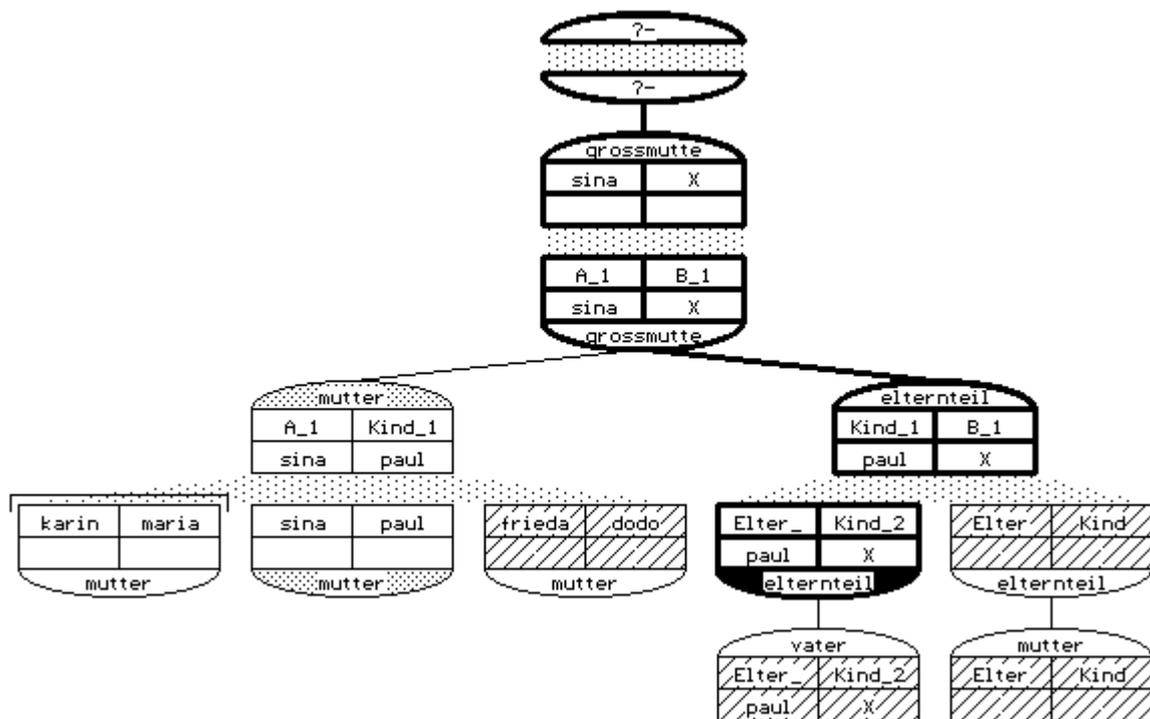


Informatik mit Prolog

von Gerhard Röhner

1. Auflage 1995



Ich kann freilich nicht sagen, ob es besser werden wird,
wenn es anders wird; aber soviel kann ich sagen,
es muß anders werden, wenn es gut werden soll.
GEORG CHRISTOPH LICHTENBERG 1793

1 Vorwort

Daß die Informatik in der gymnasialen Oberstufe anders werden muß, damit das Fach überleben kann, das fordern viele Didaktiker mit Nachdruck. Aber wie das geschehen soll, ist sehr umstritten und überzeugende Konzepte sind wenig in Sicht. Im Grunde genommen geht es um die Frage, ob eine so dynamische Disziplin wie die Informatik, deren Entwicklung auch im Hochschulbereich keineswegs abgeschlossen ist und die durch die rasante Entwicklung der Technik vor täglich neuen Herausforderungen steht, überhaupt ihren Platz in der Schule behaupten kann. Die allgemeinbildenden Fächer der gymnasialen Oberstufe sind im wesentlichen auf statische Inhalte festgelegt, es herrschen die analytischen Denkweisen vor. Dem beruflichen Schulwesen ist es vorbehalten, Wissen zu vermitteln, das auf die Konstruktion von Produkten ausgerichtet ist und bei dem synthetisierende Denkweisen vermittelt werden. Ist also die Informatik in der gymnasialen Oberstufe fehl am Platz?

Angesichts der immer noch steigenden Bedeutung der Informations- und Kommunikationstechnologien für die Gesellschaft und den Einzelnen wäre es absurd, den Ausstieg der allgemeinbildenden Schule aus diesem Bereich zu betreiben. In der gymnasialen Oberstufe ist die Informatik das einzige Grundlagenfach, das eine integrierte Sicht auf die vielfältigen Anwendungsgebiete dieser Techniken ermöglicht und grundlegende Verfahren und Methoden zu vermitteln vermag. Da die Bezugsinhalte raschen Veränderungen unterworfen sind, muß auch das Fach selbst - in den Anwendungsbezügen, nicht in den Grundlagen - dynamisch sein. Ein Rückzug auf nur „fundamentale Ideen“ (Informatik-Duden) oder statische Inhalte wie „Sprache“ (CLAUS) würde Informatik in der Praxis zu einem langweiligen Fach machen, für das sich Schülerinnen und Schüler nicht motivieren lassen. Da Informatik aber ein freiwilliges Angebot darstellt, das von der Motivation und Begeisterung für die Sache lebt, muß Informatik dynamisch bleiben - oder es wird mangels Interesse verschwinden.

Informatik mit PROLOG ist ein Ansatz, der m.E. einen Weg bieten kann. Einerseits bietet PROLOG die Möglichkeit, viele grundlegende Verfahren der Informatik, elementare Methoden und fundamentale Ideen kennenzulernen. Der Zugang hierzu ist für den Anfänger leichter als mit herkömmlichen Programmiersprachen, allerdings stellt PROLOG mit zunehmender Komplexität der Probleme hohe Anforderungen an das logische Denken und Durchdringen

der Aufgabenstellung. Andererseits sind Bezüge zu sehr interessanten Anwendungsgebieten wie maschinelle Sprachverarbeitung, Expertensysteme, künstliche Intelligenz herstellbar, die von großer aktueller Bedeutung sind und mit herkömmlichen Programmiersprachen gar nicht oder nur schwer erschlossen werden können.

Gerhard RÖHNER hat im Zusammenhang mit der Lehrerweiterbildung für Informatik an der TH Darmstadt ein Konzept für den Unterricht entwickelt, das dem neuen Kursstrukturplan in Hessen Rechnung trägt und die aktuelle didaktische Diskussion um das Fach Informatik in der gymnasialen Oberstufe aufgreift. Ich glaube, daß dieses Konzept - verbunden mit dem Angebot zweier ausgereifter PROLOG-Systeme - eine gute Chance hat, dem Informatikunterricht neue Impulse zu vermitteln. Gleichzeitig stellt dieses Konzept eine Realisierungsmöglichkeit für die Umsetzung des Kursstrukturplans in schulische Realität dar. Weitere mögliche Umsetzungsvarianten werden in dieser HIBS-Reihe folgen.

Jürgen BURKERT

federführendes Mitglied der Kursstrukturplangruppe Informatik

INHALTSVERZEICHNIS**1 PROLOG IN DER SCHULE**

- 1.1 Wozu Prolog in der Schule? 1
- 1.2 Prolog zwischen Anspruch und Wirklichkeit 5
- 1.3 Allgemeinbildung und Informatikunterricht 8
- 1.4 Bezug zum Hessischen Kursstrukturplan 11
- 1.5 Hinweise für den Unterricht 12
- 1.6 Klassifikation und Einsatzmöglichkeiten 16

2 FAKTEN, REGELN UND ANFRAGEN

- 2.1 *fiæ*-Prolog 19
- 2.2 TV-SWI-Prolog 20
- 2.3 Fakten 21
- 2.4 Regeln 22
- 2.5 Rekursive Prädikate 23
- 2.6 Anfragen 24
- 2.7 ProVisor - Visualisierung von Prolog 25
- 2.8 Maschinelles logisches Schließen 29
- 2.9 Aufgaben 32

3 ABLAUFVERFOLGUNG UND VERANSCHAULICHUNG

- 3.1 Trace 35
- 3.2 Das Vierport- oder Boxen-Modell 37
- 3.3 Spuren 38
- 3.4 Aufgaben 40

4 DATENBANKEN

- 4.1 Hotelangebote von Froh-Reisen 41
- 4.2 Aufgaben 43
- 4.3 Hotelbuchung 44
- 4.4 Weitere Aufgaben 45

5 LISTEN

- 5.1 Einführung von Listen 47
- 5.2 Punktschreibweise für Listen 50
- 5.3 Listenoperationen - Von Pascal nach Prolog 52
 - 5.3.1 Fallstudie member 53
 - 5.3.2 Fallstudie append 55
- 5.4 Kopf-Rest-Methode 56
- 5.5 Aufgaben 58

6 ARITHMETIK

- 6.1 Integer-Arithmetik 63
- 6.2 Aufgaben 65

7 EIN- UND AUSGABE

- 7.1 Standard-Prädikate zur Ein- und Ausgabe 67
- 7.2 Anwendungen 68
- 7.3 Nutzung der Systembibliotheken von TV-SWI-Prolog 70
- 7.4 Formatierte Ausgaben 71
- 7.5 Aufgaben 72

8 DER CUT !

- 8.1 So wirkt der Cut 73
- 8.2 Beispiele zum Cut 76
- 8.3 Aufgaben 81

9 UNIFIKATION

- 9.1 Unifikation als Teil des Prolog-Beweisers 83
 - 9.1.1 Familienbeziehungen 83
 - 9.1.2 Listenzerlegung mittels append 85
- 9.2 Definition und Unifikationsregeln 91
- 9.3 Ein Unifikations-Algorithmus 92
- 9.4 Aufgaben 94

10 SYMBOLISCHES DIFFERENZIEREN

- 10.1 Ableitungsregeln 97
- 10.2 Cuts in Ableitungsregeln 99
- 10.3 Potenzfunktionen 100
- 10.4 Kettenregel 101
- 10.5 Allgemeine Funktionen 101
- 10.6 Vereinfachung arithmetischer Ausdrücke 102
- 10.7 Aufgaben 103

11 WISSENSBASIS UND REGELSYSTEME

- 11.1 Hinzufügen und Löschen von Klauseln 105
- 11.2 Einfache Anwendungen 107
- 11.3 Mengenprädikate 109
- 11.4 Herleitung von findall 110
- 11.5 Ein Regelsystem zur Bestimmung von Säugetierarten 111
- 11.6 Aufgaben 113

12 ICE-AUSKUNFTSSYSTEM

- 12.1 Modellbildung 117
- 12.2 Der ICE-Experte 120
- 12.3 Zugbegleiter 121
- 12.4 Abfahrtsplan 123
- 12.5 Zugauskunft 125
- 12.6 Heuristische Suche im ICE-Netz 127
- 12.7 Aufgaben 129

13 AUSKUNFTS- UND REISEBUCHUNGSSYSTEM

- 13.1 Datenbankmodell 131
- 13.2 Benutzungsschnittstelle 133
- 13.3 Gebietsverwaltung 136
- 13.4 Hotelverwaltung 137
- 13.5 Kundenverwaltung 138
- 13.6 Buchungsverwaltung 139
- 13.7 Aufgaben 140

14 SUCHVERFAHREN

- 14.1 Graphen 141
- 14.2 Tiefensuche 143
- 14.3 Breitensuche 144
- 14.4 Heuristische Suche 146
- 14.5 Hüpf-Schiebe-Puzzle 148
- 14.6 Hüpf- und Schiebe-Züge 149
- 14.7 Tiefensuche für das Hüpf-Schiebe-Puzzle 150
- 14.8 Breitensuche für das Hüpf-Schiebe-Puzzle 151
- 14.9 Heuristische Suche für das Hüpf-Schiebe-Puzzle 152
- 14.10 Bewertung der Suchverfahren für das Hüpf-Schiebe-Puzzle 154
- 14.11 Das 8er-Puzzle 156
- 14.12 Beschränkte Tiefensuche für das 8er-Puzzle 158
- 14.13 Breitensuche für das 8er-Puzzle 159
- 14.14 Heuristische Suche für das 8er-Puzzle 160
- 14.15 Bewertung der Suchverfahren für das 8er-Puzzle 161
- 14.16 Aufgaben 162

15 TERME

- 15.1 Klassifikation von Termen 165
- 15.2 Zusammengesetzte Terme 166
- 15.3 Term-Vergleichsoperatoren 168
- 15.4 Strukturoperatoren 169
- 15.5 Beispiele für Termuntersuchungen 171
 - 15.5.1 Zählen von Variablen in Termen 171
 - 15.5.2 Partielle Auswertung arithmetischer Ausdrücke 172
- 15.6 Wurzel-Knoten-Methode 173
- 15.7 Aufgaben 174

16 GRAMMATIKEN UND FORMALE SPRACHEN

- 16.1 Grammatiken 178
 - 16.1.1 Grammatik einfacher deutscher Sätze 178
 - 16.1.2 Arithmetische Ausdrücke 179
 - 16.1.3 Palindrome 180
 - 16.1.4 0-1-Wörter 180
- 16.2 Definition einer Grammatik 181
 - 16.2.1 Grammatik für $a^n b^n c^n$ 181

- 16.3 Chomsky-Hierarchie der Grammatiken 182
- 16.4 Syntaxdiagramme 183
- 16.5 Modellierung von Grammatiken in Prolog 185
- 16.6 Erzeugte Sprache - Breitensuche in Graphen 187
- 16.7 Syntaktische Analyse mit Akzeptoren 190
- 16.8 Aufgaben 192

17 AUTOMATEN

- 17.1 Schaltnetze 193
 - 17.1.1 Logisches Schließen 194
 - 17.1.2 Addieren 195
 - 17.1.3 Prüfung binär codierter Dezimalziffern 196
- 17.2 Speicher 197
- 17.3 Konzeption des endlichen Automaten 198
 - 17.3.1 Getränkeautomat 198
 - 17.3.2 Automatensteuerung eines Aufzugs 199
 - 17.3.3 Akzeptor für Bezeichner 201
 - 17.3.4 Akzeptor für Real-Zahlen 202
- 17.4 Definition des endlichen Automaten 203
- 17.5 Modellierung endlicher Automaten mit Prolog 203
- 17.6 Zeichen und Strings in Prolog 205
- 17.7 Modellierung von Akzeptoren 207
- 17.8 Erzeugte Sprache - ein Graphenproblem 209
- 17.9 Nichtdeterministischer endlicher Automat 211
- 17.10 Automaten mit ϵ -Übergängen 212
- 17.11 Reguläre Ausdrücke 215
- 17.12 Die Grenzen endlicher Automaten 216
- 17.13 Alternative Zugänge 216
- 17.14 Spezialisieren durch Entfalten 217
- 17.15 Syntaxdiagramme und Automaten 220
- 17.16 Aufgaben 222

18 KELLERAUTOMATEN

- 18.1 Konzeption des Kellerautomaten 225
 - 18.1.1 Die Sprache $L_2 = \{a^n b^n \mid n \in \mathbb{N}\}$ 227
 - 18.1.2 Arithmetische Ausdrücke 229
 - 18.1.3 Palindrome 230
- 18.2 Definition des Kellerautomaten 231
- 18.3 Modellierung von Kellerautomaten in Prolog 232
- 18.4 Erzeugte Sprache 234
- 18.5 Direktes Kellern 235
- 18.6 Spezialisieren durch Entfalten 236
- 18.7 Interpretation arithmetischer Ausdrücke 237
- 18.8 Die Grenzen von Kellerautomaten 242
- 18.9 Aufgaben 243

19 TURINGMASCHINEN

- 19.1 Konzeption der Turingmaschine 245
 - 19.1.1 Akzeptor für die Sprache $L_2 = \{a^n b^n \mid n \in \mathbb{N}\}$ 247
 - 19.1.2 Turingmaschine zum Addieren 248
 - 19.1.3 Algorithmische Grundstrukturen 249
- 19.2 Definition der Turingmaschine 250
- 19.3 Modellierung von Turingmaschinen in Prolog 251
- 19.4 Berechenbarkeit und Turingmaschine 255
- 19.5 Grenzen der Turingmaschine 256
- 19.6 Aufgaben 260

20 PARSER UND INTERPRETER

- 20.1 Strichlisten 261
- 20.2 mini-LOGO 263
- 20.3 mini-PASCAL 267
 - 20.3.1 Scanner für mini-PASCAL 269
 - 20.3.2 Parser für mini-PASCAL 270
 - 20.3.3 Interpreter für mini-PASCAL 273
 - 20.3.4 Compiler für mini-PASCAL 275
- 20.4 Aufgaben 280

21 MASCHINELLE SPRACHVERARBEITUNG

- 21.1 Wortproblem - nichtdeterministische Kellerautomaten 281
- 21.2 Ableitungsbäume und Parser 284
- 21.3 Verarbeitung natürlicher Sprache 287
- 21.4 Computerlinguistik im Unterricht 290
- 21.5 ELIZA 293
- 21.6 Aufgaben 297

22 EXPERTENSYSTEME

- 22.1 Architektur eines Expertensystems 299
- 22.2 Wissensdomäne - Bittere Pillen 301
- 22.3 Wissenrepräsentation - Struktur von Fakten und Regeln 306
- 22.4 Fakten über Medikamente 308
- 22.5 Regeln zur Verabreichung von Schmerzmitteln 309
- 22.6 Schnittstelle zu Prolog 312
- 22.7 Ergänzungen zur Wissensrepräsentation 312
- 22.8 Trennung von Wissensbasis und Expertensystemshell 313
- 22.9 Aufgaben 315

23 KOMPONENTEN EINER EXPERTENSYSTEMSHELL

- 23.1 Die Inferenzmaschine 317
- 23.2 Erklärungskomponente - Warum-Fragen 320
- 23.3 Erklärungskomponente - Wie-Fragen 322
- 23.4 Aufgaben 325

A GLOSSAR 327**B LITERATURVERZEICHNIS 331****C ABBILDUNGSVERZEICHNIS 335****D TABELLENVERZEICHNIS 339****E INDEX 341**

1 Prolog in der Schule

1.1 Wozu Prolog in der Schule?

Im Informatikunterricht sollen die Schüler Kenntnisse und Fähigkeiten erwerben, die ihnen die Nutzung von Computern als ein Hilfsmittel zur Problemlösung erlaubt und die Orientierung in einer technisch orientierten Gesellschaft erleichtert. Die Verbindung zwischen Mensch und Maschine wird durch die Programmiersprache hergestellt, mit ihr können Problemlösungen auf den Computer übertragen werden.

Die Frage, ob eine bestimmte Programmiersprache für den Informatikunterricht geeignet ist, ist klassisch und aktuell zugleich. Seit einigen Jahren ist die Tendenz zu beobachten, sich von Pascal, dem Esperanto des imperativen Programmierstils, zu lösen. Mehrere Gründe sind hierbei zu nennen:

- Imperative Programmiersprachen haben eine komplizierte Syntax, sind zu maschinennah und daher fehleranfällig. Informatikunterricht darf kein Programmierkurs sein, aber gerade dazu ermuntert die komplizierte Syntax. komplizierte Syntax
- Die ausschließliche Verwendung des imperativen Programmierstils bindet informatische Inhalte und informatisches Denken zu sehr an dieses Programmierparadigma. Eine Entkopplung informatischen Denkens von der Programmiersprache ist nur durch die Abkehr vom Sprachmonoismus zu erreichen. Sprachmonoismus
- Wesentliche Problembereiche werden erst durch eine deklarative Sprache wie Prolog zugänglich. Wissensverarbeitung, Intellektik und Expertensysteme beziehen sich auf das grundsätzliche Verhältnis von Mensch und Maschine und dürfen daher in einem Informatik-Curriculum nicht fehlen. Auch der Bereich der Sprachverarbeitung - Sprache bildet einen wesentlichen Teil der Mensch-Maschine-Kommunikation - kann durch imperative Sprachen nicht erschlossen werden. breiteres Spektrum von Problemlöseansätzen

Demgegenüber lassen sich einige Vorteile benennen, die der Einsatz von Prolog mit sich bringt. Vorteile von Prolog

Schubert in [Sub1]:

- Informatisches Problemlösen beginnt mit der Analyse des Problems und der Modellierung und Strukturierung des Lösungsplans. Die logische Durchdringung des Problems und der Konzeption einer Problemlösung steht im Vordergrund. Mangelnde Berücksichtigung der Problembedingun- logische Durchdringung des Problems

gen durch zu frühe Konzentration auf Implementierungsfragen können Problemlösungen zum Scheitern bringen. Mit Prolog muß sich der Schüler weniger um die Frage kümmern, wie er ein Problem löst, und kann in stärkerem Maße sich auf die logischen Aspekte des Problems konzentrieren.

Baumann in [Bau 4]:

- komplexe Problemstellungen • Bereits im Anfangsunterricht können mit Prolog komplexe Problemstellungen der Informatik behandelt werden, projektartiges Vorgehen ist möglich, weil wegen der einfachen Syntax ein *Lernen auf Vorrat* (Programmierkurs) nicht erforderlich ist.
- Querverbindungen zu Fächern • Wegen des vorwiegend prädikativen Denkstils und seiner Möglichkeiten zur Wissensrepräsentation lassen sich zahlreiche Querverbindungen zu anderen Schulfächern herstellen.
- bessere Eingangsvoraussetzungen • Die Eingangsvoraussetzungen für Schülerinnen sind wesentlich besser, da Prolog in der Hobbyszene unbekannt ist und der vermeintliche Wissensvorsprung mancher Schüler bezüglich technischer und systemspezifischer Details im Unterricht keine Rolle spielt.

Inwieweit die Meinung der Fachleute auch von Informatiklehrerinnen und Informatiklehrern geteilt wird, kann ansatzweise folgender Auswertung eines Fragebogens entnommen werden. Die Fragen detaillieren und ergänzen die zuvor genannten Vor- und Nachteile. Die Antworten stammen von den Teilnehmern des 5. Weiterbildungskurses Informatik am Standort Darmstadt. Sie wurden am Ende des 3. Semesters im Anschluß an eine Vorlesung über Intellektik (Künstliche Intelligenz) gegeben. Die Antworten auf die ersten beiden Fragen geben Hinweise auf den Kenntnisstand:

Tabelle 1-1
Auswertung eines Fragebogens zu Prolog

Bewertung	++	+	0	-	--
Ich habe die Grundkonzepte von Prolog verstanden.	5	10	1		
Ich fühle mich fit, Prolog im Unterricht einzusetzen.	4	3	5	3	1
Prolog ist mir zu abstrakt	1	1	3	5	5
Prolog ist eine syntaktisch einfache Sprache	4	8	4		
Prolog ist semantisch schwieriger als Pascal.	2	2	5	7	
Da Prolog in der Anwendungsprogrammierung keine Rolle spielt, sollte man auch in der Schule die Finger davon lassen.			2	7	7
Informatische Inhalte lassen sich mit Prolog besser als mit Pascal vermitteln.	1	2	6	4	3

Prolog könnte man schon im Informatikunterricht der SI einsetzen.		9	2	1	3
Prolog verstellt den Blick für Fragestellungen, die die Effizienz von Lösungen betreffen.	2	3	6	5	
Prolog fördert das logische Denken.	3	7	5	1	
Die Sprachstrukturen von Prolog entsprechen eher den Denkstrukturen der Schüler.		3	6	2	3
Prolog reduziert auf das Wesentliche		8	2	5	
Bei der Programmierung in Prolog steht das WAS im Vordergrund.	2	9	2	3	
Prolog erfordert keinen ausführlichen Programmierkurs zu Anfang.	2	9	3	1	1
Man sollte in der Oberstufe hauptsächlich mit Prolog arbeiten.		1	1	10	4
Mit Prolog lassen sich besser Querverbindungen zu anderen Fächer herstellen.	1	3	4	6	2
In Prolog wird unreflektiert mit Rekursion umgegangen.		6	6	4	
Bereits im Anfangsunterricht lassen sich mit Prolog komplexe Problemstellungen bearbeiten.	2	8	3	3	
Da Prolog in der Hobbyszene unbekannt ist, sind die Startbedingungen bei einem Anfangsunterricht mit Prolog für Schülerinnen und Schüler gleichmäßiger.	6	7	1	1	
Ein gleichberechtigter Zugang zur Informatik ist mit Prolog besser erreichbar.	4	4	6	1	
Bildungsziele des Informatikunterrichts lassen sich mit Prolog besser erreichen.		1	3	10	1
Der Effizienzvorsprung prozeduraler Sprachen bei der Anwendungs- und Systemprogrammierung bzw. die Unterlegenheit von Prolog auf diesem Gebiet ist für die Bildungsziele des Informatikunterrichts irrelevant.	1	4	3	7	
Schleifen sind einfacher als Rekursion.	2	8	4	1	
Die Programmierung von Listen ist in Pascal einfacher als in Prolog.	1	3	4	7	1
Problemlösungen in Pascal sind einfacher als Problemlösungen in Prolog.		1	10	4	

Prolog verringert die Gefahr eines Sprachkurses	1	10	4	1	
Prolog eignet sich nicht für die Projektarbeit, weil dafür wichtige Konzepte von der Sprache nicht unterstützt werden. (Information-Hiding, Modularisierung, Schnittstellen)		7	4	4	
Prolog ist wegen der Rekursion eine schwierige Programmiersprache.	1	4	4	6	1
Pascal-Programme sind verständlicher als Prolog-Programme.	1	7	3	5	
Ein Pascal-Programm läßt den Programmablauf besser erkennen als ein Prolog-Programm.	3	6	5	2	
Prolog ist ein Gewinn für den Informatikunterricht	2	11	3		

Bei vielen Fragen gibt es eine weitgehend gleiche Bewertung durch Fachleute und Lehrern. Schauen wir uns an, wo es Meinungsunterschiede gibt:

Die These, daß sich informatische Inhalte mit Prolog besser vermitteln lassen, wird eher abgelehnt. Dies könnte mit unterschiedlichen Auffassungen darüber zusammenhängen, was eigentlich informatische Inhalte sind. Sieht man Inhalte weitgehend durch die Pascal-Brille, so wird man die These wohl eher ablehnen. Bei zunehmender Vertrautheit mit Prolog dürfte sich diese Einschätzung ändern. Man lernt neue Inhalte und Problemlösungsmöglichkeiten kennen, die in analoger Weise in Pascal nicht behandelt werden können.

Daß sich mit Prolog besser Querverbindungen zu anderen Fächern herstellen lassen, wird so nicht gesehen. Hierzu fehlen meines Erachtens auch entsprechende Beispiele. Allerdings meine ich schon, daß mittels Prolog neue Anwendungsgebiete der Informatik, die auch auf andere Fächer ausstrahlen, unterrichtlich erschlossen werden können.

Die Beurteilung der These *Bildungsziele des Informatikunterrichts lassen sich mit Prolog besser erreichen* steht im Kontrast zu den Überlegungen von Frau Lehmann in [Leh1]. Sie geht der Frage nach, welche Aufgaben die allgemeinbildende Schule hat und analysiert dann anhand des Allgemeinbildungsbegriffs von Bussmann/Heymann [Bus1], welchen Beitrag die Informatik und welchen Beitrag Prolog zu Realisierung der Ziele leisten können. Dabei wird deutlich, daß Prolog ein gutes Medium zur Erreichung allgemeinbildender Ziele ist.

Prolog als Medium
zur Erreichung
allgemeinbildender
Ziele

Die große Zustimmung bei der letzten Frage stellt eine Ermunterung für alle Informatiklehrerinnen und -lehrer dar, sich auf diese Programmiersprache einzulassen.

1.2 Prolog zwischen Anspruch und Wirklichkeit

Durch die Auswahl von Prolog als Programmiersprache bei der Entwicklung der 5. Rechnergeneration in Japan im Oktober 1981 hat diese Sprache eine große Bedeutung bekommen. Sie wird insbesondere im Bereich der künstlichen Intelligenz (Intellektik) verwendet. Im Software-Engineering wird Prolog zum Prototyping und zu formaler Spezifikation benutzt.

Prolog kann derzeit als beste deklarative Sprache zur Verwendung im Informatikunterricht angesehen werden, denn es ist eine syntaktisch einfache Sprache mit den der Sprache inhärenten Mechanismen der Unifikation, Resolution und des automatischen Rückverfolgens (Backtracking). Diese Mechanismen machen die Mächtigkeit der Sprache aus. Dazu passend verfügt Prolog von Hause aus über die Datenstruktur *Baum*, aus welcher sich die aus imperativen Programmiersprachen bekannten Datenstrukturen durch Spezialisierung ableiten lassen. Auch die Verfügbarkeit preiswerter Interpreter und schulbezogener Lehrbücher spielen für den Einsatz einer deklarativen Sprache im Unterricht eine nicht zu unterschätzende Rolle.

Verheißungsvoll klingen einführende Worte in einschlägigen Büchern zur Programmiersprache Prolog:

Clocksinn/Mellish in [Clo1]:

Nach unseren Erfahrungen finden Programmieranfänger Prolog-Programme verständlicher als entsprechende Programme in herkömmlichen Sprachen.

mächtige
Sprache durch
Unifikation,
Resolution und
Backtracking

verständliche
Programme

Sterling in [Ste1]:

Deklaratives Programmieren reinigt unser Gehirn, macht den Verstand klar und ermöglicht es, sich auf das Wesentliche des Problems zu konzentrieren, ohne zu sehr in operationellen Details steckenzubleiben. Prolog ist ein Werkzeug für das Denken.

Konzentration auf
das Wesentliche

Bratko in [Bra1]:

Erfahrungen und Ergebnisse für konventionelle Programmiersprachen wie beispielsweise Pascal können sich sogar als hinderlich für die frische Art zu denken, die Prolog erfordert, erweisen.

frische Art zu
denken

Beim unterrichtlichen Einstieg in Prolog wird man vielleicht auch von diesem Optimismus getragen. Die typischen Einstiegsprobleme sind ja auch motivierend und von Schülern ohne große Schwierigkeiten zu meistern. Charakteristisch dafür sind einerseits die Dateiverwaltung, hier als Beispiel eine Schülerdatei mit einer Abfrage, welche die Schülerinnen der Klasse 8b selektiert:

Einstiegsprobleme:
Dateiverwaltung

```
schueler('9a', 'Schmidt', 'Heinz', m).
schueler('8b', 'Maier', 'Christine', w).
schueler('10c', 'Mehlhorn', 'Anna', w).
?- schueler('8b', Name, Vorname, w).
```

andererseits die Verwandtschaftsprogramme, mit einfachen Fakten und Regeln:

Verwandtschafts-
beziehungen

```
weiblich(lea).
weiblich(doris).
maennlich(gerhard).
elternteil(doris, max).
mutter(X, Y):- weiblich(X), elternteil(X,Y).
?- mutter(X, max).
```

Mit der Datenbank- und Regelprogrammierung kann man schon viel vom deklarativen Programmierstil und seiner Einsatzfähigkeit im Bereich der Wissensdarstellung und -verarbeitung erkennen und vermitteln.

Doch leider ist der Optimismus unbegründet. Spätestens mit der Einführung von rekursiven Listenprädikaten ergeben sich massive Schwierigkeiten. Jetzt ist es nämlich nicht mehr so einfach möglich, Problemlösungstechniken auf analoge Fälle zu übertragen. Und was noch schwerer wiegt, auch die bisher erlernte Programmiermethodik ist nicht mehr einsetzbar. In dieser schwierigen Situation wird aus Optimismus leicht Frustration.

fehlende explizite
Kontrollstrukturen

Die Ursachen der Frustration sind leicht auszumachen. Prolog kennt keine expliziten Kontrollstrukturen für die Wiederholung und Fallunterscheidung und keine Wertzuweisung an Variablen. Lediglich Sequenz, Rekursion und Prozeduraufruf sind vorhanden. Das Fehlen der Kontrollstrukturen und der Wertzuweisung raubt dem Schüler die wichtigsten Werkzeuge zur Problemlösung, welche er mit dem imperativen Programmierparadigma kennengelernt hat.

komplizierte
Semantik

Der Vorteil der einfachen Syntax wird durch den Nachteil der komplizierten Semantik eingetauscht. Ein Prolog-Programm macht das dynamische Ablaufverhalten nicht mehr transparent, im Programmtext fehlen entsprechende Strukturen. In imperativen Programmiersprachen geben die Schlüsselwörter für Fallunterscheidungen und Wiederholungen Hinweise auf den Programmablauf.

Sequenz, Selektion und Iteration sind elementare Grundstrukturen des Denkens und Handelns. Wir finden Sie in allen Alltagsalgorithmen wieder, sei dies nun ein Kochrezept oder eine Reparaturanleitung für ein liegengebliebenes Auto. Der Verzicht auf explizite Sprachkonstrukte für diese Grundstrukturen wiegt schwer.

Baumann äußert sich hierzu in [Bau1] wie folgt: „Ungeklärt ist, ob mit Prolog als erster Programmiersprache nicht eine notwendige mentale Entwicklungsphase übersprungen wird. Denn das Denken in Imperativen und sequentiellen Handlungsabläufen scheint etwas Elementares zu sein, das auf der alltäglichen Handlungspraxis basiert und damit vielleicht für den Schüler eine Quelle der Intuition ist, die nicht einfach ausgelassen oder übersprungen werden sollte.“

Als weiterer Grund für Schwierigkeiten mit Prolog können die fehlenden guten Lehr- und Lernmaterialien genannt werden. Lehrbücher auf Hochschulniveau sind als Vorlage für Schulunterricht wenig geeignet. Meist sind es im wesentlichen Textbücher, in denen für die Veranschaulichung wenig getan wird. Schuladäquate Bücher gibt es noch nicht in ausreichender Vielfalt.

Die beiden vorliegenden Materialbände sollen zusammen mit der Visualisierungs-Software ProVisor und dem Prolog-Interpreter TV-SWI-Prolog Lehrern und Schülern einen Einstieg in Prolog erleichtern. Er geht davon aus, daß man Prolog nur dann versteht und damit arbeiten kann, wenn man gute Veranschaulichungen für Datenstrukturen sowie Aufbau und Ablauf von Prolog-Programmen einsetzt und an die Vorkenntnisse der Schülerinnen und Schüler anknüpft. Die Veranschaulichungen werden meistens durch Einsatz von ProVisor vorgenommen. Das Vierportmodell stellt eine weiteren wichtige Veranschaulichungsvariante dar. Für die Fehlersuche sollte dieses Modell zur Verfügung stehen. Bei der Einführung rekursiver Listenprädikate wird deutlich, was der Autor unter Anknüpfung an Vorkenntnisse versteht. An zwei Beispielen wird exemplarisch und detailliert dargestellt, wie man vom prozeduralen Stil geprägt zu echten Prolog-Lösungen kommen kann. Über diese Brücke ist ein einfacher Weg von Pascal nach Prolog möglich.

Grundstrukturen des Denkens und Handelns

Denken in Imperativen und sequentiellen Handlungsabläufen

ProVisor und TV-SWI-Prolog

Die Brücke von Pascal nach Prolog

1.3 Allgemeinbildung und Informatikunterricht

Dem Kursstrukturplan Informatik liegt Klafkis Allgemeinbildungsbegriff [Kla1] zugrunde. Er konstituiert sich aus drei Bedeutungsmomenten:

- Allgemeinbildung ist Bildung für alle zur Selbstbestimmungs-, Mitbestimmungs- und Solidaritätsfähigkeit.
- Allgemeinbildung ist kritische Auseinandersetzung mit den allgemeinen Fragen und Problemen, die uns angehen.
- Allgemeinbildung ist Bildung aller humaner Fähigkeitsdimensionen des Menschen.

Klafkis Kernthese lautet:

„Allgemeinbildung bedeutet ein geschichtlich vermitteltes Bewußtsein von zentralen Problemen der Gegenwart und - soweit voraussehbar - der Zukunft zu gewinnen, Einsicht in die Mitverantwortlichkeit aller angesichts solcher Probleme und Bereitschaft, an ihrer Bewältigung mitzuwirken. Abkürzend kann man von einer Konzentration auf epochaltypische Schlüsselprobleme (z.B. Friedens- und Umweltfrage, gesellschaftlich produzierte Ungleichheit, Ich-Du-Beziehung (Liebe, Sexualität)) unserer Gegenwart und Zukunft sprechen.“

epochaltypische
Schlüsselprobleme

Ein epochaltypisches Schlüsselproblem sind für ihn Gefahren und Möglichkeiten der neuen Technologien, weswegen Klafki auch eine IKG fordert:

IKG

„Wir brauchen in einem zukunftsorientierten Bildungssystem auf allen Schulstufen und in allen Schulformen eine gestufte, kritische informations- und kommunikationstechnologische Grundbildung als Moment einer neuen Allgemeinbildung; "kritisch", das heißt so, daß die Einführung in die Nutzung und in ein elementarisertes Verständnis der modernen, elektronisch arbeitenden Kommunikations-, Informations- und Steuerungsmedien immer mit der Reflexion über ihre Wirkungen auf die sie benutzenden Menschen, über die möglichen sozialen Folgen des Einsatzes solcher Medien und über den möglichen Mißbrauch verbunden werden.“

Die GI-Empfehlungen für das Fach Informatik in der Sekundarstufe II allgemeinbildender Schulen [GI1] gehen von Klafkis Allgemeinbildungsbegriff aus und versuchen, den spezifischen Beitrag des Informatikunterrichts zur Allgemeinbildung herauszuarbeiten. Dabei wird festgestellt, daß der Informatikunterricht bisher von der traditionellen Auffassung der Informatik als Strukturwissenschaft geprägt ist. Demgemäß stellt der Algorithmusbegriff die Klammer zwischen technischen, theoretischen und praktischen Themenstellungen dar und ermöglicht deren Behandlung unter einem gemeinsamen Blickwinkel.

Angesichts der Entwicklung der Informatik und der Verbreitung der Informationstechnik müssen heute jedoch ein breiteres Spektrum von Problemlöseansätzen und vielfältige Anwendungsaspekte weit stärker als bisher betrachtet werden. Die GI-Empfehlungen nennen insbesondere: Paradigmenwechsel im Programmiersprachenbereich, heuristische Strategien, etwa im Zusammenhang mit wissensbasierten Systemen, Wissensrepräsentation und -verarbeitung, Techniken der Parallelverarbeitung, Behandlung der Informations- und Kommunikationstechnologien unter dem Blickwinkel der Sozialverträglichkeit in gesellschaftlichen Anwendungsfeldern

breiteres Spektrum von Problemlöseansätzen

Um die Entwicklungen in der Informatik aufzunehmen und dem obigen Allgemeinbildungsbegriff zu genügen, werden neue *Sichtweisen* für den Informatikunterricht benötigt. Als ordnendes Prinzip legen die GI-Empfehlungen den *Bezug des Menschen zum Computer* zugrunde. Damit hat der Algorithmus seine Pflicht und Schuldigkeit getan, wir verabschieden uns vom algorithmenorientierten Unterricht.

Bezug des Menschen zum Computer

Aus dem ordnenden Prinzip werden drei Sichtweisen abgeleitet

- Wechselwirkung Mensch-Computer
- Formalisierung und Automatisierung geistiger Arbeit
- Informatiksysteme, Gesellschaft und Umwelt

neue Sichtweisen für den Informatikunterricht

welche später zur Strukturierung der Ziele und Inhalte des Informatikunterrichts benutzt werden. Bezogen auf die neuen Sichtweisen wird der Beitrag des Informatikunterrichts zum Bildungs- und Erziehungsauftrag der Schule in folgenden Punkten gesehen:

- Förderung eines verantwortungsbewußten Umgangs mit Informationen und Erziehung zu verantwortlichem Handeln
- Reflexion des Verhältnisses von Menschen zur Informationstechnik
- Förderung eines gleichberechtigten Zugangs zur Technik
- Vermittlung verschiedener Problemlösungs- und Gestaltungsmethoden und deren Beurteilung
- Förderung des schöpferischen Denkens
- Förderung der Fähigkeit zu Kommunikation und Kooperation

Beiträge des Informatikunterrichts zum Bildungs- und Erziehungsauftrag der Schule

In Ergänzung zum grundsätzlichen Inhalt sind in den GI-Empfehlungen vier Unterrichtsprojekte skizziert, welche nicht algorithmenorientiert sind, sondern gemäß den Intentionen der Empfehlungen neue Sichtweisen und Problemlösungsansätze anhand konkreter Anwendungen illustrieren:

- Einsatz der Informationstechnik in der Textilindustrie
- Informationstechnologie im Warenhaus Europa
- Maschinelle Sprachverarbeitung
- Logische Programmierung

Zwei dieser Unterrichtseinheiten lassen sich letztlich nur unter Einsatz von Prolog realisieren:

Unterrichtsprojekt *Maschinelle Sprachverarbeitung*

- | | |
|--|--|
| Unterrichtsprojekt
Maschinelle
Sprach-
verarbeitung | <ul style="list-style-type: none"> - Anwendung: automatische Übersetzung und Dialogsysteme - Unterscheidung: geschriebene und gesprochene Sprache - Unterrichtseinstiege: <ul style="list-style-type: none"> - Zeitungsartikel über Mißerfolge der maschinellen Übersetzung, Probleme mit Kontext, - Benutzung, Analyse und Modifikation eines Programms, das allein mit der Technik des Musterabgleichs auf eingegebene Sätze mit scheinbar sinnvollen Antworten reagiert (ELIZA) - Einführung der linguistischen Begriffe Syntax, Semantik und Pragmatik - Techniken syntaktischer Sprachverarbeitung: formale Grammatik, Ersetzungsregeln in Backus-Naur-Form, Einführung syntaktische Kategorien wie <Satz>, <Nominalteil>, <Verbalteil>, <Substantiv>, <Verb>,... - Programmierprojekt: Parser für gegebene Grammatik, Erweiterung der Grammatik, Ausgabe von Strukturbäumen |
|--|--|

Unterrichtsprojekt *Logische Programmierung*

- | | |
|--|--|
| Unterrichtsprojekt
Logische
Programmierung | <ul style="list-style-type: none"> - aufstellen von Relationen zwischen Objekten des Problemraumes - Konzentration auf Beschreibung des Problems, Abläufe, deren Konstruktion und Steuerung sind weniger relevant - erkennen, daß das Prolog-System Aussagen nicht überprüfen kann, sondern kritiklos verarbeitet - Diskussion von Modellen: Vierportmodell, Ableitungsbaum, UND-ODER-Beweisbaum - Verbindung zur Theorie: Terminierung bei Linksrekursion, Heuristik durch Reihenfolge der Ziele |
|--|--|

Mit dem in diesem Heft vorliegenden Material können Sie problemlos beide Unterrichtsprojekte durchführen. Insbesondere gibt es spezielle Kapitel zu Grammatiken und zur Maschinellen Sprachverarbeitung.

Besonderer Wert wird auf die Veranschaulichung mittels Modellen gelegt. Das Vierportmodell wird nicht nur zur Veranschaulichung des Traceslaufs sondern auch zur Veranschaulichung des Aufbaus und Ablaufs von Prolog-Programmen eingesetzt. Die Wirkungsweise des Cuts wird im Vierportmodell neu gedeutet. Mit der didaktischen Software ProVisor steht ein alternatives Veranschaulichungsmodell zur Verfügung, das alle Daten- und Kontrollstrukturen, sowie Unifikation, Resolution und Backtracking visualisieren kann.

1.4 Bezug zum Hessischen Kursstrukturplan

Die im neuen Hessischen Kurstrukturplan genannten Aufgaben und Ziele des Informatikunterrichts sind sowohl von Klafkis Allgemeinbildungsbegriff als auch von den GI-Empfehlungen beeinflusst. Die Orientierung am ordnenden Prinzip *Bezug des Menschen zum Computer* und den drei genannten Sichtweisen läßt sich an den vom Kursstrukturplan genannten *zentralen Aufgaben des Schulfachs Informatik* leicht erkennen:

1. Die Persönlichkeitsentwicklung des Einzelnen durch Förderung seiner Urteils- und Handlungsfähigkeit und seines verantwortungsbewußten Umgangs mit Information und Technik
2. Vermittlung von theoretischen und technischen Grundlagen der Informations- und Kommunikationstechniken und ihrem Beitrag zur Entwicklung von Kultur und Wissenschaft
3. Sensibilisierung für die Probleme bei der Entwicklung zu einer Informationsgesellschaft als gesellschaftliches Schlüsselproblem, das uns alle angeht

Aufgaben und Ziele des Informatikunterrichts

Aus diesen Aufgaben ergeben sich die folgenden drei *Inhaltsbereiche* des Kursstrukturplans:

Inhaltsbereiche des Kursstrukturplans

Im Inhaltsbereich *Werkzeuge und Methoden zum Problemlösen* geht es exemplarisch um Methoden zur Analyse und Beschreibung von realen Sachzusammenhängen, deren Modellierung und die dabei meist notwendige Reduktion der Wirklichkeit. Der vorliegende Materialband stellt dazu die notwendigen Kenntnisse und Werkzeuge für den Einsatz von Prolog bereit und thematisiert in diesem Zusammenhang die Modellierung von Wissensbereichen mittels Fakten und Regeln (Familienstammbaum, Symbolisches Differenzieren), Datenbanken (ICE-Auskunftssystem, Reisebuchungssystem), Suchverfahren auf Graphen und Expertensysteme. Die inhaltlichen Voraussetzungen für die im Leistungskurs erforderliche weitere Programmiersprache sind durch das Material abgedeckt.

Im Inhaltsbereich *Universelle symbolverarbeitende Maschine - Mensch und Maschine* stehen die theoretischen und technischen Grundlagen der maschinellen Informationsverarbeitung sowie ihre prinzipiellen Möglichkeiten und Grenzen im Vordergrund der Betrachtung. In den Kapiteln über Grammatiken, Automaten und Kellerautomaten werden die grundlegenden Begriffe anhand vieler Beispiele eingeführt, Maschinenmodelle entworfen und die maschinelle Verarbeitung mit Prolog demonstriert. Anwendungen werden in den Kapiteln über *Parser und Interpreter* und *Expertensysteme* gezeigt. Die zugehörigen Aufgaben geben Hinweise auf weitere Anwendungen.

Im Inhaltsbereich *Informations- und kommunikationstechnische Systeme in Gesellschaft und Umwelt* geht es um die Auseinandersetzung mit modernen Informationssystemen auf der Grundlage von Datenbanken. Dazu werden in diesen Materialien zwei ausbaufähige Beispiele gebracht: ein ICE-Auskunfts- und eine Reisebuchungssystem. Weitere Beispiele finden Sie in [Bau7], wo es um die Prüfungsfachwahl unter Berücksichtigung der länderspezifischen Vorschriften geht und in [Bau6] mit einem in Prolog realisiertem Bibliothekssystem.

1.5 Hinweise für den Unterricht

Wesentliche Teile des Materialbandes sind als Begleit- und Übungsmaterial zur einer Vorlesung über Intellektik (Künstliche Intelligenz) am Standort Darmstadt der Lehrerweiterbildung Informatik entstanden. Nachträglich wurden die Kapitel 12 bis 14 und 16 bis 21 ergänzt, um insbesondere Material zur Gestaltung des Inhaltsbereichs *Universelle symbolverarbeitende Maschine - Mensch und Maschine* mit Prolog bereitzustellen. Andere Kapitel wurden überarbeitet, um die Bezüge zur Vorlesung aufzuheben.

Einführung in
Prolog

Kapitel 2 eignet sich zur Einführung in Prolog. Es kann im Rahmen des Wahlpflichtunterrichts Informatik in den Klassen 9 und 10 eingesetzt, aber auch zur Einführung in den Jahrgangsstufen 11 bis 13 benutzt werden.

das Spur-Modell
als didaktische
Reduktion

Auf Kapitel 3 sollte man zurückgreifen, wenn Fehlersuche mittels Trace benötigt wird. Das Vierportmodell liefert hierfür das notwendige gedankliche Modell. Das Spur-Modell stellt eine didaktische Reduktion des Vierportmodells dar, bei dem weniger detailliert dafür übersichtlicher die Abarbeitung einer Anfrage verfolgt werden kann.

Datenbank-
konzepte mit
Prolog

Kapitel 4 bringt einfache Datenbankkonzepte mit Prolog und ist aufgrund des geringen Anforderungsniveaus wie Kapitel 1 für die Einführung in Prolog geeignet. Das vorgestellte Auskunfts- und Reisebuchungssystem wird in Kapitel 13 mit einer Benutzungsschnittstelle und Verwaltungsprogrammen ergänzt.

Als Alternative oder Ergänzung steht das ICE-Auskunftssystem aus Kapitel 12 zur Verfügung, wobei man sich zunächst auf Implementierung der Datenbank und interaktive Abfragen beschränken wird.

Anspruchsvolle Problemlösungen mit Prolog kommen ohne Listen nicht aus. Die Einführung der Listen in Kapitel 5 orientiert sich nicht an der naiven Verwendung von Listen, sondern an der generellen Konzeption von Datenstrukturen in Prolog. Die Veranschaulichung der Listen zeigt die zugrundeliegende Baumstruktur und verweist auf die Punktnotation für Listen. Sofern auf Strukturuntersuchungen wie in Kapitel 15 und darauf aufbauenden Programmier-techniken im weiteren Verlauf des Unterrichts nicht eingegangen werden soll, kann auf die Thematisierung der Punktnotation verzichtet werden. Die Unterkapitel 5.3 *Listenoperationen - von Pascal nach Prolog* und 5.4 *Kopf-Rest-Methode* bilden zentrale Bausteine dieses Materialbandes, da sie Basistechniken für die Arbeit mit Prolog darstellen. Dementsprechend werden im Aufgabenteil viele Übungsmöglichkeiten geboten. Mit den Inhalten aus den Kapitel 2 bis 5 hat man sich ein sicheres Fundament für die Arbeit mit Prolog geschaffen. Vielfältige Themenbereiche können auf der Basis dieses Fundaments unterrichtlich behandelt werden.

Kapitel 6 stellt arithmetische Aspekte von Prolog im Überblick dar. Die Zusammenstellung der diversen Operatoren ist nützlich, ansonsten spielt die Arithmetik in Prolog eine untergeordnete Rolle.

In Kapitel 7 werden grundlegende Ein- und Ausgabeprädikate vorgestellt, welche für die programmgesteuerte Ein- und Ausgabe genutzt werden können. Zudem wird thematisiert, wie man Systembibliotheken nutzen kann.

Der Cut ist Thema von Kapitel 8. Die Wirkungsweise des Cut wird erklärt und in den verschiedenen Veranschaulichungsmodellen erläutert. Nach der Darstellung psychologischer Schwierigkeiten im Zusammenhang mit dem Cut wird an vielen Beispielen der Einsatz des Cut demonstriert.

Kapitel 9 befaßt sich ausführlich mit der Unifikation als einem wesentlichen Verfahren, auf denen die Prolog-Maschine aufbaut. Das Resolutions- und das Backtracking-Verfahren werden implizit in Kapitel 2 thematisiert. Diesem Kapitel liegt die Überzeugung zugrunde, daß nur das klare Verständnis der Verfahren, auf denen Prolog basiert, zum sicheren Umgang mit Prolog befähigt.

Kapitel 10 ist dem *Symbolischen Differenzieren* gewidmet. Es zeigt an einem überzeugenden Beispiel den deklarativen Charakter von Prolog, das Zusammenwirken von Resolution und Unifikation bei der Lösungsfindung und die sich daraus ergebende maschinelle, künstliche Intelligenz.

Einführung von
Listen

Basistechniken für
die Arbeit mit
Prolog

Arithmetik

Ein- und Ausgabe

der Cut !

