The Pragmatic Programmers

Sieben Wochen, sieben Sprachen

Verstehen Sie die modernen Sprachkonzepte

Bruce A. Tate

Übersetzt von Peter Klicman

Deutsche Übersetzung von

O'REILLY®

Bruce A. Tate, Sieben Wochen, sieben Sprachen, O´Reilly, ISBN 97838972132279783897213227

Inhaltsverzeichnis

Widmung 1

Danksagung 3

Vorwort 7

1 Einführung 11

1.1 Wahnsinn mit Methode 11
1.2 Die Sprachen 13
1.3 Kaufen Sie dieses Buch 15
1.4 Kaufen Sie dieses Buch nicht 17
1.5 Ein letzter Punkt 20

2 Ruby 21

2.1 Ein wenig Geschichte 22
2.2 Tag 1: Ein Kindermädchen finden 24
2.3 Tag 2: Vom Himmel herab 32
2.4 Tag 3: Tiefgreifende Veränderung 45
2.5 Ruby zusammengefasst 53

3 Io 59

3.1 Einführung in Io 59
3.2 Tag 1: Blaumachen und rumhängen 60
3.3 Tag 2: Der Würstchenkönig 74
3.4 Tag 3: Die Parade und andere sonderbare Orte 83
3.5 Io zusammengefasst 92

4 Prolog 97

4.1 Über Prolog 98
4.2 Tag 1: Ein ausgezeichneter Fahrer 99

4.3 Tag 2: Fünfzehn Minuten für Wapner . . . 112
4.4 Tag 3: Die Bank sprengen . . . 124
4.5 Prolog zusammengefasst . . . 137

5 Scala **141**

5.1 Über Scala . . . 141
5.2 Tag 1: Die Burg auf der Anhöhe . . . 146
5.3 Tag 2: Gesträuch beschneiden und andere neue Tricks . . . 161
5.4 Tag 3: Sich durch die Fusseln schneiden . . . 176
5.5 Scala zusammengefasst . . . 186

6 Erlang **191**

6.1 Einführung in Erlang . . . 191
6.2 Tag 1: Menschlich erscheinen . . . 196
6.3 Tag 2: Die Form ändern . . . 207
6.4 Tag 3: Die rote Pille . . . 219
6.5 Erlang zusammengefasst . . . 232

7 Clojure **237**

7.1 Einführung in Clojure . . . 238
7.2 Tag 1: Luke trainieren . . . 239
7.3 Tag 2: Yoda und die Macht . . . 258
7.4 Tag 3: Ein Auge für Böses . . . 272
7.5 Clojure zusammengefasst . . . 282

8 Haskell **287**

8.1 Einführung in Haskell . . . 287
8.2 Tag 1: Logisch . . . 288
8.3 Tag 2: Spocks große Stärke . . . 305
8.4 Tag 3: Gedankenverschmelzung . . . 316
8.5 Haskell zusammengefasst . . . 332

9 Zusammenfassung **337**

9.1 Programmiermodelle . . . 337
9.2 Nebenläufigkeit . . . 341
9.3 Programmierkonstrukte . . . 343
9.4 Ihre Sprache finden . . . 346

A Bibliografie **347**

Index **349**

Bruce A. Tate, Sieben Wochen, sieben Sprachen, O´Reilly, ISBN 97838972132279783897213227

Sally Dibbs, Dibbs, Sally. 461-0192.

► Raymond

Kapitel 4

Prolog

Ah, Prolog. Manchmal beeindruckend clever und manchmal ebenso frustrierend. Verblüffende Antworten erhalten Sie nur, wenn Sie wissen, wie die Frage zu stellen ist. Denken Sie an *Rain Man*.[1] Ich erinnere mich, wie die Hauptfigur Raymond ohne nachzudenken Sally Dibbs' Telefonnummer herunterrasselte, nachdem er in der Nacht das Telefonbuch gelesen hatte. Bei Raymond und Prolog frage ich mich zu gleichen Teilen: „Wie konnte er das wissen?" und „Wie konnte er das nicht wissen?" Er ist eine unglaubliche Wissensquelle, wenn man es denn hinbekommt, die Fragen richtig zu stellen.

Prolog stellt eine deutliche Abweichung von den anderen Sprachen dar, die wir bisher betrachtet haben. Sowohl Io als auch Ruby werden als *imperative Sprachen* bezeichnet. Imperative Sprachen sind wie Rezepte: Sie sagen dem Computer ganz genau, was er tun soll. Imperative Sprachen höherer Ordnung haben vielleicht eine größere Hebelwirkung, doch letztendlich stellen Sie eine Einkaufsliste aller Zutaten zusammen und beschreiben Schritt für Schritt, wie man einen Kuchen backt.

Ich habe einige Wochen mit Prolog herumgespielt, bevor ich mich an dieses Kapitel heranwagte. In der Einstiegsphase benutzte ich verschiedene Tutorials, darunter eines von J. R. Fisher[2] (um einige Beispiele durchzuarbeiten) und ein weiteres von A. Aaby[3] (Hilfe im Bezug auf Struktur und Terminologie), und machte sehr viele Experimente.

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

1 *Rain Man*. DVD. By Barry Levinson. 1988; Los Angeles, CA: MGM, 2000.
2 http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/contents.html
3 http://www.lix.polytechnique.fr/~liberti/public/computing/prog/prolog/prolog-tutorial.html

Prolog ist eine *deklarative Sprache*. Sie geben ein paar Fakten und Schlussfolgerungen und überlassen der Sprache das „Denken“. Es ist so, wie zu einem guten Bäcker zu gehen: Sie beschreiben die Eigenschaften der Torte, die Sie mögen, und überlassen es dem Bäcker, (basierend auf den von Ihnen aufgestellten Regeln) die Zutaten herauszusuchen und die Torte zu backen. Bei Prolog müssen Sie nicht wissen, *wie*. Der Computer übernimmt das für Sie.

Im Internet finden Sie problemlos Beispiele für Programme, die ein Sudoku mit weniger als 20 Zeilen Code lösen, Rubiks Würfel knacken und berühmte Rätsel wie die Türme von Hanoi (etwa ein Dutzend Codezeilen) lösen. Prolog war eine der ersten erfolgreichen logischen Programmiersprachen. Sie treffen Aussagen mit reiner Logik, und Prolog ermittelt, ob sie wahr sind. Ihre Aussagen können Lücken aufweisen, die Prolog so zu ergänzen versucht, dass Ihre unvollständigen Fakten wahr werden.

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

4.1 Über Prolog

Im Jahr 1972 von Alain Colmerauer und Phillipe Roussel entwickelt, war Prolog eine logische Programmiersprache, die sich bei der Verarbeitung natürlicher Sprachen großer Beliebtheit erfreute. Heutzutage bildet die altehrwürdige Sprache die programmtechnische Grundlage zur Lösung unterschiedlichster Probleme von der Disposition bis zu Expertensystemen. Sie können diese regelbasierte Sprache verwenden, um Logik auszudrücken und Fragen zu stellen. Wie SQL arbeitet Prolog mit Datenbanken, doch die Daten bestehen aus logischen Regeln und Beziehungen. Wie SQL besteht Prolog aus zwei Teilen: einem, der die Daten ausdrückt, und einem, der diese Daten abfragt. Bei Prolog haben die Daten die Form logischer Regeln. Hier die Grundbausteine:

- *Fakten.* Ein Faktum ist eine grundlegende Aussage über eine Welt. (Babe ist ein Schwein; Schweine mögen Schlamm.)
- *Regeln.* Eine Regel ist eine Folgerung aus den Fakten einer Welt. (Ein Tier mag Schlamm, wenn es ein Schwein ist.)
- *Query.* Eine Query (Abfrage) ist eine Frage zu einer Welt. (Mag Babe Schlamm?)

Fakten und Regeln wandern in eine *Knowledge Base*, also eine Wissensdatenbank. Ein Prolog-Compiler übersetzt die Wissensdatenbank in eine Form, die sich für Queries eignet. Wenn wir im Folgenden die Beispiele durchgehen, werden Sie Prolog benutzen, um Ihr Wissen (für

Bruce A. Tate, Sieben Wochen, sieben Sprachen, O´Reilly, ISBN 97838972132279783897213227

die Wissensdatenbank) auszudrücken. Dann werden Sie die Daten direkt abrufen. Außerdem werden Sie Prolog verwenden, um Regeln zu verknüpfen und sich Dinge sagen lassen, die Sie noch nicht wussten.

Doch genug der Hintergrundinformationen. Legen wir los.

4.2 Tag 1: Ein ausgezeichneter Fahrer

In *Rain Man* erzählt Raymond seinem Bruder, dass er ein ausgezeichneter Fahrer sei, was in seinem Fall bedeutet, dass er die Sache bei 10 km/h auf einem Parkplatz ganz ordentlich erledigt. Er benutzt dabei alle wesentlichen Elemente (das Lenkrad, die Bremsen, und den Gashebel), wenn auch in einem etwas beschränkten Umfeld. Genau das ist heute unser Ziel. Wir werden Prolog nutzen, um einige Fakten auszudrücken, ein paar Regeln aufstellen und einige einfache Queries durchführen. Wie Io ist Prolog syntaktisch eine extrem einfache Sprache, deren Syntaxregeln man sehr schnell erlernen kann. Der Spaß beginnt erst so richtig, wenn Sie die Konzepte auf interessante Art und Weise verknüpfen. Wenn das Ihr erstes Zusammentreffen mit Prolog ist, garantiere ich Ihnen, dass Sie entweder Ihr Denken ändern oder kläglich scheitern werden. Den genaueren Aufbau sparen wir uns für einen anderen Tag auf.

Eins nach dem anderen. Sie benötigen eine funktionierende Installation. Ich verwende für dieses Buch GNU-Prolog in der Version 1.3.1. Doch Vorsicht: Die Dialekte sind verschieden. Ich tue mein Bestes, um auf der sicheren Seite zu bleiben, doch wenn Sie sich für eine andere Prolog-Version entscheiden, müssen Sie Ihre Hausaufgaben machen und herausfinden, worin sich der Dialekt unterscheidet, den Sie verwenden. Unabhängig von der verwendeten Version werden Sie hier erfahren, wie man die Sprache grundsätzlich benutzt.

Grundlegende Fakten

Bei einigen Sprachen liegt die Groß- und Kleinschreibung völlig im Ermessen des Programmierers. Bei Prolog ist die Schreibweise des ersten Buchstaben von Bedeutung: Beginnt ein Wort mit einem Kleinbuchstaben, ist es ein *Atom*, also ein fester Wert (wie ein Ruby-Symbol). Beginnt es mit einem Großbuchstaben oder einem Unterstrich, handelt es sich um eine *Variable*. Die Werte von Variablen können sich ändern, die von Atomen nicht. Lassen Sie uns eine einfache Wissensdatenbank mit ein paar Fakten aufbauen. Geben Sie Folgendes in einem Editor ein:

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

prolog/friends.pl

```
likes(wallace, cheese).
likes(grommit, cheese).
likes(wendolene, sheep).

friend(X, Y) :- \+(X = Y), likes(X, Z), likes(Y, Z).
```

Die obige Datei ist eine Wissensdatenbank mit Fakten und Regeln. Die ersten drei Anweisungen sind Fakten, und die letzte Anweisung ist eine Regel. Fakten sind direkte Beobachtungen aus unserer Welt. Wir wollen zuerst nur die ersten drei Zeilen betrachten. Diese drei Zeilen sind Fakten. `wallace`, `grommit` und `wendolene` sind Atome. Sie können Sie wie folgt lesen: `wallace` mag `cheese`, `grommit` mag `cheese` und `wendolene` mag `sheep`. Lassen wir diese Fakten in Aktion treten.

Starten Sie den Prolog-Interpreter. Wenn Sie GNU-Prolog benutzen, geben Sie den Befehl `gprolog` ein. Um die Datei zu laden, geben Sie dann Folgendes ein:

```
| ?- ['friends.pl'].
compiling /Users/batate/prag/Book/code/prolog/friends.pl for byte code...
/Users/batate/prag/Book/code/prolog/friends.pl compiled, 4 lines read -
997 bytes written, 11 ms

yes

| ?-
```

Wenn Prolog nicht auf ein Zwischenergebnis wartet, antwortet es mit `yes` oder `no`. In unserem Fall wurde die Datei erfolgreich geladen, weshalb es mit `yes` antwortet. Wir können nun einige Fragen stellen. Die einfachsten Fragen sind Ja/Nein-Fragen zu Fakten. Stellen Sie einige Fragen:

```
|?- likes(wallace, sheep).

no
| ?- likes(grommit, cheese).

yes
```

Diese Fragen sind intuitiv verständlich. Mag Wallace sheep? Nein. Mag Grommit cheese Ja. Sie sind nicht besonders interessant: Prolog plappert einfach die Fakten nach. Es wird etwas spannender, wenn Sie damit beginnen, etwas Logik einfließen zu lassen. Werfen wir einen Blick auf Schlussfolgerungen.

