

Semantic Web - Rules & Prolog

Pascal Recktenwald
Matrikelnummer: 2030710

precktenwald@xantippe.cs.uni-sb.de

Saarbrücken, 30. April 2007

Seminar: A.I. Tools

Wintersemester 2006/2007

Betreuer:

Dr. Michael Kipp

Dr. Alassane Ndiaye

Dr.-Ing. Dominik Heckmann

Dipl.-Inform. Michael Feld



Universität des Saarlandes
Fachbereich 6.2, Informatik
Lehrstuhl für Künstliche Intelligenz
Prof. Dr. Dr. h.c. mult. W. Wahlster



Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Abstract

Eines der aktuellen Forschungsziele der Informatik ist es, im Web bislang rein symbolisch repräsentiertes Wissen auch für Computer verständlich zu machen. Mit dem Semantic Web Layer Cake existiert eine Zielvorgabe zur Lösung dieser Aufgabe. Teil dieser Zielvorgabe ist ein Logik-Framework, mit dem über die mit Hilfe von Regeln logisch beschriebenen Zusammenhänge im Web Schlüsse gezogen werden können. Prolog, welches auf Hornklauseln basiert, ist ein solches Logik-Framework. Fakten und Regeln stellen in Prolog eine Wissensbasis zur Verfügung an die Anfragen gestellt werden können. Diese Anfragen werden dann vom System beantwortet. Dazu verfügt Prolog über grundlegende Mechanismen wie Matching, Backtracking und eine Baum-basierte Datenstrukturierung. Verschiedene vordefinierte Prädikate dienen unter anderem zur Steuerung der Antwortsuche.

Inhaltsverzeichnis

1	Einleitung	4
2	Regeln	6
3	Prolog	7
3.1	Geschichte	7
3.2	Hornklauseln	8
3.3	Datenobjekte	9
3.3.1	Konstanten	9
3.3.2	Variablen	10
3.3.3	Strukturen	10
3.4	Grundlegende Konzepte	11
3.4.1	Fakten	11
3.4.2	Anfragen	12
3.4.3	Regeln	14
3.5	Weiterführende Konzepte	16
3.5.1	Rekursion	16
3.5.2	Matching	18
3.5.3	Backtracking	19
3.6	Vordefinierte Prädikate	21
3.6.1	Arithmetik	21
3.6.2	Der Cut-Operator	22
3.7	Einfluss und Anwendung	25
4	Zusammenfassung	25
	Literatur	27

1 Einleitung

Eines der Ziele in der Informatik ist es vorhandenes Wissen zusätzlich zu der für den Menschen lesbaren Form formal so zu repräsentieren, dass es auch von Computern verstanden werden kann. Zu diesem Wissen zählt zum Beispiel das Wissen über Fakten und Regeln in Geschäftsprozessen und juristischen Verfahren sowie die Inhalte von Dokumenten und Internetseiten. Durch das Verständlichmachen des Wissens für den Computer sollen beispielsweise Anfragen anstelle ihrer Repräsentation anhand ihrer Bedeutung bearbeitet werden können.

Das World Wide Web, in dem ein Teil dieses Wissen öffentlich zugänglich gemacht wird, ist in seiner klassischen Form (*Web 1.0*) ein rein syntaktisches Web, vgl. [Wah06b]. Die Daten werden beispielsweise mit XML¹ strukturiert und das Layout erfolgt mit HTML², siehe [Wah06a, S. 22]. Der Mensch ist dabei durch sein Kontextwissen in der Lage den Inhalt der im World Wide Web gespeicherten Daten zu verstehen. Computer hingegen können diese Daten zwar repräsentieren, jedoch haben sie keinerlei Verständnis über den Inhalt dieser Daten. Um dies zu ermöglichen, benötigen sie zusätzlich eine Repräsentation der zugrunde liegenden Begriffe und deren Zusammenhänge.

Das Ziel der Wissensverarbeitung ist es daher vom syntaktischen Web wegzugehen, hin zu einem semantischen Web, in dem zusätzlich zu der Struktur und dem Layout der Daten auch deren Inhalt so repräsentiert wird, dass auch Computer diese Daten verstehen können. Fensel et al. beschreiben dies in dem Buch „*Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential*“ [FWLH03] als Schritt von einem *Web of Links*³ hin zu einem *Web of Meaning*⁴. Tim Berners-Lee, der als Begründer des World Wide Web gilt, entwickelte deshalb mit dem *Semantic Web Layer Cake* eine Zielvorgabe für die Umsetzung eines semantischen Netzwerkes, dem so genannten *Semantic Web*, siehe [BLHL01]:

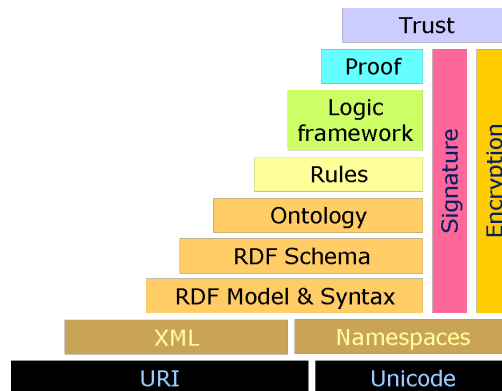


Abbildung 1: Semantic Web Layer Cake

¹Extensible Markup Language

²Hypertext Markup Language

³deutsch: Netz aus Hyperlinks

⁴deutsch: Netz der Bedeutung

Dieses Konzept geht sogar noch über das reine Web of Meaning hinaus und beschreibt das so genannte *Web of Trust*⁵ in dem alle Applikationen die Aussagen die sie verarbeiten auf ihre Glaubwürdigkeit hin überprüfen. Alle Aussagen im Web tauchen dabei in einem bestimmten Kontext auf und damit die Applikationen die Glaubwürdigkeit dieser Aussagen einschätzen können, muss ihnen dieser Kontext durch das Semantic Web zur Verfügung gestellt werden.

Der Semantic Web Layer Cake (siehe Abbildung 1) ist dazu wie folgt aufgebaut: Die Grundlage bilden die so genannten *Uniform Resource Identifiers* (URIs) und XML. URIs sind kompakte Zeichenketten, die mit dem Unicode Zeichensatz kodiert werden. Sie dienen dazu, Ressourcen im World Wide Web zu identifizieren um Beziehungen zwischen diesen ausdrücken zu können. Mit XML wird eine Syntax für die Strukturierung von Dokumenten zur Verfügung gestellt. An dieser Stelle werden jedoch keine Aussagen über die semantische Bedeutung dieser Dokumente getroffen. *XML-Namespaces* definieren dabei die Struktur und die Inhaltselemente der XML-Dokumente.

Darüber bietet das *RDF*⁶-*Model* ein einfaches Datenmodell, mit dem Ressourcen referenziert und Beziehungen zwischen diesen beschrieben werden können. Ein RDF-basiertes Modell kann dabei in XML-Syntax repräsentiert werden. *RDF Schema* (oder kurz: *RDF-S*) stellt ein Vokabular zum Beschreiben von Eigenschaften und Klassen von RDF-Ressourcen zur Verfügung, wobei eine Semantik für die Verallgemeinerungshierarchien dieser Eigenschaften und Klassen beschrieben wird. Da dieses Vokabular zum Beschreiben von Eigenschaften und Klassen jedoch sehr eingeschränkt ist, vergrößert *OWL*⁷ das Vokabular unter anderem durch Relationen zwischen Klassen (z.B. Disjunktheit), Kardinalitäten (z.B. „genau eines“), Charakteristiken von Eigenschaften (z.B. Symmetrie) und größeren Möglichkeiten bei der Typisierung von Eigenschaften. Mit Hilfe von OWL lassen sich nun Ontologien definieren, die die Konzepte einer bestimmten Domäne beschreiben. Über diese Konzepte können dann in der Ebene der Regeln (*Rules*) Aussagen und Regeln formuliert werden.

Will man nun die Glaubwürdigkeit solcher Aussagen und Regeln überprüfen, benötigt man eine Logik, die es ermöglicht, bestimmte Aussagen auf ihre Redundanz oder Inkonsistenz zu überprüfen. Das Logik-Framework des Semantic Web Layer Cake stellt deshalb eine formale Struktur bereit, mit der Computersysteme über die formulierten Regeln Schlüsse ziehen⁸ können. Können durch einen solches Inferenzsystem (*Reasoner*) schließlich sogar Beweise über die Korrektheit solcher Aussagen geführt werden, kann mit Hilfe von Signaturen und Verschlüsselungen letztendlich das Ziel Vertrauen (*Trust*) erreicht werden.