Unifikation

maschinelle künst-
liche Intelligenz

Manipulation der Wissensbasis das Mengenprädikat <i>findall</i>	<p>In Kapitel 11 werden Prädikate vorgestellt, mit denen zur Laufzeit Fakten und Regeln ergänzt, abgefragt beziehungsweise gelöscht werden können. Mit diesen Prädikaten können lernfähige wissensbasierte Systeme realisiert werden. Benötigt man alle Lösungen eines Teilziels, um sie beispielsweise sortiert auszugeben oder als Liste weiterverarbeiten zu können, so verwendet man dazu das <i>findall</i>-Prädikat. Es wird vorgestellt und eine einfache Implementierung erläutert. <i>findall</i> wird intensiv in Kapitel 14 eingesetzt, wenn es um die Breitensuche und heuristische Suche geht. Abschließend wird in Kapitel 12 ein einfaches Regelsystem zur Bestimmung von Säugetierarten vorgestellt, welches im Aufgabenteil um eine Regelsystem zur Empfehlung von Arzneimitteln ergänzt wird. Diese Regelsysteme können als vereinfachte Modelle für Expertensysteme betrachtet werden und in dieser Funktion Kapitel 22 vorbereiten.</p>
Informationssysteme	<p>Ein ICE-Auskunftssystem ist Inhalt von Kapitel 12, Kapitel 13 bringt auf vergleichbarem Niveau ein Auskunfts- und Reisebuchungssystem. Zunächst wird eine datenbanktechnische Modellierung auf der Basis von Entity-Relationship-Diagrammen durchgeführt, auf deren Basis einfache interaktive Abfragen möglich sind. Anschließend werden für Standardabfragen Algorithmen entworfen beziehungsweise eine Benutzungsschnittstelle realisiert. Die beiden Systeme sind Beispiele für mögliche Themenstellungen im Kurs 12/II über Informations- und Dateiverwaltungssysteme.</p>
Suchverfahren	<p>Kapitel 14 setzt sich mit typischen Suchverfahren auseinander. Sie werden am Beispiel eines bewerteten Graphen eingeführt und später auf Zustandsgraphen angewendet. Dabei wird deutlich, wie Probleme mit klar definierten Problemzuständen und Zustandsübergängen sich in einheitlicher Weise durch die vorgestellten Suchverfahren lösen lassen.</p>
fortgeschrittene Programmier-techniken	<p>Kapitel 5 brachte die Basis-Programmiertechniken in Prolog, welche für die Arbeit mit linearen Strukturen benötigt werden. Kapitel 15 befaßt sich mit fortgeschrittenen Programmiertechniken für die Arbeit mit verzweigten Baumstrukturen. Vermutlich ist Ihnen von Pascal die recht unterschiedliche algorithmische Verarbeitung von Listen und Binärbäumen bekannt. Operationen auf Listen können mit iterativen Methoden implementiert werden, für Bäume benötigt man rekursive Verfahren. Eine vergleichbare Situation liegt in Prolog vor: zur Listenbearbeitung benötigt man den Listenoperator „ “ in Verbindung mit endrekursiven (= iterativen) Klauseln, zur Bearbeitung zusammengesetzter Terme den univ-Operator „=..“ in Kombination mit der Wurzel-Knoten-Methode. Die Verwendung des <i>univ</i>-Operators stellt eine kleine Hürde dar, bringt allerdings wegen der damit vorhandenen Verfügbarkeit von Bäumen erhebliche Vorteile mit sich. Mit dem Verzicht auf den <i>univ</i>-Operator verzichtet man gleichfalls auf die Nutzung dynamischer Baumstrukturen und muß sich mit Listen begnügen. Die fortgeschrittenen Programmiertechniken dieses</p>
univ-Operator =.. und Wurzel-Knoten-Methode	

Kapitels werden beispielsweise zur Konstruktion von Parsebäumen in den Kapiteln *Parser und Interpreter* und *Maschinelle Sprachverarbeitung* und zur Analyse von Termbäumen im Kapitel *Symbolisches Differenzieren* benutzt.

Mit Kapitel 16 beginnt der zweite Band, der sich hauptsächlich mit dem Inhaltsbereich *Universelle symbolverarbeitende Maschine - Mensch und Maschine* widmet. Zunächst werden Grammatiken und formale Sprachen eingeführt und der Zusammenhang mit Syntaxdiagrammen und Kontrollstrukturen aufgezeigt. Die formalen Sprachen dienen später der Charakterisierung unterschiedlicher Automatenmodelle, die Syntaxdiagramme der anschaulichen Beschreibung formaler Sprachen.

Grammatiken und formale Sprachen

Das umfangreiche Kapitel 17 setzt sich mit den endlichen Automaten auseinander. Automaten werden anhand mehrerer Beispiele eingeführt, definiert, in Prolog modelliert und zur Generierung formaler Sprachen eingesetzt. Der Bezug zu den regulären Ausdrücken wird hergestellt, womit dann die Grenzen der Automaten deutlich werden. Abschließend geht es um Techniken, mit denen effizient Automaten für spezifische Anwendungen konstruiert werden können.

endliche Automaten und reguläre Ausdrücke

In Kapitel 18 geht es um die Kellerautomaten, welche als Automaten mit unbegrenztem Speicher und eingeschränktem Speicherzugriff anzusehen sind. Am Beispiel der Erkennung und Auswertung arithmetische Ausdrücke wird die Leistungsfähigkeit von Kellerautomaten verdeutlicht.

Kellerautomaten

Mit den Turingmaschinen, den leistungsfähigsten Automaten, setzt sich Kapitel 19 auseinander. Die verwendeten Beispiele lassen erkennen, dass Turingmaschinen einerseits so leistungsfähig wie heutige und zukünftige Computer sind (Churchsche These), andererseits aber auch den Turingmaschinen Grenzen gesetzt sind (Halteproblem).

Turingmaschine, Churchsche These und Halteproblem

Interessante Anwendungen der theoretischen Konzepte werden in den Kapiteln 20 und 21 gebracht. Schwerpunkt sind Parser und Interpreter für graphische Befehle der Programmiersprache LOGO, Interpreter und Übersetzer für mini-PASCAL und die maschinellen Sprachverarbeitung.

Parser und Interpreter und maschinelle Sprachverarbeitung

In den beiden abschließenden Kapiteln 22 und 23 findet eine vertiefte Auseinandersetzung mit Expertensystemen statt. Eine Expertensystem-Shell wird mit einer Wissensdomäne aus Fakten und Regeln zum Bereich Schmerzmittel versehen. Das so gewonnene Expertensystem arbeitet als Schmerzmittelberater. Die Analyse der Inferenzmaschine und Erklärungskomponente für Wie- und Warum-Fragen gibt Einblick in die Architektur und Arbeitsweise eines Expertensystems auf einem anspruchsvollen Niveau.

Expertensysteme

1.6 Klassifikation und Einsatzmöglichkeiten

Der Umfang der zwei Bände macht es nötig, die Kapitel zu klassifizieren und Einsatzmöglichkeiten aufzuzeigen, um verschiedene unterrichtliche Intentionen verwirklichen zu können.

Klassifikation

Die Klassifikation findet hinsichtlich des Anspruchsniveaus und der Sprachelemente statt. Beim Anspruchsniveau wird zwischen gering, mittel und hoch unterschieden. Kapitel, in denen neue Sprachelemente eingeführt werden, werden mit dem Buchstaben N gekennzeichnet. Da sich die Materialbände nicht als Sprachlehrbuch verstehen, gibt es weder eine systematische Einführung in die Programmiersprache Prolog, noch eine vollständige Beschreibung der Sprachelemente. Der Leser sei hierzu auf die zahlreichen Sprachlehrbücher und das Manual von TV-SWI-Prolog verwiesen.

Einsatzmöglichkeiten

Die ersten Kapitel dienen der Einführung in das deklarative Programmieren mit Prolog auf elementarer Ebene. Dieser Einführungskurs kann problemlos schon im Wahlpflichtunterricht Informatik der Sekundarstufe I aber auch in der Jahrgangsstufe 11 gehalten werden, wobei die Schülerinnen und Schüler am Beispiel der Familienbeziehungen den Computer als logisch schließende Maschine kennen lernen.

Für einen Kurs *Algorithmen und Datenstrukturen*, beispielsweise in der Jahrgangsstufe 12/I, geben die Kapitel aus dem ersten Band reichlich Material. Schwerpunkte bilden die Standardalgorithmen auf Listen in Kapitel 5 und Termen in Kapitel 15, sowie die Suchverfahren in Kapitel 14.

Einen Kurs über *Informations- und Dateiverwaltungssysteme* in der Jahrgangsstufe 12/II kann man auf der Grundlage des *ICE-Auskunfts-* und des *Auskunfts- und -Reisebuchungssystems* gestalten. Die Systeme können problemlos erweitert werden und eignen sich für Projektarbeit.

Der zweite Band gibt reichhaltig Material zu einem Kurs *Grundlagen der Theoretischen Informatik* in der Jahrgangsstufe 13/I mit den Schwerpunkten Automaten, Formale Sprachen und Interpreter. Der Umfang des gebotenen Materials macht eine Auswahl von Themen insbesondere bei den Automaten nötig.

In der Jahrgangsstufe 13/II kann man als Wahlthema eine Kurs *Künstliche Intelligenz* anbieten, den man zwecks Vertiefung auf spezielle Teilaspekte, wie zum Beispiel *Expertensysteme* oder *Maschinelle Sprachverarbeitung*, einschränken kann. Grundsätzlich können Fragestellungen der Künstlichen Intelligenz auch in früheren Jahrgangsstufen behandelt werden. Es kommt dann darauf an, daß man das geeignete Abstraktionsniveau findet, auf dem man die Inhalte darstellt und bearbeitet.

Nr	Kapitel	Anspruchsniveau	neue Sprachelemente	Einführungskurs	Algorithmen und Datenstrukturen	Informations- und Dateiverwaltungssysteme	Grundlagen der Theoretischen Informatik	Künstliche Intelligenz	Expertensysteme	Maschinelle Sprachverarbeitung
1	Prolog in der Schule	-								
2	Fakten, Regeln und Anfrag.	1	N	S	S	S	S	S	S	S
3	Ablaufverfolgung und Ver.	2		E	S	S	S	S	S	S
4	Datenbanken	1		S	E	S				
5	Listen	2	N	E	S	S	S	S	S	S
6	Arithmetik	1	N		S	S				
7	Ein- und Ausgabe	2	N		E	S	E	S	S	S
8	Der Cut!	2	N		S		E	S	S	S
9	Unifikation	2	N		S		E	S	S	S
10	Symbolisches Differenzie.	2			S			S		
11	Wissensbasis und Regelsys.	2	N		S	S		S		
12	ICE-Auskunftssystem	2				S				
13	Auskunfts- und Buchungsys.	2				S				
14	Suchverfahren	2			S			S		
15	Terme	3	N		S		E	S	S	
16	Grammatiken	2					S	S		S
17	Automaten	2					S			
18	Kellerautomaten	2					S			
19	Turingmaschinen	2					E			
20	Parser und Interpreter	2-3					S			
21	Maschinelle Sprachverarbei.	2-3						S		S
22	Expertensysteme	3							S	
23	Komponenten einer E.-Shell	3							S	

 Tabelle 1-2
 Klassifikation und
 Einsatzmöglich-
 keiten der Kapitel

Anspruchsniveau

1 = gering

2 = mittel

3 = hoch

Sprachelemente

N = Neu

Einsatzmöglichkeit

S = Stammkapitel

E = Erweiterung

Die Kapitel müssen nicht unbedingt in der Reihenfolge, wie sie in den beiden Bänden beziehungsweise in den Einsatzmöglichkeiten auftreten im Unterricht behandelt werden. Auch ist es nicht erforderlich, Kapitel komplett durchzuarbeiten. Beispielsweise kann man elementare Kenntnisse über Terme, wie sie in den Unterkapiteln 15.1 und 15.2 dargestellt werden, schon relativ früh behandeln und die viel anspruchsvolleren Unterkapitel 15.3 und 15.4 auslassen. Ähnliches gilt für Kapitel 11 über *Wissensbasis und Regelsysteme*. Hier kann man die Unterkapitel 11.1 *Hinzufügen und Löschen von Klauseln* und 11.3 *Mengenprädikate* nach Bedarf einsetzen, ohne den Zusammenhang zu verlieren. Insbesondere lassen sich auch die kurzen Kapitel über *Arithmetik* sowie *Ein- und Ausgabe* integriert in andere Fragestellungen darstellen.

2 Fakten, Regeln und Anfragen

2.1 fiæ-Prolog

Die Entwicklungsumgebung von *fiæ-Prolog* [Knü1] basiert auf dem Turbo-Vision-System von Turbo-Pascal. Dies hat den Vorteil, daß Schülerinnen und Schüler sofort in dieser Umgebung arbeiten können. Der Bildschirm enthält neben den obligatorischen Menü- und Statuszeilen ein Input- und ein Output-Fenster. Mit der Funktionstaste F6 kann man zum nächsten Fenster wechseln, alternativ läßt sich mit Alt-<Fensternummer> ein nummeriertes Fenster aktivieren.

Turbo-Vision

Im Input-Fenster befindet sich das Anfrage-Symbol „?-“; als Kennzeichen dafür, daß hier lediglich Anfragen formuliert werden dürfen. Fakten und Regeln können über das Input-Fenster nicht direkt eingegeben werden, sie müssen konsultiert werden. Der Editor des Input-Fensters ist anachronistisch. Auch in der neuen Version 3.1 von *fiæ-Prolog* steht lediglich die *Rücktaste* für Korrekturen zur Verfügung. Ärgern Sie sich also ruhig, wenn die *Pfeil-Links-Taste* den Cursor um ein Zeichen nach rechts bewegt.

Anfrage-Symbol ?-
im Input-Fenster

Der Hersteller empfiehlt, Anfragen in das Clipboard zu schreiben, da sie dort wie üblich editiert werden können. Man markiert die Anfrage im Clipboard und übernimmt sie mittels *Clipboard/paste* beziehungsweise *Shift-Ins* in das Input-Fenster.

Clipboard

Ausgaben erscheinen im Output-Fenster, in das Sie stets mit Alt-2 wechseln können. Die Kapazität des Output-Fensters umfaßt einige Bildschirmseiten. Die Ausgaben können gelesen, aber nicht editiert werden.

Output-Fenster
mit Alt-2

Zur Bearbeitung von Anfragen benötigt der Prolog-Interpreter eine Wissensbasis aus Fakten und Regeln. Man schreibt Fakten und Regeln mit einem Editor auf und speichert sie in einer Datei. Die Anfrage *?- consult(Dateiname)* bewirkt, daß der Prolog-Interpreter die Datei liest und Fakten und Regeln in seiner Wissensbasis ablegt. Über den Menüpunkt *File/consult* kann man eine Datei bequem auswählen und konsultieren. Haben Sie nach *File/open* oder *File/new* einen Prolog-Quelltext im Editor, so können Sie diesen direkt über *Edit/consult* konsultieren.

Wissensbasis aus
Fakten und Regeln
mit *consult* bzw.
reconsult laden

Im File- und Edit-Menü gibt es den Menüpunkt *reconsult* zur nachträglichen Korrektur der Wissensbasis. Einblick in die interne Wissensbasis erhalten Sie über das *Database*-Menü.

2.2 TV-SWI-Prolog

fiæ-Prolog ist ein Prolog-System, das für Ausbildungszwecke konzipiert wurde. Effizienz spielte bei der Entwicklung keine Rolle, der Sprachumfang wurde auf den Edinburg-Standard beschränkt. Bei etwas umfangreicheren Daten oder Programmen kann man recht schnell die Grenzen von *fiæ*-Prolog erreichen.

mächtiges
und schnelles
SWI-Prolog
ab 386er

Demgegenüber ist SWI-Prolog als ein Prolog-System konzipiert, das mächtig und schnell ist, einen umfangreichen Sprachschatz und eine Schnittstelle zur Programmiersprache C aufweist, in C und Prolog geschrieben ist und daher für Experimente mit logischer Programmierung genutzt werden kann. Von dem auf einer SUN-Workstation entwickelten Prolog-System gibt es mittlerweile auch eine Portierung für MS-DOS-Maschinen ab 386er aufwärts.

SWI-Prolog mit
Turbo-Vision

Angesichts der Schwächen von *fiæ*-Prolog und der Stärken von SWI-Prolog hat der Autor die Entwicklung einer Turbo-Vision-Oberfläche für SWI-Prolog beauftragt. Das so entstandene TV-SWI-Prolog vereinigt die Vorteile von SWI-Prolog mit den Vorteilen von Turbo-Vision und ProVisor und stellt damit die zur Zeit beste Wahl für einen schülergerechten Prolog-Interpreter dar. Im Unterschied zu *fiæ*-Prolog gibt es anstelle getrennter Input- und Output-Fenster ein gemeinsames Fenster.

TV-SWI-Prolog sollten Sie mit einer Batchdatei starten, welche einige benötigte Umgebungsvariablen setzt:

Batchdatei
TVSWIPL.BAT
zum Starten von
TV-SWI-Prolog

```
CD \TVSWIPL
SET GO32=EMU .\EMU387
SET GO32TEMP=C:/TEMP
SET SWI_HOME_DIR=.
TVSWIPL
```

Mit `CD \TVSWIPL` wird in das Systemverzeichnis gewechselt. Der `SET GO32=EMU .\EMU387` Befehl wird nur bei 386er-Systemen ohne Koprozessor benötigt, um Real-Berechnungen zu emulieren. Die Umgebungsvariable `GO32TEMP` legt fest, wohin Dateien ausgelagert werden können. Dies sollte eine Festplatte sein. `SWI_HOME_DIR` ist unbedingt nötig, damit TV-SWI-Prolog seine Systemdateien findet.

Die Standard-Dateierweiterung lautet im Unterschied zu *fiæ*-Prolog „PL“. Der Dateiauswahldialog von TV-SWI-Prolog bietet alle „P??“-Dateien an, um sowohl *fiæ*-Prolog als auch TV-SWI-Prolog Dateien auswählen zu können.

Achten Sie bitte darauf, daß TV-SWI-Prolog zwischen Prädikatsnamen und öffnender Klammer keine Leerzeichen duldet. Vor der Verwendung von *fiæ*-Prolog-Programmen in TV-SWI-Prolog können Sie im Editor die Zeichenfolge „(“ durch die Zeichenfolge „(“ ersetzen.

keine Leerzeichen
zwischen Funktor
und Klammer

```
vater (paul, karin) wird in fiæ-Prolog akzeptiert und
vater(paul, karin) in TV-SWI-Prolog.
```

2.3 Fakten

Ein Prolog-Programm besteht aus Fakten, Regeln und Anfragen. Fakten und Regeln laden Sie mit *consult* in die interne Wissensbasis, Anfragen schreiben Sie in das Input-Fenster.

Ein Faktum beschreibt eine Eigenschaft eines Objekts oder eine Beziehung zwischen mehreren Objekten.

```
maennlich(paul).           /* Paul ist männlich. */
maennlich(fritz).
maennlich(steffen).
maennlich(robert).

weiblich(karin).          /* Karin ist weiblich. */
weiblich(lisa).
weiblich(maria).
weiblich(sina).

vater(steffen, paul).     /* Steffen ist Vater von Paul */
vater(fritz, karin).
vater(steffen, lisa).
vater(paul, maria).

mutter(karin, maria).    /* Karin ist Mutter von Maria */
mutter(sina, paul).
```

Beispiele für
Fakten

Jedes Faktum beginnt mit einem Namen, dem sogenannten *Funktor*, welcher mit einem Kleinbuchstaben beginnen muß. Auf den Funktor folgt in Klammern eine Liste der *Argumente*. Eine Faktendefinition schließt mit einem *Punkt*.

Funktor, Argumen-
te und Punkt

Ein Bezeichner, der mit einem Großbuchstaben beginnt, steht für eine *Variable*. Da Fakten Konstanten und keine Variablen beinhalten, beginnen auch alle Argumente mit Kleinbuchstaben.

Variablen-
bezeichner mit
Großbuchstaben

Setzt man die Vornamen in Anführungsstriche, so kann man die gewohnte Namensschreibweise benutzen:

```
maennlich('Paul').
```

Prädikat: Funktor und Stelligkeit	Die vierzehn aufgeführten Fakten gehören zu vier verschiedenen <i>Prädikaten</i> : <i>maennlich</i> , <i>weiblich</i> , <i>vater</i> und <i>mutter</i> . Jedes Prädikat wird durch seinen <i>Funktor</i> und seine <i>Stelligkeit</i> (Arität, Anzahl der Argumente) charakterisiert. In der Literatur findet man daher oft Prädikatsbezeichnungen der Art <i>maennlich/1</i> , <i>weiblich/1</i> , <i>vater/2</i> und <i>mutter/2</i> . <i>fiæ</i> -Prolog verwendet diese Schreibweise bei der Ausgabe des Inhaltsverzeichnisses der Wissensbasis mit dem Menüpunkt <i>Database/dir</i> .
maennlich/2	
Klauseln: Fakten und Regeln	Die vier Prädikate unterscheiden sich in der Anzahl der <i>Klauseln</i> (Fakten und Regeln), welche für ein Prädikat definiert sind. Zu den Prädikaten <i>maennlich</i> , <i>weiblich</i> und <i>vater</i> gibt es je vier Klauseln, zum Prädikat <i>mutter</i> gibt es lediglich zwei Klauseln.

2.4 Regeln

Eine Regel stellt eine logische Aussage in Form einer Wenn-Dann-Beziehung dar. Mit einer Regel können aus bekannten Fakten neue Fakten logisch gefolgert werden.

Beispiele für Regeln

```
elternteil(E, Kind):- vater(E, Kind).
elternteil(E, Kind):- mutter(E, Kind).

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

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

grossmutter(A, B):- mutter(A, Kind),
                    elternteil(Kind, B).

bruder(X, Y):- maennlich(X),
               elternteil(E, X),
               elternteil(E, Y),
               X \== Y.
```

Regel: Regelkopf :- Regelrumpf.	Jede Regel besteht aus einem <i>Regelkopf</i> , dem Prolog-Atom ':- ' und einem <i>Regelrumpf</i> . Der Regelkopf stellt die logische Schlußfolgerung (Konklusion) dar, welche sich aus dem Regelrumpf ergibt. Der Regelrumpf enthält die Konjunktion aller Bedingungen (Prämissen), aus welcher sich die Konklusion er-
------------------------------------	--

gibt. Syntaktisch wird die Konjunktion zweier Prämissen durch das trennende Komma dargestellt.

Gibt es zur Herleitung einer Konklusion mehrere Alternativen, wie bei den Prädikaten *elternteil/2* und *vorfahr/2*, so verteilt man diese üblicherweise auf mehrere Klauseln. Stattdessen könnte man die Disjunktion der Prämissen syntaktisch auch durch Semikola ausdrücken.

Alternativen auf Klauseln verteilen

Im Unterschied zu Fakten enthalten Regeln meist nur Variablen. Der Gültigkeitsbereich einer Variablen bezieht sich lediglich auf die Regel, in der sie auftritt. Der Variablenbezeichner *Kind* tritt in obigen Regeln neun mal auf. Tatsächlich werden damit vier verschiedene Variablen bezeichnet.

Gültigkeitsbereich von Variablen

Der Operator `\==` in der *bruder*-Regel bedeutet *nicht identisch*, ansonsten wäre jeder Sohn sein eigener Bruder. Zwar werden in der *bruder*-Regel zwei verschiedene Variablen benutzt, diese können aber mit dem gleichen Wert *instanziiert* werden.

instanzierte Variable

2.5 Rekursive Prädikate

Interessiert man sich für seine Vorfahren, so kann man dafür weitere Regeln formulieren:

```
grosselternteil(Vorfahr, Nachfahr):-
    elternteil(Vorfahr, Person),
    elternteil(Person, Nachfahr).
urgrosselternteil(Vorfahr, Nachfahr):-
    elternteil(Vorfahr, Person),
    grosselternteil(Person, Nachfahr).
ururgrosselternteil(Vorfahr, Nachfahr):-
    elternteil(Vorfahr, Person),
    urgrosselternteil(Person, Nachfahr).
```

nicht rekursive
vorfahr-Prädikate

Es ist natürlich müßig, für jede weitere Generation von Vorfahren eine weitere Regel anzugeben, zumal das Bildungsschema der Regeln klar ist. Das Schema für die Vorfahr-Regeln kann durch ein rekursives *vorfahr*-Prädikat erfaßt werden:

```
vorfahr(X, Y):-
    elternteil(X, Y).
vorfahr(X, Y):-
    elternteil(X, Z),
    vorfahr(Z, Y).
```

rekursives
vorfahr-Prädikat

Die erste *vorfahr*-Klausel ist nicht-rekursiv und sorgt für die Terminierung der Rekursion, die zweite *vorfahr*-Klausel enthält den rekursiven Aufruf.

2.6 Anfragen

Hat man mittels *consult* Fakten und Regeln in die Wissensbasis des Prolog-Interpreters geladen, so können Anfragen im Input-Fenster gestellt werden.

Beispiele für
Anfragen

```
?- maennlich(paul).           Antwort: yes.
?- weiblich(gerhard).        Antwort: no.
?- maennlich(roehner).        Antwort: no.
?- mutter(karin, maria).      Antwort: yes.
?- elternteil(sina, paul).     Antwort: yes.
?- sohn(karin).               Antwort: no.
?- weiblich(maria), sohn(paul). Antwort: no.
```

Entscheidungs-
fragen

Kommen in der Anfrage nur Konstanten und keine Variablen vor, so antwortet der Prolog-Interpreter mit *yes* oder *no*. *Yes* wird ausgegeben, wenn die Anfrage aus den Fakten und Regeln ableitbar ist, anderenfalls *no*. Wie die dritte Anfrage zeigt, hat *no* keinesfalls die Bedeutung von nicht. Für interessantere Anfragen verwendet man Variablen, wie in folgenden Beispielen:

Ergänzungs-
fragen

```
?- maennlich(Mann).
  Antworten:  Mann = paul, Mann = fritz,
              Mann = steffen, Mann = robert

?- mutter(sina, Kind).
  Antwort:    Kind = paul

?- elternteil(Elter, maria).
  Antworten:  Elter = paul, Elter = karin

?- weiblich(Tochter), vater(V, Tochter).
  Antworten:  Tochter = karin, V = fritz
              Tochter = maria, V = paul
              Tochter = lisa, V = steffen

?- weiblich(Tochter), vater(_, Tochter).
  Antworten:  Tochter = karin, Tochter = maria,
              Tochter = lisa
```

```
?- vater(X, paul); mutter(X, paul).
    Antworten: X = steffen, X = sina
```

anonyme Variable In der vorletzten Anfrage kommt eine anonyme Variable vor, welche mit dem Unterstrich „_“ notiert wird. Mit anonymen Variablen kann man die Ausgabe unerwünschter Information unterdrücken, wie es der Vergleich mit der vorangehenden Anfrage zeigt. Die letzte Anfrage macht deutlich, daß der winzige syntaktische Unterschied zwischen „;“ und „,“ einen großen semantischen Unterschied ausmacht: Konjunktion mit Komma und Disjunktion mit Semikolon.

maschinelles logisches Schließen

2.7 ProVisor - Visualisierung von Prolog

Es ist schon frappierend, daß eine typisch menschliche Fähigkeit, nämlich das logische Schließen, auch von Maschinen geleistet werden kann. Es gibt in unserer Wissensbasis kein Faktum *elternteil(paul, maria)*, dennoch ermittelt der Prolog-Interpreter aufgrund der Anfrage *elternteil(Elter, maria)* die Lösung *Elter = paul*.

Zum Verständnis und zur Beurteilung wissensbasierter Informatiksysteme

Und-Oder-Beweisbaum

muß man Einblick in deren Aufbau und Wirkungsweise nehmen. Dies soll durch Visualisierung des Aufbaus und Ablaufs von Prolog-Programmen anhand von Und-Oder-Beweisbäumen geschehen.

Herkömmliche Medien beschränken sich auf die Beschreibung der Ar-

Visualisierung mit ProVisor

beitsweise eines Prolog-Interpreters anhand eines Beispiels oder die Skizze eines fertigen Beweisbaums. Mit *ProVisor* stehen uns ganz neuartige Zugänge zur Verfügung. ProVisor ermöglicht die individuelle, interaktive und schrittweise Abarbeitung selbstgeschriebener Prolog-Programme und zeigt die Zwischenschritte grafisch als Und-Oder-Beweisbaum an.

Da ProVisor ebenfalls über die Benutzungsoberfläche Turbo-Vision verfügt und sich der Bildschirmaufbau an dem von *fiæ*- und TV-SWI-Prolog orientiert, kostet der Einsatz von ProVisor keine zusätzliche Unterrichtszeit. Da-

Bearbeiten von Anfragen

teilschnittstelle, Editor, Fensterverwaltung und Prolog-Interpreter sind wie in *fiæ*- und TV-SWI-Prolog vorhanden. Angenehm ist, daß die Unannehmlichkeiten von *fiæ*-Prolog in ProVisor ausgemerzt sind, die Anfrage ordentlich editiert werden kann und im Anfragefenster mit der *Pfeil-Auf-Taste* die zuvor eingegebenen Anfragen angesehen, bearbeitet und nochmals ausgeführt werden können.

Grundsätzlich läuft ProVisor auch auf einem PC, XT oder AT mit Herculeskarte, allerdings ist dann die Ausführungsgeschwindigkeit nicht berauschend. Dies rührt daher, daß Prolog eine interpretierende Sprache ist und in ProVisor sämtliche Ein- und Ausgaben über den Grafikschiem erfolgen, da dieses System eine Adaption von Turbo-Vision für Grafikbildschirme verwendet.

Die Ausführungsgeschwindigkeit läßt sich durch Verwendung einer RAM-Disk (RAMDRIVE.SYS) erhöhen, in die man die Overlay-Datei PROVISO.ROV lädt. Beachten Sie bei der Installation von ProVisor die Konfigurationsdatei PROVISO.PTH, in welche Sie bitte die zugehörigen Pfade eintragen, zum Beispiel:

Konfigurationsdatei

PROVISO.PTH

```
OVR = C:\PROLOG\PROVISO
BGI = C:\PROLOG\PROVISO
LOG = C:\PROLOG
BMP = C:\PROLOG\PROVISO
PRO = C:\PROLOG
OVRBUFFER = 128
EDITBUFFER = 32
WINDOWBUFFER = 64
```

graphische
Grundelemente von
ProVisor

Die graphischen Grundelemente von ProVisor sind Knoten und Kanten. Jeder Knoten besteht aus einem oberen oder unteren Halboval und in der Standard-Einstellung aus zwei Parameterleisten. Im Halboval steht der Prädikatsname (Funktor), in der oberen Parameterleiste die Argumente und in der unteren Parameterleiste Variableninstanzierungen. Knoten mit oberen Halbovalen stehen für Anfragen oder Teilziele, die während der Abarbeitung eines Regelrumpfes auftreten.

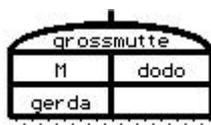
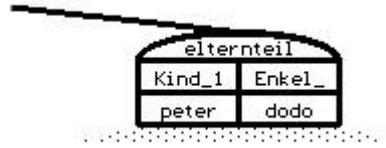


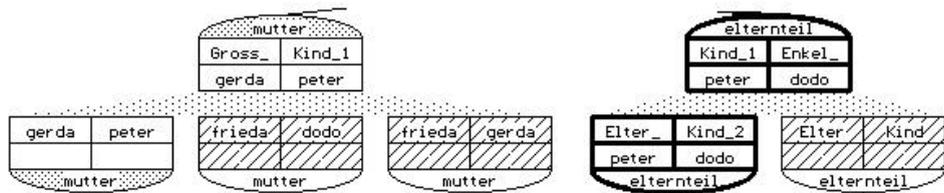
Abb. 2-1
Anfrage- und
Teilzielknoten



Oder-Verknüpfung
der Lösungs-
kandidaten

Zur Lösung einer Anfrage oder eines Teilziels, was selbst wiederum eine Anfrage darstellt, sucht der Prolog-Interpreter aus der Wissensbasis Klauselköpfe mit gleichem Funktor und gleicher Arität heraus. Jeder gefundene Klauselkopf kann potentiell zu einer Lösung führen, weswegen die Lösungskandidaten in Form einer Oder-Verknüpfung mit dem Anfrage- oder Teilzielknoten über ein punktiertes Trapez verbunden werden. Die Klauselköpfe der Lösungskandidaten werden mit einem unteren Oval versehen. So wie ein Deckel auf einen Topf paßt, so paßt ein Anfrage- oder Teilzielknoten mit oberem Oval zu einem oder mehreren Knoten für Lösungskandidaten mit unterem Oval.

Abb. 2-2
Oder-Knoten



Läßt sich der Klauselkopf eines gefundenen Lösungskandidaten mit dem Anfrage- oder Teilzielknoten unifizieren, so wird der Klauselkopf mit seinem Klauselrumpf aus einem oder mehreren Teilzielen in Form einer Und-Verknüpfung durch Strecken miteinander verbunden. Bei Fakten entfällt dieser Schritt, sie stellen die Blätter im Und-Oder-Beweisbaum dar.

Und-Verknüpfung für Anwendung einer Regel

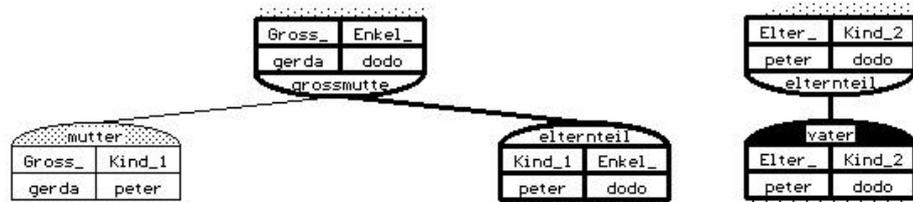


Abb. 2-3 Und-Knoten

Das Prolog-Programm aus unseren Fakten, Regeln und der Anfrage *?- tochter(maria, Elter)*. setzt ProVisor in folgenden Und-Oder-Beweisbaum um.

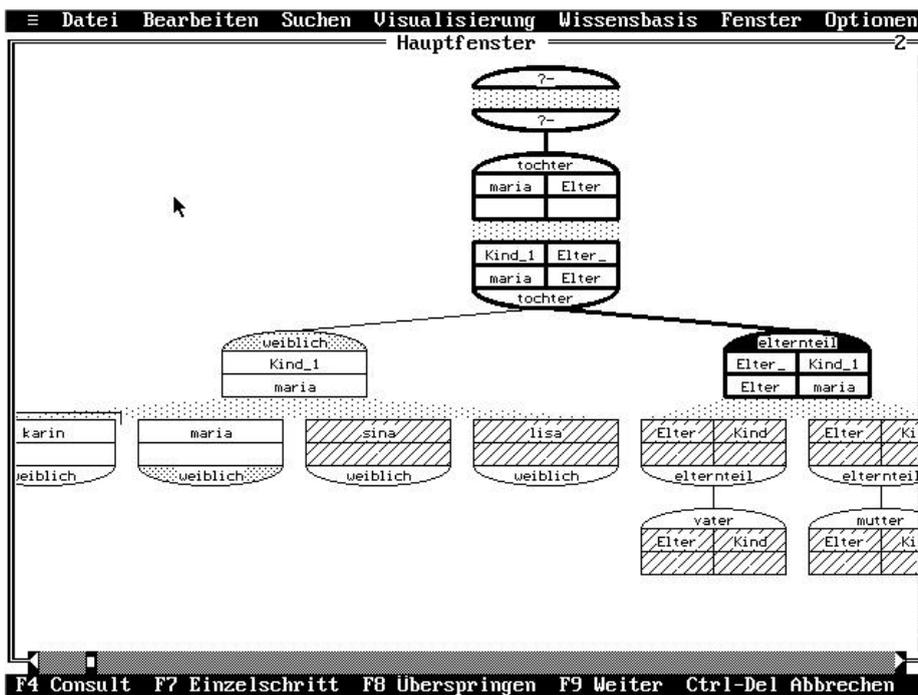


Abb. 2-4 Und-Oder-Beweisbaum

Der Anfrageknoten „?-“ mit den zwei Halbovalen ohne Parameterleisten wird mit dem Knoten *tochter(maria, Elter)* durch eine Strecke verbunden.

Der Regelkopf der *tochter*-Regel ergibt einen Und-Knoten im Beweisbaum, da der Regelrumpf aus einer Konjunktion der beiden Teilziele *weib-*

lich(Kind) und *elternteil(Elter, Kind)* besteht. Die Teilzeile werden als Unterknoten durch Strecken mit dem übergeordneten Und-Knoten verbunden.

Zum Knoten *weiblich(Kind)* gibt es in der Wissensbasis vier Fakten. Jedes Faktum kann potentiell zu einer Lösung führen, wir haben vier Alternativen zur Verfügung. Der Knoten *weiblich(Kind)* ist also ein Oder-Knoten, dessen mögliche Alternativen über ein punktiertes Trapez mit dem übergeordneten Oder-Knoten verbunden werden.

Zum Teilziel *elternteil(Elter, Kind)* gibt es die beiden Regeln

```
elternteil(E, Kind):- vater(E, Kind).
elternteil(E, Kind):- mutter(E, Kind).
```

Der *elternteil*-Knoten ist daher ebenfalls ein Oder-Knoten mit den beiden Regeln als Alternativen. Die Und-Knoten der beiden Regelköpfe sind mit den zugehörigen Knoten der Regelrümpfe *vater* und *mutter* verbunden. Weder der *vater*- noch der *mutter*-Knoten sind mit den zugehörigen Fakten verbunden, was damit zusammenhängt, daß ProVisor den Beweisbaum schrittweise und nur soweit aufbaut, wie es für eine Lösung der Anfrage nötig ist.

aktiver Pfad und
aktueller Knoten

Der aktive Pfad im Beweisbaum ist durch fett umrahmte Knoten hervorgehoben. Der aktuelle Knoten befindet sich am Ende des aktiven Pfades und wird durch inverse Darstellung des Ovals gekennzeichnet. Im nächsten Schritt geht es vom aktuellen Knoten, also vom Elternteil-Knoten aus, mit der Suche nach Vätern weiter.

schraffierte
Parameterleisten

Die Parameterleisten noch nicht besuchter Knoten werden schraffiert dargestellt. Bei besuchten Knoten verschwindet die Schraffur. Unter anderem erkennt man daran, daß die beiden Fakten *weiblich(karin)* und *weiblich(maria)* schon besucht wurden. Der Deckel über *weiblich(karin)* zeigt an, daß dieser Fakt für den gesuchten Beweis nicht verwendet werden konnte.

punktierte Ovale

Erfolgreiche Teilziele erkennt man an punktierten Ovalen. Unser Beweisbaum enthält derzeit zwei erfolgreiche Knoten: das Faktum *weiblich(maria)* und den Oder-Knoten *weiblich(Kind)*.

Übersichtsbaum

Da Beweisbäume recht groß werden können, kann man im Anzeigefenster scrollen. Zudem können Sie über den Wahlpunkt *Visualisierung/Übersichtsbaum* einen verkleinerten Beweisbaum darstellen, dessen Knoten allerdings keine Textinformation mehr beinhalten.

vereinfachter Und-
Oder-Beweisbaum

Mit einer vereinfachten Symbolik können Sie und Ihre Schüler Und-Oder-Beweisbäume in Übungsphasen und Klausuren verwenden. Bei Knoten läßt man die Umrahmung weg und notiert lediglich Prädikatsnamen und Argumente. Für Und- und Oder-Verknüpfungen verwendet man Strecken. Und-Verknüpfungen erhalten zusätzlich einen Verbindungsbogen.

In dieser vereinfachten Symbolik sieht unser Beispiel wie folgt aus:

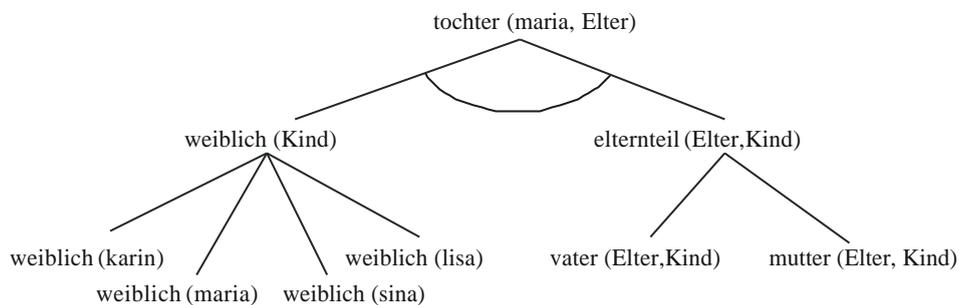


Abb. 2-5
vereinfachter Und-
Oder-Beweisbaum

2.8 Maschinelles logisches Schließen

Läßt man mit ProVisor den Prolog-Interpreter schrittweise arbeiten, so wird sukzessive nach Maßgabe des zugrundeliegende Resolutionsverfahrens der Beweisbaum aufgebaut. Aus der Wissensbasis werden die zur Anfrage passenden Regeln und Fakten herausgesucht und auf Verwendbarkeit hin untersucht. Eine Präzisierung des Begriffs verwendbar ist *unifizierbar*. Die Unifikation eines Teilziels mit einem Faktum führt dazu, daß die Variablen des Teilziels mit Konstanten des Faktums *instanziiert* werden.

Resolution

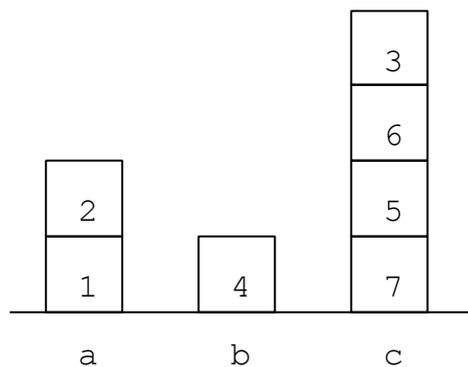
Unifikation

Offene Teilziele werden genauso behandelt, wie der Anfrageterm. Mit dem ersten Teilziel geht es los, wobei der Baum in die Tiefe erstellt wird. Die Fortsetzung der Tiefensuche endet, wenn im Beweisbaum ein Blatt, also ein Faktum erreicht wird. Wenn ein Teilziel erfüllt ist, geht es im Beweisbaum schrittweise zurück, um gemäß dem Resolutionsprinzip die noch offene Teilziele von Und-Knoten zu beweisen. Dabei wird der Beweisbaum in der Breite erstellt.

2.9 Aufgaben

- 1a) Starten Sie Ihren Prolog-Interpreter, legen Sie eine neue Datei an und geben Sie im Editor unsere Familienbeziehungen als Fakten und Regeln ein.
 - b) Überprüfen Sie die Anfragen aus Kapitel 2.5.
 - c) Formulieren Sie Anfragen zu: Eltern von Paul, Bruder von Lisa, Kinder von Karin, Großmutter von Maria, Großvater von Maria, alle Großmütter, alle Väter, alle Eltern-Kind-Beziehungen, sind Lisa und Paul Geschwister, Vorfahren von Maria.
2. Wissensbasis - Consult versus Reconsult (nur für *fiæ*-Prolog)
- a) Zur Erledigung von Aufgabe 1 haben Sie Ihre Familiendatei konsultiert. Lassen Sie sich das Inhaltsverzeichnis und das Listing der Wissensbasis anzeigen.
 - b) Konsultieren Sie die Familiendatei nochmals. Wie hat sich die Wissensbasis verändert?
 - c) Rekonsultieren Sie die Familiendatei. Was ändert sich dadurch? Stellen Sie jetzt die ursprüngliche Wissensbasis wieder her.
 - d) Ergänzen Sie in der Familiendatei eine Regel für *istOmaVon(Oma, Enkel)*. Wie nimmt man die neue Regel in die bestehende Wissensbasis auf?
3. Geben Sie Regeln an für:
- a) `grosselternteil(Grosselter, Enkel).`
 - b) `istKindVon(Kind, Elter).`
 - c) `istTanteVon(Tante, Person).`
 - d) `sindGeschwister(Kind1, Kind2).`
 - e) `verheiratet(Mann, Frau).`
4. Verfolgen Sie die Arbeitsweise des Prolog-Interpreters mit ProVisor für Beispiele aus Aufgabe 1c.

5. An einem runden Tisch sitzen symmetrisch sechs Personen:
 Alfred, Anton, Antonia, Annemarie, Anna und August.
- a) Erstellen Sie eine Wissensbasis mit dem Prädikat *rechts_neben(X, Y)* mit der Bedeutung: X sitzt direkt rechts neben Y.
- b) Ermitteln Sie soweit möglich folgende Antworten:
 Wer sitzt rechts neben Anna?
 Von wem ist Antonia der linke Nachbar?
 Wer sind die Nachbarn von Anton?
 Wer sitzt Alfred gegenüber?
 Wer sitzt wem gegenüber?
- c) Geben Sie Regeln an für:
`links_neben(Links, Rechts):- ?`
`nachbar_von(Mitte, Links, Rechts):- ?`
`gegenueber(Hier, Dort):- ?`
6. Auf den Plätzen a, b und c liegen die Blöcke 1, 2, 3, 4, 5, 6 und 7 in der gezeichneten Anordnung, welche wie folgt in einem Prolog-Programm repräsentiert ist: *auf(1, a), auf(2, 1), auf(4, b),... auf(3, 6)*.



- a) Programmieren Sie als Regel ein *istPlatz*-Prädikat, welches Plätze charakterisiert:
`?- istPlatz(a).` Antwort: *yes.*
`?- istPlatz(2).` Antwort: *no.*
- b) Programmieren Sie ein *ueber*-Prädikat. *ueber(X, Y)* soll gelten, falls Block X über Block Y liegt, wobei X nicht unbedingt direkt auf Y liegen muß.

7. In der *Pizzeria Lucania* ist die Auswahl zwar nicht sehr groß, aber das Essen ist vorzüglich. Die Karte zeigt:

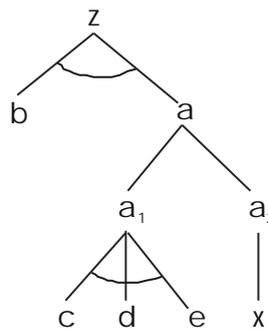
Salat: Mista 5,-; Capriciosa 8,-
 Fleisch: Cotoletta 11,-; Filetto ai Ferri 20,-
 Nudeln: Spaghetti Bolognese 8,-; Lasagne 10,-; Canelloni 11,-
 Fisch: Calamari Fritti 15,-; Sogliola 18,-
 Dessert: Tiramisu 4,-; Cassata 5,-

- a) Erfassen Sie die Speisekarte durch Fakten.
 b) Ein Menü besteht aus Salat, Hauptgericht und Dessert. Hauptgerichte sind Fleisch, Nudeln oder Fisch. Formulieren Sie Regeln für
 - Hauptgerichte
 - Menüs

8. Stellen Sie das folgende abstrakte Prolog-Programm als Und-Oder-Baum dar:

b:- j, k.
 a:- b, c, d.
 b:- e, f.
 d:- m, n.
 b:- g, h, i.
 d:- l.
 ?- a.

9. Welches abstrakte Prolog-Programm hat folgenden Und-Oder-Baum:



3 Ablaufverfolgung und Veranschaulichung

3.1 Trace

Mit ProVisor steht ein ausgezeichnetes Werkzeug zur Verfügung, mit dem man den Ablauf eines Prolog-Programms verfolgen und Fehler suchen kann. Probieren Sie dies ruhig mal mit der linksrekursiven Version des *vorfahr*-Prädikats aus:

```
vorfahr(X, Y):- elternteil(X, Y).
vorfahr(X, Y):- vorfahr(X, Z), elternteil(Z, Y).
```

linksrekursives
vorfahr-Prädikat

Die schrittweise Abarbeitung eines Prolog-Programms wird auch in üblichen Prolog-Interpretern unterstützt, allerdings in rein textlicher Form. Den Trace-Modus schaltet man in *fiæ*-Prolog über die Anfrage *?- trace* ein. In TV-SWI-Prolog reicht das Drücken der Funktionstaste F10, bei ProVisor ist standardmäßig der Trace-Modus eingeschaltet. Nach dem Einschalten treten im Output-Fenster einige Abkürzungen auf:

CALL: Ein Teilziel wird vom System zur Bearbeitung aufgerufen.
EXIT: Das angezeigte Teilziel wurde erfolgreich bewiesen.
REDO: Backtracking. Das Teilziel soll nochmal bewiesen werden.
FAIL: Das Teilziel konnte nicht bewiesen werden.

Trace-Ports

Betrachten wir zum Trace ein kleines Beispiel:

```
artikel(der).

nomen(hund).
nomen(jaeger).

verb(bellt).
verb(flieht).
verb(schiesst).

nominalphrase(X, Y):- artikel(X), nomen(Y).
verbalphrase(Z)      :- verb(Z).
satz(X, Y, Z)        :- nominalphrase(X, Y),
                        verbalphrase(Z).
```

Beispiel zum
Trace

Die Anfrage *?- satz(der, jaeger, bellt)*. liefert in ProVisor das folgende Trace-Protokoll, das bei diesem ausgewählten Beispiel gut überschaubar ist und einen Einblick in die Arbeitsweise des Prolog-Interpreters gewährt.

Trace von ZWERG-
Prolog

```
?- satz(der, jaeger, bellt).
CALL: satz(der, jaeger, bellt)
CALL: nominalphrase(der, jaeger)
CALL: artikel(der)
EXIT: artikel(der)
CALL: nomen(jaeger)
EXIT: nomen(jaeger)
EXIT: nominalphrase(der, jaeger)
CALL: verbalphrase(bellt)
CALL: verb(bellt)
EXIT: verb(bellt)
EXIT: verbalphrase(bellt)
EXIT: satz(der, jaeger, bellt)
satz(der, jaeger, bellt)
yes
REDO: satz(der, jaeger, bellt)
REDO: verbalphrase(bellt)
REDO: verb(bellt)
FAIL: verb(bellt)
FAIL: verbalphrase(bellt)
REDO: nominalphrase(der, jaeger)
REDO: nomen(jaeger)
FAIL: nomen(jaeger)
REDO: artikel(der)
FAIL: artikel(der)
FAIL: nominalphrase(der, jaeger)
FAIL: satz(der, jaeger, bellt)
no
```

Trace-Protokoll

Mit *notrace* in *fiæ*-Prolog beziehungsweise über den Menüpunkt *Optionen Trace An/Aus* in ProVisor schalten Sie den Trace-Modus wieder ab. In TV-SWI-Prolog wird der Trace-Modus automatisch nach jedem Tracelauf abgeschaltet. Über den Menüpunkt *Optionen, Protokoll An/Aus* kann die Trace-Ausgabe von ProVisor und TV-SWI-Prolog in die Datei TRACE.LOG protokolliert werden. In *fiæ*-Prolog schalten Sie die Protokollierung mit *?- tell('trace.log')* ein und beenden sie mit *?- told*.

3.3 Spuren

Der Trace-Modus eignet sich in der Regel nur bei kleinen Beispielen zur Ablaufverfolgung. Kommt Rekursion ins Spiel, und das ist in Prolog fast immer der Fall, so sind die Trace-Protokolle in der Regel viel zu detailliert, um in den ellenlangen Listings die Information zu finden, für die man sich eigentlich interessiert. Das Trace-Protokoll wird zur Trace-Katastrophe.