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

Bruce A. Tate, Sieben Wochen, sieben Sprachen, O´Reilly, ISBN 97838972132279783897213227

Grundlegende Folgerungen und Variablen

Probieren wir die `friend`-Regel aus:

```
| ?- friend(wallace, wallace).

no
```

Prolog arbeitet sich also durch die von uns aufgestellten Regeln und beantwortet Fragen mit `yes` oder `no`. Es steckt mehr dahinter, als man denken könnte. Sehen wir uns die `friend`-Regel noch einmal an:

Damit X ein friend von Y sein kann, darf X nicht gleich Y sein. Sehen wir uns den ersten Teil rechts von `:-` an, den man als *Teilziel (subgoal*) bezeichnet. `\+` steht für die logische Negation; `\+(X = Y)` bedeutet also `X` ist nicht gleich `Y`.

Probieren Sie weitere Queries aus:

```
| ?- friend(grommit, wallace).

Yes
| ?- friend(wallace, grommit).

yes
```

Auf Deutsch formuliert ist X ein Freund von Y, wenn wir beweisen können, dass X irgendein Z mag und Y das gleiche Z mag. Sowohl `wallace` als auch `grommit` mögen `cheese`, weshalb die Queries wahr sind.

Tauchen wir in den Code ein. Bei diesen Queries ist X ungleich Y, wodurch das erste Teilziel nachgewiesen ist. Die Query verwendet das zweite und das dritte Teilziel `likes(X, Z)` und `likes(Y, Z)`. `grommit` und `wallace` mögen `cheese`, wodurch das zweite und dritte Teilziel nachgewiesen wären. Probieren wir eine weitere Query aus:

```
| ?- friend(wendolene, grommit).

no
```

In diesem Fall muss Prolog mehrere mögliche Werte für X, Y und Z durchprobieren:

- `wendolene`, `grommit` und `cheese`
- `wendolene`, `grommit` und `sheep`

Keine Kombination erfüllt beide Ziele, das `wendolene` `Z` mag und `grommit` `Z` mag. Es existiert keine passende Kombination, weshalb die Logik-Engine `no` zurückgibt, d. h. sie sind keine Freunde.

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

Lassen Sie uns die Terminologie ein wenig formalisieren. Das hier …

```
friend(X, Y) :- \+(X = Y), likes(X, Z), likes(Y, Z).
```

… ist eine Prolog-Regel mit den drei Variablen `X`, `Y` und `Z`. Wir nennen diese Regel `friend/2`, als Abkürzung für `friend` mit zwei Parametern. Die Regel hat drei Teilziele, getrennt durch Kommata. Alle müssen erfüllt sein, damit die Regel wahr ist. Unsere Regel besagt also, dass `X` ein Freund von `Y` ist, wenn `X` und `Y` nicht gleich sind und `X` und `Y` das gleiche `Z` mögen.

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

Die Lücken füllen

Wir haben Prolog benutzt, um einige Ja/Nein-Fragen zu beantworten, doch wir können mehr als das. In diesem Abschnitt werden wir die Logik-Engine benutzen, um alle für eine Abfrage möglichen Treffer zu finden. Zu diesem Zweck werden Sie in Ihrer Query eine *Variable* verwenden.

Betrachten wir die folgende Wissensdatenbank:

prolog/food.pl

```
food_type(edamer, cheese).
food_type(tuc, cracker).
food_type(spam, meat).
food_type(sausage, meat).
food_type(jolt, soda).
food_type(yes, dessert).

flavor(sweet, dessert).
flavor(savory, meat).
flavor(savory, cheese).
flavor(sweet, soda).

food_flavor(X, Y) :- food_type(X, Z), flavor(Y, Z).
```

Wir hab einige Fakten. So etwas wie `food_type(edamer, cheese)` gibt an, dass es sich um Nahrung eines bestimmten Typs handelt. Andere wie `flavor(sweet, dessert)` beschreiben den charakteristischen Geschmack eines Nahrungstyps. Schließlich gibt es noch eine Regel namens `food_flavor`, die den Geschmack eines Nahrungsmittels schlussfolgert. Ein Nahrungsmittel `X` hat einen `food_flavor` `Y`, wenn das Nahrungsmittel ein `food_type` `Z` ist und `Z` gleichzeitig den charakteristischen Geschmack aufweist. Wir wollen das kompilieren …

```
| ?- ['code/prolog/food.pl'].
compiling /Users/batate/prag/Book/code/prolog/food.pl for byte code...
/Users/batate/prag/Book/code/prolog/food.pl compiled,
```

Bruce A. Tate, Sieben Wochen, sieben Sprachen, O´Reilly, ISBN 97838972132279783897213227

```
12 lines read - 1557 bytes written, 15 ms

(1 ms) yes
```

… und ein paar Fragen stellen:

```
| ?- food_type(What, meat).

What = spam ? ;

What = sausage ? ;

no
```

Das ist interessant. Wir haben Prolog gefragt, „welcher Wert für was erfüllt die Query `food_type(What, meat)`". Prolog hat einen gefunden: `spam`. Als wir dann `;` eingegeben haben, um von Prolog eine weitere Antwort zu erhalten, erhielten wir `sausage`. Diese Werte zu finden, war leicht, da die Abfragen auf grundlegenden Fakten basieren. Wir wollten dann eine weitere Antwort, und Prolog antwortete mit `no`. Dieses Verhalten kann leicht variieren. Wenn Prolog erkennt, dass es keine weiteren Möglichkeiten gibt, wird der Bequemlichkeit halber `yes` ausgegeben. Kann Prolog ohne weitere Berechnungen nicht sofort ermitteln, ob es weitere Alternativen gibt, fragt es nach der nächsten und gibt `no` aus. Dieses Feature ist tatsächlich sehr bequem. Kann Prolog Ihnen früher eine Information geben, dann macht es das auch. Probieren wir weitere Queries aus:

```
| ?- food_flavor(sausage, sweet).

no

| ?- flavor(sweet, What).

What = dessert ? ;

What = soda

yes
```

Nein, eine `Wurst` ist nicht süß. Welche Nahrungsmittel sind süß? `dessert` und `soda`. Das sind Fakten. Doch Sie können Prolog auch Zusammenhänge herstellen lassen:

```
| ?- food_flavor(What, savory).

What = edamer ? ;

What = spam ? ;

What = sausage ? ;

no
```

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

Denken Sie daran, dass `food_flavor(X, Y)` eine Regel ist, kein Fakt. Wir fordern von Prolog alle Werte an, die die Anfrage „Welche Nahrungsmittel haben einen herzhaften Geschmack?“ erfüllen. Prolog muss die einfachen Fakten über Nahrungsmittel, Typen und Geschmack verknüpfen, um zu einer Schlussfolgerung zu kommen. Die Logik-Engine muss die möglichen Kombinationen durchgehen, für die alle Ziele zutreffen.

Karten einfärben

Als etwas spektakuläreres Prolog-Beispiel wollen wir die gleiche Idee verwenden, um eine Landkarte einzufärben, genauer gesagt eine Karte des Südostens der USA. Wir betrachten die Staaten aus Abbbildung 4.1. Wir wollen nicht, das sich zwei Staaten mit derselben Farbe berühren.

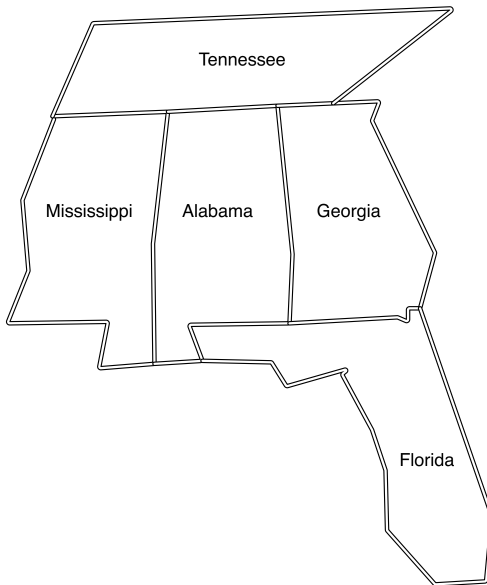

Abbildung 4.1: Karte der südöstlichen USA

Wir halten die folgenden einfachen Fakten fest:

prolog/map.pl

```
different(red, green). different(red, blue).
different(green, red). different(green, blue).
different(blue, red). different(blue, green).

coloring(Alabama, Mississippi, Georgia, Tennessee, Florida) :-
  different(Mississippi, Tennessee),
  different(Mississippi, Alabama),
```

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

Bruce A. Tate, Sieben Wochen, sieben Sprachen, O´Reilly, ISBN 97838972132279783897213227

```
  different(Alabama, Tennessee),
  different(Alabama, Mississippi),
  different(Alabama, Georgia),
  different(Alabama, Florida),
  different(Georgia, Florida),
  different(Georgia, Tennessee).
```

Wir verwenden drei Farben. Wir teilen Prolog die Gruppen verschiedener Farben mit, die beim Einfärben der Karte verwendet werden sollen. Als Nächstes folgt eine einzelne Regel. Diese teilt Prolog mit, welche Staaten Nachbarn sind, und fertig. Probieren Sie es aus:

```
| ?- coloring(Alabama, Mississippi, Georgia, Tennessee, Florida).

Alabama = blue
Florida = green
Georgia = red
Mississippi = red
Tennessee = green ?
```

Offensichtlich gibt es eine Möglichkeit, diese fünf Staaten mit drei Farben einzufärben. Sie erhalten die weiteren möglichen Kombinationen, indem Sie a eingeben. Mit einem Dutzend Zeilen sind wir durch. Die Logik ist völlig simpel, ein Kind könnte sie herausfinden. An dieser Stelle müssen Sie sich selbst fragen ...

Wo ist das Programm?

Wir haben keinen Algorithmus! Versuchen Sie, das Problem mit einer prozeduralen Sprache ihrer Wahl zu lösen. Ist ihre Lösung leicht zu verstehen? Überlegen Sie, was Sie tun müssen, um komplexe logische Probleme wie diese mit Ruby oder Io zu lösen. Eine mögliche Lösung könnte wie folgt aussehen:

1. Erfassen und organisieren Sie Ihre Logik.
2. Drücken Sie die Logik in einem Programm aus.
3. Finden Sie alle möglichen Lösungen.
4. Lassen Sie alle möglichen Lösungen durch Ihr Programm laufen.

Und Sie müssten das Programm immer und immer wieder schreiben. Prolog erlaubt Ihnen, die Logik über Fakten und Schlussfolgerungen auszudrücken und Fragen zu stellen. Sie sind bei dieser Sprache nicht dafür verantwortlich, ein Schritt-für-Schritt-Rezept aufzubauen. Bei Prolog geht es nicht darum, Algorithmen für logische Probleme zu entwickeln, sondern darum, eine Welt so zu beschreiben, wie Sie ist, und

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

logische Probleme zu präsentieren, die der Computer zu lösen versuchen kann.

Lassen Sie den Computer die Arbeit erledigen!

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

Unifizierung, Teil 1

An diesem Punkt müssen wir einen Schritt zurücktreten und etwas auf die Theorie eingehen. Beschäftigen wir uns etwas mit der *Unifizierung*. Einige Sprachen verwenden Variablenzuweisungen. Bei Java oder Ruby bedeutet `x=10` beispielsweise die Zuweisung von `10` an die Variable `x`. Die Unifizierung zweier Strukturen versucht, diese beiden Strukturen identisch zu machen. Nehmen wir die folgende Wissensdatenbank:

prolog/ohmy.pl

```
cat(lion).
cat(tiger).

dorothy(X, Y, Z) :- X = lion, Y = tiger, Z = bear.
twin_cats(X, Y) :- cat(X), cat(Y).
```

In diesem Beispiel steht = für das Unifizieren, oder „mach beide Seiten gleich“. Wir haben zwei Fakten: `lions` und `tigers` sind `cats`. Es gibt auch zwei einfache Regeln. Bei `dorothy/3` sind `X`, `Y` und `Z` gleich `lion`, `tiger` und `bear`. Bei `twin_cats/2` ist `X` eine `cat` und `Y` auch. Wir können diese Wissensbasis nutzen, um etwas mehr Licht in die Unifizierung zu bringen.