Bis zu der Ebene der Ontologien ist das Konzept des Semantic Web Layer Cake bereits weitestgehend umgesetzt: Mit URIs, XML, RDF, RDF-S und OWL wurden bereits mehrere Vorschläge für die Umsetzung von Ebenen des Semantic Web Layer Cake vom World Wide Web Consortium⁹ als Standards übernommen. Außerdem gibt es bereits eine Vielzahl von OWL-Ontologien für die

⁵sinngemäß: „Netz des Vertrauens“

⁶Resource Description Framework

⁷Web Ontology Language

⁸englisch: reasoning

⁹kurz: W3C (siehe <http://www.w3.org/>)

verschiedensten Domänen. Das aktuelle Ziel der Forschung auf dem Gebiet des Semantic Web ist es somit, Standards für die oberen Ebenen des Semantic Web Layer Cake zu entwickeln, um dem Ziel Vertrauen Schritt für Schritt näher zu kommen.

Kapitel 2 dieser Arbeit zeigt deshalb zunächst einen kurzen Einblick in die Ebene der Regeln und stellt eine mögliche Regelsprache für das Semantic Web vor. Im darauf folgenden Kapitel wird das Hauptthema dieser Arbeit, das Logik-Framework *Prolog*, vorgestellt. Kapitel 3 gibt dazu zunächst eine Einführung in die Geschichte von Prolog und dessen logischer Grundlage, die so genannten *Hornklauseln*. In den darauf folgenden Abschnitten werden die grundlegenden Datenobjekte und Konzepte von Prolog vorgestellt sowie näher auf die weiterführende Techniken *Rekursion*, *Matching* und *Backtracking* eingegangen. Am Ende von Kapitel 3 werden schließlich noch einige vordefinierte Prädikate und weitere Anwendungsgebiete von Prolog neben dem Semantic Web vorgestellt. Als Grundlage für Kapitel 3 und die darin verwendeten Beispiele dient das Buch „*PROLOG Programming for Artificial Intelligence*“ von Ivan Bratko [Bra00]. Kapitel 4 fasst diese Arbeit in einem abschließenden Fazit noch einmal zusammen.

2 Regeln

Regeln sind formalisierte Konditionalsätze der Form „Wenn A gilt, dann gilt auch B“. Der „Wenn“-Teil der Regel bezeichnet man auch als *left hand side (LHS)* oder *Prämisse* und den „Dann“-Teil als *right hand side (RHS)* oder *Konklusion*.

In den Ontologien des Semantic Web werden implizit bereits Regeln formuliert. Betrachtet man das Beispiel „*Frau* ist ein Unterkonzept des Konzeptes *Person*“ (Subsumption), so wird implizit eine Regel formuliert, die besagt dass jede Instanz des Konzeptes *Frau* auch automatisch eine Instanz des Konzeptes *Person* ist. In den Ontologien können auf diese Weise aber nur sehr einfache Regeln ausgedrückt werden. Mit speziellen Regelsprachen ist es jedoch möglich neue und komplexere Konstrukte zu formulieren. Die Regeln werden dabei nicht wie in den Ontologien implizit modelliert, sondern explizit deklariert. Dadurch erhält man eine größere Ausdrucksstärke der Ontologie und der daraus ableitbaren Schlussfolgerungen. Allerdings verschlechtert sich dadurch auch die Berechenbarkeit bzw. die Entscheidbarkeit, vgl. [Die05]. Eine Regel, die man mit Hilfe einer Regelsprache definieren könnte, wäre beispielsweise die folgende:

$$\forall X, Y, Z : hasParent(X, Y) \wedge hasBrother(Y, Z) \Rightarrow hasUncle(X, Z)$$

Diese Regel sagt aus, dass jeder der ein Elternteil hat, welches wiederum einen Bruder hat, auch einen Onkel hat. Der Onkel ist dabei gerade der Bruder des Elternteils.

Zurzeit gibt es jedoch noch keinen Standard für eine Regelsprache im Semantic Web. Im Jahr 2004 wurde mit der Regelsprache *SWRL*¹⁰ allerdings ein Vorschlag dafür beim World Wide Web Consortium eingereicht, siehe [HPSB⁺04].

¹⁰Semantic Web Rule Language

SWRL basiert auf einer Kombination von Untersprachen der Web Ontologie Language OWL (OWL Lite und OWL DL)¹¹ und Datalog RuleML¹². Die Syntax von SWRL kann dabei in XML oder in RDF gehalten sein. Obwohl sowohl OWL DL als auch Datalog RuleML entscheidbar sind, ist SWRL mit OWL DL und Datalog RuleML bereits nicht mehr entscheidbar.

Generische Repräsentationsmechanismen wie SWRL erlauben somit die konsistente Abbildung von Regeln und Fakten auf konkrete Inferenzsysteme wie zum Beispiel CLIPS/JESS oder auch Prolog. In diesen Inferenzsystemen können dann schließlich die für das Semantic Web benötigten Schlüsse über die mit Regelsprachen formulierten Aussagen gezogen werden. Das folgende Kapitel stellt nun mit Prolog ein solches Inferenzsystem vor.

3 Prolog

Um im Semantic Web über die mit Hilfe von Regelsprachen definierten Aussagen und Regeln logisch schließen zu können, benötigt man ein Logik-Framework. Hier kommen potentiell auch die verschiedenen Systeme in Frage, die aus der Logik-Programmiersprache Prolog entwickelt wurden. Dieses Kapitel stellt deshalb die Programmiersprache Prolog näher vor und zeigt anhand des Prolog-Systems *SWI-Prolog* wie mit Prolog Inferenzen gezogen werden können.

Der Name Prolog steht für *Programming in Logic*. Der Grund hierfür ist, dass die Syntax von Prolog an die Prädikatenlogik erster Stufe¹³ angelehnt ist. Prolog gehört in der Familie der deklarativen Programmiersprachen somit zu den logischen Programmiersprachen wie beispielsweise auch JESS oder OPS-5. Abschnitt 3.1 zeigt im Folgenden einen Rückblick auf die Geschichte der Programmiersprache Prolog und der daraus entwickelten Systeme.

3.1 Geschichte

Im Jahr 1965 veröffentlichte John Alan Robinson an der Rice University in Houston, Texas mit dem Artikel „*A machine-oriented logic based on the resolution principle*“ [Rob65] wichtige Grundlagen zur automatisierbaren Resolution. Die Resolution ist ein Widerlegungsverfahren, das versucht aus der Verneinung einer Aussage einen logischen Widerspruch herzuleiten anstatt direkt deren Allgemeingültigkeit zu zeigen. Auf diese Grundlagen geht ein Algorithmus zur Unifikation von prädikatenlogischen Formeln zurück, der für den Nachweis der Unerfüllbarkeit einer prädikatenlogischen Formel entscheidend ist.

Aufgrund dieser Erkenntnisse implementierte Alain Colmerauer im Jahr 1972 in Marseilles mit dem ersten Prolog-Interpreter ein effizientes System zur Resolution. Die Theorie stammte von Robert Kowalski, einem Forscher der Universität Edinburgh. Maarten van Emden, ebenfalls Forscher der Universität Edinburgh, war für die experimentelle Vorführung zuständig. Prolog baute die bis dahin vorhandene Logikprogrammierung somit zu einer echten Programmiersprache aus.

¹¹OWL Lite/DL schränken OWL soweit ein, dass sie vollständige Inferenz ermöglichen.

¹²Entscheidbare Untermenge der Regel-Markup-Sprache RuleML (Rule Markup Language).

¹³*englisch*: First Order Predicate Logic (FOPL)

1977 entwickelte David D.H. Warren an der Universität von Edinburgh einen ersten Prolog-Compiler. Dieser implementierte den so genannten DEC-10 Standard, der heute auch als Edinburgh Standard bezeichnet wird. Dieser Prolog-Standard ist bis heute sehr weit verbreitet.

Durch die Wahl von Prolog als Sprache für Japans *5th Generation Computer Project* erlangte Prolog im Jahr 1981 weltweite Akzeptanz als Programmiersprache. In den Jahren 1983 bis 1993 verbreiteten sich verschiedenste Prolog-Systeme und Prolog schaffte den Sprung von den Universitäten in die Industrie. Deshalb bezeichnete Peter Van Roy diese Zeit 1993 in seinem gleichnamigen Artikel auch als „*The Wonder Years of Sequential Prolog Implementation*“ [VR94]. In den Jahren 1995 und 2000 wurde mit den ISO-Prolog-Standards Teil I und II schließlich auch ein ISO-Standard für Prolog verabschiedet.

Bis heute haben sich viele Prolog-Systeme entwickelt und Prolog wurde in verschiedenste Programmierumgebungen eingebettet. Neben kommerziellen Systemen wie SICStus-, QUINTUS- oder LPA-Prolog (für Windows) und kostenlosen Systemen wie LPA-Prolog für DOS gibt es auch freie Prolog-Systeme wie beispielsweise Ciao- oder SWI-Prolog. Bei den freien Prolog-Systemen hat vor allem SWI-Prolog eine weite Verbreitung erlangt. SWI-Prolog bietet neben seiner Stabilität eine Vielzahl von Erweiterungen, darunter mit XPCE auch ein Toolkit zum Entwickeln von grafischen Applikationen in Prolog.

Abschnitt 3.2 stellt im Folgenden mit den Hornklauseln die logische Grundlage vor, auf der die Programmiersprache Prolog basiert.

