sind $t_1, \ldots t_n$ Terme, so ist auch $P(t_1, \ldots t_n)$ eine (atomare) Formel.

b) Für alle Formeln F und G sind auch die Konjunktion ($F \wedge$ G), die Disjunktion ($F \vee G$) und die Implikation ($F \Rightarrow G$) Formeln.

c) Für jede Formel F ist auch die Negation ($\neg F$) eine Formel.

d) Ist x eine Variable und F eine Formel, so sind auch ($\exists x\, F$) und ($\forall x\, F$) Formeln.

[WI BHS 07, S. 114]

Prädikatenlogische Formeln sind zum Beispiel:

ist_gerade(4)

*ist_gerade(*6*)*

$\forall x$ *ist_gerade(mal(x,* 2*))*

Das Prädikatensymbol *ist_gerade()* dient zur Darstellung der arithmetischen Eigenschaft „gerade“, *mal()* bedeutet analog die Multiplikation. Der Allquantor bedeutet in diesem Zusammenhang: „Für alle *x* gilt diese prädikatenlogische Formel“.

Es ist möglich, prädikatenlogische Formeln in eine Klauselform zu konvertieren:

Definition 1.03: Klauselform prädikatenlogischer Formeln

Ein Literal ist eine atomare Formel oder deren Negation. Eine Klausel ist eine Menge von Literalen.

Die folgende Darstellung einer prädikatenlogischen Form heißt Klauselform:

$\{ \{L_{1,1}, ..., L_{1,m1}\}, \ldots \{L_{n,1}, ..., L_{n,mn}\} \}$.

Dabei sind $L_{i,j}$ Literale. Sie entspricht der Formel

$\forall x_1, ... x_k((L_{1,1} \vee ... \vee L_{1,m1}) \wedge \ldots \wedge (L_{n,1} \vee ... \vee L_{n,mn}))$,

wobei $\{x_1, ... x_k\}$ die Menge der in $L_{i,j}$ vorkommenden Variablen ist.

[WI BHS 07, S. 122]

Dazu folgendes Beispiel:

Gegeben sind die Prädikate *Saeugetier(x), Sieht_Aus_Wie_Fisch(x)* und *Wal(x)*. Das Prädikat *Sieht_Aus_Wie_Fisch(x)* wird im Folgenden durch die Kurzschreibweise *SAWF(x)* dargestellt.

Ausgedrückt in prädikatenlogischen Formeln erhält man

$\forall x$ (*Saeugetier(x)* $\wedge$ *SAWF(x)* $\Rightarrow$ *Wal(x)*)

Die Umformung in eine Klausel ergibt

{ $\neg$*Saeugetier(x),* $\neg$*SAWF(x), Wal(x)* }

In einer Klausel werden die Prädikate, die mit dem Allquantor $\forall$ *x P(x)* verknüpft sind, in der Form { *P(x)* } notiert.

Der Implikationspfeil kann aufgrund der Äquivalenz $A \Rightarrow B \equiv \neg A \vee B$ (siehe Abbildung 1.3) ersetzt werden:

Gegeben ist eine logische Formel:

$\mathbf{A_1 \wedge A_2 \wedge \ldots \wedge A_n \Rightarrow H}$

Gelesen sagt man: Wenn A_1 und A_2 und … und A_n dann gilt H.

Da $(\mathbf{A_1 \wedge A_2 \wedge \ldots \wedge A_n \Rightarrow H}) \equiv (\neg(\mathbf{A_1 \wedge A_2 \wedge \ldots \wedge A_n}) \vee \mathbf{H})$

$\equiv (\neg\mathbf{A_1} \vee \neg\mathbf{A_2} \vee \ldots \vee \neg\mathbf{A_n} \vee \mathbf{H})$ gilt,

sieht die Klausel dazu wie folgt aus:

$\{\neg\mathbf{A_1}, \neg\mathbf{A_2}, \ldots, \neg\mathbf{A_n}, \mathbf{H}\}$

Ein Beispiel, um prädikatenlogische Formeln in die Klauselform zu konvertieren, sieht folgendermaßen aus:

$Informatiker($Stefan$) \wedge \forall x \, (Informatiker(x) \Rightarrow Kann_programmieren(x))$

Daraus ergeben sich die beiden Klauseln:

1) { *Informatiker(*Stefan*)* }
2) { $\neg$*Informatiker(x), Kann_programmieren(x)* }

Die beiden Klauseln in Klauselform dargestellt:

{ {*Informatiker(*Stefan*)* }, { $\neg$*Informatiker(x), Kann_programmieren(x)* } }

(vgl. [WI BHS 07, S. 122])

1.3 Hornklauseln

In der Logik unterscheidet man hauptsächlich zwischen drei Arten logischer Formeln, den Fakten, den Regeln und den Anfragen, die als Informationsträger dienen. Es existieren noch weitere logische Formeln. Für viele Anwendungen reicht es jedoch, zwischen diesen drei Formeln zu unterscheiden.

1) Ein Fakt, auch Faktum genannt, wird durch eine einelementige Klausel dargestellt: **{ A }**

   Beispiel: { *Saeugetier(*Delfin*)* }

   Dieser Fakt beschreibt den Delfin als Säugetier.

2) Eine Regel wird durch die logische Formel $A_1 \wedge A_2 \wedge \ldots \wedge A_n \Rightarrow H$ dargestellt.

   Gelesen sagt man: Wenn A_1 und A_2 und … und A_n dann gilt H.

   Da $(A_1 \wedge A_2 \wedge \ldots \wedge A_n \Rightarrow H) \equiv (\neg A_1 \vee \neg A_2 \vee \ldots \vee \neg A_n \vee H)$ gilt,

   sieht die Darstellung einer Regel als Klausel wie folgt aus:

   $\{\neg A_1, \neg A_2, \ldots, \neg A_n, H \}$

   Beispiel: Aus den prädikatenlogischen Formeln

   $\forall x$ (*Saeugetier(x)* $\wedge$ *SAWF(x)* $\Rightarrow$ *Wal(x)*)

   erhält man nach Umformung in eine Klausel:

   { $\neg$*Saeugetier(x),* $\neg$*SAWF(x), Wal(x)* }

3) Eine Anfrage wird wie folgt in die Wissensbasis eingegeben:

   { $\neg H_1, \neg H_2, \ldots, \neg H_n$ }

   Die folgende Anfrage ist die <u>negierte</u> Anfrage:

   „Folgt $H_1 \wedge H_2 \wedge H_n$ aus der Wissensbasis?“

   Die negierte Form in Klauseldarstellung nennt man <u>Zielklausel</u>.

Eine Wissensbasis stellt dem Interpreter Wissen zur Verfügung. Dieses Wissen kann mit Hilfe einer Zielklausel erfragt werden.

Beispiel: { ¬*Wal(*Delfin*)* }

Bei Betrachtung der drei Arten von logischen Formeln lässt sich eine Gemeinsamkeit feststellen: In jeder Formel existiert höchstens ein positives Literal. Klauseln, in denen höchstens ein positives Literal existiert, nennt man Hornklauseln.

Definition 1.04: Hornklauseln

„Eine Hornklausel ist eine Klausel mit maximal einem positiven Literal. Eine nicht leere Hornklausel ohne positives Literal heißt Zielklausel (engl. goal clause), eine nicht leere Hornklausel ohne negatives Literal heißt Faktum, eine Hornklausel mit positiven und negativen Literalen heißt Regel“ [WI BHS 07, S. 130].

Die Programmiersprache Prolog unterscheidet auch zwischen Fakten, Regeln und Zielklauseln.

Die Syntax von Fakten, Regen und Zielklauseln sieht in Prolog wie folgt aus:

1) Fakten: **A.**

   Fakten implizieren immer die Conclusio [true].

   Zum Begriff Conclusio:

   Bei der Implikation, zum Beispiel A ⇒ B, unterscheidet man zwischen der Prämisse und der Conclusio. In diesem Fall ist A die Prämisse und B die Conclusio.

   Beispiel: *saeugetier(*feldermaus*)*.

   *saeugetier(*delfin).

   *sAWF(*delfin*)*.

   *kann_fliegen(*fledermaus*)*.

2) Regeln: **H :- A_1, A_2, … A_n.**

Die Zeichen „:-“ stehen für eine nach links gerichtete Implikation. Die Regel besteht aus der Conclusio H und der Prämisse A, die in diesem Fall aus A_1, A_2, … A_n besteht.

Beispiel: *wal(X)* :- *saeugetier(X), sAWF(X).*

3) Zielklauseln: **?- H_1, H_2, …, H_n.**

Der Prolog-Interpreter stellt Anfragen nicht in der Form „:-“ sondern in der Form „?-“.

Beispiele: ?- *saeugetier(*fledermaus*)*.

⇒ *true*, denn die Aussage ist wahr.

?- *wal(*delfin*)*.

⇒ *true*, denn die Aussage ist wahr.

?- *wal(*fledermaus*)*.

⇒ *false*, die Aussage ist falsch.

Abschließende Definition 1.05:

„Ein logisches Programm ist eine endliche Menge von Regeln und Fakten“ [WI BOE 07, S. 130].

(vgl. [WI BHS 07, S. 129])

2 Aufbau von Prolog

2.1 Wissensbasis – Interpreter – Anwender

Das Prinzip von Prolog besteht darin, Wissen für den Anwender in Form von Fakten und Regeln bereitzustellen. Der Anwender kann dieses Wissen mit Hilfe von Anfragen an das Prolog-Programm entnehmen. Prolog versucht, die Anwenderanfragen mit einem Interpreter, der mit Hilfe von verschiedenen Methoden auf das Wissen zugreift, zu beantworten.

Ein Prolog-Programm besteht im Wesentlichen aus zwei Teilen. Wir unterscheiden in unserem Quellcode dabei zwischen Fakten und Regeln, den sogenannten Klauseln. Fakten und Regeln bezeichnen wir als Wissen, welches ein Bestandteil des Quellcodes ist. Zusammengefasst spricht man von einer Wissensbasis, die diese Fakten und Regeln enthält:

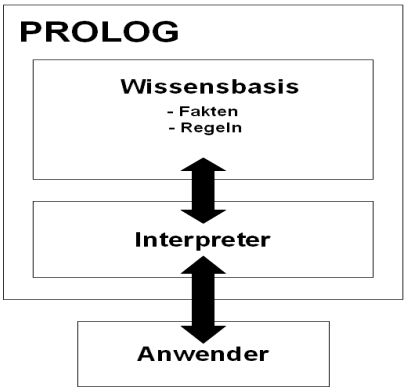

Abbildung 2.1: Modell eines Prolog-Programms

Die Abbildung zeigt das Modell der Prolog-Programmierung. Im oberen Teil sieht man die Wissensbasis, mit ihr verknüpft steht der Interpreter. Der Anwender stellt nun Anfragen, die entweder wahr oder falsch (***true*** oder ***false***) sein können. Der Interpreter erzeugt für die Anfrage auf Grundlage der Wissensbasis eine Folge logischer Ableitungen. Daraufhin erhält der Anwender eine Antwort auf seine Frage: Entweder ein *true*, das heißt, die Anfrage bzw. die aufgestellte Behauptung konnte erfolgreich abgeleitet werden und ist somit wahr. Andernfalls liefert Prolog ein *false* zurück.

(vgl. [PRO KS 03, S. 10])

2.2 Fakten

Der erste Bestandteil der Wissensbasis sind die Fakten. Sie dienen Prolog als Grundlage und stehen im Zusammenhang mit den Regeln, die unter 2.3 näher erläutert werden.

Im Folgenden wird anhand des Beispiels Familienbeziehungen („Familienbeziehungen.pl“) der Aufbau der Fakten beschrieben.

Wir betrachten dazu die Familie Waldmann in einem Baumdiagramm:

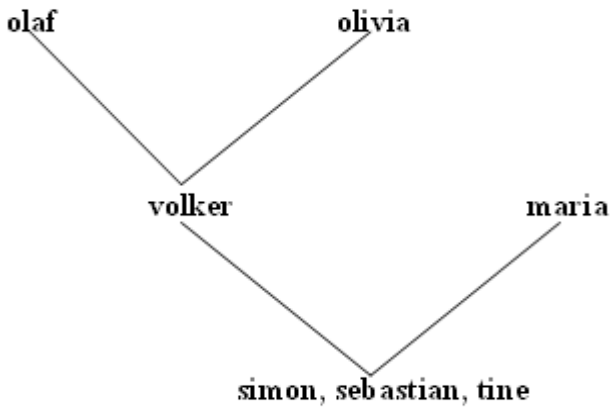

Abbildung 2.2: Stammbaum der Familie Waldmann

Das Ziel ist jetzt, die Informationen, die dieser Stammbaum über die Familie Waldmann liefert, als Fakten in Prolog darzustellen.

Bei der Syntax von Prolog ist besonders auf Groß- und Kleinschreibung zu achten:

Atome sowie Fakten- und Regel-Funktoren müssen mit einem Kleinbuchstaben beginnen, Variablen mit einem Großbuchstaben. Genaueres zur Syntax folgt in Kapitel 3.

Zur Darstellung der Informationen des Stammbaums unterscheiden wir in diesem Beispiel zunächst zwischen männlich und weiblich. Die Personen *olivia*, *maria* und *tine* werden als weiblich, *olaf*, *volker, simon* und *sebastian* als männlich dargestellt.

Die folgende Abbildung zeigt, wie sich diese Fakten in Prolog darstellen lassen:

```
weiblich(olivia).
weiblich(maria).
weiblich(tine).

maennlich(olaf).
maennlich(volker).
maennlich(simon).
maennlich(sebastian).
```

Abbildung 2.3: Fakten I zum Stammbaum der Familie Waldmann

Jeder Fakt besteht aus einem Funktor, hier *maennlich*, und einem oder mehreren Argumenten. Bei den Fakten aus Abbildung 2.3 existiert jeweils ein Argument zu einem Funktor.

Die Wissensbasis besteht jetzt aus sieben Fakten. Daraus lässt sich schließen, dass z.B. *olivia weiblich* und *sebastian maennlich* ist. Die genauere Betrachtung der Syntax wird in Kapitel 3 behandelt.

Als nächstes ist es sinnvoll, die Mutter-Kind- bzw. die Vater-Kind-Beziehungen zu erfassen. Dazu betrachten wir erneut den Stammbaum in Abbildung 2.2: *olaf* ist der Vater von *volker* und *olivia* ist die Mutter von *volker*. Analog dazu lassen sich auch die Beziehungen zu den anderen Personen des Familienstammbaums ausdrücken. Abbildung 2.4 zeigt die Darstellung dieser Fakten in Prolog:

```
mutter(olivia, volker).
mutter(maria, simon).
mutter(maria, sebastian).
mutter(maria, tine).

vater(olaf, volker).
vater(volker, simon).
vater(volker, sebastian).
vater(volker, tine).
```

Abbildung 2.4: Fakten II zum Stammbaum der Familie Waldmann

Hier haben wir Fakten mit jeweils zwei Argumenten. Die Wissensbasis hat nun alle wichtigen Fakten, die wir für den Stammbaum der Familie Waldmann benötigen werden, erfasst.

(vgl. [PRO RÖH 07, S. 8f])

2.3 Regeln

Der zweite Teil der Wissensbasis besteht aus Regeln. Diese Regeln werden vom Programmierer aufgestellt. Stellt ein Anwender Anfragen an das Programm, versucht der Interpreter die Anfrage durch die Fakten und die Regeln abzuleiten, um zu überprüfen, ob die Anfrage, das heißt die Behauptung des Anwenders, wahr oder falsch ist.

Das Beispiel für eine einfache Regel knüpft wieder an den Stammbaum der Familie Waldmann an: Die Regel *elternteil* in der folgenden Abbildung ist erfüllt, wenn es sich bei *Elter* um einen Elternteil eines Kindes handelt. Dazu wird geprüft, ob *Elter vater* oder *mutter* eines Kindes (*Kind*) ist.

```
elternteil(Elter, Kind) :- vater(Elter, Kind);
                           mutter(Elter, Kind).
```

Abbildung 2.5: Regel elternteil zum Stammbaum der Familie Waldmann

Eine Regel besteht aus einem Regelkopf, hier *elternteil(Elter, Kind)* und einem Regelrumpf, hier *vater(Elter, Kind)*; *mutter(Elter, Kind)*. In Prolog wird der logische ODER-Operator durch das Semikolon ausgedrückt.

Ein Anwender kann jetzt z.B. die Anfrage stellen, ob *olaf* ein Elternteil von *volker* ist. Prolog liefert in diesem Fall ein *true* zurück, das heißt die Aussage ist wahr, da diese abgeleitet werden kann: *olaf* ist laut den Fakten in Abbildung 2.4 der Vater (*vater*) von *volker*.

Eine zweite Regel für diesen Familienstammbaum sieht wie folgt aus:

```
sohn(Kind, Elter) :- maennlich(Kind),
                     elternteil(Elter, Kind).
```

Abbildung 2.6: Regel sohn zum Stammbaum der Familie Waldmann

In Abbildung 2.6 sind die zwei Bedingungen der Regel *sohn* mit einem Komma verknüpft: Das Komma ist in Prolog der logische UND-Operator. Diese Regel prüft, ob ein Kind ein Sohn von einem Elter ist.

Bei dieser Regel wird zunächst mit Hilfe der Fakten aus Abbildung 2.3 geprüft, ob das Kind *maennlich* ist. Danach wird auf die Regel in Abbildung 2.5 zurückgegriffen: Ist *Elter* ein Elternteil von dem angegeben Kind? Lässt sich beides erfolgreich ableiten, liefert Prolog auch hier ein *true* zurück.

Setzt man zum Beispiel für *Kind volker* und für *Elter olivia* ein, liefert Prolog ein *true* an den Anwender zurück.

Möchte man prüfen, ob *sebastian* der Sohn von *olivia* ist, setzt man für *Kind sebastian* und für *Elter olivia* ein: Diese Behauptung ist nicht wahr, denn sie kann nicht aus den Fakten und Regeln dieses Beispiels abgeleitet werden. Prolog liefert also ein *false* für falsch zurück.

(vgl. [PRO RÖH 07, S. 8f])

3 Syntax

3.1 Strukturen

Strukturen spielen in Prolog eine wichtige Rolle. Prolog bietet eine einfache Art von Strukturen an, die sogenannten Terme.