Meta-Interpreter Daher hat der Autor der *Spur*-Konzeption aus [Stel] folgend mit einer in [Fuc1] beschriebenen Methode einen Meta-Interpreter für *fiæ*-Prolog programmiert, welcher in Form von Spuren, die Lösungssuche eines Prolog-Interpreters darstellt.

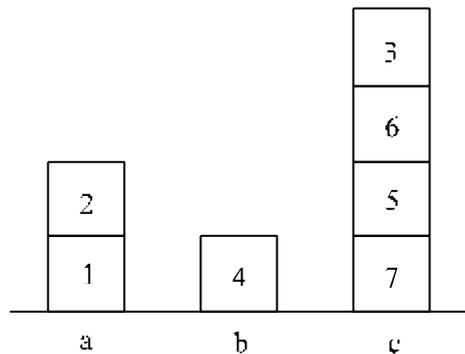
1. Beispiel für den Einsatz des *spur*-Prädikats

```
?- spur(satz(der, jaeger, bellt)).

satz(der, jaeger, bellt)
  nominalphrase(der, jaeger)
    artikel(der)
      nomen(jaeger)
        verbalphrase(bellt)
          verb(bellt)
yes
```

Der Metainterpreter rückt entsprechend der Aufruftiefe die zu lösenden Teilziele ein. Dies ist insbesondere bei Rekursionen sehr hilfreich, da sich Rekursionstiefen an den zugehörigen Einrückungen ablesen lassen.

2. Beispiel für den Einsatz des *spur*-Prädikats



Beschreibung einer Klötzchenwelt

```
auf(1, a). auf(2, 1). auf(4, b).
auf(7, c). auf(5, 7). auf(6, 5). auf(3, 6).

ueber(X, Y):- auf(X, Y).
ueber(X, Y):- auf(Z, Y), ueber(X, Z).
```

```
?- spur(ueber(3,c)).
```

```
ueber(3,c)
  auf(3,c)   nein
ueber(3,c)
  auf(7,c)
  ueber(3,7)
    auf(3,7)   nein
  ueber(3,7)
    auf(5,7)
  ueber(3,5)
    auf(3,5)   nein
  ueber(3,5)
    auf(6,5)
  ueber(3,6)
    auf(3,6)
```

```
Yes
```

Einrückungen
gemäß der
Aufruftiefe

Die Benutzerschnittstelle des Metainterpreters ist das *spur*-Prädikat. Die zu untersuchende Anfrage geben Sie dem *spur*-Prädikat als Argument mit. Zuvor müssen Sie den Metainterpreter aus der Datei SPUR.PRO konsultieren.

3.4 Aufgaben

1. Skizzieren Sie im Boxen-Modell alle Lösungen der Anfrage:
`?- satz(der, X, Y).`
2. Kontrollieren Sie die Lösung aus Aufgabe 1, indem Sie mit ProVisor ein Trace-Protokoll erstellen. Benutzen Sie die Funktionstaste F9, um jeweils bis zur nächsten Lösung zu tracen.
3. Skizzieren Sie im Vierport-Modell die Abarbeitung der Anfrage:
`?- satz(die, hund, bellt).`
4. Die Konjunktion von Teilzielen wird durch Reihenschaltung von Boxen im Boxen-Modell dargestellt. Wie kann die Disjunktion von Teilzielen dargestellt werden?

Zeichnen Sie für die Anfrage `?- a.` bei gegebenen Regeln

`a:- b, c.`

`a:- d, e, f.`

einen Vierport-Graphen.

5. Verfolgen Sie die Spur der Anfrage aus Aufgabe 6.

4 Datenbanken

4.1 Hotelangebote von Froh-Reisen

In Prolog lassen sich Aspekte und Konzepte relationaler Datenbanken auf elementarem Niveau behandeln. Dies soll in diesem Kapitel am Beispiel des fiktiven Reiseveranstalters *Froh-Reisen* deutlich werden.

Im Katalog bietet *Froh-Reisen* Pauschalreisen in verschiedene Gebiete des Mittelmeerraums an. Die Qualität der angebotenen Hotels wird durch eine entsprechende Anzahl von Sternen charakterisiert. Zur eindeutigen Identifizierung wird jedes Hotel mit einem Buchungscode versehen.

Die Hotelangebote legen wir in der Prolog-Wissensbasis ab. Dazu definieren wir ein *hotel*-Prädikat mit folgenden Argumenten:

```
hotel(Code, Name, Sterne, Gebiet).
```

Die folgenden Beispiele aus dem Hotelbestand zeigen, daß Namen auch mit Großbuchstaben geschrieben werden können, ohne gleich als Variable mißverstanden zu werden. Man setzt dazu Textkonstanten, wie in Pascal, in Anführungsstriche.

```
hotel(emkat1n, 'Turo Pins', 3, 'Mallorca').
hotel(emfaa1n, 'Alba', 3, 'Mallorca').
hotel(emeaf1n, 'Floriana', 4, 'Mallorca').
hotel(egda01n, 'Green Golf', 3, 'Gran Canaria').
hotel(gpah01n, 'Achaios', 2, 'Peloponnes').
hotel(gsg03n, 'Samos Sun', 4, 'Samos').
hotel(zldh01n, 'Odessa', 4, 'Zypern').
hotel(tdah13n, 'Sidi Slim', 2, 'Djerba').
hotel(tdaa01n, 'Dar Jerba', 2, 'Djerba').
hotel(maah16n, 'Les Dunes Dor', 3, 'Agadir').
hotel(maah10n, 'Ali Baba', 3, 'Agadir').
```

Beispiele zur
hotel-Relation

Auf diesem Datenbestand sind die beiden folgenden relationalen Grundoperationen möglich:

relationale Grundoperationen

Selektion - Auswahl bestimmter Datensätze

Selektion

Beispiel: Hotels auf Djerba

```
?- hotel(Code, Name, Sterne, 'Djerba').
```

Projektion *Projektion* - Auswahl bestimmter Datenfelder

Beispiel: In welchen Urlaubsgebieten gibt es Hotels?

```
?- hotel(_, Name, _, Gebiet).
```

Froh-Reisen bietet 1- oder 2-wöchige Reisen an. Die Preise richten sich nach der Reisesaison. Am billigsten ist die Saison A, mittlere Preise gibt es in der Saison B und am teuersten ist die Saison C. Die Preise pro Person erfassen wir im Prädikat

```
preis(Code, Saison, Wochen, Preis).
```

Für das Hotel *Turo Pins* sind hier alle Preise aufgeführt:

Beispiele zur
preis-Relation

```
preis(emkat1n, a, 1, 1012).
preis(emkat1n, a, 2, 1439).
preis(emkat1n, b, 1, 1031).
preis(emkat1n, b, 2, 1479).
preis(emkat1n, c, 1, 1162).
preis(emkat1n, c, 2, 1729).
```

Die *hotel*- und *preis*-Relation können über einen *Join* zusammengeführt werden:

Join *Join* - Verknüpfung zweier Tabellen

Beispiel: Hotelpreise

```
?- hotel(Code, Name, Sterne, Gebiet),
   preis(Code, Saison, Wochen, Preis).
```

Der Join wird hier über das Schlüsselattribut *Code* durchgeführt und liefert eine virtuelle Tabelle mit der Struktur:

```
hotelpreise(Code, Name, Sterne, Gebiet,
            Saison, Wochen, Preis).
```

Die Saison richtet sich nach dem Reisegebiet, dem Flughafen des Abflugs und dem Abflugtermin. Zwecks Vereinfachung nehmen wir an, daß sich im Laufe

eines Monats die Saison nicht ändert. Damit können wir zu jedem Reisegebiet und Flughafen die jeweilige Reisesaison angeben:

```
saison(Gebiet, Flughafen, Monat, Saison).
```

Für das Reisegebiet *Mallorca* gilt beispielsweise folgende Saisontabelle:

```
saison('Mallorca', 'Frankfurt', 5, a).
saison('Mallorca', 'Frankfurt', 6, b).
saison('Mallorca', 'Frankfurt', 7, c).
saison('Mallorca', 'Frankfurt', 8, c).
saison('Mallorca', 'Frankfurt', 9, b).
saison('Mallorca', 'Frankfurt', 10, a).
saison('Mallorca', 'Stuttgart', 5, a).
saison('Mallorca', 'Stuttgart', 6, b).
saison('Mallorca', 'Stuttgart', 7, b).
saison('Mallorca', 'Stuttgart', 8, c).
saison('Mallorca', 'Stuttgart', 9, b).
```

Beispiele zur
saison-Relation

4.2 Aufgaben

1. Stellen Sie Anfragen:
 - a) Welche Hotels auf Mallorca sind im Angebot?
 - b) Bietet *Froh-Reisen* das Hotel *Sidi Slim* auf der tunesischen Insel *Djerba* an?
 - c) In welchem Urlaubsgebiet liegt das Hotel *Les Dunes Dor*?
 - d) Welche preiswerten Hotels werden angeboten?
 - e) Gibt es in *Agadir* ein 3-Sterne-Hotel? Ein 4-Sterne-Hotel?
 - f) Wie teuer ist das *Ali Baba* in der Hauptsaison?
 - g) Welche Billigstangebote kommen für einen Kurzurlaub in Frage?
 - h) In welchen Hotels kann man in der Reisezeit B 2 Wochen Urlaub für weniger als 1400,- DM machen.
 - i) Wie teuer ist im August ab Frankfurt das *Samos Sun*?
 - j) Fliegt man im Juli besser ab Stuttgart oder ab Frankfurt nach Gran Canaria?
 - k) Was kosten zwei Wochen Urlaub in Zypern?
 - l) Ist im April auf *Samos* schon Saison?
 - m) In welchen Urlaubsgebieten ist im Oktober noch Saison?

- 2a) Formulieren Sie eine Regel für das Prädikat *billigurlaub*(*Hotel, Gebiet, Kosten*), das möglichst billige Hotels empfiehlt.

- b) Was liefert die Anfrage:
?- billigurlaub(Hotel, Gebiet, Kosten), Kosten < 900.
- c) Mit schulpflichtigen Kindern kann man nur in den Ferien Urlaub machen.
Geben Sie ein Variante von *billigurlaub* für die Ferienzeit an.

4.3 Hotelbuchung

Hat man sich für eine Pauschalreise des Reiseveranstalters *Froh-Reisen* entschieden, so kann man in einem Reisebüro die gewünschte Reise buchen. Zunächst müssen Name und Adresse des Kunden zusammen mit einer Kundennummer erfaßt werden:

```
kunde(Nr, Name, StrasseNr, PLZ, Ort).
```

Beispiele:

Beispiele zur
kunde-Relation

```
kunde(0101, 'Plum', 'Untergasse 4', 64285, 'Darmstadt').
kunde(0102, 'Rettig', 'Odenwaldstr. 17', 64293, 'Darmstadt').
kunde(0104, 'Gaydoul', 'Auf der Beune 8', 64807, 'Dieburg').
kunde(0105, 'Laaber', 'Kochstr. 23a', 64291, 'Darmstadt').
kunde(0113, 'Bozdag', 'Kostheimerstr. 18', 64354, 'Reinheim').
```

Zu den Buchungsdaten gehören der Buchungscode des Hotels, die Anzahl der Personen, das Abflugdatum, die Reisedauer und der Flughafen:

```
buchung(Nr, Code, Personen, Datum, Dauer, Flughafen).
```

Das Datum besteht aus den drei Bestandteilen Tag, Monat und Jahr. Wir speichern Kalenderdaten deshalb in dreistelligen *datum*-Strukturen:

```
datum(Tag, Monat, Jahr).
```

Als Beispiele sind einige Buchungsdatensätze angegeben:

```
buchung(0101, egda01n, 3, datum(2,10, 95), 2, 'Frankfurt').
buchung(0102, gpah01n, 2, datum( 3,9, 94), 2, 'Stuttgart').
buchung(0104, emkat1n, 2, datum(23,7, 94), 2, 'Frankfurt').
buchung(0118, emeaf1n, 1, datum(17,8, 94), 2, 'Stuttgart').
buchung(0104, emkat1n, 3, datum(17,7, 95), 1, 'Frankfurt').
```

Die Aufgaben im nachfolgenden Kapitel zeigen, welche Probleme mit diesem Reisebuchungssystem bearbeitet werden können. In Kapitel 13 wird das Auskunftssystem und Reisebuchungssystem wieder aufgegriffen. Es werden ein Entity-Relationship-Diagramm und eine Benutzerschnittstelle entwickelt.

4.4 Weitere Aufgaben

3. Stellen Sie Anfragen:
 - a) Wo wohnt Familie *Ewert*?
 - b) Wohin fährt Familie *Darlapp* dieses Jahr in Urlaub?
 - c) Wer fährt in das Hotel *Floriana*?
 - d) Welche Buchungen liegen für *Djerba* vor?
 - e) Fliegt jemand in der Hauptsaison?
 - f) In welcher Saison macht Herr *Skrypcak* Urlaub?
4. Entwickeln Sie ein Prädikat, mit festgestellt werden kann, ob ein Hotel zu einem bestimmten Zeitpunkt buchbar ist.
5. Entwickeln Sie ein Prädikat, mit dem die Kosten einer Flugreise ermittelt werden können.
6. Welche Reisegewohnheiten haben die Familien *Gaydoul* bzw. *Skrypcak*?
7. Finanzkräftigen Reisenden soll im Rahmen einer Werbeaktion Fernreisen angeboten werden. Erstellen Sie eine entsprechende Adreßliste.
8. Der Ehemann der Reisekauffrau Monika S. führt ein Kindermodengeschäft. Er hätte gerne eine Adreßliste potentieller Kundinnen.
9. Modellieren Sie ein
 - a) Bibliothekssystem
 - b) Video-Center
 - c) Fahrplan-Auskunftssystem

5 Listen

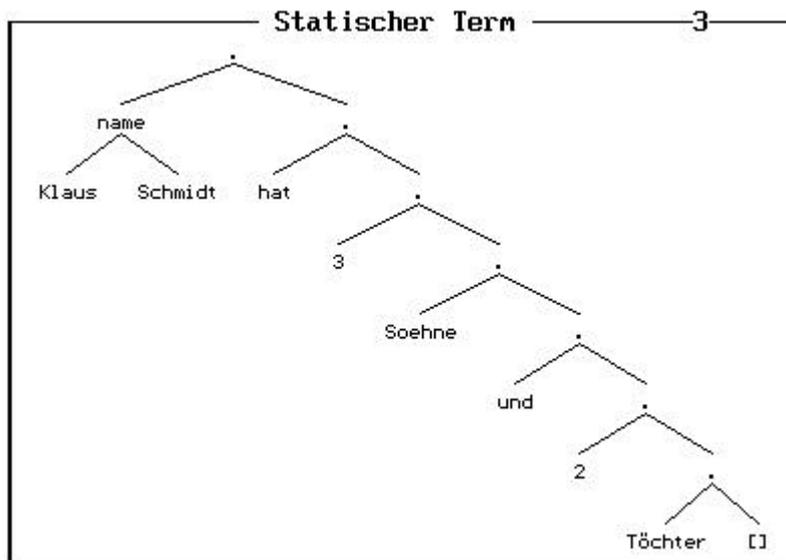
5.1 Einführung von Listen

Eine Liste ist eine geordnete Folge von Elementen beliebiger Länge. Die Ordnung impliziert, daß es ein erstes, zweites, drittes, usw. Element gibt, wobei die Bedeutung der Reihenfolge sich aus der konkreten Anwendung ergibt. Die Elemente einer Liste können beliebige Terme sein: Konstanten, Variablen, Strukturen und sogar auch andere Listen. Im Unterschied zu Listen in imperativen Programmiersprachen können die Listenelemente in Programmiersprachen der KI unterschiedliche Datentypen aufweisen.

[name('Klaus', 'Schmidt'), hat, 3, Soehne, und, 2, 'Töchter'].

Das erste Element ist eine Prolog-Struktur mit dem Funktor *name* und den beiden Argumenten *Klaus* und *Schmidt*, das zweite Element ist die Konstante *hat* und das dritte Element die Konstante *3*. Es folgen die Variable *Soehne* und die Konstanten *und*, *2* und *Töchter*.

Über den Befehl *Visualisierung/Term* können Sie sich in ProVisor ein Bild von der Liste machen:



Listen:
geordnete Folge
von Elementen

Beispiel einer Liste
mit 7 Elementen

Abb. 5-1
Visualisierung
einer Liste mit
sieben Elementen
durch ProVisor

Die Liste wird als entarteter Binärbaum dargestellt. Führen Sie gedanklich um die Wurzel eine Linksdrehung um 45 Grad aus, dann sehen Sie eine echte Liste. Die Elemente der Liste treten im Bild als Blätter auf, welche jeweils links

an innere Knoten angehängt sind. Das Blatt [] signalisiert das Ende der Liste, entspricht daher in PASCAL der Konstanten *Nil*. Die inneren Knoten sind mit einem Punkt „.“ bezeichnet, da der Punkt der Listenfunktorknoten ist.

Die Liste ist eine grundlegende dynamische Datenstruktur. Wie viele Elemente eine Liste aufnimmt, wird oft erst während des Programmlaufs entschieden. In imperativen Programmiersprachen nutzt man den Heap, um zur Laufzeit Speicher für weitere Listenelemente zu erhalten. In Zeigervariablen verwaltet man die Adressen, welche zum Listenaufbau nötig sind. Eine typische Datenstruktur für eine lineare Liste sieht in PASCAL so aus:

Listen in Pascal

```

TYPE ListePtrTyp = ^ListeRecTyp;
   ListeRecTyp = RECORD
       Element: ElementTyp;
       ToNext : ListePtrTyp;
   END;
    
```

Liste als rekursive Datenstruktur

Diese Deklaration beinhaltet alles Wesentliche, was eine Liste charakterisiert. Eine Liste ist offenbar eine rekursive Datenstruktur. Sie besteht entweder aus einem *Kopfelement*, in der Deklaration *Element*, und einer *Restliste*, hier also *ToNext*, oder der leeren Liste, *Nil* bzw. []. Die Dynamik dieser Definition ergibt sich daraus, daß die Restliste beliebig lang sein kann.

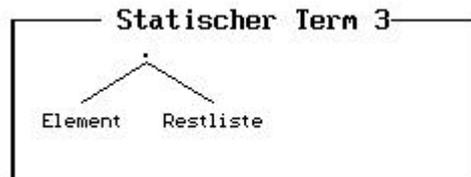
Liste-Operator „|“

Mit der bisherigen Listenschreibweise in Prolog läßt sich die Dynamik nicht beschreiben, nur Listen fester Länge können wir notieren. Die Notation wird um den Liste-Operator „|“ erweitert. Mit ihm kann man aus einem Kopfelement und einer Restliste eine neue Liste zusammensetzen oder eine bestehende Liste in Kopfelement und Restliste aufteilen:

Beispiel:

[Element | Restliste]

Abb. 5-2
Visualisierung der
Listenstruktur
[Element|Restliste]



Diese Liste besteht aus einem Element und einer Restliste. Da über die Restliste nichts näheres ausgesagt ist, sie kann leer sein oder auch viele Elemente beinhalten, ist die Länge der Gesamtliste nicht bekannt. Wir haben also eine Notation zur Verfügung, mit der wir Listen beliebiger Länge beschreiben kön-

nen. Diese Notation ist weitaus wichtiger, als die anfangs eingeführte Notation für Listen bekannter Länge.

Achten Sie darauf, daß der Listen-Operator „|“ unterschiedliche Objekte verbindet. Links vom „|“ steht ein einzelnes Listenelement, rechts davon eine komplette Liste.

- $[a \mid [b, c]]$

ist die Liste mit dem Kopfelement a und der Restliste $[b, c]$. Eine einfachere Schreibweise dafür ist natürlich $[a, b, c]$.

Beispiele zur
Verwendung des
Listen-Operators „|“

- $[\text{gelb} \mid []]$

ist die Liste mit dem Kopfelement gelb und der leeren Restliste. Offenbar hat diese Liste nur ein Element, es ist die Liste $[\text{gelb}]$.

- $?- Y = [b, c, d], X = [a \mid Y]$

macht aus der dreielementigen Liste Y die vierelementige Liste $X = [a, b, c, d]$.

- $?- [X \mid Y] = [a, b, c, d]$

zerlegt die Liste $[a, b, c, d]$ in den Kopf $X = a$ und die Restliste $Y = [b, c, d]$.

- $?- [[a, b], [c, d], [e, f]] = [\text{Element} \mid \text{Restliste}]$

liefert $\text{Element} = [a, b]$ und $\text{Restliste} = [[c, d], [e, f]]$, weil das Kopfelement der dreielementigen Liste $[a, b]$ ist und die Restliste aus zwei Elementen besteht, die zweielementige Listen sind.

- $?- \text{NeuerStack} = [\text{name}('Klaus', 'Schmidt') \mid \text{Stack}]$

setzt das strukturierte Element $\text{name}('Klaus', 'Schmidt')$ vor die alte Liste und erzeugt so eine neue Liste, die ein Element mehr enthält. Man kann dies als die Push-Operation bei der Datenstruktur Keller deuten.

Push-Operation

- `?- [_ | NeuerStack] = Stack`

Pop-Operation entfernt das Kopfelement aus der Liste namens Stack und liefert die Restliste NeuerStack. Dies entspricht der Pop-Operation.

5.2 Punktschreibweise für Listen

einfache
Listennotation

Die obigen Beispiele zeigen, daß man in Prolog weitaus einfacher mit Listen arbeiten kann, als dies in Pascal der Fall ist. Dies liegt unter anderem daran, daß eine viel einfachere Notation zur Verfügung steht. Die Listenelemente werden einfach in eckigen Klammern aufgezählt und durch Kommata getrennt. Für Listenoperationen und den Zugriff auf das Kopfelement oder die Restliste benutzt man den Listen-Operator „|“.

Punktschreibweise
für Listen

Die Klammerschreibweise spielt in Prolog nur für die Ein- und Ausgabe eine Rolle. Intern benutzt Prolog die Punktschreibweise für Listen. Sie basiert auf der Idee, alle Datenstrukturen auf Baumstrukturen zurückzuführen.

Beispiele:

Abb. 5-3
Struktur als Baum

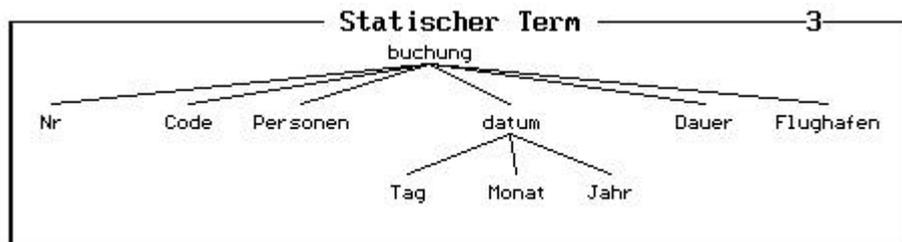


Abb. 5-4
Rechenterm als
Baum

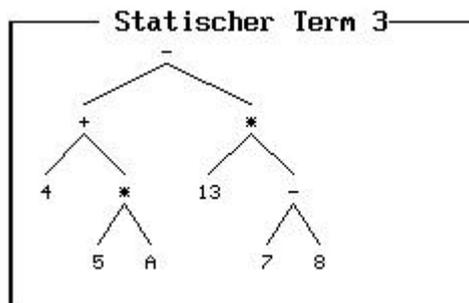




Abb. 5-5
Visualisierung der
Liste [a, b]

Wie man sieht, befinden sich die eigentlichen Daten an den Blättern der Bäume. Die inneren Knoten bilden das Gerüst für den Aufbau der jeweiligen Datenstruktur. Das buchungs-Beispiel aus Abbildung 5-3 zeigt, daß die inneren Knoten den jeweiligen Funktor aufnehmen, und die Liste der Argumente als Teilbäume angefügt werden. Dies wird im folgenden Bild nochmals herausgestellt:

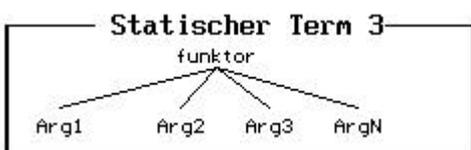


Abb. 5-6
Funktork und
Argumente in der
Baumdarstellung

In Präfix-Schreibweise: `funktork(Arg1, Arg2, Arg3, ArgN)`.

Bei arithmetischen Termen sind die Operatoren die Funktoren, als Operanden kommen Zahlen und Variablen vor. Man schreibt sie normalerweise in Infix-Notation. Selbstgebildete Datenstrukturen, wie zum Beispiel *buch* und *schueler*, schreibt man üblicherweise in Präfix-Notation. Für zweistellige Strukturen ist die Umwandlung von der Präfix- in die Infix-Schreibweise und umgekehrt möglich:

Präfix- und Infix-
Notation

Präfix
`elternteil(heike, robert)`
`mutter(petra, anna)`
`*(2, 4)`
`+(8, 3)`
`-(9, *(4, 5))`
`*(-(9, 4), 5)`

Infix
`heike elternteil robert`
`petra mutter anna`
`2 * 4`
`8 + 3`
`9 -(4 * 5)`
`(9 - 4) * 5`

Beispiele für
Präfix- und Infix-
Notation

Punktschreibweise von Listen Bei Listen ist der Punkt „.“ der Funktor einer zweistelligen Struktur. Die Punktschreibweise von Listen verwendet die Präfix-Notation, bei der dem Funktor in Klammern die beiden Argumente folgen.

Beispiele:

```

[]                leere Liste
.(a, [])         einelementige Liste [a]
.(a, .(b, []))  zweielementige Liste [a, b]
.(a, .(b, .(c, []))) dreielementige Liste [a, b, c]

```

display-Prädikat Wie Sie sehen, ist die Punktschreibweise sehr unhandlich, weswegen man die einfachere Klammernotation für Listen verwendet. Das Systemprädikat *display* gibt übrigens Strukturen immer in der Präfix-Schreibweise aus.

Beispiele:

```

?- display(name('Anton', 'Meier')).
                                liefert: name(Anton, Meier)
?- display([a, b]).             liefert: .(a, .(b, []))
?- display(2+3*4).             liefert: +(2, *(3, 4))

```

Baumstrukturen Die Beispiele machen den Unterschied zwischen Sein und Schein in Prolog deutlich. So unterschiedliche scheinende Dinge wie Strukturen, Listen und arithmetische Terme sind intern nichts anderes als Baumstrukturen.

5.3 Listenoperationen - Von Pascal nach Prolog

Lerntheorie In einigen Prolog-Lehrbüchern wird die These vertreten, daß man alles über das Programmieren vergessen solle, wenn man anfängt mit Prolog zu programmieren. Vergessen wir diesen Unsinn! Aus der Lerntheorie wissen wir, daß man am besten lernt, wenn das Neue auf Altem aufbauen kann, wenn es in das schon bestehende mentale Netz assimiliert werden kann.

Natürlich hat man mit dem prozeduralen und ablauforientierten Konzept von Pascal Schwierigkeiten, wenn man es einfach auf Prolog übertragen will. Da macht es aber mehr Sinn, diese Schwierigkeiten zu analysieren, als bestehendes Wissen zu paralysieren.

Zur Beschreibung von Prädikaten verwenden wir im folgenden eine Notation für Argumente, wie sie im Handbuch von SWI-Prolog zur Darstellung der System-Prädikate benutzt wird. Argumente werden mit den Kennzeichen „+“, „-“ oder „?“ versehen. „+“ bedeutet, daß das Argument eine Eingabe für das Prädikat ist, „-“ kennzeichnet Ausgaben und „?“ Eingabe oder Ausgabe. Die Kennzeichen werden nur zur Beschreibung, nicht aber beim Aufruf von Prädikaten benutzt.

Notation für Argumente: +, -, ?

5.3.1 Fallstudie member

Betrachten wir das *member*-Prädikat. Typisch Prolog ist:

```
member(X, [X|_]).
member(X, [_|Y]):- member(X, Y).
```

Der Pascal-Programmierer sieht das anders. Für ihn ist *member* eine boolesche Funktion, die prüft, ob ein Element E in einer Liste L vorkommt. Er macht daher als Ansatz einen Funktionskopf mit der Parameterliste E und L und untersucht erst im zweiten Schritt, wie member realisiert werden könnte.

```
member(+E, +L)          /* Funktionskopf */
```

Schnittstelle von member

Die Vorgehensweise, erst die Schnittstelle und dann die Implementierung zu realisieren, ist typisch für strukturiertes Programmieren, aber eher atypisch für Prolog.

Nun zur Implementierung. L muß mindestens ein Element und deshalb die Struktur [X|Y] haben. Ist X = E sind wir fertig, ansonsten prüfen wir, ob X in der Restliste Y vorhanden ist. Die If-Then-Else Struktur läßt sich als Oder-Verknüpfung, realisiert durch das Semikolon, in Prolog nachbilden:

```
member(E, L):- L = [X|Y], (X = E; member(E, Y)).
```

Pascal-artige Lösung

Jetzt schauen wir uns an, wie wir von dieser Lösung zur Prolog-Lösung kommen. Verschiedene Fälle werden in Pascal mittels If- oder Case-Anweisung innerhalb einer Prozedur behandelt. Prolog verteilt die Fälle auf mehrere Klauseln zu einem Prädikat:

```
member(E, L):- L = [X|Y], X = E.
member(E, L):- L = [X|Y], member(E, Y).
```

Einsatz der Unifikation Der Unifikationsmechanismus von Prolog erlaubt es, die beiden Teilziele in der ersten Klausel zusammenzufassen. Ob L mindestens ein Element hat und ob das erste Element gleich E ist läßt sich in einem Schritt erledigen. Das geht in Pascal nicht! Hier erweist sich Prolog als mächtiger und der Quelltext wird kürzer:

```
member(E, L):- L = [E|Y].
member(E, L):- L = [X|Y], member(E, Y).
```

Parameterkonzept von Prolog Das Parameterkonzept von Prolog kennt keine Wert- und Variablenparameter. Die zulässigen Ein- und Ausgaben werden allein durch die Implementierung bestimmt. Daher ist es auch vollkommen unproblematisch, Strukturen als Parameter zu übergeben. Bei Variablenparameter in Pascal geht das nicht, weil dort als aktuelle Parameter nur Variablen zugelassen sind. Die Lösung vereinfacht sich daher zu:

```
member(E, [E|Y]).
member(E, [X|Y]):- member(E, Y).
```

Verwendung anonymer Variabler Jetzt sorgt keine explizit programmierte Kontrollstruktur für die richtige Auswahl, dies erledigt implizit der Unifikationsmechanismus. In der ersten Klausel kommt es nicht auf das X , in der zweiten nicht auf das Y an. Statt der benannten Variablen X und Y verwenden wir daher die anonyme Variable „_“ und gelangen somit zum typischen Prolog-Stil:

```
member(E, [E|_]).
member(E, [_|Y]):- member(E, Y).
```

Die gefundene Lösung leistet mehr, als ursprünglich beabsichtigt war. Vorgeesehen war, daß *member* zu gegebenem Element E und gegebener Liste L prüft, ob E in L vorkommt. Dies drückte sich in der Spezifikation durch zwei „+“-Zeichen aus: *member(+E, +L)*. Insgesamt funktioniert *member* in den Varianten:

vielseitiges	<code>member(+E, +L).</code>	Prüft, ob E in L vorkommt.
member	<code>member(-E, +L).</code>	Liefert ein E aus L .
	<code>member(+E, -L).</code>	Liefert variables L mit E .
	<code>member(-E, -L).</code>	Liefert variables L mit variablem E .

Member kann also in allen vier möglichen Instanzierungsmustern aufgerufen werden. Dafür benutzt man dann die Kennzeichnung der Parameter mit dem Fragezeichen:

```
member(?E, ?L).
```

5.3.2 Fallstudie append

Fast jedes Prolog-Lehrbuch gibt `append` wie folgt an:

```
append([], L, L).
append([X|L1], L2, [X|L3]):- append(L1, L2, L3).
```

Ein Prolog-Anfänger wird das nicht verstehen, sofern er nicht die Genese dieses Prädikats vermittelt bekommt. Betrachten wir nun die Genese aus der Pascal-Sicht: die Liste `L2` soll an die Liste `L1` angehängt werden, um die Resultatliste `L3` zu erhalten. Der Pascal-Programmierer sieht somit eine Prozedur mit zwei Wert- und einem Variablenparameter und macht daher den Ansatz:

```
append(+L1, +L2, -L3)    /* Prozedurkopf */
```

Schnittstelle von
`append`

Die Implementierung von `append` kann durch eine Schleife erfolgen, die zum letzten Element von `L1` geht und an dieses Element das erste Element von `L2` anhängt. Alternativ kann man eine rekursive Lösung versuchen: Hängt man an eine leere Liste `L1` eine Liste `L2` an, so erhält man wieder die Liste `L2`. Ist `L1` aber nicht leer, so hängt man `L2` an die Restliste von `L1` an und setzt zum Schluß den Kopf von `L1` davor. Damit erhalten wir folgende pascalartige Implementierung:

rekursive
Implementierung

```
append(L1, L2, L3):-
  (L1 = [], L3 = L2);
  (L1 = [X|R1], append(R1, L2, R3), L3 = [X|R3]).
```

Auftrennen der Fallunterscheidung liefert:

```
append(L1, L2, L3):-
  L1 = [], L3 = L2.
append(L1, L2, L3):-
  L1 = [X|R1],
  append(R1, L2, R3), L3 = [X|R3].
```

Disjunktion durch
Verteilen auf mehrere
Klauseln

Übernahme von Strukturen in die Parameter, bzw. Argumente in der Prolog-Terminologie, ergibt:

```
append([], L2, L3):- L3 = L2.
append([X|R1], L2, [X|R3]):- append(R1, L2, R3).
```

Verwendung von
Termen in
Argumenten

Realisierung der *Wertzuweisung* durch den Unifikationsmechanismus führt dann zur Prolog-Lösung:

Wertzuweisung durch Unifikation

```
append([], L2, L2).
append([X|R1], L2, [X|R3]):- append(R1, L2, R3).
```

Auch diesmal leistet die Implementierung mehr als beabsichtigt. Das Prädikat *append* kann ebenfalls in allen möglichen Instanzierungsmustern aufgerufen werden:

```
append (?L1, ?L2, ?L3).
```

5.4 Kopf-Rest-Methode

In der Methodik des Programmierens lernt man den Umgang mit Bäumen. Dabei werden verschiedene Bearbeitungsmöglichkeiten für Bäume diskutiert. Die drei wichtigsten von insgesamt sechs möglichen sind:

WLR: für Wurzel-Links-Rechts-Bearbeitung (Preorder),
 LWR :für Links-Wurzel-Rechts-Bearbeitung (Inorder),
 LRW: für Links-Rechts-Wurzel-Bearbeitung (Postorder).

Kopf und Rest von Listen

Listen sind entartete Bäume, bei denen auf die Wurzel nur ein entarteter Teilbaum folgt. Wir nennen die Wurzel eines solchen Baumes Kopf und den entarteten Teilbaum Rest. Von den sechs Bearbeitungsmöglichkeiten bleiben bei entarteten Bäumen nur zwei übrig:

KR steht für Kopf-Rest-Bearbeitung,
 RK steht für Rest-Kopf-Bearbeitung.

Grundmuster von Listenoperationen

Eine Listenoperation kann demnach nach zwei verschiedenen Grundmustern ablaufen: Zuerst wird die Liste in Kopf und Rest aufgeteilt, dann wird entweder zunächst der Kopf und dann die Restliste oder zuerst die Restliste und dann der Kopf bearbeitet. Abschließend werden die beiden Teilergebnisse aus der Kopf- und Rest-Bearbeitung zum gesuchten Endergebnis zusammengesetzt. Diese beiden Grundmuster fassen wir unter dem Begriff *Kopf-Rest-Methode* zusammen.

Kopf-Rest-Methode

Beispiel 1: Länge einer Liste

Mit dem zweistelligen Prädikat $length(L, N)$ soll die Länge N einer Liste L ermittelt werden. Die leere Liste hat die Länge 0. Ist die Liste nicht leer, kann sie in Kopf und Rest aufgeteilt werden. Die Länge des Kopfes ist 1, die Länge der Restliste ergibt sich unter Ausnutzung von Rekursion aus $length(Rest, N1)$.

Kopf-Rest-Methode
zur
Bestimmung der
Länge einer Liste

```
length([], 0).
length(L, N):-
    L = [Kopf | Rest],
    length(Rest, N1),
    N is 1 + N1.
```

Beispiel 2: Liste umkehren

Mit dem Prädikate $reverse(Liste1, Liste2)$ soll eine Liste umgekehrt werden können. Die algorithmische Idee besteht darin, den Kopf der Liste abzutrennen, die Restliste umzukehren und abschließend den Kopf an das Ende der umgekehrten Restliste anzuhängen. Für die Liste [a,b,c,d,e] sehen die Bearbeitungsschritte also wie folgt aus:

1. [a,b,c,d,e]
2. a und [b,c,d,e]
3. a und [e,d,c,b]
4. [e,d,c,b,a]

Kopf-Rest-Methode
zur
Umkehrung einer
Liste

```
reverse(Liste1, Liste2):-
    Liste1 = [Kopf|Rest1],           /* in Kopf und Rest aufteilen */
    reverse(Rest1, Rest2),          /* Rest-Behandlung */
    append(Rest2, [Kopf], Liste2). /* Kopf-Behandlung */
reverse([], []).
```

Wegen der besonderen Rolle der *Kopf-Rest-Methode* bei Problemlösungen in Prolog werden im anschließenden Aufgabenteil viele dazu passende Übungs-

aufgaben angeboten. Sie dienen dem Ziel, die Kopf-Rest-Methode einzuschleifen.

5.5 Aufgaben

1. Welche Antworten liefern folgende Anfragen?
 - a) `[X | Y] = [rhein, elbe, wesen, mosel]`.
 - b) `[X | [weser, mosel]] = [elbe, wesen, mosel]`.
 - c) `[X | [weser, mosel]] = [rhein, elbe, wesen, mosel]`.
 - d) `[Z | Rs] = [1, 2, 3, 4, 5]`.
 - e) `[3, 4, 5] = [X | Rs]`.
 - f) `[Kopf | Rs] = [a]`.
 - g) `[X | Rs] = []`.
 - h) `[X | Rs] = [[]]`.

2. Prüfen Sie, ob eine Liste vorliegt und geben Sie gegebenenfalls die Liste in Klammernotation an:
 - a) `.[[], []]`
 - b) `.(a, b)`
 - c) `.(ich, .(und, .(du, [])))`
 - d) `.(a, .(b))`
 - e) `.(X, .(12, .(m(hugo), .(ende, []))))`
 - f) `.(.(a, .(b, [])), .(b, .(a, [])))`
 - g) `.(a, [], .(b, []))`
 - h) `.(a, .(b, .(c, [])), .(d, []))`

3. Geben Sie die folgenden Listen in der Punktnotation an:
 - a) `[1, 2]`
 - b) `[a, b+c, [d-f]]`
 - c) `[er, +, sie, =, es]`
 - d) `[[], [2, 3]]`
 - e) `[a, b, c]`
 - f) `[a, [b, c]]`

4. Geben Sie ein Prädikat *ist_Liste(L)* an, das prüft, ob eine Liste L vorliegt.

5. Schreiben Sie ein Prädikat zum zeilenweisen Ausgeben der Elemente einer Liste. Verwenden Sie dabei das *write*-Prädikat zum Schreiben und das *nl*-(new line)-Prädikat für den Zeilenvorschub.

6. Das Prädikat *gleich* soll prüfen, ob zwei Listen elementweise gleich sind.

- ?- gleich([a,b,c], [a,b,c]). Antwort: yes.
7. Mit dem Prädikat *praeFix* soll festgestellt werden können, ob eine erste Liste den Beginn einer zweiten Liste darstellt.
 ?- praeFix([a, b], [a, b, c, d]). Antwort: yes.
8. Aufgabe 7 für *suffix*:
 ?- suffix([c, d], [a, b, c, d]). Antwort: yes.
9. Geben Sie ein Prädikat an, das die Elemente einer Zahlenliste aufsummiert.
10. Definieren Sie ein Prädikat *gezaehlt*, mit dem das Vorkommen eines Elementes in einer Liste gezählt werden kann.
 ?- gezaehlt(a, [b, a, c, a, d, b], Anzahl).
 Antwort: Anzahl = 2.
11. Die Elemente einer Liste seien natürliche Zahlen. Realisieren Sie ein *inc*-Prädikat, das alle Elemente der Liste um eins erhöht.
12. Geben Sie ein Prädikat an, das ein Element E einer Liste L hinzufügt.
13. Das Verdopplungsprädikat soll eine Liste liefern, die alle Elemente der Ausgangsliste doppelt enthält.
 ?- doppel([3, 4, a], X). liefert X = [3, 3, 4, 4, a, a]
14. Entwerfen Sie ein Listenprädikat *remove(X, AltListe, NeuListe)*, das aus der gegebenen *AltListe* alle Elemente *X* entfernt.
 ?- remove(5, [3, 5, 1, 5, 9, 4, 3, 5], L).
 Antwort: L = [3, 1, 9, 4, 3]
15. Aus einer Zahlenliste sollen die negativen Zahlen gestrichen werden.
16. Erläutern Sie:
 a) member(E,L):- append(L1, [E|L2], L).
 b) lösche(E,L,R):- append(L1,[E|L2],L), append(L1,L2,R).
 c) teilliste(T, L):- append(L1, L2, L),
 append(T, L2ohneT, L2).
17. Entwerfen Sie ein Listenprädikat *subst(Alt, Neu, Liste1, Liste2)*, das in einer *Liste1* das Element *Alt* überall, wo es vorkommt, durch das Element *Neu* ersetzt.

18. Entwickeln Sie ein Prädikat, das das letzte Element einer Liste liefert.
19. Eine platte Liste soll eine Liste sein, die keine Liste als Elemente enthält, Wenn eine Liste als Element vorkommt, dann sollen deren Elemente in der gegebenen Reihenfolge in die Gesamtliste an der Stelle eingefügt werden, wo die Liste als Element stand. Es ist ein Prädikat *plaetten/2* zu entwerfen.

Beispiel: `?- plaetten([1,2,[3,4],[],[[5,6],7,[]]),L2)`
 liefert `L2 = [1, 2, 3, 4, 5, 6, 7]`

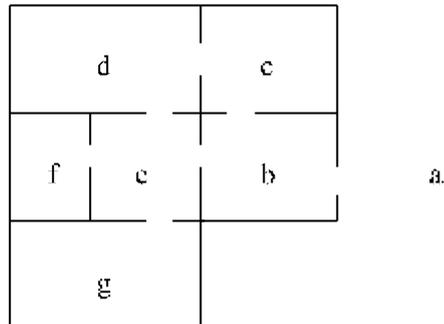
Gliedern Sie das Problem in folgende Fälle auf: Der Kopf der Liste kann keine Liste, die leere Liste oder eine nichtleere Liste sein. Beachten Sie auch den Fall, daß der Kopf nicht existiert.

20. Einige Fakten *bedeutung/2* seien gegeben. Dabei soll im 1. Argument ein englischer Begriff und im 2. Argument die deutsche Entsprechung stehen. Auf Feinheiten, Geschlecht, Fälle usw. verzichten Sie. Eine Liste mit englischen Begriffen soll mit *uebersetzen/2* in eine Liste mit den deutschen Entsprechungen übersetzt werden. Falls für einen Begriff kein Faktum existiert, bleibt der Begriff unübersetzt. Beispiel:

```
bedeutung(the, der).
bedeutung(man, mann).
bedeutung(woman, frau).
bedeutung(knows, kennt).
bedeutung(well, gut).
```

```
?- uebersetzen([the, woman, knows, Prolog, well], L)
liefert L = [der, frau, kennt, Prolog, gut]
```

21. Das Labyrinth stellt den Grundriß eines Hauses mit sechs Räumen dar. Sie kommen von draußen, also von a, herein und wollen zum Telefon.



Der Standort des Telefons wird durch das Fakt *TelefonIn(g)* beschrieben. Durch welche Türen müssen Sie gehen, um zum Telefon zu gelangen? Lassen wir dieses Problem mit einem Prolog-Programm lösen. Das Labyrinth läßt sich durch die vorhandenen Türen beschreiben:

```
tuer(a, b).
tuer(b, e).
tuer(b, c).
tuer(d, e).
tuer(c, d).
tuer(e, f).
tuer(g, e).
```

Was wir jetzt noch brauchen ist ein Prädikat *sucheWeg*, das uns einen Weg von a zum Telefon sucht. Dabei besteht jedoch die Gefahr einer Endlosschleife b, e, d, c, b, e usw. Endlosschleifen lassen sich dadurch vermeiden, daß man sich die schon besuchten Räume in einer Liste merkt. Einen weiteren Raum betritt man nur, wenn er noch nicht in der Merklste steht (not member).

6 Arithmetik

6.1 Integer-Arithmetik

Prolog ist keine besonders geeignete Sprache zur Lösung numerischer Probleme. Die Arithmetik beschränkt sich oft auf elementare Berechnungen. Dabei reicht die Verwendung von Integer-Zahlen meist aus.

In *fiæ*-Prolog gibt es nur 32-Bit-Integer-Zahlen. Damit wird der Bereich von -2.147.483.648 bis 2.147.483.647 abgedeckt. In TV-SWI-Prolog stehen auch Real-Arithmetik und die üblichen mathematischen Funktionen, wie zum Beispiel *sqrt*, *log* und *sin*, zur Verfügung. Hinweise dazu finden Sie im Referenz-Handbuch [Wie1] oder durch das System-Prädikat *help/1*.

Die Grundrechenarten schreibt man in gewohnter Weise:

$X + Y$	Addition
$X - Y$	Subtraktion
$X * Y$	Multiplikation
X / Y	Integer-Division, entspricht <code>div</code> in TV-SWI-Prolog // benutzen
$X \text{ mod } Y$	Rest der Integer-Division

Grundrechenarten

Bei den Vergleichsoperatoren müssen Sie auf die unüblichen Schreibweisen achten:

$X ::= Y$	numerisch gleich
$X =\backslash= Y$	numerisches ungleich
$X < Y$	X ist kleiner als Y
$X > Y$	X ist größer als Y
$X =< Y$	X ist kleiner als oder gleich Y, in Pascal ' <code><=</code> '
$X >= Y$	X ist größer als oder gleich Y

Vergleichsoperatoren

Wegen der Vielfalt von Vergleichsoperatoren in Prolog machen wir hier eine zusammenfassende Übersicht:

Op	arithmetische Vergleiche	Op	Term-Vergleiche
<code>::=</code>	numerisch gleich	<code>=</code>	ist unifizierbar mit
<code>=\=</code>	numerisch ungleich	<code>\=</code>	ist nicht unifizierbar mit
<code><</code>	ist numerisch kleiner als	<code>==</code>	ist identisch mit
<code>></code>	ist numerisch größer als	<code>\==</code>	ist nicht identisch mit
<code>=<</code>	ist kleiner oder gleich		

Tabelle 6-1
Zusammenstellung
von Vergleichsoperatoren

>=	ist größer oder gleich		
----	------------------------	--	--

Numerische Vergleichsoperationen lassen sich nur dann anwenden, wenn die Terme auswertbar sind, das heißt alle Operanden Zahlen oder instanziierte Variablen sind.

Kombinierte Vergleiche lassen sich in TV-SWI-Prolog einfach über *between(+Unten, +Oben, ?Wert)* ausdrücken. Zudem steht *succ(?Zahl1, ?Zahl2)* zur Verfügung.

Der *is*-Operator Neben der automatischen Auswertung arithmetischer Ausdrücke durch die Vergleichsoperatoren gibt es die Auswertung über den *is*-Operator. Dem *is* entspricht in Pascal die Wertzuweisung mit „:=“. Der *is*-Operator muß vom Unifikationsoperator „=“ unterschieden werden.

?- X = 4 + 3 ergibt die Antwort X = 4 + 3

?- X is 4 + 3 ergibt die Antwort X = 7

Achten Sie beim Minuszeichen darauf, daß es in verschiedenen Bedeutungen auftreten kann:

Bedeutung des Minuszeichens	als Vorzeichen einer negativen Zahl,	zum Beispiel: -923
	als Präfix-Operator,	zum Beispiel: X is - Y
	als Infix-Operator,	zum Beispiel: X is 4 - 5

fix-Prolog erwartet nach dem Minuszeichen ein Leerzeichen, wenn es als Operator gedacht ist. Als Vorzeichen ist es ohne Leerzeichen direkt vor die Zahl zu schreiben.

6.2 Aufgaben

1. Welche Antworten gibt der Prolog-Interpreter?
 - a) ?- 3 <= 4.
 - b) ?- X >= 5.
 - c) ?- X = 2, X < 3.
 - d) ?- X = 2, Y = 3, X < Y.
 - e) ?- X is 5 + 6.
 - f) ?- 6 * -7 is Y.
 - g) ?- 4*5 < 21
 - h) ?- 4*5 is 20.
 - i) ?- 4*5 = 20.
 - j) ?- 20 is 4*5.
 - k) ?- 20 = 4*5.

2. Geben Sie ein Prädikat *summe(X, Y, S)* an, das in S die Summe von X und Y liefert.

3. Schreiben Sie ein Prädikat *teilt(X, Y)*, was erfüllt sein soll, falls X ein Teiler von Y ist.

4. Schreiben Sie ein Prädikat *max(X, Y, Max)*, das im dritten Argument das Maximum der beiden Zahlen X und Y liefert.

5. Analysieren Sie das folgende Prädikat:


```
zahl(1).
zahl(X):- zahl(Y), X is Y + 1.
```

Was passiert, wenn man die Regel ersetzt durch:

```
zahl(X): Y is X - 1, zahl(Y).
```

6. Realisieren Sie ein Fakultätsprädikat *fak(N, F)* zur Berechnung von n!.
 Hinweis: $n! = n * (n-1)! = n * (n-1) * (n-2) * \dots * 1$.
 Beispiel: $6! = 6 * 5! = 6 * 5 * 4! = 6 * 5 * 4 * 3 * 2 * 1 = 720$

7. Definieren Sie ein Prädikat *max_liste(L, Max)* derart, daß *Max* an die größte Zahl der Liste L gebunden wird. Zur Realisierung von *max_liste* muß man Rekursion einsetzen. Wählen Sie als Terminierungsbedingung: *max_liste([X], X)*.

7 Ein- und Ausgabe

7.1 Standard-Prädikate zur Ein- und Ausgabe

Zur Steuerung der Ausgabe stehen folgende Standard-Prädikate zur Verfügung:

<code>nl</code>	Gibt das Steuerzeichen für den Zeilenvorschub aus.	Ausgabe-Prädikate
<code>tab(+Anzahl)</code>	Gibt <i>Anzahl</i> Leerzeichen aus.	
<code>put(+Ascii)</code>	Gibt das Zeichen mit dem betreffenden ASCII-Code aus.	
<code>write(+Term)</code>	Gibt <i>Term</i> in üblicher Schreibweise aus.	
<code>display(+Term)</code>	Gibt <i>Term</i> in Präfix-Notation aus.	

Für die Eingabe haben wir diese Standard-Prädikate:

<code>read(-Term)</code>	Liest einen beliebigen Prolog-Term ein.	Eingabe-Prädikate
<code>get0(-Zeichen)</code>	Liest ein einzelnes Zeichen ein und gibt dessen ASCII-Code an.	
<code>get(-Zeichen)</code>	Liest ein einzelnes Zeichen ein. Zeichen mit ASCII-Code ≤ 32 werden überlesen.	
<code>skip(+Zeichen)</code>	Liest solange Zeichen, bis der ASCII-Wert des gelesenen Zeichens mit dem Argument übereinstimmt.	