3.2 Hornklauseln

Prolog-Programme sind Sammlungen von Hornklauseln. Diese wurden erstmals 1951 von Alfred Horn vorgestellt, siehe [Hor51]. Hornklauseln sind Disjunktionsterme mit maximal einem positiven Literal. Dadurch ergeben sich drei mögliche Arten von Hornklauseln, genannt *Regeln*, *Fakten* und *Anfragen*.

Regeln sind Hornklauseln mit genau einem positiven Literal P und n negativen Literalen P_1, P_2, \dots, P_n . Dies lässt sich sowohl in einer Mengen- als auch in einer Regelschreibweise definieren:

$$\{P, \neg P_1, \neg P_2, \dots, \neg P_n\} \quad \text{bzw.} \quad P \leftarrow P_1 \wedge P_2 \wedge \dots \wedge P_n$$

Analog zu den Regeln des Semantic Web bezeichnet man dabei P als Konklusion und P_1, P_2, \dots, P_n als Prämissen. Der Pfeil in der Regelschreibweise lässt sich als umgedrehter Implikationspfeil interpretieren. Das heißt wenn P_1, P_2, \dots, P_n gelten, gilt auch P .

Hornklauseln mit genau einem positiven Literal P und ohne negative Literale nennt man Fakten. Fakten entsprechen somit Regeln, deren Konklusion ohne Angabe von Prämissen immer wahr ist:

$$\{P\} \quad \text{bzw.} \quad P \leftarrow \quad (\text{oder kurz : } P)$$

Die dritte Möglichkeit sind Anfragen. Diese bestehen aus n negativen Literalen P_1, P_2, \dots, P_n und keinem positiven Literal und stellen somit Regeln ohne Konklusion dar:

$$\{\neg P_1, \neg P_2, \dots, \neg P_n\} \quad \text{bzw.} \quad \leftarrow P_1 \wedge P_2 \wedge \dots \wedge P_n$$

Fakten, Regeln und Anfragen in Prolog unterscheiden sich davon lediglich in der Notation (siehe Abschnitt 3.4). Die Vereinigung aller Regeln und Fakten bezeichnet man als (Hornklausel-) Wissensbasis. An eine solche Wissensbasis können dann Anfragen gestellt werden. Dabei gilt die so genannte Closed World Assumption (CWA), das heißt alles was nicht explizit vom System als wahr bewiesen werden kann, wird als falsch angenommen.

Ein Prolog-Programm aus Fakten, Regeln und Anfragen kann entweder aus einer Datei eingelesen und abgearbeitet oder interaktiv im Interpreter eingegeben werden. Dabei ist es möglich, Regeln und Fakten zur Laufzeit hinzuzufügen oder zu löschen. Ziel ist es, Anfragen an die so aufgestellte Wissensbasis zu stellen, die dann vom System beantwortet werden. Um Fakten, Regeln und Anfragen in Prolog definieren zu können, gibt es in Prolog verschiedene Datenobjekte. Diese werden nun in Abschnitt 3.3 vorgestellt.

3.3 Datenobjekte

Um in Prolog Regeln, Fakten und Anfragen über Objekte formulieren zu können, stehen mit Konstanten, Variablen und Strukturen drei verschiedene Arten von Datenobjekten¹⁴ zur Verfügung. Die Prolog-Systeme erkennen den Typ eines Objekts anhand seiner syntaktischen Form. Deshalb müssen keine weiteren Informationen wie beispielsweise eine Datentyp-Deklaration angegeben werden, damit Prolog den Typ eines Objektes erkennt. Bei allen Datenobjekten handelt es sich um Zeichenketten, die nach bestimmten Regeln aus Großbuchstaben, Kleinbuchstaben, Ziffern sowie Sonderzeichen gebildet werden können.

3.3.1 Konstanten

Die Konstanten umfassen so genannte *Atome* sowie Zahlen. Atome können auf drei verschiedene Arten gebildet werden:

- (1) Zeichenketten aus Buchstaben, Ziffern und dem Unterstrich '_', beginnend mit einem Kleinbuchstaben: zum Beispiel `sarah` oder `s_J79`
- (2) Zeichenketten aus Sonderzeichen wie beispielsweise `<-->` oder `::==`
- (3) Zeichenketten, eingefasst in Hochkommas wie zum Beispiel `'S_Jones 79'`

Bei der Verwendung von Zeichenketten, die nur aus Sonderzeichen bestehen, ist jedoch etwas Vorsicht geboten. Einige Zeichenketten, die sich auf diese Art bilden lassen, haben in Prolog bereits eine vordefinierte Bedeutung und werden deshalb vom System nicht als Atom behandelt, sondern als Steuerzeichen. Ein Beispiel hierfür ist die bei der Definition von Regeln benötigte Zeichenkette `':-'` (siehe Abschnitt 3.4.3). Die in Prolog darstellbaren Zahlen umfassen sowohl ganze Zahlen (*Integer*) als auch reelle Zahlen (*Real*).

¹⁴In Prolog auch als *Terme* bezeichnet.

3.3.2 Variablen

Variablen sind Zeichenketten bestehend aus Buchstaben, Ziffern und dem Unterstrich '_'. Variablen beginnen immer mit einem Großbuchstaben oder dem Unterstrich, wie zum Beispiel `X`, `Sarah_J79`, `_` oder `_x23B`. Der Gültigkeitsbereich einer Variablen umfasst die jeweilige Klausel, in der die Variable auftritt. Das heißt tritt eine Variable `X15` in mehreren verschiedenen Klauseln auf, handelt es sich dabei um verschiedene Variablen. Jedes Auftreten der Variablen `X15` innerhalb derselben Klausel bedeutet hingegen dieselbe Variable.

Eine besondere Variable ist die so genannte *anonyme Variable*, die nur aus einem einzelnen Unterstrich besteht. Diese Variable kann für jede Variable eingesetzt werden, die nur einmal innerhalb einer Klausel vorkommt. Ein Beispiel für die Anwendung der anonymen Variablen zeigt Abschnitt 3.4.2.

3.3.3 Strukturen

Strukturierte Objekte (*Strukturen*) sind Objekte, die aus mehreren Komponenten zusammengesetzt sind. Die interne Behandlung erfolgt jedoch als einzelnes Objekt. Die Komponenten selbst können wiederum jede Art von Datenobjekt sein. Beispielsweise kann das Datum als Struktur mit den drei Komponenten Tag, Monat und Jahr gesehen werden. Um die Komponenten zu einem einzelnen Objekt zusammenzufügen zu können, benötigt man einen geeigneten *Funktor*. Ein geeigneter Funktor für eine Struktur, die das Datum repräsentieren soll, wäre zum Beispiel `date`. Damit lässt sich dann das Datum „1. Mai 2005“ in der an die Prädikatenlogik angelehnte Prolog-Syntax schreiben als `date(1, may, 2001)`. Dabei ist `date` der Funktor, `1` und `2001` sind Integerzahlen und `may` ist ein Atom. Alle Tage im Mai 2005 können zum Beispiel mit der folgenden Struktur repräsentiert werden (vgl. Abbildung 2): `date(Day, may, 2001)`. `Day` ist eine Variable und kann zu einem späteren Zeitpunkt in der Ausführung zu jedem Objekt instantiiert werden.

Alle strukturierten Objekte werden von Prolog intern in der Form eines Baumes repräsentiert. Die Wurzel des Baumes ist der jeweilige Funktor und die Komponenten bilden die Kinder des Wurzelknotens. Abbildung 2(b) zeigt die Baumdarstellung anhand des Beispiels `date(Day, may, 2001)`. Ist eine der Komponenten selbst wiederum eine Struktur, so stellt diese einen Unterbaum des Baumes dar, der das gesamte Objekt repräsentiert.



Abbildung 2: Das Datum als Struktur: (a) in Prolog-Syntax; (b) als Baum.

Als Konvention für die folgenden Beispiele gilt, dass sowohl Strukturen als auch Konstanten nur mit Kleinbuchstaben dargestellt werden und Variablen immer mit einem Großbuchstaben beginnen und ansonsten nur aus Kleinbuchstaben bestehen.

3.4 Grundlegende Konzepte

Mit Hilfe der in Abschnitt 3.3 vorgestellten Datenobjekte lassen sich nun die Hornklauseln, das heißt die Fakten, Anfragen und Regeln, auch in Prolog definieren. Die Syntax dieser grundlegenden Konzepte von Prolog ist dabei sehr stark an die Regelschreibweise der Hornklauseln angelehnt (vgl. Abschnitt 3.2).