Bei Termen unterscheidet man zwischen *atomar*, *komplex* und *variabel*.

3.1.1 Atomare Strukturen

Eine atomare Struktur ist eine Zeichenfolge von Buchstaben und/oder Ziffern.

Eine Zeichenfolge ohne Buchstaben, die nur aus Ziffern besteht, wird als Zahl bezeichnet, ansonsten spricht man von Atomen. Es ist darauf zu achten, dass Zeichenfolgen mit Buchstaben immer mit einem kleinen Buchstaben beginnen müssen, sonst werden sie nicht als atomare, sondern als variable Struktur identifiziert.

Beispiele für atomare Strukturen:

```
olaf
olivia
12345
volkerWaldmann
volker1945
```

Abbildung 3.1: atomare Strukturen

Atomare Strukturen werden in Prolog auch als Konstanten bezeichnet.

3.1.2 Komplexe Strukturen

Komplexe Terme bestehen aus einem Funktor und aus einem oder mehreren Argumenten. Der Funktor muss ein Atom sein. Die Argumente werden hinter dem Funktor in runde Klammern eingeschlossen. Mehrere Argumente werden durch Kommata getrennt.

Abbildung 3.2 zeigt zunächst die komplexen Terme *weiblich* mit einem Argument sowie die komplexen Terme *mutter* mit zwei Argumenten.

```
weiblich(olivia).
weiblich(maria).

mutter(olivia, volker).
mutter(maria, simon).
```

Abbildung 3.2: Fakten mit einem bzw. zwei Argumenten

Die komplexen Terme mit einem Argument bedeuten in dieser Abbildung: *olivia* ist *weiblich* bzw. *maria* ist *weiblich*. Die komplexen Terme mit zwei Argumenten sind folgendermaßen zu verstehen: *olivia* ist die *mutter* von *volker* bzw. *maria* ist die *mutter* von *simon*. Demnach steht das erste Argument für den Namen der Mutter und das zweite für den Namen des Kindes.

Der allgemeine Aufbau eines komplexen Terms ist der folgende:

Funktor(*Argument*$_1$**,** *Argument*$_2$**, ...,** *Argument*$_n$**)**

Die vier Argumente in Abbildung 3.2 gehören zu den Prädikaten *weiblich* und *mutter*. Ein Prädikat lässt sich durch den Funktor (z.B. *weiblich*) und seine Stelligkeit (die Anzahl seiner Argumente) beschreiben.

Definition 2.01: Prädikat [PRO RÖH 07]

„Menge aller Klauseln mit gleichem Funktor und gleicher Stelligkeit. Beispiel: *elternteil/2* ist das zweistellige Prädikat *elternteil*.“ [PRO RÖH 07, Glossar, S. 164]

Ein Prädikat kann wie folgt beschrieben werden (Kurzschreibweise):

Funktor/N

Hinter dem Funktor folgt ein Schrägstrich gefolgt von *N*: *N* steht für die Anzahl der Argumente. Zu dem Prädikat *weiblich/1* gibt es gemäß Abbildung 3.2 z.B. die Klauseln (Regeln oder Fakten) *weiblich(olivia)* und *weiblich(maria)*.

(vgl. [PRO RÖH 07, S. 9])

Beispiel:

weiblich/1

Ein Argument darf wieder ein komplexer Term sein:

```
vater(sohn(volker, olaf), simon).
```

Abbildung 3.3: Fakt mit einem komplexen Term als erstes Argument

In Abbildung 3.3 ist Argument$_1$ *sohn(volker, olaf)* ein komplexer Term. Diese Abbildung ist wie folgt zu verstehen: *volker* ist *sohn* von *olaf* und *volker* ist *vater* von *simon*.

Abschließende Definition:

Definition 2.02: Prädikat [PRO BK 91]

„Ein Prädikat beschreibt einen Sachverhalt und wird durch einen Prädikatsnamen gekennzeichnet. Sofern es die Beziehung von Objekten beschreibt, werden diese als Argumente angegeben. Die Argumente werden in runde Klammern eingeschlossen und durch Kommata voneinander getrennt. Dabei ist die Position der Argumente von Bedeutung. Die Anzahl der Argumente bestimmt die Stelligkeit. Prädikate mit gleichem Prädikatsnamen und verschiedener Stelligkeit gelten als verschieden.“. [PRO BK 91, Glossar, S.321f]

3.1.3 Variable Strukturen

Variable Strukturen sind wie atomare Strukturen eine Zeichenfolge aus Buchstaben oder/und Ziffern. Der Unterschied besteht darin, dass das erste Zeichen entweder ein Großbuchstabe oder ein Unterstrich sein muss, damit Prolog die Zeichenfolge als Variable und nicht als atomare Struktur interpretiert.

Beispiele für variable Strukturen:

```
Kind
Elter
X
Y323
KarlHeinz
_abc
_
```

Abbildung 3.4: variable Strukturen

Der alleinige Unterstrich kennzeichnet eine anonyme Variable. Das heißt, die Variable muss aus syntaktischen Gründen gesetzt werden, der Anwender interessiert sich jedoch nicht für ihren Wert.

(vgl. [PRO KS 03, S. 15ff])

3.2 Punktnotation, Regeloperator und logische Operatoren

3.2.1 Punktnotation

In der Programmiersprache Prolog werden jeder Fakt, jede Regel und jede Anfrage an die Wissensbasis mit einem Punkt abgeschlossen. Dies ist vielleicht zunächst eine Umgewöhnung, da z.B. in der Programmiersprache Java ein Kommando mit einem Semikolon abgeschlossen wird. Der Punkt entspricht in anderen Programmiersprachen wie z.B. C, C++ oder Java dem Semikolon.

```
weiblich(olivia).
weiblich(maria).

mutter(olivia, volker).
```

Abbildung 3.5: Beispiel für die Punktnotation

Die Formulierung von Fakten erfolgt, wie in Abbildung 3.5 dargestellt, als komplexer Term, der durch einen Punkt abgeschlossen wird.

3.2.2 Regeloperator

Regeln werden in der Form

Regelkopf :- Regelrumpf.

formuliert. Der Doppelpunkt gefolgt von einem Minuszeichen wird als Regeloperator bezeichnet, er kann als „wenn“ ausgesprochen werden.

(vgl. [PRO BK 91, S. 22] und [PRO RÖH 07, S.9])

3.2.3 Logische Operatoren

3.2.3.1 UND

Der Regelrumpf besteht meistens aus mehreren Bedingungen. Diese Bedingungen können mit einem logischen UND oder einem logischen ODER verknüpft werden. Das logische UND, die Konjunktion, wird in Prolog mit dem Komma dargestellt:

```
tochter(Kind, Elter) :- weiblich(Kind),
                        elternteil(Elter, Kind).
```

Abbildung 3.6: tochter-Regel

Der Regelrumpf kann abgeleitet werden, wenn die Variable *Kind*, die man als Variable identifizieren kann, da sie mit einem Großbuchstaben beginnt, als *weiblich* aus den Fakten abgeleitet werden kann und die Regel *elternteil* ebenfalls abgeleitet werden kann. Können beide Bedingungen des Regelrumpfes erfüllt werden, ist die Regel ableitbar.

3.2.3.2 ODER

Das logische ODER, die Disjunktion, wird in Prolog durch ein Semikolon ausgedrückt. Betrachtet wird noch einmal die Regel *elternteil*:

```
elternteil(Elter, Kind) :- vater(Elter, Kind);
                           mutter(Elter, Kind).
```

Abbildung 3.7: elternteil-Regel

Diese Regel ist erfüllt, wenn die erste oder die zweite Bedingung des Regelrumpfes erfüllt werden kann.

3.2.3.3 NOT

Der NOT-Operator wird in Prolog durch einen umgekehrten Schrägstrich gefolgt von einem Plus-Zeichen dargestellt. Für das Beispiel „Stammbaum der Familie Waldmann“ ist es sinnvoll, z.B. eine Regel zu erstellen, die prüft, ob eine Person des Stammbaumes keine Kinder hat. Diese Regel lässt sich umsetzen, indem man überprüft, ob eine Person entweder Vater oder Mutter ist. Sie lässt sich aber auch mit dem NOT-Operator ausdrücken:

```
hat_keine_Kinder(Person) :- (maennlich(Person); weiblich(Person)),
                            \+ vater(Person, X),
                            \+ mutter(Person, X).
```

Abbildung 3.8: Regel hat_keine_Kinder

Die Person (*Person*) muss zunächst *maennlich* oder *weiblich* sein und weder *vater* noch *mutter* von einer anderen Person *X* des Stammbaumes sein.

Die Klammerung ist in Prolog auch möglich und muss hier eingesetzt werden, da das ODER schwächer als das UND bindet.

Analog dazu ist es auch möglich, eine Regel *hat_Kinder* aufzustellen:

```
hat_Kinder(Person) :- (maennlich(Person); weiblich(Person)),
                      vater(Person, X);
                      mutter(Person, X).
```

Abbildung 3.9: Regel hat_Kinder

Die Bedingungen dieser Regel existieren ohne den NOT-Operator.

(vgl. [PRO KS 03, S. 99f])

3.3 Arithmetik

3.3.1 Operatoren

In Prolog sind arithmetische Operationen möglich. Es besteht auch hierbei wieder ein Unterschied zu prozeduralen Programmiersprachen in der Syntax.

Die folgende Tabelle listet Standard-Rechenoperatoren auf, mit denen in Prolog gearbeitet werden kann:

X	+	*Y*	Addition von *X* und *Y*
X	-	*Y*	*Y* wird von *X* subtrahiert
X	*	*Y*	*X* wird mit *Y* multipliziert
X	/	*Y*	Real-Division von *X* durch *Y*
X	//	*Y*	ganzzahlige Division von *X* durch *Y*
X	*mod*	*Y*	gibt als Ergebnis den Rest der ganzzahligen Division aus

[PRO RÖH 07, S. 31]

Als nächstes sind Vergleichsoperationen wichtig:

X	=:=	*Y*	prüft, ob *X* nummerisch gleich *Y* ist
X	=\=	*Y*	prüft auf nummerische Ungleichheit
X	<	*Y*	ist *X* kleiner als *Y*
X	>	*Y*	ist *X* größer als *Y*
X	=<	*Y*	ist *X* kleiner oder gleich *Y*
X	>=	*Y*	ist *X* größer oder gleich *Y*

[PRO RÖH 07, S. 31]

Mathematische Funktionen wie z.B. *sqrt/1*, *log/1* oder *sin/1*, *cos/1*, etc. stellt SWI-Prolog zur Verfügung. Die Zahl Pi ist in Prolog als Konstante „*Pi*“ aufrufbar.

3.3.2 Mathematische Operationen

Die Wertzuweisung in Prolog erfolgt nicht durch das Gleichheitszeichen sondern durch das *is*. Die folgenden Beispiele zeigen, wie man in SWI-Prolog auf der Konsole einfache Rechenoperationen durchführen kann:

```
1 ?- X is 10 + 100.
X = 110.

2 ?- X is 15 - 10.
X = 5.

3 ?- X is 20 mod 10.
X = 0.

4 ?-
```

Abbildung 3.10: Mathematische Operationen I

Bei der ersten Anfrage weist das *is* der Variablen *X* das Ergebnis der Summe aus 10 und 100 zu. Anfrage zwei zeigt die Subtraktion zweier Zahlen. Anfrage drei benutzt das *mod*.

```
4 ?- X = 12 + 100.
X = 12+100.

5 ?- X is 12 + 100.
X = 112.

6 ?-
```

Abbildung 3.11: Mathematische Operationen II

Die Abbildung 3.11 zeigt zweimal die Addition der Werte 12 und 100 mit zwei unterschiedlichen Ergebnissen. Nur der zweite Fall mit dem *is*-Operator zeigt das gewünschte Ergebnis. Das Gleichheitszeichen „=“ hat eine andere Bedeutung und darf nicht mit dem *is* verwechselt werden: Das Gleichheitszeichen weist einem Ausdruck einen anderen zu und führt keine arithmetischen Operationen (hier: 12 + 100) aus.

Die folgende Abbildung zeigt, wie es möglich ist, mit dem Gleichheitszeichen und dem *is* das Ergebnis der arithmetischen Operation zweier Zahlen zu erhalten.

```
6 ?- X = 90 + 10, Y is X.
X = 90+10,
Y = 100.

7 ?-
```

Abbildung 3.12: Mathematische Operationen III

Die Variable *X* wird zunächst mit der Zahl „90“, dem „+“ und der Zahl „10“ belegt. *X* wird *Y* mittels *is* übergeben. Dann steht in der Variablen *Y* die Summe aus der Zahl „90“ und der Zahl „10“.

Im nächsten Beispiel wird ein Wert *Y* aus einem vom Anwender gegebenen Wert *X* berechnet. Es existieren drei Regeln (*kosten1*, *kosten2*, *kosten3*), zwischen denen der Anwender bei seiner Anfrage wählen kann.

```
kosten1(X, Y) :- X > 0,
                 X < 10,
                 Y is X mod 2.

kosten2(X, Y) :- X >= 10,
                 X <  100,
                 Y is X mod 20.

kosten3(X, Y) :- X >= 100,
                 Y is X mod 200.
```

Abbildung 3.13: kosten-Regeln 1, 2 und 3

Die drei Regeln unterscheiden sich durch den Wertebereich von *X*: Ruft der Anwender eine der drei gegebenen *kosten*-Regeln auf und liegt der eingegebene Wert der Variable *X* im definierten Bereich, wird ein Modulo berechnet. Das Ergebnis wird mittels *is* an die Variable *Y* übergeben und dem Anwender angezeigt. Die drei erstellten Regeln arbeiten mit mathematischen Vergleichsoperatoren, dem logischen UND und der Wertzuweisung durch das *is*.

Die folgende Abbildung zeigt Eingaben des Anwenders, aus denen Prolog anhand der Regeln *Y* errechnet:

```
1 ?- kosten1(3, Y).
Y = 1.

2 ?- kosten2(17, Y).
Y = 17.

3 ?- kosten2(100, Y).
false.

4 ?- kosten3(227, Y).
Y = 27.

5 ?- kosten3(603, 3).
true.

6 ?-
```

Abbildung 3.14: Anfragen zu den kosten-Regeln 1, 2 und 3

Dazu wird die jeweilige Regel mit einem Wert einer Variablen aufgerufen. Ein Punkt schließt die Abfrage des Anwenders ab.

Der Interpreter prüft in der gewünschten Regel, ob *X* die ersten beiden Bedingungen erfüllt und errechnet daraufhin den Modulo. *3 mod 2* ergibt *1* und *17 mod 20* ergibt *17*.

Die dritte Anfrage liefert dem Anwender ein *false*. Denn der *X*-Wert 100 liegt nicht im definierten Bereich von Regel 2. Die zweite Bedingung in Regel 2 kann somit nicht abgeleitet werden.

Bei Eingabe der Anfrage fünf erhält der Anwender keinen Wert für *Y*, sondern ein *true*: In der Anfrage behauptet der Anwender, dass *603 mod 200* in der Kosten-Regel 3 das Ergebnis *3* ergibt. Diese Behauptung ist richtig und Prolog liefert ein *true* zurück.

Die folgende Abbildung zeigt noch einmal die drei Regeln für die Kostenberechnung. Im Unterschied zu Abbildung 3.13 tragen alle drei Regeln den gleichen Namen (Regelkopf). Beim Aufruf muss der Anwender also nicht mehr unterscheiden, welche Regel er aufruft. Prolog entscheidet, welche Regel zum Beweisen der Anfrage ausgewählt werden muss. Da sich alle drei Regeln gegenseitig ausschließen, ist es nicht möglich, dass sich für *Y* zwei verschiedene Werte ergeben.

```
kosten(X, Y) :- X > 0,
                X < 10,
                Y is X mod 2.

kosten(X, Y) :- X >= 10,
                X <  100,
                Y is X mod 20.

kosten(X, Y) :- X >= 100,
                Y is X mod 200.
```

Abbildung 3.15: kosten-Regeln II

Die Abbildung 3.16 enthält die gleichen Anfragen wie in Abbildung 3.14. Allerdings bezieht sich der Anwender bei seiner Anfrage nicht mehr konkret auf eine Regel. Die Ergebnisse, die Prolog zurückgibt, entsprechen bis auf Anfrage Nummer 3 denen aus Abbildung 3.14.

```
1 ?- kosten(3, Y).
Y = 1 .

2 ?- kosten(17, Y).
Y = 17 .

3 ?- kosten(100, Y).
Y = 100.

4 ?- kosten(227, Y).
Y = 27.

5 ?- kosten(603, 3).
true.

6 ?-
```

Abbildung 3.16: Anfragen zu den kosten-Regeln II

In Abbildung 3.14 hatte *X* bei dieser Anfrage einen Wert, der durch die Regel 2 nicht abgeleitet werden konnte. Zu dieser Anfrage kann der Interpreter jetzt ein Ergebnis an den Anwender zurückgeben, da sich die Anfrage des Anwenders nicht mehr konkret auf eine Regel (Regel 2) bezieht, sondern auf alle Regeln des Quelltextes mit dem gemeinsamen Namen *kosten*. Die Regel *kosten* ist dreimal im Quelltext vertreten, somit gibt es für Prolog drei Möglichkeiten, die Anfrage abzuleiten. Vorher hatte der Interpreter nur die Möglichkeit, Regel 2 abzuleiten.

Abschließend lässt sich feststellen, dass es in Prolog möglich ist, mehrere Regeln mit gleicher Signatur (gleichem Regelkopf) aufzustellen. Der Interpreter hat dann mehrere Möglichkeiten, Anfragen zu beweisen und es ergeben sich eventuell verschieden lautende Antworten.

(vgl. PRO RÖH 07, S. 31 und S. 44])

4 Arbeiten mit Prolog, Arbeitsweise von Prolog

Nachdem in den letzten Kapiteln der Aufbau von Prolog und seine Syntax beschrieben wurden, handelt dieses Kapitel von der Benutzung und der Arbeitsweise des SWI-Prolog.

Im folgenden Kapitel werden sich die Beispiele auf das Beispielprogramm „Familienbeziehungen.pl“ beziehen.