Etwas gewöhnungsbedürftig ist die Verwendung von ASCII-Codes bei der Ein- und Ausgabe einzelner Zeichen. Ohne ASCII-Tabelle wird es schwierig.

Die Quälerei mit den ASCII-Codes ist in TV-SWI-Prolog etwas entschärft. Die Sequenz `0'<Zeichen>` kann anstelle des ASCII-Codes benutzt werden. Zum Beispiel steht `0'f` für den ASCII-Code des Kleinbuchstabens `f`.

Der Vergleich von *fiæ*-Prolog und TV-SWI-Prolog zeigt sehr schön, daß ohne präzise Festlegung der Semantik einer Programmiersprache Systemprädikate ganz unterschiedlich implementiert sein können. In *fiæ*-Prolog reagiert das `get0`-Prädikat direkt auf einen Tastendruck, in TV-SWI-Prolog erst wenn man die *Eingabetaste* drückt. Dem `get0` entspricht in der Funktionsweise das `get_single_char`-Prädikat von TV-SWI-Prolog.

`0'<Zeichen>`

`get_single_char`

7.2 Anwendungen

Um etwas mit den Zeichenprädikaten vertraut zu werden, betrachten wir zwei Beispiele:

Beispiel 1: Aufbau eines Menüsystems

FRHOTEL.PL, FRKUNDEN.PL und FRBUCHEN.PL aus Kapitel 4 vorher konsultieren.

Menüsystem

```

start:-
    write('Hauptmenü'), nl, nl,
    write('1: Hotels'), nl,
    write('2: Kunden'), nl,
    write('3: Buchungen'), nl,
    write('0: Ende'), nl, nl,
    write('Ihr Wunsch: '),
    get(C),
    auswahl(C),
    start.

auswahl(0'1):- /* 0'1 ist ASCII 49 */
    listing(hotel), weiter.
auswahl(0'2):-
    listing(kunde), weiter.
auswahl(0'3):-
    listing(buchung), weiter.
auswahl(0'0):-
    fail.

weiter:-
    write('Bitte eine Taste drücken. '),
    get0(_), nl, nl.

```

Wiederholungsschleife durch Endrekursion

Was man in Pascal mit einer Repeat-Schleife macht, erledigt man in Prolog mit End-Rekursion. Für jeden Schleifendurchlauf ruft sich *start* selbst rekursiv auf. Die Schleife terminiert, wenn das *auswahl*-Prädikat fehlschlägt. Dies tritt bei Eingabe von 0 ein. Die Auswahl der Menüprozeduren erfolgt durch Unifikation mit dem Argument des *auswahl*-Prädikats.

Beispiel 2: Einlesen eines Wortes

Ein Wort soll zeichenweise eingelesen und dann aus den eingelesenen Buchstaben zusammengesetzt werden. Wir sammeln dazu in einer Liste die ASCII-Codes aller eingegebenen Buchstaben. Mit dem Systemprädikat *name(?Atom, ?ASCII-Liste)*, kann aus der Liste der ASCII-Codes das gesuchte Wort-Atom generiert werden.

das System-Prädikat *name*

Beispiele für die Verwendung von *name*:

Anfrage: `?- name(N, [65, 66, 67]).` Antwort: `N = ABC`

Anfrage: `?- name('LWB', L).` Antwort: `L = [76, 87, 66]`

Nach diesen Vorbereitungen kann implementiert werden:

```
lese_wort(Wort):-
    lese_buchstabenliste(Buchstabenliste),
    name(Wort, Buchstabenliste).

lese_buchstabenliste([Buchstabe|Buchstabenliste]):-
    lese_buchstabe(Buchstabe),
    lese_buchstabenliste(Buchstabenliste).
lese_buchstabenliste([]).

lese_buchstabe(Buchstabe):-
    get0(Buchstabe), put(Buchstabe),
    buchstabe(Buchstabe).
```

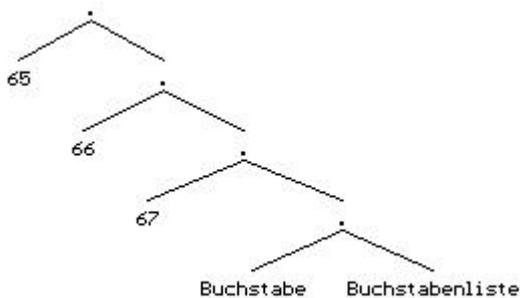


Abb. 7-1
Aufbau einer Buchstabenliste mit Hilfe einer unvollständigen Datenstruktur

In Abbildung 7-1 wird veranschaulicht, wie mit Hilfe einer unvollständigen Datenstruktur sukzessive die Buchstabenliste aufgebaut werden kann. Die Variable *Buchstabenliste* wird beim Aufruf von *lese_buchstabenliste* mit der Liste *[Buchstabe|Buchstabenliste]* unifiziert. *Buchstabe* wird mit dem gele-

senen Buchstaben instanziiert und der Vorgang mit der Variablen *Buchstabenliste* fortgesetzt.

```

Buchstaben          buchstabe(Z):-
erkennen            65 =< Z, Z =< 90.
                    buchstabe(Z):-
                    97 =< Z, Z =< 122.
                    buchstabe(142). /* Ä */
                    buchstabe(153). /* Ö */
                    buchstabe(154). /* Ü */
                    buchstabe(132). /* ä */
                    buchstabe(129). /* ü */
                    buchstabe(148). /* ö */

```

Interessant ist die Behandlung des Wortendes. Wird kein Buchstabe eingegeben, so liefern sowohl *buchstabe* als auch *lese_buchstabe fail*. Dies führt wegen des automatischen Backtrackings zur Auswahl der zweiten Klausel von *lese_buchstabenliste*. Im Argument steht die übergebene Buchstaben-Restliste, welche nun mit der leeren Liste unifiziert wird. Damit ist die Buchstabenliste komplett und kann in ein Wort umgewandelt werden.

7.3 Nutzung der Systembibliotheken von TV-SWI-Prolog

Das System-Prädikat *read* kann nur für die Eingabe von Prolog-Termen benutzt werden, d.h. daß jede Eingabe mit einem Punkt abgeschlossen werden muß und Wörter mit Großbuchstaben in Anführungsstriche zu setzen sind, damit sie nicht als Variablen gedeutet werden.

Zur einfachen Eingabe beliebiger Texte und Zahlen kann man wie zuvor mühsam zeichenweise einlesen, oder aber einfach auf die Systembibliothek von TV-SWI-Prolog zugreifen. Sie besteht aus allen PL-Dateien im /LIB-Unterverzeichnis. In Pascal müssen die genutzten Bibliotheken explizit in einer USES-Anweisung aufgeführt werden, TV-SWI-Prolog lädt Systembibliotheken automatisch, wenn ein Bibliotheksprädikat genutzt wird.

die Bibliothek
READLN.PL

Die Bibliothek *READLN.PL* bietet drei *readln*-Prädikate zum einfachen und flexiblen Eingeben von Texten und Zahlen an. Einzelheiten können Sie der Dokumentation entnehmen. Mit dem Prädikat *lese_string* kann ein beliebiger String eingelesen werden, ohne ihn in Anführungsstriche setzen und mit einem Punkt beenden zu müssen:

```
lese_string(String):-
    readln([String|_], _, _, " .,0123456789", uppercase),
    !.
lese_string('').
```

einen String einlesen

Mit dem *lese_zahl*-Prädikat liest man bequem Zahlen ein:

```
lese_zahl(Zahl):-
    readln([Zahl|_]).
```

eine Zahl einlesen

In der Bibliothek *CTYPES.PL* gibt es das Prädikat *is_alpha(+Zeichen)*, mit dem geprüft werden kann, ob ein Zeichen ein Buchstabe ist. Damit können die beiden obigen *buchstabe*-Regeln zusammengefaßt werden:

die Bibliothek CTYPES.PL

```
buchstabe(Z):- is_alpha(Z).
```

Die Umlaute werden nicht als Buchstaben erkannt, weswegen die *buchstabe*-Fakten beibehalten werden.

7.4 Formatierte Ausgaben

Für formatierte Ausgaben stellt TV-SWI-Prolog das *format*-Prädikat zur Verfügung. Um mit den ungewohnten und schwierigen Formatierungsanweisungen besser umgehen zu können, sind im folgenden vier Hilfsprädikate angegeben, mit denen typische Formatierungsprobleme gelöst werden können:

```
linksbuendig(Ausgabe, Breite):-
    format('~w~t~*|', [Ausgabe, Breite]).
```

linksbuendig

```
rechtsbuendig(Ausgabe, Breite):-
    format('~t~w~*|', [Ausgabe, Breite]).
```

rechtsbuendig

```
zentriert(Ausgabe, Breite):-
    format('~t~w~t~*|', [Ausgabe, Breite]).
```

zentriert

```
linie_zeichnen(Zeichen, Breite):-
    name(Zeichen, [Ascii]),
    format('~*t~*|', [Ascii, Breite]).
```

linie_zeichnen

Zum Beispiel gibt `rechtsbuendig('Uhrzeit', 20)` den Text `Uhrzeit rechtsbündig` in einem 20 Zeichen breiten Feld aus.

7.5 Aufgaben

- 1a) Wieso ist in Beispiel 1 die Klausel *auswahl(48):- fail.* überflüssig?
 - b) Die Schleife soll nur terminieren, wenn 0 eingegeben wird. Bei anderen Zeichen soll das Menü neu angezeigt werden. Welche Änderungen sind nötig?
 - c) Ergänzen Sie eine Klausel, so daß beim Beenden des Menüs ein *yes* kommt.
2. Entwerfen Sie ein Programm, das eingegebene Zeichen klassifiziert. Möglicher Bildschirmdialog:

```
Zeichen eingeben: Ziffer: 4
Zeichen eingeben: Großbuchstabe: A
Zeichen eingeben: Kleinbuchstabe: k
Zeichen eingeben: Sonderzeichen: 3
Zeichen eingeben: Sonderzeichen: 27
Zeichen eingeben: No
```

3. Ergänzen Sie Beispiel 2 um ein Prädikat, mit dem ein Satz eingegeben werden kann. Der Satz soll als Liste von Wort-Atomen gespeichert werden.
4. Definieren Sie ein Prädikat *plural(+Wort, -WortImPlural)*, das englische Substantive in ihren Plural konvertiert.

Anfrage: `plural(table, X).`

Antwort: `X = tables.`

8 Der Cut !

8.1 So wirkt der Cut

Der Prolog-Interpreter arbeitet nach einem festen Verfahren, das auf Resolution, Backtracking und Unifikation aufbaut. Auf dieses Verfahren kann man keinen Einfluß nehmen, aber man kann die Lösungssuche von Prolog steuern. Erstens durch Anordnung von Fakten und Regeln im Programm, weil die verwendete Resolution die Wissensbasis sequentiell durchsucht und Teilziele von links nach rechts bearbeitet. Zweitens durch Beeinflussung des Backtracking. Backtracking läßt sich mit dem fail-Prädikat erzwingen und mit dem cut-Prädikat verhindern.

Resolution

Backtracking

Der Cut ist ein nullstelliges Prädikat, das stets erfüllt ist und wie fast alle anderen Systemprädikate deterministisch ist, also nicht reerfüllt werden kann. Den Cut schreibt man mit dem Ausrufezeichen „!““. Er wird verwendet, um Zweige des Suchbaums abzuschneiden. Das reduziert die Suchdauer und führt zu effizienteren Programmen. Schneidet man Zweige ab, die keine Lösungen enthalten, so spricht man von *grünen Cuts*, bei Zweigen mit Lösungen von *roten Cuts*. Letztere sollte man vermeiden!

Grüne und rote Cuts

Nehmen wir an, zur Erfüllung eines Ziels wird das Prädikat p , zu dem es zwei Klauseln gibt, aufgerufen.

```
p:- q1,!, q2.
p:- q3.
```

Der Cut wird erreicht, wenn das Teilziel q_1 erfüllt werden kann. Das Teilziel „!““ wird stets erfüllt. Es hat als Seiteneffekt die Wirkung, daß es Prolog auf alle Entscheidungen seit dem Aufruf von p festlegt, das heißt, daß kein Backtracking mehr im Ziel q_1 sowie in der zweiten p -Regel stattfindet, wohl aber im Ziel q_2 . Der Cut wirkt also lokal und global:

Wirkung des Cut

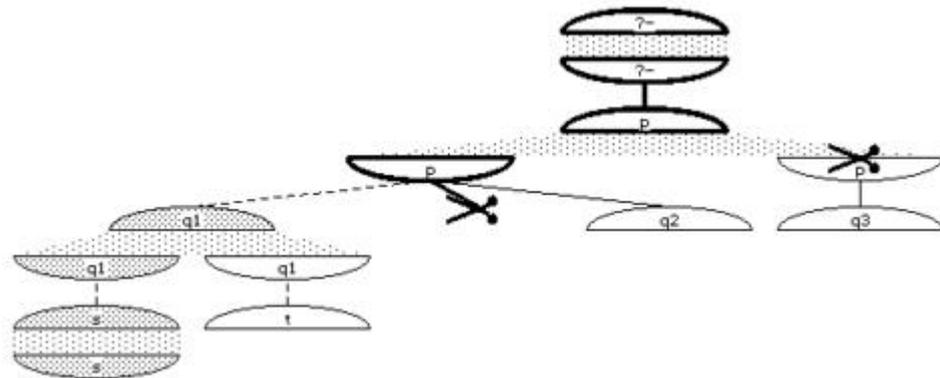
Die *lokale Wirkung* bezieht sich auf die Klausel, in der er vorkommt. Er schneidet im Beweisbaum alle noch nicht untersuchten Zweige ab, die im aktuellen Knoten links vom Cut liegen.

lokale Wirkung

Die *globale Wirkung* bezieht sich auf das Prädikat, in der er vorkommt. Die Knoten aller nachfolgenden Klauseln zum selben Prädikat werden abgeschnitten.

globale Wirkung

Abb. 8-1
Visualisierung des
Cut mit ProVisor

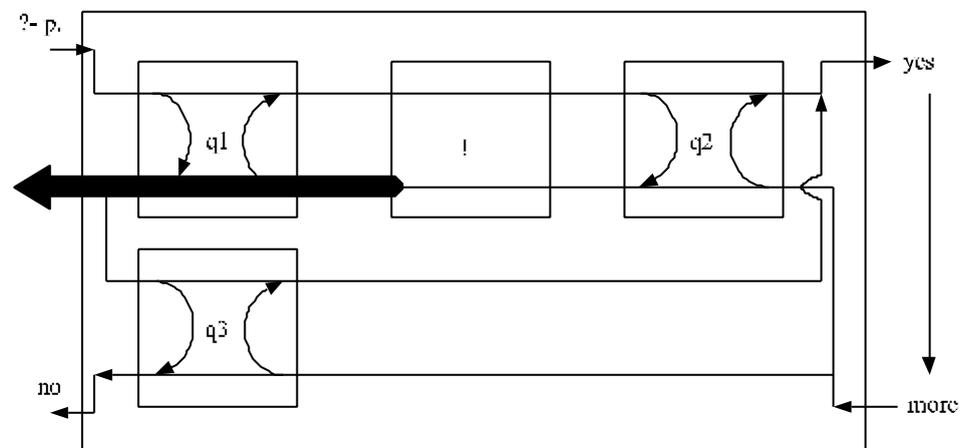


ProVisor stellt den *Cut* als Schere dar, die in zwei verschiedenen Bedeutungen benutzt wird. Zum einen vertritt das Scherensymbol das unscheinbare Ausrufezeichen und repräsentiert in dieser Funktion im Beweisbaum ein Teilziel ($q1$, \times , $q2$). Zum anderen stellt Sie die Operation des Abschneidens nachfolgender Klauseln dar (\times (p :- $q3$)).

Die lokale Wirkung wird durch Strichelung der Kanten links vom Cut dargestellt, die globale Wirkung durch das abschneidende Scherensymbol.

Veranschaulichen wir den Cut noch im Vierport-Modell.

Abb. 8-2
Visualisierung des
Cut im
Vierport-Modell



Beachten Sie dabei folgendes:

- Innerhalb der Cut-Box gibt es keine direkte Verbindung vom CALL-Eingang zum FAIL-Ausgang, weil das Teilziel ! immer gelingt und keine direkte Verbindung vom REDO-Eingang zum EXIT-Ausgang, weil der Cut deterministisch ist und somit nicht reerfüllt werden kann.

- Die lokale Wirkung bezieht sich auf die links vom Cut stehenden Teilziele. Wie der dicke *Cut-Pfeil* zeigt, sind alternative Lösungen nicht mehr möglich.
- Die globale Wirkung bezieht sich auf die unterhalb vom Cut stehenden Klauseln des Prädikats *p*. Auch sie werden durch den Cut-Pfeil von der weiteren Lösungssuche ausgeschlossen.
- Backtracking ist offensichtlich nur noch in den rechts vom Cut stehenden Teilzielen möglich, weil der Cut-Pfeil darauf keinen Einfluß nimmt.
- Der Cut-Pfeil wird erst dann gesetzt, wenn die Cut-Box durch den CALL-Eingang betreten wird.

Es besteht eine eingeschränkte Analogie zwischen dem *Cut* in Prolog und *Exit* in Pascal: die Bearbeitung des aktuellen Prädikats bzw. der aktuellen Prozedur wird beendet.

Analogie zwischen Cut und Exit

Der Cut wird von Prolog-Neulingen oft mißverstanden. Meines Erachtens hat das unter anderem folgende psychologische Ursachen:

Psychologische Probleme mit dem Cut

- Wir wissen, daß sich Gegebenheiten der Gegenwart nur auf die Zukunft, aber nicht auf die Vergangenheit auswirken. Zudem ordnen wir unbewußt einer Sequenz $q_1, !, q_3$ auch immer eine zeitliche Reihenfolge zu: erst q_1 , dann $!$ und schließlich q_3 . Der Cut wirkt auf die *Vergangenheit* q_1 .
- In konventionellen Suchbäumen schneidet der Cut nach *rechts* hin Zweige ab, welche im Programmtext aber *links* oder unterhalb des Cuts stehen. Bei Und-Oder-Beweisbäumen werden sowohl links als auch rechts Zweige abgeschnitten.

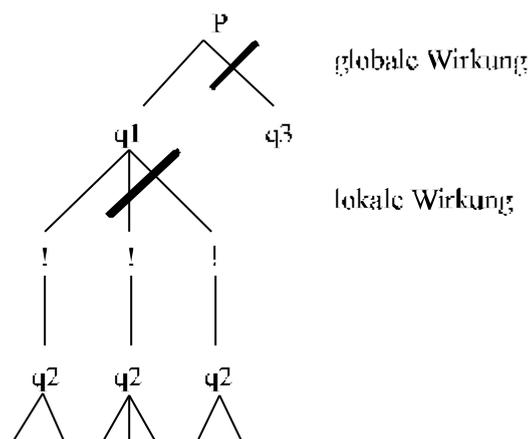


Abb. 8-3 Globale und lokale Wirkung des Cut im konventionellen Suchbaum

- Grundsätzlich gehen Menschen davon aus, daß Ursachen nur lokale Wirkungen haben. Das trifft beim Cut nicht zu.

8.2 Beispiele zum Cut

Beispiel 1: Familienbeziehungen

Wir betrachten das folgende Programm

Familien-
beziehungen

```

kind(K, E):- weiblich(E), !, mutter(E, K).
kind(K, E):- vater(E, K).
weiblich(karin).
weiblich(sina).

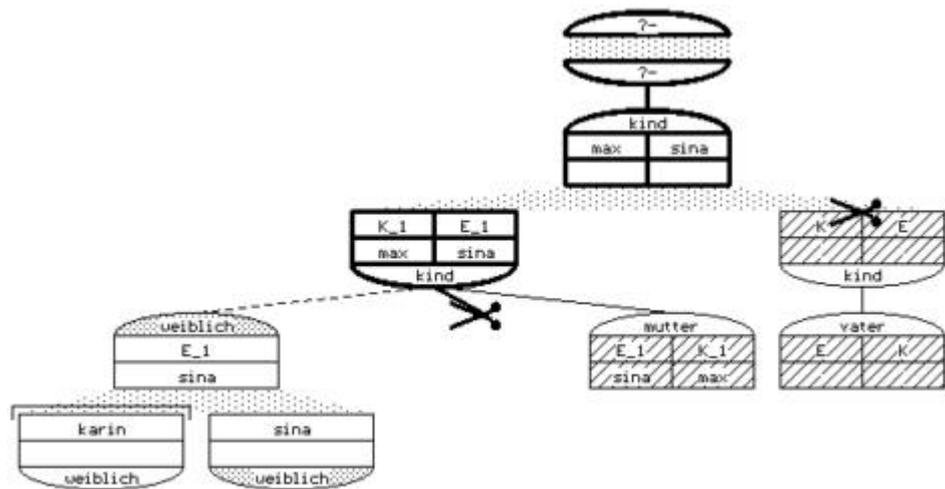
mutter(karin, maria).
mutter(sina, paul).
vater(steffen, paul).
vater(fritz, karin).
vater(paul, maria).
    
```

mit der Anfrage:

```
?- kind(max, karin).
```

Weil Karin weiblich ist, wird der *Cut* ausgeführt. Da das Teilziel *mutter(karin, max)* nicht erfüllt werden kann, würden ohne *Cut* nun alternative Lösungen für *weiblich* und danach Lösungen für *vater(karin, max)* gesucht. Der *Cut* verhindert diese nutzlose Suche. Wir wissen, daß es keine anderen Lösungen mit *vater* geben kann und teilen dieses Wissen dem Prolog-Interpreter durch den *Cut* mit.

Abb. 8-4
Wirkung des Cut
am Beispiel des
Prädikats *kind*



Beispiel 2: Vollständige Fallunterscheidung

In einem Land wird das Briefporto nach folgenden Regeln berechnet. Briefe unter 50 g kosten 1,70 DM, von 50 g bis 100 g 2,40 DM und über 100g kosten sie 3,20 DM. Mit Prolog-Regeln schreiben wir:

```
porto(Gewicht, '1,70 DM'):- Gewicht < 50.
porto(Gewicht, '2,40 DM'):- 50 =< Gewicht, Gewicht < 100.
porto(Gewicht, '3,20 DM'):- 100 =< Gewicht.
```

vollständige
Fallunterscheidung

Eine mögliche Anfrage wäre:

```
?- porto(70, Porto). mit dem Ergebnis: Porto = 2,40 DM
```

Im Trace-Modus sieht man, daß Prolog stets alle drei Klauseln auf Lösungen hin untersucht. Wir, aber leider nicht Prolog, wissen, daß es nach Ermittlung der ersten Lösung keinen Zweck hat, weiterzusuchen. Wenn ein Fall eintritt, sind wir sicher, daß keine weiteren Lösungen existieren können, da es sich um eine vollständige Fallunterscheidung handelt. Also kann man hier ruhigen Gewissens Cuts verwenden. Sie sind auf alle Fälle grün, schneiden also keine Lösungen ab.

```
porto(Gewicht, '1,70 DM'):- Gewicht < 50, !.
porto(Gewicht, '2,40 DM'):- 50 =< Gewicht, Gewicht < 100, !.
porto(Gewicht, '3,20 DM'):- 100 =< Gewicht.
```

Mit dem Cut läuft das *porto*-Prädikat effizienter. Effizienzsteigerung ist das primäre Ziel der Cut-Verwendung. Beschnittene Suchbäume können schneller durchsucht werden und benötigen weniger Speicherplatz.

Effizienzsteigerung
durch den Cut

Beispiel 3: Roter Cut

Das member-Prädikat hatten wir wie folgt programmiert:

```
member(X, [X|_]).
member(X, [_|Y]):- member(X, Y).
```

Rote Cuts

Die Anfrage `?- member(a, [a, b, c, a, d, a]).` führt zu drei Lösungen, weil *a* dreimal in der Liste enthalten ist. Unterdrücken wir daher die weiteren Lösungen mit einem Cut:

```
member(X, [X|_]) :- !.
member(X, [_|Y]) :- member(X, Y).
```

Die gleiche Anfrage liefert jetzt nur noch wie gewünscht eine Lösung. Zwei Lösungen wurden vom Lösungsbaum abgeschnitten. Es liegt also ein *roter Cut* vor. Das scheint nicht weiter schlimm zu sein. Aber wenn *member* in einem anderen Zusammenhang benutzt wird, kann das drastische Folgen haben.

Vorsicht bei roten Cuts

Die erste *member*-Version liefert bei der neuen Anfrage `?- member(X, [a, b, c, a, d, a])` sechs Lösungen, während die *member*-Version mit Cut nur noch eine Lösung liefert. Man hätte vielleicht die vier Lösungen a, b, c und d erwartet. Aber nach der ersten Lösung ist schon Schluß! Rote Cuts können daher nur mit größter Vorsicht eingesetzt werden. Deshalb sind sie auch rot. Grüne Cuts sind dagegen harmlos.

Beispiel 4: Backtracking verhindern oder erzwingen

das System-Prädikat *once*

Das Prädikat *once* dient dazu, ein Ziel nur einmal auszuführen; Backtracking wird verhindert:

```
once(Ziel) :- call(Ziel), !.
```

Gegensätzlich dazu arbeitet das Prädikat *all*, mit dem Backtracking mittels *fail* erzwungen wird, und somit alle Lösungen bestimmt werden:

```
all(Ziel) :- call(Ziel), fail.
```

Beispiel 5: Repeat-Schleife

das System-Prädikat *repeat*

Mit dem Systemprädikat *repeat* lassen sich Schleifen programmieren.

Beispiel:

Endlos-Schleife

```
echo :- repeat,
        read(X),
        write(X),
        fail.
```

Obige Schleife ist eine Endlos-Schleife, weil *repeat* beliebig oft erfüllt werden kann. Ersetzen wir *fail* durch einen *Cut*, so werden alternative Lösungen von *repeat* verworfen und somit terminiert die Schleife wieder:

```
echo:- repeat,
        read(X),
        write(X),
        !.
```

Einmal-Schleife

Jetzt wird die Schleife aber nur einmal durchlaufen. Der Cut darf erst dann erreicht werden, wenn die Endebedingung auftritt:

```
echo:- repeat,
        read(X),
        write(X),
        X == quit,
        !.
```

Schleife mit
Endebedingung

Der allgemeine Aufbau einer *repeat*-Schleife ist also gegeben durch:

```
initialisiere,
repeat,
    wiederhole etwas
Ende-Bedingung,
!,
terminiere.
```

Struktur einer
repeat-Schleife

Beispiel 6: If-Then-Else

Die in imperativen Programmiersprachen vorhandene If-Then-Else-Struktur

If-Then-Else in
Prolog

```
IF Bedingung
  THEN Anweisung1
  ELSE Anweisung2
```

läßt sich mit Hilfe des Cuts in Prolog nachbilden:

```
if_then_else(Bedingung, Anweisung1, Anweisung2):-
    Bedingung, !, Anweisung1.
if_then_else(Bedingung, Anweisung1, Anweisung2):-
    Anweisung2.
```

Gute Prolog-Interpreter kennen für die If-Then-Else Struktur den „->“ Operator. Er wird in folgender Form eingesetzt:

Der If-Then-Else-
Operator ->

```
Bedingung -> Anweisung1 ; Anweisung2
```


Beispiele für den
Einsatz des
If-Then-Else-
Operators „->“

```
max(X, Y, Z) :-
    X >= Y -> Z = X; Z = Y.
```

```
eingeben(Etwas) :-
    read(Etwas),
    (atomic(Etwas) ->
     write('Atom');
     write('Term')),
    write(' eingegeben.').
```

Beispiel 7: Cut-Fail-Kombination

System-Prädikate
mit dem Cut

Der Cut erlaubt zusammen mit dem System-Prädikat *fail* die Definition von Systemprädikaten in Prolog. Ein Beispiel ist *not*:

```
not(X) :- call(X), !, fail.
not(X).
```

Ist *X* erfüllbar, so tritt der Cut in Aktion und anschließend wird mit *fail* ein Fehlschlag erzeugt. Da der Cut alternative Lösungen für *call(X)* und für die Anwendung der zweiten Klausel verhindert, schlägt das Gesamtziel *not(X)* wie gewünscht fehl. Ist *X* mittels *call(X)* nicht erfüllbar, so tritt die zweite Klausel in Aktion und liefert die gesuchte Antwort *yes*.

8.3 Aufgaben

- 1a) Testen Sie das folgende *maximum*-Prädikat mit den Anfragen:

```
?- maximum(7, 3, X).
?- maximum(3, 7, X).
?- maximum(7, 3, 7).
?- maximum(7, 3, 3).
```

```
maximum(X, Y, X):- X >= Y, !.
maximum(X, Y, Y).
```

- b) Schreiben Sie unter Verwendung von Cut ein Prädikat

```
minimum(+X, +Y -Z)
```

das in Z das Minimum der beiden Zahlen X und Y liefert.

2. Programmieren Sie ein Prädikat *loesche(+X, +Xs, -Ys)*, das alle in der Liste Xs vorkommenden X löscht. Ys sei die X-freie Ergebnisliste. Verwenden Sie zur Effizienzsteigerung Cuts.

3. Gegeben seien die Fakten *p(1)*, *p(2)* und *p(3)*.

- a) Sagen Sie die Antworten auf die Anfragen voraus:

```
?- p(X).
?- p(X), p(Y).
?- p(X), !, p(Y).
```

- b) Was ändert sich, wenn *p(2)* durch *p(2):- !.* ersetzt wird?

4. Gesucht ist ein Programm, das unbestimmt viele Zeichen einliest und wieder ausgibt. Immer wenn ein „a“ vorkommt wird es als „b“ ausgegeben. Mit „z“ soll aufgehört werden.

- a) Analysieren Sie

```
go:- repeat,
    einZeichen(X),
    X = 122,
    !.
einZeichen(X):- get0(X), X = 97, !, put(98).
einZeichen(X):- get0(X), put(X).
```

- b) Formulieren Sie eine einwandfreie Lösung.

5. Verbessern Sie die folgende beim Backtracking in eine fatale Schleife führende Definition des Prädikates *fak(N, F)* zur Berechnung der Fakultät einer Zahl N durch Einfügung eines Cut.

```
fak(1, 1).
fak(N, F):- N1 is N - 1, fak(N1, F1), F is N * F1.
```

6. Das *menge*-Prädikat reduziert eine Liste zu einer Menge, das heißt es entfernt mehrfach auftretende Elemente. Zum Beispiel liefert *?- menge([a, b, a, c, b], X)* die Lösung $X = [a, b, c]$. Tritt jedoch ein Element mindestens dreimal in der Liste auf, so wird das richtige Ergebnis beim Backtracking mehrfach ausgegeben.

```
menge([X], [X]).
menge([X|Y], Z):- member(X, Y), menge(Y, Z).
menge([X|Y], [X|Z]):- not member(X, Y), menge(Y, Z).
```

- a) Woran liegt das?
- b) Wie kann man das mit Hilfe eines Cut vermeiden?

Mit selbst definierten Systemprädikaten können Sie nicht vorhandene Systemprädikate in ProVisor ergänzen. Die Eintragungen nehmen Sie in der ProVisor-Datei SYSTEM.PRO vor.

7. Definieren Sie *nonvar* unter Verwendung von *var* und *Cut-Fail*.
8. Definieren Sie das System-Prädikat „\=" unter Verwendung von „=" und *Cut-Fail*.

9 Unifikation

In diesem Kapitel wird der Unifikationsmechanismus detailliert untersucht. Er bildet neben dem Resolutions- und Backtrackingverfahren das dritte wesentliche Verfahren, auf dem jeder Prolog-Interpreter aufbaut.

9.1 Unifikation als Teil des Prolog-Beweisers

Zunächst betrachten wir die Unifikation anhand zweier bekannter Beispiele.

9.1.1 Familienbeziehungen

1. `mutter(karin, maria).`
2. `mutter(sina, paul).`
3. `vater(steffen, paul).`
4. `vater(fritz, karin).`
5. `elternteil(E, Kind):- vater(E, Kind).`
6. `elternteil(E, Kind):- mutter(E, Kind).`

Unifikation am
Beispiel der Familienbeziehungen

Anfrage: `?- elternteil(X, paul).`

Das Verfahren, nachdem der Prolog-Interpreter eine Anfrage bearbeitet, haben wir kennengelernt. Ein zentraler Bestandteil dieses Verfahrens ist die Unifikation. Das Ziel der Unifikation besteht darin, durch geeignete Variablenbindungen zwei Terme gleichzumachen. Ist eine Unifikation möglich, so wird sie als Substitution der betroffenen Variablen ausgedrückt.

Terme
gleichmachen

Substitution

Die Anfrage läßt sich mit den ersten vier Klauseln des Programms nicht gleichmachen, da die Funktoren *vater* bzw. *mutter* sich vom Funktor *elternteil* der Anfrage unterscheiden. Die fünfte Klausel kann mit der Anfrage unifiziert werden: *Kind* wird an die Konstante *paul* gebunden und *E* an die Variable *X*. Die Bindung zweier freier Variabler bedeutet, daß eine anschließende Bindung einer der beiden Variablen an einen Term gleichzeitig die andere Variable an diesen Term bindet.

Bindung freier
Variabler

Die Anfrage und die fünfte Klausel

```
elternteil(X, paul)
elternteil(E, Kind)
```

können somit durch die Substitution $E = X$ und $Kind = paul$ unifiziert werden. Im nächsten Schritt wird die *elternteil*-Regel angewendet und das ursprüngliche Ziel durch das neue Ziel ersetzt:

?- vater(E, paul).

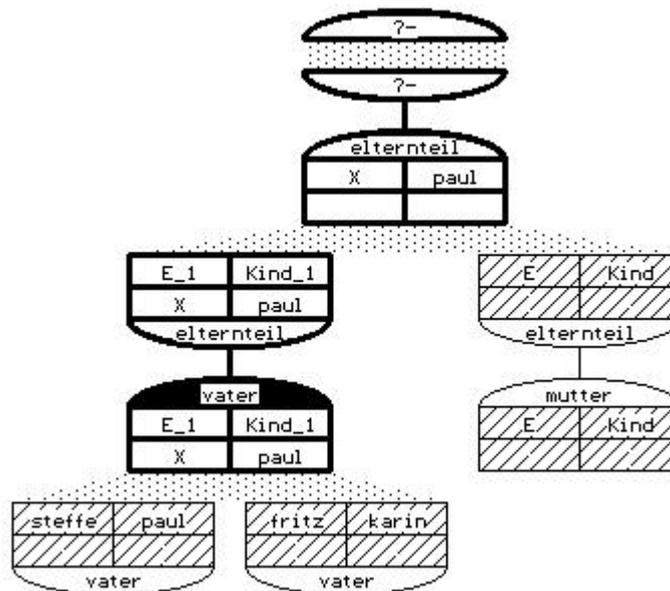
Die beiden Klauseln 1 und 2 können nicht mit dem Ziel unifiziert werden, weil die Funktoren *vater* bzw. *mutter* verschieden sind. Die Unifikation gelingt bei der dritten Klausel:

vater(E, paul)
 vater(steffen, paul)

lassen sich durch die Substitution $E = steffen$ gleichmachen. Damit wird, wie schon erläutert, gleichzeitig X an *steffen* gebunden. Das System hat somit nach zwei erfolgreichen Unifizierungen die Lösung $X = steffen$ gefunden.

Was eben durch viele Worte ausgedrückt wurde, soll nun anhand zweier Bilder nochmals dargestellt werden. In Bild 9-1 sehen wir, wie durch die Unifikation der Anfrage mit der ersten *elternteil*-Klausel die Variable E mit der Anfrage-Variablen X unifiziert und die Variable *Kind* mit der Konstanten *paul* instanziiert wurde.

Abb. 9-1
 Instanzierung von
 Variablen bei einer
 Unifikation



In Bild 9-2 wird das Teilziel $vater(E, paul)$ mit dem ersten $vater$ -Fakt unifiziert. Dabei wird die Variable E mit $steffen$ instanziiert. Da X mit E unifiziert ist, wird dadurch auch die Variable X mit $steffen$ instanziiert:

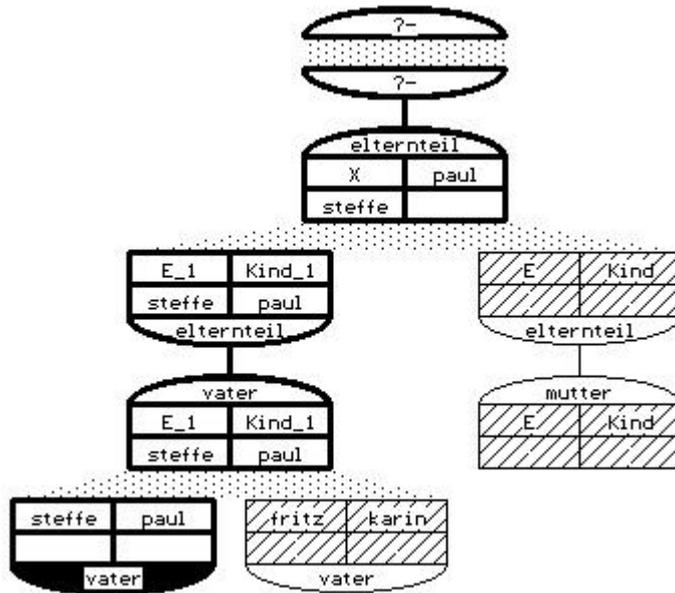


Abb. 9-2
Ein Unifikations-
schritt

9.1.2 Listenzerlegung mittels append

Append haben wir wie folgt implementiert:

```
append([X|L1], L2, [X|L3]):- append(L1, L2, L3).
append([], L, L).
```

Bei der Anfrage $?- \text{append}(X, Y, [a, b, c, d])$. versucht der Prolog-Interpreter die Anfrage-Klausel mit den Programm-Klauseln zu unifizieren. Durch welche Variablensubstitutionen läßt sich die Anfrage mit der ersten Klausel gleichmachen?

Unifikation bei einer *append*-Anfrage

```
append(X, Y, [a, b, c, d]).
append([X|L1], L2, [X|L3])
```

Da X in der Anfrage und in der Regel auftritt, muß man zunächst eine Variablenumbenennung vornehmen. Ein Prolog-Interpreter macht dies immer bei der Auswahl und Anwendung einer Klausel.

Variablenumbenennung

```
append(X, Y, [a, b, c, d])
append([X_1|L1_1], L2_1, [X_1|L3_1])
```

Die beiden Klauseln lassen sich dann durch folgende Substitution unifizieren:

$$X = [X_1|L1_1], Y = L2_1, X_1 = a, L3_1 = [b, c, d]$$

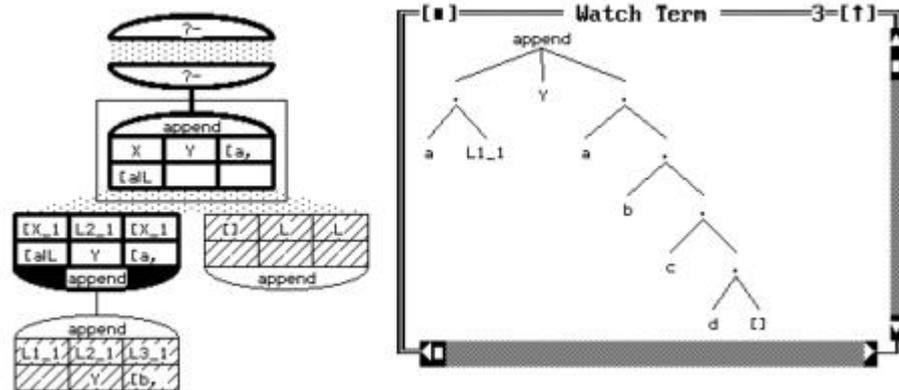
Man erhält durch die Substitution den Term:

$$\text{append}([a|L1_1], L2_1, [a|[b, c, d]])$$

Generierung einer Teillösung durch Unifikation

Im linken Teilbild sehen wir den Beweisbaum nach der ersten Unifikation. Im rechten Teilbild ist der aktuelle Knoten im Detail als dynamischer Watch-Term dargestellt. Man sieht daran sehr schön, wie durch Unifikation die erste Teillösung entstanden ist. Wenn es eine Lösung für X gibt, dann muß X eine Liste der Gestalt [a|L1_1] sein.

Abb. 9-3
Dynamische Watch-Terme in ProVisor



Erzeugung dynamischer Watch-Terme

Um dynamische Watch-Terme mittels ProVisor darzustellen, wechseln Sie mit *Alt-2* in das Hauptfenster und klicken mit der Maus den gewünschten Knoten an. Sie können natürlich nur aktive Knoten auswählen, also solche die auf dem aktuellen Pfad liegen. Die Auswahl können Sie auch über die Tastatur vornehmen. Bei gedrückter *Shift-Taste* wählen Sie den Knoten mit den Pfeiltasten aus.

Wir setzen jetzt die Lösungssuche fort. Die Anfrage wird nach erfolgreicher Unifikation gemäß der ersten Klausel durch das neue Ziel ersetzt:

$$\text{append}(L1_1, L2_1, [b, c, d])$$

Mit dem neuen Teilziel wird in der gleichen Weise wie eben beschrieben verfahren. Die Unifikation mit $\text{append}([X_2|L1_2], L2_2, [X_2|L3_2])$ liefert die Substitution

$$L1_1=[X_2|L1_2], L2_1=L2_2, X_2=b, L3_2=[c, d]$$

also insbesondere $L1_1 = [b|L1_2]$, was einen weiteren Schritt im Aufbau des Lösungsterms für X darstellt. Dies sehen wir auch im folgenden Bild:

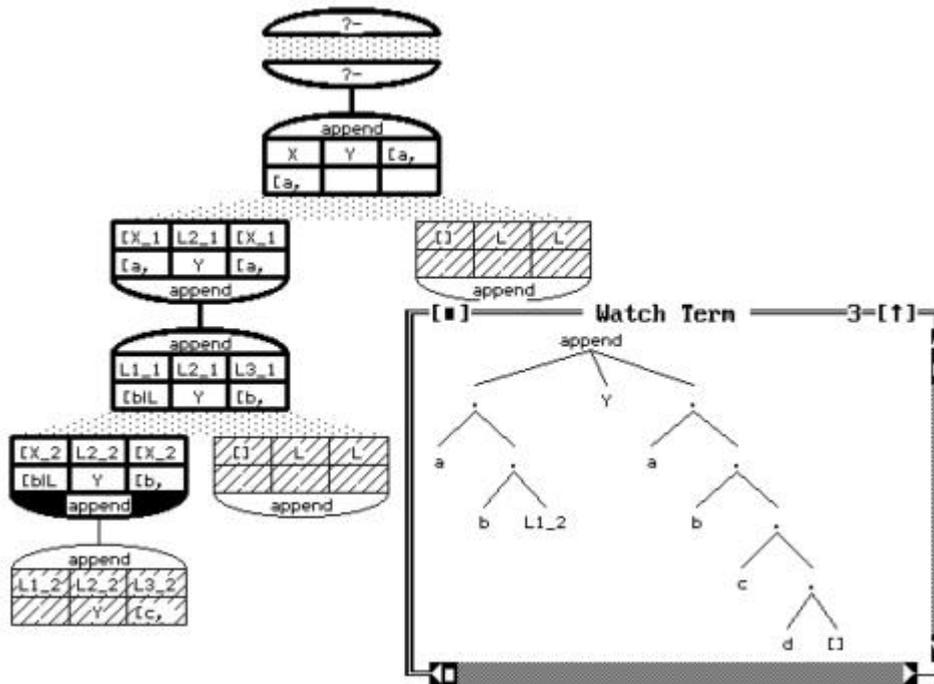


Abb. 9-4
Termgenerierung durch mehrfaches Unifizieren

Nach zwei weiteren erfolgreichen Unifikationen erreicht man das Ziel $append(L1_4, L2_4, [])$. Zu diesem Zeitpunkt haben wir folgende Situation vorliegen:

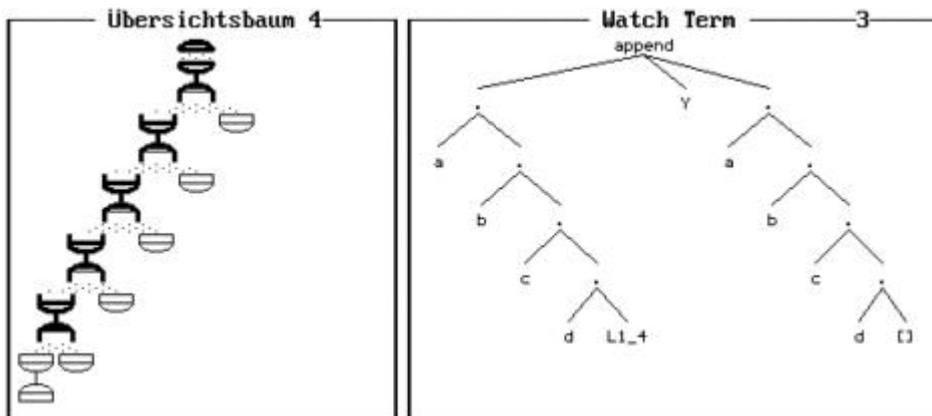


Abb. 9-5
Visualisierung von append kurz vor der ersten Lösung

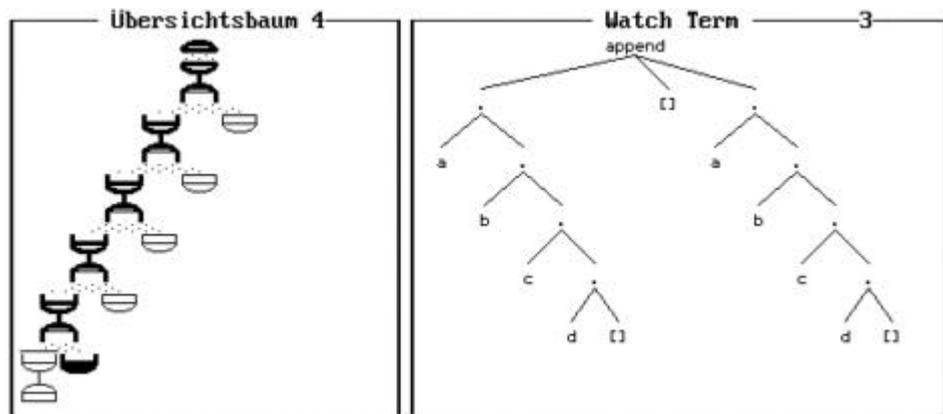
Das neue Ziel kann mit der ersten *append*-Klausel nicht unifiziert werden. Die Unifikation scheitert beim dritten Argument. [] und $[X_5|L3_5]$ können durch

keine Variablensubstitution gleichgemacht werden. Die eine Liste ist leer, die andere Liste hat mindestens ein Element. Die Unifikation ist aber mit dem Kopf der zweiten Klausel möglich:

```
append(L1_4, L2_4, [])
append([], L_1, L_1)
```

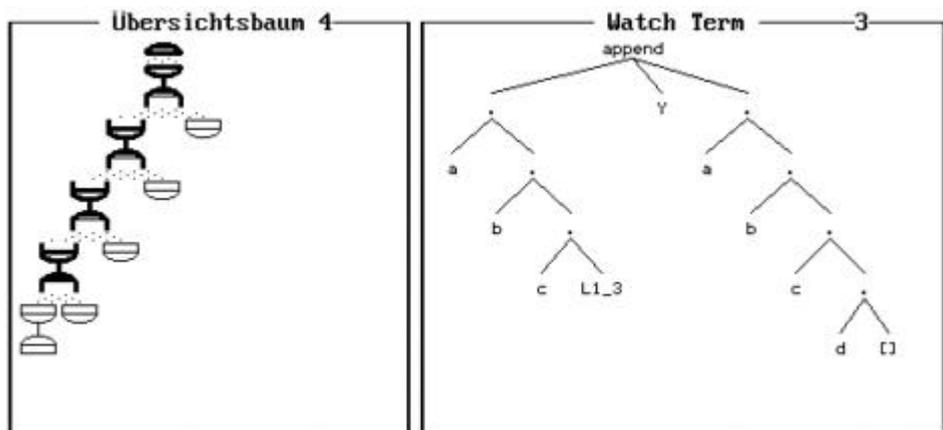
Diese beiden Terme werden durch die Substitution $L1_4 = [], L2_4 = L_1, L_1 = []$ gleichgemacht. Damit ist der Lösungsterm für X komplett aufgebaut und wegen $Y = L2_1 = L2_2 = L2_3 = L2_4 = L_1 = []$ ist jetzt auch Y gefunden. Da die zweite append-Klausel ein Fakt ist, ist die erste Lösung mit $X = [a, b, c, d]$ und $Y = []$ gefunden.

Abb. 9-6
Visualisierung
von append
1. Lösung



Fordert man weitere Lösungen an, so führt der Prolog-Interpreter Backtracking durch. Es geht zu dem Knoten im Beweisbaum zurück, an dem noch weitere Alternativen zur Verfügung stehen. Dies ist der Knoten, bei dem zum dritten Mal die erste append-Klausel ausgewählt wurde.

Abb. 9-7
Visualisierung
von append
kurz vor der
zweiten Lösung

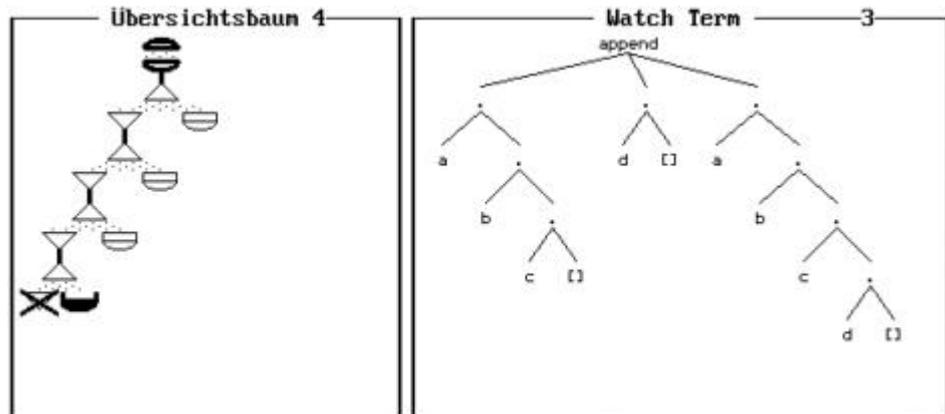


Das dortige Teilziel kann auch mit der zweiten append-Klausel unifiziert werden:

```
append(L1_3, L2_3, [d])
append([], L_1, L_1)
```

Die Unifikation liefert $L1_3 = []$, $L2_3 = L_1$ und $L1_1 = [d]$. Wegen $X = [a, b, c|L1_3]$ ergibt sich $X = [a, b, c|[]] = [a, b, c]$ und wegen $Y = L2_1 = L2_2 = L2_3 = L_1 = [d]$ haben wir $Y = [d]$. Damit ist die zweite Lösung gefunden: $X = [a, b, c]$ und $Y = [d]$.

