aus

Oliver J. Bott, Peter Fricke, Uta Priss, Michael Striewe (Hrsg.)

Automatisierte Bewertung in der Programmierausbildung

Digitale Medien in der Hochschullehre Band 6
2017, 420 Seiten, br., 42,90 €, ISBN 978-3-8309-3606-0

Waxmann Verlag GmbH
www.waxmann.com info@waxmann.com

5 Automatisierte Programmbewertung in der deduktiven/logischen Programmierung am Beispiel der Prolog-Ausbildung

Tobias Thelen, Helmar Gust und Elmar Ludwig

Zusammenfassung
In der Prolog-Ausbildung als Teil des Studienmoduls „Künstliche Intelligenz“ wird an der Unversität Osnabrück automatisierte Programmbewertung sowohl für den Übungs- als auch den Klausurenbetrieb genutzt. Dieses Kapitel skizziert zunächst die Besonderheiten der Programmiersprache Prolog, legt anschließend die didaktischen Prämissen des Grader-Einsatzes dar und diskutiert schließlich anhand von Beispielaufgaben und Einsatzerfahrungen die Möglichkeiten der autmatisierten Programmbewertung für Prolog.

5.1 Einleitung

5.1.1 Die Programmiersprache Prolog

Die Programmiersprache Prolog wurde zu Beginn der 1970er Jahre in Marseille, Frankreich entwickelt, um natürlichsprachliche Dialogsysteme zu entwerfen (vgl. [CR96]). Prolog ist eine dynamisch typisierte interpretierte Sprache, die im Kern einen Theorembeweiser für eine Teilmenge der Prädikatenlogik enthält. Die Prädikatenlogik wird in vielen Wissenschaftsbereichen verwendet, um Argumente zu formalisieren und daraus folgende Schlussfolgerungen auf ihre Gültigkeit zu überprüfen. Auf diese Weise weist zumindest der Sprachkern von Prolog wohldefinierte semantische Eigenschaften auf. In der Künstlichen Intelligenz wird Prädikatenlogik als ein wichtiger Formalismus zur Wissensrepräsentation verwendet und

Prolog dementsprechend als Programmiersprache, die „intelligentes“ Schlussfolgern modelliert und ermöglicht (vgl. z. B. [Bra01]).

Im Gegensatz zu den meisten verbreiteten Programmiersprachen sind Prolog-Programme deklarativ zu verstehen: Anstatt Schritt für Schritt den Lösungsweg niederzulegen, beschreibt ein Prolog-Programm Ausgangsdaten und Eigenschaften von Lösungen, die dann vom Prolog-Interpreter anhand eines Beweisverfahrens gesucht werden. In wesentlichen Konzepten unterscheidet sich Prolog radikal von anderen Programmiersprachen, so zum Beispiel beim Variablenkonzept. Variablen in Prolog sind logische Variablen, die entweder frei (unbelegt) oder gebunden (belegt) sind und dann ihre Bindung nicht mehr ändern können. Prolog ermöglicht nichtdeterministische Programme, d. h. Programme, die mehr als eine Lösung haben. So ist es typisch, dass ein Prolog-Programm eine Lösung generiert und dann aufgefordert wird, weitere Lösungen zu generieren.

Prolog bietet mit komplexen Termen und Listen zwei eingebaute flexible Datenstrukturen. Auch hier treten wieder Besonderheiten von Prolog zutage: Für den Vergleich und die Zuweisung von Datenstrukturen und Variablen kommt ein Unifikationsverfahren zum Einsatz, das nach Möglichkeiten sucht, Strukturen durch die Belegung darin enthaltener Variablen einander anzugleichen. Dieses Prinzip spielt eine zentrale Rolle für den Beweisprozess und stellt einen wichtigen Schlüssel für das Verständnis von Prolog dar. Die Verarbeitung von Listen kann nur rekursiv erfolgen, so dass Rekursion in Prolog-Programmen der Regelfall und nicht die Ausnahme ist.

Die deklarative Herangehensweise, die Verwendung von Rekursion, eingebaute komplexe Datenstrukturen und der Unifikationsmechanismus sorgen dafür, dass Prolog-Programme in aller Regel sehr kompakt sind. Mit nur wenigen Codezeilen lassen sich komplexe Programme realisieren. Zwei Beispiele sollen einige besonders typische Eigenschaften von Prolog verdeutlichen.

```
sehenswuerdigkeit(osnabrueck, schloss).
sehenswuerdigkeit(osnabrueck, museum).
sehenswuerdigkeit(fuerstenau, schloss).
sehenswuerdigkeit(hannover, schloss).
sehenswuerdigkeit(hannover, see).
sehenswuerdigkeit(bramsche, museum).
besuchstipp(X) :-
    sehenswuerdigkeit(X, schloss).
besuchstipp(X) :-
    sehenswuerdigkeit(X, S1),
    sehenswuerdigkeit(X, S2),
    S1 \= S2.
```