4.1 Arbeiten mit dem SWI-Prolog

Als erstes muss der Quellcode von der Datei „Familienbeziehungen.pl“ konsultiert werden. Dazu wird in der SWI-Prolog-Kommandozeile der Befehl

consult('*Dateiname.pl*'**).**

eingegeben. Mit *consult* werden Fakten und Regeln aus einer Datei in die Wissensbasis geladen (kompiliert). Um die Beispieldatei „Familienbeziehungen.pl“ zu konsultieren hat der Befehl folgende Form:

consult('Familienbeziehungen.pl'**).**

Es erscheint folgende Ausgabe auf der Konsole des SWI-Prolog:

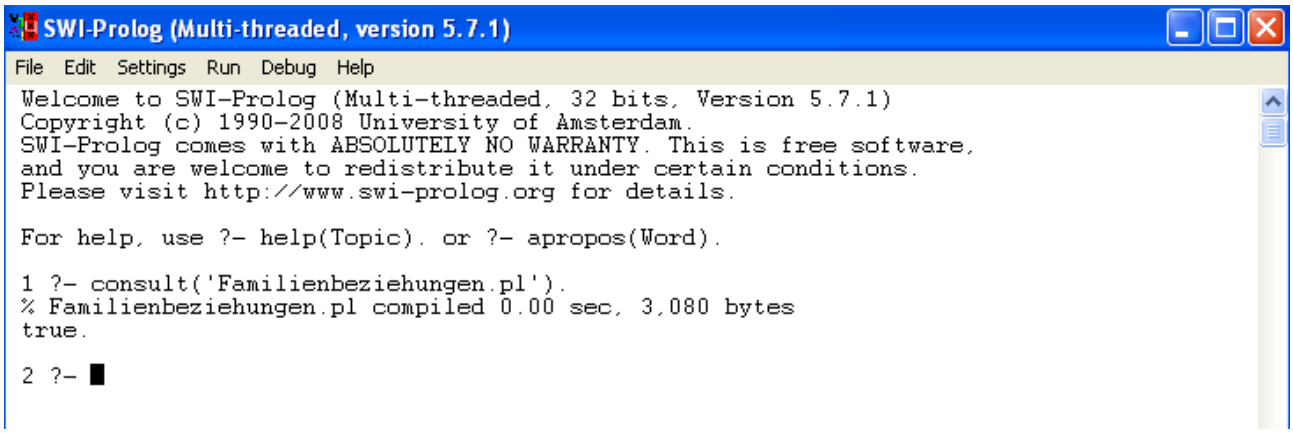

Abbildung 4.1: Konsultierung eines Prolog-Programms

Ist die Konsultierung wie in der Abbildung erfolgreich abgeschlossen, ist Prolog jetzt bereit, Anfragen (sogenannte Behauptungen) seitens des Anwenders entgegenzunehmen. Durch das Fragezeichen, gefolgt von einem Minuszeichen, signalisiert Prolog, dass der Interpreter bereit ist, Anfragen entgegenzunehmen.

Die folgende Abbildung zeigt zwei einfache Behauptungen/Anfragen des Anwenders:

```
2 ?- weiblich(olivia).
true.

3 ?- weiblich(olaf).
false.

4 ?-
```

Abbildung 4.2: Anfrage an das Programm "Familienbeziehungen.pl" I

Der Anwender stellt als erstes die Behauptung auf, dass *olivia* weiblich ist. Prolog liefert als Antwort ein *true* zurück. Jetzt weiß der Anwender, dass seine Behauptung wahr ist. Prolog konnte aus den Fakten der Wissensbasis, die in Kapitel 2 zu dem Stammbaum der Familie Waldmann in den Quellcode der Datei „Familienbeziehungen.pl“ eingetragen worden sind, schlussfolgern, dass die Behauptung wahr ist.

Bei der zweiten Behauptung des Anwenders liefert Prolog ein *false* an den Anwender zurück, denn *olaf* ist nicht als *weiblich* in der Wissensbasis eingetragen.

Als nächstes möchte der Anwender wissen, ob *volker* ein Elternteil von *tine* ist. Dazu greift er auf die Regel *elternteil* zu:

```
4 ?- elternteil(volker, tine).
true
```

Abbildung 4.3: Anfrage an das Programm "Familienbeziehungen.pl" II

Da laut der Wissensbasis *volker* als Vater von *tine* abgeleitet werden kann, liefert Prolog ein *true* zurück; die Aussage ist wahr.

Im nächsten Schritt möchte der Anwender nicht konkret nach einer Lösung suchen, sondern durch den Einsatz einer Variablen versuchen, mehrere Lösungen zu erhalten. Dazu stellt er dieselbe Anfrage an Prolog wie bisher, nur ersetzt er *volker* durch eine Variable mit dem Namen *X*. Er möchte somit von Prolog alle Elternteile von *tine* angezeigt bekommen. Nach Betätigung der Return-Taste erscheint aber zunächst nur eine Lösung für *X*:

```
5 ?- elternteil(X, tine).
X = volker
```

Abbildung 4.4: Anfrage an das Prolog-Programm "Familienbeziehungen.pl" III

Durch Eingabe eines Semikolons sucht Prolog nach einer weiteren Lösung für *X*. Diesen Vorgang, ein Semikolon zu setzen und Prolog aufzufordern, nach weiteren Lösungen zu suchen, kann man beliebig oft wiederholen, bis Prolog *false* an den Anwender zurückgibt. *false* bedeutet in diesem Fall, dass es keine weitere Lösung für *X* gibt.

```
5 ?- elternteil(X, tine).
X = volker ;
X = maria ;
false.

6 ?-
```

Abbildung 4.5: Anfrage an das Programm "Familienbeziehungen.pl" IV

Hinter ein *true* oder ein *false* setzt Prolog einen Punkt. Dieser Punkt bedeutet, dass der Dialog zwischen Anwender und Prolog-Interpreter abgeschlossen ist; es können keine weiteren Lösungsalternativen zur gestellten Anfrage seitens des Anwenders gestellt werden und es gibt keine weitere Antwort des Interpreters zu der gestellten Anfrage.

Somit springt der Eingabezeiger in die nächste Zeile hinter das Minuszeichen mit vorangestelltem Fragezeichen. Es können jetzt neue Anfragen an Prolog gestellt werden.

4.2 Arbeitsweise von Prolog

Dieser Abschnitt beschreibt die Arbeitsweise des Prolog-Interpreters mit den notwendigen Werkzeugen Resolution, Unifikation und Backtracking.

4.2.1 Resolution

Die Mechanismen eines Prolog-Interpreters beruhen auf der Aussagen- und Prädikatenlogik. Der Interpreter erzeugt für die Anfrage eines Anwenders auf Grundlage der Wissensbasis eine Folge logischer Ableitungen, wodurch der Anwender eine Antwort auf seine Frage erhält. Wie geht der Interpreter mit der Anfrage um und vor allem: Womit arbeitet dieser?

Der Prolog-Interpreter verwendet eine Implementierung der SLDNF-Resolution als Werkzeug, um die Anfragen eines Anwenders verarbeiten zu können. Resolution bedeutet „algorithmischer Test für die Unerfüllbarkeit einer Formel, der auf rein syntaktischen Umformungsregeln beruht“ [WI BHS 07, S. 104]. Bei mehreren Regeln, also einer Menge von Regeln, spricht man von einem Kalkül.

Die Resolutionsableitung besteht aus mehreren einzelnen Schritten, den sogenannten Resolutionsschritten. Gegeben sind zwei geeignete Formeln. Aus diesen wird eine dritte Formel abgeleitet. Diese wird als Resolvente bezeichnet und zur Formelmenge hinzugefügt.

Beispiel:

Gegeben sind zwei Klauseln: $\{A, B\}$ und $\{\neg A, C\}$.

Man setzt zum Beispiel A=“es regnet“, B=“Ich gehe ins Schwimmbad“

und C=“Ich gehe ins Kino“.

Als Implikation ($X \Rightarrow Y$ statt: $(\neg X) \vee Y$) gelesen kann man jetzt sagen:

„Wenn es nicht regnet, gehe ich ins Schwimmbad“.

„Wenn es regnet, gehe ich ins Kino“.

Die Resolvente aus beiden Klauseln ist {B, C}:

„Ich gehe ins Schwimmbad oder ins Kino“.

(vgl. [WI BHS 07, S.105])

Der Prolog-Interpreter durchläuft die Wissensbasis nach bestimmten Regeln und Verfahren, um eine Anfrage des Anwenders ableiten zu können. Diese Anfrage wird dazu negiert (Zielklausel). Ziel der Resolution ist es, „nach einer endlichen Anzahl von Resolutionsschritten eine unerfüllbare Formel abzuleiten“ [WI BHS 07, S. 104]. Damit ist die Unerfüllbarkeit der ursprünglichen Formelmenge gezeigt. Die Unerfüllbarkeit wird durch die leere Klausel ({} oder □) dargestellt, die sogenannte leere Resolvente (vgl. [INH RP 02, S. 572]), um damit einen Widerspruch zur Behauptung (Anfrage/Zielklausel) zu erhalten; ist dies erfolgt, liefert Prolog ein *true* an den Anwender zurück, womit die Anfrage erfolgreich war. Lässt sich die leere Resolvente nicht ableiten, liefert Prolog ein *false* an den Anwender zurück (vgl. [TI HO 09, S. 139]). Dieses Prinzip ist mit dem Widerspruchsbeweis in der Mathematik (indirekter Beweis) vergleichbar: Um etwas beweisen zu können, nimmt man an, dass die Behauptung falsch ist und versucht dies zu beweisen. Führt die Beweisführung zu einem Widerspruch, lässt sich schlussfolgern, dass die ursprüngliche Behauptung richtig war.

Die SLD-Resolution bedeutet „Select-Linear-Definite“-Resolution [INH RP 02, S. 572] und ist ein Beweisverfahren. Sie ist eine lineare Resolution für definite Klauseln mit Selektionsregeln (engl. computation rule). Eine definite Klausel ist eine Formel mit genau einem positiven Literal. Man unterscheidet dabei zwischen Fakten und Regeln (siehe Kapitel 1.3). Die SLD-Resolution gilt speziell für Hornklauseln. Hornklauseln bestehen aus definiten Klauseln und Zielklauseln.

Die SLDNF-Resolution ist die „SLD-Resolution plus Negation als Fehlschlag“ [INH RP 02, S. 574]. Diese spezielle Art der Resolution ist eine „Kombination von SLD-Resolution zur Ableitung positiver Literale und von Negation als Fehlschlag zur Ableitung negativer Literale“ [INH RP 02, S. 574]. Bei der SLDNF-Resolution gilt: „Für jedes positive Literal, das in einer SLDNF-Ableitung auftaucht, wird ein Beweis mit SLD-Resolution geführt, für jedes negative Literal ein separater Beweis durch Negation als Fehlschlag“ [INH RP 02, S. 575]. Ein negatives Ziel darf nur selektiert werden, wenn es variablenfrei ist. Erst wenn der Beweis eines negativen Literals beendet ist, darf

ein anderes Literal ausgewählt werden. Diese spezielle Form des Resolutionsprinzips wird verwendet, damit logische Schlüsse aus einem Programm, das heißt aus seiner Wissensbasis in Form von Fakten und Regeln, entnommen werden können.

(vgl. [INH RP 02, S. 571 bis 575])

4.2.2 Unifikation

In Kapitel 4.1 wurde verdeutlicht, wie ein Anwender Anfragen an ein Prolog-Programm stellen und Antworten entnehmen kann.

Es ist interessant zu wissen, wie Prolog mit Anfragen des Anwenders umgeht und nach einer Lösung sucht. Dazu wird in diesem Abschnitt die Unifikation erläutert. Der Begriff Unifikation bedeutet *etwas vereinen*, *gleichmachen*, *etwas miteinander identisch machen*. Der Unifikationsalgorithmus, der in diesem Abschnitt abgebildet ist, versucht, zwei Terme miteinander zu vereinen. Bei dem Algorithmus wird unterschieden, ob es sich dabei um atomare, variable oder komplexe Strukturen handelt und ob diese unifizierbar sind.

Die Anfrage in Abbildung 4.2 beantwortet Prolog mit *true*, denn genau dieser Fakt steht in der Wissensbasis. Bei einer Anfrage versucht Prolog also, diese mit einem Fakt oder einer Regel zu unifizieren. Dazu werden die Terme bezüglich Aufbau und Argumente miteinander verglichen. Ist die Unifikation z.B. mit einem Fakt möglich, ist die Anfrage ableitbar und Prolog liefert ein *true* an den Anwender zurück. Eine Regel kann, nachdem der Regelkopf mit der Anfrage unifiziert werden konnte, erfolgreich abgeleitet werden, wenn die Bedingungen im Regelrumpf erfolgreich abgeleitet werden können.

Beim Einsatz einer Variablen in Abbildung 4.4 benutzt Prolog ebenfalls die Unifikation: Prolog gibt dem Anwender die Information, durch welche atomare Struktur, dass heißt in diesem Beispiel, durch welche Person des Stammbaumes der Familie Waldmann die Variable *X* ersetzt (substituiert) werden kann, um die Anfrage erfolgreich ableiten zu können. Man spricht auch von der Instanzierung einer Variablen. Als Resultat kommen dafür entweder keine, eine oder mehrere Lösungen in Frage.

Bei der Unifikation untersucht der Prolog-Interpreter Strukturen bzw. Terme nach ihrem Funktor, ihrer Stelligkeit, das heißt der Anzahl und der Reihenfolge der Argumente.

Im Folgenden zeigt ein Unifikationsalgorithmus, wie zwei allgemeine Terme miteinander unifiziert werden können:

Eingabe: zwei Terme T_1 und T_2

Ausgabe: die Überdeckung T_3 der beiden Terme, falls die Überdeckung existiert; ansonsten ist die Antwort nein

Methode: Teste, welcher der drei folgenden Fälle für T_1 und T_2 zutrifft, und gib die zugehörige Überdeckung T_3 als Ergebnis zurück:

1) Ein atomarer Term unifiziert mit demselben atomaren Term

   falls T_1 und T_2 dieselben atomaren Terme sind, dann ist T_3 gleich T_1

2) Eine Variable unifiziert mit jedem Term

   a) falls T_1 und T_2 beide Variablen sind, dann ist T_3 gleich T_1 und T_1 gleich T_2

   b) falls T_1 eine Variable und T_2 ein nichtvariabler Term ist, dann sind T_3 und T_1 gleich T_2

   c) falls T_2 eine Variable und T_1 ein nichtvariabler Term ist, dann sind T_3 und T_2 gleich T_1

3) Unifikation komplexer Terme

   falls T_1 ein komplexer Term mit Funktor F_1, der Stelligkeit n und den Argumenten $A_{11}, A_{12}, \ldots, A_{1n}$ ist

   und T_2 ein komplexer Term mit Funktor F_2, der Stelligkeit m und den Argumenten $A_{21}, A_{22}, \ldots, A_{2m}$ ist

   und für F_1 und F_2 Fall 1 dieser Unifikationsvorschrift zutrifft und die Stelligkeiten gleich sind, d.h. $n=m$

und für alle Paare von Argumenten $<A_{11}, A_{21}>$, $<A_{12}, A_{22}>$, …, $<A_{1n}, A_{2m}>$ dieser Unifikationsvorschrift „Überdeckungen“ $T_{31}, T_{32}, \ldots, T_{3n}$ liefert, ohne dass die dabei nötigen Variablensetzungen mehr als einen Wert annehmen müssen

dann besteht T_3 aus dem Funktor F_1 und den Argumenten $T_{31}, T_{32}, \ldots, T_{3n}$.

Ansonsten lassen sich die beiden Terme nicht unifizieren.

[PRO KS 03, S. 20]

4.2.3 Backtracking

In Abschnitt 4.1 wurde erklärt, dass man Prolog nach dem Finden einer Lösung durch Eingabe eines Semikolons zur Suche nach einer weiteren Lösung auffordern kann. Dazu setzt Prolog Backtracking (dt. zurücksetzen) ein: Backtracking bedeutet, dass die bisher gefundene Lösung verworfen und zum letzten Alternativpunkt zurückgegangen wird, um nach einer alternativen Lösung zu suchen.

Um den Vorgang des Backtrackings genauer betrachten zu können, folgen in den nächsten Abbildungen Fakten und Regeln und ein vereinfachter Und-Oder-Beweisbaum:

```
weiblich(olivia).
weiblich(maria).
weiblich(tine).

maennlich(olaf).
maennlich(volker).
maennlich(simon).
maennlich(sebastian).


mutter(olivia, volker).
mutter(maria, simon).
mutter(maria, sebastian).
mutter(maria, tine).

vater(olaf, volker).
vater(volker, simon).
vater(volker, sebastian).
vater(volker, tine).

elternteil(Elter, Kind) :- vater(Elter, Kind);
                           mutter(Elter, Kind).

sohn(Kind, Elter) :- maennlich(Kind),
                     elternteil(Elter, Kind).
```

Abbildung 4.6: Wissensbasis zum Stammbaum der Familie Waldmann

Hier sind noch einmal die Fakten und Regeln zu dem Stammbaum der Familie Waldmann aus Kapitel 2 aufgelistet.

Im Folgenden wird die Regel *sohn* betrachtet:

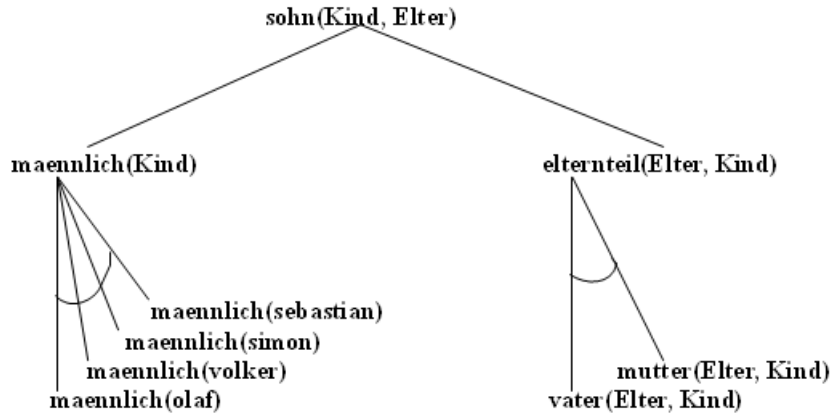

Abbildung 4.7: Und-Oder-Beweisbaum

Der abgebildete Und-Oder-Beweisbaum ([PRO RÖH 07, S. 12]) zeigt die Regel *sohn*. Die beiden Bedingungen für diese Regel sind *maennlich(Kind)* und *elternteil(Elter, Kind)*. Sie müssen beide erfüllt sein, damit die Regel *sohn* erfolgreich abgeleitet werden kann. Die Fakten, die *maennlich(Kind)* zugeordnet sind, sind Oder-verknüpft. Ebenso sind die Regeln *vater(Elter, Kind)* und *mutter(Elter, Kind)* Oder-verknüpft. Das heißt, ein Fakt unter *maennlich(Kind)* und eine der beiden Regeln unter *elternteil(Elter, Kind)* müssen erfüllt sein, damit die Regel *sohn* erfüllt ist.