Zuerst wollen wir die erste Regel nutzen. Ich kompiliere unser Wissen und führe dann eine einfache Abfrage ohne Parameter durch:

```
| ?- dorothy(lion, tiger, bear).

yes
```

Denken Sie daran, was die Unifizierung bedeutet: „Finde die Werte, bei denen beide Seiten gleich sind“. Auf der rechten Seite bindet Prolog `X`, `Y` und `Z` an `lion`, `tiger` und `bear`. Diese passen zu den entsprechenden Werten auf der linken Seite, die Unifizierung war also erfolgreich. Prolog meldet `yes`. Dieser Fall ist einfach, doch wir können ihn ein wenig aufpeppen. Die Unifizierung funktioniert auf beiden Seiten der Implikation. Probieren Sie Folgendes:

```
| ?- dorothy(One, Two, Three).

One = lion
Three = bear
```

Bruce A. Tate, Sieben Wochen, sieben Sprachen, O´Reilly, ISBN 97838972132279783897213227

```
Two = tiger

yes
```

Dieses Beispiel verwendet eine zusätzliche Dereferenzierungsschicht. In den Zielen unifiziert Prolog X, Y und Z zu lion, tiger und bear. Auf der linken Seite unifiziert Prolog X, Y und Z zu One, Two und Three und gibt dann das Ergebnis zurück.

Sehen wir uns nun die letzte Regel an: twin_cats/2. Die Regel besagt, dass twin_cats(X, Y) wahr ist, wenn Sie beweisen können, dass X und Y beides Katzen sind. Probieren Sie es aus:

```
| ?- twin_cats(One, Two).

One = lion
Two = lion ?
```

Prolog gibt das erste Beispiel zurück. lion und lion sind beides Katzen. Sehen wir uns an, wie die Sprache darauf kommt:

1. Wir haben die Query twin_cats(One, Two) angestoßen. Prolog bindet One an X und Two an Y. Um das lösen zu können, muss sich Prolog durch die Ziele arbeiten.

2. Das erste Ziel ist cat(X).

3. Wir besitzen zwei passende Fakten: cat(lion) und cat(tiger). Prolog probiert das erste Faktum aus, bindet X an lion und macht dann mit dem nächsten Ziel weiter.

4. Prolog bindet nun Y an cat(Y). Prolog kann dieses Ziel auf die gleiche Weise lösen wie das erste und wählt lion.

5. Wir haben beide Ziele zufriedengestellt, die Regel ist also erfolgreich. Prolog gibt die erfolgreichen Werte für One und Two aus und meldet yes.

Wir besitzen nun die erste Lösung, für die unsere Regeln zutreffen. Manchmal reicht eine Lösung aus. Manchmal braucht man mehr als eine. Wir können uns nun eine Lösung nach der anderen ansehen, indem wir ; eingeben, oder wir sehen uns den ganzen Rest an, indem wir a drücken.

```
Two = lion ? a

One = lion
Two = tiger
```

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

```
One = tiger
Two = lion

One = tiger
Two = tiger

(1 ms) yes
```

Beachten Sie, dass Prolog die Liste aller Kombinationen von X und Y durcharbeitet und dabei die in den Zielen und den entsprechenden Fakten angegebenen Informationen nutzt. Wie Sie später sehen werden, ermöglicht die Unifizierung ein anspruchsvolles Matching, das auf der Struktur der Daten basiert. Das ist genug für den ersten Tag. Am zweiten Tag werden wir uns schwierigeren Dingen zuwenden.

Prolog in der Praxis

Es war ein bisschen befremdlich, ein solches „Programm" zu sehen. Bei Prolog gibt es kaum ein detailliertes Schritt-für-Schritt-Rezept, sondern nur eine Beschreibung des Kuchens, der nach dem Backen aus dem Ofen kommt. Beim Lernen von Prolog half es mir enorm, dass ich jemanden interviewen durfte, der die Sprache in der Praxis eingesetzt hat: Ich habe mit Brian Tarbox gesprochen, der diese Logiksprache verwendet hat, um für ein Forschungsprojekt Arbeitspläne für die Arbeit mit Delphinen zu entwickeln.

Ein Interview mit Brian Tarbox, Delfinforscher

Bruce: Was können Sie uns über Ihre Erfahrung beim Erlernen von Prolog erzählen?

Brian: Ich lernte Prolog in den späten 1980ern, als ich an der University of Hawaii in Manoa studierte. Ich forschte am Kewalo Basin Marine Mammal Laboratory über die kognitiven Fähigkeiten Großer Tümmler. Zu der Zeit bemerkte ich, dass die Leute im Labor größtenteils Theorien darüber diskutierten, wie Delfine denken. Wir arbeiteten hauptsächlich mit einer Delfinin namens Akeakamai, oder kurz Ake. Viele unserer Debatten begannen mit „Hmm, Ake sieht das wahrscheinlich so und so …"

Ich wollte in meiner Masterarbeit ein ausführbares Modell entwickeln, das unsere Ansichten von Akes Vorstellung von der Welt widerspiegeln sollte (oder zumindest den kleinen Teil davon, an dem wir forschten). Mit unserem ausführbaren Modell Akes tatsächliches Verhalten vorhersagen zu können, sollte unsere Theorien zu ihrem Denken verifizieren.

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

Bruce A. Tate, Sieben Wochen, sieben Sprachen, O´Reilly, ISBN 97838972132279783897213227

Prolog ist eine wundervolle Sprache, aber die Ergebnisse können ziemlich schräg sein. Ich erinnere mich an meine ersten Experimente. Ich schrieb so etwas wie x = x + 1 und Prolog antwortete „no“. Sprachen sagen nicht einfach „no“. Sie können die falsche Antwort zurückliefern oder die Kompilierung kann fehlschlagen, aber ich kannte noch keine Sprache, die mir Widerworte gab. Also rief ich den Prolog-Support an und sagte, dass die Sprache „no“ antwortete, wenn ich versuchte, den Wert einer Variablen zu ändern. Ich wurde gefragt: „Warum sollten Sie den Wert einer Variablen ändern wollen?“ Na ja, welche Sprache lässt einen nicht den Wert einer Variablen ändern? Sobald man Prolog begriffen hat, versteht man, dass Variablen entweder bestimmte Werte haben oder nicht gebunden sind, aber zu dem Zeitpunkt war das verwirrend.

Bruce: Wie haben Sie Prolog genutzt?

Brian: Ich habe zwei Hauptsysteme entwickelt: einen Delfinsimulator und einen Laborarbeitsplaner. Das Labor sollte jeden Tag vier Experimente mit den vier Delfinen durchführen. Sie müssen wissen, dass Forschungsdelfine eine unglaublich knappe Ressource sind. Jeder Delphin arbeitete an unterschiedlichen Experimenten, und jedes Experiment verlangte unterschiedliches Personal. Einige Rollen, wie etwa die des Delfintrainers, konnten nur von wenigen Leuten übernommen werden. Andere Aufgaben, wie die Datenaufzeichnung, konnten von verschiedenen Leuten erledigt werden, verlangten aber trotzdem ein gewisses Training. Für die meisten Experimente waren sechs bis zu einem Dutzend Leute notwendig. Wir hatten Doktoranden, Studenten und Earthwatch-Freiwillige. Jede Person hatte ihren eigenen Zeitplan und ihre ganz eigenen Fähigkeiten. Einen Arbeitsplan zu finden, der alle auslastet und sicherstellt, dass alle Arbeiten erledigt werden, wurde für einen vom Personal zur Vollzeitbeschäftigung.

Ich wollte versuchen, einen Prolog-basierten Arbeitsplaner zu entwickeln. Es stellte sich heraus, dass die Sprache für dieses Problem wie gemacht zu sein schien. Ich entwickelte eine Reihe von Fakten, die die Fähigkeiten und Zeitpläne der einzelnen Personen und die Anforderungen aller Experimente beschrieben. Ich konnte Prolog dann grundsätzlich sagen: „Mach es so und so!“ Für jede in einem Experiment aufgeführte Aufgabe sollte die Sprache eine verfügbare Person mit den geforderten Kenntnissen finden und an diese Aufgabe binden. Das würde so lange weitergehen, bis entweder alle Anforderungen des Experiments erfüllt waren oder eine Lösung unmöglich war. Konnte Prolog keine gültige Bindung finden, löste es frühere Bindungen auf und versuchte es mit einer anderen Kombination. Letztendlich würde es entwe-

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

der einen gültigen Arbeitsplan finden oder das Experiment als zu stark gebunden deklarieren.

Bruce: Gibt es Beispiele für Fakten, Regeln oder Aussagen im Bezug auf Delfine, die für unsere Leser von Interesse sein könnten?

Brian: Ich erinnere mich an eine bestimmte Situation, in der der simulierte Delfin uns dabei half, Akes tatsächliches Verhalten zu verstehen. Ake reagierte auf eine gestenreiche Zeichensprache mit „Sätzen" wie „spring durch den Reifen" oder „berühr den rechten Ball mit der Schwanzflosse". Wir gaben ihr Anweisungen und sie reagierte.

Ein Teil meiner Forschung bestand darin, ihr neue Worte wie „nicht" beizubringen. In diesem Kontext bedeutete „berühr den Ball nicht", dass sie alles außer dem Ball berühren durfte. Dieses Problem war für Ake schwer zu lösen, doch eine Zeit lang machte die Forschung gute Fortschritte. An einem Punkt ließ sie sich aber einfach unter Wasser sinken, sobald wir ihr die Anweisung gaben. Wir verstanden das nicht. Das ist eine sehr frustrierende Situation, weil Sie einen Delfin nicht fragen können, warum er etwas macht. Also präsentierten wir die Trainingsaufgabe dem simulierten Delfin und erhielten ein sehr interessantes Ergebnis. Zwar sind Delfine sehr clever, doch versuchen sie generell, die einfachste Antwort auf ein Problem zu finden. Wir hatten unserem simulierten Delfin einige Heuristiken mitgegeben. Es stellte sich heraus, dass Akes Zeichensprache ein „Wort" für eines der Fenster im Tank enthielt. Die meisten Trainer hatten dieses Wort vergessen, weil es nur selten genutzt wurde. Der simulierte Delfin entdeckte die Regel, dass „Fenster" eine richtige Antwort auf „nicht Ball" war. Es war auch die richtige Reaktion auf „nicht Reifen" „nicht Tunnel" und „nicht Frisbee". Wir hatten versucht, dieses Muster zu meiden, indem wir vor jedem Versuch die Objekte im Tank veränderten, aber das Fenster konnten wir natürlich nicht entfernen. Es stellte sich heraus, dass Ake direkt neben das Fenster schwamm, wenn sie sich auf den Boden des Beckens sinken ließ, auch wenn ich das Fenster nicht sehen konnte!

Bruce: Was finden Sie an Prolog am besten?

Brian: Das deklarative Programmiermodell ist sehr reizvoll. Wenn Sie das Problem beschreiben können, haben Sie es generell gelöst. Bei den meisten Sprachen habe ich an irgendeiner Stelle versucht, mit dem Computer zu diskutieren: „Du weißt, was ich meine, mach es einfach!" Compilerfehler bei C und C++ wie „fehlendes Semikolon" sind dafür ein typisches Beispiel. Wenn Du ein Semikolon erwartest, warum fügst du keines ein und siehst, ob es das Problem löst?

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

Bruce A. Tate, Sieben Wochen, sieben Sprachen, O´Reilly, ISBN 97838972132279783897213227

Bei Prolog musste ich beim Arbeitsplan-Problem im Wesentlichen nur sagen, „Ich möchte einen Tag, der wie folgt aussieht, also mach mir einen!“, und das Programm machte mir einen.

Bruce: Wo hatten Sie die größten Schwierigkeiten?

Brian: Prolog schien für Probleme einen Alles-oder-nichts-Ansatz zu verwenden, zumindest bei den Problemen, an denen ich arbeitete. Beim Laborarbeitsplan lief das System 30 Minuten und gab dann entweder einen wunderschönen Plan oder einfach „no“ aus. „No“ hieß in diesem Fall, dass wir den Tag zu stark verplant hatten und es keine vollständige Lösung gab. Es lieferte uns aber keine Teillösung und keine Informationen darüber, wo wir uns verplant hatten.

Was man hier erkennt, ist ein extrem leistungsfähiges Konzept. Sie müssen nicht die Lösung eines Problems beschreiben, sondern nur das Problem. Und die Sprache für die Beschreibung des Problems ist Logik, reine Logik. Beginnen Sie mit Fakten und Folgerungen, und Prolog erledigt den Rest. Prolog-Programme bilden eine höhere Ebene der Abstraktion. Pläne und Verhaltensmuster sind Beispiele für Probleme, die gut zu Prolog passen.