Dieses Prolog-Programm ist ein typisches Beispiel für eine deduktive Datenbank. Das Prädikat sehenswürdig enthält (analog zu einer Datenbanktabelle) Aussagen über Orte und ihre Sehenswürdigkeiten in Form von Prolog-Fakten. An diese Datenbank können zum Beispiel im Prolog-Interpreter interaktiv Anfragen gestellt werden (% leitet Zeilenkommentare ein):

```
% Gibt es in Osnabrück ein Museum?
[1] ?- sehenswuerdigkeit(osnabrueck, museum).
true
% In welcher Stadt gibt es einen See?
[2] ?- sehenswuerdigkeit(X, see).
X=hannover
% In welcher Stadt gibt es Schloss und Museum?
[3] ?- sehenswuerdigkeit(X, schloss),
       sehenswurdigkeit(X, museum).
X=osnabrueck
% In welcher Stadt gibt es ein Museum?
[4] ?- sehenswuerdigkeit(X, museum).
X=osnabrueck ;
X=bramsche ;
false
```

Anfrage [4] zeigt den nichtdeterministischen Charakter von Prolog-Programmen: Anfragen können mehrere gültige Antworten haben, die Prolog sukzessive berechnen kann, bis keine weiteren korrekten Antworten mehr herleitbar sind.

```
[5] ?- besuchstipp(osnabrueck).
true
[6] ?- besuchstipp(X).
X=osnabrueck ; X=hannover ;
X=fuerstenau ; X=osnabrueck ;
X=hannover ;
false
```

Die Regel Besuchstipp besteht aus zwei Teilregeln (Klauseln), die als Oder-verknüpfte Implikationen („`:-`“ steht für den umgedrehten Imkplikationspfeil) zu lesen sind. Eine Stadt ist also dann ein Besuchstipp, wenn sie über ein Schloss verfügt oder mindestens zwei Sehenswürdigkeiten aufweist. Das Oder ist nichtexklusiv, deshalb erscheinen Osnabrück und Hannover, die beide Kriterien erfüllen, in der Ergebnisliste doppelt.

Das zweite Beispiel ist ein rekursiv definiertes Prädikat mit drei Argumenten, das wahr ist, wenn das dritte Argument eine Liste enthält, die als Verkettung der Listen in den ersten beiden Argumenten zu sehen ist.

```
verkette([],L,L).
verkette([H|R],L2,L3) :-
    verkette(R,L2,[H|L3]).
```

Dieses sehr einfach erscheinende Prädikat bedient eine Fülle von Anwendungsfällen, was Prolog-Anfänger häufig vor große Schwierigkeiten stellt. Als Standardlesart wird häufig die Verketten-Operation angenommen, also eine prozedurale Lesart, in der zwei gegebene Listen zu einer dritten, noch unbekannten verkettet werden. Da aber jeder Parameter auch mit einer freien Variablen belegt werden kann, ist diese Möglichkeit nur eine von vielen.

```
[1] ?- verkette([1,2],[3,4],[1,2,3,4]).
true
[2] ?- verkette([1,2,3],[4,5,6],X).
X = [1,2,3,4,5,6]
[3] ?- verkette(X,[4,5,6],[1,2,3,4,5,6]).
X = [1,2,3]
[4] ?- verkette([1,2,3],X,[1,2,3,4,5,6]).
X = [4,5,6]
```

Anfrage [1] prüft, ob die drei gegebenen Listen die Verkettungsrelation erfüllen, Anfrage [2] belegt X mit der Verkettung der beiden gegebenen Listen. Das Prädikat kann aber auch verwendet werden, um bekannte Präfixe (Anfrage [4]) oder Suffixe (Anfrage [3]) abzuspalten. All diese Anfragen haben nur eine Lösung.

Wird nur eine oder gar keine Variable belegt, ergeben sich weitere Möglichkeiten, die mehrere Lösungen haben und dem nichtdeterministischen Programmieren zuzurechen sind.

```
[5] ?- verkette(L1,L2,[1,2,3,4]).
[6] ?- verkette(L, L, L2).
```

Anfrage [5] erzeugt alle möglichen Zerteilungen der als drittes Argument gegebenen Liste. In der interaktiven Verwendung des Prolog-Interpreters müssen all diese Lösungen nacheinander explizit angefordert werden. Bei der Ausführung eines Prolog-Programms (d. h. der Suche nach einem Beweis samt Variablenbelegungen für die gestellte Anfrage) passiert die Generierung einer weiteren Lösung immer dann, wenn sich ein möglicher Lösungsweg als Sackgasse erwiesen hat

und der Prolog-Interpreter weitere Lösungenmöglichkeiten durchprobiert. Denkbar wäre hier zum Beispiel die Verwendung von Anfrage [5] in einem Programm, das prüfen soll, ob es eine Zerlegung einer Liste von Zahlen gibt, so dass beide Teile die gleiche Summe aufweisen. In diesem Programm würde die Zeile `verkette(L1,L2,Liste)` dafür sorgen, dass bei der Suche nach einer Lösung wie im Beispiel alle möglichen Zerlegungen von einer gegebenen `Liste` in zwei Teile generiert werden.