Stellt der Anwender jetzt die Anfrage

```
1 ?- sohn(simon, X).
X = volker
```

Abbildung 4.8: sohn-Anfrage I

sucht Prolog die passenden Fakten und Regeln aus der Wissensbasis heraus und versucht damit, die Anfrage zu beweisen. Setzt der Anwender jetzt ein Semikolon, fordert er Prolog auf, nach einer neuen Lösung zu suchen. Dabei setzt Backtracking ein.

```
1 ?- sohn(simon, X).
X = volker ;
X = maria
```

Abbildung 4.9: sohn-Anfrage II

Backtracking bedeutet, der Prolog-Interpreter verwirft die gefundene Lösung und sucht nach einer Alternativlösung. Dazu geht der Interpreter an den letzten Alternativpunkt zurück, das heißt im Beweisbaum zu *elternteil(Elter, Kind)*. Dort war zuvor für die Lösung die Regel *vater(Elter, Kind)* verwendet worden. Jetzt schlägt der Interpreter einen anderen Weg ein: Mit der Regel *mutter(Elter, Kind)* wird nach einer Alternativlösung gesucht. *Kind* ist immer noch mit „simon“ initialisiert, *Elter* wird mit Hilfe des Fakts *maria* unifiziert.

Als Ausgabe auf der Konsole wird dem Anwender jetzt *maria* angezeigt. Setzt der Anwender wieder ein Semikolon, erscheint folgende Ausgabe:

```
1 ?- sohn(simon, X).
X = volker ;
X = maria ;
false.

2 ?-
```

Abbildung 4.10: sohn-Anfrage III

Prolog liefert ein *false*. Der Interpreter verwarf auch die Lösung *maria* und hat durch Backtracking versucht, eine Alternativlösung zu finden. Erfolglos.

An diesem einfachen Beispiel wurde das Schema des Backtracking verdeutlicht. Der Beweisende, in diesem Fall der Prolog-Interpreter, versucht also, wie man es in dem Und-Oder-Beweisbaum sehen kann, an den Verzweigungspunkten den Beweis alternativ fortzusetzen.

In diesem Fall wurde gezeigt, dass der Anwender Prolog zum Backtracking auffordert.

Prolog benutzt Backtracking auch während der Suche nach einer Lösung immer dann, wenn der aktuelle Lösungsweg keine Lösung bietet: Prolog arbeitet die Datenbasis mit den Fakten und Regeln stur von oben nach unten ab. Führt der aktuelle Lösungsweg zu keiner Lösung (der

Interpreter gerät während der Suche nach einem Beweis in eine Sackgasse), setzt Backtracking ein. Lässt sich durch Backtracking keine Lösung finden, da laut der Wissensbasis keine Lösung existiert, liefert Prolog ein *false* an den Anwender zurück und signalisiert dem Anwender damit, dass die gestellte Anfrage nicht abgeleitet werden kann.

Zusammenfassung:

Die Mechanismen bzw. Methoden, die Prolog zur Lösungsfindung einsetzt, sind eng miteinander verknüpft. Alle drei Verfahren beruhen auf verschiedenen Algorithmen. Die Unifikation versucht, Terme miteinander zu vereinen. Backtracking setzt immer dann ein, wenn der aktuelle Lösungsweg in eine Sackgasse führt oder der Anwender weitere Lösungen anfordert. Die Resolution versucht die Anfrage eines Anwenders erfolgreich durch eine endliche Anzahl von Resolutionsschritten abzuleiten; dabei ist das Ziel, die leere Resolvente zu erhalten, denn genau dann liefert Prolog ein *true* an den Anwender zurück.

(vgl. [PRO RÖH 07, S. 12f])

5 Rekursion

Die Rekursion existiert in prozeduralen Programmiersprachen. Der Begriff Rekursion kommt aus dem Lateinischen (lat. recurrere) und bedeutet „zurücklaufen“ oder „zurückkehren“. Ein typisches Beispiel für die Verwendung der Rekursion in der prozeduralen Programmierung ist die Berechnung der Fakultät *n!*.

Rekursion ist auch in Prolog möglich. Es folgt ein Beispiel für die Rekursion in Prolog, das sich auf das Beispielprogramm „Familienbeziehungen.pl“ der Familie Waldmann bezieht.

```
weiblich(olivia).
weiblich(maria).
weiblich(tine).

maennlich(udo).
maennlich(olaf).
maennlich(volker).
maennlich(simon).
maennlich(sebastian).


mutter(olivia, volker).
mutter(maria, simon).
mutter(maria, sebastian).
mutter(maria, tine).

vater(udo, olaf).
vater(olaf, volker).
vater(volker, simon).
vater(volker, sebastian).
vater(volker, tine).

elternteil(Elter, Kind) :- vater(Elter, Kind);
                           mutter(Elter, Kind).


vorfahr(Person1, Person2):- elternteil(Person1, Person2).

vorfahr(Person1, Person2):- elternteil(Person1, PersonX),
                            elternteil(PersonX, Person2).

vorfahr(Person1, Person2):- elternteil(Person1, PersonX),
                            elternteil(PersonX, PersonY),
                            elternteil(PersonY, Person2).
```

Abbildung 5.1: Wissensbasis mit vorfahr-Regeln

Die Familie Waldmann ist durch die Fakten *weiblich*, *maennlich*, *mutter*, *vater*, sowie die Regel

elternteil abgebildet. Jetzt soll eine neue Regel mit dem Namen *vorfahr* in die Wissensbasis eingefügt werden. Diese Regel soll prüfen, ob *Person1* ein Vorfahr von einer *Person2* ist. Abbildung 5.1 stellt die Wissensbasis nach Einfügung der Regel *vorfahr* dar.

Zusätzlich zu den bereits bekannten Fakten aus Kapitel 2 wurden die Fakten *maennlich(udo)* und *vater(udo, olaf)* in die Wissensbasis eingefügt. Damit existieren nun insgesamt 4 Generationen (vgl. dazu Abbildung 2.2).

Ein Anwender möchte jetzt wissen, ob *olaf* ein Vorfahre von *tine* ist:

```
1 ?- vorfahr(olaf, tine).
true
```

Abbildung 5.2: vorfahr-Anfrage I

Prolog liefert dem Anwender ein *true* zurück. Die Behauptung des Anwenders ist also wahr: *olaf* ist ein Vorfahre von *tine*.

Nur warum?

Der Interpreter arbeitet die Wissensbasis zeilenweise von oben nach unten ab. Als erstes versucht er, die Anfrage mit der ersten Regel von *vorfahr* zu beweisen. Doch dieser Versuch scheitert, da *Person1* (in diesem Fall *olaf*) kein Elternteil von *Person2* (in diesem Fall *tine*) ist. Das stimmt, denn wie wir wissen, ist *olaf* der Großvater von *tine*.

Hier setzt Backtracking (siehe Kapitel 4.2.3) ein: Prolog erkennt, dass die Anfrage mit dieser *vorfahr*-Regel nicht bewiesen werden kann und geht zu dem Punkt zurück, von dem aus eine alternative Lösung möglich sein könnte. Diese Alternative ist die *vorfahr*-Regel Nummer zwei.

Da *olaf* ein Elternteil von *PersonX* und *PersonX* ein Elternteil von *tine* ist, lässt sich diese Regel ableiten und Prolog liefert *true* an den Anwender zurück. Prolog findet durch die Fakten heraus, dass man durch Instanzierung der Variablen *PersonX* mit *volker* beide Regeln von *elternteil*, die

sich in der *vorfahr*-Regel befinden, ableiten kann. Denn *olaf* ist *vater* von *volker*, somit auch *elternteil* von *volker*, *volker* ist *vater* von *tine* und somit ein Elternteil von *tine*.

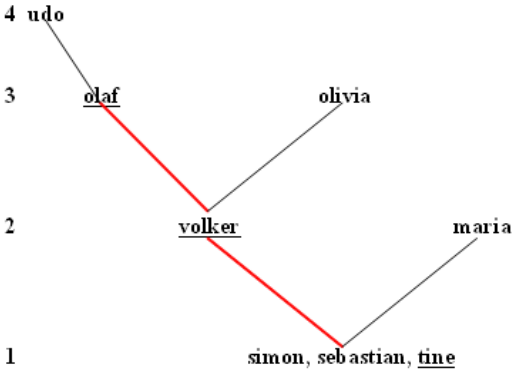

Abbildung 5.3: Familienstammbaum der Familie Waldmann II

Die erste *vorfahr*-Regel wird eingesetzt, um Vorfahrbeziehungen zwischen zwei Generationen zu beweisen. Die zweite *vorfahr*-Regel wird dagegen verwendet, um Vorfahrbeziehungen zwischen drei Generationen zu beweisen, wie z.B. die Vorfahrbeziehung zwischen *olaf* und *tine*. Analog dazu kommt bei vier Generationen die dritte *vorfahr*-Regel zum Einsatz.

Zusammenfassend lässt sich zu diesem Beispiel folgende Aussage festhalten: Es existieren drei *vorfahr*-Regeln, die benötigt werden, um diese Regel bei vier Generationen anwenden zu können. Analog dazu würden bei fünf Generationen vier *vorfahr*-Regeln benötigt etc. Bei großen Familienstammbäumen mit vielen Ahnen wäre allerdings der Aufwand zu groß.

Aus dieser Schlussfolgerung ergibt sich die Frage nach einem Verfahren, mit dem die Anwendung der *vorfahr*-Regel unabhängig von der Anzahl der Generationen möglich ist.

Um dieses Problem lösen zu können, setzt man Rekursion ein und ersetzt die vorherigen *vorfahr*-Regeln durch die beiden Regeln in der folgenden Abbildung:

```
vorfahr(Person1, Person2) :- elternteil(Person1, Person2).

vorfahr(Person1, Person2) :- elternteil(Person1, PersonX),
                             vorfahr(PersonX, Person2).
```

Abbildung 5.4: rekursive vorfahr-Regel

SWI-Prolog kennzeichnet in der zweiten Regel durch Unterstreichen, dass es sich um einen rekursiven Aufruf handelt Durch diese beiden Regeln erspart man sich viel Schreibarbeit!

Bei jeder Anfrage, bei der es um den Beweis von Vorfahrbeziehungen zwischen mehr als zwei Generationen geht, wird vorfahr-Regel Nummer zwei angewendet. Dabei wird rekursiv so lange die Regel *vorfahr* aufgerufen, bis *vorfahr*-Regel Nummer eins abgeleitet werden kann:

```
1 ?- spur(vorfahr(udo, simon)).
vorfahr(udo, simon)
   elternteil(udo, simon)
      vater(udo, simon)    nein
      mutter(udo, simon)    nein
vorfahr(udo, simon)
   elternteil(udo, _G584)
      vater(udo, olaf)
   vorfahr(olaf, simon)
      elternteil(olaf, simon)
         vater(olaf, simon)    nein
         mutter(olaf, simon)    nein
      vorfahr(olaf, simon)
         elternteil(olaf, _G619)
            vater(olaf, volker)
         vorfahr(volker, simon)
            elternteil(volker, simon)
               vater(volker, simon)
true
```

Abbildung 5.5: Anfrage vorfahr(udo, simon) mit Hilfe des spur/1-Prädikats

Die Abbildung 5.5 (Anfrage *vorfahr(udo, simon)*) mit Hilfe des *spur/1*-Prädikats [PRO RÖH 07, S. 18], veranschaulicht die Aufrufe, die beweisen, dass *udo* ein *vorfahr* von *simon* ist. Um zu beweisen, dass *udo* ein *vorfahr* von *simon* ist, werden in diesem Beispiel zwei rekursive Aufrufe benötigt.

Zum Abschluss des Kapitels der dementsprechende Aufruf im SWI-Prolog (ohne das *spur/1*-Prädikat):

```
2 ?- vorfahr(udo, simon).
true
```

Abbildung 5.6: vorfahr-Anfrage II

(vgl. [PRO RÖH 07, S. 9f])

6 Listen

Ebenso wie in der prozeduralen Programmierung gibt es in Prolog Listen. Die Rekursion spielt auch hier eine große Rolle. SWI-Prolog stellt dem Anwender einige nützliche Listenprädikate (sogenannte *Built-in predicates*) zur Verfügung, die das Arbeiten erleichtern. Einige dieser Listenprädikate werden in diesem Kapitel kurz vorgestellt.

Listen in Prolog können Listen von Konstanten, Variablen oder Termen sein oder auch Listen, die aus Listen bestehen.

Listen in Prolog werden in folgender Syntax angegeben:

[]

[a,b,c]

[udo,olaf,olivia]

[a|[b,c]]

[Kopf|Rest]

[[a,b,c]|d]

[vater(olaf, volker), mutter(olivia, volker)]

Die beiden zueinander eckig geschlossenen Klammern stellen eine leere Liste da. Einzelne Listenelemente sind durch Kommata getrennt.

Das Besondere bei Listen in Prolog ist der Listen-Operator oder auch Listenkonstruktor „|“. Dieser Listenkonstruktor teilt eine Liste in einen Kopf und einen Rest auf.

Für Operationen mit Listen ist dies wichtig zu wissen. Oft greift man rekursiv auf eine Liste zu, indem man zunächst den Kopf der Liste betrachtet und dann den Rest der Liste so lange aufruft, bis eine Abbruchbedingung eintritt. Eine solche kann z.B. die leere Liste ([]) sein.

Im folgenden Beispiel werden zwei Regeln gezeigt, die prüfen, ob ein Listenelement in einer Liste vorkommt:

```
gehoert_zu_Liste(Element, [Element|_]).

gehoert_zu_Liste(Element, [_|Rest]) :- gehoert_zu_Liste(Element, Rest).
```

Abbildung 6.1: Regeln gehoert_zu_Liste

Die erste Regel ist die Abbruchbedingung. Die zweite Regel arbeitet mit Rekursion und ruft *gehoert_zu_Liste* rekursiv auf.

```
1 ?- gehoert_zu_Liste(bruno, [adam, bert, bruno, wilhelm]).
true
```

Abbildung 6.2: gehoert_zu_Liste-Anfrage I

Bei dem Aufruf des Anwenders in Abbildung 6.2 liefert Prolog ein *true* zurück, das heißt, *bruno* gehört zu der angegebenen Liste.

```
2 ?- spur(gehoert_zu_Liste(bruno, [adam, bert, bruno, wilhelm])).
gehoert_zu_Liste(bruno, [adam, bert, bruno, wilhelm])
   gehoert_zu_Liste(bruno, [bert, bruno, wilhelm])
      gehoert_zu_Liste(bruno, [bruno, wilhelm])
true
```

Abbildung 6.3: gehoert_zu_Liste-Anfrage I mit Hilfe des spur/1-Prädikats

Die Abbildung 6.3 verdeutlicht mit Hilfe des *spur/1*-Prädikats [PRO RÖH 07, S. 18], wie Prolog zu diesem Ergebnis kommt: Nachdem ein Anwender die Anfrage gestellt hat, versucht Prolog, die Anfrage mit der ersten Regel von *gehoert_zu_Liste* zu beweisen. Da der Kopf der Liste *adam* aber nicht gleich *bruno* ist, lässt sich diese Regel nicht erfüllen. Somit versucht Prolog, die Anfrage mit der zweiten Regel abzuleiten. Diese zweite Regel ruft rekursiv *gehoert_zu_Liste* auf, allerdings nur mit dem Rest, also mit *bert*, *bruno* und *wilhelm*.

Damit beginnt Prolog wieder bei der ersten Regel: Diesmal wird *bert* mit *bruno* verglichen. Auch diese beiden Listeneinträge sind verschieden, so kommt es durch die zweite Regel wieder zur Rekursion und zum erneuten Aufruf der Restliste, die diesmal aus *bruno* und *wilhelm* besteht.

Wieder wird Regel 1 aufgerufen. Diesmal ist der Kopf der Liste *bruno* identisch mit dem gesuchten

Element *bruno*. Somit lässt sich die erste Regel ableiten und Prolog liefert dem Anwender ein *true* zurück.

Listenprädikate in Prolog entsprechen Standardmethoden zur Listenverarbeitung in prozeduralen Programmiersprachen (z.B. Methoden der Javaklasse *LinkedList*).

Die wichtigsten Prädikate werden im Folgenden kurz vorgestellt:

is_list(Liste)	Es wird geprüft, ob *Liste* eine Liste ist
member(Element, Liste)	Prüft, ob *Element* zur Liste *Liste* gehört
length(Liste, L1)	Die Variable *L1* gibt die Länge einer gegebenen Liste zurück. Für den Fall, dass nur die Länge *L1* gegeben ist, erzeugt Prolog eine Liste *L1* mit leeren Knoten.
sort(Liste, Sorted_Liste)	Sortiert *Liste* und gibt diese an *Sorted_Liste* zurück. Zusätzlich werden Duplikate entfernt.
msort(Liste, Sorted_Liste)	Liefert das gleiche Ergebnis wie *sort* mit dem Unterschied, dass Duplikate nicht entfernt werden.
append(Liste1, Liste2, Liste3)	Erstellung einer *Liste3*, die *Liste1* und *Liste2* so beinhaltet, dass die Argumente von *Liste1* vor den Argumenten von *Liste2* in *Liste3* stehen.

Term =.. Liste

Ein Term ist gegeben, z.B. der Term *vater(udo, olaf)*. Durch diesen Aufruf wird der Term in *Liste* geschrieben, also *Liste = [vater, udo, olaf]*. Das Gleichheitszeichen gefolgt von zwei Punkten ist der sogenannte *Univ-Operator.*

Dieses Prinzip funktioniert auch andersherum: Aus einer gegebenen Liste entsteht ein Term.