3.4.1 Fakten

Fakten dienen der Repräsentation von Objekten und Sachverhalten. Will man beispielsweise Verwandtschaftsverhältnisse wie „Pam ein Elternteil von Bob“ oder „Pat ist ein Elternteil von Jim“ darstellen (siehe Abbildung 3(a)), so kann man diese mit der Prädikatenlogik-Syntax wie folgt als Strukturen definieren:

```
parent(pam, bob).  
parent(pat, jim).
```

Auf diese Weise kann man mit Hilfe von Fakten ganze Wissensbasen aufstellen. Das Beispiel aus Abbildung 3(a) kann man so zu einem Stammbaum (siehe Abbildung 3(b)) erweitern, indem man Fakten über weitere Familienmitglieder und deren Verwandtschaftsverhältnisse zu der Wissensbasis hinzufügt:

```
parent( pam, bob). % Pam is a parent of Bob  
parent( tom, bob). % Pat is a parent of Jim  
parent( tom, liz). % ...  
parent( bob, ann).  
parent( bob, pat).  
parent( pat, jim).
```

Listing 1: Einfache Wissensbasis über einen Familienstammbaum.

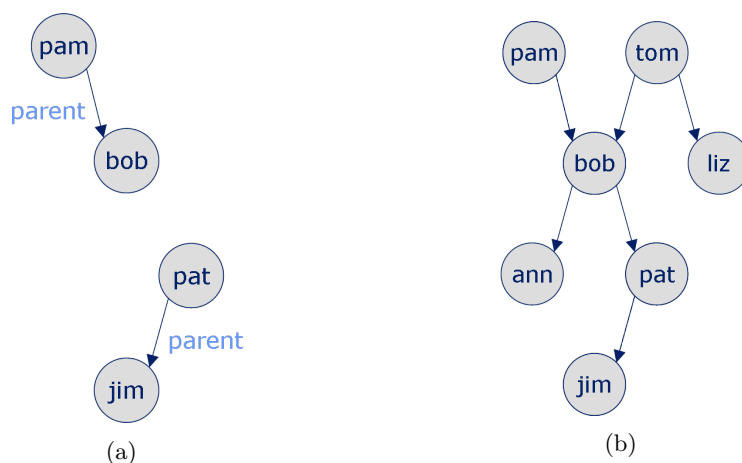


Abbildung 3: Grafische Darstellung der Relation `parent`: (a) einzelne Fakten; (b) Familienstammbaum.

3.4.2 Anfragen

An eine gegebene Wissensbasis lassen sich nun Anfragen stellen, die vom System beantwortet werden. Dazu betreibt Prolog eine systematische Antwortfindung durch Resolution. Ist die Antwort positiv, ist sie logisch ableitbar. Ist die Antwort negativ, kann aufgrund der gegebenen Datenbasis keine positive Antwort abgeleitet werden (vgl. Closed World Assumption). Listing 2 zeigt die Anfrage „Ist Bob ein Elternteil von Pat?“. Das System antwortet mit **yes**, das heißt für die angefragte Wissensbasis aus Listing 1 ist Bob ein Elternteil von Pat.

```
?- parent( bob, pat).  
  
yes
```

Listing 2: Anfrage „Ist Bob ein Elternteil von Pat?“

Auf die Anfragen „Ist Liz ein Elternteil von Pat?“ und „Ist Tom ein Elternteil von Ben?“ antwortet das System jeweils mit **no** (siehe Listing 3). Bei der ersten Anfrage sind Liz und Pat dem System zwar bekannt, aber die Relation **parent** ist für diese beiden Objekte in der Wissensbasis nicht erfüllt. Bei der zweiten Anfrage ist dem System das Objekt Ben gänzlich unbekannt, weshalb das System aufgrund der geltenden Closed World Assumption mit **no** antwortet.

```
?- parent( liz, pat).  
  
no  
  
?- parent( tom, ben).  
  
no
```

Listing 3: Anfragen die von Prolog mit **no** beantwortet werden.

Will man wissen, *von wem* Bob ein Elternteil ist, so kann man diese Anfrage mit Hilfe einer Variablen formulieren:

```
?- parent( bob, X).  
  
X = ann;  
X = pat;  
no
```

Listing 4: Anfrage mit Variable: „Von wem ist Bob ein Elternteil?“

Auf die Anfrage `?- parent(bob, X).` antwortet das System zunächst mit `X = ann`, das heißt Bob ist ein Elternteil von Ann. An dieser Stelle kann der

Benutzer im Interpreter entscheiden, ob das System weitere Antworten finden, oder die Suche an dieser Stelle abbrechen soll. Veranlasst man das System dazu, eine weitere Antwort zu finden¹⁵, erhält man als weitere Antwort `X = pat`. Lässt man das System nun nochmals weiter suchen, antwortet das System mit `no`, das heißt es konnten keine weiteren Antworten gefunden werden.

Weiterhin ist es auch möglich, mehrere Variablen in einer Anfrage zu verwenden. So gibt das System auf die Anfrage `?- parent(X, Y) .` alle Paare (X, Y) aus, für die X ein Elternteil von Y ist:

```

?- parent( X, Y) .

X = pam
Y = bob;

X = tom
Y = bob;

yes

```

Listing 5: Anfrage mit mehreren Variablen: „Wer ist ein Elternteil von wem?“

In Listing 5 wurde die Suche nach zwei Ergebnissen vom Benutzer abgebrochen. Das System antwortet an dieser Stelle mit `yes`, da mindestens eine positive Antwort gefunden werden konnte.

Der Einsatz der anonymen Variablen ist zum Beispiel nützlich, wenn man bei einer Anfrage nicht wissen will, durch welche konkrete Variablenbelegungen eine Anfrage erfüllt wird, sondern nur ob diese überhaupt erfüllt ist oder nicht. Jedes Auftreten von `'_'` in einer Klausel bedeutet dabei die Verwendung einer neuen anonymen Variablen. Will man beispielsweise nur wissen, ob Bob überhaupt ein Elternteil ist und nicht von welchen Familienmitgliedern, so kann man die Anfrage aus Listing 4 mit der anonymen Variablen wie folgt modifizieren:

```

?- parent( bob, _).

yes

```

Listing 6: Anfrage mit anonymer Variable: „Ist Bob ein Elternteil?“

Das System antwortet auf die Anfrage `?-parent(bob, _)` also nur mit `yes`, ohne dabei die konkreten Instanzen der verwendeten Variablen auszugeben, für die Bob ein Elternteil ist.

An eine gegebene Wissensbasis wie in Listing 1 lassen sich auch komplexere Anfragen stellen, die über das einfache Abfragen von gegebenen Fakten hinausgehen. Will man beispielsweise wissen, wer in dem gegebenen Stammbaum ein Großelternteil von Jim ist, so kann man dies nicht mehr einfach abfragen wie in

¹⁵In SWI-Prolog möglich durch die Eingabe des Zeichens `';`

den bisher gezeigten Beispielen, da keine Fakten über eine Relation *Großeltern-
teil* gegeben sind. Allerdings lässt sich diese Anfrage mit Hilfe der gegebenen
Fakten der Relation *Elternteil* wie in Abbildung 4 dargestellt definieren.

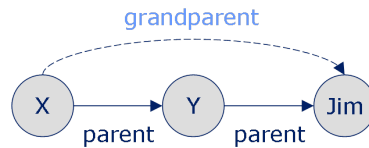


Abbildung 4: Darstellung der Relation *grandparent* mit Hilfe von zwei *parent* Relationen.

Die Frage wird wie folgt in zwei Teilanfragen aufgeteilt:

- Wer ist ein Elternteil von Jim? Sei dies *Y*.
- Wer ist ein Elternteil von *Y*? Sei dies *X*.

Dann ist *X* ein Großelternanteil von Jim. Die beiden Teilanfragen werden also über die Variable *Y* miteinander verknüpft. Auf diese Weise lässt sich die Frage sehr leicht als Prolog-Anfrage formulieren, wobei die beiden einzelnen Teile der Anfrage mit Hilfe eines Kommas konjunktiv mit einander verknüpft werden:

```
?- parent( Y, Jim), parent( X, Y).

X = bob
Y = pat;

yes
```

Listing 7: Anfrage: „Wer ist ein Großelternanteil von Jim?“

Das System findet mit *X = bob* und *Y = pat* also genau eine Antwort. Das heißt Pat ist ein Elternteil von Jim und Bob wiederum ein Elternteil von Pat (siehe auch Abbildung 3(b)). Deshalb ist Bob ein Großelternanteil von Jim.

3.4.3 Regeln

Regeln dienen der Repräsentation von Abhängigkeiten, Kausalitäten, und Strukturbeziehungen zwischen Objekten. Betrachtet man als Beispiel die Relation *Mutter*, so wäre der naive Ansatz, die einzelnen Fakten über diese Relation zu der Wissensbasis hinzuzufügen:

```
mother( pam, bob). % Pam is the mother of Bob
mother( pat, jim). % Pat is the mother of Jim
```

Listing 8: Beispiel: Naive Implementierung der Relation *mother*.

Je nach Art und der Größe der Relation kann dies jedoch sehr aufwendig sein. Eleganter wäre es, die Relation *Mutter* mit Hilfe der Relation *Elternteil* sowie Informationen über das Geschlecht der einzelnen Familienmitglieder zu definieren. Dazu formulieren wir zwei neue Relationen *male* und *female*, um das Geschlecht der Familienmitglieder in der Wissensbasis angeben zu können:

```
parent( pam, bob). % Pam is a parent of Bob
parent( tom, bob). % Pat is a parent of Jim
parent( tom, liz). % ...
parent( bob, ann).
parent( bob, pat).
parent( pat, jim).

female( pam). % Pam is female
female( liz). % ...
female( ann).
female( pat).

male( tom). % Tom is male
male( bob). % ...
male( jim).
```

Listing 9: Wissensbasis über einen Familienstammbaum mit Informationen über das Geschlecht der Familienmitglieder.