Anfrage [6] verwendet nur freie Variablen, fordert aber, dass das erste und zweite Argument identisch sind. Die Anfrage generiert beliebig viele Lösungen (bis zum Erreichen von Speicherlimits), die alle ausschließlich freie Variablen enthalten. Die Anfrage generiert somit alle Muster für Listen, die als Verdopplung einer Liste betrachtet werden können. Solch eine Form der Mustergenerierung wird häufig für Generate-And-Testansätze verwendet: Ein Generator-Ziel wie in Anfrage [6] generiert sukzessive alle möglichen Muster für erlaubte Lösungen, die anschließend getestet werden. Diese Form der Programmierung ist in der Regel nicht besonders effizient, aber besonders einfach zu implementieren.

Die beiden in diesem Abschnitt präsentierten Beispiele sollten einen kleinen Einblick in die Programmierung mit Prolog bieten und dabei vor allem aufzeigen, dass sehr kompakte Programme bereits komplexe Verhaltensweisen hervorrufen können. Die Gründe dafür liegen in der automatischen Beweisstrategie, der Verwendung logischer Variablen mit Unifikationsmechanismus und der rekursiven Verarbeitung komplexer Datenstrukturen.

5.1.2 Der Studiengang Cognitive Science

Die noch relativ junge Disziplin Kognitionswissenschaft befasst sich in interdisziplinärer Weise mit Fragestellungen rund um das Phänomen (menschlicher) Kognition, d. h. des menschlichen Geistes. An der Universität Osnabrück werden seit 1998 internationale kognitionswissenschaftliche Studiengänge (Bachelor, Master, Promotionsprogramm) eingerichtet, die in großer Breite kognitionswissenschaftliche Fragestellungen abbilden. Die Studiengänge sind mittlerweise sehr gut etabliert und ziehen Studierende mit sehr unterschiedlichen Hintergründen und Interessenspektren in großer Zahl an.

Insbesondere der Bachelorstudiengang zählt mit über 120 Studienanfängern pro Jahr zu den größeren Studiengängen an der Universität Osnabrück und steht an der Schwelle zum Massenfach. Das Bachelorprogramm besteht aus 8 Teilmodulen, die jeweils eine kognitionswissenschaftliche Teildisziplin abdecken. Im Einzelnen sind dies: Computerlingustik, Informatik, Kognitive (Neuro-)Psychologie,

Künstliche Intelligenz, Mathematik, Neuroinformatik, Neurowissenschaft (Neurobiologie, Neurophysiologie) und Philosophie des Geistes. Studierende müssen in jedem dieser Module ein bis zwei Einführungs- bzw. Grundlagenveranstaltungen belegen, und in fünf der acht Module weiterführende Wahlpflichtveranstaltungen belegen. Darüber hinaus gibt es einen individuell zu füllenden Profilbildungsbereich mit 22-33 Leistungspunkten und ein verpflichtendes Auslandssemester.

5.2 Einsatzszenario: Introduction to Artificial Intelligence and Logic Programming

Die Vorlesung/Übung „Introduction to Artificial Intelligence and Logic Programming“ (Intro AI) ist eine Pflichtverstaltung im Modul „Artificial Intelligence“. Sie wird also von allen Studierenden des Bachelorprogramms besucht und ist laut Studienplan für das zweite Fachsemester empfohlen. Zu den Voraussetzungen zählen im Wesentlichen eine Einführung in die Logik und die Informatikeinführung „Informatik A – Algorithmen und Datenstrukturen“, die im ersten Fachsemester besucht werden. Parallel zu „Intro AI“ werden überlicherweise die ebenfalls sehr arbeitsintensiven Vorlesungen und Übungen zur Einführung in die Computerlinguistik und die Philosphie des Geistes belegt.

Die Veranstaltung besteht aus Vorlesung, Übung und Tutorien. Im Vorlesungsteil werden Konzepte eingeführt, die im Übungsteil mit praktischen Programmieranteilen in Prolog verknüpft werden. Studentische Tutoren führen Tutorien für Gruppen von ca. 30 Studierenden durch, die der Wiederholung und Festigung des Wissens dienen. Die von den Studierenden zu erbringenden Leistungen bestehen aus vier Teilen:

- zwei Blöcke von Übungsaufgaben, die in Teams von bis zu vier Studierenden als wöchentliche Hausarbeiten zu erledigen sind,
- zwei Klausuren, eine in der Mitte und eine am Ende des Semesters.