Beispiel:

Gegeben ist der Term mit zwei Argumenten:

vater(udo, olaf)

Mit Hilfe des sogenannten *Univ-Operators* kann dieser Term in eine Liste *X* umgewandelt werden:

```
3 ?- vater(udo, olaf) =.. X.
X = [vater, udo, olaf].

4 ?- 
```

Abbildung 6.4: Univ-Operator

Zusammenfassung:

Nach dem im Beispiel erläuterten Schema, den Kopf einer Liste zu betrachten und zu verarbeiten und den Rest der Liste rekursiv aufzurufen, bis z.B. die Liste leer ist, arbeiten die meisten Listenprädikate, die Prolog dem Programmierer zur Verfügung stellt. Man unterscheidet also zwischen einer Regel mit einer Abbruchbedingung, die erfüllt ist, wenn z.B. die Liste leer ist, und einer Regel, die sich selbst solange rekursiv aufruft, bis die Abbruchbedingung erreicht ist.

(vgl. [PRO KS 03, S. 37ff] und [PRO RÖH 07, S. 22 bis S. 28])

7 Die Ein- und Ausgabe und die dynamische Wissensbasis

SWI-Prolog bietet Möglichkeiten, von einem Anwender Eingaben von der Konsole einzulesen und Ergebnisse, die z.B. durch das Ableiten von Regeln bestimmt wurden, auf der Konsole auszugeben. Zudem ist es möglich, die Wissensbasis dynamisch zu erweitern. Des Weiteren wird in diesem Kapitel gezeigt, wie Daten exportiert und importiert werden können.

7.1 Ein- und Ausgabe auf der Konsole

Um einen Dialog mit einem Anwender führen zu können benötigt ein Prolog-Programm Eingaben des Anwenders und Ausgaben auf der Konsole, um dem Anwender Ergebnisse anzeigen zu können. SWI-Prolog stellt dem Programmierer dazu folgende Prädikate (*Built-in predicates*) zur Verfügung:

write(Term)	Gibt einen Term auf der Konsole aus. Alternativschreibweise: *writeln(Term)*. *ln* verursacht einen Zeilenumbruch nach der Ausgabe des Terms
read(Term)	Liest einen Term des Anwenders von der Konsole ein
get_char(Zeichen)	Liest das nächste Zeichen ein, das der Anwender eingibt

Möchte man z.B. mit *writeln* eine Ausgabe auf der Konsole machen und die Wörter mit einem Großbuchstaben beginnen lassen, lässt sich dies mittels folgender Syntax durchführen:

writeln('Hallo Familie Waldmann').

Würde man nicht die Hochkommata setzen, interpretiert Prolog ein mit Großbuchstaben beginnendes Wort als Variable.

Hierzu zwei Ausgaben:

```
1 ?- writeln('Hallo Familie Waldmann').
Hallo Familie Waldmann
true.

2 ?- writeln(Hallo).
_G271
true.

3 ?-
```

Abbildung 7.1: Ausgaben mit writeln

Im zweiten Fall interpretiert Prolog *Hallo* als Variable. Da diese Variable vorher nicht mit einem Wert belegt worden ist, erscheint keine sinnvolle Ausgabe.

Das folgende Beispiel zeigt einen Dialog zwischen einem Anwender und Prolog:

```
dialog :- writeln('Herzlich Willkommen'),
          writeln('Bitte geben sie einen Term ein:'),
          read(X),
          writeln('Vielen Dank'),
          writeln(X),
          writeln('Auf Wiedersehen').
```

Abbildung 7.2: Regel dialog

Durch die Anfrage *dialog* wird die abgebildete Regel abgeleitet:

```
1 ?- dialog.
Herzlich Willkommen
Bitte geben sie einen Term ein:
|: vater(udo, olaf).
Vielen Dank
vater(udo, olaf)
Auf Wiedersehen
true.

2 ?-
```

Abbildung 7.3: Konsolenausgabe zu der Anfrage dialog

Der Anwender wird aufgefordert, einen Term einzugeben. Nach der Eingabe des Terms erfolgt die Ausgabe des Terms auf der Konsole. Prolog liefert ein *true*, denn die Regel konnte erfolgreich abgeleitet werden.

(vgl. [PRO BK 91, S. 64] und [PRO RÖH 07, S. 40ff])

7.2 Die dynamische Wissensbasis

Die Wissensbasis, die im Quelltext erstellt worden ist, lässt sich dynamisch erweitern. Es können einzelne Fakten oder Regeln dynamisch hinzugefügt und auch wieder gelöscht werden. Dynamisch bedeutet, dass die erzeugten Fakten oder Regeln zur Laufzeit erzeugt werden. In der Regel gehen sie beim Beenden des SWI-Prolog wieder verloren.

Prolog bietet dem Programmierer folgende Prädikate (*Built-in predicates*) an:

asserta(X)	*X* wird an der vordersten Stelle der Wissensbasis eingetragen
assertz(X)	*X* wird an das Ende der Wissensbasis eingetragen
retract(X)	*X* wird aus der Wissensbasis gelöscht
retractall(X)	Alle Fakten oder Regeln mit dem Aufbau *X* werden gelöscht

Im Folgenden ein Beispiel für das dynamische Ändern einer Wissensbasis: Gegeben ist eine leere Wissensbasis, die um zwei Fakten der Familie Waldmann erweitert wird. Mit dem Aufruf *vater(volker, Kinder)* wird Prolog aufgefordert, nach einem Kind von Volker zu suchen. Als Suchraum dient die zuvor erstellte dynamische Wissensbasis. Nach der Ausgabe der gefundenen Kinder *tine* und *simon* werden die Fakten mit dem Aufbau *vater(volker, Kinder)* gelöscht.

```
1 ?- asserta(vater(volker, simon)).
true.

2 ?- asserta(vater(volker, tine)).
true.

3 ?- vater(volker, Kinder).
Kinder = tine ;
Kinder = simon.

4 ?- retractall(vater(volker, Kinder)).
true.

5 ?- vater(volker, Kinder).
false.

6 ?-
```

Abbildung 7.4: Dynamische Wissensbasis

Nachdem alle eingetragenen Fakten aus der Wissensbasis gelöscht worden sind, liefert die Anfrage *vater(volker, Kinder)* ein *false*.

Bei diesem Beispiel wurde nur auf der Konsole gearbeitet. Wird *asserta* oder eines der anderen drei Prädikate, die zur dynamischen Wissensbasis gehören, im Quelltext verwendet, muss der Programmierer den Term, der eingefügt werden soll, als dynamisch deklarieren. Die Syntax dazu sieht wie folgt aus:

:- dynamic Funktor/Stelligkeit.

Hinter dem Wort *dynamic* folgt der Name des Fakts oder der Regel, die eingefügt werden soll. Hinter dem Schrägstrich wird die Anzahl der Argumente angegeben (vgl. Kapitel 3.1.2). Der Punkt schließt die Deklaration ab.

Wird gemäß des Beispiels in Abbildung 7.4 *asserta* in einem Quelltext einer Regel verwendet, sieht die dynamische Deklaration des Fakts *vater* mit zwei Argumenten wie folgt aus:

:- dynamic vater/2.

(vgl. [PRO BK 91, S. 110f] und [PRO RÖH 07, S.58])

7.3 Dateien exportieren und importieren

SWI-Prolog liefert dem Programmierer die Möglichkeit, Dateien zu importieren sowie Dateien aus Prolog zu exportieren.

Anhand eines Beispiels werden zunächst der Export und danach der Import, bei dem Fakten in die Wissensbasis dynamisch eingefügt werden, erläutert.

7.3.1 Export

Zunächst werden zwei Prädikate (*Built-in predicates)* vorgestellt, mit denen Dateien zum Lesen oder Schreiben geöffnet und auch wieder geschlossen werden können:

open(Dateiname, Operation, Datei-Variable)

Mit dem Prädikat *open* wird eine Datei geöffnet. Für den Parameter *Dateiname* gibt der Programmierer den Dateinamen vollständig und mit Typ-Endung in Hochkommata an. Als *Operation* wird zwischen *read* und *write* unterschieden. Die *Datei-Variable* kann beliebig gewählt werden, muss allerdings mit einem Großbuchstaben beginnen.

close(Datei-Variable)

Mit *close* wird die Datei nach Beendigung wieder geschlossen. Die *Datei-Variable* von *open* und *close* muss identisch sein.

write(Datei-Variable, Inhalt)

Mit *write* können Inhalte in die Datei geschrieben werden. Die *Datei-Variable* muss identisch mit der von *open* und *close* sein. *Inhalt* ist beliebig.

Im folgenden Beispiel gibt es vier Fakten und drei Regeln.

Als erstes stellt der Anwender die Anfrage *exportieren(Dateiname)*. *Dateiname* steht z.B. für *'Familie_Waldmann.csv'* und wird vom Anwender an die Regel übergeben. Diese Regel öffnet zunächst diese Datei (siehe Abbildung 7.5, vgl. [PRO RÖH 07, S.40f]). Nachdem die Datei geöffnet worden ist, wird versucht, die Regel *familie_exportieren* abzuleiten.

```
vater(olaf, volker).
vater(volker, simon).
vater(volker, sebastian).
vater(volker, tine).

exportieren(Dateiname):- open(Dateiname, write, Datei),
                         familie_exportieren(Datei),
                         close(Datei).

familie_exportieren(Datei):- vater(Vater, Kind),
                             write(Datei, Vater),
                             write(Datei, ';'),
                             write(Datei, Kind),
                             nl(Datei),
                             fail.
familie_exportieren(_).
```

Abbildung 7.5: Regeln für den Export

Die Regel *familie_exportieren* versucht einen Fakt aus der Wissensbasis abzuleiten. Dieser Fakt wird mittels *write* in die Datei geschrieben. Die einzelnen Argumente des jeweiligen Fakts werden in die Datei geschrieben und durch Semikola getrennt. Das Prädikat *nl(Datei)* erzeugt einen Zeilenumbruch in der Datei. Das *fail*-Prädikat in der letzten Zeile dieser Regel verhindert, dass die Regel erfüllt werden kann (siehe dazu Kapitel 8.1). Somit setzt Backtracking ein: Der letzte Alternativpunkt ist der Fakt aus der Wissensbasis. Prolog sucht nach einer alternativen Lösung; findet Prolog eine alternative Lösung, wird auch dieser Fakt in die Datei geschrieben. Dieser Vorgang wird solange wiederholt, bis keine alternative Lösung mehr existiert. Diese Regel lässt sich dann nicht mehr weiter ableiten. So setzt Backtracking zum nächsten Alternativpunkt zurück, der Regel *familie_exportieren*. Diese Regel existiert zweimal. Da sich die erste der beiden gleichnamigen Regeln nicht mehr ableiten lässt, versucht der Prolog-Interpreter die zweite Regel abzuleiten. Da diese Regel keine Bedingungen enthält, lässt sich diese Regel ableiten. Jetzt wird die Datei in der Regel *exportieren* wieder geschlossen. Der Interpreter liefert nach dem Schließen der Datei ein *true* an den Anwender zurück.

```
1 ?- exportieren('Familie_Waldmann.csv').
true.

2 ?-
```

Abbildung 7.6: exportieren-Anfrage

Der Anwender hat in Abbildung 7.6 der Datei, die exportiert wird, den Namen „Familile_Waldmann.csv“ mit dem Aufruf (= Kommandozeile 1) gegeben. Abbildung 7.7 zeigt den Inhalt der erzeugten Datei:

Abbildung 7.7: Datei "Familie_Waldmann.csv"

Die CSV-Datei enthält genau die Fakten, die im Quelltext eingetragen sind. Jede Zeile steht für einen Datensatz, also einen Fakt. Die einzelnen Argumente eines Fakts werden durch Semikola getrennt.

(vgl. [PRO RÖH 07, S. 40f])

7.3.2 Import

Um Datensätze in Prolog importieren zu können, muss jeder einzelne Datensatz in seine Argumente zerlegt und anschließend dynamisch in die Wissensbasis eingefügt werden. Um den Import realisieren zu können, werden jetzt einige für den Import benötigte Prädikate (*Built-in predicates*) vorgestellt:

read_line_to_codes(Datei-Variable, Codes)

Dieses Prädikat liest eine Zeile aus einer Datei ein, die mit der *Datei-Variable* geöffnet worden ist. Alle Zeichen in dieser Zeile werden in der Form von ASCII-*Codes* eingelesen und an die Variable *Codes* in Form einer Liste übergeben. Jedes Element der Liste *Codes* stellt ein einzelnes Zeichen im ASCII-*Code* da.

atom_codes(Atom, Codes)

atom_codes bekommt *Codes* vom Prädikat *read_line_to_codes* übergeben und wandelt die Liste mit ASCII-*Codes* in ein Atom um. Demnach wird z.B. aus dem Semikolon, dargestellt als Zahl „59“, in der Liste *Codes* das Semikolon „;“. *Atom* enthält dann eine Zeichenkette wie sie z.B. in einer Zeile in der CSV-Datei in Abbildung 7.7 dargestellt ist.

concat_atom(Liste, 'Trennsymbol', Atom)

Dieses Prädikat erstellt eine Liste, die aus dem Inhalt der Variable *Atom* besteht: *Atom* wird auf das *Trennsymbol* hin untersucht und geteilt. Das Trennsymbol muss in Hochkommata angegeben werden.

at_end_of_stream(Datei)

Dieses Prädikat ist dann erfüllt, wenn nach der zeilenweisen Verarbeitung einer Datei der letzte Datensatz erreicht worden ist. Ist das Ende der Datei erreicht, ist dieses Prädikat erfüllt und kann abgeleitet werden.

Das folgende Beispiel knüpft an das Export-Beispiel an: Ziel ist es jetzt, die exportierte Datei „Familie_Waldmann.csv“ dynamisch in die Wissensbasis zu importieren.

Dazu der folgende Quelltext (vgl. [PRO RÖH 07, S. 40f]):

```
:- dynamic vater/2.

importieren(Dateiname):- open(Dateiname, read, Datei),
                         familie_importieren(Datei),
                         close(Datei).

familie_importieren(Datei):- at_end_of_stream(Datei).

familie_importieren(Datei):- read_line_to_codes(Datei, Codes),
                             atom_codes(Atom, Codes),
                             concat_atom(Liste, ';', Atom),
                             Liste = [Vater, Kind],
                             assertz(vater(Vater,Kind)),
                             familie_importieren(Datei).
```

Abbildung 7.8: Regeln für den Import

In der ersten Zeile wird der Term *vater* gemäß Kapitel 7.2 als dynamisch deklariert. Der Aufbau gleicht dem des Export-Programms: Eine Regel *importieren*, bei der der Anwender den Dateinamen übergibt, ruft eine Regel *familie_importieren* auf. Diese Regel existiert wie beim Export-Programm zweimal: Die erste Regel ist die Abbruchbedingung, die erfüllt ist, wenn alle Datensätze eingelesen sind. Die zweite Regel importiert zeilenweise eine Datei.

```
1 ?- importieren('Familie_Waldmann.csv').
true
```

Abbildung 7.9: importieren-Anfrage

Die Anfrage in Abbildung 7.9, die der Anwender tätigt, liefert ein *true* zurück. Daraus folgt, dass alle Datensätze der aufgerufenen CSV-Datei in die Wissensbasis dynamisch eingetragen worden sind.

Interessant ist es jetzt noch zu wissen, wie die Variablen *Code*, *Atom* und *Liste* inhaltlich aussehen. Dazu folgende Abbildungen:

```
[111, 108, 97, 102, 59, 118, 111, 108, 107, 101, 114]
```

Abbildung 7.10: Variable Codes

Abbildung 7.10 stellt den Inhalt der Variablen *Codes* dar. Es ist die Darstellung der ersten Zeile in ASCII-*Codes*, also des ersten Datensatzes der CSV-Datei. Zum Beispiel steht die Zahl „59“ für das Semikolon, das gemäß Abbildung 7.7 zwischen *olaf* und *volker* steht.

```
olaf;volker
```

Abbildung 7.11: Variable Atom

Abbildung 7.11 zeigt den Inhalt der Variablen *Atom*. *Atom* wird vom Prädikat *atom_codes* aus der Variablen *Codes* gebildet.

```
[olaf, volker]
```

Abbildung 7.12: Variable Liste

Liste enthält in diesem Beispiel zwei Elemente: Die zwei Argumente des Terms, der mit *asserta* gebildet wird.

Wurde *importieren* gemäß Abbildung 7.9 erfolgreich abgeleitet, kann der Anwender jetzt Anfragen an die Wissensbasis stellen:

```
2 ?- vater(volker, tine).
true.

3 ?- 
```

Abbildung 7.13: Anfragen an die dynamisch erweiterte Wissensbasis

Die Anfrage *vater(volker, tine)* kann erfolgreich abgeleitet werden, da der Fakt aus der CSV-Datei dynamisch in die Wissensbasis eingefügt worden ist.

(vgl. [PRO RÖH 07, S. 40f])

8 fail und der cut

In Kapitel 4.2 wurden die Verfahren vorgestellt, mit denen Prolog intern arbeitet. Das Backtracking ist eines dieses Verfahren. Prolog stellt dem Anwender zwei Prädikate (*Built-in predicates*) zur Verfügung, mit denen es möglich ist, Einfluss auf das Backtracking zu nehmen. Mit den Prädikaten, die in diesem Kapitel vorgestellt werden, ist es möglich, Backtracking, das heißt die Suche nach Alternativlösungen, zu erzwingen oder Backtracking zu unterbinden.

8.1 fail

Wie im Kapitel 4.2.3 beschrieben wurde, kann der Anwender oder der Programmierer Prolog auffordern, nach Alternativlösungen zu suchen.

Mit dem Einsatz des Prädikats *fail* fordert man Prolog auf, alle Alternativlösungen zu finden. Der Grund liegt darin, dass *fail* nicht erfüllt werden kann. Erreicht der Prolog-Interpreter in einer Regel das *fail*-Prädikat, ist er gezwungen, Backtracking einzusetzen, da *fail* nicht erfüllt werden kann.

Das *fail*-Prädikat lässt sich sinnvoll einsetzen, wenn Prolog ohne *fail* nur eine Lösung liefert, man aber alle möglichen Lösungen angezeigt bekommen möchte.