Diese Wissensbasis dient im Folgenden als auch Grundlage für weitere Beispiele und Definitionen. Mit Hilfe dieser Informationen lässt sich die Relation *Mutter* wie folgt definieren:

Für alle X, Y gilt: X ist die Mutter von Y falls
 X ein Elternteil von Y ist und X weiblich ist.

Dies entspricht einer Hornklausel-Regel mit der Konklusion „ X ist die Mutter von Y “ und den Prämissen „ X ist ein Elternteil von Y “ und „ X ist weiblich“. Dies lässt sich in der in Abschnitt 3.2 gezeigten Regelschreibweise schreiben als:

$$mother(X, Y) \leftarrow parent(X, Y) \wedge female(X)$$

Diese Hornklausel lässt sich nun sehr einfach in eine Prolog-Regel übersetzen:

```
mother( X, Y ) :- parent( X, Y ), female(X).
```

Listing 10: Regel-Definition der Relation *mother*.

Der Pfeil der Hornklausel wird in Prolog zu `:-` und das logische UND wird durch ein Komma übersetzt. Ein Punkt schließt die Regel ab. Analog zu den Hornklausel-Regeln bezeichnet man auch in Prolog den linken Teil der Regel als Regel-Kopf, Konklusion oder LHS und den rechten Teil als Regel-Körper, Prämisse oder RHS.

3.5 Weiterführende Konzepte

Die in den vorangegangenen Abschnitten vorgestellten Konzepte dienen als Grundlage für die Arbeit mit Prolog. Zur Beantwortung der Anfragen verwendet Prolog mit Matching und Backtracking zwei wichtige Konzepte, die neben dem Konzept der Rekursion in den folgenden Abschnitten vorgestellt werden.

3.5.1 Rekursion

Eines der wichtigsten weiterführenden Konzepte in Prolog ist die Rekursion. Denn nur mit Hilfe der Rekursion lassen sich in Prolog Probleme mit signifikanter Komplexität lösen. Betrachtet man als Beispiel die Relation *Vorfahre*, so wäre der naive Ansatz diese Relation in Prolog zu implementieren, jeden einzelnen Grad der Verwandtschaft als einzelne Prolog-Regel zu definieren:

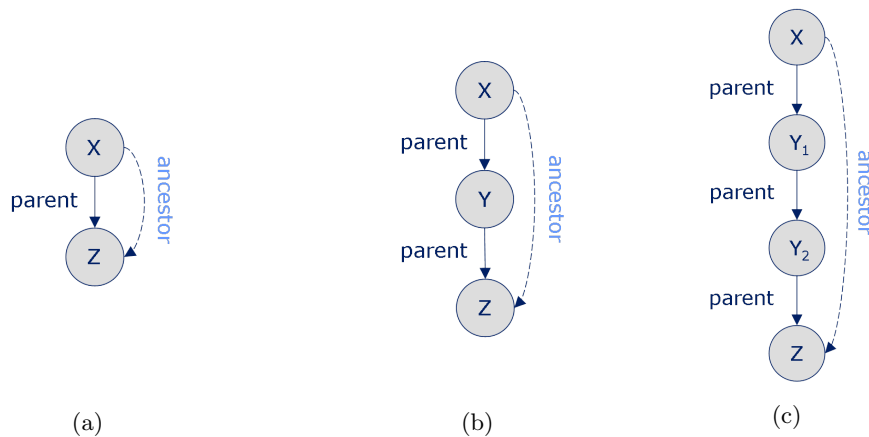


Abbildung 5: Naive Implementierung der Relation `ancestor`: (a) direkte Vorfahren; (b) und (c) indirekte Vorfahren.

Die erste Regel beschreibt die direkten Vorfahren, das heißt die Eltern (siehe Abbildung 5(a)):

- Ist X ein Elternteil von Z , dann ist X auch ein Vorfahre von Z .

Die weiteren Regeln beschreiben dann die indirekten Vorfahren (vgl. Abbildung 5(b) und 5(c)):

- Wenn X ein Elternteil von Y ist, und Y wiederum ein Elternteil von Z (d.h. X ist ein Großelternteil von Z), so ist X ein Vorfahre von Z .
- Wenn X ein Elternteil von Y_1 ist, Y_1 ein Elternteil von Y_2 und Y_2 wiederum ein Elternteil von Z (d.h. X ist ein Ur-Großelternteil von Z), so ist X ein Vorfahre von Z .
- usw.

Dieser Ansatz ist jedoch sehr umständlich und vor allem unvollständig, da es unendlich viele Verwandtschaftsgrade gibt, die auf diese Weise definiert werden

müssten. Man muss also an einer Stelle aufhören die Relation *Vorfahre* auf diese Weise zu definieren. Deshalb kann die Vorfahr-Relation mit dieser Definition nur bis zu einem bestimmten Verwandtschaftsgrad dargestellt werden. Es gibt jedoch auch eine elegante und vollständige Definition, mit der Vorfahren in einer beliebigen Tiefe dargestellt werden können. Die direkten Vorfahren können dabei wie beim naiven Ansatz definiert werden (vgl. Abbildung 5(a)):

X ist ein (direkter) Vorfahre von Z , falls X ein Elternteil von Z ist.

Die indirekten Vorfahren werden nun wie folgt definiert:

X ist ein (indirekter) Vorfahre von Z , falls es ein Y gibt, sodass X ein Elternteil von Y ist und Y ein *Vorfahre* von Z ist.

Der Schlüssel dieser Formulierung ist die Verwendung von *Vorfahre* selbst in der eigenen Definition. Dies nennt man *Rekursion*. Auf den ersten Blick mag dies verwunderlich sein, wenn man sich die Frage stellt: Kann man während der Definition von etwas dieselbe Definition schon verwenden, obwohl sie noch gar nicht vollständig ist? Dies ist jedoch logisch vollkommen korrekt, wie Abbildung 6 illustriert:

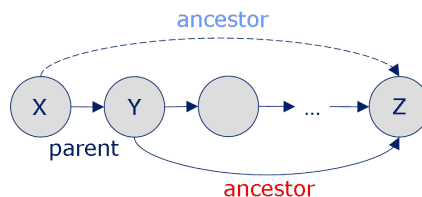


Abbildung 6: Rekursive Definition der Relation **ancestor**.

Die folgenden beiden Klauseln übersetzen diese Regeln in die Prolog-Syntax:

```
ancestor( X, Z) :- % direkter Vorfahre
    parent( X, Z).
```

Listing 11: Relation **ancestor**: Regel zur Definition von direkten Vorfahren.

```
ancestor( X, Z) :- % indirekter Vorfahre
    parent( X, Y),
    ancestor( Y, Z).
```

Listing 12: Relation **ancestor**: Regel zur Definition von indirekten Vorfahren.

Die Relation *Vorfahre* wird also durch zwei Regeln definiert: eine Regel für die direkten Vorfahren und eine Regel für die indirekten Vorfahren. Die beiden Regeln besitzen denselben Kopf und werden vom System als ODER-verknüpfte Alternativen betrachtet. Beschreiben wie in diesem Fall mehrere Klauseln *eine*

Relation, so bezeichnet man diese Klauseln zusammen als Prozedur. Die Reihenfolge der Prozeduren und Regeln spielt dabei eine große Rolle, da Prolog bei der Suche nach Fakten und Regeln die Wissensbasis von oben nach unten abarbeitet (siehe Abschnitt 3.5.3).

3.5.2 Matching

Stellt man an das System mit Hilfe der in Abschnitt 3.5.1 definierten rekursiven Regel die Frage „Vom wem ist Pam eine Vorfahrin?“, so antwortet das System mit Bob, Ann, Pat und Jim (vgl. Abbildung 3(b)):

```
?- ancestor( pam, X).  
  
X = bob;  
X = ann;  
X = pat;  
X = jim;  
no
```

Listing 13: Anfrage mit rekursiver Regel: „Von wem ist Pam eine Vorfahrin?“

Damit Prolog diese Anfrage beantworten kann, muss die rekursive Regel für die Relation `ancestor` angewendet werden, da keine Fakten über diese Relation in der Wissensbasis vorhanden sind. Das Anwenden einer Regel bedeutet das Erfüllen von einem oder mehreren im Regelkörper definierten Zielen. Das Erfüllen dieser Regeln heißt wiederum zu zeigen, dass das Ziel aus den gegebenen Fakten und Regeln logisch folgt. Dazu müssen die Variablen in deren Definition entsprechend der Anfrage instantiiert werden. Dazu benutzt Prolog das so genannte *Matching*. Dieser Prozess erhält als Eingabe zwei Terme und überprüft, ob die beiden Terme *matchen*. Man sagt zwei gegebene Terme *matchen*, falls

- (1) sie identisch sind
- (2) oder die Variablen in beiden Termen zu Objekten instantiiert werden können, sodass die beiden Terme durch die Substitution der Variablen durch diese Objekte identisch werden.