Die Veranstaltung baut inhaltlich vor allem auf der Einführung in die Logik auf und verdeutlicht das Potenzial der Prädikatenlogik erster Stufe als Wissensrepräsentationsformalismus. Von dieser Basis aus wird Prolog als automatisiertes Beweisverfahren motiviert, das (künstliche) Intelligenz als Schlussfolgern aus bekannten Fakten und Regeln darstellt. In den ersten Wochen spielt daher die „Übersetzung“ natürlichsprachlicher Aussagen in Prädikatenlogik und von dort in Prolog eine große Rolle. Neben einer Einführung in die (sehr einfache) Prolog-Syntax steht dabei auch das ungewohnte Variablenkonzept samt Unifikationsverfahren im

Vordergrund. Die erste Hälfte der Veranstaltung wird mit der prozeduralen Semantik (wie arbeitet der Beweiser?), der in Prolog sehr einfachen Implementation kontextfreier Grammatiken und der Verarbeitung komplexer Datenstrukturen, d. h. vor allem der Verwendung von Rekursion zur Verarbeitung von Listen, abgerundet.

In der zweiten Hälfte der Veranstaltungen stehen zunächst extra-logische Eigenschaften von Prolog im Vordergrund, vor allem die Kontrolle des Beweisverfahrens und dynamische Wissensbasen. Anschließend werden einige ausgewählte Probleme aus der Künstlichen Intelligenz (KI) in den Block genommen und mit verschiedenen Verfahren implementiert. Leitendes Grundprinzip ist dabei die Vorstellung von „Intelligenz als Suche“, d. h. Probleme werden so formuliert, dass ihre Lösungen als Suchraum aufgefasst werden können. Wird ein solcher Suchraum als Baum verstanden, lassen sich Baumtraversierungsalgorithmen als Suchstrategien für KI-Probleme anwenden. Auf diese Weise werden uninformierte und informierte Suchverfahren in Prolog implementiert und auf klassische Probleme wie Schiebepuzzle, Wegfindung und Zahlenrätsel angewandt. Den Abschluss bilden Alternativen und Erweiterungen dieser Herangehensweise wie Constraint-Logisches Programmieren, Meta-Programmierung, und Prolog-Varianten wie Datalog und F-Logic.

5.3 Prolog im Übungsbetrieb

Die wöchentlichen Übungsaufgaben der Vorlesung/Übung bestehen überwiegend aus Prolog-Aufgaben. Häufig verwendete Aufgabentypen sind die Formulierung von Prolog-Anfragen, die Vervollständigung von Prolog-Programmen und die vollständige Eigenimplementation von Prolog-Programmen.

Als Grader-Umgebung wird das LMS Stud.IP mit dem Vips-Plugin (s. Kapitel 14 und 20) verwendet und in zwei verschiedenen Arbeitsweisen eingesetzt: Als Auswertungshilfe für die Aufgabenkorrektur und als Arbeitsunterstützung für Studierende. Die Kombination dieser beiden Modi ist ein typisches Merkmal von intelligenten tutoriellen Systemen (ITS), die zusätzlich eine Benutzermodellierungs- und Ablaufsteuerungskomponente haben. Der Prolog-Grader für Vips basiert auf Vorarbeiten, die so ein ITS für Prolog umsetzen ([Bön+99], [Pey+00], [Pey02]), allerdings auch als Unterstützung für den Übungsbetrieb gedacht sind. Auch wenn Rolf Schulmeister den ITS für die Prolog-Programmierung vorwirft, rein akademischen Charakter zu haben ([Sch97, S. 198]), haben sich die hier verfolgten Ansätze in der Praxis vor allem als Auswertungsunterstützung bewährt,

wie auch andere Arbeiten für die Programmbewertung demonstrieren ([Bra+06], [BKW05]).

In letzterem Fall wird nicht die Bewertungsfunktion der Grader-Umgebung verwendet, sondern die Möglichkeit, Prolog-Programme aus der Stud.IP-/Vips-Eingabe heraus auszuführen und die Berechnungsergebnisse angezeigt zu bekommen. Das Lernmanagementsystem wird auf diese Weise zu einer einfachen Entwicklungsumgebung, in der Programme formuliert, getestet und gespeichert werden können. In der Praxis wird diese Möglichkeit relativ selten genutzt, da die Implementation dieser Umgebung vergleichsweise rudimentär ist. Die Darstellung der Ergebnisse erfolgt nicht über AJAX-Requests, d. h. es baut sich jedesmal die komplette Seite neu auf, was bei kleinen Anfragen und schrittweisem Vorgehen zu merklichen Verzögerungen führt. Zum anderen fehlt eine History-Funktion, um ältere Anfragen zu wiederholen. Es gibt kein Syntax-Highlighting und keine Möglichkeit, Programm-, Anfrage- und Ausgabebereiche frei anzuordnen. Daher verwenden die Studierenden entweder eine lokale SWI-Prolog-Installation auf ihren Arbeitsrechnern oder nutzen die komfortablere Online-Entwicklungsumgebung SWISH[1].