Was wir am ersten Tag gelernt haben

Heute haben wir die grundlegenden Bausteine der Sprache Prolog kennengelernt. Statt Schritte zu programmieren, die Prolog zu einer Lösung führen, haben wir Wissen mittels reiner Logik kodiert. Prolog hat dann die schwierige Aufgabe übernommen, dieses Wissen zu verknüpfen, um Lösungen zu finden. Wir haben unsere Logik in Wissensdatenbanken gepackt und diese dann abgefragt.

Nach dem Aufbau einiger Wissensdatenbanken haben wir diese kompiliert und abgefragt. Die Abfragen (Queries) wiesen zwei Formen auf. Zum einen konnte die Query einfach ein Faktum angeben, und Prolog sagte uns, ob es stimmte oder nicht. Zum anderen konnten wir eine Query mit einer oder mehreren Variablen angeben, und Prolog berechnete dann alle Möglichkeiten, die es gab, um diese Fakten wahr werden zu lassen.

Wir haben gelernt, dass Prolog sich durch Regeln arbeitet, indem es die Klauseln nacheinander durchgeht. Für jede Klausel versucht Prolog, die gewünschten Ziele zu erreichen, indem es alle möglichen Kombinationen von Variablen durchgeht. Alle Prolog-Programme funktionieren auf diese Weise.

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

In den noch kommenden Abschnitten werden wir komplexere Schlussfolgerungen treffen. Wir werden auch sehen, wie man rechnen kann und komplexere Datenstrukturen wie Listen verwendet. Und wir zeigen Strategien, mit denen man über solche Listen iteriert.

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

Tag 1: Selbststudium

Finden Sie

- einige freie Einführungen in Prolog,
- ein Support-Forum (es gibt verschiedene) und
- eine Onlinereferenz für die von Ihnen verwendete Prolog-Version.

Machen Sie Folgendes:

- Bauen Sie eine einfache Wissensdatenbank mit einigen Ihrer Lieblingsbücher und -autoren auf.
- Finden Sie alle Bücher in Ihrer Wissensdatenbank, die von einem Autor geschrieben wurden.
- Bauen Sie eine Wissensdatenbank mit Musikern und Instrumenten auf. Stellen Sie auch Musiker und ihre Musikrichtung dar.
- Finden Sie alle Musiker, die Gitarre spielen.

4.3 Tag 2: Fünfzehn Minuten für Wapner

Der mürrische Richter Wapner aus „The People's Court" ist eine Obsession der Zentralfigur in *Rain Man*. Wie die meisten Autisten ist Raymond von allem besessen, was ihm vertraut ist. Er klammert sich an Richter Wapner und „The People's Court". Nachdem Sie sich durch diese rätselhafte Sprache gekämpft haben, sind Sie jetzt vielleicht bereit für Dinge, bei denen es Klick macht. Vielleicht sind Sie einer jener glücklichen Leser, bei denen es immer ganz von alleine Klick macht, aber wenn nicht, sollten Sie Ihren Mut zusammennehmen: Heute gibt es definitiv „fünfzehn Minuten für Wapner". Warten Sie geduldig ab. Wir brauchen noch weitere Werkzeuge in unserem Werkzeugkasten. Wir wollen lernen, wie man mit Rekursion, Mathematik und Listen umgeht. Los geht's.

Bruce A. Tate, Sieben Wochen, sieben Sprachen, O´Reilly, ISBN 97838972132279783897213227

Rekursion

Ruby und Io sind imperative Sprachen. Sie beschreiben jeden Schritt des Algorithmus. Prolog ist die erste der deklarativen Sprachen, die wir uns ansehen. Wenn Sie mit Collections wie Listen oder Bäumen arbeiten, verwenden Sie oft Rekursion anstelle von Iteration. Wir wollen uns die Rekursion ansehen und zeigen, wie man mit ihr Probleme mit einfachen Schlussfolgerungen lösen kann. Dann werden wir die gleiche Technik auf Listen und die Mathematik anwenden.

Sehen Sie sich die folgende Datenbank an. Sie enthält den umfangreichen Stammbaum der Waltons, der Figuren aus einem Film von 1963 und der nachfolgenden Serie. Sie drückt die Vaterbeziehung aus und leitet daraus eine Vorfahrenbeziehung ab. Weil „Vorfahre" Vater, Großvater oder Urgroßvater bedeuten kann, müssen wir die Regeln verschachteln oder iterieren. Da wir mit einer deklarativen Sprache arbeiten, werden wir sie verschachteln. Eine Klausel der `ancestor`-Klausel wird `ancestor` nutzen. In diesem Fall ist `ancestor(Z, Y)` ein rekursives Teilziel. Hier sehen Sie die Wissensdatenbank:

prolog/family.pl

```
father(zeb,         john_boy_sr).
father(john_boy_sr, john_boy_jr).

ancestor(X, Y) :-
    father(X, Y).
ancestor(X, Y) :-
    father(X, Z), ancestor(Z, Y).
```

`father` bildet die Kernmenge der Fakten, die unser rekursives Teilziel möglich machen. Die Regel `ancestor/2` besitzt zwei Klauseln. Besteht eine Regel aus mehreren Klauseln, muss nur eine Regel zutreffen, damit die Regel wahr wird. Betrachten Sie die Kommata zwischen den Teilzielen als `UND`- und die Punkte zwischen zwischen den Klauseln als `ODER`-Bedingungen. Die erste Klausel besagt, dass „X ein Vorfahre (engl.: ancestor) von Y ist, wenn X der Vater von Y ist". Das ist eine klare Beziehung. Wir können die Regel so ausprobieren:

```
| ?- ancestor(john_boy_sr, john_boy_jr).

true ?

no
```

Prolog meldet `true`, `john_boy_sr` ist ein Vorfahre von `john_boy_jr`. Diese erste Klausel ist von einem Faktum abhängig.

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

Die zweite Klausel ist etwas komplexer: `ancestor(X, Y) :- father(X, Z), ancestor(Z, Y)`. Diese Klausel besagt, das X ein Vorfahre von Y ist, wenn wir beweisen können, dass X der Vater von Z ist und Z gleichzeitig ein Vorfahre von Y.

Puh. Lassen Sie uns die zweite Klausel verwenden:

```
| ?- ancestor(zeb, john_boy_jr).

true ?
```

Ja, `zeb` ist ein Vorfahre von `john_boy_jr`. Wie immer können wir in einer Query Variablen nutzen:

```
| ?- ancestor(zeb, Who).

Who = john_boy_sr ? a

Who = john_boy_jr no
```

Und wir sehen, dass `zeb` ein Vorfahre von `john_boy_jr` und `john_boy_sr` ist. Das `ancestor`-Prädikat funktioniert auch anders herum:

```
| ?- ancestor(Who, john_boy_jr).

Who = john_boy_sr ? a

Who = zeb

(1 ms) no
```

Das ist ein wunderbare Sache, weil wir diese Regel unserer Wissensdatenbank für zwei Zwecke nutzen können. Wir können damit die Vorfahren ermitteln, aber auch die Nachkommen.

Ein Wort der Warnung: Wenn Sie rekursive Teilziele verwenden, müssen Sie sehr vorsichtig sein, weil jedes rekursive Teilziel Platz auf dem Stack benötigt, der irgendwann mal überläuft. Deklarative Sprachen lösen dieses Problem häufig mit einer Technik, die man als *Endrekursionsoptimierung* bezeichnet. Wenn Sie das rekursive Teilziel am Ende einer rekursiven Regel platzieren können, optimiert Prolog den Aufruf und bereinigt den Aufruf-Stack. Auf diese Weise bleibt die Nutzung des Speichers konstant. Unser Aufruf ist Endrekursiv („tail recursive"), weil das rekursive Teilziel `ancestor(Z, Y)` das letzte Ziel der rekursiven Regel ist. Wenn ihr Prolog-Programm mit einem Stack-Überlauf abstürzt, wissen Sie, dass es an der Zeit ist, nach einer Möglichkeit zu suchen, die Sache mithilfe von Endrekursion zu optimieren.

Nachdem wir diesen letzten organisatorischen Punkt geklärt haben, wollen wir uns Listen und Tupel ansehen.

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

Bruce A. Tate, Sieben Wochen, sieben Sprachen, O´Reilly, ISBN 97838972132279783897213227

Listen und Tupel

Listen und Tupel sind wichtige Bestandteile von Prolog. Sie geben Listen mit [1, 2, 3] an und Tupel mit (1, 2, 3). Listen sind Container variabler Länge, während Tupel Container fester Länge sind. Sowohl Listen als auch Tupel werden sehr viel mächtiger, wenn man sie unter dem Aspekt der Unifizierung betrachtet.

Unifizierung, Teil 2

Denken Sie daran, dass Prolog versucht, beide Seiten übereinstimmen zu lassen, wenn es Variablen unifiziert. Zwei Tupel stimmen überein, wenn die Anzahl der Elemente übereinstimmt und alle Elemente gleich sind. Sehen wir uns einige Beispiele an:

```
| ?- (1, 2, 3) = (1, 2, 3).

yes
| ?- (1, 2, 3) = (1, 2, 3, 4).

no
| ?- (1, 2, 3) = (3, 2, 1).

no
```

Zwei Tupel sind gleich, wenn alle Elemente gleich sind. Das erste Tupel ist ein Treffer, die Tupel im zweiten Beispiel haben nicht die gleiche Anzahl von Elementen, und im dritten Beispiel liegen die Elemente nicht in der gleichen Reihenfolge vor. Nun mischen wir ein paar Variablen unter:

```
| ?- (A, B, C) = (1, 2, 3).

A = 1
B = 2
C = 3

yes
| ?- (1, 2, 3) = (A, B, C).

A = 1
B = 2
C = 3

yes
| ?- (A, 2, C) = (1, B, 3).

A = 1
B = 2
C = 3

yes
```

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

Es spielt tatsächlich keine Rolle, auf welcher Seite die Variablen stehen. Die Unifizierung erfolgt, wenn Prolog sie gleich machen kann. Kommen wir nun zu den Listen. Sie können wie Tupel arbeiten:

```
| ?- [1, 2, 3] = [1, 2, 3].

yes
| ?- [1, 2, 3] = [X, Y, Z].

X = 1
Y = 2
Z = 3

yes
| ?- [2, 2, 3] = [X, X, Z].

X = 2
Z = 3

yes
| ?- [1, 2, 3] = [X, X, Z].

no
| ?- []= [].
```

Die beiden letzten Beispiele sind interessant. `[X, X, Z]` und `[2, 2, 3]` werden unifiziert, weil Prolog sie mit `X = 2` gleichsetzen kann. `[1, 2, 3] = [X, X, Z]` werden nicht unifiziert, weil wir `X` für die erste und die zweite Position verwendet haben und die Werte unterschiedlich sind. Listen besitzen eine Fähigkeit, die Tupel nicht haben. Sie können Listen mit `[Head|Tail]` zerlegen. Wenn Sie eine Liste mit diesem Konstrukt unifizieren, bindet `Head` das erste Element der Liste und `Tail` den Rest:

```
| ?- [a, b, c] = [Head|Tail].

Head = a
Tail = [b,c]

yes
```

`[Head|Tail]` unifiziert keine leere Liste, aber ein Liste mit einem Element funktioniert:

```
| ?- [] = [Head|Tail].

no
| ?- [a] = [Head|Tail].

Head = a
Tail = []

yes
```

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

Bruce A. Tate, Sieben Wochen, sieben Sprachen, O´Reilly, ISBN 97838972132279783897213227

Durch verschiedene Kombinationen kann das Ganze recht kompliziert werden:

```
| ?- [a, b, c] = [a|Tail].

Tail = [b,c]

(1 ms) yes
```

Prolog erkennt das a und unifiziert den Rest mit `Tail`. Man kann dieses `Tail` auch noch weiter in `Head` und `Tail` aufteilen:

```
| ?- [a, b, c] = [a|[Head|Tail]].

Head = b
Tail = [c]

yes
```

Oder man kann das dritte Element ermitteln:

```
| ?- [a, b, c, d, e] = [_, _|[Head|_]].

Head = c

yes
```

_ ist ein Platzhalter („Wildcard") und unfiziert mit allem. Er bedeutet: „Es ist mir egal, was an dieser Position steht." Wir haben Prolog angewiesen, die ersten beiden Elemente zu überspringen und den Rest in `Head` und `Tail` aufzuteilen. Das `Head` greift sich das dritte Element; das abschließende _ schnappt sich das `Tail`, ignoriert also den Rest der Liste.