Hier einige Beispiele:

- Die beiden Terme `date(D, M, 2001)` und `date(D1, may, Y1)` *matchen*, da D zu D1, M zu may und Y1 zu 2001 instantiiert werden können.
- Die Terme `date(1, M, Y)` und `date(2, may, 2001)` *matchen nicht*. Der Funktor ist zwar gleich, aber beim ersten Argument handelt es sich um zwei verschiedene Integerzahlen, die nicht substituiert werden können.
- Auch die beiden Terme `date(D, M, 2001)` und `d(D, M, 2001)` *matchen nicht*, obwohl die Argumentenliste die gleiche ist. Der Grund hierfür ist, dass sich die beiden Terme bereits in ihrem Funktor unterscheiden.

Das Matching in Prolog ist vergleichbar mit der Unifikation in der Logik. Allerdings wurde das Matching in den meisten Prolog-Systemen auf eine Art und Weise implementiert, die nicht exakt der Unifikation entspricht. Deshalb verzichtete man bewusst auf die Bezeichnung Unifikation für den in Prolog angewandten Algorithmus und bezeichnete ihn als Matching. Die Unifikation verlangt den so genannten *Occurs Check*, auf den in vielen Prolog-Systemen aus Effizienzgründen verzichtet wird. Bei einem Occurs Check wird für eine gegebene Variable überprüft, ob diese in einem gegebenen Term auftaucht. Diese Überprüfung würde das Matching in Prolog ineffizient machen. Wie auch die Unifikation berechnet Prolog dabei jedoch stets die allgemeinste Instantiierung, für die die beiden Terme matchen. Betrachtet man zum Beispiel die Anfrage

```
?- mother(pam, bob).
```

so sind keine Fakten über die Relation `mother` in der Wissensbasis vorhanden. Allerdings gibt es die Regel

```
mother(X, Y) :- parent(X, Y), female(X).
```

gegen die Prolog die Anfrage matchen kann. Dabei wird `X` mit `pam` und `Y` mit `bob` instantiiert. Die gesamte Regel wird also instantiiert zu

```
mother(pam, bob) :- parent(pam, bob), female(pam).
```

Nun wendet Prolog die Regel an und versucht ihre beiden Prämissen zu erfüllen. Das heißt das neue Ziel, das es zu erfüllt gilt ist also

```
parent(pam, bob), female(pam).
```

Diese Fakten sind in der Wissensbasis erfüllt und Prolog antwortet mit `yes`.

3.5.3 Backtracking

Zur systematischen Antwortfindung benutzt Prolog eine Art der Tiefensuche, das so genannte *Backtracking*. Das heißt bei einer Anfrage startet Prolog eine Tiefensuche nach Regeln, deren Kopf das Ziel matchen. Für die Anfrage

```
?- ancestor( tom, pat).
```

sind die einzigen relevanten Klauseln die Regeln der Relation `ancestor` (siehe Listing 14). Abbildung 7 zeigt den Suchbaum des Backtracking-Algorithmus für diese Anfrage. Die erste anwendbare Regel, die Prolog in der Wissensbasis

```
ancestor( X, Z) :- % ar1
    parent( X, Z).

ancestor( X, Z) :- % ar2
    parent( X, Y),
    ancestor( Y, Z).
```

Listing 14: Regeln der Relation `ancestor`.

findet, ist die in Listing 14 mit `ar1` markierte Regel der `ancestor`-Prozedur. Das heißt `ancestor(X, Z)` gilt, wenn `parent(X, Z)` gilt. `X` wird an dieser Stelle mit `tom` instantiiert und `Y` mit `pat`. Das neue Ziel ist somit

```
parent( tom, pat)
```

Dieses Ziel kann anhand der gegebenen Wissensbasis nicht erfüllt werden. Deshalb geht Prolog zurück zu der Stelle, an der es zuletzt eine Entscheidung getroffen hat (*Backtracking*). In diesem Fall ist dies das Ursprungsziel. Denn an dieser Stelle kann Prolog mit der Regel `ar2` noch eine zweite Regel anwenden, die gegen das aktuelle Ziel gematcht werden kann. Nach dieser Regel gilt `ancestor(X, Z)` wenn `parent(X, Y)` und `ancestor(Y, Z)` gelten. `X` wird hier wieder mit `tom` instantiiert und `Y` wieder mit `pat`. Das neue Ziel lautet also

```
parent( tom, Y), ancestor( Y, pat).
```

Das erste der beiden Teilziele kann durch `Y = bob` erfüllt werden, da der Fakt `parent(tom, bob)` in der Wissensbasis enthalten ist (siehe Listing 9). Das zweite Teilziel wird mit `Y = bob` zum neuen Hauptziel

```
ancestor( bob, pat).
```

Nun wendet Prolog mit der Regel `ar1` wieder die erste passende Regel an, die gefunden wird. Da es sich hier um eine neue Instanz dieser Regel handelt, werden neue Variablen verwendet. Die Regel ist erfüllt, wenn `parent(X', Y')` erfüllt ist. `X'` und `Z'` werden mit `bob` bzw. `pat` instantiiert. Somit lautet das neue Ziel

```
parent( bob, pat).
```

Dieses Ziel ist erfüllt da dies als Fakt in der Wissensbasis steht (vgl. Listing 9) und Prolog antwortet auf die ursprüngliche Anfrage mit `yes`.

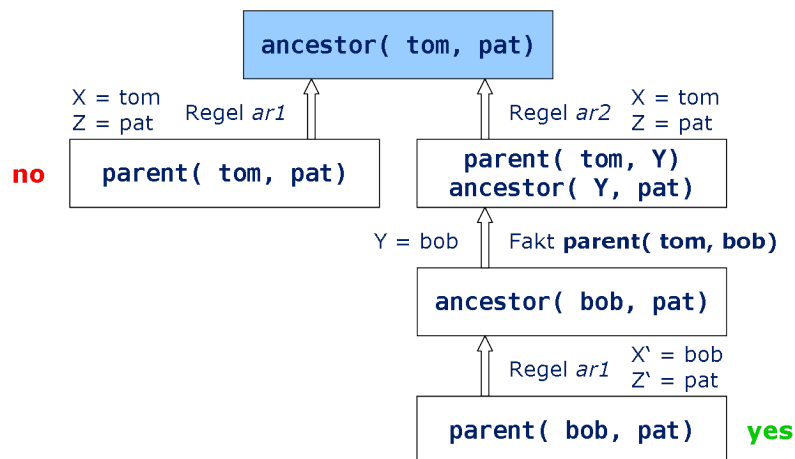


Abbildung 7: Vollständiger Backtracking-Baum für die Erfüllung des Ziels `ancestor(tom, pat)`.

3.6 Vordefinierte Prädikate

Prolog bietet eine Vielzahl von vordefinierten Prädikaten, die dem Programmierer die Arbeit erleichtern, da er sich nicht selbst um die Implementierung dieser Prädikate kümmern muss. Unter anderem stehen Prädikate für grundlegende Rechenoperationen, für die Verarbeitung von Listen, für die Ein- und Ausgabe von Daten sowie zur Steuerung der internen Verarbeitung zu Verfügung. Ein Beispiel für ein vordefiniertes Prädikat ist `different`. Dazu betrachten wir die folgende Prolog-Definition der Relation *Schwester*:

```
sister( X, Y ) :-      % X is a sister of Y if
  parent( Z, X ),      % X and Y have the
  parent( Z, Y ),      % same parent,
  female( X ),         % X is female and
  different( X, Y ).   % X and Y are different.
```

Listing 15: Definition der Relation `sister`.

Das von Prolog zur Verfügung gestellte Prädikat `different` überprüft in dieser Regel, ob `X` und `Y` auch wirklich verschiedene Objekte sind. Verzichtet man in der Definition auf diese Überprüfung, antwortet Prolog auf die Anfrage

```
?- sister( X, pat ).
```

mit `X = ann` und `X = pat`. Das heißt Pat wäre für Prolog ohne diese Überprüfung eine Schwester von sich selbst.