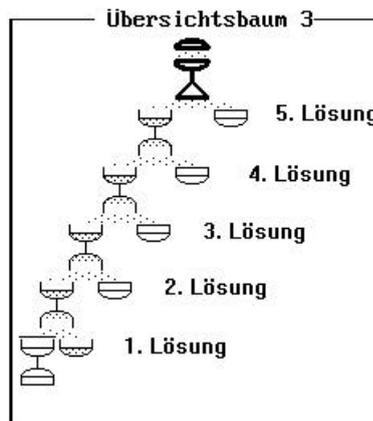
Abb. 9-8
Visualisierung
von append
2. Lösung



Backtracking liefert drei weitere Lösungen:

```
X = [a, b], Y = [c, d]; X = [a], Y = [b, c, d] und
X = [], Y = [a, b, c, d]
```

Im Übersichtsbaum können wir abschließend die Lage der fünf Lösungen nochmals kennzeichnen:



5. Lösung: $x = [], Y = [a, b, c, d]$
4. Lösung: $x = [a], Y = [b, c, d]$
3. Lösung: $x = [a, b], Y = [c, d]$
2. Lösung: $x = [a, b, c], Y = [d]$
1. Lösung: $x = [a, b, c, d], Y = []$

Abb. 9-9
 Visualisierung
 von append
 Übersicht zu allen
 Lösungen

Die erste Lösung entsteht dadurch, daß viermal die erste und dann die zweite Klausel ausgewählt wird. Die zweite Lösung entsteht durch dreimalige Auswahl der ersten und dann der zweiten Klausel. Die letzte Lösung ergibt sich durch direkte Anwendung der zweiten Klausel. Würde man die Reihenfolge der Klauseln umdrehen, so entstünden auch die Lösungen in umgekehrter Reihenfolge.

Die hier benutzte Darstellung war sehr detailreich. Sie eignet sich, um einmal im Detail die Arbeitsweise des Prolog-Interpreters und den Unifikationsmechanismus nachzuvollziehen. Daher hat diese Darstellung exemplarischen Charakter. In der Regel kann man sich mit kompakteren Darstellungen begnügen, wie Sie beispielsweise das spur-Prädikat liefert.

die Spur des
 append-Prädikats

```
?- spur(append(X, Y, [a, b, c, d])).
append([a|L1_5],Y_1,[a,b,c,d])
  append([b|L1_10],L2_5,[b,c,d])
    append([c|L1_15],L2_10,[c,d])
      append([d|L1_20],L2_15,[d])
        append([],[],[])
X = [a,b,c,d] Y = []
```

```
      append([], [d], [d])
X = [a,b,c] Y = [d]
    append([], [c,d], [c,d])
X = [a,b] Y = [c,d]
  append([], [b,c,d], [b,c,d])
X = [a] Y = [b,c,d]
append([], [a,b,c,d], [a,b,c,d])
X = [] Y = [a,b,c,d]
No
```

Definition der
Unifikation

9.2 Definition und Unifikationsregeln

Unter *Unifikation* (*Gleichmachen, Zur-Deckung-Bringen, Matching*) zweier Terme versteht man die Ersetzung der in ihnen vorkommenden Variablen durch Terme derart, daß die in ihnen vorkommenden Variablen als Zeichenfolgen gleich sind. Zwei Terme heißen unifizierbar, wenn sie durch geeignete Ersetzung der Variablen in diesem Sinne gleich gemacht werden können.

Es gibt im allgemeinen mehrere Möglichkeiten, zwei Terme durch Ersetzung der Variablen gleichzumachen (zu unifizieren). Beispielsweise kann man die Terme $p(X)$, $p(f(Y))$ einmal durch die Ersetzung $X = f(Y)$, aber auch durch die Ersetzung $X = f(a)$, $Y = a$ unifizieren.

Die allgemeinen Regeln, um zu entscheiden, ob zwei Terme S und T unifizieren, lauten folgendermaßen:

Unifikationsregeln

1. Sind S und T Konstanten, dann unifizieren S und T nur, wenn sie das gleiche Objekt sind.
2. Ist S eine Variable und T beliebig, dann unifizieren beide; S wird zu T substituiert. Falls T eine Variable ist, wird T durch S substituiert.
3. Sind S und T zusammengesetzte Terme, dann unifizieren Sie nur, wenn
 - (a) sie den gleichen Funktor haben, und
 - (b) alle einander entsprechenden Komponenten unifizieren.

Tabelle 9-1
Zusammenfassung möglicher Unifikationspartner

unifiziert mit	Konstante $K1$	Variable $V1$	zusammengesetzter Term $T1$
Konstante $K2$	falls $K1 = K2$	ja, mit $V1 = K2$	nein
Variable $V2$	ja, mit $V2 = K1$	ja, mit $V1 = V2$	ja, mit $V2 = T1$
Term $T2$	nein	ja, mit $V1 = T1$	falls Regel 3 erfüllt

9.3 Ein Unifikations-Algorithmus

Der folgende Unifikations-Algorithmus stammt aus [Ste1]. Er zeigt, wie man mit Hilfe eines Kellers unifizieren kann.

Eingabe: Zwei zu unifizierende Terme S und T

Ausgabe: Die allgemeinste Substitution U von S und T , oder Scheitern

Unifikations-
Algorithmus

```

Initialisiere die Substitution  $U$  zu leer, den Keller mit der
Gleichung  $S = T$ , und Scheitern mit falsch.
solange der Keller nicht leer ist
  hole  $X = Y$  vom Keller
  falls
     $X$  eine Variable ist, die nicht in  $Y$  vorkommt
      ersetze  $X$  im Keller und in  $U$  durch  $Y$ 
      füge  $X = Y$  zu  $U$  hinzu
     $Y$  eine Variable ist, die nicht in  $X$  vorkommt
      ersetze  $Y$  im Keller und in  $U$  durch  $X$ 
      füge  $Y = X$  zu  $U$  hinzu
     $X$  und  $Y$  identische Konstanten oder Variablen sind
      fahre fort
     $X$  gleich  $f(X_1, \dots, X_n)$  und  $Y$  gleich  $f(Y_1, \dots, Y_n)$  für einen Funktor
     $f$  und  $n > 0$  ist
      schreibe  $X_i = Y_i$ ,  $i = n, \dots, 1$  auf den Keller
  ansonsten
    Scheitern := wahr
wenn Scheitern,
  dann gebe Scheitern aus,
ansonsten gebe  $U$  aus.

```

Beispiel zum
Unifikations-
Algorithmus

Wir betrachten als Beispiel die Unifikation der Terme $append([a, b], [c, d], Ls)$ und $append([X/Xs], Ys, [X/Zs])$. Der Keller wird mit der Gleichung

$$append([a, b], [c, d], Ls) = append([X/Xs], Ys, [X/Zs])$$

initialisiert. Die Gleichung wird wieder vom Keller geholt und untersucht. Da beide Terme den gleichen Funktor *append* und die gleiche Stelligkeit 3 haben, schreiben wir drei Gleichungen für die Argumente auf den Keller:

Initialisierung des
Kellers

$$[a, b] = [X|Xs], [c, d] = Ys \text{ und } Ls = [X|Zs].$$

oberste
Kellergleichung

Die oberste Gleichung $[a, b] = [X|Xs]$ wird vom Keller geholt. Diese beiden zusammengesetzten Terme haben den gleichen Funktor „|“ und die Stelligkeit 2, und so werden zwei Gleichungen, $[b] = Xs$ und $a = X$ auf den Keller geschrieben. Beim Fortfahren wird die Gleichung $a = X$ vom Keller geholt. Für sie trifft der zweite Fall im Unifikations-Algorithmus zu: X ist eine Variable, die in der Konstanten a nicht vorkommt. Alle Vorkommen von X im Keller werden durch a ersetzt. Davon ist die Gleichung $Ls = [X|Zs]$ betroffen, welche in $Ls = [a|Zs]$ geändert wird. Die Gleichung $X = a$ wird zu der anfänglich leeren

zweite
Kellergleichung

Substitution hinzugefügt, und der Algorithmus fährt fort.

Die nächste vom Keller geholte Gleichung ist $[b] = Xs$. Für sie trifft wiederum der zweite Fall zu. $Xs = [b]$ wird zu der Menge der Substitutionen hinzugefügt, und der Keller wird nach Vorkommen von Xs durchsucht. Es gibt keine, und die nächste Gleichung wird abgeholt.

dritte
Kellergleichung

Der zweite Fall deckt auch $[c, d] = Ys$ ab. Die Substitution $Ys = [c, d]$ wird zur Menge der Unifikationen hinzugefügt, und die letzte Gleichung $Ls = [a|Zs]$ wird abgeholt. Sie wird vom symmetrischen ersten Fall behandelt. Ls kommt in $[a|Zs]$ nicht vor, daher wird die Gleichung zum Unifikator hin-

Substitution als
Ergebnis der Unifi-
kation

zugefügt, und der Algorithmus terminiert erfolgreich. Der Unifikator ist

$$\{X = a, Xs = [b], Ys = [c, d], Ls = [a|Zs]\}$$

Die vom Unifikator erzeugte gemeinsame Instanz lautet:

$$\text{append}([a, b], [c, d], [a|Zs]).$$

9.4 Aufgaben

Das einfache Gleichheitszeichen „ $=$ “ ist der Unifikationsoperator von Prolog. Die Anfrage $?- T1 = T2$. ist also genau dann erfüllbar, wenn sich die beiden Terme unifizieren lassen.

1. Unifizieren Sie falls möglich

- a) `lehrer(meier, X, Y) = lehrer(Z, mathe, 10)`.
- b) `vorname(hugo) = U`.
- c) `jagt(U, katze) = jagt(katze, maus)`.
- d) `buch(autor(Name, Vorname), Titel) = buch(autor(X, franz), Y)`.
- e) `buch(autor(Name, Vorname), Titel) = buch(autor, X, Y, amerika)`.
- f) `ausleihe(A, B, C, D) = ausleihe(_, _, datum(28,7,89), datum(T, M, J))`.
- g) `ausleihe(A, B, C, D) = rueckgabe(A, B, C, D)`.
- h) `rueckgabe(A, B, datum(12,8,89)) = rueckgabe(X, buch(PROLOG, bratko), Y)`.
- i) `rueckgabe(A, B, datum(A, B, A)) = rueckgabe(X, buch(PROLOG, bratko), Y)`.
- j) `datum(T, M, 1983) = datum(T1, mai, J1)`.
- k) `punkt(A, B) = punkt(1, 2)`.
- l) `punkt(A, B) = punkt(X, Y, Z)`.
- m) `plus(2, 2) = 4`.
- n) `A=h(A)`
- o) `dreieck(punkt(-1, 0), P2, P3) = dreieck(P1, punkt(1, 0), punkt(0, Y))`.
- p) `append([b], [c, d], L) = append([X|Xs], Ys, [X|Zs])`.
- q) `hanoi(s(N), A, B, C, Zs) = hanoi(s(s(0)), a, b, c, Xs)`.

2. Die arithmetischen Operatoren sind alle linksassoziativ. $8-5-3$ ist also als $(8-5)-3$ zu verstehen. Beachten Sie diesen Hinweis bei folgenden Anfragen. Sind Sie sich über den Aufbau eines Terms nicht sicher, so zeichnen Sie am besten die Strukturen als Bäume auf. Unifizieren Sie:

- a) $X - Y = 8 - 5 - 3$.
- b) $X / Y = 8 / 4 / 2$.
- c) $X - Y = 3 + 4 - 5 - 2$.
- d) $X + Y = 3 + 4 - 5 - 2$.

- 3a) Analysieren Sie das folgende Prädikat zum Unifizieren. Informieren Sie sich in Kapitel 15 über den *univ*-Operator „=..“.

```
unify(Term1, Term2):-
    var(Term1), var(Term2), Term1 = Term2.
unify(Term1, Term2):-
    var(Term1), nonvar(Term2), Term1 = Term2.
unify(Term1, Term2):-
    nonvar(Term1), nonvar(Term2), Term2 = Term1.
unify(Term1, Term2):-
    nonvar(Term1), nonvar(Term2),
    Term1 =.. [Funktork|Argumentliste1],
    Term2 =.. [Funktork|Argumentliste2],
    unify_list(Argumentliste1, Argumentliste2).

unify_list([], []). /* gleiche Stelligkeit! */
unify_list([Arg1|Rest1], [Arg2|Rest2]):-
    unify(Arg1, Arg2),
    unify_list(Rest1, Rest2).
```

- b) Ergänzen Sie das *unify*-Prädikat um ein drittes Argument. Dieses Argument soll der Unifikator werden. Organisieren Sie den Unifikator als Liste mit Einträgen der Art *sub(Variable, Wert)*. Für unser Beispiel aus Kapitel 8.3 soll also folgender Unifikator ermittelt werden.

```
[sub(X,a), sub(Xs,[b]), sub(Ys,[c,d]), sub(Ls,[a|Zs])]
```

Erläutern Sie, warum Prolog nicht das gewünschte Ergebnis liefert.

4. Entwerfen Sie ein Prädikat

```
substituiere(+Alt, +Neu, +AlterTerm, -NeuerTerm)
```

das in *AlterTerm* alle Vorkommen von *Alt* durch *Neu* ersetzt und damit *NeuerTerm* erzeugt. *Alt*, *Neu* und *AlterTerm* sollen grund sein, das heißt keine Variablen enthalten. Variablen würden einige Schwierigkeiten bringen.

Beispiele:

```
?- substituiere(katze,hund,besitzt(petra, katze), X).
liefert: X = besitzt(petra, hund)
```

```
?- substituiere(c, a*b, f(c, a+c)-b/c, X).
liefert: X = f(a*b, a+a*b)-b/(a*b)
```


10 Symbolisches Differenzieren

Symbolisches Differenzieren mit Prolog ist ein Paradebeispiel für *deklaratives Programmieren* und für *Künstliche Intelligenz*. Wenn die nötigen Mathematikkenntnisse vorhanden sind, ergibt die Bearbeitung dieses Themas einen schönen und ertragreichen Anlaß, über das Verhältnis Mensch - Maschine und über natürliche sowie künstliche Intelligenz nachzudenken.

Deklaratives
Programmieren und
Künstliche Intelli-
genz

10.1 Ableitungsregeln

Wir konzipieren ein Prädikat *ableiten*, mit dem beliebige Funktionsterme nach einer Variablen differenziert werden können. Als Schnittstelle von *ableiten* legen wir fest:

Prädikat zum Ab-
leiten von Termen

```
ableitung(+Term,+Ableitungsvariable,-AbgeleiteterTerm).
```

Beispiele für Anfragen und Antworten sind:

```
?- ableitung(x+1, x, A).      liefert: A = 1+0
?- ableitung(x*x-2, x, A).   liefert: A = x*1+1*x-0
```

Die Implementierung beschränkt sich auf die Wiedergabe der aus der Mathematik bekannten Ableitungsregeln in Form von Prolog-Klauseln:

Ableitungsregeln

```
ableitung(Konstante, X, 0):-
    atomic(Konstante),
    Konstante \== X, !.
ableitung(X, X, 1):- !.
ableitung(-T, X, -Ta):- !,
    ableitung(T, X, Ta).
ableitung(T1 + T2, X, T1a + T2a):- !,
    ableitung(T1, X, T1a),
    ableitung(T2, X, T2a).
ableitung(T1 - T2, X, T1a - T2a):- !,
    ableitung(T1, X, T1a),
    ableitung(T2, X, T2a).
ableitung(Konstante * T, X, Konstante * Ta):-
    atomic(Konstante),
    Konstante \== X, !,
    ableitung(T, X, Ta).
```

Konstantenregel

Ableitung von x
Vorzeichenregel

Summenregel

Differenzregel

Faktorregel

Produktregel	<pre>ableitung(T1 * T2, X, T1 * T2a + T2 * T1a):- !, ableitung(T1, X, T1a), ableitung(T2, X, T2a).</pre>
Quotientenregel	<pre>ableitung(T1 / T2, X, (T1a * T2 - T1 * T2a)/(T2*T2)):- !, ableitung(T1, X, T1a), ableitung(T2, X, T2a).</pre>
Ableitung von Grundfunktionen	<pre>ableitung(sin(X), X, cos(X)). ableitung(cos(X), X, -sin(X)). ableitung(ln(X), X, 1/X). ableitung(e(X), X, e(X)).</pre>

Mit dem Aufschreiben der Prolog-Klauseln für die Ableitungsregeln ist im Prinzip alles schon erledigt. Wir haben dem Prolog-Interpreter in deklarativer Form das relevante Wissen mitgeteilt. Damit ist er in der Lage, beispielsweise folgende Aufgabe zu lösen:

```
?- ableitung(x*(sin(x)/ln(x)),x,A).
```

Lösung: $A = x * ((\cos(x) * 1 * \ln(x) - \sin(x) * (1/x * 1)) / (\ln(x) * \ln(x))) + \sin(x) / \ln(x) * 1$

Die Lösung kommt zwar nicht in der Form, wie wir es gewohnt sind, aber sie ist richtig. Wenn man bedenkt, welche Schwierigkeiten Grundkurs-Schüler oft mit solchen Aufgaben haben, so ist es doch sehr erstaunlich, daß die alleinige Mitteilung der verfügbaren Ableitungsregeln den Rechner befähigt, komplizierte Ableitungen zu berechnen. Was ist also das Geheimnis dieser intelligenten Maschinen? Resolution, Backtracking und Unifikation!

intelligente Maschinen

Zur Lösung der Anfrage `ableitung(x*(sin(x)/ln(x)),x,A)` wird die Wissensbasis nach einem Regelkopf durchsucht, der mit dem Anfrageterm unifizierbar ist. Dies trifft zum erstenmal bei der Konstantenregel zu. Da der Term $x * (\sin(x) / \ln(x))$ keine Konstante ist, setzt automatisch Backtracking ein. Als nächstes ist der Regelkopf für die Faktorregel mit der Anfrage unifizierbar. Da die Prolog-Konstante x nicht von der Ableitungsvariablen verschieden ist, setzt abermals Backtracking ein, wodurch als nächstes die Produktregel ausgewählt wird. Sie ist anwendbar und reduziert gemäß dem Prinzip *Teile und Herrsche* die Lösung des gestellten Problems auf die Ableitung der beiden Faktoren x und $\sin(x) / \ln(x)$.

durch Resolution, Backtracking und Unifikation

Mustererkennung durch Unifikation, systematische Untersuchung aller Alternativen

Das Unifikationsverfahren wird zur Mustererkennung eingesetzt, mit dem die auf einen Ableitungsterm anwendbaren Ableitungsregeln ermittelt werden. Backtracking sorgt dafür, daß bei der Suche nach anwendbaren Ableitungsregeln bei Bedarf alle Alternativen systematisch in Betracht gezogen werden.

10.2 Cuts in Ableitungsregeln

Grüne Cuts schneiden keine Lösungen ab, verhindern aber die unnötige Suche in Zweigen, von denen wir wissen, daß die keine weiteren Lösungen enthalten.

Steht fest, daß in $f(x) = a$ a eine Konstante ist, so wird nach der Konstantenregel abgeleitet. Mit einem grünen Cut teilen wir dem Prolog-Interpreter mit, daß die alternativen Ableitungsregeln keine weiteren Lösungen bringen.

grüner Cut

```
ableitung (Konstante, X, 0):-
  atomic (Konstante),
  Konstante \= X,
  !.
```

Wenn klar ist, daß es sich in $f(x) = x + x^2$ um eine Summe handelt, so wird nach der Summenregel abgeleitet. Weitere Ableitungsregeln müssen nicht abgesucht werden:

```
ableitung (T1 + T2, X, T1a + T2a):-
  !,
  ableitung (T1, X, T1a),
  ableitung (T2, X, T2a).
```

Die Funktion $f(x) = 3 * x$ kann sowohl mit der Faktor- als auch mit der Produktregel abgeleitet werden. Die Lösung mit der Faktorregel reicht uns. Also verwenden wir einen roten Cut, um die Lösung mit der Produktregel zu vermeiden:

roter Cut

```
ableitung (Konstante * T, X, Konstante * Ta):-
  atomic (Konstante),
  Konstante \= X,
  !,
  ableitung (T, X, Ta).          /* roter Cut */

ableitung (T1 * T2, X, T1 * T2a + T2 * T1a):-
  !,
  ableitung (T1, X, T1a),
  ableitung (T2, X, T2a).          /* grüner Cut */
```

10.3 Potenzfunktionen

Mit den bisherigen Ableitungsregeln können keine Potenzfunktionen differenziert werden. Erstens fehlt eine Notation für Potenzfunktionen und zweitens die Umsetzung der Potenzregel in ein entsprechendes Prädikat.

Unter Ausnutzung der bisherigen Sprachmitteln könnte ein Polynom wie zum Beispiel $x^3 + 2x^2 - 7x + 3$ in Prolog als Struktur `pot(x,3) + 2*pot(x,2) - 7*x+3` geschrieben werden, wobei `pot(x, n)` für die Potenzfunktion x^n steht. In diesem Sinne kann `pot(x, n)` als Verallgemeinerung des Ansatzes betrachtet werden, der bei den Grundfunktionen `sin(x)`, `cos(x)`, `ln(x)` und `e(x)` verwendet wird: die Grundfunktionen haben ein Argument, die Potenzfunktionen zwei.

Schöner und praktikabler, aber nicht nötig, ist die Nutzung selbstdefinierter Operatoren. Mit der Anfrage `?- op(300, yfx, ^)` wird `^` als linksassoziativer Infix-Operator definiert. Das geht in *fiw*-Prolog nur über das Input-Fenster, in TV-SWI-Prolog können Anfragen auch im Quelltext stehen und werden dann während des Konsultierens bearbeitet. Der neue Operator stellt lediglich eine syntaktische Vereinfachung dar und hat zunächst nur für die Ein- und Ausgabe von Potenzen eine Bedeutung.

Obiges Polynom kann nunmehr in der Form `x^3+2*x^2-7*x+3` eingegeben, verarbeitet und ausgegeben werden. Setzen wir die Potenzregel wie folgt in eine Prolog-Klausel um

```
ableitung(X^N, X, N*X^(N - 1)):- atomic(N), N \== X.
```

so erhalten wir auf die Anfrage

```
?- ableitung(x^3+2*x^2-7*x+3,x,A).
A = 3*x^(3-1)+2*(2*x^(2-1))-7*1 + 0 als Lösung.
```

Die Semantik des `^`-Operators muß nicht festgelegt werden, weil die Bedeutung von x^n in diesem formalen System keine Rolle spielt. Erst wenn Funktionswerte berechnet werden sollen, muß die Bedeutung des `^`-Operators festgelegt werden. Dazu formulieren wir vorab das Prädikat *potenz*:

Berechnung von
Potenzen

```
potenz(T ^ 0, 1):-!.
potenz(T ^ 1, T):-!.
potenz(T1 ^ T2, Wert):-
    integer(T1), integer(T2),
    T3 is T2 - 1,
    potenz(T1 ^ T3, Wert1),
    Wert is Wert1 * T1, !.
potenz(T, T).          /* falls keine potenz-Regel anwendbar ist */
```

10.4 Kettenregel

Die Kettenregel $f(g(x))' = f'(g(x)) \cdot g'(x)$ läßt sich einfach umsetzen, wenn man den *univ*-Operator (siehe Kapitel 15) zur Verfügung hat. Er zerlegt den Ausdruck $f(g(x))$ in die Liste $[f, g(x)]$ aus der man das interessierende $g(x)$ entnehmen kann. $f(g(x))$ muß nach $g(x)$ und $g(x)$ nach x differenziert werden:

```
ableitung(T, X, Ta*Ga):-
  T =.. [F, G],
  ableitung(T, G, Ta),
  ableitung(G, X, Ga).
```

Kettenregel über
univ-Operator

Etwas komplizierter wird es, wenn man ohne *univ*-Operator auskommen will. Man muß dann für jede Grundfunktion die Ableitungsregel hinsichtlich der Kettenregel verallgemeinern:

```
ableitung(sin(T), X, cos(T)*Ta):-
  ableitung(T, X, Ta).
ableitung(cos(T), X, -sin(T)*Ta):-
  ableitung(T, X, Ta).
ableitung(ln(T), X, 1/T*Ta):-
  ableitung(T, X, Ta).
ableitung(e^T, X, Ta*e^T):-
  ableitung(T, X, Ta).
ableitung(T^N, X, N*T^(N - 1)*Ta):-
  atomic(N),
  N \== X,
  ableitung(T, X, Ta).
```

Kettenregel über
Ableitung der
Grundfunktionen

10.5 Allgemeine Funktionen

Um nicht nur mit bekannten sondern auch mit allgemeinen Funktionen wie $f(x)$ arbeiten zu können, ergänzen wir eine Ableitungsregel für allgemeine beziehungsweise unbekannte Funktionen. Wegen $f(x)' = f'(x)$ müßte man den Funktionsnamen mit der Verzierung ' versehen. Durch die damit verbundenen Ausgabeprobleme nehmen wir stattdessen den Buchstaben S (für Strich) als Verzierung.

$f(x)' = f'(x)$

Zur Implementierung der Ableitungsregel benötigt man den *univ*-Operator, um den Funktionsnamen vom Argument zu trennen. Über das Systemprädikat *name* zerlegt man den Funktionsnamen in eine Liste, ergänzt diese um den Buchstaben S und setzt dann den Funktionsterm wieder zusammen:

Implementierung
 der
 Ableitungsregel
 $f(x)' = f'(x)$

```

ableitung(T, X, Ta):-
    T=..[F, X],
    name(F, L1), append(L1, [83], L2),
    name(G, L2),
    Ta=..[G, X].
    
```

Beispiele:

```

?- ableitung(f(x),x,A).           Lösung: A=fS(x)
?- ableitung(f(x)+g(x),x,A).     Lösung: A=fS(x)+ gS(x)
?- ableitung(f(g(x)),x,A).       Lösung: A=fS(g(x))*gS(x)
?- ableitung(x^n,x,A).           Lösung: A=n*x^(n-1)
?- ableitung(ln(f(x)),x,A).      Lösung:
A=1/f(x)*fS(x)
    
```

zur Reihenfolge der
 Ableitungs-
 regeln

Auf die Reihenfolge der Ableitungsregeln kommt es an. So muß die Faktorregel vor der Produktregel stehen, da sonst die Produktregel angewendet wird, auch wenn die einfachere Faktorregel anwendbar wäre. Die Kettenregel muß ganz am Schluß stehen, weil Sie anderenfalls schon bei den Grundfunktionen zum Zuge käme und dabei eine unendliche Rekursion anstoßen würde. Insbesondere muß auch die Ableitungsregel für allgemeine Funktionen vor der Kettenregel in der Wissensbasis stehen.

10.6 Vereinfachung arithmetischer Ausdrücke

Die Ableitungsterme, welche durch das *ableitung*-Prädikat berechnet werden, sehen recht kompliziert aus, weil keinerlei Vereinfachungen vorgenommen werden. Mit etwas Aufwand lassen sich die Ableitungsterme vereinfachen. Wir betrachten dazu einen Ansatz, der ohne den *univ*-Operator auskommt, dafür mehr Schreibarbeit macht. Wir definieren ein Prädikat *vereinfachen(+Term, -VereinfachterTerm)* zur Vereinfachung von Summen, Differenzen, Produkten und Quotienten. Beispiel:

Vereinfachung von
 Summen und Diffe-
 renzen

```

vereinfachen(T1 + T2, V):-
    vereinfachen(T1, T1v),
    vereinfachen(T2, T2v),
    summe(T1v + T2v, V).
vereinfachen(T1 - T2, V):-
    vereinfachen(T1, T1v),
    vereinfachen(T2, T2v),
    summe(T1v - T2v, V).
    
```

Es vereinfacht zunächst die beiden Operanden, um dann die Summe zu vereinfachen. Zur Vereinfachung von Summen benutzen wir aus der Mathematik bekannte Fakten und Regeln.

Beispiele:

Vereinfachung
von Summen

```

summe(0 + T, T).
summe(T - T, 0).
summe(T1 + T2, T):- T is T1 + T2.
summe(A*T + B*T, S):- summe(A+B, A1), produkt(A1*T, S).
    
```

Falls keine Vereinfachungsmöglichkeit gefunden wird, geben wir die Originalsumme als Vereinfachung zurück:

```

summe(T, T).
    
```

10.7 Aufgaben

1. Entwickeln Sie ein Prädikat zur Berechnung der n-ten Ableitung.
2. Geben Sie eine Ableitungsregel in Prolog für die Ableitung von $f(x)^{g(x)}$ an.
3. Ergänzen Sie weitere *vereinfachen*-, *summe*- und *produkt*-Klauseln.
4. Testen Sie Ihre Lösung aus 3. an:

```

t1(X):- ableitung(4*x^3+3*x^2 - 4*x + 7, x, A),
         vereinfachen(A,X).
t2(X):- ableitung(x^2/x,x,A), vereinfachen(A,X).
t3(X):- ableitung((x^2+3*x+4*x)/(x^2 - 4),x,A),
         vereinfachen(A, X).
    
```

5. Berechnen Sie Funktionswerte von Ableitungsfunktionen! Beispiel:

```

?- ableitung(x^3+2*x^2 - 4*x +3,x,X),
   substituieren(x, 4, X, Y), /* Kapitel 9, Aufgabe 4 */
   vereinfachen(Y, Z).
    
```

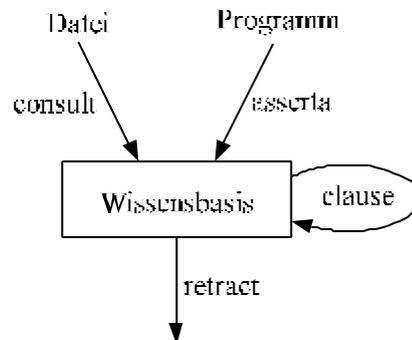
6. Entwickeln Sie Prädikate zum symbolischen Integrieren.

11 Wissensbasis und Regelsysteme

11.1 Hinzufügen und Löschen von Klauseln

Fakten und Regeln wurden bislang mittels *consult* und *reconsult* aus Quelldateien oder dem Editor in die Prolog-Wissensbasis geladen. Mit dem Systemprädikat *assert* können Klauseln, die ein Prolog-Programm selbst generiert hat, in die Wissensbasis aufgenommen werden:

Abb. 11-1
Prädikate zur
Manipulation der
Wissensbasis



Fakten mit *assert*
ergänzen

Lernen

Assert gibt es in den beiden Varianten *asserta* und *assertz*. *Asserta(+Klausel)* fügt die neue Klausel vor den bereits bestehenden Klauseln ein, *assertz(+Klausel)* hängt eine Klausel hinten an. Das *a* in *asserta* steht somit für den Anfang, das *z* in *assertz* für das Ende der Wissensbasis. Betrachten wir dazu folgendes Beispiel, bei dem die Wissensbasis zunächst zwei Klauseln zum Prädikat *maennlich* hat:

```

maennlich(heinz).
maennlich(manfred).
  
```

asserta

Nach *asserta(maennlich(fritz))* sieht die Wissensbasis so aus:

```
maennlich(fritz).
maennlich(heinz).
maennlich(manfred).
```

Das anschließende `assertz(maennlich(jens))` ergibt:

`assertz`

```
maennlich(fritz).
maennlich(heinz).
maennlich(manfred).
maennlich(jens).
```

Mit `assert` können Sie auch *Regeln* der Wissensbasis zufügen. Zur Syntax einiger Prolog-Interpreter gehört, daß Argumente eine Priorität kleiner 1000 haben müssen ist. Da der Regel-Operator „:-“ die Priorität 1200 hat, liefert `asserta(a:- b)` einen Syntaxfehler. Klammern reduzieren die Priorität auf 0, daher schreibt man `asserta((a:- b))`.

Regeln mit `assert`
ergänzen

Beispiele:

```
asserta((elternteil(E, Kind):- vater(E, Kind))).
assertz((member(X, [_|L]):- member(X, L))).
```

Klauseln mit `retract`

löschen

Vergessen

Zum Löschen von Klauseln aus der Wissensbasis gibt es das Prädikat `retract(+Klausel)`. `?- retract(maennlich(manfred))` ergibt die Wissensbasis:

```
maennlich(fritz).
maennlich(heinz).
maennlich(jens).
```

und `?- retract(maennlich(X))` führt zur Löschung der ersten *maennlich*-Klausel der Wissensbasis, mit dem Ergebnis:

```
maennlich(heinz).
maennlich(jens).
```

`retractall`

Standard-Prolog kennt nur das einfache `retract`-Prädikat, welches backtrackingfähig ist. Außer dem System-Prädikat `retract` bieten *fiæ*- und TV-SWI-Prolog `retractall`, zum Löschen aller Klauseln eines Prädikats.

Fakten und Regeln inspizieren Mit dem Systemprädikat *clause(+Klauselkopf, -Klauselrumpf)* kann ein Programm auf seine Fakten und Regeln zugreifen! Die Anfrage:

Rekapitulieren `?- clause(append(X, Y, Z), R).`

liefert der Reihe nach alle Klauselrumpfe zum *append/3*-Prädikat, sofern diese konsultiert sind.

Nur *retract*, *repeat* und *clause* sind in Standard-Prolog *backtracking-fähig*. Alle anderen Systemprädikate sind *deterministisch*. Die Autoren von *fiæ*-Prolog haben, im Unterschied zu Standard-Prolog, alle Systemprädikate deterministisch angelegt. Um dennoch in Standard-Prolog programmieren zu können, sind die drei genannten Ausnahmen in der *AUTOFIX.PRO*-Datei definiert. Mit *AUTOFIX.PRO* hat man also Standard-Prolog zur Verfügung.

Assert und *retract* erlauben es, ein Prolog-Programm dynamisch zu ändern. Aus der Sicht der Programmentwicklung ist das sehr problematisch, weil es schwer ist, Fehler in sich ändernden Programmen zu lokalisieren. Andererseits hat man damit Möglichkeiten, effiziente oder auch selbst lernende Programme zu schreiben und das braucht man für KI-Programme.

Clause erlaubt es, auf vorhandene Fakten und Regeln zuzugreifen. Dies wird insbesondere zur Konstruktion von Metainterpreter genutzt, also Prolog-Interpretern, die in Prolog geschrieben sind. Das *spur*-Prädikat ist ein solcher Metainterpreter.

11.2 Einfache Anwendungen

Beispiel 1: Zufallszahlen

Pseudozufallszahlen können mit der Kongruenzmethode erzeugt werden. Man beginnt mit einer Startzahl *S* und erzeugt nach der Formel

$$S = (9749 * S) \text{ mod } 131072$$

jeweils eine neue Zufallszahl. Zum Merken der letzten Zufallszahl benutzt man die Wissensbasis. Dort legt man ein Faktum ab, das die letzte Zufallszahl enthält:

```
basis(4567).
```

eine Implementierung von *random*

```
random(Zufallszahl):-
```

```

basis(Zahl),
Zufallszahl is 9749 * Zahl mod 131072,
retract(basis(Zahl)),
asserta(basis(Zufallszahl)).

```

Beispiel 2: Fibonaccizahlen

Die Fibonaccizahlen lassen sich am einfachsten rekursiv definieren:

```

fib(1) = 1
fib(2) = 1
fib(n) = fib(n-2) + fib(n-1) für n > 2

```

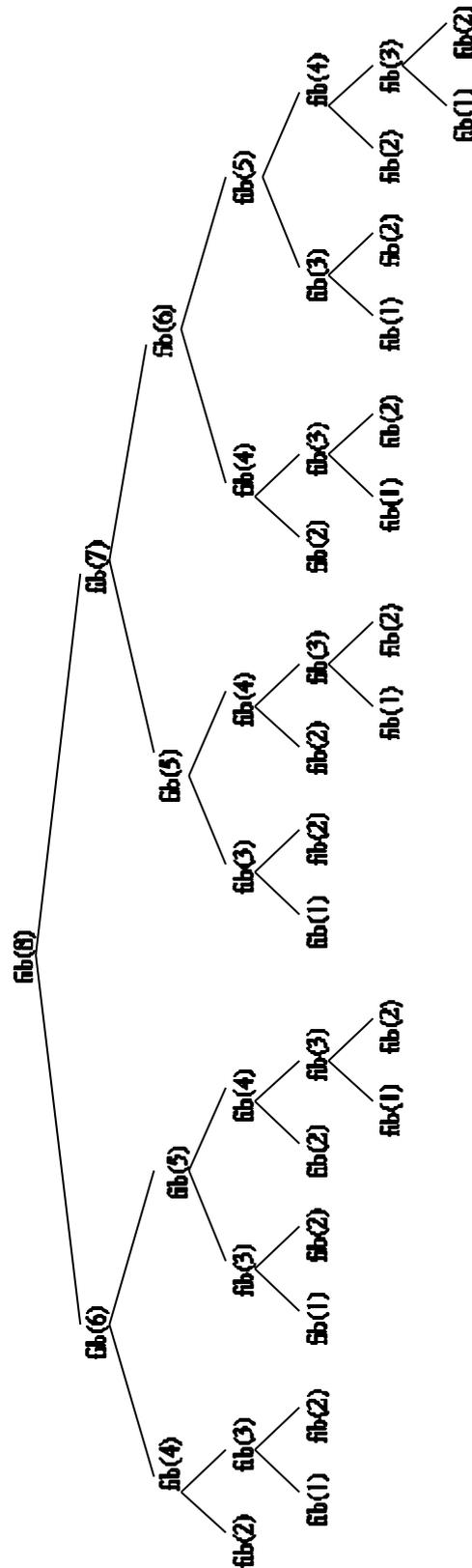
rekursive
Definition der
Fibonaccizahlen

Der Anfang der Fibonaccifolge ergibt sich daraus zu:

1, 1, 2, 3, 5, 8, 13, 21...

Ein erster Versuch, $fib(n)$ zu berechnen, besteht in:

Abb. 11-2
Doppelte
Rekursion bei den
Fibonacci-Zahlen



```

fib(1, 1).
fib(2, 1).

fib(N, F):-
  N > 2,
  N2 is N - 2,
  fib(N2, F2),
  N1 is N - 1,
  fib(N1, F1),
  F is F2 + F1.
  
```

Diese Methode ist höchst ineffektiv, weil durch die doppelte Rekursion im *fib*-Prädikat die Anzahl der rekursiven Aufrufe exponentiell wächst. Machen Sie sich das an Abbildung 12-2 klar. Man sieht, daß *fib(4)* 5-mal und *fib(3)* 8-mal berechnet wird.

Hier kann man die Wissensbasis geschickt einsetzen, indem man neu berechnete Werte sogleich in der Wissensbasis speichert. Damit spart man wiederholtes Berechnen und ersetzt es durch ein Nachschlagen in der Wissensbasis. Die neue *fib*-Regel lautet:

```

fib(N, F):-
  N > 2,
  N2 is N - 2,
  fib(N2, F2),
  N1 is N - 1,
  fib(N1, F1),
  F is F2 + F1,
  asserta(fib(N,F)).
  
```

11.3 Mengenprädika-

te

Wir betrachten als einfaches Beispiel unser Familienprogramm mit den Klauseln:

```
vater(steffen, paul).
vater(fritz, karin).
vater(steffen, lisa).
vater(paul, maria).
```

Um alle Kinder von *steffen* zu bestimmen, stellt man die Frage `?- vater(steffen, Kind).` und erhält nacheinander vom System alle Lösungen in der Art

```
Kind = paul
Kind = lisa
No.
```

wobei jede neue Lösung vom System angefordert werden muß. Will man sich das Anfordern der Lösungen ersparen, so verwendet man eine *failure-driven loop*:

```
?- vater(steffen, Kind), fail.
```

Auf diese Anfrage gibt das System aber keine Lösungen aus, da das Ziel nie erreicht wird. Nur wenn das System eine Lösung findet, gibt es die zugehörige Variablenbelegung aus. Das *fail*-Prädikat verhindert die Erreichung des Ziels, hat aber den Zweck, nach Erfüllung des Teilziels *vater(steffen, Kind)* Backtracking einzuleiten. Wir schieben daher eine Ausgabeanweisung ein und erhalten so alle Lösungen ohne Nachfrage angezeigt:

```
?- vater(steffen, Kind), write(Kind), nl, fail.
```

Es gibt Anwendungsfälle, bei denen man nicht an den einzelnen Lösungen, sondern an der Lösungsmenge interessiert ist. Beispielsweise könnten Sie mit der Lösungsmenge bestimmen, wie viele Kinder *steffen* hat oder die Kinder in sortierter Reihenfolge ausgeben.

Im nächsten Kapitel entwickeln wir das *findall*-Prädikat, mit dem eine Liste aller Lösungen erzeugt werden kann. Für unser Beispiel könnte es in folgender Form verwendet werden:

```
?- findall(Kind, vater(steffen, Kind), KindListe).
```

Kind ist die Lösungsvariable, das Lösungsziel steht in der Mitte und als drittes Argument erscheint die Lösungsliste.

11.4 Herleitung von *findall*

Unter Ausnutzung der Wissensbasis können wir *findall* relativ leicht realisieren. Als erstes erzeugen wir alle Lösungen und speichern sie in der Wissensbasis ab. Im zweiten Schritt sammeln wir alle gespeicherten Lösungen auf. Beim Aufsammeln löschen wir die Lösungen wieder aus der Wissensbasis, um eine unnötige Speicherbelastung zu vermeiden.

Sie haben oben gesehen, wie man alle Lösungen eines Ziels erzeugen kann. Nun sollen die Lösungen nicht ausgegeben, sondern in der Wissensbasis gespeichert werden. Zum Speichern verpackt man eine Lösung in ein Faktum, dessen Funktor wir *gefunden* nennen. Damit haben wir:

alle Lösungen in
der Wissensbasis
ablegen

```
/* findall(+Term, +Ziel, -Liste) */
findall(Loesung, Ziel, _):-
    call(Ziel),
    assertz(gefunden(Loesung)),
    fail.
```

Nun stehen alle Lösungen in der Wissensbasis. Zum Aufsammeln verwenden wir eine weitere Klausel für *findall*. Da obiges *findall* wegen *fail* als letztem Teilziel nie gelingen kann, sorgt Backtracking dafür, daß die nächste *findall*-Klausel auch benutzt wird.

alle Lösungen
aufsammeln

```
findall(_, _, Liste):-
    sammeln(Liste).
```

Das Aufsammeln ist etwas schwieriger. Wir brauchen eine geeignete Terminierungsbedingung. Das Aufsammeln endet genau dann, wenn es keine *gefunden*-Klausel mehr gibt. Mit dem Systemprädikat *clause*, läßt sich das nachprüfen. *Clause* wird mit zwei Argumenten aufgerufen: das erste gibt den Klauselkopf, das zweite den Klauselrumpf an. Fakten haben keinen Rumpf. Da Fakten immer wahr sind, gibt man im *clause*-Prädikat *true* als Rumpf an:

alle Lösungen
gefunden

```
sammeln([]):-
    not(clause(gefunden(_), true)).
```

Damit ist auch der Fall erfaßt, daß ein Ziel gar keine Lösung hat. Sie erhalten dann die leere Liste als Ergebnis. Im allgemeinen hat man Lösungen, die man mit End-Rekursion aufsammelt. Anstelle von *clause* verwendet man hier *retract*, da damit die Lösungen wieder gelöscht werden:

weitere Lösungen
aufsammeln

```
sammeln([Kopf|Rest]):-
    retract(gefunden(Kopf)),
    sammeln(Rest).
```

Damit haben wir eine Lösung für das *findall*-Prädikat gefunden. In TV-SWI-Prolog stehen *findall* und die beiden weiteren Mengenprädikate *bagof* und *setof* als Systemprädikate zu Verfügung.

11.5 Ein Regelsystem zur Bestimmung von Säugetierarten

Wir betrachten ein einfaches Regelsystem nach [Bur1], mit dem man aus gewissen beobachtbaren Merkmalen die Tierart eines bestimmten Säugetieres bestimmen kann. Das System enthält acht Regeln, welche die Begriffe der folgenden Begriffshierarchie mit Hilfe spezifischer Merkmale definieren:

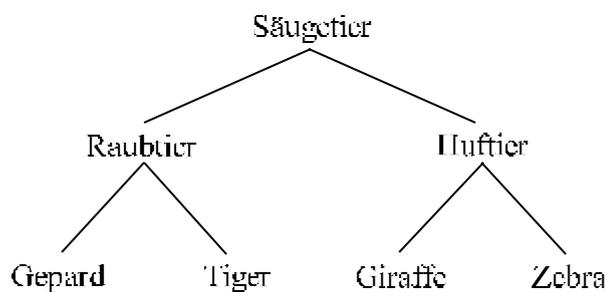


Abb. 11-3
Begriffshierarchie
für Säugetierarten

- R1 Wenn das Tier ein Raubtier ist,
ein lohfarbenes Fell hat und
dunkle Flecken hat,
dann ist es ein Gepard.
- R2 Wenn das Tier ein Raubtier ist,
ein lohfarbenes Fell hat und
schwarze Streifen hat,
dann ist es ein Tiger.
- R3 Wenn das Tier ein Huftier ist,
lange Beine hat,
einen langen Hals hat,
ein lohfarbenes Fell hat und
dunkle Flecken hat,
dann ist es eine Giraffe.

Regeln zur Be-
stimmung von
Säugetierarten

- R4 Wenn das Tier ein Huftier ist,
ein weißes Fell hat und
schwarze Streifen hat,
dann ist es ein Zebra.
- R5 Wenn das Tier Fleisch frißt,
dann ist es ein Raubtier.
- R6 Wenn das Tier spitze Zähne hat und
Pranken hat,
dann ist es ein Raubtier.
- R7 Wenn das Tier Hufe hat,
dann ist es ein Huftier.
- R8 Wenn das Tier wiederkäut,
dann ist es ein Huftier.

Wir erstellen ein Prolog-Programm, das durch Auswertung der Regeln und Erfragen von Merkmalen die Tierart bestimmt. Die Regeln werden durch Klauseln zum Prädikat *ist_ein* erfaßt. Beispiele:

Modellierung der
Regeln

```
ist_ein('Gepard'):-
    ist_ein('Raubtier'),
    merkmal('hat lohfarbenes Fell '),
    merkmal('hat dunkle Flecken ').

ist_ein('Raubtier'):-
    merkmal('hat spitze Zähne '),
    merkmal('hat Klauen ').
```

Das Prädikat *merkmal* erfragt ein Merkmal vom Benutzer und wertet die Eingabe aus.

erfragte Merkmale
auswerten

```
merkmal(Merkmal):-
    erfrage(Merkmal, Antwort),
    auswerte(Merkmal, Antwort).
```

fallspezifisches
Wissen erfragen

```
erfrage(Merkmal, Antwort):-
    write(Merkmal), write('?'),
    read(Antwort),
    write(Antwort), nl.
```

Wenn das Tier ein Merkmal hat, so muß das Prädikat *merkmal* erfüllt sein, sonst muß es fehlschlagen. Daher:

```
auswerte(Merkmal, 'ja').
auswerte(Merkmal, 'nein'):- fail.
```

Das Regelsystem wird durch *run* gestartet:

```
run:-
  ist_ein(X), nl,
  write('Das Tier könnte ein '),
  write(X), write(' sein.').
```

In Aufgabe 10 gibt es Vorschläge, wie Sie die unbefriedigende Dialogführung des Regelsystems durch Einsatz der Wissensbasis verbessern können.

11.6 Aufgaben

1. In der Wissensbasis stehen Fakten der Art *jagt(Jaeger, Beute)*. Sie wollen alle Beutetiere wissen. Wie verwenden Sie dazu *findall*?
2. Es sind folgende Fakten gegeben:


```
mann(hugo). mann(emil). mann(karl). mann(udo).
elternteil(hugo, udo). elternteil(emil, susi).
elternteil(emil, karl). elternteil(anna, udo).
```

 - a) Es sollen alle Kinder in einer Liste gesammelt werden. Formulieren Sie eine entsprechende Anfrage.
 - b) Die Ziele können auch zusammengesetzt sein. Formulieren Sie eine Anfrage, um alle Väter in einer Liste zu sammeln.
3. In der Wissensbasis sind Fakten mit Namen, Beruf und Alter von Personen gespeichert. Es soll die älteste Person ermittelt werden.


```
person(karl, bauer, 26).
person(susi, lehrerin, 25).
person(udo, ingenieur, 30).
person(kurt, kaufmann, 26).
person(ute, informatikerin, 38).
```
4. In der Wissensbasis sei unser Familienprogramm vorhanden. Ein Vater ist über die Tastatur einzugeben. Anschließend soll ermittelt werden, wie viele Kinder der Vater hat. Schreiben Sie dazu ein geeignetes Prolog-Prädikat.
5. Wie sammeln Sie alle Nachbarräume von Raum *e* im Labyrinth von Kapitel 5, Aufgabe 21 in einer Liste?
6. Die Binomialkoeffizienten können rekursiv definiert werden:


```
bin(n, 0) = 1,
```

```
bin(n, n) = 1,
bin(n, k) = bin(n-1, k-1) + bin(n-1, k).
```

Formulieren Sie ein Prolog-Prädikat zur Berechnung von Binomialkoeffizienten einmal ohne und einmal mit *assert*. Vergleichen Sie die beiden Lösungen.

7. Definieren Sie ein Prädikat *delete(+Funktork, +Arität)*, das alle Klauseln zu einem Funktork gegebener Arität löscht. Verwenden Sie zur Lösung das Systemprädikat *retract*. Achtung: In TV-SWI-Prolog hat meine Lösung funktioniert, aber nicht in *fiæ*-Prolog.
8. Das Spiel Nimm ist für zwei Personen gedacht. Am Anfang liegt ein Haufen Streichhölzer auf einem Tisch. Abwechselnd nimmt jeder Spieler höchstens 3 aber mindestens 1 Streichholz weg. Gewonnen hat derjenige Spieler, der die letzten Streichhölzer wegnimmt. Der folgende Zugberater sucht den kompletten Spielbaum ab, um einen Gewinnzug zu finden. Dies dauert schon bei kleinen Streichholzhaufen lange. Erklären Sie dies und verbessern Sie den Zugberater durch Lernen von Gewinn- und Verlustpositionen.

```
/* Zugberater für Nimm */

verlust(0).
verlust(X):- not(gewinn(X)).

gewinn(1). gewinn(2). gewinn(3).
gewinn(X):-
    X > 3,
    (X1 is X - 1; X1 is X - 2; X1 is X - 3),
    verlust(X1).

zug(X):-
    (X1 is X - 1; X1 is X - 2; X1 is X - 3),
    verlust(X1),
    Wert is X - X1,
    write('Nimm '),
    write(Wert), write(' Streichhölzer!'), nl.
zug(X):-
    write('Der Gegner kann gewinnen '),
    write('nimm beliebig. '), nl.
```

9. Analysieren Sie das folgende *findall* aus Clocksin & Mellish:

```
findall(X, G, _):-
    asserta(found(mark)),
    call(G),
    asserta(found(X)),
    fail.
findall(_, _, L):- collect_found([], M), !, L = M.
collect_found(S, L):-
```

```
    getnext(X), !, collect_found([X|S], L).  
collect_found(L, L).  
getnext(X):- retract(found(X)), !, X \== mark.
```

- 10a) Komplettieren Sie das Regelsystem aus 12.5 und führen Sie einige Tierbestimmungen durch.
- b) Sie werden feststellen, daß einige Fragen sich wiederholen. Analysieren Sie mit Hilfe des *spur*-Prädikats die Ursache hierfür.
- c) Zur Vermeidung der Wiederholungsfragen können wir die Wissensbasis benutzen. Dort legen wir Fakten zum zweistelligen Prädikat *hat* ab. Damit werden die Benutzereingaben gespeichert und stehen für die spätere Verwendung zur Verfügung. Gibt der Benutzer an, daß das Tier ein Fleischfresser ist, so speichern Sie

```

    hat('Fleischfresser ', 'ja')      ab, anderenfalls
    hat('Fleischfresser ', 'nein')

```

- Ergänzen Sie das Abspeichern der Fakten in den *auswerte*-Klauseln.
- d) Führen Sie eine zweite Klausel für das *merkmal*-Prädikat ein, welche vor dem Abfragen und Auswerten zunächst die Wissensbasis durchsucht. Wenn die Antwort dort gefunden wird, muß der Benutzer kein zweites Mal gefragt werden.
- e) Nachdem ein Tier bestimmt ist, müssen alle *hat*-Fakten wieder gelöscht werden. Ändern Sie entsprechend *run* ab.
11. Zur Bestimmung eines geeigneten Medikamentes bei vorgegebenen Beschwerden eines Patienten seien folgende Regeln und Fakten gegeben:

Regeln:

- R1 Wenn der Patient P unter dem akuten Symptom S leidet und bei dem Symptom S das Medikament M wirksam ist und das Medikament für P nicht unverträglich ist, dann soll der Patient P das Medikament einnehmen.
- R2 Wenn der Patient P unter dem chronischen Symptom S leidet und das Medikament M Nebenwirkungen bei S hat, dann ist das Medikament M für den Patienten P unverträglich.