Das sollte reichen, um loslegen zu können. Unifizierung ist ein mächtiges Werkzeug. Und zusammen mit Listen und Tupeln wird es noch leistungsfähiger.

Sie sollten nun ein grundlegendes Verständnis der elementaren Prolog-Datenstrukturen besitzen und wissen, wie die Unifizierung funktioniert. Wir sind nun so weit, dass wir diese Elemente mit Regeln und Folgerungen kombinieren können, um grundlegende mathematische Operationen mithilfe von Logik anzugehen.

Listen und Mathematik

Im nächsten Beispiel werden Sie sehen, wie man Rekursion und Mathematik auf Listen anwendet. Wir werden zählen, summieren und Durchschnittswerte ermitteln. Fünf Regeln erledigen die ganze Arbeit.

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

prolog/list_math.pl

```
count(0, []).
count(Count, [Head|Tail]) :- count(TailCount, Tail), Count is TailCount + 1.

sum(0, []).
sum(Total, [Head|Tail]) :- sum(Sum, Tail), Total is Head + Sum.

average(Average, List) :- sum(Sum, List), count(Count, List),Average is Sum/Count.
```

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

Das einfachste Beispiel ist count. Sie benutzen es so:

```
| ?- count(What, [1]).

What = 1 ? ;

no
```

Die Regeln sind extrem einfach. Die Anzahl einer leeren Liste ist 0. Die Anzahl einer Liste entspricht der Anzahl von Tail plus eins. Sehen wir uns Schritt für Schritt an, wie das funktioniert:

- Wir stoßen die Query count(What, [1]) an. Diese kann mit der ersten Regel nicht unifiziert werden, weil die Liste nicht leer ist. Um unser Ziel zu erreichen, machen wir mit der zweiten Regel weiter: count(Count, [Head|Tail]). Wir unifizieren, indem wir Count an Was binden, Head an 1 und Tail an [ ].

- Nach der Unifizierung ist count(TailCount, []) das erste Ziel. Wir versuchen, dieses Teilziel zu beweisen. Diesmal wird über die erste Regel unifiziert. Dadurch wird TailCount an 0 gebunden. Die erste Regel ist nun erfüllt, und wir können uns dem zweiten Ziel zuwenden.

- Nun evaluieren wir Count is TailCount + 1. Wir können Variablen unifizieren. TailCount ist an 0 gebunden, und wir binden Count an 0+ 1, also 1.

Und das war's. Wir haben keinen rekursiven Prozess definiert, sondern logische Regeln. Das nächste Beispiel addiert die Elemente einer Liste auf. Hier noch einmal der Code für diese Regeln:

```
sum(0, []).
sum(Total, [Head|Tail]) :- sum(Sum, Tail), Total is Head + Sum.
```

Dieser Code arbeitet genau wie die count-Regel. Er hat außerdem zwei Klauseln, einen Basisfall und einen rekursiven Fall. Die Verwendung ist ähnlich:

```
| ?- sum(What, [1, 2, 3]).
```

Bruce A. Tate, Sieben Wochen, sieben Sprachen, O´Reilly, ISBN 97838972132279783897213227

```
What = 6 ? ;

no
```

Sieht man sich das „imperativ“ an, funktioniert `sum` genau so, wie man es bei einer rekursiven Sprache erwartet. Die Summe einer leeren Liste ist null, und die Summe des Rests ist der Kopfteil (`Head`) plus der Summe des Fußteils (`Tail`).

Doch es gibt noch eine andere Interpretation. Wir haben Prolog eigentlich noch nicht mitgeteilt, wie man Summen berechnet. Wir haben Summen bloß durch Regeln und Ziele beschrieben. Um bestimmte Ziele erreichen zu können, muss die Logik-Engine bestimmte Teilziele erreichen. Die deklarative Interpretation ist wie folgt: „Die Summe einer leeren Liste ist null und die Summe einer Liste `Total`, wenn wir beweisen können, dass die Summe von Head plus Tail `Total` ist“. Wir ersetzen die Rekursion durch die Vorstellung von Zielen und Teilzielen.

In gleicher Weise ist die Anzahl bei einer leeren Liste null. Die Anzahl einer Liste entspricht eins für `Head` plus der Anzahl von `Tail`.

Wie bei der Logik können diese Regeln aufeinander aufbauen. Zum Beispiel können Sie `sum` und `count` zusammen nutzen, um den Mittelwert (`Average`) zu berechnen:

```
average(Average, List) :- sum(Sum, List), count(Count, List), Average  is Sum/Count.
```

Der Mittelwert von `List` ist also `Average`, wenn Sie Folgendes beweisen können:

- Die Summe dieser Liste ist `Sum`,
- die Anzahl dieser Liste ist `Count` und
- `Average` (also der Durchschnitt) ist `Sum/Count`.

Und es funktioniert genau so, wie Sie es erwarten:

```
| ?- average(What, [1, 2, 3]).

What = 2.0 ? ;

no
```

Regeln in beiden Richtungen verwenden

Jetzt sollten Sie ganz gut verstanden haben, wie Rekursion funktioniert. Ich schalte nun einen Gang hoch und spreche über eine kleine

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

Regel namens append. Die Regel append(List1, List2, List3) ist wahr, wenn List3 gleich List1 + List2 ist. Das ist eine sehr mächtige Regel, die Sie vielseitig einsetzen können.

Dieses kleine Stück Code hat es in sich. Sie können es auf unterschiedliche Art verwenden. Es ist ein Lügendetektor.

```
| ?- append([oil], [water], [oil, water]).

yes
| ?- append([oil], [water], [oil, slick]).

no
```

Es baut Listen auf:

```
| ?- append([tiny], [bubbles], What).

What = [tiny,bubbles]

yes
```

Es subtrahiert Listen:

```
| ?- append([dessert_topping], Who, [dessert_topping, floor_wax]).

Who = [floor_wax]

yes
```

Und es berechnet mögliche Permutationen:

```
| ?- append(One, Two, [apples, oranges, bananas]).

One= []
Two = [apples,oranges,bananas] ? a

One = [apples]
Two = [oranges,bananas]

One = [apples,oranges]
Two = [bananas]

One = [apples,oranges,bananas]
Two = []

(1 ms) no
```

Eine Regel liefert Ihnen also vier Möglichkeiten. Man möchte meinen, dass der Aufbau einer solchen Regel viel Code verlange. Finden wir heraus, wie viel es genau ist. Wir wollen das Prolog-append nachbilden, doch wir nennen es concatenate. Wir gehen das in mehreren Schritten an:

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

Bruce A. Tate, Sieben Wochen, sieben Sprachen, O´Reilly, ISBN 97838972132279783897213227

1. Wir schreiben eine Regel namens `concatenate(List1, List2, List3)`, die eine leere Liste mit `List1` verketten kann.

2. Wir fügen eine Regel ein, die ein Element aus `List1` mit `List2` verkettet.

3. Wir fügen eine Regel ein, die zwei und drei Elemente aus `List1` mit `List2` verkettet.

4. Wir sehen uns an, was wir verallgemeinern können.

Unser erster Schritt besteht darin, eine leere Liste mit `List1` zu verketten. Das ist eine recht einfache Regel:

prolog/concat_step_1.pl

```
concatenate([], List, List).
```

Kein Problem. `concatenate` ist wahr, wenn der erste Parameter eine Liste und die beiden nächsten Parameter gleich sind.

Das funktioniert:

```
| ?- concatenate([], [harry], What).

What = [harry]

yes
```

Weiter mit dem nächsten Schritt. Wir fügen eine Regel ein, die das erste Elemente von `List1` an den Anfang von `List2` setzt:

prolog/concat_step_2.pl

```
concatenate([], List, List).
concatenate([Head|[]], List, [Head|List]).
```

Für `concatenate(List1, List2, List3)` zerlegen wir `List1` in Head und Tail, wobei `Tail` eine leere Liste ist. Wir zerlegen unser drittes Element in Head und Tail und benutzen den Head von `List1` und `List2` als Tail. Vergessen Sie nicht, die Wissensdatenbank zu kompilieren. Auch das funktioniert:

```
| ?- concatenate([malfoy], [potter], What).

What = [malfoy,potter]

yes
```

Nun können wir eine Reihe weiterer Regeln definieren, die Listen der Länge 2 und 3 verketten. Sie funktionieren auf die gleiche Art und Weise:

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

prolog/concat_step_3.pl

```
concatenate([], List, List).
concatenate([Head|[]], List, [Head|List]).
concatenate([Head1|[Head2|[]]], List, [Head1, Head2|List]).
concatenate([Head1|[Head2|[Head3|[]]]], List, [Head1, Head2, Head3|List]).

| ?- concatenate([malfoy, granger], [potter], What).

What = [malfoy,granger,potter]

yes
```

Wir haben also einen Basisfall und eine Strategie, bei der jedes Teilziel die erste Liste verkleinert und die dritte Liste vergrößert. Die zweite bleibt unverändert. Wir besitzen nun genug Informationen, um das Ergebnis zu verallgemeinern. Hier die Verkettung mithilfe verschachtelter Regeln:

prolog/concat.pl

```
concatenate([], List, List).
concatenate([Head|Tail1], List, [Head|Tail2]) :-
  concatenate(Tail1, List, Tail2).
```

Dieser kurze und knappe Codeblock ist unglaublich einfach zu erklären. Die erste Klausel besagt, dass die Verkettung einer leeren Liste mit `List` genau diese `Liste` ergibt. Die zweite Klausel besagt, dass die Verkettung von `List1` mit `List2` genau dann `List3` ergibt, wenn die Head-Elemente von `List1` und `List3` gleich sind und Sie beweisen können, dass die Verkettung des Tail-Elements von `List1` mit `List2` das Tail-Element von `List3` ist. Die Einfachheit und Eleganz dieser Lösung sind ein Beleg für die Leistungsfähigkeit von Prolog.

Sehen wir uns an, was es mit der Query `concatenate([1, 2], [3], What)` macht. Wir gehen für jeden Schritt die Unifizierung durch. Denken Sie daran, dass wir die Regeln schachteln; jedes Mal, wenn wir versuchen, ein Teilziel zu überprüfen, haben wir es also mit anderen Variablen zu tun. Ich werde die wichtigen davon mit einem Buchstaben markieren, damit Sie den Überblick behalten. Ich zeige Ihnen, was passiert, wenn Prolog versucht, das nächste Teilziel zu beweisen.

- Wir beginnen mit
  `concatenate([1, 2], [3], What)`
- Die erste Regel trifft nicht zu, weil `[1, 2]` keine leere Liste ist. Wir unifizieren das zu
  `concatenate([1|[2]], [3], [1|Tail2-A]) :- concatenate([2], [3], [Tail2-A])`

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

Bruce A. Tate, Sieben Wochen, sieben Sprachen, O´Reilly, ISBN 97838972132279783897213227

Alles außer dem zweiten Tail-Element wird unifiziert. Wir machen nun mit den Zielen weiter. Lassen Sie uns die rechte Seite unifizieren.

- Wir versuchen, die Regel `concatenate<>([2], [3], [Tail2-A])` anzuwenden. Das liefert uns Folgendes:
  `concatenate([2|[]], [3], [2|Tail2-B]) :- concatenate([], [3], Tail2-B)`
  Beachten Sie, dass `Tail2-B` das Tail-Element von `Tail2-A` ist. Es ist nicht mit dem original `Tail2` identisch. Doch nun müssen wir die rechte Seite erneut unifizieren.

- `concatenate([], [3], Tail2-C) :- concatenate([], [3], [3])`

- So, wir wissen, dass `Tail2-C` `[3]` ist. Nun können wir uns durch die Kette zurückarbeiten. Sehen wir uns den dritten Parameter an und tragen `Tail2` bei jedem Schritt ein. `Tail2-C` ist [3], d. h. [2|`Tail2-2`] ist [`2, 3`], und schließlich ist [1|`Tail2`] [`1, 2, 3`]. Was ist also [`1, 2, 3`]?

Prolog erledigt hier eine ganze Menge Arbeit für Sie. Gehen Sie die Liste durch, bis Sie es verstehen. Die Unifizierung verschachtelter Teilziele ist ein Kernkonzept für die komplizierteren Aufgaben in diesem Buch.

Nun haben Sie einen tieferen Einblick in eine der vielseitigsten Prolog-Funktionen gewonnen. Nehmen Sie sich etwas Zeit, um sich die Lösungen anzusehen, und stellen Sie sicher, dass Sie sie verstanden haben.