Beispiel:

```
ausgabe :- write('Der Stammbaum ').
ausgabe :- write('der Familie ').
ausgabe :- write('Waldmann').

start :- ausgabe.
```

Abbildung 8.1: Ausgabe-Regeln

Die Wissensbasis des abgebildeten Beispielprogramms enthält dreimal die Regel *ausgabe*. Stellt der Anwender die Anfrage „start.“, versucht Prolog *start* abzuleiten. Dazu muss die Regel *ausgabe* abgeleitet werden. Es gibt drei Möglichkeiten. Der Prolog-Interpreter entscheidet sich für die erste Ausgabe-Regel, die in der Wissensbasis steht, leitet diese ab und liefert ein *true* zurück.

Doch mit dieser Antwort ist der Anwender nicht zufrieden: Benötigt werden alle möglichen Ableitungen. Dazu kann der Anwender Prolog wie in Kapitel 4 mit der Eingabe von Semikola auffordern, nach Alternativlösungen zu suchen:

```
1 ?- start.
Der Stammbaum
true ;
der Familie
true ;
Waldmann
true.

2 ?-
```

Abbildung 8.2: start-Anfrage I

Diese Ausgabe ist sehr unübersichtlich und stellt einen unnötigen Aufwand für den Anwender dar. Mit dem Einbau des *fail*-Prädikats ist das Setzen des Semikolons, nachdem eine Lösung gefunden worden ist, überflüssig:

```
ausgabe :- write('Der Stammbaum ').
ausgabe :- write('der Familie ').
ausgabe :- write('Waldmann').

start :- ausgabe,
         fail.
```

Abbildung 8.3: ausgabe-Regeln mit fail

Stellt ein Anwender an diese Wissensbasis die gleiche Anfrage wie an die obige Wissensbasis, erhält er durch den Einsatz des *fail*-Prädikats alle möglichen Lösungen zu der Regel *start*.

```
2 ?- start.
Der Stammbaum der Familie Waldmann
false.

3 ?-
```

Abbildung 8.4: start-Anfrage II

Am Ende der Ausgabe auf der Konsole liefert Prolog ein *false*: Die Ergebnismenge ist erschöpft, es können keine weiteren Lösungen gefunden werden. Daraus ergibt sich, dass *start* nicht bewiesen werden kann.

Genauso wie der Programmierer das *fail* in die Wissensbasis eingebaut hat, gibt es auch eine Möglichkeit für den Anwender mit dem Prädikat *fail* zu arbeiten. Bei der Anfrage an die Wissensbasis in Abbildung 8.1 kann der Anwender auch die Anfrage „start, fail.“ stellen. Diese Anfrage hat den gleichen Effekt und führt zur gleichen Ausgabe wie in Abbildung 8.4.

Das folgende Beispiel bezieht sich auf Listen aus Kapitel 6. Das Prädikat *append(Liste1, Liste2, Liste3)* dient zur Erstellung einer *Liste3*, die *Liste1* und *Liste2* so beinhaltet, dass die Argumente von *Liste1* vor den Argumenten von *Liste2* in *Liste3* stehen, vorausgesetzt der Anwender setzt für *Liste1* und *Liste2* Listen und für *Liste3* eine Variable bei der Anfrage ein.

append kann auch anders verwendet werden: Setzt der Anwender für *Liste1* und *Liste2* Variablen und für *Liste3* eine Liste ein, so liefert Prolog mögliche Belegungen von den Variablen *Liste1* und *Liste2*, die dann als Listen ausgegeben werden und aus der *Liste3* bestehen. Prolog teilt also *Liste3* auf und belegt die Variablen *Liste1* und *Liste2* mit Knoten der *Liste3*. *Liste1* und *Liste2* werden dann als Listen interpretiert.

```
1 ?- append(A, B, [anton, bruno, claus, dieter, wilhelm]).
A = [],
B = [anton, bruno, claus, dieter, wilhelm] ■
```

Abbildung 8.5: append-Anfrage I

Bei dem Aufruf in Abbildung 8.5 teilt der Interpreter die Liste auf, indem *A* die leere Liste bildet und *B* die vom Anwender eingegebene Liste enthält. Mit dem Semikolon fordert der Anwender jetzt Prolog auf, nach Alternativlösungen zu suchen:

```
1 ?- append(A, B, [anton, bruno, claus, dieter, wilhelm]).
A = [],
B = [anton, bruno, claus, dieter, wilhelm] ;
A = [anton],
B = [bruno, claus, dieter, wilhelm] ■
```

Abbildung 8.6: append-Anfrage II

Die Alternativlösung: Liste *A* enthält jetzt *anton*. Liste *B* besteht jetzt aus *claus*, *dieter* und *wilhelm*.

Der Anwender kann jetzt so oft ein Semikolon setzen, bis die Ergebnismenge erschöpft ist und keine alternativen Lösungen mehr existieren.

```
1 ?- append(A, B, [anton, bruno, claus, dieter, wilhelm]).
A = [],
B = [anton, bruno, claus, dieter, wilhelm] ;
A = [anton],
B = [bruno, claus, dieter, wilhelm] ;
A = [anton, bruno],
B = [claus, dieter, wilhelm] ;
A = [anton, bruno, claus],
B = [dieter, wilhelm] ;
A = [anton, bruno, claus, dieter],
B = [wilhelm] ;
A = [anton, bruno, claus, dieter, wilhelm],
B = [] ;
false.

2 ?-
```

Abbildung 8.7: append-Anfrage III

Insgesamt existieren sechs verschiedene Lösungen. Mit Hilfe des Standard-Prädikats *fail* ist es möglich, Prolog nach Ausgabe einer Lösung jedes Mal dazu aufzufordern, nach alternativen Lösungen zu suchen. Die folgende Abbildung zeigt einen Quelltext, der neben dem *fail* auch *write* zur Ausgabe der Listen *A* und *B* enthält.

```
liste_teilen(A, B, Liste) :- append(A, B, Liste),
                             write('\nA: '),
                             write(A),
                             write('\tB: '),
                             write(B),
                             fail.
```

Abbildung 8.8: Regel liste_teilen

Durch den Einsatz des *fail*-Prädikats lässt sich die Regel *liste_teilen* nicht vollständig ableiten. Bei jedem Erreichen von *fail* setzt Backtracking ein. Die Regel wird erneut aufgerufen und es wird dem Anwender mittels *write* eine alternative Lösung angezeigt. Existieren keine alternativen Lösungen mehr, gibt Prolog ein *false* an den Anwender zurück. Der Dialog zur gestellten Anfrage zwischen Anwender und Prolog-Interpreter ist dann beendet.

Der Aufruf von der Konsole sieht wie folgt aus:

```
1 ?- liste_teilen(A, B, [anton, bruno, claus, dieter, wilhelm]).

A: []   B: [anton, bruno, claus, dieter, wilhelm]
A: [anton]      B: [bruno, claus, dieter, wilhelm]
A: [anton, bruno]       B: [claus, dieter, wilhelm]
A: [anton, bruno, claus]        B: [dieter, wilhelm]
A: [anton, bruno, claus, dieter]        B: [wilhelm]
A: [anton, bruno, claus, dieter, wilhelm]       B: []
false.

2 ?- 
```

Abbildung 8.9: liste_teilen-Anfrage

(vgl. [PRO BK 91, S. 85 bis 89], [PRO KS 03, S. 41] und [PRO RÖH 07, S. 46])

8.2 Der cut

Der *cut* ist das zweite wichtige Prädikat, mit dem das Backtracking beeinflusst werden kann. Der Name ist *cut* und syntaktisch wird der *cut* mit einem Ausrufezeichen dargestellt.

Der *cut* hat die entgegengesetzte Wirkung wie das *fail*-Prädikat: Der *cut* ist immer erfüllt, und durch seinen Einsatz ist es Prolog nicht mehr gestattet, in einer Regel nach Alternativlösungen zu suchen.

„Das Prädikat „cut“ wirkt also wie eine Mauer, die „von links kommend“ überwunden werden kann. Wird diese Mauer anschließend „von rechts kommend“ - durch (tiefes) Backtracking - erreicht, so kann sie nicht „übersprungen“ werden“. [PRO BK 91, S. 94]

Dieses Zitat erklärt noch einmal den Sinn des *cuts*. Der *cut* wird erfüllt, kann also von links überwunden werden. Backtracking wird aber verhindert, da der *cut* von rechts nicht mehr überwindbar ist.

Das folgende Beispiel bezieht sich auf das Beispielprogramm „Familienbeziehungen.pl“ der Familie Waldmann. Dazu wird eine Regel durch den Einbau des *cut*-Prädikats verändert.

```
sohn(Kind, Elter) :- maennlich(Kind),
                     elternteil(Elter, Kind),
                     !.
```

Abbildung 8.10: sohn-Regel mit cut

Die Regel *sohn* wird mit dem Ausrufezeichen, dem *cut*, erweitert. Stellt der Anwender jetzt eine Anfrage, für die es mehrere Lösungsalternativen gibt, indem er für *Kind* oder *Elter* eine Variable einsetzt, verhindert der *cut*, dass nach mehr als einer Lösung gesucht werden kann. Ist also eine Lösung gefunden, wird der Dialog zwischen Prolog und dem Programmierer beendet. Es besteht keine Möglichkeit mehr, Alternativlösungen mit Hilfe des Semikolons anzufordern, da Prolog den Dialog beendet hat und der Interpreter jetzt bereit ist, neue Anfragen entgegen zu nehmen.

```
1 ?- sohn(X, volker).
X = simon.

2 ?-
```

Abbildung 8.11: sohn-Anfrage (mit cut)

Der *cut* wird im allgemeinen eingesetzt, um die Arbeitszeit für den Prolog-Interpreter zu reduzieren. Damit soll verhindert werden, dass Prolog unnötig nach alternativen Lösungen sucht.

Man schneidet also sozusagen Zweige des Suchbaumes, der Prolog in Form der Wissensbasis zur Verfügung steht, ab, um die Suche nach der gewünschten Lösung zu beschleunigen. Es ist wichtig, darauf zu achten, was abgeschnitten wird: Der Programmierer unterscheidet deshalb zwischen dem grünen und dem roten *cut*. Ein *cut* wird immer dann als grüner *cut* bezeichnet, wenn der Programmierer Zweige abschneidet, die keine Lösungen enthalten. Andernfalls spricht man von einem roten *cut*.

Das folgende Beispiel, bekannt aus Kapitel 3.3.2, enthält jetzt einen grünen *cut*, da keine Lösungen abgeschnitten werden. Hierbei wird unnötiges Suchen nach einer Alternativlösung unterbunden:

```
kosten(X,Y) :- X > 0,
               X < 10,
               Y is X mod 2,
               !.

kosten(X,Y) :- X >= 10,
               X <  100,
               Y is X mod 20,
               !.

kosten(X,Y) :- X >= 100,
               Y is X mod 200.
```

Abbildung 8.12: kosten-Regeln mit dem cut

Da sich die Regeln durch die arithmetischen Vergleiche gegenseitig ausschließen und dadurch immer nur eine Regel bei einem Aufruf erfüllt sein kann, wird hier der *cut* in die ersten beiden Regeln als letzte Bedingung eingebaut. Wurde die erste Regel erfolgreich abgeleitet, verhindert der *cut* das Suchen nach alternativen Lösungen in den Regeln zwei und drei. Analog gilt dies auch für Regel 2. Bei Regel 3 ist kein *cut* notwendig, denn nach der dritten *kosten*-Regel existiert keine weitere *kosten*-Regel mehr im Quelltext und somit auch keine alternative Lösung, nach der gesucht werden kann.

Da durch den Einsatz des *cuts* keine Lösungen abgeschnitten werden, liegt hier ein grüner *cut* vor.

Der Einsatz der Prädikate *fail* und *cut* muss von einem Programmierer genau überdacht werden: Bei Einsatz eines dieser beiden Prädikate an einer falschen Stelle im Quelltext entstehen möglicherweise falsche Ergebnisse, da z.B. Lösungen abgeschnitten werden können.

(vgl. [PRO BK 91, S. 94ff] und [PRO RÖH 07, S. 43ff])

9 Logische Rätsel

Mit der Programmiersprache Prolog ist es möglich, logische Rätsel zu lösen, sogenannte Logikrätsel. Der Programmierer muss das Rätsel dazu geeignet modellieren. Der Prolog-Interpreter versucht dann, durch Ableiten eine Lösung zu finden.

Dieses Kapitel zeigt an einem einfachen Beispiel, wie es möglich ist, ein Logikrätsel mit Hilfe von Prolog zu lösen.

9.1 Alphametics

Alphametics sind Logikrätsel. Gegeben sind eine Gleichung oder mehrere Gleichungen, ausgedrückt in Buchstaben. Jeder einzelne Buchstabe repräsentiert eine Zahl von 0 bis 9. Keine Zahl beginnt mit einer 0. Als Rechenoperationen sind einfache Operationen wie z.B. die Addition, die Subtraktion und die Multiplikation möglich.

SWI-Prolog stellt dem Programmierer das Prädikat (*Built-in predicate*) *permutation(L1, L2)* zur Verfügung. *L1* und *L2* stehen für Listen mit der gleichen Länge. Wie der Name des Prädikats schon erschließen lässt (lat. permutare = vertauschen), gibt das Prädikat mögliche Zuordnungen einzelner Knoten der Listen *L1* und *L2* zueinander an den Anwender zurück.

Beispiel:

```
start :- permutation([A, B, C], [1, 2, 3]),
         ausgabe(A, B, C),
         fail.

ausgabe(A,B,C) :- write('A= '), writeln(A),
                  write('B= '), writeln(B),
                  write('C= '), writeln(C),
                  writeln('').
```

Abbildung 9.1: Beispielregel mit dem Prädikat permutation

Gegeben ist der Quelltext eines kleinen Prolog-Programms mit zwei Regeln. Ruft der Anwender die Regel *start* auf, versucht Prolog diese Regel abzuleiten. *permutation* ist mit zwei Listen belegt, *L1* enthält die Großbuchstaben (Variablen in Prolog) *A*, *B* und *C*, *L2* enthält die Zahlen 1, 2 und 3. Durch den einmaligen Aufruf des Prädikats *permutation(L1, L2)* wird jeder Knoten jeder Liste jeweils einem Knoten der anderen Liste zugeordnet. Als nächstes versucht der Prolog-Interpreter die Regel *ausgabe(A, B, C)* abzuleiten, die der Ausgabe der Knotenzuordnung der beiden Listen dient. *fail* wird eingesetzt, damit die Regel *start* nicht abgeleitet werden kann. Dadurch setzt Backtracking ein. Der Prolog-Interpreter leitet *permutation(L1, L2)* solange ab, bis die Ergebnismenge erschöpft ist und dem Anwender alle möglichen Belegungen über die Regel *ausgabe(A, B, C)* angezeigt worden sind.

```
1 ?- start.
A= 1
B= 2
C= 3

A= 2
B= 1
C= 3

A= 3
B= 1
C= 2

A= 1
B= 3
C= 2

A= 2
B= 3
C= 1

A= 3
B= 2
C= 1

false.

2 ?-
```

Abbildung 9.2: start-Anfrage (mit permutation)

Abbildung 9.2 zeigt die Ausgabe der Regel *permutation(L1, L2)*. Den Buchstaben (Variablen) A, B, C wird jeweils eine der drei Zahlen 1, 2 oder 3 zugeordnet.

Als nächstes ist folgendes Rätsel gegeben:

SEND + MORE = MONEY

Jeder Buchstabe steht für eine Ziffer, keine Zahl beginnt mit der Null. Für welchen Buchstaben steht welche Ziffer?

Dazu der folgende Quelltext (vgl. [PRO RÖH 07, S. 32]):

```
berechne :- permutation([0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
                        [S, E, N, D, M, O, R, Y, _, _]),
            S > 0,
            M > 0,

            S*1000 + E*100 + N*10 + D + M*1000 + O*100 + R*10 + E

            =:=

            M*10000 + O*1000 + N*100 + E*10 + Y,

            Loesung = [S, E, N, D, +, M, O, R, E, =, M, O, N, E, Y],

            writeln('\nAusgabe der Lösung:\n'),
            ausgabe(Loesung).


ausgabe([]).

ausgabe([Kopf|Rest]) :- write(Kopf),
                        ausgabe(Rest).
```

Abbildung 9.3: Regel berechne mit permutation

Im Quelltext existieren zwei verschiedene Regeln, einmal die Regel *berechne* und zweimal die Regel *ausgabe*. *permutation* ist mit zwei Listen belegt. Die erste Liste enthält die Zahlen 0 bis 9. Die zweite Liste enthält alle Buchstaben, die im Rätsel enthalten sind. Die Buchstaben S und M müssen größer Null sein, denn sie stehen am Anfang der Zahlen. Der Ausdruck „=:=“ bedeutet nummerisch gleich (vgl. Kapitel 3.3.1), demnach muss die Summe von „SEND“ und „MORE“ nummerisch gleich „MONEY“ sein.

Die Liste *Loesung* enthält die Lösung des Rätsels, welches mit der Regel *ausgabe* ausgegeben wird. Die Ausgabe der Lösung erfolgt zeichenweise rekursiv, denn die Lösung ist in einer Liste hinterlegt. Die erste *ausgabe*-Regel ist die Abbruchbedingung, damit die Regel *berechne* erfolgreich abgeleitet werden kann.

Die Ausgabe auf der Konsole sieht wie folgt aus:

```
1 ?- berechne.

Ausgabe der Lösung:

9567+1085=10652
true
```

Abbildung 9.4: berechne-Anfrage (mit permutation)

In diesem Programm versucht Prolog mittels Backtracking durch geeignete Belegung der Buchstaben die Regel *berechne* erfolgreich abzuleiten. Dieser Vorgang dauert bei der Ausführung der Anfrage *berechne* einige Sekunden, denn es gibt viele Variationsmöglichkeiten!

(vgl. [PRO RÖH 07, S. 32])

10 Syntaxprüfung von Nominalphrasen

Mit Hilfe der Programmiersprache Prolog ist es möglich, Satzstrukturen auf ihre Syntax hin zu prüfen und festzustellen, ob eine Satzphrase die Wortfolge betreffend grammatikalisch korrekt gebildet ist. Die Syntaxprüfung ist kontextfrei. Sie prüft die Anordnung der einzelnen Wörter in einer Satzphrase und berücksichtigt nicht deren Bedeutung. Dieses Kapitel zeigt, wie eine Satzphrase in Prolog abgebildet werden kann.