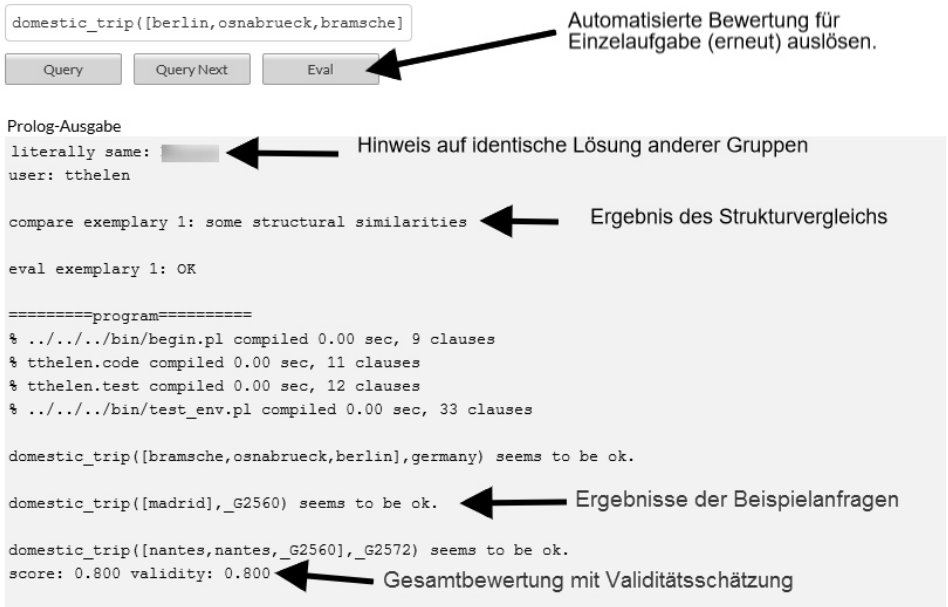

Abbildung 5.1: Bewertung einer Übungsaufgabe als Auswertungshilfe für die Korrektur

1 http://swish.swi-prolog.org/

Die Studierenden werden allerdings aufgefordert, die Funktionsfähigkeit ihrer abgegebenen Lösung in der Stud.IP-/Vips-Umgebung zu testen, da vor allem die Korrektur auf die interaktiven Möglichkeiten zurückgreift. Grundsätzlich gibt es bei der automatischen Korrektur zwei zweifelsfreie Fälle: Entweder wurde eine leere Lösung abgegeben (bzw. am vorgegebenen Programm nichts verändert), oder eine Lösung, die einer Musterlösung entspricht.

Als Auswertungshilfe für die Korrektur wendet der Grader dann zwei verschiedene Strategien an (s. Kapitel 14), die auch im Beispieloutput (s. Abbildung 5.1) zu erkennen sind:

1. Vergleich der resultierenden Variablenbelegungen: Getestet wird immer mit einer Menge vorgegebener Testanfragen, die als Prologanfragen ausgewertet werden und als Resultat eine Menge von Variablenbelegungen liefern. Diese Variablenbelegungen werden zwischen einer oder mehrerer gegebenen Musterlösungen und der abgegebenen Lösung verglichen, und zwar nicht nur eine/die erste mögliche Belegung, sondern deren Gesamtmenge. Da auch unendlich große Lösungsmengen möglich sind, ist die Maximalzahl der Vergleiche zu beschränken. Im Beispiel liefern alle Beispielanfragen korrekte Ergebnisse.

2. Vergleich der Programmstruktur: Da Prolog-Programme wie oben demonstriert sehr kompakt sind, ist es vergleichsweise gut möglich, die Struktur von Lösungen zu bewerten. Dazu werden stufenweise strukturelle Vereinfachungen vorgenommen (Entfernung von Kommentaren, Entfernung von Variablennamen etc.) und Vergleiche mit mehreren Musterlösungen, die als unterschiedlich gut gekennzeichnet werden können, vorgenommen. Im Beispiel werden nur „some similarities“ konstatiert. Die zugrunde liegende Lösung verwendet ein falsches Kriterium für den Rekursionsabbruch, was durch den Strukturvergleich trotz fehlender Testanfrage für diesen Fall erkannt wird.

Die vorgenommene Bewertung wird aus beiden Aspekten berechnet und den Korrekteuren zugänglich gemacht. Dabei wird auch graphisch unterschieden, ob eine Korrektur im oben skizzierten Sinne sicher ist (also nicht manuell kontrolliert werden muss), oder nur als Bewertungsvorschlag zu verstehen ist. In letzterem Fall nutzen die korrigierenden Tutoren typischerweise die Möglichkeit, Testanfragen direkt über Stud.IP/Vips an die abgegebene Lösung zu stellen und die Ausgaben mit der vorgeschlagenen Bewertung abzugleichen. Der Beispielscreenshot oben schlägt 4/5 Punkte als Bewertung vor (5 Punkte maximal * Bewertung 0.8 = 4).