Was wir am zweiten Tag gelernt haben

In diesem Abschnitt haben wir uns den grundlegenden Bausteinen zugewandt, mit deren Hilfe Prolog Daten organisiert: Listen und Tupel. Wir haben außerdem Regeln verschachtelt. Das erlaubt es uns, Probleme auszudrücken, die andere Sprachen mit Iteration lösen würden. Wir haben einen genaueren Blick auf die Prolog-Unifizierung geworfen und darauf, wie Prolog arbeitet, um mit beiden Seiten von `:-` und `=` mithalten zu können. Wir haben beim Schreiben von Regeln gesehen, dass wir logische Regeln beschreiben und nicht Algorithmen, und haben es dann Prolog überlassen, sich den Weg zur Lösung zu bahnen.

Wir haben auch Mathematik genutzt, und zwar grundlegende Arithmetik und verschachtelte Teilziele, um Summen und Durchschnitte zu berechnen.

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

Schließlich haben Sie gelernt, Listen zu verwenden. Wir haben ein oder mehrere Variablen innerhalb einer Liste mit Variablen verglichen und (noch wichtiger) das Head-Element einer Liste und die restlichen Elemente über das `[Head|Tail]`-Muster mit Variablen verglichen. Wir haben diese Technik genutzt, um rekursiv über Listen zu iterieren. Diese Grundbausteine dienen uns als Grundlage für die komplexen Probleme, die wir an Tag 3 lösen werden.

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

Tag 2: Selbststudium

Finden Sie Folgendes:

- Einige Implementierungen von Fibonacci-Folgen und -Brüchen. Wie funktionieren sie?
- Eine reale Community, die Prolog nutzt. Welche Probleme löst man heutzutage mit der Sprache?

Wenn Sie etwas anspruchsvolleres suchen, in das Sie sich verbeißen können, probieren Sie es mit den folgenden Problemen:

- Eine Implementierung der Türme von Hanoi. Wie funktioniert sie?
- Was sind einige der Probleme beim Umgang mit „Nicht“-Ausdrücken?
- Warum muss man bei Prolog mit der Negation so vorsichtig sein?

Machen Sie Folgendes:

- Kehren Sie die Elemente einer Liste um.
- Finden Sie das kleinste Element einer Liste.
- Sortieren Sie die Elemente einer Liste.

4.4 Tag 3: Die Bank sprengen

Sie sollten nun besser verstehen, warum ich Rain Man, den Autisten mit Savant-Syndrom, für Prolog gewählt habe. Auch wenn sie manchmal nur schwer zu verstehen ist, ist es verblüffend, sich Programmierung auf diese Weise vorzustellen. Eine meiner Lieblingsstellen in *Rain Man* ist, als Rays Bruder erkennt, dass Ray Karten zählen kann. Raymond und sein Bruder fahren nach Vegas und sprengen die Bank. In diesem Abschnitt werden Sie eine Seite von Prolog kennenlernen, die Ihnen ein Lächeln ins Gesicht zaubern wird. Die Kodierung der Bei-

Bruce A. Tate, Sieben Wochen, sieben Sprachen, O´Reilly, ISBN 97838972132279783897213227

spiele in diesem Kapitel hat mich gleichermaßen wahnsinnig und glücklich gemacht. Wir werden zwei berühmte Rätsel lösen, die genau der Kragenweite von Prolog entsprechen, nämlich Probleme mit Randbedingungen zu lösen.

Vielleicht wollen Sie sich an einigen dieser Rätsel selbst versuchen. Dann sollten Sie die Regeln beschreiben, die Sie bezüglich der Spiele kennen. Sie sollten nicht versuchen, Prolog eine Schritt-für-Schritt-Lösung zu zeigen. Wir beginnen mit einem kleinen Sudoku. Sie können dann im Rahmen Ihrer täglichen Übungen größere aufbauen. Danach wenden wir uns dem klassischen Acht-Damen-Problem zu.

Sudokus lösen

Das Programmieren des Sudokus hatte für mich etwas Magisches. Ein Sudoku ist ein Raster aus Zeilen, Spalten und Kästchen. Ein typisches Rätsel verwendet ein 9x9-Raster, bei dem einige Kästchen gefüllt sind und einige nicht. Jedes Kästchen des Rasters besitzt eine Nummer, bei einem 9x9-Quadrat von 1 bis 9. Ihre Aufgabe besteht darin, die Kästchen so mit Ziffern aufzufüllen, dass jede Zeile, jede Spalte und das Quadrat alle Ziffern enthält.

Wir wollen mit einem 4x4-Sudoku beginnen. Die Konzepte sind gleich, nur die Lösung ist kürzer. Wir wollen damit beginnen, die Welt so zu beschreiben, wie wir sie kennen. Abstrakt betrachtet, haben wir ein Brett mit vier Spalten, vier Zeilen und vier Quadraten. Die Tabelle zeigt die Quadrate 1 bis 4:

1	1	2	2
1	1	2	2
3	3	4	4
3	3	4	4

Die erste Aufgabe besteht darin, zu entscheiden, wie die Query aussehen soll. Das ist einfach. Wir haben ein Rätsel und eine Lösung der Form `sodoku(Puzzle, Solution)`. Der Benutzer kann ein Rätsel in Form einer Liste eingeben, wobei er unbekannte Zahlen durch Unterstriche ersetzt:

```
sodoku([_, _, 2, 3,
        _, _, _, _,
        _, _, _, _,
        3, 4, _, _],
        Solution).
```

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

Wenn eine Lösung existiert, liefert Prolog sie zurück. Als ich dieses Rätsel mit Ruby löste, musste ich mir Gedanken um die Algorithmen zur Lösung dieses Problems machen. Bei Prolog ist das nicht der Fall. Ich muss nur die Regeln des Spiels angeben.

Hier sind sie:

- Für ein gelöstes Rätsel müssen die Zahlen im Rätsel und in der Lösung gleich sein.
- Ein Sudoku-Brett ist ein Raster aus 16 Zellen mit Werten von 1 bis 4.
- Das Spielbrettt besteht aus vier Zeilen, vier Spalten und vier Quadraten.
- Ein Rätsel ist gültig, wenn sich die Elemente jeder Zeile, jeder Spalte und jedes Quadrats nicht wiederholen.

Wir wollen am Anfang beginnen. Die Zahlen in der Lösung und im Rätsel müssen übereinstimmen:

prolog/sudoku4_step_1.pl

```
sudoku(Puzzle, Solution) :-
        Solution = Puzzle.
```

Wir haben tatsächlich Fortschritte gemacht. Unser „Sudoku-Löser" funktioniert für den Fall, dass es keine leeren Stellen gibt:

```
| ?- sudoku([4, 1, 2, 3,
             2, 3, 4, 1,
             1, 2, 3, 4,
             3, 4, 1, 2], Solution).

Solution = [4,1,2,3,2,3,4,1,1,2,3,4,3,4,1,2]

yes
```

Das Format ist nicht schön, doch der Zweck ist klar. Wir erhalten 16 Zahlen (Zeile für Zeile) zurück. Doch wir sind ein wenig zu gierig:

```
| ?- sudoku([1, 2, 3], Solution).

Solution = [1,2,3]

yes
```

Unser Spielbrett ist ungültig, oder unser Lösungsprogramm meldet eine gültige Lösung. Natürlich müssen wir das Spielbrett auf 16 Elemente beschränken. Es gibt noch ein weiteres Problem. Die Werte in den Zellen können beliebig sein:

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

Bruce A. Tate, Sieben Wochen, sieben Sprachen, O´Reilly, ISBN 97838972132279783897213227

```
| ?- sudoku([1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6], Solution).

Solution = [1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6]

yes
```

Damit die Lösung gültig ist, muss sie Zahlen zwischen 1 und 4 verwenden. Dieses Problem wirkt sich für uns auf zweierlei Weise aus: Zum einen können wir einige ungültige Lösungen erlauben, zum anderen besitzt Prolog nicht genügend Informationen, um mögliche Werte für jede Zelle zu testen. Mit anderen Worten ist die Ergebnismenge nicht *geerdet*, d. h. wir haben keine Regeln definiert, die mögliche Werte für jede Zelle einschränken, weshalb Prolog die Werte nicht ermitteln kann.

Wir wollen diese Probleme lösen, indem wir die nächste Regel des Spiels implementieren. Regel 2 besagt, dass das Spielbrett 16 Felder mit Werten zwischen 1 und 4 besitzt. GNU-Prolog besitzt ein fest eingebautes Prädikat namens `fd_domain(Liste, Untergrenze, Obergrenze)`, um mögliche Werte auszudrücken. Dieses Prädikat gibt „wahr" zurück, wenn alle Werte der Liste zwischen Unter- und Obergrenze (einschließlich) liegen. Wir müssen nur sicherstellen, dass alle Werte des Sudokus im Bereich von 1 bis 4 liegen.

prolog/sudoku4_step_2.pl

```
sudoku(Puzzle, Solution) :-
        Solution = Puzzle,
        Puzzle = [S11, S12, S13, S14,
                  S21, S22, S23, S24,
                  S31, S32, S33, S34,
                  S41, S42, S43, S44],
        fd_domain(Puzzle, 1, 4).
```

Wir haben `Puzzle` mit einer Liste von 16 Variablen unifiziert und die Domäne der Zellen auf Werte zwischen 1 und 4 beschränkt. Nun scheitern wir, wenn das Rätsel nicht gültig ist:

```
| ?- sudoku([1, 2, 3], Solution).

no

| ?- sudoku([1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6], Solution).

no
```

Nun gelangen wir zum Kernstück der Lösung. Regel 3 besagt, dass unser Spielbrett aus Zeilen, Spalten und Quadraten besteht. Sie können erkennen, warum wir die Zellen so benannt haben. Die Zeilen zu beschreiben, ist ein einfacher Prozess:

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

```
Row1 = [S11, S12, S13, S14],
Row2 = [S21, S22, S23, S24],
Row3 = [S31, S32, S33, S34],
Row4 = [S41, S42, S43, S44],
```

Das gilt auch für die Spalten:

```
Col1 = [S11, S21, S31, S41],
Col2 = [S12, S22, S32, S42],
Col3 = [S13, S23, S33, S43],
Col4 = [S14, S24, S34, S44],
```

Und für die Quadrate:

```
Square1 = [S11, S12, S21, S22],
Square2 = [S13, S14, S23, S24],
Square3 = [S31, S32, S41, S42],
Square4 = [S33, S34, S43, S44].
```

Wenn wir unser Spielbrett in Teile zerlegt haben, können wir mit der nächsten Regel weitermachen. Das Spielbrett ist nur gültig, wenn alle Zeilen, Spalten und Quadrate keine sich wiederholenden Elemente enthalten. Wir werden ein Prädikat von GNU-Prolog verwenden, um auf sich wiederholende Elemente zu prüfen. `fd_all_different(List)` gibt „wahr" zurück, wenn alle Elemente der Liste unterschiedlich sind. Wir müssen eine Regel aufbauen, die überprüft, ob alle Zeilen, Spalten und Quadrate gültig sind. Wir verwenden dafür eine einfache Regel:

```
valid([]).
valid([Head|Tail]) :-
    fd_all_different(Head),
    valid(Tail).
```

Dieses Prädikat ist gültig, wenn alle Listen verschieden sind. Die erste Klausel besagt, dass eine leere Liste gültig ist. Die zweite Klausel besagt, dass eine Liste gültig ist, wenn die Einträge des ersten Elements alle verschieden sind und der Rest der Liste gültig ist.