10.1 Beispiel: Prüfung der Nominalphrase

Ein Satz besteht aus mehreren Satzteilen. Im Folgenden wird ein Satzteil auch als Phrase bezeichnet. Mehrere Satzphrasen zusammen bilden einen Satz. Im folgenden Beispiel soll geprüft werden, ob eine von einem Anwender eingegebene Nominalphrase nach bestimmten Regeln syntaktisch korrekt gebildet ist.

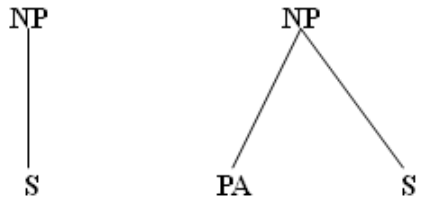

Abbildung 10.1: Nominalphrase

Die Nominalphrase eines deutschen Satzes ist hier vereinfacht dargestellt. Sie besteht entweder aus einem Substantiv oder aus einem Pronomen bzw. einem Artikel, gefolgt von einem Substantiv. Wenn ein Anwender nun eine Wortfolge an das Prolog-Programm übergibt, soll der Prolog-Interpreter mit einem *true* oder einem *false* signalisieren, ob die übergebene Wortfolge gemäß der vorgegebenen Möglichkeiten aus Abbildung 10.1 die Syntax betreffend korrekt gebildet ist.

Dazu ist eine Prolog-Wissensbasis gegeben (vgl. [PRO CM 87, S. 248]):

```
% Fakten
% Pronomen/Artikel
pa(['Die'|S], S).
pa(['Der'|S], S).

%Substantive
s(['Eisenbahn'|S], S).
s(['Garten'|S], S).


%Regeln
%Nominalphrasen
np(S0, S) :- s(S0, S).

np(S0, S) :- pa(S0, S1),
             s(S1, S).
```

Abbildung 10.2: Fakten und Regeln für die NP-Prüfung

Die Regeln entsprechen den zwei möglichen Nominalphrasen aus Abbildung 10.1. Es existieren 4 Fakten; zwei Fakten mit dem Funktor *pa* und zwei Fakten mit dem Funktor *s*. Die beiden Regeln mit der Bezeichnung *np* bilden jeweils eine Nominalphrase ab. Stellt ein Anwender eine Anfrage, indem er z.B. lediglich „Eisenbahn“ an das Prolog-Programm übergibt, so soll das Prolog-Programm „Eisenbahn“ als Nominalphrase erkennen. Genauso soll eine Nominalphrase „Die Eisenbahn“ erkannt werden. Die Wortfolge „Eisenbahn Eisenbahn“ dagegen soll keine Nominalphrase sein und nicht akzeptiert werden. Wortfolgen, bei denen mindestens ein Wort nicht in der Faktenbasis existiert, werden ebenfalls vom Prolog-Interpreter nicht erfolgreich abgeleitet. Abbildung 10.3 verdeutlicht das:

```
1 ?- np(['Eisenbahn'], []).
true .

2 ?- np(['Die', 'Eisenbahn'], []).
true .

3 ?- np(['Eisenbahn', 'Eisenbahn'], []).
false.

4 ?-
```

Abbildung 10.3: np-Anfrage

Der Anwender übergibt die Wortfolgen in Listenform und schließt jedes Wort in Hochkommata ein, um einzelne Worte auch mit einem Großbuchstaben beginnen lassen zu können. Jedes Wort ist ein Listenelement und wird in der Liste durch Kommata von anderen getrennt. Neben dieser Liste

übergibt der Anwender eine leere Liste an das Prolog-Programm: Eine *np*-Regel kann erfolgreich abgeleitet werden, wenn die Eingabeliste (Liste mit Wortfolgen) mit der zweiten übergebenen Liste (der leeren Liste) übereinstimmt.

Um dies genauer verstehen zu können, muss man die Fakten dieser Wissensbasis genauer betrachten: Wird bei der ersten Anfrage gemäß des Unifikationsalgorithmus (vgl. Kapitel 4.2.2) die Liste mit dem Wort „Eisenbahn“ mit der Variablen *S0* und die leere Liste mit der Variablen *S* unifiziert, kann diese Regel erfolgreich abgeleitet werden, wenn ein dementsprechender Fakt dazu in der Wissensbasis existiert. Der Fakt *s* (siehe Abbildung 10.2) enthält als erstes Argument auch eine Liste. Wenn die übergebene Liste *S0* einer Regel *np* eine Liste ist, und der Kopf der Liste, d.h. das erste Listenelement (z.B. „Eisenbahn“) identisch mit dem ersten Element der Liste in diesem Fakt (z.B. „Eisenbahn“) ist, lässt sich dieser Teil der Regel erfolgreich ableiten. Bei diesem Vorgang wird die Restliste (in der ersten Anfrage ist dies die leere Liste, da die eingegebene Liste des Anwenders nur aus einem Listenelement besteht) mit der Variablen *S* unifiziert.

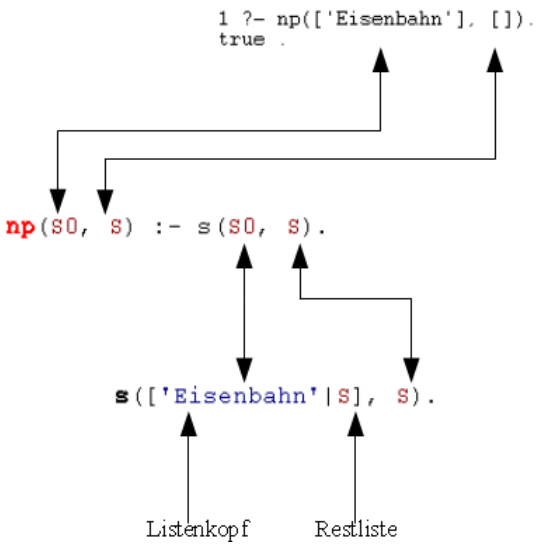

Abbildung 10.4: Skizze zur Veranschaulichung der np-Anfrage

Da der Anwender die Variable *s* als leere Liste bei der Anfrage festgelegt hat, liefert der Prolog-Interpreter jetzt ein *true* zurück, denn durch den Fakt *s* muss die Variable *S* bei der Aufteilung der Liste ['Eisenbahn'] in Listenkopf und Restliste die leere Liste sein, wenn 'Eisenbahn' als Kopf der Liste betrachtet wird (vgl. Abbildung 10.4).

Prolog erkennt ebenfalls die zweite Anfrage in Abbildung 10.3 als korrekte Nominalphrase. Bei der

dritten Anfrage gibt der Prolog-Interpreter ein *false* zurück, denn die Wortfolge „Eisenbahn Eisenbahn“ ist keine korrekte Nominalphrase.

Dieses Beispiel lässt sich theoretisch auf weitere Nominalphrasen und auf ganze Sätze erweitern.

(vgl. [PRO CM 87, S.243 bis 248])

10.2 DCG

Die sogenannte „Definite Clause Grammar“ oder kurz DCG wurde entwickelt, um einfache Parser für Grammatiken in Prolog schreiben zu können. Die DCG ist demnach eine Kurzschreibweise, die inhaltlich die gleiche Bedeutung wie der Quellcode in Abbildung 10.2 hat. Intern übersetzt Prolog den Code, der in DCG-Syntax geschrieben ist, zurück in den herkömmlichen Prolog-Code. In DCG-Syntax sieht der Quellcode aus Abbildung 10.2 wie folgt aus (vgl. PRO CM 87, S. 249]):

```
% Fakten
% Pronomen/Artikel
pa --> ['Die'].
pa --> ['Der'].

% Substantive
s --> ['Eisenbahn'].
s --> ['Garten'].


% Regeln
% Nominalphrasen
np --> s.

np --> pa,
       s.
```

Abbildung 10.5: Fakten und Regeln für die NP-Syntaxprüfung in DCG-Syntax

Bei dieser Darstellung gibt es einen Regelkopf (z.B. *np*) gefolgt von einem Pfeil („-->“) und einem Regelrumpf (z.B. *pa*, *s*). Die weiteren in Abbildung 10.2 enthaltenen Variablen und Listen werden hier weggelassen.

(vgl. [PRO CM 87, S. 249f])

11 Eine bidirektionale Schnittstelle zwischen Java und Prolog (JPL)

11.1 Allgemein

Mit JPL ist es möglich, eine Schnittstelle zwischen der Programmiersprache Java und der Programmiersprache Prolog herzustellen. JPL ist eine Menge von Java-Klassen und C-Funktionen und benutzt das Java Native Interface (JNI) um eine Verbindung mit dem Prolog Foreign Language Interface herzustellen (vgl. [JPL SI 09]).

„Mit Hilfe von JNI können aus der Java-VM heraus plattformspezifische Funktionen verwendet werden. Auch umgekehrt funktioniert dieser Weg“ [JI UL 06, S. 1383]. Demnach ist das JNI eine Schnittstelle, mit der aus Java heraus auf Funktionen und Methoden anderer Programmiersprachen wie z.B. C zugegriffen werden kann. Dies funktioniert auch umgekehrt: In Java geschriebene native Methoden können auch von anderen Programmiersprachen aus aufgerufen werden.

„Prolog itself only provides for embedding in the C-language (compatible to C++). Embedding in Java is achieved using a C-glue between the Java and Prolog C-interfaces“ [RE WI 08, S. 270]. Demnach wird die Schnittstelle FLI, die für die Kommunikation zwischen der Programmiersprache C (C++) und Prolog konzipiert ist, genutzt, um die Programmiersprache Java mit Prolog kommunizieren lassen zu können.

Kommunikation mit Java beinhaltet in diesem Zusammenhang zwei Möglichkeiten:

- Ein Java-Programm kommuniziert mit einem Prolog-Programm, indem dieses durch das Java-Programm aufgerufen wird. Anfragen können an die Wissensbasis gestellt werden und das Prolog-Programm gibt Antworten zurück.
- Ein Prolog-Programm ruft Java-Klassen und Methoden auf, die während der Laufzeit gefunden werden können („it is completely dynamic: no precompilation is required to manipulate any Java classes which can be found at run time, and any objects which can be instantiated from them“ [JPL SI 09]. Dabei können Parameter an Java übergeben und Rückgabewerte an das Prolog-Programm zurückgegeben werden.

Im Folgenden wird an einem Beispiel gezeigt, wie aus einem Java-Programm heraus auf ein Prolog-Programm zugegriffen werden kann. In einem zweiten Beispiel nutzt ein Prolog-Programm Java-Standard-Klassen und -Methoden.

11.2 Beispiel: Aufruf eines Prolog-Programms aus einem Java-Programm

Das folgende Programm für den Aufruf und das Befragen eines Prolog-Programms von einem Java-Programm aus bezieht sich auf ein Beispielprogramm, das ein Anwender nach der Installation des SWI-Prolog-Programms im Installationsordner/Verzeichnis findet. Dies gilt auch für das Beispielprogramm im nächsten Abschnitt.

Gegeben ist das Prolog-Beispielprogramm „Familienbeziehungen.pl“ (siehe Kapitel 2, Abbildung 2.2). Des Weiteren ist ein Java-Quellcode („Familienbeziehungen.java“) gegeben. Durch den Import von *jpl.Query* kann in dem Java-Programm auf Klassen und Methoden zugegriffen werden, die im JPL-Paket enthalten sind. Eine zugehörige API (*application programming interface*) findet sich auf der SWI-Prolog Homepage ([JPL SI 09]).

```
import jpl.Query;
import java.util.Hashtable;

public class Familienbeziehungen{

  public static void main(String argv[]){
    try{
      String s1="consult('Familienbeziehungen.pl')";
      Query q1=new Query(s1);
      System.out.println("Konsultierung des Prolog-Programms:");
      System.out.println("Prolog-Antwort:");
      System.out.println(q1.hasSolution());
      q1.close();

      String s2="vater(volker, simon)";
      Query q2=new Query(s2);
      System.out.println("\nAnfrage: Ist \"volker\" der Vater von \"simon\"?");
      System.out.println("Prolog-Antwort: ");
      System.out.println(q2.hasSolution());
      q2.close();
```

Abbildung 11.1: Ausschnitt I aus dem Java-Quellecode "Familienbeziehungen.java"

Um eine Anfrage an ein Prolog-Programm stellen zu können, wird in diesem Beispiel eine Instanz vom *Query* angelegt. Der angelegte *String s1* enthält genau den Befehl, der sonst über die Prolog-Konsole eingegeben werden müsste. Mit dem Methodenaufruf *hasSolution()* wird die Anfrage an Prolog übergeben und Prolog liefert als Rückgabewert entweder ein *true* oder ein *false* an das Java-Programm zurück. Zusätzlich können Ausgaben, die Prolog sonst auf der Prolog-Konsole ausgibt, hierbei mit ausgegeben werden. Nachdem die Anfrage gestellt und die Antwort zurück an das Java-Programm gekommen ist, wird die *Query* mit dem Methodenaufruf *close()* wieder geschlossen. Der Methodenaufruf *close()* muss nicht zwingend gesetzt werden, er stellt aber nach dem Stellen der Anfrage und dem Erhalt einer Antwort das Schließen der *Query* sicher.

In diesem Beispiel führt die erste Instanz von *Query* die Konsultierung des Prolog-Programms „Familienbeziehungen.pl“ durch; die Prolog-Antwort ist bei erfolgreicher Konsultierung *true*. Eine erfolgreiche Konsultierung bedeutet, dass die konsultierte Prolog-Datei der Prolog-Wissensbasis hinzugefügt worden ist. Die zweite *Query* stellt eine Anfrage, die das Prolog-Programm mit *true* oder *false* beantworten kann. Die Konsolenausgabe dieses Java-Programms sieht wie folgt aus:

```
Konsultierung des Prolog-Programms:
Prolog-Antwort:
% Familienbeziehungen.pl compiled 0.00 sec, 4,812 bytes
true

Anfrage: Ist "volker" der Vater von "simon"?
Prolog-Antwort:
true
```

Abbildung 11.2: Konsolenausgabe I des Java-Programms "Familienbeziehungen.java"

Nach der Konsultierung des Prolog-Programms „Familienbeziehungen.pl“ liefert Prolog bei der Java-Standardausgabe den Inhalt, der sonst auf der Prolog-Konsole erscheinen würde: „% Familienbeziehungen.pl compiled 0.00 sec, 4,812 bytes“ und zusätzlich ein *true*.

In der nächsten Anfrage liefert Prolog ein *true* an das Java-Programm zurück; demnach existiert ein Fakt in der Wissensbasis, der besagt, dass *volker* der Vater von *simon* ist.

Des Weiteren ist es möglich, konkrete Lösungen für Anfragen mit Variablen an das Java-Programm zurückzugeben, die dann z.B. mit einer *for-Schleife* ausgegeben werden können. Ein Beispiel bietet der folgende Quelltextausschnitt, der sich in diesem Beispielprogramm befindet:

```
String s3="maennlich(X)";
Query q3=new Query(s3);
Hashtable[] solutions=q3.allSolutions();

System.out.println("\nAnfrage: Ausgabe aller maennlichen Personen,");
System.out.println("die in der Prolog-Wissensbasis hinterlegt sind:");
System.out.println("Prolog-Antwort: ");
for(int i=0; i<solutions.length; i++){
  System.out.println("X"+(i+1)+"= "+solutions[i].get("X"));
}
q3.close();
```

Abbildung 11.3: Ausschnitt II aus dem Java-Quellcode "Familienbeziehungen.java"

Der Rückgabewert des Methodenaufrufs *allSolutions()* speichert in einem *Array* vom Typ *Hashtable* Lösungen für die Anfrage, die im *String s3* hinterlegt ist. *Hashtable* muss vorher mittels *import* importiert werden. Die Ausgabe des Ergebnisses, das entweder aus keiner, einer oder aus mehreren Lösungen für die Variable *X* bestehen kann, erfolgt in einer *for-Schleife* auf der Konsole:

```
Anfrage: Ausgabe aller maennlichen Personen,
die in der Prolog-Wissensbasis hinterlegt sind:
Prolog-Antwort:
X1= olaf
X2= volker
X3= simon
X4= sebastian
```

Abbildung 11.4: Konsolenausgabe II des Java-Programms "Familienbeziehungen.java"

Dieses Beispielprogramm enthält einen *try*- und einen *catch*-Block, um gegebenenfalls auf Fehler reagieren zu können. In Abbildung 11.1 sieht man den *try*-Block; der *catch*-Block ist in der folgenden Abbildung dargestellt:

```
catch(jpl.PrologException e){
  System.out.println("Fehler im Programm \"Familienbeziehungen.java\"");
  System.out.println(e.getMessage());
}
```

Abbildung 11.5: Ausschnitt III aus dem Java-Quellcode "Familienbeziehungen.java"

Führt die Ausführung des *try*-Blocks zu einem Fehler, wird der *catch*-Block in Abbildung 11.5 ausgeführt.

11.3 Beispiel: Aufruf von Standard Java-Klassen und Methoden aus einem Prolog-Programm

Wie im vorherigen Abschnitt erläutert, kann aus einem Prolog-Programm heraus auf Java-Klassen und Methoden zugegriffen werden. Das folgende Beispiel hat dieselbe Thematik: Gegeben ist das Prolog-Beispielprogramm „Famlienbeziehungen.pl“.

Ein Anwender gibt einen Namen einer Person ein und das Programm prüft, ob diese Person als männlich in der Wissensbasis hinterlegt ist. Ist die gesuchte Person als *maennlich* in der Wissensbasis hinterlegt, wird dies dem Anwender mitgeteilt („Person ist maennlich!“), andernfalls enthält der Anwender „Person ist nicht maennlich!“. Werden vom Anwender keine Angaben gemacht, erhält der Anwender die Mitteilung „Keine Eingabe erfolgt!“.

In diesem Beispiel greift das Prolog-Programm auf die Standard-Klasse *JFrame* und *JOptionPane* zu. Die Idee dabei ist die Erstellung eines Prolog-Programms, welches die grafische Oberfläche der Programmiersprache Java zum Einlesen von Eingaben des Anwenders und zur Ausgabe der Ergebnisse, die das Prolog-Programm aufgrund der Eingabe (Anfrage) erzeugt, benutzt.