Fakten:

- F1 Aspirin ist wirksam bei Kopfschmerzen.
- F2 Aspirin ist wirksam bei Grippe.
- F3 Lomotal ist wirksam bei Kopfschmerzen.
- F4 Lomotal ist wirksam bei Durchfall.
- F5 Tentamin ist wirksam bei Grippe.
- F6 Tentamin ist wirksam bei Durchfall.
- F7 Aspirin hat Nebenwirkungen bei Magengeschwüren.
- F8 Lomotal hat Nebenwirkungen bei Leberschäden.
- F9 Tentamin hat Nebenwirkungen bei Bluthochdruck.

- a) Definieren Sie Prolog-Fakten für die beiden Prädikate
- `ist_wirksam_bei(Medikament, Symptom)` und
 - `hat_Nebenwirkung_bei(Medikament, Symptom)`
- b) Übersetzen Sie die Regeln in zwei Prolog-Regeln
- `soll_einnehmen(Medikament, Patient)` und
 - `ist_ungeeignet(Medikament, Patient)`.
- c) Definieren Sie die Prädikate
- `leidet_unter(Symptom)` und
 - `chronisches_symptom(Symptom)`
- als abfragbare Prädikate, bei denen die Antwort vom Benutzer des Programms eingegeben wird.
- d) Mit dem System soll z.B. der folgende Dialog geführt werden können:

```
?- soll_einnehmen(Medikament).
```

```
Welche Beschwerden hat der Patient? Kopfschmerzen.  
Leidet der Patient unter Magengeschwüren? ja.  
Leidet der Patient unter Leberschäden? nein.
```

```
Der Patient soll Lomotal einnehmen.
```

12 ICE-Auskunftssystem

12.1 Modellbildung

Die Deutsche Bundesbahn bietet elektronische Bahnauskunft auf Diskette, CD-ROM und über Datex-J (BTX) an. Wir wollen im folgenden selbst ein

Informatiksysteme
solches Informationssystem für die Bahnauskunft konzipieren und realisieren, um dabei typische Informatikmethoden wie zum Beispiel Analyse, Modellierung und Konstruktion komplexer Informatiksysteme kennenzulernen und anzuwenden.

Wir gehen vom *CityFahrplan* aus [Bah1], in welchem alle ICE-, EC- und IC-Verbindungen aufgeführt sind. Beschränkt man sich auf die ICE-Züge und die relevanten Strecken, so kommt man zu folgendem Streckennetz:

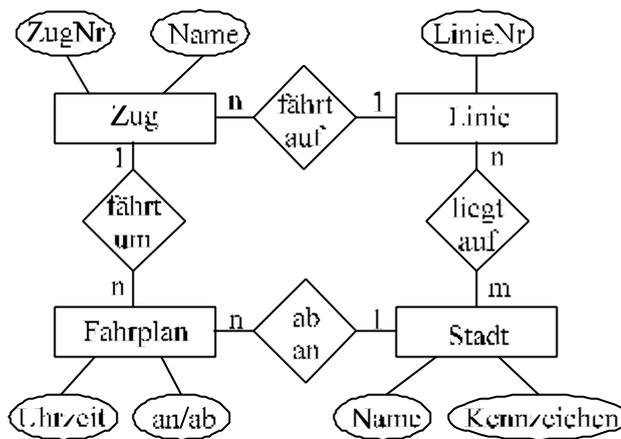
Abb. 12-1
ICE-Streckennetz

Objekte und Beziehungen

Das Streckennetz besteht aus den *Linien* 1, 3, 4, 5 und 6. Auf einer Linie liegen mehrere *Städte* und eine Stadt kann zu mehreren Linien gehören. Über das Kennzeichen kann eine Stadt eindeutig identifiziert werden. Der *CityFahrplan* gibt zu jeder Linie mehrere *Züge* an, die zu unterschiedlichen Zeiten auf den entsprechenden Linien fahren. Jeder Zug hat eine eindeutige Nummer und einen klangvollen Namen. Beispielsweise fährt der ICE 638 *Alster-Kurier* auf der Linie 1 von Hamburg nach Köln. Der *Fahrplan* gibt darüber Auskunft, wann ein Zug anhält beziehungsweise abfährt.

Man kann den interessierenden Realitätsausschnitt in Form eines Entity-Relationship-Diagramms (Objekt-Beziehungs-Diagramm) modellieren:

Abb. 12-2
ER-Diagramm
des ICE-
Auskunftsystems



Die Rechtecke stellen die Objekte dar, die Ovale die Eigenschaften der Objekte und die Rauten die Beziehungen der Objekte. Die Abbildung des Entity-Relationship-Diagramms auf Relationen könnte durch vier Relationen für die Objekte und vier Relationen für die Beziehungen erfolgen. Die drei 1-n Beziehungen lassen sich aber durch Aufnahme des Schlüsselattributs der 1-Seite in die Relation der n-Seite einsparen. Zudem kann die Relation für die Linien in die Beziehungs-Relation *liegt auf* integriert werden, weil Linie nur ein einziges Attribut hat.

Es reichen also zur Modellierung des Streckennetzes und Fahrplans vier Relationen aus:

Abbildung des ER-Diagramms auf Relationen

- stadt(Name, Kennzeichen)
- linie(LinieNr, Kennzeichen)
- zug(LinieNr, ZugNr, Name)
- fp(ZugNr, Uhrzeit, Kennzeichen, an/ab)

Man sieht, nichts ist praktischer als eine gute Theorie!

In [Dei1] kann man nachlesen, wie man auch ohne Kenntnisse der Datenbanktheorie zu einem ähnlichen Entwurf kommt. Das dort vorgestellte Unterrichtsprojekt liegt diesem Kapitel zugrunde.

Zur Illustration sind nachfolgend zu jeder Relation einige Datensätze in Prolog-Klauserform angegeben:

```
stadt('Berlin', b).
stadt('Frankfurt', f).
stadt('Fulda', fd).
stadt('Hamburg', hh).
stadt('Kassel', ks).
```

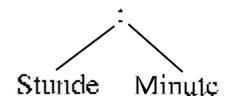
typische
Datensätze der
beteiligten
Relationen

```
linie(4, hh).
linie(4, ha).
linie(4, goe).
linie(4, ks).
linie(4, fd).
```

```
zug(3, 672, 'Markgraf').
zug(3, 776, 'Schauinsland').
zug(3, 670, 'Heinrich Hoffmann').
zug(3, 76, 'Panda').
zug(3, 774, 'Franz Kruckenberg').
```

```
fp(791, 8:13, ks, ab).
fp(791, 8:41, fd, an).
fp(791, 8:43, fd, ab).
fp(791, 9:39, f, an).
fp(791, 9:43, f, ab).
fp(791, 10:24, ma, an).
```

Uhrzeiten werden mit dem Binäroperator „:“ als zweistellige Terme geschrieben. Zeitvergleiche sind über die *Term-Vergleichsoperatoren* (z.B.: @< und @>=, siehe Kapitel 15) guter Prolog-Interpreter möglich.



Die *stadt*-, *linie*- und *zug*-Relationen können mit überschaubarem Aufwand auf den Rechner übertragen werden. Mühsam und fehlerträchtig ist die Erfassung des Fahrplans, weil er aus mehreren Hundert *fp*-Fakten besteht. Die komplette Datenbasis steht Ihnen als Datei *FAHRPLAN.PL* zur Verfügung, im Unterricht wird man die Schüler zumindest teilweise diese Datenbasis erstellen lassen.

12.2 Der ICE-Experte

Auf der Grundlage des umfangreichen Datenmaterials können alle möglichen Fragen zum ICE-Streckennetz beantwortet werden. Beispiele:

Welche Städte liegen auf der Linie 3?

```
?- linie(3, Stadt).    oder besser
?- linie(3, Stadt), stadt(Name, Stadt).
```

Auf welcher Linie fährt der ICE 882?

```
?- zug(Linie, 882, _).
```

Welche Züge verkehren auf der Linie 5?

```
?- zug(5, ZugNr, Name).
```

Wann fährt der ICE 684 in Fulda ab?

```
?- fp(684, Zeit, fd, ab).
```

Hat Darmstadt einen ICE-Anschluß?

```
?- linie(_, da).
```

Gibt es einen ICE namens *Diamant*?

```
?- zug(_, _, 'Diamant').
```

Fährt der ICE 792 über Kassel?

```
?- zug(Linie, 792, _), linie(Linie, ka).
```

Was ist die Endstation des ICE 996?

```
?- fp(996, _, Stadt, an), not fp(996, _, Stadt, ab).
```

Wann fahren Züge von München nach Berlin?

```
?- fp(Nr, Abfahrt, mue, ab), fp(Nr, Ankunft, b, an).
```

Kann man in Frankfurt vom ICE 896 in den ICE 585 umsteigen?

```
?- fp(896, Ankunft, f, an), fp(585, Abfahrt, f, ab),
Ankunft @< Abfahrt.
```

Hält der *Isar-Sprinter* in Stuttgart?

```
?- zug(_, Nr, 'Isar-Sprinter'), fp(Nr, _, st, an).
```

In welchen Städten hält der ICE 576?

```
?- fp(576, Ankunft, Stadt, an).
```

Mit welchen Zügen kommt man aus Frankfurt vor 17.00 Uhr in Berlin an?

```
?- fp(Nr, _, f, ab), fp(Nr, Ankunft, b, an),
Ankunft @< 17:00.
```

12.3 Zugbegleiter

Wenn man für jeden Zug die Abfahrts- und Ankunftszeiten chronologisch in der Datenbasis ablegt, kann man relativ einfach einen Zugbegleiter erstellen:

```
zugbegleiter(Zug):-
    zug(_, Zug, Name),
    write('Zugbegleiter: '), write(Zug), tab(2),
    write(Name), nl, nl,
    write('   Zeit   Ort'), nl,
    linie_zeichnen(-, 22), nl,
    zeige_stationen(Zug),
    linie_zeichnen(-, 22).

zeige_stationen(Zug):-
    fp(Zug, Zeit, Station, AbAn),
    tab(2), write_time(Zeit),
    tab(1), write(AbAn),
    stadt(Name, Station),
    tab(1), linksbuendig(Name, 22), nl,
    fail.
zeige_stationen(_).
```

Programm für
einen Zugbegleiter

Das Prädikat *zugbegleiter* erstellt eine Überschrift und läßt dann von *zeige_stationen* alle Stationen der Reihenfolge nach ausgeben. Das automatische Backtracking wird durch *fail* als letztem Teilziel erreicht.

```
?- zugbegleiter(639).

Zugbegleiter: 639   Alster-Kurier
   Zeit   Ort
-----
 6:26 ab Köln
 6:45 an Düsseldorf
 6:47 ab Düsseldorf
 7:08 an Essen
 7:10 ab Essen
 7:50 an Münster
 7:52 ab Münster
 9:46 an Hamburg
-----
```

Beispiel eines Zug-
begleiters

Die Kosmetik wird durch formatierte Ausgaben erreicht, welche wir in Kapitel 7 kennengelernt haben. Für Uhrzeiten benutzen wir eine eigenständige formatierte Ausgabe, um bei Minuten führende Nullen ergänzen zu können:

formatierte Ausgabe von Zeiten

```
write_time(Stunden : Minuten):-  
    rechtsbuendig(Stunden, 2),  
    write(':'),  
    write_minuten(Minuten).  
write_minuten(Minuten):-  
    Minuten < 10, write('0'), fail.  
write_minuten(Minuten):-  
    write(Minuten).
```

Einen unformatierten und manuell zu erzeugenden Zugbegleiter erhält man durch die einfache Anfrage:

```
?- fp(639, Zeit, Stadt, AnAb).
```

12.4 Abfahrtsplan

An jedem Bahnhof gibt es Abfahrtspläne, aus denen man entnimmt, zu welchen Zeiten Züge mit welchem Ziel fahren. Den rudimentärsten Abfahrtsplan erhält man durch die Anfrage

```
?- fp(Nr, Abfahrt, f, ab).
```

wobei im Beispiel wegen ‚f‘ im dritten Argument der Abfahrtsplan für Frankfurt generiert wird. Um auch zu erfahren, wohin ein Zug fährt und wann er dort ankommt, muß man den Zielbahnhof eines Zuges aus dem Fahrplan ermitteln. Man kann den Zielbahnhof nicht aus der zum Zug gehörigen Linie ermitteln, weil beispielsweise in den Abendstunden nicht mehr alle Züge bis zum letzten Bahnhof auf der Linie fahren und damit der Zielbahnhof nicht mit Endbahnhof der Linie übereinstimmt.

Der Zielbahnhof eines Zuges wird durch das Prädikat *zielbahnhof(+ZugNr, -Stadt)* ermittelt:

```
zielbahnhof(Nr, Stadt):-
    fp(Nr, _, Stadt, an),
    not fp(Nr, _, Stadt, ab).
```

Zielbahnhof

Damit kann ein besserer Abfahrtsplan generiert werden:

```
?- fp(Nr, Abfahrt, f, ab),
    zielbahnhof(Nr, Ziel),
    fp(Nr, Ankunft, Ziel, an).
```

Die Abfahrten sind allerdings noch nicht chronologisch geordnet. Um dies zu erreichen, erstellt man eine Liste aller Abfahrten und sortiert sie nach den Abfahrtszeiten. Die erste Aufgabe erledigt man mit dem System-Prädikat *findall/3*, die zweite mit dem System-Prädikat *sort(+UnsortierteListe, -SortierteListe)*.

findall und sort

Das angegebene Prädikat *direkte_abfahrt/2* erzeugt für eine gegebene Stadt die sortierte Abfahrtsliste. Abfahrten werden als *ab/4*-Terme mit Abfahrtszeit, Zugnummer, Zielbahnhof und Ankunftszeit von *findall* gebildet und in der Liste *Abfahrten1* gespeichert.

```
direkte_abfahrt(Von, Abfahrten):-
    findall(ab(Abfahrt, Nr, Nach, Ankunft),
        (fp(Nr, Abfahrt, Von, ab),
         zielbahnhof(Nr, Nach),
         fp(Nr, Ankunft, Nach, an))),
    Abfahrten1),
```

```
sort(Abfahrten1, Abfahrten).
```

Term-
Vergleichsoperator

Abfahrten1 wird von *sort* zur Liste *Abfahrten* sortiert. Zum Sortieren benutzt *sort* einen *Term-Vergleichsoperator*. Bei gleichartigen Termen richtet sich die Sortierfolge der Terme nach dem ersten Argument. Da dies die Abfahrtszeit in den *ab/4*-Termen ist, wird nach den Abfahrtszeiten sortiert.

Eine formatierte Ausgabe ist wie folgt möglich:

Programm für
einen Abfahrtsplan

```
abfahrt(Stadt):-
    stadt(Name, Stadt), nl,
    write('Abfahrt: '), write(Name), nl, nl,
    write('    ab Zug nach        an'), nl,
    linie_zeichnen(-, 31), nl,
    direkte_abfahrt(Stadt, Abfahrten),
    zeige_abfahrten(Abfahrten),
    linie_zeichnen(-, 31).

zeige_abfahrten([Kopf|Rest]):-
    Kopf = ab(Abfahrt, Nr, Nach, Ankunft),
    tab(2),
    write_time(Abfahrt),
    rechtsbuendig(Nr, 4), tab(1),
    stadt(Stadt, Nach),
    linksbuendig(Stadt, 12),
    write_time(Aankunft), nl,
    zeige_abfahrten(Rest).
zeige_abfahrten([]).
```

Damit erhält man beispielsweise folgenden Abfahrtsplan:

Beispiel eines
Abfahrtsplans

```
Abfahrt: Freiburg
      ab  Zug nach      an
-----
    6:33 776 Hamburg    12:21
    8:33  76 Hamburg    14:21
    9:33 774 Hamburg    15:21
   13:33  70 Hamburg    19:21
   15:33 770 Hamburg    21:22
   21:01 270 Frankfurt   23:16
-----
```

12.5 Zugauskunft

Sie wollen morgens, frühestens ab 7⁰⁰ Uhr, von Frankfurt nach Berlin fahren. Wann fährt ein ICE? Wir beschränken uns zunächst auf Direktverbindungen. Interaktiv geht das einfach durch:

```
?- fp(Nr, Abfahrt, f, ab),
    7:00 @=< Abfahrt,
    fp(Nr, Ankunft, b, an),
    Abfahrt @< Ankunft.
```

Als Standardabfrage an unsere Datenbank formulieren wir:

```
direkte_verbindung_ab(Von, Nach, Ab):-
    fp(Nr, Abfahrt, Von, ab),
    Ab @=< Abfahrt,
    fp(Nr, Ankunft, Nach, an),
    Abfahrt @< Ankunft,
    zeige_kopf,
    zeige_verbindung(Von, Abfahrt, Nr, Nach, Ankunft).
```

Programm für
eine Zugauskunft

Mit etwas Aufwand

```
zeige_kopf:-
    nl,write('Reiseverbindung   Deutsche Bundesbahn'),
    nl, write('Bahnhof           Uhr           Zug'),
    nl, linie_zeichnen(-, 38), nl.

zeige_verbindung(Von, Abfahrt, Nr, Nach, Ankunft):-
    stadt(Name1, Von), linksbuendig(Name1, 20),
    write('ab '), write_time(Abfahrt),
    write('   ICE '), write(Nr), tab(1), nl,
    stadt(Name2, Nach), linksbuendig(Name2, 20),
    write('an '), write_time(Ankunft), nl.
```

erhält man folgende formatierte Ausgabe:

```
Reiseverbindung   Deutsche Bundesbahn
Bahnhof           Uhr           Zug
-----
Frankfurt         ab 7:18   ICE 696
Berlin           an 12:10
```

Beispiel einer
Zugauskunft

Die Lösung für eine direkte Verbindung lässt sich problemlos auf eine Umsteige-
verbindung erweitern. Eine Umsteigeverbindung ist nichts anderes als die
Kopplung zweier Direktverbindungen. Zuerst führt eine Direktverbindung vom

Startbahnhof zum Umsteigebahnhof, dann die zweite Direktverbindung vom Umsteigebahnhof zum Zielbahnhof.

Um den Anschlußzug zu erreichen, muß man am Umsteigebahnhof vor der Abfahrtszeit des Anschlußzuges ankommen.

Programm für
eine Umsteige-
verbindung

```
umsteige_verbindung_ab(Von, Nach, Ab):-
    fp(Nr1, Abfahrt1, Von, ab),
    Ab @=< Abfahrt1,
    fp(Nr1, Ankunft1, Umsteige, an),
    Abfahrt1 @< Ankunft1,
    fp(Nr2, Abfahrt2, Umsteige, ab),
    Ankunft1 @< Abfahrt2,
    fp(Nr2, Ankunft2, Nach, an),
    Abfahrt2 @=< Ankunft2,
    zeige_kopf,
    zeige_verbindung(Von, Abfahrt1, Nr1,
                     Umsteige, Ankunft1),
    zeige_verbindung(Umsteige, Abfahrt2, Nr2,
                     Nach, Ankunft2),
    berechne_dauer(Abfahrt1, Ankunft2, Dauer),
    zeige_dauer(Dauer).
```

Beispiel einer Umsteigeverbindung zwischen Freiburg und Ulm:

Beispiel
einer Umsteige-
verbindung

Reiseverbindung Bahnhof	Deutsche Bundesbahn Uhr	Zug
-----	-----	-----
Freiburg	ab 13:33	ICE 70
Mannheim	an 15:02	
Mannheim	ab 15:27	ICE 595
Ulm	an 17:03	
Dauer: 3:30 h		

Die beiden Teillösungen lassen sich zu einer Gesamtlösung zusammensetzen, bei der man nicht wissen muß, ob eine Direkt- oder Umsteigeverbindung möglich ist:

Direkt- und Um-
steigeverbindung

```
verbindung_ab(Von, Nach, Ab):-
    direkte_verbindung_ab(Von, Nach, Ab), !.
verbindung_ab(Von, Nach, Ab):-
```

`umsteige_verbindung_ab(Von, Nach, Ab).`

Die Erweiterung auf mehrmaliges Umsteigen liegt auf der Hand und kann problemlos auf der Basis des Prädikats *umsteige_verbindung_ab* implementiert werden.

12.6 Heuristische Suche im ICE-Netz

Daß die bisherige Lösung auch Tücken hat, zeigt die folgende Zugauskunft:

Reiseverbindung Bahnhof	Deutsche Bundesbahn Uhr	Zug	Eine Zugauskunft mit Tücken

München	ab 7:20	ICE 682	
Kassel	an 10:36		
Kassel	ab 12:45	ICE 71	
Freiburg	an 16:27		

Man kann vier Stunden eher in Freiburg sein, wenn man in Mannheim und nicht in Kassel umsteigt! Zur Suche nach einer möglichst günstigen Verbindung kann man die heuristische Suche einsetzen. Im folgenden werden die Kenntnisse über die heuristische Suche aus Kapitel 14 vorausgesetzt.

Als heuristische Bewertung nimmt man am besten die Ankunftszeit einer Verbindung und verwaltet die Liste der potentiellen Verbindungen in der prioritätsgesteuerten Warteschlange. Die Warteschlange wird mit dem Startbahnhof initialisiert. Mittels *findall* werden alle Direktverbindungen vom Startbahnhof aus bestimmt und nach der Ankunftszeit mittels *sortiere_liste_ein* in die Prioritätswarteschlange eingefügt.

Ankunftszeit als
heuristische Be-
wertung

Wenn der Zielbahnhof im Kopf der Prioritätswarteschlange steht, ist die zeitlich kürzeste Verbindung gefunden. Die Liste der zugehörigen Städte wird mit *reverse* umgekehrt, um die gefundene Verbindung bequem anzeigen zu können.

Entspricht die Stadt im Kopf der Warteschlange nicht dem Zielbahnhof, werden Umsteige-Verbindungen gesucht. Es werden alle Städte bestimmt, die von dieser Stadt aus direkt erreichbar sind, insgesamt also durch eine Umsteige-Verbindung erreichbar sind. Diese Liste wird wiederum Prioritätswarteschlange einsortiert. Beim Einsortieren wird darauf geachtet, daß von verschiedenen Verbindungen, die zur gleichen Stadt führen, nur die jeweils günstigste in der Warteschlange verbleibt. Ungünstigere Verbindungen werden aus der Warteschlange entfernt. Dies reduziert den Speicher- und Suchaufwand enorm.

Um mehrmaliges Benutzen derselben Linie zu verhindern, wird zu jeder Verbindung eine Liste der bisher benutzten Linien mitgeführt. Umsteigen wird nur auf bislang nicht benutzte Linien erlaubt.

Zyklustest für
Linien

Zur weiteren Verringerung des Speicheraufwands der heuristischen Suche, wird für jede Verbindung lediglich die Liste der bisher benutzten Bahnhöfe gespeichert. In Verbindung mit der Abfahrtszeit lassen sich aus dieser Liste später auch die Zugnummern, Abfahrts- und Ankunftszeiten ermitteln.

Programm für die
heuristische Suche

```

heuristischesuche(Start, Ziel, AbStart):-
    stadt(_, Start),
    stadt(_, Ziel),
    heuristischesuchel([ver(AbStart, [Start], [])],
                       Ziel, AbStart).

heuristischesuchel(Pfade, Ziel, AbStart):-
    Pfade = [ver(_, [Ziel|Rest], _)|_],
    reverse([Ziel|Rest], Verbindungen),
    zeige_kopf,
    zeige_verbindungen(Verbindungen, AbStart).

heuristischesuchel([Pfad|Pfade], Ziel, AbStart):-
    Pfad = ver(Ab, [Stadt1|Verbindungen], Linien),
    findall(ver(Ankunft, [Stadt2, Stadt1|Verbindungen],
              [Linie|Linien]),
            (linie(Linie, Stadt1),
             linie(Linie, Stadt2),
             not member(Linie, Linien),
             naechste_verbindung(Stadt1, Stadt2,
                                 Ab, _, _, Ankunft)),
            Verbindungen1),
    sortiere_liste_ein(Verbindungen1, Pfade, NeuePfade),
    heuristischesuchel(NeuePfade, Ziel, AbStart).
    
```

Das ICE-Streckennetz suggeriert, daß zwischen zwei Städten einer Linie genau eine Verbindung existiert. In Wirklichkeit sind es viele, denn eine Teilstrecke der Linie kann zu unterschiedlichen Zeiten befahren werden. Für die Fahrplanauskunft interessiert allerdings nur die zeitlich günstigste. Das Prädikat *naechste_verbindung* ermittelt zu zwei Städten einer Linie und dem frühest möglichen Abfahrtszeitpunkt *Ab*, die tatsächliche und günstigste Abfahrtszeit, die Zugnummer und die Ankunftszeit:

die nächste
Verbindung

```

naechste_verbindung(Stadt1, Stadt2, Ab,
                   Abfahrt, Nr, Ankunft):-
    fp(Nr, Abfahrt, Stadt1, ab),
    Ab @< Abfahrt,
    fp(Nr, Ankunft, Stadt2, an),
    Abfahrt @< Ankunft,
    !.
    
```

Die heuristische Suche liefert als Fahrplanauskunft für die Strecke München-Freiburg nun:

Reiseverbindung Bahnhof	Deutsche Bundesbahn Uhr	Zug
München	ab 7:46	ICE 598
Mannheim	an 10:32	
Mannheim	ab 10:58	ICE 771
Freiburg	an 12:27	

verbesserte
Zugauskunft durch
heuristische Suche

Man kann also eine halbe Stunde später losfahren, ist vier Stunden früher am Ziel und zahlt auch noch weniger, weil die Fahrstrecke insgesamt kürzer ist.

12.7 Aufgaben

1. Ergänzen Sie die Prädikate *direkte_verbindung_ab* und *umsteige_verbindung_ab* um die Berechnung und Ausgabe der Fahrtdauer.
2. Bei den Prädikaten *direkte_verbindung_ab* und *umsteige_verbindung_ab* wurde die früheste Abfahrtszeit vorgegeben. Entwickeln Sie entsprechende Prädikate, bei denen die späteste Ankunftszeit vorgegeben wird.
3. Gibt man die späteste Ankunftszeit vor, so kann es leicht passieren, daß man Verbindungen mit sehr frühen Abfahrtszeiten erhält. Entwickeln Sie Verfahren, mit denen möglichst späte Abfahrtszeiten gefunden werden.
4. Entwickeln Sie analog zum Abfahrtsplan einen Ankunftsplan.
5. Die Deutsche Bundesbahn gibt Städteverbindungen heraus, bei denen zu zwei vorgegeben Städten alle täglichen Verbindungen aufgeführt werden. Entwickeln Sie ein Prädikat zur Berechnung von Städteverbindungen.
6. Bestimmen Sie mit heuristischer Suche Zugverbindungen für späteste Ankunftszeiten.

13 Auskunfts- und Reisebuchungssystem

In Kapitel 4 haben wir ein Datenbankmodell für den Reiseveranstalter *Froh-Reisen* entworfen. Auf der Basis dieses Datenbankmodells soll nun ein Auskunfts- und Reisebuchungssystem erstellt werden, mit dem interaktiv Reiseziele ausgewählt und gebucht werden können.

Die vorhandene Datenbank wird dazu mit einer Benutzungsschnittstelle und Verwaltungskomponenten gekoppelt. Über die Benutzungsschnittstelle interagiert der Anwender mit dem Auskunfts- und Reisebuchungssystem, die Verwaltungskomponenten organisieren die Abfragen und Pflege der Datenbank.

Das so entstehende Auskunfts- und Reisebuchungssystem stellt wie das ICE-Auskunftssystem aus Kapitel 12 ein nicht triviales Informatiksystem dar. Analyse, Entwurf, Implementierung und Test eines solchen Systems sind typische informatische Aktivitäten, welche von Schülerinnen und Schüler in sinnhaftem Zusammenhang erlebt und ausgeführt werden können.

Benutzungsschnittstelle und Verwaltungskomponenten

Informatiksystem

13.1 Datenbankmodell

Das Datenbankmodell wurde in Kapitel 4 in Form mehrerer Prädikate entworfen. Zur besseren Übersicht lohnt es sich, ein Entity-Relationship-Diagramm zu zeichnen. Es faßt die Objekte und Beziehungen zwischen den Objekten in sehr anschaulicher Weise zusammen. Das Ergebnis ist auf der nächsten Seite zu sehen.

Entity-Relationship-Diagramm

Das Diagramm besteht aus fünf Objekttypen, dargestellt durch Rechtecke, und vier Beziehungen, dargestellt durch Rauten. In den Ovalen stehen die Attribute der Objekttypen beziehungsweise Beziehungen.

Objekttypen und Beziehungen

Ein Kunde wird durch die Kundennummer, Name und Adresse bestehend aus Straße, Postleitzahl und Ort beschrieben. Ein Hotel hat ebenfalls eine eindeutige Nummer, den Buchungscode, einen Namen und eine gewisse Anzahl von Sternen, welche über die Qualität des Hotels Auskunft gibt. Die Preise pro Person richten sich nach der Saison und der Anzahl der gebuchten Wochen.

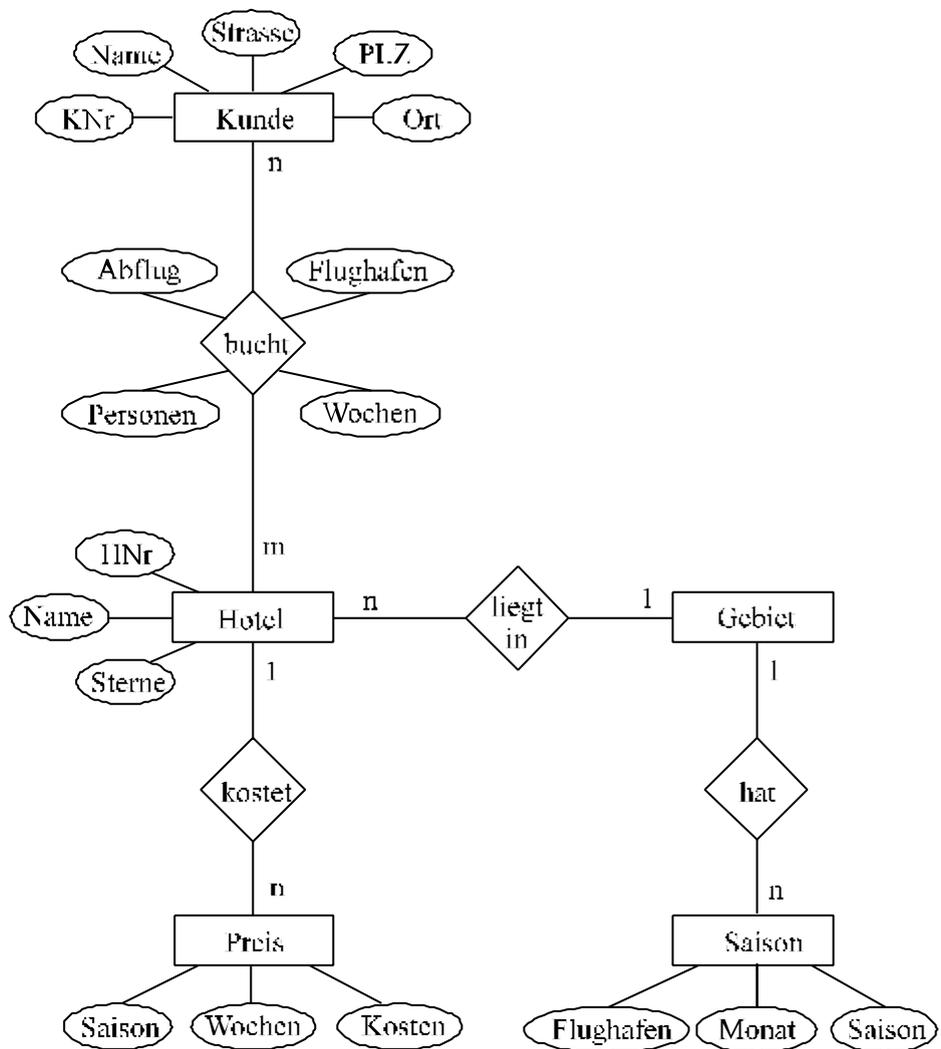
Hotels liegen in Gebieten, welche in Urlaubsmonaten ab bestimmten Flughäfen angeflogen werden. Die Saison richtet sich einerseits nach den klimatischen Verhältnissen im Urlaubsgebiet, andererseits nach dem Datum des Reiseantritts. In den Schulferien und Sommermonaten haben die Reiseveranstalter Hochsaison. Unsere Modellierung begnügt sich mit monatsweisem Saisonwechsel.

Die Beziehung *bucht* benötigt eigene Attribute, in denen der Abflugtermin, der Abflug-Flughafen, die Anzahl der reisenden Personen und die Dauer der

Reise in Wochen festgehalten wird. Auf eine Differenzierung hinsichtlich Erwachsene und Kinder wurde verzichtet. Das Buchungssystem könnte diesbezüglich realitätsnäher ausgebaut werden, weil normalerweise Kinderermäßigungen gewährt werden. Die *bucht*-Beziehung ist vom Grad n-m, da ein Hotel von mehreren Kunden gebucht werden kann und ein Kunde mehrere Hotels buchen kann, wenn er beispielsweise regelmäßig beim selben Reiseveranstalter bucht.

Die drei anderen Beziehungen sind vom Typ 1-n ein Hotel hat mehrere Preise, in einem Gebiet liegen mehrere Hotels und dieses Gebiet wird von verschiedenen Flughäfen aus in den Urlaubsmonaten angefliegen.

Abb. 13-1
ER-Diagramm des
Auskunfts- und
Reisebuchungs-
systems



Setzt man das Entity-Relationship-Diagramm in Relationen um, so benötigt man für die Objekttypen *Kunde*, *Hotel*, *Preis* und *Saison* eigene Relationen. Für die Gebiete benötigt man keine Relation, weil keine Attribute gespeichert werden. Das wäre anders, wenn man beispielsweise den Flughafen oder die Zeitverschiebung des Urlaubsgebiets berücksichtigen müßte. Eine weitere Relation fällt für die n-m-Beziehung *bucht* an, in welcher außer den Fremdschlüsseln Kunden- und Hotelnummer noch die Attribute der Beziehung selbst zu speichern sind.

Abbildung des ER-Diagramms auf Relationen

Die 1-n-Beziehungen werden durch Übernahme der Schlüsselattribute der 1-Seite in die Relation der n-Seite realisiert. Eigene Relationen sind hier nicht nötig. Insgesamt wird das ER-Diagramm also durch folgende fünf Relationen beschrieben:

```
kunde(KNr, Name, Strasse, PLZ, Ort).
hotel(HNr, Name, Sterne, Gebiet).
preis(HNr, Saison, Wochen, Kosten).
saison(Gebiet, Flughafen, Monat, Saison).
buchen(KNr, HNr, Abflug, Personen, Flughafen, Wochen).
```

Die Relationen des Auskunfts- und Reisebuchungssystems

13.2 Benutzungsschnittstelle

Die Benutzungsschnittstelle wird durch ein kleines Menü realisiert, über das die verschiedenen Verwaltungsfunktionen aktiviert werden können. Zu Beginn der Arbeiten wird das Auskunfts- und Reisebuchungssystem durch Setzen von Anfangswerten und Laden der Datenbank initialisiert. Zum Abschluß der Arbeiten wird die Datenbank mit den aktuellen Werten wieder gespeichert.

Menü als Benutzungsschnittstelle

```
start:-
    initialisierung,
    menue,
    terminierung.

initialisierung:-
    titel,
    retractall(wahl(_)),
    asserta(wahl(gebiet(_))),
    asserta(wahl(hotel(_))),
    asserta(wahl(kunde(_))),
    writeln('Lade Datenbank...'),
    tab(2), consult(frkunden),
    tab(2), consult(frbuchen),
    tab(2), consult(frhotel), nl.
```

Initialisierung der Datenbank

```

titel:-
  writeln('----- Buchungs- und Auskunftssystem -----'),
  writeln('----- Froh-Reisen GmbH, Darmstadt -----'),
  nl.

```

Hauptmenü

```

menue:-
  nl, writeln('Hauptmenü: '), nl,
  writeln(' 1 - Urlaubsgebiete'),
  writeln(' 2 - Hotels'),
  writeln(' 3 - Kunden'),
  writeln(' 4 - Buchen'),
  writeln(' 5 - neuer Kunde'),
  writeln(' 0 - Beenden'), nl,
  write(' Ihre Wahl: '),
  get_single_char(Ch),
  wahl_ausfuehren(Ch),
  Ch \== 0'0, !,
  menue.
menue.

```

Terminierung der
Datenbank

```

terminierung:-
  titel,
  writeln('Speichere Datenbank...'),
  write(' Kunden...'),
  rename_file('frkunden.pl', 'frkunden.bak'),
  tell('frkunden.pl'), listing(kunde), told,
  writeln(' ok!'),
  write(' Buchungen...'),
  rename_file('frbuchen.pl', 'frbuchen.bak'),
  tell('frbuchen.pl'), listing(buchung), told,
  writeln(' ok!'),
  retractall(wahl(_)).

```

End-Rekursion Die Menü-Schleife wird durch End-Rekursion realisiert. Die Rekursion terminiert bei Eingabe von 0.

Für jeden Menüpunkt gibt es eine *wahl_ausführen*-Klausel, welche das Menüsystem mit der betreffenden Verwaltungskomponente koppelt:

```

% --- Menüverwaltung -----

wahl_ausfuehren(0'1):-
    writeln(' Urlaubsgebiete'), nl,
    gebiete_zeigen,
    gebiet_waehlen.

wahl_ausfuehren(0'2):-
    writeln(' Hotels'), nl,
    linksbuendig('Nummer', 9),
    linksbuendig('Name', 15),
    linksbuendig('Kategorie', 10),
    linksbuendig('Gebiet', 10), nl,
    linie_zeichnen('-', 40), nl,
    hotels_zeigen,
    hotel_waehlen,
    weiter.

wahl_ausfuehren(0'3):-
    writeln(' Kunden'), nl,
    write('Name oder Nummer: '),
    lese_string(Kunde),
    bearbeite_kunde(Kunde),
    weiter.

wahl_ausfuehren(0'4):-
    writeln(' Buchen'), nl,
    hole_kunde(Kunde),
    hole_hotel(Hotel),
    buchen(Hotel, Kunde).

wahl_ausfuehren(0'5):-
    writeln(' neuer Kunde'), nl,
    retractall(wahl(_)),
    asserta(wahl(gebiet(_))),
    asserta(wahl(hotel(_))),
    asserta(wahl(kunde(_))).

wahl_ausfuehren(_).

```

Schnittstelle
zwischen
Benutzungs- und
Verwaltungs-
komponenten

13.3 Gebietsverwaltung

Anzeige und
Auswahl von
Urlaubsgebieten

Im Rahmen der Gebietsverwaltung sollen Urlaubsgebiete angezeigt und ausgewählt werden können. Da wir für die Urlaubsgebiete keine eigene Relation haben, bestimmen wir die verfügbaren Gebiete mittels *findall* aus der *hotel*-Relation. Weil dort aber ein Gebiet in der Regel mehrmals vorkommt, müssen die Daten vor der Ausgabe gefiltert werden. Dies erledigt das *sort*-Prädikat von TV-SWI-Prolog. Es sortiert eine Liste und eliminiert dabei die Doubletten:

```
% --- Gebietsverwaltung -----
gebiete_zeigen:-
    findall(Gebiet, hotel(_, _, _, Gebiet), Listel),
    sort(Listel, Liste2),
    gebiete_zeigen(Liste2),

gebiete_zeigen([K|R]):-
    tab(2), writeln(K),
    gebiete_zeigen(R).
gebiete_zeigen([]).
```

Gebiete ermitteln
und anzeigen

Nach der Anzeige hat der Benutzer die Möglichkeit, ein Urlaubsgebiet auszuwählen.

Gebiet
auswählen

```
gebiet_waehlen:-
    nl, write('Gewünschtes Gebiet: '),
    lese_string(Gebiet), nl,
    bearbeite_gebiet(Gebiet).

%-- kein Gebiet gewählt
bearbeite_gebiet('').

%-- vorhandenes Gebiet
bearbeite_gebiet(Gebiet):-
    hotel(_, _, _, Gebiet),
    retract(wahl(gebiet(_))),
    asserta(wahl(gebiet(Gebiet))).
```

Speicherung des
erfragten Wissens

Das ausgewählte Gebiet wird als *wahl*-Fakt mit dem Argument *gebiet(Gebiet)* gespeichert. Bei der nachfolgenden Hotelverwaltung kann die Anzeige auf die Hotels des ausgewählten Gebiets beschränkt werden.

13.4 Hotelverwaltung

Die Hotelverwaltung kann die Hotels des ausgesuchten Gebiets anzeigen und den Benutzer eine Hotelwahl durchführen lassen. Darüber hinaus stellt sie Operationen zum Anzeigen und Auswahl eines Hotels zur Verfügung.

Hotels anzeigen
und auswählen

Das gewünschte Hotel kann wahlweise über die Hotelnummer oder über den Hotelnamen angegeben werden. Die Hotelverwaltung entscheidet, ob ein gültige Hotelauswahl vorgenommen wurde und speichert das Ergebnis wie zuvor beim Gebiet in einem *wahl*-Fakt, diesmal mit dem Argument *hotel*(*HotelNummer*).

```
hotel_waehlen:-
    nl, write('gewünschtes Hotel: '),
    lese_string(Hotel), nl,
    bearbeite_hotel(Hotel).
```

```
%-- kein Hotel gewaehlt
bearbeite_hotel(''):- !.
```

```
%-- Hotelname
bearbeite_hotel(Hotel):-
    hotel(HotelNr, Hotel, _, _),
    retract(wahl(hotel(_))),
    asserta(wahl(hotel(HotelNr))),
    zeige_hotel(HotelNr), !.
```

Auswahl über
Hotelname

```
%-- Hotelnummer
bearbeite_hotel(Hotel):-
    hotel(Hotel, _, _, _),
    retract(wahl(hotel(_))),
    asserta(wahl(hotel(Hotel))),
    zeige_hotel(Hotel), !.
```

Auswahl über
Hotelnummer

Wenn der Benutzer im Hauptmenü die Buchungsverwaltung aufruft, wird unter anderem das ausgewählte Hotel aus der Wissensbasis geholt und zur Kontrolle angezeigt. Wurde noch keine Hotel ausgewählt, so wird automatisch die Hotelverwaltung zur Auswahl eines Hotels aufgerufen:

```
hole_hotel(HotelNr):-
    wahl(hotel(HotelNr)), var(HotelNr),
    wahl_ausfuehren(0'2),
    fail.
hole_hotel(HotelNr):-
    wahl(hotel(HotelNr)), nonvar(HotelNr),
    zeige_hotel(HotelNr).
```

13.5 Kundenverwaltung

Kunden auswählen
und erfassen

Die Auswahl eines Kunden kann ebenfalls über die Nummer oder den Namen erfolgen. Im Unterschied zur Hotelverwaltung können allerdings neue Kunden erfaßt werden. Dies geschieht automatisch, wenn der Benutzer einen neuen Namen eingibt.

Die Nummer des neuen Kunden wird durch Addition von 1 zur bislang größten Kundennummer errechnet. Dabei wird das Systemprädikat *last* von TV-SWI-Prolog genutzt.

Eingabe der
Kundendaten

```
bearbeite_kunde(Kunde):-
    atom(Kunde),
    writeln('Adresse des neuen Kunden: '), nl,
    write('Strasse und Nr.: '), lese_string(Strasse),
    write('Postleitzahl   : '), lese_string(PLZ),
    write('Ort           : '), lese_string(Ort),
    findall(KNr, kunde(KNr,_,_,_,_), Listel),
    sort(Listel, Liste2),
    last(KNr1, Liste2),
    KundenNr is KNr1 + 1,
    speicher_kunde(kunde(KundenNr,Kunde,Strasse,PLZ,Ort)),
    !.
```

Aufnahme der
Kundendaten
in die Datenbank

```
speicher_kunde(kunde(KundenNr,Kunde,Strasse,PLZ,Ort)):-
    nl, write('Adresse korrekt (ja/nein): '),
    lese_string(Antwort),
    Antwort = 'ja',
    assert(kunde(KundenNr, Kunde, Strasse, PLZ, Ort)),
    retract(wahl(kunde(_))),
    asserta(wahl(kunde(KundenNr))).
speicher_kunde(_).
```

Die Nummer des ausgewählten Kunden wird ebenfalls in einem *wahl*-Fakt gespeichert, diesmal mit dem Argument *kunde(Kundennummer)*.

13.6 Buchungsverwaltung

Mit der Buchungsverwaltung kann man für einen Kunden ein bestimmtes Hotel buchen. Zunächst werden das Abflugdatum und der Abflug-Flughafen erfaßt. Wenn zum gewünschten Abflugdatum kein Abflug möglich ist, werden falls möglich Ausweichflughäfen angeboten.

```
buchen(Hotel, Kunde):-
    nl, writeln('Abflug'),
    write('am: '), lese_datum(Tag, Monat, Jahr),
    write('ab: '), lese_string(Flughafen),
    bestimme_saison(Hotel, Flughafen, Monat, Saison), nl,
    write('Personen: '), lese_zahl(Personen),
    write('Wochen : '), lese_zahl(Wochen),
    bestimme_preis(Hotel, Personen, Wochen, Saison, Preis), nl,
    write('Gesamtpreis: '), write(Preis), writeln(' DM'),
    write('Buchen (ja/nein): '), lese_string(Buchen),
    Buchen = 'ja',
    asserta(buchung(Kunde, Hotel, Personen,
        datum(Tag, Monat, Jahr), Wochen, Flughafen)),
    writeln('Gebucht.').
buchen(_, _).
```

Eingabe der
Buchungsdaten

Aufnahme der
Buchungsdaten
in die Datenbank

```
bestimme_saison(Hotel, Flughafen, Monat, Saison):-
    hotel(Hotel, _, _, Gebiet),
    saison(Gebiet, Flughafen, Monat, Saison), !.
bestimme_saison(Hotel, _Flughafen, Monat, Saison):-
    hotel(Hotel, _, _, Gebiet),
    findall(FHafen,
        saison(Gebiet, FHafen, Monat, Saison), FHafen),
    (FHafen = [] ->
        write('Abflug nicht möglich. Keine Saison in '),
        writeln(Gebiet);
        write('Abflug möglich ab: '), writeln(FHafen)),
    fail.
```

Danach werden die Personenzahl und Reisedauer erfaßt und aus den eingegebenen Daten und der Preis-Datenbank die Kosten der Reise errechnet. Nach einer Bestätigung kann Buchung ausgeführt werden.

```
bestimme_preis(Hotel, Personen, Wochen, Saison, Preis):-
    preise(Hotel, Saison, Wochen, Kosten),
    Preis is Personen * Kosten, !.
```

Ermittlung des
Reisepreises

13.7 Aufgaben

1. Analysieren Sie die Prädikate *initialisierung* und *terminierung* hinsichtlich
 - a) der Behandlung der Datenbank,
 - b) der Behandlung der *wahl*-Fakten.
2. Der Benutzer soll informiert werden, wenn er ein nicht vorhandenes Gebiet gewählt hat. Ergänzen Sie eine betreffende *bearbeite_gebiet*-Klausel.
3. Implementieren Sie die Anzeige der Hotels im ausgewählten Urlaubsgebiet.
4. Bei ungültigen Adressen sollen Buchungen nicht durchgeführt werden.
5. Informieren Sie den Benutzer, wie viele Wochen ein Hotel gebucht werden kann, wenn er eine unzulässige Wochenzahl buchen will.
6. Welche Erweiterungsmöglichkeiten sehen Sie?

14 Suchverfahren

14.1 Graphen

Historisch gesehen spielte die Untersuchung von Strategiespielen in der Künstlichen Intelligenz eine wichtige Rolle, weil in diesen abgeschlossenen Miniwelten sich die Wirksamkeit intelligenter Problemlösemethoden gut studieren läßt. Als Ergebnis dieser Untersuchungen gibt es heute unter anderem Schach-Programme gegen die der amtierende Schachweltmeister sich sehr anstrengen muß, um nicht zu verlieren, und gegen die Vereinsspieler keine Chance mehr haben.

Problemlösemethoden der künstlichen Intelligenz

Schach-Programme und auch Programme für andere Strategiespiele, wie zum Beispiel Reversi oder Dame, beruhen im wesentlichen auf systematischen Suchverfahren. Auch bei der Sprachverarbeitung, dem Planen, dem Beweis mathematischer Aussagen, der Mustererkennung und sonstige Problemfeldern spielen Suchverfahren eine entscheidende Rolle.

Am Beispiel der Pfadsuche in einem Graphen betrachten wir drei grundlegende Suchverfahren:

- Tiefensuche
- Breitensuche
- Heuristische Suche

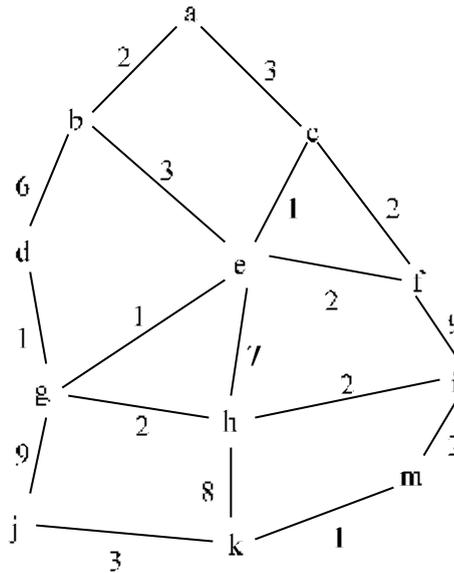
grundlegende Suchverfahren

Einerseits ist das Graphenbeispiel elementar genug, um nicht durch technische Details unnötig vom eigentlichen algorithmischen Kern der verschiedenen Suchverfahren abzulenken. Andererseits ist aber der Graph doch so allgemein, daß auch Spezialfälle wie Baum und Netzwerk darin enthalten sind. Die Diskussion der Suchverfahren für Pfade in Graphen gibt Ihnen das notwendige Rüstzeug, um auch sonstige graphentheoretische Probleme lösen zu können.