3.6.1 Arithmetik

Für grundlegende arithmetische Operationen stehen unter anderem Prädikate für die Addition, Subtraktion, Multiplikation und Division zur Verfügung. Abbildung 8(a) zeigt eine Übersicht über die wichtigsten Prädikate für grundlegende arithmetische Operationen. Betrachtet man nun beispielsweise die Anfrage

```
?- X = 1 + 2.
```

so antwortet Prolog darauf mit `X = 1 + 2` und nicht mit `X = 3` wie man erwarten würde. Der Grund hierfür ist, dass diese Anfrage nur eine andere Schreibweise ist für

```
?- X = +(1, 2).
```

Und nichts veranlasst Prolog an dieser Stelle dazu, den Operator `'+'` auch wirklich anzuwenden. Um dies zu erreichen gibt es in Prolog das vordefinierte Prädikat `'is'`. Formuliert man mit Hilfe von diesem Prädikat die Anfrage um zu

```
?- X is 1 + 2.
```

antwortet das System wie gewollt mit `X = 3`. Auch für den Wertevergleich bietet Prolog eine Vielzahl vordefinierter Prädikate. Eine Übersicht über die in Prolog

verfügbaren Prädikate zum Wertevergleich zeigt Abbildung 8(b). Hier gilt es zu beachten, dass die Gleichheit von Werten mit dem Prädikat `'=:='` überprüft wird, und nicht mit dem Prädikat `'='`. Bei der Verwendung des Prädikats `'='` würde Prolog nicht die Werte der beiden Objekte miteinander vergleichen, sondern prüfen ob die beiden Objekte matchen (siehe Abschnitt 3.5.2).

<p><code>+</code> : Addition <code>-</code> : Subtraktion <code>*</code> : Multiplikation <code>/</code> : Gleitkomma Division <code>**</code> : Potenz <code>//</code> : Ganzzahl Division <code>mod</code> : Modulo</p>	<p><code>></code> : größer <code><</code> : kleiner <code>>=</code> : größer-gleich <code>=<</code> : kleiner-gleich <code>=/=</code> : Ungleichheit <code>:=:=</code> : Gleichheit</p>
(a)	(b)

Abbildung 8: Vordefinierte Prädikate in Prolog: (a) grundlegende arithmetische Operationen; (b) Prädikate für den Wertevergleich.

3.6.2 Der Cut-Operator

Wie bereits erwähnt, spielt die Reihenfolge der Klauseln eine wichtige Rolle. Eine weitere Möglichkeit die Verarbeitung innerhalb des Prolog-Systems zu steuern ist der so genannte *Cut-Operator* `'!'`. Dies ist eines der wichtigsten und mächtigsten vordefinierten Prädikate. Mit dem Cut-Operator ist es möglich, das automatische Backtracking zu verhindern. Auf der einen Seite ist das automatische Backtracking zwar ein nützliches Konzept, weil es den Programmierer von der Last befreit, das Backtracking explizit selbst zu programmieren, andererseits kann unkontrolliertes Backtracking jedoch zu Ineffizienz führen.

Um die Anwendung des Cut-Operators zu demonstrieren, sehen wir uns ein Beispiel an, dessen Ausführung unnötiges Backtracking verursacht. Dazu betrachten wir die Stufenfunktion f aus Abbildung 9. Die Relation zwischen X und Y kann durch drei Regeln beschrieben werden:

Regel 1: Für $X < 3$ ist $Y = 0$

Regel 2: Für $3 \leq X$ und $X < 6$ ist $Y = 2$

Regel 3: Für $6 \leq X$ ist $Y = 4$

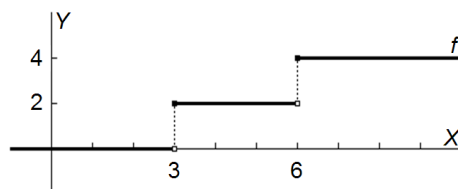


Abbildung 9: Eine zweistufige Funktion.

```

f(X,0) :- X < 3.           % Regel f1
f(X,2) :- 3 =< X, X < 6. % Regel f2
f(X,4) :- 6 =< X.         % Regel f3

```

Listing 16: Die Stufenfunktion aus Abbildung 9 als binäre Relation in Prolog.

Dies lässt sich wie in Listing 16 dargestellt als binäre Relation in Prolog definieren. Betrachtet man sich nun für diese Definition von f den Backtracking-Baum für die Anfrage „Ist $X = 1$ für $Y > 2$?“, das heißt für

$?- f(1,Y), 2 < Y.$

(siehe Abbildung 10(a)), ist das zu erfüllende Hauptziel zunächst

$f(1,Y), 2 < Y.$

Die erste gefundene Regel, die dieses Ziel matcht, ist die Regel $f1$. Y wird mit 0 instantiiert und die Regel $f1$ angewendet. Das neue Ziel lautet nun:

$1 < 3, 2 < 0.$

Das Teilziel $1 < 3$ ist erfüllt. Das zweite Teilziel wird zum neuen Hauptziel:

$2 < 0$

Dieses Ziel kann jedoch nicht erfüllt werden. Deshalb geht Prolog mit dem Backtracking-Algorithmus zurück zum Ursprungsziel und wendet mit der Regel $f2$ die nächste Regel an, die das Ziel matcht. Y wird nun mit 2 instantiiert und das neue Ziel lautet:

$3 =< 1, 1 < 6, 2 < 2.$

Da 3 nicht kleiner-gleich 1 ist, kann dieses Ziel nicht erfüllt werden und Prolog geht mit dem Backtracking-Algorithmus wieder zurück zum Ursprungsziel. Die letzte Regel, die Prolog nun anwenden kann, ist die Regel $f3$. Nun wird Y mit 4 instantiiert und das neue Ziel lautet:

$6 =< 1, 2 < 4.$

Da 6 nicht kleiner-gleich 1 ist, beendet Prolog an dieser Stelle das Backtracking und antwortet dem Benutzer mit `no` weil während der Suche keine positive Antwort gefunden werden konnte. Die Prolog-Regeln für die Stufenfunktion f lassen sich nun durch den Einsatz des Cut-Operators wie folgt modifizieren:

```

f(X,0) :- X < 3, !.        % f1'
f(X,2) :- 3 =< X, X < 6, !. % f2'
f(X,4) :- 6 =< X.         % f3'

```

Listing 17: Definition der Stufenfunktion f mit Cut-Operator.

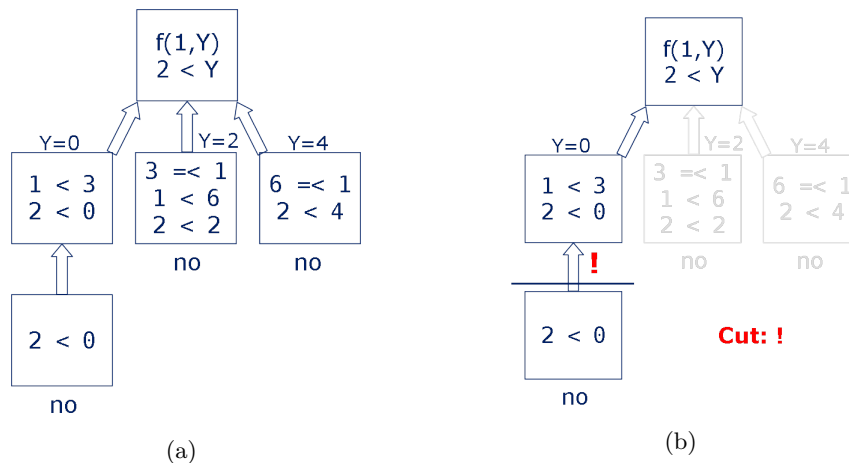


Abbildung 10: Backtracking-Bäume für die Anfrage $?- f(1, Y), 2 < Y$. (a) für die Definition von f ohne Cut-Operator; (b) für die Definition von f mit Cut-Operator.

Das '!' Symbol wird jetzt das Backtracking an den Stellen unterbinden, an denen es im Programm auftaucht. Stellt man nun für diese modifizierte Regeln an das System wieder die Anfrage $?- f(1, Y), 2 < Y$, so erzeugt Prolog denselben linken Zweig wie für die Regeln ohne den Cut-Operator (vgl. Abbildung 10(b) mit Abbildung 10(a)). Bei der Anwendung der Regel $f1'$ merkt sich Prolog jedoch, dass das Programm an dieser Stelle mit einem Cut-Operator markiert ist. Die Suche nach einem positiven Ergebnis schlägt in diesem Zweig wieder bei dem Ziel $2 < 0$ fehl. Nun versucht Prolog mit dem Backtracking-Algorithmus wieder zurückzugehen und trifft an der markierten Stelle auf den Cut-Operator. Prolog bricht das Backtracking deshalb an dieser Stelle ab. Die beiden Alternativen Zweige, die den Regeln $f2'$ und $f3'$ entsprechen, werden nicht generiert. Das heißt der Suchbaum ist gegenüber den Regeln ohne Cut-Operator für dieselbe Anfrage wesentlich kleiner. Das Ergebnis bleibt jedoch gleich. Somit kann in diesem Beispiel durch den Einsatz des Cut-Operators die Suchzeit wesentlich verkürzt werden, ohne das Ergebnis dabei zu beeinträchtigen.

Betrachtet man sich die Regeln der Funktion genauer, so sieht man dass der Einsatz des Cut-Operators an dieser Stelle auch tatsächlich Sinn macht. Denn wenn man schon in Regel $f1'$ festgestellt hat, dass X kleiner ist als 3, kann $3 \leq X \wedge X < 6$ schon nicht mehr gelten, genauso wenig wie $6 \leq X$. Diese beiden Überprüfungen kann man sich durch den gezielten Einsatz des Cut-Operators also ersparen.