Bleibt uns nur noch, die `valid(Liste)`-Regel aufzurufen:

```
valid([Row1, Row2, Row3, Row4,
       Col1, Col2, Col3, Col4,
       Square1, Square2, Square3, Square4]).
```

Ob Sie es glauben oder nicht, wir sind fertig. Unser Programm kann ein 4x4-Sudoku lösen:

```
| ?- sudoku([_, _, 2, 3,
             _, _, _, _,
             _, _, _, _,
             3, 4, _, _],
             Solution).
```

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

Bruce A. Tate, Sieben Wochen, sieben Sprachen, O´Reilly, ISBN 97838972132279783897213227

```
Solution = [4,1,2,3,2,3,4,1,1,2,3,4,3,4,1,2]

yes
```

Bringen wir das in eine etwas freundlichere Form, haben wir die Lösung:

4 1 2 3

2 3 4 1

1 2 3 4

3 4 1 2

Hier noch einmal das vollständige Programm:

prolog/sudoku4.pl

```
valid([]).
valid([Head|Tail]) :-
    fd_all_different(Head),
    valid(Tail).

sudoku(Puzzle, Solution) :-
        Solution = Puzzle,

        Puzzle = [S11, S12, S13, S14,
                  S21, S22, S23, S24,
                  S31, S32, S33, S34,
                  S41, S42, S43, S44],

        fd_domain(Solution, 1, 4),

         Row1 = [S11, S12, S13, S14],
         Row2 = [S21, S22, S23, S24],
         Row3 = [S31, S32, S33, S34],
         Row4 = [S41, S42, S43, S44],

         Col1 = [S11, S21, S31, S41],
         Col2 = [S12, S22, S32, S42],
         Col3 = [S13, S23, S33, S43],
         Col4 = [S14, S24, S34, S44],

         Square1 = [S11, S12, S21, S22],
         Square2 = [S13, S14, S23, S24],
         Square3 = [S31, S32, S41, S42],
         Square4 = [S33, S34, S43, S44],

         valid([Row1, Row2, Row3, Row4,
                Col1, Col2, Col3, Col4,
                Square1, Square2, Square3, Square4]).
```

Wenn Sie Ihre Prolog-Erleuchtung noch nicht hatten, sollte Ihnen dieses Beispiel einen Schubs in die richtige Richtung geben. Wo ist das Programm? Tja, wir haben kein Programm geschrieben. Wir haben die

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

Regeln des Spiels beschrieben: Das Spielbrett besteht aus 16 Zellen mit Zahlen zwischen 1 und 4, und in keiner der Zeilen, Spalten und Quadrate dürfen sich Werte wiederholen. Das Rätsel benötigt zur Lösung ein paar Dutzend Zeilen Code und keinerlei Wissen über irgendwelche Sudoku-Lösungsstrategien. In den täglichen Übungen erhalten Sie die Chance, ein neunzeiliges Sudoku zu lösen. Das sollte nicht allzu schwer sein.

Dieses Rätsel ist ein großartiges Beispiel für die Art von Problemen, die Prolog gut lösen kann. Wir haben eine Reihe von Einschränkungen, die sich einfach ausdrücken, aber nur schwer lösen lassen. Sehen wir uns ein weiteres Rätsel an, bei dem es um stark eingeschränkte Ressourcen geht: das Acht-Damen-Problem.

Acht Damen

Beim Acht-Damen-Problem werden acht Damen auf einem Schachbrett plaziert. Keine der Damen darf die gleiche Zeile, Spalte oder Diagonale nutzen. Auf den ersten Blick mag das ein triviales Problem sein. Nur ein Kinderspiel. Auf einer anderen Ebene kann man die Zeilen, Spalten und Diagonalen als beschränkte Ressourcen betrachten. Die Industrie ist voller Probleme, die eine Lösung derart beschränkter Systeme verlangen. Schauen wir uns an, wie wir dieses Problem mit Prolog lösen können.

Sehen wir uns zuerst an, wie die Query aussehen muss. Wir können jede Dame als `(Row, Col)` beschreiben, ein Tupel mit Zeile und Spalte. Ein Brett (`Board`) ist eine Liste von Tupeln. `eight_queens(Board)` erreicht sein Ziel, wenn wir über ein gültiges Brett verfügen. Unsere Query wird wie folgt aussehen:

```
eight_queens([(1, 1), (3, 2), ...]).
```

Sehen wir uns die Ziele an, die wir erfüllen müssen, um das Rätsel zu lösen. Wenn Sie sich an diesem Spiel versuchen wollen, ohne sich die Lösung anzusehen, sehen Sie sich nur diese Ziele an. Die vollständige Lösung behandle ich erst später in diesem Kapitel.

- Auf einem Brett sind acht Damen.
- Jede Dame hat eine Zeile von 1 bis 8 sowie eine Spalte von 1 bis 8.
- Zwei Damen dürfen nicht in der gleichen Zeile stehen.
- Zwei Damen dürfen nicht in der gleichen Spalte stehen.

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

Bruce A. Tate, Sieben Wochen, sieben Sprachen, O´Reilly, ISBN 97838972132279783897213227

- Zwei Damen dürfen nicht in derselben Diagonalen stehen (Südwest nach Nordost).
- Zwei Damen dürfen nicht in derselben Diagonalen stehen (Nordwest nach Südost).

Zeilen und Spalten müssen einmalig sein, doch bei den Diagonalen müssen wir etwas vorsichtiger sein. Jede Dame liegt auf zwei Diagonalen. Die eine verläuft von unten links (Nordwest) nach oben rechts (Südost) und die andere von oben links nach unten rechts (siehe Abbildung 4.2). Doch diese Regeln sollten sich recht einfach programmieren lassen.

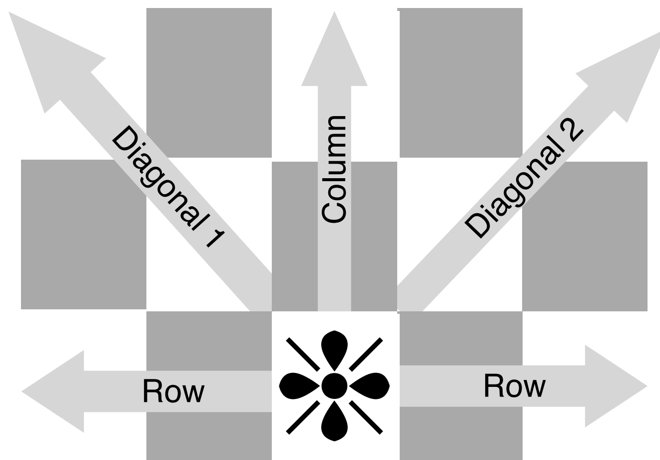

Abbildung 4.2: Regeln für die acht Damen

Wir wollen erneut mit dem ersten Punkt der Liste beginnen. Ein Spielbrett besitzt acht Damen. Das bedeutet, dass unsere Liste die Größe 8 haben muss. Das ist einfach. Wir können das von uns an früherer Stelle entwickelte `count`-Prädikat oder einfach das fest in Prolog eingebaute Prädikat `length` verwenden. `length(List, N)` gibt „wahr“ zurück, wenn die `Liste` `N` Elemente besitzt. Diesmal zeige ich Ihnen nicht jedes Ziel in Aktion, sondern gehe mit Ihnen die Ziele durch, die wir zur Lösung des gesamten Problems erreichen müssen. Hier also das erste Ziel:

```
eight_queens(List) :- length(List, 8).
```

Als Nächstes müssen wir sicherstellen, dass jede Dame aus unserer Liste gültig ist. Wir entwickeln eine Regel, die überprüft, ob eine Dame gültig ist:

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

```
valid_queen((Row, Col)) :-
    Range = [1,2,3,4,5,6,7,8],
    member(Row, Range), member(Col, Range).
```

Das Prädikat member macht genau, was Sie erwarten: Es überprüft die Zugehörigkeit. Eine Dame ist gültig, wenn sowohl die Zeile als auch die Spalte Integer-Werte zwischen 1 und 8 sind. Als Nächstes entwickeln wir eine Regel, die überprüft, ob das gesamte Spielbrett aus gültigen Damen besteht:

```
valid_board([]).
valid_board([Head|Tail]) :- valid_queen(Head), valid_board(Tail).
```

Ein Spielbrett ist gültig, wenn es leer ist, und auch wenn das erste Element eine gültige Dame und der Rest des Spielbretts gültig ist.

Weiter geht's. Die nächste Regel lautet, dass zwei Damen nicht dieselbe Zeile verwenden dürfen. Um die nächsten Einschränkungen lösen zu können, benötigen wir ein wenig Hilfe. Wir zerlegen das Programm in kleinere Teile, die uns dabei helfen, das Problem zu beschreiben: Was sind Zeilen, Spalten und Diagonalen? Zuerst kommen die Zeilen dran. Wir entwickeln eine Funktion namens rows(Queens, Rows). Diese Funktion liefert „wahr" zurück, wenn Rows die Liste der Row-Elemente aller Damen ist.

```
rows([], []).
rows([(Row, _)|QueensTail], [Row|RowsTail]) :-
  rows(QueensTail, RowsTail).
```

Hier brauchen wir ein wenig Fantasie, wenn auch nicht allzu viel. rows für eine leere Liste ist die leere Liste und rows(Queens, Rows) ist Rows, wenn die Zeile der ersten Dame in der Liste dem ersten Element von Rows entspricht, und wenn rows des Tail-Elements von Queens mit dem Tail-Element von Rows übereinstimmt. Falls Sie das verwirrt, gehen Sie sie mit ein paar Testlisten durch. Glücklicherweise funktionieren die Spalten genauso, nur dass wir hier die Spalten anstelle der Zeilen verwenden:

```
cols([], []).
cols([(_, Col)|QueensTail], [Col|ColsTail]) :-
  cols(QueensTail, ColsTail).
```

Die Logik funktioniert exakt wie bei den Zeilen, nur dass wir diesmal anstelle des ersten das zweite Element des Damen-Tupels prüfen.

Nun gilt es, die Diagonalen zu nummerieren. Die einfachste Lösung bilden einige einfache Additionen und Subtraktionen. Wenn Nord und West 1 sind, weisen wir den von Nordwest nach Südost verlaufenden

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

Bruce A. Tate, Sieben Wochen, sieben Sprachen, O´Reilly, ISBN 97838972132279783897213227

Diagonalen den Wert `Col - Row` zu. Hier das Prädikat, das diese Diagonalen festhält:

```
diags1([], []).
diags1([(Row, Col)|QueensTail], [Diagonal|DiagonalsTail]) :-
  Diagonal is Col - Row,
  diags1(QueensTail, DiagonalsTail).
```

Diese Regel funktioniert genau wie `rows` und `cols`, besitzt aber eine weitere Einschränkung: `Diagonal is Col -- Row`. Beachten Sie, dass das keine Unifizierung ist! Es handelt sich um ein Prädikat und stellt sicher, dass wir eine fundierte Lösung abliefern. Abschließend verarbeiten wir Südost nach Nordwest wie folgt:

```
diags2([], []).
diags2([(Row, Col)|QueensTail], [Diagonal|DiagonalsTail]) :-
  Diagonal is Col + Row,
  diags2(QueensTail, DiagonalsTail).
```

Diese Formel ist etwas komplizierter, also probieren Sie ruhig einige Werte aus, bis Sie sich sicher sind, dass Damen mit der gleichen Summe von Zeile und Spalte tatsächlich auf derselben Diagonalen liegen. Nachdem wir nun über die Regeln verfügen, mit deren Hilfe wir Zeilen, Spalten und Diagonalen beschreiben können, müssen wir nur noch sicherstellen, dass die Zeilen, Spalten und Diagonalen alle unterschiedlich sind.

Damit Sie noch mal den ganzen Kontext sehen, folgt die vollständige Lösung. Die letzten acht Klauseln bilden die Tests für Zeilen und Spalten.

prolog/queens.pl

```
valid_queen((Row, Col)) :-
    Range = [1,2,3,4,5,6,7,8],
    member(Row, Range), member(Col, Range).

valid_board([]).
valid_board([Head|Tail]) :- valid_queen(Head), valid_board(Tail).

rows([], []).
rows([(Row, _)|QueensTail], [Row|RowsTail]) :-
  rows(QueensTail, RowsTail).

cols([], []).
cols([(_, Col)|QueensTail], [Col|ColsTail]) :-
  cols(QueensTail, ColsTail).

diags1([], []).
diags1([(Row, Col)|QueensTail], [Diagonal|DiagonalsTail]) :-
  Diagonal is Col - Row,
  diags1(QueensTail, DiagonalsTail).
```

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

```
diags2([], []).
diags2([(Row, Col)|QueensTail], [Diagonal|DiagonalsTail]) :-
  Diagonal is Col + Row,
 diags2(QueensTail, DiagonalsTail).

eight_queens(Board) :-
  length(Board, 8),
  valid_board(Board),

rows(Board, Rows),
cols(Board, Cols),
diags1(Board, Diags1),
diags2(Board, Diags2),

fd_all_different(Rows),
fd_all_different(Cols),
fd_all_different(Diags1),
fd_all_different(Diags2).
```

Jetzt würde das Programm laufen, wenn Sie es ausführen … und laufen … und laufen. Es gibt einfach zu viele Kombinationen, um sie effektiv durchgehen zu können. Wenn wir mal scharf nachdenken, wissen wir aber, dass es nur eine Dame pro Zeile geben kann. Man kann der Lösung näher kommen, indem man folgendes Spielbrett vorgibt:

```
| ?- eight_queens([(1, A), (2, B), (3, C), (4, D), (5, E), (6, F), (7, G), (8, H)]).

A= 1
B= 5
C= 8
D= 6
E= 3
F= 7
G= 2
H= 4?
```