5.3.1 Übungstyp: Formulierung von Prolog-Anfragen

Dieser Übungstyp ist vor allem in den ersten Wochen relevant, wenn Grundkonzepte von Prolog erarbeitet werden. Typischerweise wird ein Prolog-Programm und eine natürlichsprachliche Beschreibung des Anfrageziels vorgegeben, die Studierenden sollen dann eine geeignete Anfrage formulieren und sowohl Anfrage als auch Ergebnis dokumentieren.

Typisches Beispiel ist hier die Modellierung von Familienstammbäumen als deduktive Datenbank (zum Beispiel aus der griechischen Mythologie entnommen), an die dann Anfragen zu stellen sind, wie: „Wie heißen die Kinder von Zeus?“, „Welche Individuen haben gemeinsame Kinder?“.

5.3.2 Übungstyp: Vervollständigung von Prolog-Programmen

Bei diesem Aufgabentyp soll ein vorgegebenes Programm ergänzt, verändert oder korrigiert werden. Ein grundsätzlicher Vorteil dieses Aufgabentyps ist es, dass auch unsichere Studierende in der Regel eine gute Vorstellung davon haben, was zu tun ist, da keine völlig freien Entscheidungen über Aufbau und Strukturierung des Programms zu treffen sind. Ein typisches Beispiel ist die Fomulierung eines einzelnen Prädikates zu einem gegebenen Programm.

```
% city(City, Country)
city(osnabrueck, germany).
city(bramsche, germany).
city(berlin, germany).
city(paris, france).
city(nantes, france).
city(marseille, france).

% domestic_trip(Tour, Country)
%  The predicate is provable if all cities
%  are located in the same country.
domestic_trip(Tour, Country) :-
   ...
```

Ergebnis ist ein typisches rekursives Prädikat mit zwei Klauseln, von denen die erste die Rekursionsbasis darstellt und die zweite den rekursiven Fall abbildet. Hier kann es zum Beispiel zwei Sturkturvarianten geben:

```
% Variante 1: Rekursionsabbruch bei
% einelementiger Liste
domestic_trip([City],Country) :-
    city(City, Country).
domestic_trip([City|Rest], Country) :-
```

```
    city(City, Country),
    domestic_trip(Rest, Country).

% Variante 2: Rekursionsabbruch bei leerer Liste
domestic_trip([],_).
domestic_trip([City|Rest], Country) :-
    city(City, Country),
    domestic_trip(Rest, Country).
```

Die zweite Variante scheint zunächst besser zu sein, da sie auf Codeduplizierung (city-Subgoal in beiden Klauseln) verzichtet. Allerdings führt sie dazu, dass auch eine leere Route als gültiger Trip bewertet wird, was der Aufgabenstellung widerspricht. Die beiden Varianten werden daher gewichtet: Variante 1 führt zu 100% der Punkte, Variante 2 nur zu 75%. Wichtig ist auch, bei der Formulierung von Testanfragen alle intendierten Verwendungsweisen mit zu berücksichtigen.

5.3.3 Übungstyp: Vollständige Eigenimplementation von Prolog-Programmen

In einem dritten Aufgabentyp sind die Studierenden angehalten, aufgrund einer Problemstellung völlig frei zu entscheiden, wie die Implementation ausgestaltet werden soll. Hierbei sind zwei wichtige Einschränkungen zu beachten:

1. Alle Programmeingaben erfolgen in einem einzigen Textfeld. Damit ist es nicht möglich, Programme zu implementieren und einzureichen, die aus mehr als einer Datei bestehen. Angesichts der typischen Kompaktheit von Prolog-Programmen ist das für die vorliegende Veranstaltung kein grundsätzliches Problem, da sich alle behandelten Problemstellungen mit maximal 200-300 Zeilen Code erledigen lassen. Sollen allerdings größere Datenbasen verwendet werden, müssen diese auch Teil der Programme sein, was im Einzelfall etwas unübersichtlich werden kann.

2. Das Programm kann nur getestet werden, wenn eine bekannte Testanfrage verwendet werden kann, d. h. zumindest das Top-Level-Prädikat einen erwarteten Namen und eine erwartete Stelligkeit aufweist.

Mit komplexer werdenden Problemstellungen und insbesondere flexibleren Möglichkeiten der Aufteilung in Teilprobleme (und Teilprädikate) besteht kaum noch die Möglichkeit, sinnvolle Musterlösungen für den Strukturvergleich anzugeben. In diesen Fällen dient die Musterlösung im Wesentlichen nur noch dazu, die Variablenbelegungen für Testfälle zu prüfen.