Beim Einsatz des Cut-Operators muss man jedoch sehr vorsichtig sein. Ein falsch gesetzter Cut-Operator schneidet unter Umständen ungewollt Lösungen im Suchbaum ab, die eigentlich durch die Suche gefunden werden sollten. Im Extremfall kann ein falsch gesetzter Cut-Operator dazu führen, dass für eine Anfrage gar keine Lösungen mehr von Prolog gefunden werden können.

Nachdem in letzten beiden Abschnitten die Konzepte vorgestellt wurden, die in Prolog verwendet werden, stellt Abschnitt 3.7 nun eine Möglichkeit vor, wie man Prolog für das Semantic Web nutzen kann. Außerdem werden weitere Anwendungsgebiete und der Einfluss von Prolog auf andere Entwicklungen vorgestellt.

3.7 Einfluss und Anwendung

Mit Hilfe der *Semantic Web Library* bietet SWI-Prolog eine Möglichkeit, Dokumente aus dem Semantic Web im RDF-, RDF-S- und OWL-Format zu lesen, zu speichern und anzufragen um beispielsweise SWRL-Regeln zu verarbeiten. Mit dem *SGML/XML Parser* können auch in XML-Syntax definierte Regeln mit SWI-Prolog eingelesen werden. Dadurch kann Prolog als Logik-Framework im Semantic Web zur Verarbeitung von Regeln eingesetzt werden.

Prolog hat im Laufe der Zeit auch viele andere Entwicklungen beeinflusst, darunter die Entwicklung der Constraint Logic Programmierung, die Entwicklung deduktiver Datenbanken und die Entwicklung von modernen KI-Systemen¹⁶. Weitere Anwendungsgebiete von Prolog sind beispielsweise die Sprachanalyse in der Computerlinguistik, automatisiertes Beweisen und Prototyping. Auf dem Gebiet der künstlichen Intelligenz wird Prolog zum Problemlösen, zur Wissensverarbeitung, in Expertensystemen und bei Spielen eingesetzt.

Neben diesen wissenschaftlichen Gebieten findet Prolog auch in kommerziellen Produkten Anwendung, zum Beispiel beim Systemmanagement von Geschäftsanwendungen. Ein Beispiel hierfür wäre die auf BIM-Prolog basierte *Tivoli Enterprise Console* von IBM.

4 Zusammenfassung

Um beispielsweise Anfragen im World Wide Web automatisch anhand ihres Bedeutungsinhaltes bearbeiten zu können, muss das bislang rein symbolisch repräsentierte Wissen auch für Computer verständlich gemacht werden. Deshalb ist es eines der aktuellen Forschungsziele der Wissensverarbeitung, ein semantisches Web zu entwickeln, in dem zusätzlich zu der symbolischen Repräsentation auch der Inhalt der Daten auf eine Art und Weise repräsentiert werden, die es Computern ermöglicht, diesen zu verstehen. Mit dem Semantic Web Layer Cake existiert auch bereits eine Zielvorgabe für die Umsetzung dieses Ziels. An dessen Spitze steht dabei als Ziel das Vertrauen. Zum Erreichen dieses Zieles muss die Glaubwürdigkeit von im Web formulierten Aussagen überprüft werden. Das heißt es muss versucht werden, solche Aussagen zu beweisen oder zu widerlegen. Um dies automatisiert erledigen zu können, müssen die Zusammenhänge im Web mit Hilfe von Regeln und Aussagen logisch beschrieben werden. Denn daraus leitet sich die Möglichkeit ab, in Inferenzsystemen Schlüsse über diese Aussagen ziehen zu können. Momentan gibt es jedoch noch keinen Standard für eine Regelsprache im Semantic Web. Mit der Semantic Web Rule Language

¹⁶Auch als *5th Generation Computer* bezeichnet.

SWRL wurde jedoch 2004 bereits ein Vorschlag dafür beim World Wide Web Consortium eingereicht.

Ein mögliches Logik-Framework, mit dem Inferenzen über diese Regeln gezogen werden können, ist Prolog. Prolog hat sich seit seiner Entwicklung in den siebziger Jahren immer weiter entwickelt und verbreitet. Die Basis von Prolog bilden Fakten, Regeln und Anfragen, so genannte Hornklauseln. Eine Sammlung von Fakten und Regeln stellt dabei eine Wissensbasis zur Verfügung, an die Anfragen gestellt werden können. Die Anfragen werden dann vom Prolog-System beantwortet. Zur Beantwortung solcher Fragen verfügt Prolog über eine kleine Menge grundlegender Mechanismen wie Matching, Backtracking und eine Baum-basierte Datenstrukturierung.

Aufgrund seines deskriptiven Charakters eignet sich Prolog speziell zur Lösung von Problemen, die (strukturierte) Datenobjekte und deren Beziehungen behandeln. Komplizierte mathematische Berechnungen, bei denen die prozeduralen Aspekte im Vordergrund stehen, sind deshalb im Gegensatz zu imperativen Programmiersprachen wie JAVA oder C++ nur sehr eingeschränkt möglich. Somit ist Prolog eine mächtige Sprache für das Gebiet der künstlichen Intelligenz und der nicht-numerischen Programmierung. Trotzdem bietet Prolog auch Prädikate, die grundlegende arithmetische Operationen ermöglichen. Hinzu kommen weitere Prädikate zur Steuerung der internen Verarbeitung, wie der mächtige Cut-Operator zum Unterbinden des automatischen Backtrackings sowie Prädikate zur Verarbeitung von Listen oder zur in Ein- und Ausgabe von Daten.

Durch die SWI-Prolog XPCE/Semantic Web Library und den SWI-Prolog XML Parser bietet sich auch die Möglichkeit, Prolog als Logik-Framework für das Semantic Web zu verwenden. Mit diesen Erweiterungen ist es möglich, Dokumente des Semantic Web zu verarbeiten (siehe Abschnitt 3.7). Bisher bringt jedoch kein Prolog-System eine native Unterstützung für diese Dokumente mit sich. Das heißt es müssen wie für SWI-Prolog spezielle Erweiterungen dafür vorhanden sein. Die Verarbeitung dieser Dokumente ist deshalb nicht in allen Prolog-Systemen möglich.

Zusammenfassend lässt sich sagen, dass das Semantic Web noch ein sehr großes Forschungsgebiet ist, auf dem noch viel Arbeit notwendig ist um das Ziel des Vertrauens zu erreichen. Momentan gilt es, einen Standard für eine Regelsprache im Semantic Web zu verabschieden und ein passendes Logik-Framework zu finden bzw. zu entwickeln. Dieses Framework sollte möglichst genau auf die Bedürfnisse dieser Regelsprache angepasst sein um maximal von den größeren Möglichkeiten, die eine eigene Regelsprache mit sich bringt, zu profitieren. Durch die speziellen Erweiterungen für SWI-Prolog würde sich dieses Inferenzsystem prinzipiell dazu eignen, Regeln wie die der Semantic Web Rule Language zu verarbeiten. Durch die Erweiterungsmöglichkeiten von SWI-Prolog wäre jedoch jederzeit auch eine Unterstützung für eine andere Regelsprache möglich.

Literatur

- [BLHL01] BERNERS-LEE, Tim ; HENDLER, James ; LASSILA, Ora: The Semantic Web. In: *Scientific American* 284 (2001), Mai, Nr. 5
- [Bra00] BRATKO, Ivan: *Prolog Programming for Artificial Intelligence*. 3rd Edition. Addison-Wesley, September 2000. – ISBN 0201-40375-7
- [Die05] DIEDERICH, Gerrit. *Bausteine für die Entwicklung von Semantic Web Anwendungen*. Dezember 2005
- [FWLH03] FENSEL, Dieter ; WAHLSTER, Wolfgang ; LIEBERMAN, Henry ; HENDLER, James: *Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential*. MIT Press, Februar 2003. – ISBN 0-262-06232-1
- [Hor51] HORN, Alfred: On Sentences Which are True of Direct Unions of Algebras. In: *The Journal of Symbolic Logic* 16 (1951), Nr. 1, S. 14–21
- [HPSB⁺04] HORROCKS, Ian ; PATEL-SCHNEIDER, Peter F. ; BOLEY, Harold ; TABET, Said ; GROSOFF, Benjamin ; DEAN, Mike: SWRL: A semantic web rule language combining OWL and RuleML. 2004. – Forschungsbericht. W3C Member Submission
- [Rob65] ROBINSON, John A.: A Machine-Oriented Logic Based on the Resolution Principle. In: *Journal of the ACM* 12 (1965), Nr. 1, S. 23–41. – ISSN 0004-5411
- [VR94] VAN ROY, Peter: 1983–1993: The Wonder Years of Sequential Prolog Implementation. In: *Journal of Logic Programming* 19,20 (1994), S. 385–441
- [Wah06a] WAHLSTER, Wolfgang. *KI = Künftige Informatik: Zwei Megatrends der Informatik*. Juli 2006
- [Wah06b] WAHLSTER, Wolfgang. *Semantische Wende - Informatik für den Menschen*. Dezember 2006