Die weiteren Beispiele dieses Kapitels zum Lösen von Puzzle-Aufgaben machen deutlich, wie sich Problemlösemethoden von konkreten Graphen auf abstrakte Zustandsgraphen übertragen lassen.

konkrete Graphen und abstrakte Zustandsgraphen

Abb. 14-1
ungerichteter,
bewerteter Graph



Im Folgenden beziehen wir uns auf ungerichteten Graphen von Abbildung 14-1. Die Kantenbewertungen spielen erst bei der heuristischen Suche eine Rolle. Sie gehen in die Berechnung der heuristischen Bewertung ein.

Die Darstellung eines Graphen in Prolog mittels Fakten ist sehr einfach. Wir führen in alphabetischer Reihenfolge alle Kanten auf:

Repräsentierung
eines gerichteten
und bewerteten
Graphen

```
kante(a, b, 2).    kante(a, c, 3).
kante(b, d, 6).    kante(b, e, 3).
...
kante(j, k, 3).
kante(k, m, 1).
```

Die Kanten-Fakten repräsentieren einen gerichteten und bewerteten Graphen. Den ungerichteten und unbewerteten Graphen erhalten wir durch:

Repräsentierung
eines ungerichteten
und unbewerteten
Graphen

```
verbunden(A, B):- kante(A, B, _).
verbunden(A, B):- kante(B, A, _).
```

14.2 Tiefensuche

Die Tiefensuche läßt sich in Prolog am einfachsten implementieren, weil der Prolog-Interpreter selbst mit Tiefensuche arbeitet. Trifft man keine geeigneten Vorkehrungen, so läuft die Tiefensuche im Graphen schnell in einen unendlichen Zyklus, zum Beispiel a, b, a, b, a, b, \dots . Zur Vermeidung von Zyklen muß man sich merken, welche Knoten schon besucht wurden. Ein Pfad kann nur dann um einen weiteren Knoten expandiert werden, wenn der Knoten nicht schon im Pfad enthalten ist. Nur in Sonderfällen, wie zum Beispiel im gerichteten Baum, kann der Zyklustest entfallen.

Tiefensuche mit
Zyklustest

Die Benutzungsschnittstelle der Tiefensuche sieht so aus:

```
tiefensuche(+Startknoten, +Zielknoten, -Lösungspfad)
```

Die Anfrage `?- tiefensuche(a, k, L)` führt zur Suche nach Pfaden von a nach k und liefert die Lösungen in der Variablen L . Wir implementieren die Tiefensuche wie folgt:

```
tiefensuche(Start, Ziel, Loesung):-
    tiefensuche(Start, [Start], Ziel, Loesung).
tiefensuche(Ziel, Pfad, Ziel, Loesung):-
    Loesung = Pfad, !.
tiefensuche(KnotenA, Pfad, Ziel, Loesung):-
    verbunden(KnotenA, KnotenN),
    not member(KnotenN, Pfad),
    tiefensuche(KnotenN, [KnotenN|Pfad], Ziel, Loesung).
```

Implementierung
der Tiefensuche

Das Prädikat `tiefensuche/3` sorgt für den korrekten Aufruf von `tiefensuche/4`. Im ersten Argument erhält `tiefensuche_4` den aktuellen Knoten, im zweiten Argument eine Liste mit dem Pfad vom Startknoten zum aktuellen Knoten. Diese Liste muß man allerdings von rechts nach links lesen, um den tatsächlichen Pfad zu erhalten. Die Pfadliste benutzen wir für den Zyklustest und zur Ausgabe der Lösung.

$L = [k, j, g, d, b, a]$ ist der Pfad $a \rightarrow b \rightarrow d \rightarrow g \rightarrow j \rightarrow k$

Beispiel einer
Pfadliste

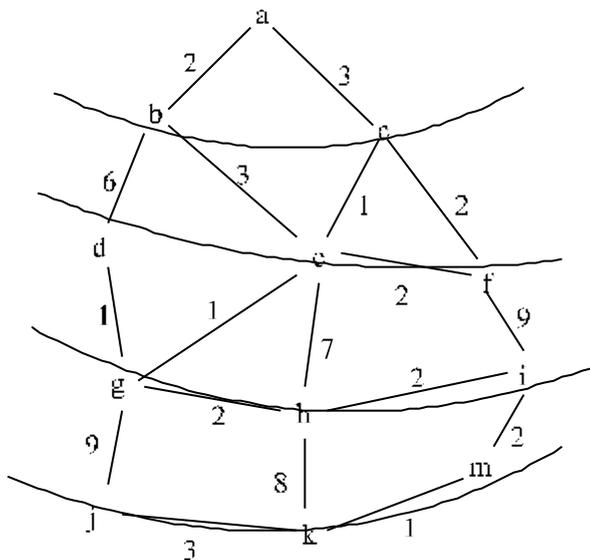
Stimmt der aktuelle Knoten mit dem gesuchten Ziel überein, so ist eine Lösung gefunden. Anderenfalls suchen wir einen Nachfolgerknoten zum aktuellen Knoten. Ist dieser nicht im aktuellen Pfad enthalten, so wird der aktuelle Pfad um den neuen Knoten erweitert und die Tiefensuche fortgesetzt. Da Prolog automatisch Backtracking durchführt, müssen wir uns nicht explizit um Alternativen kümmern oder bei Fehlversuchen zurücksetzen.

14.3 Breitensuche

zweidimensionaler
Suchhorizont der
Breitensuche

Die Tiefensuche ist uns von Prolog und von binären Suchbäumen her wohlbe-
kannt. Sie hat einen eindimensionalen Suchhorizont, das heißt die Suche geht
immer von aktuellen Knoten aus weiter. Im Gegensatz dazu hat die Breitensuche
einen zweidimensionalen Suchhorizont. Man kann sich vorstellen, daß am
Startknoten eine Welle gestartet wird, die sich schrittweise im Graphen aus-
breitet. Die Wellenfront gibt die aktuellen Knoten an, von denen aus die Suche
fortgesetzt wird.

Abb. 14-2
Wellenfronten der
Breitensuche



Der Reihe nach werden also besucht

- a
- b, c
- d, e, f
- g, h, i
- j, k, m

Soll beispielsweise ein Pfad von a nach g gesucht werden, so ist die Front der
Suchwelle nach zwei Schritten bis d, e und f vorangekommen. Von d als auch
von e aus kann man im nächsten Schritt das Ziel g erreichen. Um dann eine
Lösung ausgeben zu können, muß man den Pfad von a nach d beziehungsweise
e gemerkt haben.

Speicherung
der Suchpfade

Wir müssen uns also für jeden Knoten der aktuellen Suchfront den Pfad
vom Start zum Knoten merken. Der Grund, daß Prolog mit Tiefen- und nicht
mit Breitensuche arbeitet, ist im enormen Aufwand zur Speicherung der Pfade

zu den Knoten der Suchfront zu sehen. Die Speicherung der Pfade in unserem Graphenbeispiel erfolgt analog zur Tiefensuche, nur müssen wir uns nicht einen Pfad, sondern in einer Liste mehrere Pfade merken. Zum Beispiel für die Suchfronten

```
a      : [[a]]
b, c  : [[b,a], [c,a]]
```

Aus der alten Suchfront entsteht schrittweise eine neue Suchfront. Man entnimmt den ersten Pfad [b, a] und bestimmt zu dessen aktuellen Kopfknoten alle Nachfolgeknoten: d und e. Die sich daraus ergebenden neuen Pfade [d,b,a] und [e,b,a] hängt man an die Liste der noch zu untersuchenden Pfade an: [[c,a], [d,b,a], [e,b,a]].

Dieses Verfahren wiederholt man. Also die Nachfolger von c sind e und f, woraus sich die beiden neuen Pfade [e,c,a] und [f,c,a] ergeben. Im nächsten Schritt sind die Pfade [[d,b,a], [e,b,a], [e,c,a], [f,c,a]] zu untersuchen. Es geht demnach mit d weiter.

Die Implementierung der Breitensuche läuft analog zur Implementierung der Tiefensuche. Es wird aber nicht ein Pfad, sondern eine Liste von Pfaden verwaltet und es werden nicht ein Nachfolger, sondern stets alle Nachfolger (findall) eines aktuellen Knotens berücksichtigt. Der relevante Unterschied zur Tiefensuche besteht darin, daß es nicht gleich mit den neuen Knoten weitergeht, sondern die neuen Knoten mit ihren zugehörigen Pfaden sich in eine Warteschlange einreihen müssen (append).

```
breitensuche(Start, Ziel, Loesung):-
    breitensuche([Start], [], Ziel, Loesung).

breitensuche(Pfad, _, Ziel, Loesung):-
    Pfad = [Ziel|_],
    Loesung = Pfad.

breitensuche(Pfad, Pfade, Ziel, Loesung):-
    Pfad = [KnotenA|_],
    findall([KnotenN|Pfad],
           (verbunden(KnotenA, KnotenN),
            not member(KnotenN, Pfad)),
           GefundenePfade),
    append(Pfade, GefundenePfade, NeuePfade),
    NeuePfade = [PfadN|RestPfade],
    breitensuche(PfadN, RestPfade, Ziel, Loesung).
```

Implementierung
der Breitensuche

Im ersten Argument von *breitensuche/4* wird der aktuelle Pfad übergeben. Mit *Pfad = [KnotenA|_]* wird aus dem Kopf des aktuellen Pfades der aktuelle Knoten bestimmt.

Bei der Tiefensuche wurde der aktuelle Knoten stattdessen eigenständig verwaltet. Mittels *findall* werden alle Nachfolgerknoten bestimmt.

zur Arbeitsweise
von findall

Zur Erinnerung *findall(+Term, +Ziel, -Liste)* sammelt alle Lösungsterme *Term* eines *Ziels* in einer *Liste*. Damit nur zyklensfreie neue Pfade berechnet werden, wählen wir das zusammengesetzte Teilziel (*verbunden (KnotenA, KnotenN), not member(KnotenN, Pfad)*). Lösungen dieses Teilziels bestehen aus Variablenwerten, welche wir zu einer neuen Liste zusammensetzen: *[KnotenN|Pfad]*. Jede solche Liste stellt einen neuen Pfad dar. Alle möglichen Lösungen werden von *findall* in der Variablen *GefundenePfade* gesammelt. Diese Liste wird im Sinne einer Warteschlange an die Liste der verbliebenen Pfade angehängt. Die Suche geht mit der so gebildeten neuen Liste wartender Pfade weiter.

Sollen mit der Breitensuche nicht nur der kürzeste, sondern auch längere Pfade gefunden werden, so müssen wir eine weitere Regel ergänzen und den Cut aus der vorangehenden Klauseln entfernen:

Breitensuche für
mehrere Lösungen

```
breitensuche(Pfad, Pfade, Ziel, Loesung):-
    Pfad = [Ziel|_], !,
    Pfade = [Pfad|PfadeN],
    breitensuche(Pfad, PfadeN, Ziel, Loesung).
```

14.4 Heuristische Suche

Heuristische
Suche durch Prioritätswarteschlange

Bei der Breitensuche haben wir neue Pfade einfach an die Warteschlange der noch zu untersuchenden Pfade angehängt. Machen wir aus der Warteschlange eine Prioritätswarteschlange, so entsteht die heuristische Suche. Zum Einfügen in eine Prioritätswarteschlange müssen wir jedem Pfad eine Priorität zuordnen. Sie wird mit der heuristischen Bewertungsfunktion berechnet. Durch die Einführung einer Heuristik werden günstige Pfade bevorzugt untersucht, während die Untersuchung von Pfaden mit geringer Priorität, also schlechter Bewertung zurückgestellt wird.

Es ist in der Regel nicht einfach, eine gute Heuristik zu finden, da jede Heuristik auf das zu untersuchende Problem abgestimmt sein muß. Schachstellungen zu bewerten ist weitaus schwieriger als Pfade in unserem Graphen. Hier nehmen wir als Bewertung einfach die Summe der Kantenbewertungen.

Datenstruktur
pfad(Pfad, Bewertung)

Um mit den Kantenbewertungen sinnvoll arbeiten zu können, ändern wir unsere Datenstruktur ab. Wir bilden einen Verbund namens *pfad*, der aus dem eigentlichen Pfad als Liste von Knoten und der Pfadbewertung besteht.

Beispiel

pfad([g,d,b,a], 9) ist der Pfad *[g,b,d,a]* mit der Bewertung 9.

Im Unterschied zur Breitensuche müssen wir bei der heuristischen Suche bei neuen Pfaden die Bewertungen berechnen und sie dann gemäß der Bewertung in die Prioritätswarteschlange einfügen. Damit ergibt sich folgende Lösung:

```

heuristischesuche(Start, Ziel, Loesung):-
    heuristischesuche(pfad([Start], 0), [], Ziel, Loesung).

heuristischesuche(Pfad, _, Ziel, Loesung):-
    Pfad = pfad([Ziel|_], _),
    Loesung = Pfad, !.

heuristischesuche(Pfad, Pfade, Ziel, Loesung):-
    Pfad = pfad([KnotenA|PfadA], KostenA),
    findall(pfad([KnotenN, KnotenA|PfadA], KostenN),
            (verbunden(KnotenA, KnotenN),
             not member(KnotenN, PfadA),
             bewerte_heuristisch(KnotenA, KostenA,
                                 KnotenN, KostenN))),
            GefundenePfade),
    sortiere_liste_ein(GefundenePfade, Pfade, NeuePfade),
    NeuePfade = [PfadN|RestPfade],
    heuristischesuche(PfadN, RestPfade, Ziel, Loesung).

```

Implementierung
der heuristischen
Suche

Soll beispielsweise die heuristische Suche mit dem $pfad([b, a], 2)$ fortgesetzt werden, so ermittelt *findall* die Lösung $GefundenePfade = [pfad([d, b, a], 8), pfad([e, b, a], 5)]$. Die heuristischen Bewertungen werden also gleich mitgeliefert, weil das Ziel von *findall* um das Teilziel *bewerte_heuristisch* ergänzt wurde. Die gefundenen Pfade werden gemäß ihrer Bewertung über *sortiere_liste_ein* in die Prioritätswarteschlange *Pfade* einsortiert. Mit dem Kopfelement *PfadN* der Prioritätswarteschlange wird die heuristische Suche fortgesetzt.

Das Prädikat *bewerte_heuristisch* ermittelt aus den bisherigen Pfadkosten und den hinzukommenden Kosten der neuen Kante die neuen Pfadkosten:

```

bewerte_heuristisch(KnotenA, KostenA,
                    KnotenN, KostenN):-
    kostet(KnotenA, KnotenN, KantenKosten),
    KostenN is KostenA + KantenKosten.

```

heuristische
Bewertung

Dabei wird das Hilfsprädikat *kostet* benutzt, das aus dem unbewerteten einen bewerteten Graphen macht.

```

kostet(A, B, Kosten):- kante(A, B, Kosten).
kostet(A, B, Kosten):- kante(B, A, Kosten).

```

Das Einfügen mehrere Elemente in eine Prioritätswarteschlange erledigen wir sukzessive durch Einfügen jeweils eines Elements in die Warteschlange. Auch hier kommt wieder die *Kopf-Rest-Methode* zum Zuge:

Kopf-Rest-Methode
zum Einfügen

```
sortiere_liste_ein([], Liste, Liste).
sortiere_liste_ein([K|R], Liste1, Liste3):-
    sortiere_element_ein(K, Liste1, Liste2),
    sortiere_liste_ein(R, Liste2, Liste3).
```

Abschließend muß noch das Prädikat *sortiere_element_ein* (+Element, +Schlange1, -Schlange2) implementiert werden, welches ein *Element* gemäß seiner Priorität in die Prioritätswarteschlange *Schlange1* einfügt und das Ergebnis in *Schlange2* liefert. Dies erledigen Sie in Aufgabe 4c.

Interpretieren wir die Kantenkosten als Länge, so liefert die so definierte heuristische Suche den kürzesten Weg von Start nach Ziel.

14.5 Hüpf-Schiebe-Puzzle

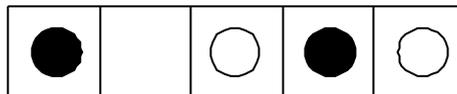
Wir wenden die Suchverfahren auf das Hüpf-Schiebe-Puzzle an. Dieses Vorgehen entspricht der Herangehensweise der klassischen künstlichen Intelligenz an Probleme. Man studiert Problemlösestrategien an überschaubaren Spielsituationen, in der Hoffnung, die dabei erzielten Ergebnisse später auf Anwendungssituationen mit Erfolg übertragen zu können.

Problemlösen als
Pfadsuche im
Zustandsraum-
Graphen

Das Beispiel des Hüpf-Schiebe-Puzzle zeigt, daß man Problemlösen als Pfadsuche im Zustandsraum-Graphen auffassen kann. Eine Spielsituation stellt einen Knoten im Zustandsraum-Graphen dar. Ein Zug führt von einer Spielsituation zu einer anderen, stellt also eine gerichtete Kante zwischen zwei Knoten im Zustandsraum-Graphen dar. Eine Lösung besteht aus einer Folge von Zügen, welche schrittweise die Anfangsposition in die Zielposition transformiert. Im Zustandsraum-Graphen entspricht die Lösung daher einem Pfad vom Anfangszustand zum Endzustand.

Das Spielfeld des Hüpf-Schiebe-Puzzles besteht aus einem 5 x 1-Spielfeld mit 2 weißen und 2 schwarzen Spielmarken. Ein Platz auf dem Spielfeld bleibt unbelegt.

Abb. 14-3
Ein Anfangszu-
stand im Hüpf-
Schiebe-Puzzle



Es gibt zwei Hüpf- und Schiebezüge. Bei einem Hüpfzug hüpf eine Spielmarke über genau eine andere Spielmarke in das freie Feld, bei einem Schiebezug verschiebt man eine Spielmarke in das direkt benachbarte freie Feld.

In Abbildung 13-3 kann die rechte schwarze Spielmarke über die linke weiße Spielmarke auf das freie Spielfeld hüpfen. Sowohl die linke weiße, als auch die linke schwarze Spielmarke können auf das freie Spielfeld verschoben werden.

Das Ziel des Hüpf-Schiebe-Puzzles besteht darin, mit möglichst wenigen Zügen von einem Startzustand zum Zielzustand zu kommen:

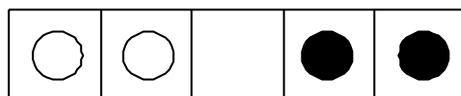


Abb. 14-4
Zielzustand im
Hüpf-Schiebe-
Puzzle

Zur Lösung des Hüpf-Schiebe-Puzzles mit dem Computer müssen die Spielsituationen durch eine geeignete Datenstruktur dargestellt werden. Wir nehmen dazu einfach eine 5-elementige Liste, in welcher eine weiße Spielmarke durch w, eine schwarze durch s und das leere Spielfeld durch l dargestellt wird. Die Endposition wird also durch [w,w,l,s,s] beschrieben.

Datenstruktur für
das Hüpf-Schiebe-
Puzzle

14.6 Hüpf- und Schiebe-Züge

Im Hüpf-Schiebe-Puzzle gibt es Hüpf- oder Schiebezüge. Jeder Zug bewirkt einen Übergang vom aktuellen Zustand zu einem neuen Zustand. Wir beschreiben Züge durch das Prädikat:

```
zug(+ZustandA, -ZustandN).
```

Zustandsübergänge

wobei *ZustandA* der aktuelle und *ZustandN* der neue Zustand ist. Die vier möglichen Züge verteilen wir auf vier verschiedene Klauseln. Zur Ausführung eines Zuges achtet man am besten auf das leere Feld und nicht auf die Spielmarken. Das leere Feld kann ein oder zwei Felder nach rechts oder links gezogen werden:

```
zug(ZustandA, ZustandN):- /* l nach rechts schieben */
    append(X, [l, K|R], ZustandA),
    append(X, [K, l|R], ZustandN).
```

Hüpf- und
Schiebe-Züge

```

zug(ZustandA, ZustandN):- /* 1 nach rechts hüpfen */
    append(X, [1, K1, K2|R], ZustandA),
    append(X, [K2, K1, 1|R], ZustandN).

zug(ZustandA, ZustandN):- /* 1 nach links schieben */
    append(X, [K, 1|R], ZustandA),
    append(X, [1, K|R], ZustandN).

zug(ZustandA, ZustandN):- /* 1 nach links hüpfen */
    append(X, [K1, K2, 1|R], ZustandA),
    append(X, [1, K2, K1|R], ZustandN).
    
```

14.7 Tiefensuche für das Hüpf-Schiebe-Puzzle

Da die Tiefensuche im Zustandsraum-Graphen in Zyklen geraten kann, muß zusätzlich zum aktuellen Zustand auch der Pfad, der vom Anfangszustand zum aktuellen Zustand geführt hat, bekannt sein. Diese beiden Angaben übernimmt das Prädikat *tiefensuche(+Zustand, +Pfad)* in den beiden Argumenten. Die erste *tiefensuche/2*-Klausel prüft, ob der aktuelle Zustand der Endzustand ist und gibt gegebenenfalls die Lösung als Liste der Zustände, die vom Anfangszustand zum Endzustand führen, aus. Die zweite *tiefensuche/2*-Klausel führt einen Zug aus, prüft ob der Zug zu keinem Zyklus führt und setzt mittels Rekursion die Suche mit dem neuen Zustand und neuen Pfad fort:

Implementierung
der Tiefensuche für
das Hüpf-Schiebe-
Puzzle

```

tiefensuche(Anfangszustand):-
    tiefensuche(Anfangszustand, [Anfangszustand]).

tiefensuche(Zustand, Pfad):-
    endzustand(Zustand),
    zeige(Pfad).

tiefensuche(ZustandA, Pfad):-
    zug(ZustandA, ZustandN),
    not member(ZustandN, Pfad),
    tiefensuche(ZustandN, [ZustandN|Pfad]).

endzustand([w,w,l,s,s]).

zeige(Loesung):-
    write('Lösung: '), nl, zeige_zuege(Loesung).

zeige_zuege([]).
zeige_zuege([K|R]):- zeige_zuege(R), write(K), nl.
    
```

14.8 Breitensuche für das Hüpf-Schiebe-Puzzle

Die Prädikate *endzustand*, *zeige* und *zeige_zuege* können ohne Änderung von der Tiefensuche übernommen werden. Während bei der Tiefensuche Zustand und Pfad zu diesem Zustand in getrennten Argumenten verwaltet wurden, fassen wir nun Zustand und Pfad zusammen. Bei Bedarf muß dann der aktuelle Zustand als Kopf der Pfadliste ermittelt werden. Die Warteschlange, in welcher die Breitensuche die Lösungskandidaten verwaltet, wird im zweiten Argument Prädikats *breitensuche/2* verwaltet.

```
breitensuche(+Pfad, +Warteschlange).
```

findall leistet die wesentliche Arbeit der Breitensuche: es sucht nicht zyklische Züge, faßt die neuen Zustände und Pfade zu neuen Pfaden [*ZustandN|Pfad*] zusammen und sammelt alle gefundenen Pfade in einer Liste namens *GefundenePfade* auf. Im Sinne einer Warteschlange wird diese Liste an die bisherige Warteschlange mittels *append* angehängt. Mit dem Kopfelement der Warteschlange wird die Breitensuche fortgesetzt:

```
breitensuche(Anfangszustand):-
    breitensuche([Anfangszustand], []).

breitensuche(Pfad, _):-
    Pfad = [Zustand|_],
    endzustand(Zustand),
    zeige(Pfad).

breitensuche(Pfad, Pfade):-
    Pfad = [ZustandA|_],
    findall([ZustandN|Pfad],
            (zug(ZustandA, ZustandN),
             not member(ZustandN, Pfad)),
            GefundenePfade),
    append(Pfade, GefundenePfade, NeuePfade),
    NeuePfade = [PfadN|RestPfade],
    breitensuche(PfadN, RestPfade).
```

Implementierung
der Breitensuche
für das Hüpf-
Schiebe-Puzzle

14.9 Heuristische Suche für das Hüpf-Schiebe-Puzzle

Die heuristische Suche läuft ähnlich ab, wie die Breitensuche, nur müssen wir zusätzlich jede Puzzlestellung bewerten und die Stellungen in der Rangfolge der Bewertungen untersuchen. Zur Verwaltung der zusätzlichen Bewertung ergänzen wir die Pfade um die Bewertung des Kopfknotens und verwalten beide Daten in *pfad*-Strukturen.

Implementierung
der Heuristischen
Suche für das
Hüpf-Schiebe-
Puzzle

```

heuristischesuche(Anfangszustand):-
    heuristischesuche(pfad([Anfangszustand], _), []).

heuristischesuche(Pfad, _):-
    Pfad = pfad([Ziel|PfadRest], _),
    endzustand(Ziel),
    zeige([Ziel|PfadRest]).

heuristischesuche(Pfad, Pfade):-
    Pfad = pfad([ZustandA|PfadA], _),
    length(PfadA, N),
    findall(pfad([ZustandN, ZustandA|PfadA], BewertungN),
        (zug(ZustandA, ZustandN),
         not member(ZustandN, PfadA),
         bewerte_heuristisch(ZustandN, N, BewertungN)),
        GefundenePfade),
    sortiere_liste_ein(GefundenePfade, Pfade, NeuePfade),
    NeuePfade = [PfadN|RestPfade],
    heuristischesuche(PfadN, RestPfade).

```

findall stellt wie bisher die Liste *GefundenePfade* der neuen, heuristisch bewerteten Pfade zusammen. Die Elemente dieser Liste werden gemäß ihrer Bewertung in die vorhandene Prioritätswarteschlange *Pfade* einsortiert. Mit dem Kopfelement *PfadN* der ergänzten Prioritätswarteschlange *NeuePfade* geht die heuristische Suche weiter.

Bewertungsfunktion
nach Maßgabe der
Theorie

Für die heuristische Suche brauchen wir eine Bewertungsfunktion für Zustände des Hüpf-Schiebe-Puzzles. Die Theorie sagt, daß wir eine Bewertungsfunktion der Art $f(n) = g(n) + h(n)$ nehmen sollten, wobei $g(n)$ die Kosten von der Startposition bis zum Zustand n bedeuten und $h(n)$ den eigentlichen heuristischen Anteil beschreibt. Wählen wir $h(n)$ so, daß $h(n) \leq h^*(n)$ ist, wobei $h^*(n)$ die minimalen Kosten vom Zustand n bis zum Zielzustand sind, so garantiert die Theorie, daß die heuristische Suche einen optimalen Pfad findet! Wir sehen wieder mal, daß nichts praktischer als eine gute Theorie ist.

Für das Hüpf-Schiebe-Puzzle ist $g(n)$ nichts anderes als die Anzahl der Züge, die wir schon von der Startposition aus gemacht haben. $g(n)$ kann somit über die Länge des bisherigen Pfades berechnet werden.

Eine einfache heuristische Schätzfunktion $h(n)$ ist die Anzahl der nicht korrekt platzierten Spielmarken im Puzzle. Wir berechnen Sie mit den Prädikaten *bewerte_heuristisch* und *bewerte_positionen*:

der heuristische Anteil

```
bewerte_heuristisch(Zustand, N, Bewertung):-
    endzustand(Endzustand),
    bewerte_positionen(Zustand, Endzustand, Fehlplaziert),
    Bewertung is N + Fehlplaziert, !.

bewerte_positionen([], [], 0).
bewerte_positionen([K|R1], [K|R2], N):-
    bewerte_positionen(R1, R2, N).
bewerte_positionen([_K1|R1], [_K2|R2], N):-
    bewerte_positionen(R1, R2, N1),
    N is N1 + 1.
```

Berechnung des heuristischen Anteils

Die Berechnung erfolgt mit der Kopf-Rest-Methode. Sind die Köpfe der Positionen gleich, so ist der Beitrag zu $h(n)$ gleich 0, anderenfalls 1.

Einsatz der Kopf-Rest-Methode

14.10 Bewertung der Suchverfahren für das Hüpf-Schiebe-Puzzle

Das Hüpf-Schiebe-Puzzle ist vergleichsweise einfach. Es sind insgesamt nur 30 verschiedene Zustände möglich. Der Zustandsgraph kann daher zum Vergleich der drei Suchverfahren komplett untersucht werden. Die folgende Tabelle enthält Ergebnisse einer Untersuchung:

Tabelle 14-1
Untersuchungs-
ergebnisse zur
Bewertung der
Suchverfahren beim
Hüpf-Schiebe-
Puzzle

A	B	C	D	E	F	G	H	J	K	L
sswwl	18	21	0,16	8	184	0,83	72	0,44	41	0,28
sswlw	18	21	0,17	7	143	0,48	44	0,25	25	0,22
sslww	13	28	0,17	8	172	0,65	73	0,44	48	0,31
swwsl	11	38	0,16	5	58	0,22	17	0,12	10	0,12
swwslw	14	30	0,22	6	80	0,28	28	0,19	17	0,16
slsww	14	29	0,17	7	120	0,38	45	0,25	24	0,19
swwsl	14	30	0,17	6	74	0,22	26	0,19	20	0,16
swlsw	10	38	0,16	7	125	0,41	55	0,31	27	0,22
slwsw	11	25	0,11	6	104	0,31	27	0,12	20	0,16
swwls	11	38	0,17	5	56	0,22	18	0,12	14	0,12
swlws	16	21	0,17	4	38	0,19	13	0,12	9	0,06
slwws	17	22	0,16	5	48	0,16	19	0,19	14	0,19
wswsl	9	11	0,17	4	27	0,12	10	0,06	7	0,06
wsslw	16	26	0,16	5	37	0,16	19	0,12	13	0,12
lssww	15	30	0,16	8	172	0,66	73	0,44	37	0,31
wswsl	16	26	0,16	3	13	0,06	7	0,12	5	0,06
wslsw	6	9	0,10	4	28	0,16	12	0,12	9	0,12
lswsw	12	25	0,16	5	53	0,16	16	0,06	12	0,06
wswls	9	11	0,12	2	10	0,12	5	0,06	4	0,06
wslws	15	26	0,16	3	14	0,09	6	0,06	6	0,12
lswws	18	23	0,21	4	27	0,16	10	0,12	8	0,12
wwssl	2	4	0,11	1	4	0,06	3	0,06	3	0,00
wlssw	7	10	0,16	5	42	0,19	18	0,06	12	0,12
lwssw	8	11	0,17	6	69	0,22	26	0,12	17	0,19
wwsls	2	4	0,11	1	4	0,03	3	0,06	3	0,06
wlsws	4	563	0,77	2	10	0,12	5	0,06	5	0,12
lwsws	5	535	0,70	3	14	0,09	7	0,12	7	0,12
wwlss	0	2	0,06	0	2	0,06	2	0,06	2	0,06
wlwss	1	3	0,11	1	3	0,09	3	0,06	3	0,00
lwwss	2	4	0,05	1	4	0,06	3	0,06	3	0,06
	10,5	55,5	0,19	4,4	59,7	0,23	22,2	0,15	14,2	0,13

Spalte A führt alle 30 verschiedenen Startpositionen auf, in der drittletzten Zeile steht die Zielposition *wwlss*. Die Spalten B, C und D gehören zur Tiefensuche, E, F und G zur Breitensuche, E, H und J zur heuristischen Suche und E, K und L zu einer heuristischen Suche mit einer besseren Bewertungsfunktion.

Spalte B zeigt an, wie lange die von der Tiefensuche gefundenen Pfade von den gegebenen Startpositionen zur Zielposition sind. Als durchschnittliche Pfadlänge ergibt sich 10,5. Spalte C gibt an, wie viele Zustände bei der Pfadsuche untersucht wurden. Im Mittel sind es 55,5. In Spalte D sind die zugehörigen Suchzeiten in Sekunden aufgeführt. Wegen der geringen Auflösung der Systemuhr ist der einzelne Wert nicht aussagekräftig, als Mittelwert ergibt sich 0,19 Sekunden.

Analyse der
Tiefensuche

Spalte E zeigt wieder die ermittelten Pfadlängen an, und zwar für die Breitensuche und die beiden heuristischen Suchen. Diese drei Suchalgorithmen finden jeweils die kürzesten Pfade von gegebener Start- zur Zielposition. Bei der Breitensuche ist dies gesichert, weil der Suchhorizont sich wellenartig ausbreitet und damit die erste gefundene Lösung gleichzeitig die mit dem kürzesten Pfad ist.

Pfadlängen bei
Breitensuche und
heuristischer
Suche

Beim heuristischen Suchen ist dies nur dann gesichert, wenn die heuristische Bewertungsfunktion $h(n)$ die tatsächlichen minimalen Kosten $h^*(n)$ unterschätzt. Dies ist bei der vorgestellten Bewertungsfunktion nicht der Fall. Die Position $[w, l, w, s, s]$ erhält die Bewertung $h=2$, weil sie sich an zwei Stellen von der Zielposition $[w, w, l, s, s]$ unterscheidet. Man kommt aber schon mit einem Schiebezug von der Position $[w, l, w, s, s]$ zur Zielposition. Obwohl hier die Bewertungsfunktion überschätzt, findet in diesem Puzzle die heuristische Suche stets die kürzesten Pfade.

Die Breitensuche ist ein Spezialfall der heuristischen Suche mit den Bewertungsfunktionen $g(n) = n$ und $h(n) = 0$, also $f(n) = n$, wobei wegen $h(n) = 0 \leq h^*(n)$ die Breitensuche eine Unterschätzung macht und damit stets kürzeste Pfade findet.

Die Spalten F und G geben die jeweils untersuchten Zustände und benötigten Zeiten der Breitensuche wieder. Es wird deutlich, daß die Tiefensuche bei der Länge der gefundenen Lösungspfade deutlich schlechter abschneidet als alle anderen Suchalgorithmen. Die Breitensuche untersucht durchschnittlich mehr Zustände und benötigt dafür auch mehr Zeit.

Analyse der
Breitensuche

In den Spalten H und J sind Anzahl der untersuchten Zustände und Zeiten für die heuristische Suche mit der vorgestellten Bewertungsfunktion aufgeführt. Mit durchschnittlich 22,2 untersuchten Zuständen inspiziert die heuristische Suche deutlich weniger Zustände als Tiefen- und Breitensuche. Die relative Zeitersparnis ist nicht so groß, weil das Bewerten der Zustände auch Zeit in Anspruch nimmt.

Analyse der heuri-
stischen Suche

Die Spalten K und L zeigen die entsprechenden Werte für eine heuristische Suche mit einer besseren Bewertungsfunktion. Sie wird in Aufgabe 6 behandelt.

Aufgrund dieser Untersuchung könnte man zum Trugschluß kommen, daß Breitensuche und heuristische Suche der Tiefensuche grundsätzlich überlegen wären. Dies ist aber nicht unbedingt so, denn bei etwas umfangreicheren Zustandsräumen kann beiden Suchalgorithmen schnell der benötigte Speicher für die Warteschlangen ausgehen.

14.11 Das 8er-Puzzle

Das 8er-Puzzle besteht aus einem 3 x 3-Spielfeld mit 8 Plättchen, die mit den Zahlen von 1 bis 8 beschriftet sind. Ein Platz auf dem Spielfeld bleibt unbelegt. Durch Verschieben der Plättchen auf das freie Spielfeld soll der geordnete Zielzustand hergestellt werden. Beispiel:

	Startzustand	Zielzustand																		
Start- und Zielzustand beim 8er-Puzzle	<table border="1" style="border-collapse: collapse; text-align: center;"> <tr><td style="padding: 2px 5px;">8</td><td style="padding: 2px 5px;">1</td><td style="padding: 2px 5px;">3</td></tr> <tr><td style="padding: 2px 5px;">2</td><td style="padding: 2px 5px;"></td><td style="padding: 2px 5px;">4</td></tr> <tr><td style="padding: 2px 5px;">7</td><td style="padding: 2px 5px;">6</td><td style="padding: 2px 5px;">5</td></tr> </table>	8	1	3	2		4	7	6	5	<table border="1" style="border-collapse: collapse; text-align: center;"> <tr><td style="padding: 2px 5px;">1</td><td style="padding: 2px 5px;">2</td><td style="padding: 2px 5px;">3</td></tr> <tr><td style="padding: 2px 5px;">8</td><td style="padding: 2px 5px;"></td><td style="padding: 2px 5px;">4</td></tr> <tr><td style="padding: 2px 5px;">7</td><td style="padding: 2px 5px;">6</td><td style="padding: 2px 5px;">5</td></tr> </table>	1	2	3	8		4	7	6	5
8	1	3																		
2		4																		
7	6	5																		
1	2	3																		
8		4																		
7	6	5																		

Die Erkenntnisse über Suchverfahren lassen sich leicht vom Hüpf-Schiebe-Puzzle auf das kompliziertere Problem des 8er-Puzzles übertragen.

Als Datenstruktur benutzen wir die wohlbekannteren Listen. Die obigen Zustände werden durch die beiden folgenden Listen dargestellt.

Datenstruktur des 8er-Puzzles	Startzustand:	[8, 1, 3, 2, x, 4, 7, 6, 5]
	Zielzustand:	[1, 2, 3, 8, x, 4, 7, 6, 5]

Vier verschiedene Züge stehen zur Verfügung: das leere Feld nach o(ben), l(inks), u(nten) oder r(chts). Jeder Zug transformiert den aktuellen Zustand in einen neuen Zustand. Der neue Zustand läßt sich aber nur mühsam aus dem vorherigen Zustand berechnen. Daher benutzen wir eine Tabelle, in der alle Züge angewendet auf die neun verschiedenen Positionen des leeren Feldes vorkommen:

```

zug([x,A,B,C,D,E,F,G,H], [A,x,B,C,D,E,F,G,H], r).
zug([x,A,B,C,D,E,F,G,H], [C,A,B,x,D,E,F,G,H], u).

zug([A,x,B,C,D,E,F,G,H], [x,A,B,C,D,E,F,G,H], l).
zug([A,x,B,C,D,E,F,G,H], [A,D,B,C,x,E,F,G,H], u).
zug([A,x,B,C,D,E,F,G,H], [A,B,x,C,D,E,F,G,H], r).

zug([A,B,x,C,D,E,F,G,H], [A,x,B,C,D,E,F,G,H], l).
zug([A,B,x,C,D,E,F,G,H], [A,B,E,C,D,x,F,G,H], u).

zug([A,B,C,x,D,E,F,G,H], [x,B,C,A,D,E,F,G,H], o).
zug([A,B,C,x,D,E,F,G,H], [A,B,C,F,D,E,x,G,H], u).
zug([A,B,C,x,D,E,F,G,H], [A,B,C,D,x,E,F,G,H], r).

zug([A,B,C,D,x,E,F,G,H], [A,x,C,D,B,E,F,G,H], o).
zug([A,B,C,D,x,E,F,G,H], [A,B,C,x,D,E,F,G,H], l).
zug([A,B,C,D,x,E,F,G,H], [A,B,C,D,G,E,F,x,H], u).
zug([A,B,C,D,x,E,F,G,H], [A,B,C,D,E,x,F,G,H], r).

zug([A,B,C,D,E,x,F,G,H], [A,B,x,D,E,C,F,G,H], o).
zug([A,B,C,D,E,x,F,G,H], [A,B,C,D,x,E,F,G,H], l).
zug([A,B,C,D,E,x,F,G,H], [A,B,C,D,E,H,F,G,x], u).

zug([A,B,C,D,E,F,x,G,H], [A,B,C,x,E,F,D,G,H], o).
zug([A,B,C,D,E,F,x,G,H], [A,B,C,D,E,F,G,x,H], r).

zug([A,B,C,D,E,F,G,x,H], [A,B,C,D,x,F,G,E,H], o).
zug([A,B,C,D,E,F,G,x,H], [A,B,C,D,E,F,x,G,H], l).
zug([A,B,C,D,E,F,G,x,H], [A,B,C,D,E,F,G,H,x], r).

zug([A,B,C,D,E,F,G,H,x], [A,B,C,D,E,x,G,H,F], o).
zug([A,B,C,D,E,F,G,H,x], [A,B,C,D,E,F,G,x,H], l).

```

Tabelle 14-2
Zusammenstellung
der Züge im
8er-Puzzle

14.12 Beschränkte Tiefensuche für das 8er-Puzzle

Reine Tiefensuche hat für das 8er-Puzzle keinen Sinn, da der Zustandsraum-Graph sehr groß ist. Er besteht aus $9!$ Knoten. Wir verwenden daher die beschränkte Tiefensuche, die nur bis zu einer vorzugebenden Tiefe in den Suchbaum eindringt. Je größer man die Tiefenschranke wählt, desto mehr nähert sich die beschränkte der unbeschränkten Tiefensuche.

Implementierung
der Tiefensuche für
das 8er-Puzzle

```

tiefensuche:-
    write('maximale Tiefe: '), read(Tiefe),
    write(Tiefe), nl,
    write('Problemnummer: '), read(Nummer),
    write(Nummer), nl,
    problem(Nummer, Startzustand),
    tiefensuche(Tiefe, [Startzustand], []).

tiefensuche(_, Pfad, Zuege):-
    Pfad = [Zustand|_],
    endzustand(Zustand),
    write('Lösung: '), write(Zuege), nl, !.
tiefensuche(Tiefe, Pfad, Zuege):-
    Tiefe >= 1,
    Pfad = [ZustandA|_],
    zug(ZustandA, ZustandN, Zug),
    not member(ZustandN, Pfad),
    tiefensuche(Tiefe - 1, [ZustandN|Pfad], [Zug|Zuege]).

```

Das Prädikat *tiefensuche/0* erfragt die maximale Tiefe und die Nummer eines Problems, wobei die Nummer der Länge eines minimalen Lösung entspricht. Beispiele:

Definition einiger
Probleme

```

problem(4, [1,4,2,8,x,3,7,6,5]).
problem(5, [1,4,2,x,8,3,7,6,5]).

```

Anschließend wird die eigentliche Suche durch *tiefensuche(+Tiefe, +Pfad, +Zuege)* gestartet. Der aktuelle Knoten wird durch den Kopf der Pfadliste übergeben. Im Argument *Zuege* wird eine Liste mit den bisherigen Zügen verwaltet. Beispielsweise beschreibt *Zuege = [l, o, r]* die Zugfolge r, o, l. Als Lösung wird die Zugfolge vom Start zum Ziel ausgegeben.

14.13 Breitensuche für das 8er-Puzzle

Die Breitensuche muß neben den Pfaden auch immer die zugehörige Operatorfolge verwalten. Wir bilden daher zusammengesetzte Terme *pfad(Pfad, Zuege)*, welche in der ersten Komponente den Pfad und in der zweiten die zugehörige Züge speichern. Die Warteschlange selbst besteht dann aus einer Liste solcher Verbunde.

```
breitensuche:-
    write('Problemnummer: '), read(Nummer),
    write(Nummer), nl,
    problem(Nummer, Startzustand),
    breitensuche(pfad([Startzustand],[]), []).

breitensuche(Pfad, _):-
    Pfad = pfad([Zustand|_], Zuege),
    endzustand(Zustand),
    write('Lösung: '), write(Zuege), nl, !.
breitensuche(Pfad, Pfade):-
    Pfad = pfad([ZustandA|PfadA], Zuege),
    findall(pfad([ZustandN, ZustandA|PfadA],[Zug|Zuege]),
        (zug(ZustandA, ZustandN, Zug),
         not member(ZustandN, PfadA)),
        GefundenePfade),
    append(Pfade, GefundenePfade, NeuePfade),
    NeuePfade = [PfadN|RestPfade],
    breitensuche(PfadN, RestPfade).
```

Implementierung
der Breitensuche
für das 8er-Puzzle

14.14 Heuristische Suche für das 8er-Puzzle

Die heuristische Suche läuft ähnlich ab, wie die Breitensuche, nur müssen wir zusätzlich jede Puzzlestellung bewerten und die Stellungen in der Rangfolge der Bewertungen untersuchen. Zur Verwaltung der zusätzlichen Bewertung ergänzen wir die Datenstruktur der Breitensuche um eine Bewertungskomponente. Sie hat die Struktur *pfad(Pfad, Zuege, Bewertung)*, woraus sich dann folgende Implementierung ergibt

Implementierung
der heuristischen
Suche für das
8er-Puzzle

```
heuristischesuche:-
    write('Problemnummer: '), read(Nummer),
    write(Nummer), nl,
    problem(Nummer, Startzustand),
    heuristischesuche(pfad([Startzustand], [], 0), []).

heuristischesuche(Pfad, _):-
    Pfad = pfad([Zustand|_], Zuege, _),
    endzustand(Zustand),
    write('Loesung: '), write(Zuege), nl, !.

heuristischesuche(Pfad, Pfade):-
    Pfad = pfad([ZustandA|PfadA], Zuege, _),
    length(Zuege, N),
    findall(pfad([ZustandN, ZustandA|PfadA],
                [Zug|Zuege], Bewertung),
            (zug(ZustandA, ZustandN, Zug),
             not member(ZustandN, Pfad),
             bewerte_heuristisch(ZustandN, N, Bewertung)),
            GefundenePfade),
    sortiere_liste_ein(GefundenePfade, Pfade, NeuePfade),
    NeuePfade = [PfadN|RestPfade],
    heuristischesuche(PfadN, RestPfade).
```

Wie beim Hüpf-Schiebe-Puzzle ist die Anzahl der nicht korrekt platzierten Plättchen im Puzzle eine einfache Schätzfunktion $h(n)$. Wir können den entsprechenden Programmcode von *bewerte_heuristisch* direkt übernehmen.

Das Prädikat *sortiere_element_ein* muß leicht angepaßt werden, da die *pfad*-Strukturen nunmehr 3-stellig sind:

```
sortiere_element_ein(K1, [K2|R], [K1, K2|R]):-
    K1 = pfad(_, _, Bewertung1),
    K2 = pfad(_, _, Bewertung2),
    Bewertung1 < Bewertung2.
sortiere_element_ein(K1, [K2|R2], [K2|R3]):-
    sortiere_element_ein(K1, R2, R3).
sortiere_element_ein(K1, [], [K1]).
```

14.15 Bewertung der Suchverfahren für das 8er-Puzzle

An dieser Stelle möchte ich einige Erfahrungen unter TV-SWI-Prolog mit den konstruierten Beispielen mitteilen, die nicht auf die Schnelle nachvollzogen werden können.

Bei der Tiefensuche hat man immer das Problem, eine geeignete Tiefenschranke anzugeben. Hat man keine Vorstellung, wieviel Züge bei gegebener Stellung erforderlich sind, um die Anfangsstellung zu erreichen, so muß man notgedrungen einen hohen Wert vorgeben, was zu sehr langen Suchzeiten führen kann. Zudem werden dann auch nicht optimale Lösungen gefunden. Wählt man die Suchschranke zu klein, so wird gegebenenfalls keine Lösung gefunden. Weiß man ungefähr, wie viele Züge erforderlich sind, so liefert die Tiefensuche recht schnell ein Ergebnis. Unter TV-SWI-Prolog liegen die Suchzeiten für Lösungswege der Länge 15 im Bereich bis eine Minute.

Bewertung der
Tiefensuche

Da die Breitensuche alle Pfade speichern muß, gibt es bei Lösungswegen ab der Länge 8 Speicherprobleme. Solche Probleme sind mit Breitensuche nicht mehr lösbar.

Bewertung der
Breitensuche

Die heuristische Suche löst Probleme bis zu Lösungswegen der Länge 15. Für schwierigere Probleme braucht man eine bessere Heuristik!

Bewertung der
heuristischen
Suche

Speichert man für die Breitensuche und heuristische Suche nur die Züge, aber nicht die Pfade, so kommt man mit der Suche erheblich weiter. Allerdings muß dann der Zyklustest jeweils aus den Zügen die Stellungen ermitteln.

14.16 Aufgaben

1. Zunächst ohne, dann mit Rechner: Welche Lösungen liefern im Graphen-Beispiel die Anfragen:
 - a) ?- `tiefensuche(a,k,L)`.
 - b) ?- `breitensuche(a,k,L)`.

- 2a) In welcher Reihenfolge werden bei der Anfrage ?- `breitensuche(a, h, L)` die Knoten besucht?
- b) Kontrollieren Sie ihr Ergebnis, indem Sie das Prädikat `breitensuche` um die Ausgabe des aktuell besuchten Knotens ergänzen.
- c) Vergleichen Sie mit der Tiefensuche.
- d) Vertauschen Sie im `append`-Prädikat der Breitensuche die beiden α -sten Argumente. In welcher Reihenfolge werden jetzt die Knoten besucht?

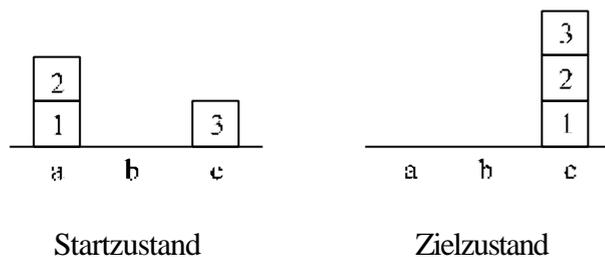
- 3a) Führen Sie die Breitensuche für die Anfrage ?- `breitensuche(a,h,L)` mit Papier und Bleistift durch.
- b) Kontrollieren Sie ihr Ergebnis durch Einfügen von Ausgabeanweisungen in das Prädikat `breitensuche`.

- 4a) Geben Sie für die heuristische Suche im Graphen die Funktionen $g(n)$ und $h(n)$ an.
- b) Inwiefern ist der Begriff heuristische Suche hier gar nicht angebracht?
- c) Implementieren Sie das Prädikat `sortiere_element_ein` von Seite 166.

5. Vergleichen Sie die Lösungen für die Anfragen:
 - ?- `tiefensuche(a,k,L)`.
 - ?- `breitensuche(a,k,L)`.
 - ?- `heuristischesuche(a,k,L)`.

6. Für das Hüpf-Schiebe-Puzzle ist eine bessere Heuristik möglich. Man zählt wie weit das leere Feld von der Zielposition 3, wie weit die linke schwarze Spielmarke von der Zielposition 4 und wie weit die rechte schwarze Spielmarke von der Zielposition 5 entfernt ist. Mit dieser Heuristik wurden die Ergebnisse in den Spalten K und L erhalten. Implementieren Sie hierfür das Prädikat `bewerte_heuristisch`.

7. Schwierig:
- Implementieren Sie für das 8er-Puzzle die Heuristik-Funktion $h(n) = P(n)$, wobei $P(n)$ die Summe aller Distanzen ist, die ein Stein von der Zielposition entfernt ist.
 - Experimentieren Sie mit dieser Heuristik-Funktion. Welche Probleme lassen sich damit lösen?
 - $P(n)$ berücksichtigt noch nicht, daß sich die Steine auf dem Weg nach Hause auch im Weg stehen. Wir verbessern daher den Ansatz zu $h(n) = P(n) + 3*S(n)$, wobei $S(n)$ sich wie folgt ergibt: Wir betrachten alle nicht zentralen Steine und addieren 2 für jeden Stein, der nicht direkt von seinem direkten Nachfolger gefolgt wird, ansonsten 0. Ein Stein in der Mitte zählt 1.
8. Auf drei Plätzen a, b und c liegen mehrere Blöcke. Es ist erlaubt, den obersten Block eines Stapels auf einen anderen Stapel oder einen freien Platz zu legen.



- Modellieren Sie die Zustände der Blockwelt durch drei Listen und definieren Sie die sechs möglichen Züge auf diesen Zuständen.
- Wenden Sie die Tiefen- und Breitensuche auf die Blockwelt an.
- Konzipieren und implementieren Sie eine Bewertungsfunktion für die heuristische Suche.