5.4 Prolog im Klausurbetrieb

Die an den Prolog-Grader angeschlossene Online-Übungsumgebung wird neben dem Übungsbetrieb auch für die Durchführung von Online-Klausuren zur Vorlesung/Übung „Introduction to Artificial Intelligence and Logic Programming“ im PC-Pool verwendet. In den Klausuren spielt Prolog eine wichtige Rolle. Der überwiegende Teil der Klausur besteht allerdings aus geschlossenen Aufgabenformaten, insbesondere Einfach- oder Mehrfachauswahlfragen.

Für Prolog-Aufgaben werden grundsätzlich die gleichen Übungstypen wie im Übungsbetrieb verwendet, allerdings in veränderter Gewichtung. Umfangreiche Programmieraufgaben spielen nur eine geringe Rolle, sehr viel häufiger sind Ergänzungs- oder Korrekturaufgaben, bei denen wenige Freiheitsgrade in der Implementation bestehen bzw. keine umfangreichen Entwurfstätigkeiten notwendig sind.

Die Grader-Umgebung wird auch hier wieder in zweierlei Hinsicht verwendet:

1. Als Korrekturunterstützung. Wie im Übungsbetrieb werden als sicher falsch oder sicher richtig erkannte Lösung vollautomatisch korrigiert und für alle anderen ein Ergebnis für die manuelle Nachkorrektur vorgeschlagen. Die Korrekteure verwenden die Möglichkeit, online Prolog-Testanfragen an die abgegebenen Lösungen zu stellen, um die Vorschläge zu überprüfen und mit effizienten Arbeitsabläufen zu einer endgültigen Bewertung zu gelangen.

2. Als Arbeitsunterstützung während der Klausur. Studierende können bei allen Prolog-Aufgaben ihre Ergebnisse testen, indem sie Testanfragen stellen bzw. vorformulierte Anfragen abschicken. Das Grader-Ergebnis im Sinne einer Bewertung des Vergleichs mit Musterlösungen wird hier nicht verwendet, sondern nur die oben skizzierte Funktionalität einer eingeschränkten Online-Entwicklungsumgebung.

Im einfachsten Fall bestehen Korrekturaufgaben darin, syntaktische und andere einfache Fehler in vorgegebenen Programmen zu finden und zu korrigieren. Im folgenden Beispiel fehlt ein Komma nach einem Subgoal in der zweiten Klausel (Syntaxfehler), und es wird ein falscher Listenoperator in der letzten Zeile verwendet (, statt |, semantischer Fehler). Der Syntaxfehler ist durch korrekte Interpretation der Prolog-Fehlermeldung beim Ausprobieren des Programms zu finden, der semantische Fehler am ehesten durch Nachvollziehen des Programms oder Ausprobieren, da das Muster `[X|F]` an dieser Stelle deutlich typischer ist als `[X,F]`.

```
% Korrigieren Sie das folgende Programm.
% (Hinweis: Es sind 2 Fehler enthalten)
rotate(_,0,L,L).
rotate(left,N,[X|R],L2) :-
    N > 0,
    append(R,[X],L1)
    N1 is N - 1,
    rotate(left,N1,L1,L2).
rotate(right,N,L,L2) :-
    N > 0,
    append(F,[X],L),
    N1 is N - 1,
    rotate(right,N1,[X,F],L2).
```

Dieser Aufgabentyp wäre natürlich auch durch einen reinen Textvergleich automatisch zu korrigieren. Der Grader-Einsatz bietet aber zum einen den Nebeneffekt der Online-Entwicklungsumgebung und ist zum anderen in der Lage, auch Teillösungen oder zu komplizierte Lösungen zu bewerten. Insbesondere letztere treten häufig auf: Variablen werden umbenannt, Klauselreihenfolgen geändert, Leerzeichen verschoben etc. und diese Lösungsversuche werden nicht vollständig rückgängig gemacht.

Ein anderer häufiger Ergänzungs- bzw. Veränderungstyp ist die Transformation von Programmen, die Negationen enthalten in solche, die stattdessen Kontrollanweisungen für den Beweisprozess verwenden („Cuts“) und umgekehrt. Hier sind nur sehr geringe Eingriffe in den Programmcode an richtiger Stelle nötig. Dabei ergeben sich die gleichen Vorteile wie bei den Korrekturaufgaben.

Freie Programmieraufgaben werden in den Klausuren wie erwähnt in nur geringem Maße genutzt, sind aber vorhanden. Im Gegensatz zum Übungsbetrieb ist hier sehr viel häufiger mit unvollständigen Lösungen zu rechnen, da die Studierenden unter sehr viel größerem Zeitdruck stehen und hoffen, auch für Lösungsansätze Punkte zu bekommen. Eine typische Aufgabenstellung sieht wie folgt aus:

```
/*
Define a predicate 'powerset' which computes
the powerset of a set.
Assume that sets are represented as sorted list.
Make sure, that the resulting powerset is sorted,
too (according to the termorder in prolog, you
may use 'sort'))
Example:
    ?- powerset([a,b,c],P)
     P = [[], [a], [a, b], [a, b, c], [a, c],
         [b], [b, c], [c]]
*/
```

Die Musterlösung zu dieser Aufgabe umfasst zwei Prädikate und insgesamt neun Zeilen. Eine solche Lösung ist noch einfach genug, dass eine hinreichende Zahl

von korrekten Lösungen sich auch strukturell so weit ähneln, dass die automatische Korrektur sie erkennen kann.