Um die Erstellung dieses Programms zu ermöglichen, werden zwei Prädikate aus der Prolog-Datei *jpl.pl* (im Ordner *library* im SWI-Prolog-Verzeichnis) vorgestellt, die ein Programmierer in sein Prolog-Programm für den Zugriff auf Java einbinden kann:

jpl_new(+Class, +Params, -Ref)

Mit diesem Aufruf versucht Prolog sich eine Instanz von einer Klasse anzulegen. *Class* steht hierbei für einen Klassennamen (z.B. *'javax.swing.JFrame'*). *Params* steht für Parameter, die dem Konstruktor übergeben werden können, z.B. *['Fenster']*, dem Titel für das Fenster, das angelegt werden soll, in Listenform in eckigen Klammern eingeschlossen. *Ref* steht für eine Variable, z.B. mit dem Namen *Frame*. Diese Variable ist eine Referenz auf das Objekt, welches angelegt werden soll (z.B. eine Referenz auf ein Objekt der Klasse *JFrame*). Bei einem Aufruf einer Methode dieses Objektes (z.B. durch *jpl_call*, siehe dazu den nächsten Abschnitt) wird diese Variable (z.B. *Frame*) mit übergeben.

jpl_call(+Ref, +Method, +Params, -Result)

Mit *jpl_call* können Methoden von Objekten oder Klassen aufgerufen werden. *Ref* steht für eine Variable (z.B. *Frame*) und ist eine Referenz, die sich auf das Objekt bezieht, deren Methode aufgerufen werden soll. *Method* ist der Methodenname der aufzurufenden Methode (z.B. *setVisible*). *Params* steht für Parameter, z.B. bei der Methode *setVisible* für den Übergabeparameter *[@true]*. Das @ wird hierbei als *Prefix-Operator* (dt. Vorsatzcode-Operator) benutzt. Mit diesem Operator können z.B. *true*, *false* und *void* dargestellt und direkt an Java übergeben werden. *Result* steht für einen Rückgabewert. Gibt es keinen, kann an dieser Stelle entweder *@void* oder die anonyme Variable (vgl. Kapitel 3.1.3) gesetzt werden.

Ein Plus- oder ein Minuszeichen bedeutet, dass das jeweilige Argument entweder vom Prolog-Programmierer vorgegeben wird (Plus), z.B. *[@true]* oder beim Aufurf z.B. der Java-Methode belegt wird (Minus), z.B. *Result*.

Es existieren weitaus mehr als die hier vorgestellten Möglichkeiten (vgl. [JPL SI 09]).

Gegeben ist die Prolog-Datei des vorangegangenen Beispiels. Diese Prolog-Datei wird durch die folgende Regel erweitert:

```
person_finden :- jpl_new('javax.swing.JFrame', ['Fenster'], Frame),

                 jpl_call(Frame, setLocation, [400,300], _),
                 jpl_call(Frame, setSize, [400,300], _),
                 jpl_call(Frame, setVisible, [@(true)], _),
                 jpl_call(Frame, toFront, [], _),

                 jpl_call('javax.swing.JOptionPane',showInputDialog,
                          [Frame, 'Bitte Personenname eingeben'], Eingabe),

                 pruefen(Eingabe, Ausgabe),

                 jpl_call('javax.swing.JOptionPane', showMessageDialog,
                          [Frame, Ausgabe], _),

                 jpl_call(Frame, dispose, [], _).
```

Abbildung 11.6: Regel person_finden

Zunächst wird beim Aufruf dieser Regel eine Instanz der Klasse *JFrame* angelegt. Die Referenz *Frame* wird bei den folgenden Methodenaufrufen durch *jpl_call* mit übergeben. Nachdem die Parameter für die Position, die Größe, die Sichtbarkeit und die Anordnung (z.B. in den Vordergrund) des Fensters mit *jpl_call* an Java übergeben worden sind, folgt das Anlegen einer Messagebox (*JOptionPane*); dabei wird die Methode *showInputDialog* aufgerufen. Als Übergabeparameter werden die Referenz des Hauptfensters (*Frame*) und die Zeichenkette „Bitte Personenname eingeben“ übergeben. Der Rückgabewert dieser Methode wird in der Variablen *Eingabe* hinterlegt. Es folgt der Aufruf der Regel *pruefen*, die im Folgenden erläutert wird. Beim Methodenaufruf *showMessageDialog* in der nächsten Zeile wird die Variable *Ausgabe* mit übergeben. In dieser Variable steht das Resultat, das das Prolog-Programm durch die Regel *pruefen* an die Variable *Ausgabe* übergeben hat. Mit dem Methodenaufruf *dispose* schließt sich das Hauptfenster.

```
pruefen(Eingabe, Ausgabe) :- Eingabe == @(null),
                             Ausgabe = 'Keine Eingabe erfolgt!'.

pruefen(Eingabe, Ausgabe) :- maennlich(Eingabe),
                             Ausgabe = 'Person ist maennlich!'.

pruefen(_, Ausgabe)       :- Ausgabe = 'Person ist nicht maennlich!'.
```

Abbildung 11.7: pruefen_Regeln

Die Regel *pruefen* existiert dreimal. Da der Fokus dieses Beispiels auf dem Aufruf von Java-Klassen und Methoden liegt, wird im Folgenden nur kurz auf diese Regeln eingegangen. Je nachdem, ob und welche Eingaben der Anwender macht, kann eine dieser Regeln erfolgreich abgeleitet werden. Das, was nach der erfolgreichen Ableitung einer dieser Regeln in der Variablen *Ausgabe* steht, wird an die Java-Methode *showMessageDialog* übergeben und von dieser verarbeitet.

Bei dem Versuch, die Regel *pruefen* erfolgreich abzuleiten, kann Backtracking einsetzen (vgl. Kapitel 4.2.3): Lässt sich z.B. die erste Regel *pruefen* nicht erfolgreich ableiten, geht der Prolog-Interpreter zum letzten Alternativpunkt zurück; dieser Alternativpunkt ist die zweite Regel *pruefen*, diese versucht Prolog erneut erfolgreich abzuleiten. Schlägt dies fehl, setzt zum zweiten Mal Backtracking ein: Die Ableitung der dritten Regel *pruefen* wird dann erfüllt, denn diese Regel enthält keine Bedingung, die in diesem Kontext nicht erfüllt werden kann.

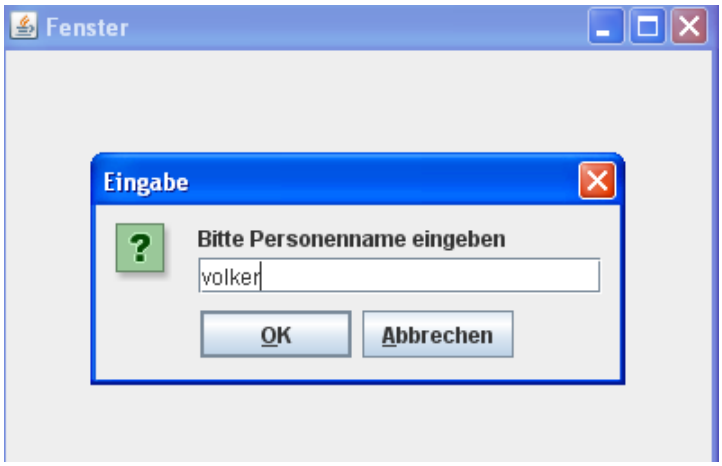

Abbildung 11.8: Ausführung des Prolog-Programms I

Bei dem Versuch, die Regel *person_finden* erfolgreich abzuleiten, öffnet sich zunächst das Hauptfenster (*JFrame*), indem sich das *InputDialog*-Fenster öffnet. Der Anwender kann nun einen Personennamen eingegeben und prüfen lassen, ob dieses Person als *maennlich* in der Wissensbasis hinterlegt ist. In diesem Fall wird der Personenname „volker“ eingegeben.

Nachdem der Anwender seine Eingabe mit dem *OK*-Button bestätigt hat, erzeugt das Programm eine Ausgabe:

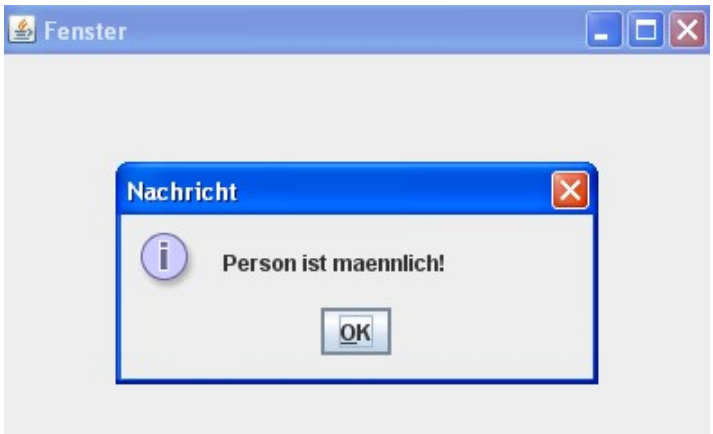

Abbildung 11.9: Ausführung des Prolog-Programms II

Das Programm bestätigt dem Anwender, dass der von ihm eingegebene Personenname als *maennlich* in der Wissensbasis hinterlegt ist. Durch Betätigen des *OK*-Buttons oder durch Klicken auf das grafisch dargestellte *X* schließen sich das *MessageDialog*-Fenster sowie das Hauptfenster (*JFrame*) und das Prolog-Programm wird beendet.

Um die oben genannten und in diesem Beispiel benutzten Prädikate verwenden zu können, muss durch

:- use_module(library(jpl)).

die Prolog-Datei „jpl.pl“ genau wie das Prolog-Programm „Familienbeziehungen.pl“ konsultiert werden. Auf der Prolog-Konsole erscheinen dann Meldungen, ob die Prolog-Datei „Familienbeziehungen.pl“ und die Prolog-Datei „jpl.pl“ sowie Prolog-Dateien, die diese Datei selbst aufruft, erfolgreich konsultiert werden konnten.

12 SWI-Prolog

Dieses Kapitel ist eine Anleitung für das Herunterladen, Installieren und Benutzen von SWI-Prolog.

12.1 Download und Installation

1) Web-Adresse: *www.swi-prolog.org*
2) Download unter der Web-Adresse: *http://www.swi-prolog.org/Download.html*
3) *Stable release* oder *Development release* auswählen
4) Die gewünschte Installationsdatei auswählen, herunterladen und anschließend die Installation starten.

Wenn Sie die JPL-Schnittstelle benutzen möchten, müssen dazu die *Umgebungsvariablen Classpath* und *Path* um folgende Einträge erweitert werden (Beispiel an einem Rechner mit dem Betriebssystem Windows XP und SWI-Prolog 5.7.1):

Classpath: C:\Programme\pl\lib\jpl.jar (für die Datei *jpl.jar*)

Path: C:\Programme\pl\bin (für die Datei j*pl.dll*)

Beispieldateien befinden sich unter C:\Programme\pl\doc\packages\examples\jpl

Genauere Informationen zur Installation von JPL finden sich unter folgendem Link:

http://www.swi-prolog.org/packages/jpl/installation.html

12.2 Arbeiten mit dem SWI-Prolog

Der folgende Abschnitt dient als kleine Bedienungsanleitung und beschreibt, wie Prolog-Dateien erstellt und konsultiert werden und Anfragen an die Wissensbasis gestellt werden können. Auf der Homepage des SWI-Prolog finden sich weitere Anleitungen.

Im SWI-Prolog ist es möglich, durch den Konsolenbefehl *help.* auf die *SWI-Prolog help* zuzugreifen.

12.2.1 Neue Prolog-Datei erstellen und konsultieren

a) SWI-Prolog öffnen (*plwin.exe*)

b) 2 Möglichkeiten, eine neue Prolog-Datei anzulegen:

- *File* dann *New* neue Prolog-Datei anlegen
- Konsolenbefehl *edit(file('Dateiname'))*

c) Es öffnet sich ein Editor, um Fakten und Regeln einzugeben

2 Möglichkeiten um Kommentare im Quelltext zu erstellen:

- Kommentar in die Zeichen /* *Kommentar* */ einschließen
- Zeichen '*%*' an den Anfang einer Zeile schreiben, in der ein Kommentar stehen soll; die ganze Zeile wird dann als Kommentar interpretiert.

d) Nach Eingabe der Fakten und Regeln mit *File* dann *Safe Buffer* speichern

e) mit *File*, dann *Quit* den Editor verlassen

f) 2 Möglichkeiten, die Datei zu konsultieren:

- File dann *Consult*
- mit dem Konsolenbefehl *consult('Dateiname.pl')*

g) SWI-Prolog gibt jetzt die Meldung auf der Konsole aus, ob die Datei ohne Fehler erfolgreich kompiliert werden konnte.

h) Bei einer fehlerlosen Kompilierung können jetzt Anfragen/Behauptungen an die Wissensbasis gestellt werden.

Alternativ ist es auch möglich, im Editor durch *Compile*, dann *Compile Buffer* die Fakten und Regeln aus dem Quelltext in die Wissensbasis zu laden.

12.2.2 Eine bestehende Prolog-Datei öffnen

a) Per Doppelklick auf die Datei wird die Datei im SWI-Prolog geöffnet und *sofort* kompiliert.

b) Bei einer fehlerlosen Kompilierung können jetzt Anfragen/Behauptungen an die Wissensbasis gestellt werden.

12.2.3 Eine bestehende Prolog-Datei bearbeiten

a) 2 Möglichkeiten

- Die Prolog-Datei wie unter 12.2.2 a)
- unter *File* dann *Edit* den Editor mit der Datei öffnen
- Konsoleneingabe: *edit('Dateiname')*

b) Zum Speichern und Verlassen wie unter 12.2.1 d) und e) vorgehen

c) 2 Möglichkeiten, die Datei zu konsultieren:

- File dann *Consult*
- mit dem Konsolenbefehl: *consult('Dateiname.pl')*

d) Bei einer fehlerlosen Kompilierung können jetzt Anfragen/Behauptungen an die Wissensbasis gestellt werden.

12.2.4 Anfragen an die Wissensbasis stellen

a) Wenn Fakten und Regeln der Wissensbasis hinzugefügt worden sind, können Behauptungen/Anfragen an die Wissensbasis gestellt werden.

b) Bei Anfragen unterscheidet man zwischen

   - konkreten Behauptungen, die nur mit wahr oder falsch (*true* oder *false*) beantwortet werden
   - Behauptungen, zu denen es mehrere Antwortmöglichkeiten gibt, durch einsetzen von Variablen

c) Eine Behauptung/Anfrage wird mit dem Zeichen '.' abgeschlossen.

   Dieses Zeichen signalisiert dem Interpreter das Ende einer Anfrage und fordert ihn auf, eine Antwort zu finden.

12.2.5 Beenden des SWI-Prolog

Signalisiert der Prolog-Interpreter dem Anwender die Eingabeaufforderung '?-', kann der SWI-Prolog und das aktuelle Programm mit dem Konsolenbefehl *halt* beendet werden.

Abbildungsverzeichnis

Literaturverzeichnis

[INH RP 02] Rechenberg, P. Und Pomberger, G.:
Informatik-Handbuch,
München, 3. Auflage 2002

[JI UL 06] Ulleboom, C.:
Java ist auch eine Insel,
Das umfassende Handbuch,
Programmieren mit der Java Standard Edition Version 5,
Bonn, 5. aktualisierte Auflage 2006

[JPL SI 09] Singleton, P., Dushin, F., Wielemaker, J.:
JPL: A bidirectional Prolog/Java interface,
Quelle: *http://www.swi-prolog.org/packages/jpl*
Datum des Zugriffs: Juli 2009

[PRO ACH 08] Achermann, B.:
Kurzeinführung in Prolog,
CH-3012 Bern,
Quelle: *http:www.iam.unibe.ch/fki/lectures/kunstliche-intelligenz/Prolog.pdf*
Datum des Zugriffs: Oktober 2008

[PRO BK 91] Bothner, P., Kähler, W.-M.:
Programmieren in PROLOG,
Eine umfassende und praxisgerechte Einführung,
Braunschweig, 1991

[PRO CM 87] Clocksin, W.F. Und Mellish, C.S.:
Programmieren in Prolog,
Berlin Heidelberg, 1987

[PRO GOL 08] Goltz, H.-J.:
Einführung in die Programmiersprache PROLOG-Antwort
Berlin,
Quelle: *www.goltz-berlin.de/lehre/prolog_ein.pdf*
Datum des Zugriffs: Oktober 2008

[PRO KS 03] König, E., Seiffert, R.:
Grundkurs PROLOG für Linguisten,
Online-Fassung vom 20. Juni 2003,
Quelle: *www.seiffert-diaz.net/einf-prolog-koenig-seiffert.pdf*

[PRO RÖH 07] Röhner, G.:
Informatik mit Prolog,
Frankfurt, 3. Auflage 2007

[RE WI 08] Wielemaker, J.:
SWI-Prolog 5.6,
Reference Manual,
Quelle: *http:www.swi-prolog.org/*
Amsterdam, August 2008
Datum des Zugriffs: Juli 2009

[SW WI 09] Wielemaker, J.:
SWI-Prolog,
Version 5.7.1,
Quelle: *http:www.swi-prolog.org/*
Datum des Zugriffs: Juli 2009

[TI HO 09] Hoffmann, D. W.:
Theoretische Informatik,
München 2009

[WI BHS 07] Boersch, I., Heinsohn, J., Socher, R.:
Wissensverarbeitung,
Eine Einführung in die Künstliche Intelligenz für Informatiker und Ingenieure,
Heidelberg, 2. Auflage 2007

weiterführende Literatur zu Prolog:

Hackmack, S.:

Erste Schritte im SWI-PROLOG,

Bremen,

Quelle: *www.fb10.uni-bremen.de/homepages/hackmack/prolog/texte/swi.pdf*

Datum des Zugriffs: Oktober 2008

Schröder, B., Schmitz, H.-C.:

Einführung in Prolog,

Bonn 2005,

Quelle: *www.ikp.uni-bonn.de/dt/lehre/material/prolog/prolog.pdf*

Datum des Zugriffs: Oktober 2008

www.wikipedia.org:

Prolog (Programmiersprache),

Quelle: *http://de.wikipedia.org/wiki/PROLOG*

Datum des Zugriffs: Oktober 2008