Das funktioniert, aber das Programm arbeitet immer noch zu lang. Wir können die Auswahlmöglichkeiten für die Zeilen leicht eliminieren und die API vereinfachen, wo wir gerade dabei sind. Hier eine leicht optimierte Fassung:

prolog/optimized_queens.pl

```
valid_queen((Row, Col)) :- member(Col, [1,2,3,4,5,6,7,8]).

valid_board([]).
valid_board([Head|Tail]) :- valid_queen(Head), valid_board(Tail).

cols([], []).
cols([(_, Col)|QueensTail], [Col|ColsTail]) :-
  cols(QueensTail, ColsTail).

diags1([], []).
diags1([(Row, Col)|QueensTail], [Diagonal|DiagonalsTail]) :-
```

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

Bruce A. Tate, Sieben Wochen, sieben Sprachen, O´Reilly, ISBN 97838972132279783897213227

```
  Diagonal is Col - Row,
  diags1(QueensTail, DiagonalsTail).

diags2([], []).
diags2([(Row, Col)|QueensTail], [Diagonal|DiagonalsTail]) :-
  Diagonal is Col + Row,
  diags2(QueensTail, DiagonalsTail).

eight_queens(Board) :-
  Board = [(1, _), (2, _), (3, _), (4, _), (5, _), (6, _), (7, _), (8, _)],
  valid_board(Board),

cols(Board, Cols),
diags1(Board, Diags1),
diags2(Board, Diags2),

fd_all_different(Cols),
fd_all_different(Diags1),
fd_all_different(Diags2).
```

Philosophisch betrachtet, haben wir eine wesentliche Änderung vorgenommen. Wir haben das Spielbrett mit `(1, _)`, `(2, _)`, `(3, _)`, `(4, _)`, `(5, _)`, `(6, _)`, `(7, _)`, `(8, _)` verglichen, um die Gesamtzahl der Permutationen deutlich zu reduzieren. Wir haben auch alle Regeln bezüglich der Zeilen entfernt. Auf meinem alten MacBook werden alle Lösungen innerhalb von drei Minuten berechnet.

Erneut ist das Endergebnis recht ansprechend. Wir haben kaum Wissen über die Lösungsmenge eingebracht. Wir haben nur die Regeln des Spiels beschrieben und ein wenig Logik angewandt, um das Ganze ein wenig zu beschleunigen. Bei den richtigen Problemen kann ich mich für Prolog tatsächlich erwärmen.

Was wir an Tag 3 gelernt haben

Heute haben wir einige Ideen zusammengefasst, die man verwenden kann, um mit Prolog klassische Denkaufgaben zu lösen. Die restriktionsbasierten Probleme weisen viele Charakteristika industrieller Anwendungen auf. Führen Sie die Restriktionen auf und zaubern Sie eine Lösung hervor. Wir würden bei der imperativen Programmierung einen SQL-Join über neun Tabellen nicht einmal in Erwägung ziehen, während wir gleichzeitig nicht zögern, logische Probleme auf diese Weise zu lösen.

Wir haben mit einem Sudoku begonnen. Prologs Lösung war beeindruckend einfach. Wir haben 16 Variablen auf Zeilen, Spalten und Quadrate abgebildet. Dann haben wir die Regeln des Spiels beschrieben und jede Zeile, Spalte und jedes Quadrat gezwungen, „einmalig“ zu

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

sein. Prolog hat sich dann methodisch durch die Möglichkeiten gearbeitet und schnell eine Lösung gefunden. Wir haben Platzhalter und Variablen verwendet, um eine intuitive API aufzubauen, doch wir haben keinerlei Hilfestellungen für die Lösungstechniken gegeben.

Als Nächstes haben wir das Rätsel der acht Damen gelöst. Wieder haben wir die Regeln des Spiels kodiert und Prolog eine Lösung ausarbeiten lassen. Dieses klassische Problem ist mit 92 Lösungen ziemlich rechenintensiv, doch selbst mit unserem einfachen Ansatz war es innerhalb weniger Minuten zu lösen.

Ich kenne noch lange nicht alle Tricks und Techniken, um anspruchsvolle Sudokus zu lösen, aber mit Prolog muss ich sie auch gar nicht wissen. Ich muss nur die Regeln des Spiels kennen.

Tag 3: Selbststudium

Finden Sie Folgendes:

- Prolog besitzt einige Features zur Ein- und Ausgabe. Finden Sie `print`-Prädikate, die Variablen ausgeben.
- Finden Sie eine Möglichkeit, die `print`-Prädikate so einzusetzen, dass nur erfolgreiche Lösungen ausgegeben werden. Wie funktionieren diese Lösungen?

Machen Sie Folgendes:

- Modifizieren Sie den Sudoku-Löser so, das er 6x6- (die Quadrate sind 3x2) und 9x9-Rätsel lösen kann.
- Lassen Sie den Sudoku-Löser schönere Lösungen ausgeben. Wer Rätsel mag, kann sich leicht in Prolog verlieren. Wenn Sie tiefer in die von mir vorgestellten Rätsel einsteigen wollen, sind die acht Damen ein guter Ausgangspunkt.
- Lösen Sie das Acht-Damen-Problem mithilfe einer Liste von Damen. Anstelle eines Tupels repräsentieren Sie jede Dame durch einen Integer-Wert zwischen 1 und 8. Bestimmen Sie die Zeile einer Dame anhand ihrer Position und die Spalte über den Wert in der Liste.

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

Bruce A. Tate, Sieben Wochen, sieben Sprachen, O´Reilly, ISBN 97838972132279783897213227

4.5 Prolog zusammengefasst

Prolog ist eine der ältesten Sprachen in diesem Buch, doch die Ideen sind auch heute noch interessant und relevant. Prolog bedeutet Programmieren mit Logik. Wir haben Prolog verwendet, um Regeln zu verarbeiten, die aus Klauseln bestehen, die wiederum aus einer Reihe von Zielen bestehen.

Prolog-Programmierung besteht aus zwei wesentlichen Schritten. Sie beginnen mit dem Aufbau einer Wissensdatenbank, die aus logischen Fakten und Schlussfolgerungen über die Problemdomäne besteht. Als Nächstes kompilieren Sie die Wissensdatenbank und stellen Fragen zu dieser Domäne. Einige der Fragen können Annahmen sein, auf die Prolog mit `yes` oder `no` antwortet. Andere Fragen verwenden Variablen. Prolog füllt diese Lücken so auf, dass diese (Ab-)Fragen wahr werden.

Anstelle einfacher Zuweisungen verwendet Prolog einen als *Unifizierung* bezeichneten Prozess, der dafür sorgt, dass Variablen auf beiden Seiten des Systems übereinstimmen. Manchmal muss Prolog viele verschiedene mögliche Kombinationen durchgehen, um die Variablen für eine Schlussfolgerung unifizieren zu können.

Stärken

Prolog ist für eine Vielzahl von Problemen geeignet, die von Flugplänen bis zu Finanzderivaten reichen. Prolog (und andere Sprachen seiner Art) zu lernen, ist nicht leicht, aber angesichts der anspruchsvollen Probleme, die es zu lösen vermag, ist es die Mühe wert.

Denken Sie an Brian Tarbox' Arbeit mit den Delfinen: Er konnte einfache Schlussfolgerungen über die Welt ziehen und mit einer komplexen Schlussfolgerung über das Verhalten von Delfinen einen Durchbruch erzielen. Er war auch in der Lage, stark eingeschränkte Ressourcen zu nehmen und mit Prolog einen Zeitplan zu finden, in den sie reinpassten. Es gibt so einige Bereiche, in denen Prolog heute noch aktiv eingesetzt wird.

Natürliche Sprachverarbeitung

Prolog wurde zuerst zur Sprachverarbeitung genutzt. Tatsächlich können Prolog-Sprachmodelle natürliche Sprache nehmen, eine Wissensbasis aus Fakten und Schlussfolgerungen anwenden und die komplexe, ungenaue Sprache in konkrete Regeln umwandeln, die für Computer geeignet sind.

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

Spiele

Spiele werden immer komplexer, insbesondere die Modellierung von Konkurrenten oder Feinden. Prolog-Modelle können das Verhalten anderer Figuren im System recht einfach ausdrücken. Prolog kann auch unterschiedliche Verhaltensweisen für verschiedene Arten von Feinden definieren, was für eine realistischere und unterhaltsamere Spielerfahrung sorgt.

Semantisches Web

Das semantische Web ist der Versuch, Dienste und Informationen mit einer Bedeutung anzureichern. Dadurch soll es einfacher werden, Anfragen zu beantworten. Ein RDF (Resource Description Framework) ermöglicht die grundlegende Beschreibung von Ressourcen. Ein Server kann diese Ressourcen in eine Wissensdatenbank kompilieren. Dieses Wissen zusammen mit Prologs natürlicher Sprachverarbeitung kann für den Endanwender eine ergiebige Erfahrung sein. Es existieren viele Prolog-Pakete, die die Art der Funktionalität im Kontext eines Webservers bereitstellen.

Künstliche Intelligenz

Künstliche Intelligenz (KI) konzentriert sich darauf, Maschinen intelligentes Verhalten beizubringen. Diese Intelligenz kann verschiedene Formen annehmen, doch in allen Fällen verändert ein „Agent" auf komplexen Regeln basierend sein Verhalten. Prolog tut sich auf diesem Gebiet hervor, insbesondere wenn die Regeln konkret sind und auf formaler Logik basieren. Aus diesem Grund wird Prolog manchmal auch als *logische Programmiersprache* bezeichnet.

Planung

Prolog glänzt bei der Arbeit mit beschränkten Ressourcen. Prolog ist häufig zur Entwicklung von Betriebssystem-Schedulern und anderen anspruchsvollen Schedulern eingesetzt worden.

Schwächen

Prolog ist eine Sprache, die mit der Zeit mithalten konnte. Dennoch ist die Sprache auf vielerlei Arten veraltet und weist signifikante Beschränkungen auf.

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

Bruce A. Tate, Sieben Wochen, sieben Sprachen, O´Reilly, ISBN 97838972132279783897213227

Nützlichkeit

Prolog glänzt in seiner Kerndomäne, doch die Logikprogrammierung ist eine recht enge Nische. Prolog ist keine Allzweck-Programmiersprache und weist im Bezug auf das Sprachdesign einige Einschränkungen auf.

Sehr große Datenmengen

Prolog verwendet eine tiefenbasierte Suche („depth-first search“) im Entscheidungsbaum, um alle möglichen Kombinationen mit den Regeln zu vergleichen. Verschiedene Sprachen und Compiler können diese Aufgabe gut optimieren. Dennoch ist diese Strat egie konzeptbedingt ziemlich rechenintensiv, insbesondere bei sehr großen Datenmengen. Deshalb sind Prolog-Nutzer außerdem gezwungen, sich Kenntnisse über die Funktionsweise der Sprache anzueignen, damit die Datenmenge handhabbar bleibt.

Mischen imperativer und deklarativer Modelle

Wie bei vielen Sprachen aus der funktionalen Familie (insbesondere denjenigen, die stark auf Rekursion setzen), müssen Sie durchschauen, wie Prolog rekursive Regeln auflöst. Häufig müssen endrekursive Regeln verwendet werden, um selbst mittelschwere Probleme lösen zu können. Es ist ganz einfach, Prolog-Anwendungen zu entwickeln, die ab einer bestimmten Datenmenge nicht mehr gut skalieren. Sie müssen häufig ein tieferes Verständnis der Funktionsweise von Prolog mitbringen, um effektive Regeln entwickeln zu können, die akzeptabel skalieren.

Abschließende Gedanken

Während ich die Sprachen in diesem Buch durcharbeitete, hätte ich mir regelmäßig selbst in den Hintern treten können. Ich musste nämlich bemerken, dass ich jahrelang Schrauben mit dem Hammer in die Wand gehauen hatte. Prolog war ein besonders schmerzliches Beispiel dafür. Wenn Sie ein Problem finden, das für Prolog besonders gut geeignet ist, sollten Sie den Vorteil wahrnehmen. In einem solchen Fall können Sie diese regelbasierte Sprache am besten in Kombination mit einer anderen Allzwecksprache einsetzen, genau wie Sie SQL in Ruby oder Java verwenden. Wenn Sie sie sorgfältig miteinander verknüpfen, werden Sie auf lange Sicht gut fahren.

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

Dies ist ein Auszug aus dem Buch "Sieben Wochen, sieben Sprachen", ISBN 978-3-89721-322-7
http://www.oreilly.de/catalog/sevenlangsevweekger/
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

Bruce A. Tate, Sieben Wochen, sieben Sprachen, O´Reilly, ISBN 97838972132279783897213227