Wenn Prolog-Aufgaben in einer Klausur verwendet werden, muss allerdings ein weiteres Problem bedacht werden: In das Anfrageinterface können beliebige Anfragen eingespeist werden, insbesondere auch solche aus anderen Aufgaben, die gegebenenfalls so konzipiert waren, dass sie ohne Zugriff auf einen Prolog-Interpreter zu lösen waren. Beispiele für solche geschlossenen Aufgaben sind: Ist folgender Ausdruck ein korrekter Prolog-Term? Welche der folgenden Aussagen treffen auf das gegebene Programm zu? Welches Ergebnis liefert folgende Anfrage?

In dem vorliegenden Kurs gab es auch solche Aufgaben und es konnte nicht ausgeschlossen werden, dass zur Beantwortung ein Prolog-Interpreter verwendet wurde. Der hier verfolgte Ansatz zur Verhinderung solcher Tricks war ein zeitlicher: Die Anzahl der Aufgaben ist so hoch gewählt, dass sie sämtlich unter Zeitdruck zu bearbeiten sind und das Kopieren eines Ausdrucks, der Aufruf einer Prolog-Aufgabe, das Einfügen und Bewerten des Ausdrucks und die Rückkehr zur ursprünglichen Aufgabe zu viel Zeit in Anspruch nehmen.

5.5 Erfahrungen, Fazit und Ausblick

Die hier skizzierte Verwendungsweise eines Prolog-Graders, der auch als einfache webbasierte Entwicklungsumgebung benutzt werden kann, hat sich sowohl für den Übungsbetrieb als auch die Verwendung in Klausuren in den letzten Jahren bewährt. Der Hauptnutzen des Grader-Einsatzes ist die gesteigerte Effizienz bei der Korrektur, die es Tutoren und Dozenten ermöglicht, sich auf unklare Fälle zu konzentrieren und dort verbesserte Rückmeldungen zu geben.

Die Teilnehmerinnen und Teilnehmer der Veranstaltung „Introduction to Artificial Intelligence and Logic Programming“ profitieren in mehrfacher Hinsicht vom Einsatz der Grader-Technologie. Zuächst stellt die Veranstaltung sie grundsätzlich vor hohe Herausforderungen: Viele der Studierenden haben kaum programmiertechnische Vorkenntnisse und interessieren sich gegebenenfalls eher für die neurowissenschaftlichen und psychologischen Aspekte des Cognitive-Science-Studiums. Die Veranstaltung findet im zweiten Fachsemester neben weiteren arbeitsintensiven Einführungsveranstaltungen statt und verlangt hohen Arbeitseinsatz. In diesem Umfeld ist das Ziel, auch praktische Kenntnisse und Fertigkeiten zu vermitteln, schwierig zu erreichen. Die konsequente Einbeziehung programmierpraktischer Aufgaben in Übungen und Klausuren verstärkt den Praxisbezug aller-

dings grundsätzlich und wird auf Auswertungsseite nur durch die automatisierte Bewertung handhabbar.

Literatur für dieses Kapitel

[BKW05] Christoph Beierle, Marija Kulaš und Manfred Widera. „A pragmatic approach to pre-testing Prolog programs“. In: *Applications of Declarative Programming and Knowledge Management*. Springer, 2005, S. 294–308.

[Bra+06] Erik Braun u. a. „Interactive Problem Solving in Prolog“. In: *arXiv preprint cs/0611014* (2006).

[Bra01] Ivan Bratko. *Prolog programming for artificial intelligence*. Pearson education, 2001.

[Bön+99] Carsten Bönnen u. a. „PLOT: Prolog-Online-Tutor“. In: *Osnabrück: Institut für Semantische Informationsverarbeitung* (1999).

[CR96] Alain Colmerauer und Philippe Roussel. „The birth of Prolog“. In: *History of programming languages—II*. ACM. 1996, S. 331–367.

[Pey+00] Christoph Peylo u. a. „A Web-based intelligent educational system for PROLOG“. In: *Proceedings of the International Workshop on Adaptive and Intelligent Web-Based Education Systems held in conjunction with ITS 2000 Montreal, Canada.* 2000, S. 85–96.

[Pey02] Christoph Peylo. *Wissen und Wissensvermittlung im Kontext von internetbasierten intelligenten Lehr- und Lernumgebungen*. Akademische Verlagsgesellschaft, 2002.

[Sch97] Rolf Schulmeister. *Grundlagen hypermedialer Lernsysteme*. Oldenbourg München, 1997.