15 Terme

15.1 Klassifikation von Termen

Die Datenobjekte von Prolog nennt man *Terme*. Ein Term ist entweder eine Konstante, eine Variable oder ein zusammengesetzter Term. Eine *Konstante* ist entweder eine Zahl oder ein Atom. *Zusammengesetzte Terme* bestehen aus einem Funktor und dessen Argumenten, sie werden auch *Strukturen* genannt. *Listen* sind spezielle zusammengesetzte Terme mit dem Funktor „*,*“ und zwei Argumenten.

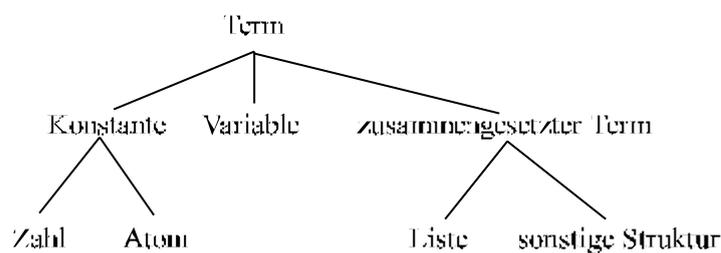


Abb. 15-1
Hierarchie der
Prolog-Terme

Zur Termanalyse stellt Prolog folgende Prädikate zur Verfügung:

`atom(X)` ist erfüllt, wenn X ein Atom ist.
`integer(X)` ist erfüllt, wenn X eine ganze Zahl ist.
`atomic(X)` ist erfüllt, wenn X eine Konstante ist.
`var(X)` ist erfüllt, wenn X eine freie Variable ist.
`nonvar(X)` ist erfüllt, wenn `var(X)` scheitert.

Systemprädikate
zur Termanalyse

Wie man sieht, fehlen entsprechende Prädikate für zusammengesetzte Terme. Deshalb holen wir das selbst nach:

```

liste([]).
liste([_|Y]):- liste(Y).
compoundterm(X):- not atomic(X), nonvar(X).
  
```

Weitere Prädikate
zur Termanalyse

Da in Prolog fast alle Daten zusammengesetzte Terme sind, werden wir uns jetzt mit diesen Termen näher beschäftigen.

15.2 Zusammengesetzte Terme

Listen, arithmetische Terme, Fakten und Regeln werden innerhalb von Prolog in einheitlicher Weise dargestellt. Nach außen hin ist davon nichts zu erkennen, weil Prolog für die Eingabe und Ausgabe spezielle Notationen verwendet: Listen gibt es mit Listenklammern aus, arithmetische Terme mit Infixoperatoren und sonstige Terme mit Präfixoperatoren. Diese Schreibweisen sind für den normalen Gebrauch von Prolog von Vorteil. Doch wenn wir hinter die Kulissen von Prolog schauen, erkennen wir ein einheitliches Repräsentationsprinzip. Wenn wir dieses Prinzip verstanden haben, haben wir auch viel von Prolog verstanden und können diese Erkenntnis nutzbringend anwenden.

Die folgenden Abbildungen zeigen, wie sich unterschiedlich aussehende zusammengesetzte Terme einheitlich als Bäume darstellen lassen:

einheitliches
Repräsentations-
prinzip

zusammengesetzter Term
person(harald, schmidt, datum(17, 10, 56))

arithmetischer Term
 $(2 + X) * 3 - 7$

Abb. 15-2
Baumdarstellung
eines zusammen-
gesetzten Terms

Abb. 15-3
Baumdarstellung
eines arithmeti-
schen Terms

Liste: [2, 3, [4, 5]].

Abb. 15-4
Baumdarstellung
einer Liste

Fakt:
`laenge([], 0).`

Regel:
`laenge([X|L],N):- laenge(L,N1),N is N1+1.`

Abb. 15-5
 Baumdarstellung
 eines Fakts

Abb. 15-6
 Baumdarstellung
 einer Regel

Prolog sieht keine Möglichkeit vor, Terme als Bäume graphisch auszugeben. ProVisor kann dies! Die Termvisualisierung von ProVisor wurde zur Erzeugung der Abbildungen 15-2 bis 15-6 benutzt. Mit dem *display*-Prädikat können Sie sich immerhin Terme in der Präfix-Schreibweise ausgeben lassen. *Display* setzt man normalerweise ein, um sich über den Aufbau eines Terms Klarheit zu verschaffen.

Termvisualisierung
 mit ProVisor

Das *display*-Prädikat liefert für unsere fünf Beispiele:

```
?- display((2 + X) * 3 - 7).
   -( * ( + ( 2, X_1 ), 3 ), 7 )

?- display(person(harald,schmidt,datum(17,10,56))).
   person(harald,schmidt,datum(17,10,56))

?- display([2, 3, [4, 5]]).
   . ( 2, . ( 3, . ( . ( 4, . ( 5, [] ) ), [] ) ) )

?- display(laenge([], 0)).
   laenge([],0)

?- display((laenge([X|L],N):- laenge(L, N1), N is N1 + 1)).
   :- ( laenge(. ( X,L ), N) , , ( laenge(L,N1), is(N,+(N1,1))) )
```

Beispiele für
 den Einsatz von
display

15.3 Term-Vergleichsoperatoren

Standardordnung für Terme
 TV-SWI-Prolog und vergleichbar gute Prolog-Interpreter bieten Vergleichsoperatoren für Terme an, welche auf folgender Standardordnung für Terme basieren:

1. Variablen < Atome < Zahlen < Terme
2. alte Variable < neue Variable
3. Atome werden nach der lexikographischen Ordnung verglichen
4. Zahlen werden nach ihrem Wert verglichen
5. Terme werden zuerst nach ihrem Funktor, dann nach ihrer Stelligkeit und zuletzt rekursiv nach ihren Argumenten verglichen, wobei der Vergleich beim ersten Argument losgeht.

Zum Term-Vergleich stellt man den üblichen Vergleichsoperatoren einen Klammeraffen @ voran. Entbehrlich ist dies bei == und \==, weil diese Operatoren nicht auf obiger Standardordnung basieren.

Tabelle 15-1
 Vergleichsoperatoren für Terme

Vergleich	Schreibweise
kleiner	Term1 @< Term2
kleiner gleich	Term1 @=< Term2
größer	Term1 @> Term2
größer gleich	Term1 @>= Term2
gleich	Term1 == Term2
ungleich	Term1 \== Term2

Beispiele für Vergleiche

bewertete Pfade: pfad(7, [a,b,c]) @< pfad(9, [b,a,c])
 Kalenderdaten: datum(1995,31,12)@>= datum(1993,17,6)
 Uhrzeiten: 9:30 @<= 10:40

Das Beispiel der Kalenderdaten macht deutlich, daß man nur dann die gewünschte Ordnung erhält, wenn man die Argumente von Termen in der richtigen Reihenfolge anordnet. Bei Kalenderdaten ist dies die Reihenfolge *Jahr, Monat, Tag*.

sort Das System-Prädikat *sort(+Liste, -SortierteListe)* sortiert Listen beliebiger Terme nach obiger Standard-Ordnung.

15.4 Strukturoperatoren

Prolog bietet zur Strukturuntersuchung die beiden Prädikate

```
functor(?Term, ?Name, ?Stelligkeit)    und    functor
arg(+ArgNum, +Term, -Arg)              und    arg
```

sowie den zweistelligen Infix-Operator „=..“ an. Er wird mit *univ* bezeichnet. Univ steht als Infix-Operator zwischen einem Term und einer Liste.

Mit dem *functor*-Prädikat kann man wahlweise zu gegebenem Term den Funktor und die Stelligkeit ermitteln oder durch Vorgabe eines Funktors und der Stelligkeit einen entsprechenden Term aufbauen:

```
?- functor(liebt(susi, peter), F, S).    Antwort: F=liebt, S=2
?- functor(3+4*5, F, S).                 Antwort: F=+, S=2
?- functor(T, liebt, 2).                 Antwort: T=liebt(_1, _2)
```

Beispiele für den Einsatz von functor

Das *functor*-Prädikat greift auf den Funktor zu, das *arg*-Prädikat auf die Argumente. *arg(+ArgNum, +Term, -Arg)* zeigt an, daß die beiden ersten Argumente instanziiert sein müssen und man das Ergebnis, das Argument mit der Nummer *ArgNum*, im dritten Argument erhält.

```
?- arg(2, liebt(susi, peter), X).       Antwort: X = peter
?- arg(2, 3+4*5, X).                   Antwort: X = 4*5
```

Beispiele für den Einsatz von arg

Der allgemeine Aufbau eines Terms ist

```
funktor(argument_1, argument_2, ... argument_N)
```

wobei die Argumente selbst wieder Terme sein können. Der Term kann mit dem *univ*-Operator in eine Liste umgebaut werden, wobei der Funktor zum Kopf der Liste wird und die Argumente den Rest der Liste bilden:

```
[funktor, argument_1, argument_2, ... argument_N]
```

Graphisch stellt sich die Situation wie folgt dar:

Abb. 15-7
Umwandlung einer
Struktur

Abb. 15-8
mit dem *univ*-
Operator
in eine Liste

Transformation
zwischen Liste
und Baum

Der *univ*-Operator macht aus einem allgemeinen Baum einen entarteten B-närbaum und umgekehrt. Er erlaubt es, einen Term in eine Liste und eine Liste in einen Term umzuwandeln. Konvertieren wir einen Term in eine Liste, so können wir mit unseren guten Kenntnissen über Listen auch Strukturuntersuchungen vornehmen. Doch zunächst nochmal unsere Beispiele:

Beispiele für den
Einsatz des *univ*-
Operators

```
?- (2 + X) * 3 - 7 =.. Y.  
Lösung: Y = [-(2 + X) * 3, 7]
```

```
?- person(harald, schmidt, datum(17, 10, 56)) =.. X.  
Lösung: X = [person, harald, schmidt, datum(17, 10, 56)]
```

```
?- [2, 3, [4, 5]] =.. X.  
Lösung: X = [., 2, [3, [4, 5]]]
```

```
?- laenge([], 0) =.. X.  
Lösung: X = [laenge, [], 0]
```

```
?- (laenge([X|L], N):- laenge(L, N1), N is N1 + 1) =.. Y.  
Lösung: Y = [:-, laenge([X|L], N), laenge(L1, N1), N is N1+1]
```

15.5 Beispiele für Termuntersuchungen

An zwei Beispielen demonstrieren wir, wie man mit dem *univ*-Operator arbeiten kann. Die grundsätzliche Vorgehensweise besteht darin, einen zusammengesetzten Term in eine Liste zu verwandeln und dann mit den bekannten Methoden der Listenverarbeitung weiter zu arbeiten.

15.5.1 Zählen von Variablen in Termen

Es soll ein Prädikat *countvar(+Term, -Anzahl)* entwickelt werden, das zählt, wie viele Variablen in einem Term vorkommen. countvar

Anfrage: `?-countvar(f(a,Y,g(X,Y,Z),[1,A,2,B,C],1+P*Q-r),N).`

Antwort: `N = 9`

1. Sonderfälle erledigen.

```
countvar(Term, 0):-                               /* Term ist eine Konstante */
    atomic(Term).                                  Konstanten
                                                    und Variablen
countvar(Term, 1):-                               /* Term ist eine Variable */
    var(Term).
```

2. Im allgemeinen Fall ist der Term zusammengesetzt und wird daher in eine Liste zerlegt. Dann läßt man mittels Listenmethoden die Anzahl der Variablen in der Liste der Argumente berechnen.

```
countvar(Term, Anzahl):-                          Term in Liste
    Term =.. [Kopf|Liste],                        transformieren
    countvarliste(Liste, Anzahl). /* Im Kopf steht der Funktor */
                                     /* der keine Variable sein kann. */
```

3. Die Liste wird nach der Kopf-Rest-Methode behandelt.

```
countvarliste([K|R], N):-                          /* Kopf ist eine Konstante */
    atomic(K),                                     Kopf-Rest-Methode
    countvarliste(R, N).                           auf Liste anwenden
countvarliste([K|R], N):-                          /* Kopf ist eine Variable */
    var(K),
    countvarliste(R, N1),
    N is N1 + 1.
countvarliste([K|R], N):-                          /* Kopf ist wiederum ein Term */
    countvar(K, N1),
    countvarliste(R, N2),
    N is N1 + N2.
countvarliste([], 0).                               /* Ende der Rekursion */
```

15.5.2 Partielle Auswertung arithmetischer Ausdrücke

Der *is*-Operator kann nur angewendet werden, wenn alle Variablen eines arithmetischen Ausdrucks instanziiert sind. Wünschenswert wäre aber ein Prädikat *auswerten(+Term, -AusTerm)*, das Terme soweit wie möglich ausrechnet und nicht mit einer Fehlermeldung abbricht.

Anfrage: ?- `auswerten(3*5 + a - f(2+3, b, 3*4+2), T)`.

Antwort: `T = 15 + a - f(5, b, 14)`

1. Sonderfälle erledigen.

Konstanten und Variablen

```
auswerten(Term, Term):-                               /* Term ist eine Konstante */
    atomic(Term).
auswerten(Term, Term):-                               /* Term ist eine Variable */
    var(Term).
```

2. Im allgemeinen Fall ist der Term zusammengesetzt. Zur Auswertung muß der Termbaum zuerst bis zu den Blättern durchlaufen werden, weil man einen Term stets von den Blättern zur Wurzel hin auswertet. Werden benachbarte Blätter wieder zu einem Term zusammengesetzt, so testet man, ob der so gebildete Term sich vereinfachen läßt.

Wurzel-Knoten-Methode

```
auswerten(Term1, Term2):-
    Term1 =.. [Funktork|Argumente],                /* Term in Liste zerlegen */
    auswerten_liste(Argumente, Werte),            /* Argumente auswerten */
    Term3 =.. [Funktork|Werte],                    /* Term zusammensetzen */
    vereinfachen(Term3, Term2).                    /* Term vereinfachen */
```

3. Das Auswerten einer Liste nehmen wir durch Auswertung aller Argumente nach der Kopf-Rest-Methode für Listen vor.

Kopf-Rest-Methode

```
auswerten_liste([], []).
auswerten_liste([Argument|Argumente], [Wert|Werte]):-
    auswerten(Argument, Wert),
    auswerten_liste(Argumente, Werte).
```

4. Nach dem Zusammensetzen ausgewerteter Argumente kann vereinfacht werden, wenn der Funktor +, -, *, mod oder Minus ist.

```
vereinfachen(T1 + T2, Wert) :- Wert is T1 + T2.
vereinfachen(T1 - T2, Wert) :- Wert is T1 - T2.
vereinfachen(T1 * T2, Wert) :- Wert is T1 * T2.
vereinfachen(T1 / T2, Wert) :- Wert is T1 / T2.
vereinfachen(T1 mod T2, Wert):- Wert is T1 mod T2.
```

```
vereinfachen(-T, Wert)      :- Wert is -T.  
vereinfachen(T, T).        /* keine Vereinfachung möglich */
```

15.6 Wurzel-Knoten-Methode

Wer mit dem Listen-Operator „|“ umgehen kann, beherrscht die Basis-Programmiertechnik von Prolog. Viele Probleme lassen sich mit Hilfe von Listen lösen. Es ist deshalb nicht unbedingt nötig, im Unterricht den univ-

dynamische
Baumstrukturen

Operator zu thematisieren. Die grundsätzliche Beschränkung besteht dann allerdings darin, daß dynamische Baumstrukturen nicht genutzt werden können.

Wenn man fortgeschrittene Programmiertechniken in Prolog einsetzen will, kommt man um die Verwendung des univ-Operators nicht herum. Er stellt gewissermaßen die Verallgemeinerung des Listen-Operators dar. Mit dem Listen-Operator bearbeitet man dynamische lineare Datenstrukturen, mit dem univ-Operator dynamische baumartige Datenstrukturen. Wegen der Bedeutung von Bäumen in der Informatik sollte man sich mit dem univ-Operator vertraut machen.

In Pascal muß man sich beim Einsatz von Baumstrukturen auf den Verzweigungsgrad des Baumes festlegen. Binärbäume spielen daher eine besondere Rolle. Der univ-Operator ist so allgemein verwendbar, daß man Bäume mit beliebigen Verzweigungsgraden algorithmisch in den Griff bekommt.

Zu einer Wurzel können n verschiedene Nachfolger-Knoten gehören. Aus

Wurzel-Knoten-
Methode

den bei Binärbäumen üblichen LWR-, WLR- und LRW-Verfahren, wird daher bei allgemeinen Bäumen die Wurzel-Knoten-Methode in den beiden Varianten

WK: zuerst die Wurzel, dann alle Nachfolger-Knoten

KW: zuerst alle Nachfolger-Knoten, dann die Wurzel

Beim zweiten Beispiel kommt die Wurzel-Knoten-Methode in der Ausprägung KW zum Tragen. Der Rechenbaum wird in die Wurzel und die Liste aller Knoten zerlegt. Die Knotenliste wird dann mit der Kopf-Rest-Methode für Listen bearbeitet. Anschließend wird ein neuer Rechenbaum mit vereinfachten Knoten und alter Wurzel zusammengebaut, welcher anschließend vereinfacht wird.

Beim ersten Beispiel kommt die WK-Methode zum Einsatz. Da die Wurzel keine Variable sein kann, müssen nur die Variablen in der Knotenliste gezählt werden.

15.7 Aufgaben

1. Wie antwortet Prolog?
 - a) `functor(summe(2, 3, 5), X, Y).`
 - b) `functor(schueler(paul, schmidt, datum(17, 5, 1977)), Name, Stellen).`
 - c) `arg(3, summe(2,3,5), Arg).`
 - d) `arg(2, [1,2,3,4], Liste).`
 - e) `hugo([a, b], [b, c]) =.. X.`
 - f) `a(b(c(1,2),3),4) =.. X.`
 - g) `[a, b] =.. X.`
 - h) `a + b*c =.. X.`
 - i) `X =.. [liebt, herbert, luise].`
 - j) `X =.. [+ , /(12, a), b].`
 - k) `X =.. [., 4, [b, c]].`

2. Realisieren Sie mit dem *functor*-Prädikat ein alternatives *compound-term*-Prädikat (siehe Seite 183).

3. Analysieren Sie
 berechne:-


```

write('X= '), read(X), write(X), nl,
write('Operator: '), read(Operator), write(Operator),
nl, write('Y= '), read(Y), write(Y), nl,
functor(Term, Operator, 2),
arg(1, Term, X),
arg(2, Term, Y),
Ergebnis is Term,
write('Ergebnis= '), write(Ergebnis), nl.
      
```

4. Definieren Sie ein Prädikat *member(+Element, +Term)* derart, daß es wahr ist, wenn *Element* in *Term* vorkommt.

5. Definieren Sie ein Prädikat *grund(+Term)* derart, daß es wahr ist, wenn *Term* keine uninstantierten Variablen enthält.

6. Es soll ein Prädikat *sammeln(+Term, -Liste)* entwickelt werden, das die Konstanten eines Terms in einer Liste sammelt. Orientieren Sie sich am *countvar*-Prädikat.

?- sammeln(f(a, Y, g(X,Y,Z)), [1,A,2,B,C], 1+P*Q-r), L).

Antwort: L = [a, 1, 2, [], 1, r]

7. Sie sollen ein *drucke*-Prädikat entwickeln, das die Struktur eines Terms durch Einrückungen darstellt.

Beispiele:

```
?- drucke(person(harald, schmidt, datum(17, 10, 56))).
person(
  harald
  schmidt
  datum(
    17
    10
    56
  )
)
```

```
?- drucke(4 * x + 6).
+(
  *(
    4
    x
  )
  6
)
```

Die Idee ist einfach. *Drucke* wird mit *drucke(Term, 0)* aufgerufen. Der Term wird in den Funktor und die Argumente zerlegt. Den Funktor geben Sie mit *write* aus und jedes Argument mit *drucke*. Das zweite Argument von *drucke* stellt die Einrücktiefe dar. Einrückungen machen Sie mit dem *tab*-Prädikat. Der Aufruf *tab(N)* gibt N Leerzeichen aus.

16 Grammatiken und Formale Sprachen

Der Wunsch nach maschineller Verarbeitung natürlicher Sprache war eine Triebfeder, aus der heraus Prolog entstand. Es verwundert daher nicht, wenn heute zur Bearbeitung linguistischer Probleme häufig Prolog benutzt wird.

maschinelle Verarbeitung natürlicher Sprache

Methoden der Sprachverarbeitung können natürlich auch auf künstliche Sprachen angewendet werden. Parser, Compiler und Interpreter für Programmiersprachen prüfen, ob Programme syntaktisch korrekt sind, übersetzen von einer künstlichen Sprache in eine andere beziehungsweise führen die Programme aus.

Sprachverarbeitung beginnt mit der Analyse von Sätzen. Ein Satz setzt sich aus einzelnen Wörtern zusammen. Wörter sind aber nicht schon die Bauteile des Satzes; sie stellen nur das Baumaterial dar, das in bestimmter Weise geformt und zusammengefügt die Bauteile des Satzes ergibt. So wie zum Beispiel aus einzelnen unzusammenhängenden Materialstücken noch keine funktionierende Maschine entsteht, so bildet eine bloße Ansammlung von Wörtern noch keinen Satz. Zum Beispiel:

Wörter und Sätze

Meine am ihren Frau bringt Jungen zu den Samstag Eltern.

Wörter, die keinen Satz bilden

Erst wenn zusammengehörige Wörter auch nebeneinander stehen, ergibt sich ein sinnvoller Satz:

Meine Frau bringt den Jungen am Samstag zu ihren Eltern.

Wörter, die einen Satz bilden

Bauteile des Satzes sind also bereits vorgefertigte Teilstücke, die in sich schon aus kleineren, fest miteinander verbundenen Einheiten zusammengesetzt sind.

Ein Satz kann nicht aus einer beliebigen Folge von Worten bestehen. Die Grammatik einer Sprache legt fest, wie ein Satz aufgebaut sein kann. Die Formulierung von Sätzen muß sich nach den Regeln der Grammatik richten. Umgekehrt richtet sich die syntaktische Analyse natürlich nach der Grammatik der Sprache. Es können nur Sätze weiterverarbeitet werden, welche grammatisch korrekt aufgebaut sind. Im Rahmen der syntaktischen Analyse muß also festgestellt werden, ob ein Satz grammatisch korrekt ist.

Grammatik einer Sprache

syntaktische Analyse

16.1 Grammatiken

16.1.1 Beispiel 1: Grammatik einfacher deutscher Sätze

Die folgende Grammatik beschreibt einfache deutsche Sätze:

Regeln einer Grammatik	(1)	Satz	→	Subjekt Prädikat Objekt
	(2)	Subjekt	→	Eigename Substantivgruppe
	(3)	Substantivgruppe	→	Artikel Substantiv
	(4)	Prädikat	→	Verb
	(5)	Objekt	→	Akkusativergänzung
	(6)	Akkusativergänzung	→	Substantivgruppe
	(7)	Verb	→	kauft liebt liest
	(8)	Artikel	→	die das ein
	(9)	Substantiv	→	Buch Mädchen Kartoffeln
	(10)	Eigename	→	Peter

Die erste Grammatikregel sagt beispielsweise aus, daß ein Satz aus einem Subjekt, Prädikat und Objekt besteht. Nach der zweiten Regel ist ein Subjekt ein Eigename oder eine Substantivgruppe, welche nach der dritten Regel aus einem Artikel und einem Substantiv besteht.

Sprachkonstrukte Die Begriffe auf den linken Seiten der Grammatikregeln kennzeichnen Sprachkonstrukte, aus denen ein Satz aufgebaut wird. Ein konkreter Satz besteht aus Wörtern. Diese treten nur auf den rechten Seiten von Grammatikregeln auf. Unser Wortschatz ist sehr eingeschränkt. Er besteht aus drei Verben, Artikeln und Substantiven, sowie dem Eigennamen Peter.

Startsymbol *Peter liebt das Mädchen* ist ein Satz unserer Grammatik. Um dies nachzuweisen, beginnen wir mit dem *Startsymbol* Satz und leiten aus ihm unter Anwendung der vorhandenen Grammatikregeln den Satz her:

Ableitung eines Satzes	Satz	→	(nach Regel 1)
	Subjekt Prädikat Objekt	→	(nach Regel 2)
	Eigename Prädikat Objekt	→	(nach Regel 10)
	Peter Prädikat Objekt	→	(nach Regel 4)
	Peter Verb Objekt	→	(nach Regel 7)
	Peter liebt Objekt	→	(nach Regel 5)
	Peter liebt Akkusativergänzung	→	(nach Regel 6)
	Peter liebt Substantivgruppe	→	(nach Regel 3)
	Peter liebt Artikel Substantiv	→	(nach Regel 8)
	Peter liebt das Substantiv	→	(nach Regel 9)

Peter liebt das Mädchen

16.1.2 Beispiel 2: Arithmetische Ausdrücke

Durch die folgende Grammatik kann die Struktur arithmetischer Ausdrücke, bei denen die vier Grundrechenarten und Klammern vorkommen dürfen, beschrieben werden.

(1) Ausdruck	→ Term Term + Ausdruck Term - Ausdruck	Grammatik für arithmetische Aus- drücke
(2) Term	→ Faktor Faktor * Term Faktor / Term	
(3) Faktor	→ a b c (Ausdruck)	

Ein Satz der zu dieser Grammatik gehörenden Sprache ist zum Beispiel $a*(b-c)$. Der Nachweis ergibt sich durch Ableitung aus dem Startsymbol *Ausdruck*:

Ausdruck	→ (nach Regel 1)	Ableitung des Aus- drucks $a*(b-c)$
Term	→ (nach Regel 2)	
Faktor * Term	→ (nach Regel 3)	
$a * \text{Term}$	→ (nach Regel 2)	
$a * \text{Faktor}$	→ (nach Regel 3)	
$a * (\text{Ausdruck})$	→ (nach Regel 1)	
$a * (\text{Term} - \text{Ausdruck})$	→ (nach Regel 2)	
$a * (\text{Faktor} - \text{Ausdruck})$	→ (nach Regel 3)	
$a * (b - \text{Ausdruck})$	→ (nach Regel 1)	
$a * (b - \text{Term})$	→ (nach Regel 2)	
$a * (b - \text{Faktor})$	→ (nach Regel 3)	
$a * (b - c)$		

Die Ableitung ist beendet, wenn der Ableitungsterm nur noch aus *Terminalen* besteht. Im ersten Beispiel waren dies die konkreten Wörter, hier sind es die Faktoren a, b und c, sowie die Klammern und Rechenzeichen. Ausdruck, Term und Faktor nennt man *Variablen* der Grammatik.

Terminale

Variable

16.1.3 Beispiel 3: Palindrome

Als nächstes Beispiel betrachten wir die Palindrome aus den Buchstaben a und b. Ein Palindrom ergibt vorwärts wie rückwärts gelesen das gleiche Wort.

Grammatik für
Palindrome

- (1) $S \rightarrow a \mid b$
- (2) $S \rightarrow a S a \mid b S b$

Die Buchstaben a und b sind die Terminale dieser Grammatik. Das Startsymbol S ist zugleich die einzige Variable. Aus S kann das Wort ababa abgeleitet werden:

Ableitung des
Palindroms ababa

$$S \rightarrow aSa \rightarrow abSba \rightarrow ababa$$

16.1.4 Beispiel 4: 0-1-Wörter

Als letztes Beispiel betrachten wir eine Grammatik, deren Sprache aus allen 0-1-Wörtern mit gleich vielen Nullen und Einsen besteht.

Grammatik für
0-1-Wörter

- (1) $S \rightarrow 0 B \mid 1 A$
- (2) $A \rightarrow 0 \mid 0 S \mid 1 A A$
- (3) $B \rightarrow 1 \mid 1 S \mid 0 B B$

Diese Grammatik hat die Terminale 0 und 1, sowie die Variablen S, A und B. Aus S kann das Wort 010110 abgeleitet werden:

Ableitung des Wor-
tes 010110

$$S \rightarrow 0B \rightarrow 01S \rightarrow 010B \rightarrow 0101S \rightarrow 01011A \rightarrow 010110$$

16.2 Definition einer Grammatik

Wir abstrahieren von den Beispielen und kommen so zu einer formalen Charakterisierung von Grammatiken, wie sie erstmals 1959 von Noam Chomsky definiert wurde:

Eine Grammatik G besteht aus vier Komponenten:

- Einer Menge von *Terminalen*. Jeder Satz der zu einer Grammatik gehörenden Sprache besteht aus Terminalen.
- Einer Menge von *Variablen*. Variablen sind grammatische Konstrukte. Jedes Konstrukt läßt sich gemäß einer Grammatikregel in Terminale oder Variablen zerlegen.
- Einem *Startsymbol*. Das Startsymbol ist eine ausgewählte Variable, gewissermaßen das allgemeinste Konstrukt der Sprache.
- Einer Menge von Regeln, oft *Produktionen* genannt, welche festlegen, wie man aus bekannten Konstrukten neue Konstrukte ableitet.

Definition einer Grammatik

Die Definition ist so allgemein, daß sie auch das folgende Beispiel einer Grammatik umfaßt:

16.2.1 Beispiel 5: Grammatik für $a^n b^n c^n$

Im Unterschied zu den bisherigen Grammatiken kommen bei dieser Grammatik auf der linken Seite auch Terminale und Worte aus Terminalen und Variablen vor.

- (1) $S \rightarrow a S B C \mid a B C$
- (2) $C B \rightarrow B C$
- (3) $a B \rightarrow a b$
- (4) $b B \rightarrow b b$
- (5) $b C \rightarrow b c$
- (6) $c C \rightarrow c c$

Beispiel einer kontextsensitiven Grammatik

Aus dem Startsymbol S können wir $aaabbbccc$ ableiten:

$$S \rightarrow aSBC \rightarrow aaSBCBC \rightarrow aaaBCBCBC \rightarrow aaaBBCCBC \rightarrow$$

$$aaaBBCBCC \rightarrow aaaBBBCCC \rightarrow aaabBBCCC \rightarrow aaabbBCCC \rightarrow$$

$$aaabbbCCC \rightarrow aaabbbcCC \rightarrow aaabbbccC \rightarrow aaabbbccc$$

Ableitung von $a^3 b^3 c^3$

16.3 Chomsky-Hierarchie der Grammatiken

Chomsky-Hierarchie Beispiel 5 läßt vermuten und die Theorie bestätigt es, daß die durch Grammatiken beschreibbaren Sprachen von der Form der Produktionen abhängig sind. Die *Chomsky-Hierarchie* nimmt diesbezüglich eine Einteilung von Grammatiken in die *Typen* 0 bis 3 vor.

- allgemein • Jede Grammatik ist automatisch vom Typ 0. Bei Typ 0 sind den Produktionen keinerlei Einschränkungen auferlegt. Man spricht auch von allgemeinen Phasenstrukturgrammatiken.
- kontextsensitiv • Eine Grammatik ist vom Typ 1 oder *kontextsensitiv*, falls bei allen Produktionen die Worte auf der rechten Seite einer Produktion mindestens so lang sind, wie die Worte auf der linken Seite.
- kontextfrei • Eine Grammatik ist vom Typ 2 oder *kontextfrei*, wenn sie vom Typ 1 ist und bei allen Produktion die linken Seiten aus einzelnen Variablen bestehen.
- regulär • Eine Grammatik ist vom Typ 3 oder *regulär*, wenn sie vom Typ 2 ist und alle rechten Seiten von Produktionen aus einem Terminal oder einem Terminal gefolgt von einer Variablen bestehen.

Die folgende Übersicht zeigt die Form der Produktionen in Abhängigkeit vom Typ der Grammatik, wobei A und B einzelne Symbole, α , β und γ Symbolfolgen aus Terminalen und Variablen, und a ein Terminal bezeichnet

Tabelle 16-1
Produktionsformen
der Chomsky-
Hierarchie

Grammatik	Typ	Produktionsformen
allgemein	0	beliebig
kontextsensitiv	1	$\alpha A \beta \rightarrow \alpha \gamma \beta$
kontextfrei	2	$A \rightarrow \alpha$
regulär	3	$A \rightarrow a \mid a B$

Die Grammatik in Beispiel 5 ist kontextsensitiv, die Beispiele 1 bis 4 zeigen kontextfreie Grammatiken. Eine reguläre Grammatik für Bezeichner aus den beiden Terminalen a und 1 ist gegeben durch:

Beispiel einer regulären Grammatik

- (1) $S \rightarrow a \mid aA$
- (2) $A \rightarrow a \mid 1 \mid aA \mid 1A$

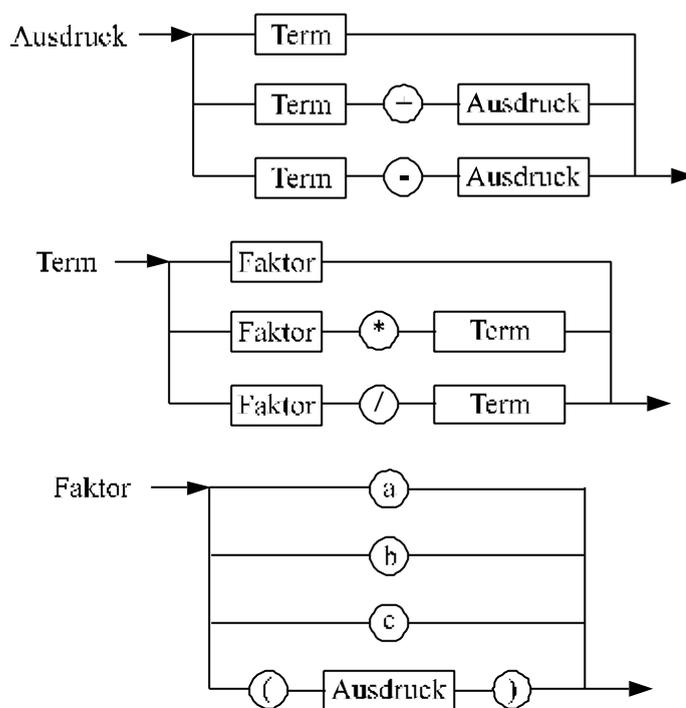
16.4 Syntaxdiagramme

Programmiersprachen sind formale Sprachen, denen in der Regel spezielle kontextfreie Grammatiken zu Grunde liegen. Grammatiken von Programmiersprachen werden üblicherweise in Backus-Naur-Form (BNF-Form) dargestellt. Wir benutzen einen vereinfachten Formalismus, bei dem beispielsweise ::= durch \rightarrow ersetzt ist.

Backus-Naur-Form

Eine anschaulichere, aber gleichwertige Darstellung von Grammatiken ist mit *Syntaxdiagrammen* möglich. Diese stellen Produktionen graphisch dar, wobei Terminale in Kreisen oder Ovalen und Variable in Rechtecken stehen. Für unser Beispiel mit den arithmetischen Ausdrücken sehen die Syntaxdiagramme wie folgt aus:

Syntaxdiagramme, Ovale und Rechtecke



Syntaxdiagramme für arithmetische Ausdrücke

Die Umwandlung von Grammatiken in dazu äquivalente Syntaxdiagramme kann schematisch erfolgen. Sequenzen von Terminalen und Symbolen setzt man in entsprechende Sequenzen von Ovalen und Rechtecken um. Alternativen, die in der Grammatik wahlweise durch den senkrechten Strich oder durch mehrere Produktionsregeln mit der gleichen linken Seite ausgedrückt werden, zeichnet man im Syntaxdiagramm durch entsprechende Verzweigungen. Die Wiederholung tritt in unserer Grammatik in Form der Rekursion auf. Dementsprechend haben wir auch rekursive Syntaxdiagramme.

Umwandlung von Grammatiken in Syntaxdiagramme

Die folgende Tabelle stellt für die grundlegenden Kontrollstrukturen den Grammatikregeln die entsprechenden Syntaxdiagramme gegenüber.

Tabelle 16-2
Gegenüberstellung
äquivalenter Kon-
trollstrukturen,
Grammatikregeln
und Syntaxdia-
gramme

Kontrollstruktur	Grammatikregel	Syntaxdiagramm
Sequenz	$A \rightarrow B C$	
Selektion	$A \rightarrow B C$	
Rekursion	$A \rightarrow B A$	
Iteration	$A \rightarrow \{ B \}$	

Schaut man sich in Büchern zu Pascal Syntaxdiagramme an, so findet man anstelle der Rekursion meist Wiederholungsschleifen. Zur Beschreibung solcher Schleifen erweitert man die Meta-Sprache der Grammatik um die geschweiften Klammern $\{ \}$. Was in geschweiften Klammern steht, kann ganz weggelassen oder beliebig oft wiederholt werden. Die entspricht in Pascal der WHILE-Schleife.

Da sich rekursive Strukturen in Prolog besser als iterative Strukturen verarbeiten lassen, erzählt folgende Bildergeschichte, wie sich eine Iteration in eine Rekursion umwandeln läßt:

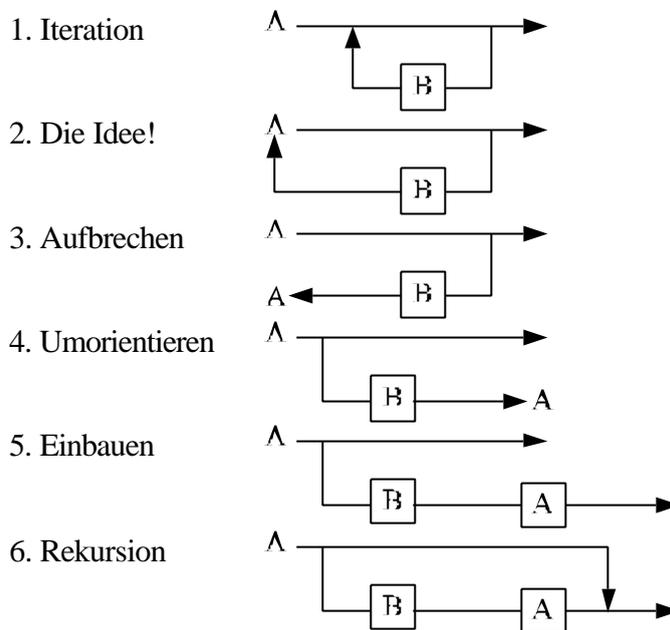


Abb. 16-1
Umwandlung eines
iterativen in ein
rekursives
Syntaxdiagramm

16.5 Modellierung von Grammatiken in Prolog

Zur maschinellen Verarbeitung von Grammatiken bilden wir deren Komponenten in Prolog ab. Für jede Komponente brauchen wir ein Prädikat.

```
terminal(X):- member(X, [kauft, liebt, liest]).
terminal(X):- member(X, [die, das, ein]).
terminal(X):- member(X, ['Buch', 'Mädchen', 'Kartoffeln'])
terminal('Peter').
```

Modellierung der
Grammatik einfacher
deutscher
Sätze in Prolog

```
variable(X):- member(X,[satz, subjekt, praedikat, objekt,
eigename, substantivgruppe,
akkusativergaenzung, artikel,
substantiv, verb]).
```

```
startsymbol(satz).
```

```
produktion(satz, [subjekt, praedikat, objekt]).
produktion(subjekt, [eigename]).
produktion(subjekt, [substantivgruppe]).
produktion(substantivgruppe,[artikel, substantiv]).
```

```

produktion(praedikat, [verb]).
produktion(objekt, [akkusativergaenzung]).
produktion(akkusativergaenzung, [substantivgruppe]).
produktion(verb, [kauft]).
produktion(verb, [liebt]).
produktion(verb, [liest]).
produktion(artikel, [die]).
produktion(artikel, [das]).
produktion(artikel, [ein]).
produktion(substantiv, ['Buch']).
produktion(substantiv, ['Mädchen']).
produktion(substantiv, ['Kartoffeln']).
produktion(eigename, ['Peter']).

```

Repräsentierung
der Grammatikre-
geln in Klauseln
des Prädikats
produktion/2

Die hier benutzten vier *terminal*-Klauseln könnte man auch zu einer Klausel zusammenfassen, indem man Verben, Artikel, Substantive und den Eigennamen in eine Liste packt. Die Variablen der Grammatik müssen klein geschrieben werden, damit sie nicht zu Prolog-Variablen werden. Das erste Argument des Prädikats *produktion/2* ist ein Kopf einer Grammatikregel, das zweite der zugehörige Rumpf. Da der Regelrumpf aus mehreren Variablen und Terminalen bestehen kann, wird er generell als Liste ausgeführt. Wir betrachten nur kontextfreie Grammatiken. Sollen auch kontextsensitive Grammatiken modelliert werden, so wird man den Regelkopf gleichfalls als Liste repräsentieren.

Die Modellierung läßt sich problemlos auf die anderen Beispiele übertragen. Für das vierte Beispiel ergibt sich:

Modellierung der
Grammatik für
0-1-Wörter in Pro-
log

```

terminal(0).
terminal(1).
variable(X):- member(X, [s,a,b]).
startsymbol(s).

produktion(s,[0,b]).
produktion(s,[1,a]).

produktion(a,[0]).
produktion(a,[0,s]).
produktion(a,[1,a,a]).

produktion(b,[1]).
produktion(b,[1,s]).
produktion(b,[0,b,b]).

```

16.6 Erzeugte Sprache - Breitensuche in Graphen

Einer Grammatik G läßt sich eine formale Sprache $L(G)$ zuordnen: $L(G)$ ist die Menge aller Worte aus Terminalen, die sich aus dem Startsymbol durch Anwendung der Produktionen erzeugen lassen.

formale Sprache
einer Grammatik

In der Theoretischen Informatik benutzt man den Begriff *Wort* für eine Folge von Terminalen und Variablen. Dies macht Sinn, wenn Terminale und Variablen mit einzelnen Buchstaben oder Ziffern bezeichnet werden. Werden hingegen aussagekräftige Bezeichner für Terminale und Variablen benutzt, so sprechen wir später auch von *Sätzen* einer Sprache. In Beispiel 4 haben wir das Wort *010110* abgeleitet, in Beispiel 1 den Satz *Peter liebt das Mädchen*.

Wörter und Sätze

Im Zusammenhang mit Grammatiken, Ableitungen und Sprachen interessieren zwei wesentliche Fragestellungen:

1. Synthese: Welche Terminalworte lassen sich überhaupt aus dem Startsymbol ableiten, was ist also $L(G)$?

Synthese-Problem

2. Analyse: Läßt sich ein gegebenes Wort aus dem Startsymbol ableiten?

Analyse- bzw.
Wort-Problem

In beiden Fällen wird eine Verbindung zwischen einem Wort und dem Startsymbol hergestellt, wobei diese Verbindung einmal beim Startsymbol beginnt, einmal dort endet.

Wir untersuchen zunächst das Problem der erzeugten Sprache und entwickeln einen Algorithmus, der $L(G)$ aufzählt. Zur Berechnung von Ableitungen benötigen wir ein geeignetes Prädikat, das einen einfachen Ableitungsschritt berechnen kann. Ein exemplarischer Ableitungsschritt ist:

Aufzählen der
erzeugten Sprache

Peter liebt Artikel Substantiv → Peter liebt das Substantiv
Das Prädikat *einfacheableitung* zur Berechnung eines einfachen Ableitungsschrittes benötigt zwei Argumente, wobei im ersten Argument eine Liste *AlteListe* aus Terminalen und Variablen übergeben wird, aus der eine neue Liste *NeueListe* berechnet wird.

ein Ableitungs-
schritt

```
einfacheableitung(+AlteListe, -NeueListe)
```

Schnittstelle des
Prädikats
einfacheableitung

Eine Variable aus *AlteListe* wird durch eine Produktion ersetzt. Im Beispiel wurde die Produktion *Artikel @ das* benutzt. Zur Implementierung des Prädikats teilen wir die Eingangsliste an einer Variablen V in zwei Listen L und R auf. L sei die Liste der Symbole vor der Variablen, R die Liste der nachfolgenden Symbole. Dann ermitteln wir eine zur Variablen V passende Produkti-

on und bestimmen den zugehörigen Rumpf der Grammatikregel. Abschließend setzen wir die drei Teillisten L, Rumpf und R zur Ergebnisliste NeueListe zusammen. Dies ergibt die folgende Implementierung:

```

Implementierung
des Prädikats ein-
facheableitung
einfacheableitung(AlteListe, NeueListe):-
    variable(V),
    aufteilen(V, AlteListe, L, R),
    !,
    produktion(V, Rumpf),
    append(Rumpf, R, RR),
    append(L, RR, NeueListe).
    
```

Bei kontextfreien Grammatiken ist die Reihenfolge, in der Variablen durch Produktionen ersetzt werden egal. Daher reicht es aus, in einer Reihenfolge abzuleiten. Der Cut sorgt dafür, daß andere Reihenfolgen nicht unnützlich untersucht werden.

Das Prädikat *aufteilen(+Element,+Liste,-LinkerTeil,-RechterTeil)* soll die Liste an der Stelle Element in zwei Listen aufteilen. LinkerTeil erhält alle vor Element stehenden Elemente, RechterTeil alle nachfolgenden. Dieses Hilfsprädikat läßt sich leicht nach der Kopf-Rest-Methode für Listen implementieren:

```

Kopf-Rest-Methode
beim Prädikat auf-
teilen
aufteilen(Element, [Element|Liste], [], Liste).
aufteilen(Element, [Kopf|Rest], [Kopf|Links], Rechts):-
    aufteilen(Element, Rest, Links, Rechts).
    
```

Mit Anfragen folgender Art kann die Funktionsweise des Prädikats *einfacheableitung* getestet werden:

```
?- einfacheableitung(['Peter',liebt,artikel,substantiv],X).
```

Einfache Ableitungen können zu mehrfachen Ableitungen erweitert werden. Die Situation ist vergleichbar mit dem Vorfahr-Eltern-Problem des einführenden Beispiels zu Familienbeziehungen in Kapitel 2. Die Eltern-Relation entspricht der einfachen Ableitung, die rekursive Vorfahr-Relation der mehrfachen Ableitung.

Relevante Unterschiede gibt es bei der Ausführung der Rekursion: Die linksrekursive Variante sucht den Ableitungsbaum in Form der Breitensuche ab, die rechtsrekursive Variante in Form der Tiefensuche:

```
ableitung(A, B):-
    einfacheableitung(A, B).
ableitung(A, B):-
    ableitung(A, C),
    einfacheableitung(C, B).
```

linksrekursiv:
Breitensuche

```
ableitung(A, B):-
    einfacheableitung(A, B).
ableitung(A, B):-
    einfacheableitung(A, C),
    ableitung(C, B),
```

rechtsrekursiv:
Tiefensuche

Die rechtsrekursive Variante ist nicht gleichwertig zur linksrekursiven. Dies zeigt sich, wenn man das Prädikat *ableitung* zur Berechnung der von der Grammatik erzeugten Sprache einsetzt. Die rechtsrekursive Variante gerät wegen der Tiefensuche leicht in einen unendlichen Ast des Suchbaums und kann dann die Worte der Sprache nicht vollständig aufzählen. Die linksrekursive Variante bekommt Probleme, wenn es sich um eine endliche Sprache handelt: nachdem Sie alle Worte aufgezählt hat, gerät Sie in eine unendliche Rekursionsschleife.

Am Beispiel einer Grammatik für beliebige 0-1-Wörter sei dies illustriert. Zur Grammatik $S \rightarrow 0 \mid 1 \mid 0S \mid 1S$ gehört der dargestellte Ableitungsbaum, der alle Ableitungen bis zur dritten Stufe enthält.

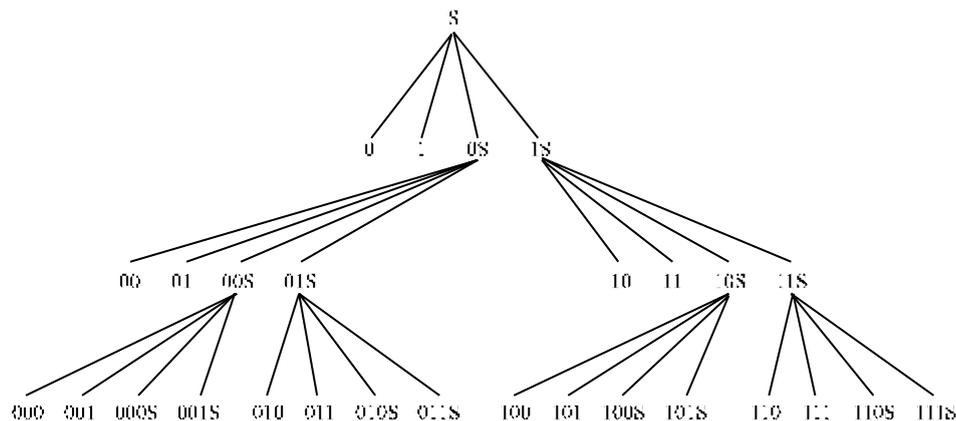


Abb. 16-2
Ableitungsbaum für
0-1-Wörter

Die rechtsrekursive Tiefensuche kommt nie in den rechten Ast. Sie liefert nacheinander die Wörter: 0, 1, 00, 01, 000, 001,...

Die linksrekursive Breitensuche hingegen zählt die Wörter dieser Sprache schön auf: 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111,...

Zur Sprache gehören nur Wörter aus Terminalen. Solche Wörter können mittels Kopf-Rest-Methode leicht identifiziert werden.

Erkennen von Terminal-Wörtern

```
terminalwort([Kopf|Rest]):-
    terminal(Kopf),
    terminalwort(Rest).
terminalwort([]).
```

Das Prädikat *erzeugteSprache* versucht ausgehend vom Startsymbol eine Ableitung zu finden, die zu einem Terminalwort führen. Mittels Backtracking werden der Reihe nach Wörter der erzeugten Sprache gefunden:

Aufzählen der erzeugten Sprache

```
erzeugteSprache(Wort):-
    startsymbol(Start),
    ableitung([Start], Wort),
    terminalwort(Wort).
```

16.7 Syntaktische Analyse mit Akzeptoren

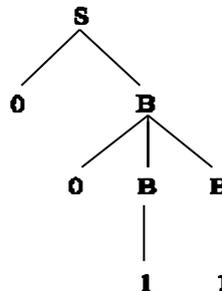
das Analyse-Problem

Mit den bisher entwickelten Prädikaten können Worte synthetisiert werden. Das ist ganz nützlich, wenn man an der von einer Grammatik erzeugten Sprache $L(G)$ interessiert ist. Weitaus wichtiger ist aber die Analyse von Worten. Die Analyse stellt fest, ob ein gegebenes Wort einer Sprache $L(G)$ angehört. Soll das Wort weiterverarbeitet werden, so muß außerdem die grammatische Struktur des Wortes in Form eines Ableitungsbaums bestimmt werden.

Ableitungsbaum

Der Ableitungsbaum zeigt in graphischer Form eine Ableitung an. Die Wurzel des Ableitungsbaumes ist das Startsymbol. Innere Knoten sind durch Variable markiert, die Blätter durch Terminalen der zugrundeliegenden Grammatik. Die Anwendung einer Produktion erzeugt die Nachfolgeknoten eines inneren Knotens. Für die Grammatik der 0-1-Wörter ist in Abbildung 16-3 der Ableitungsbaum der Ableitung $S \rightarrow 0B \rightarrow 00BB \rightarrow 001B \rightarrow 0011$ angegeben.

Abb. 16-3
Ableitungsbaum für
das Wort 0011



In einem Compiler übernimmt der *Parser* (dt. = Zerteiler) die Aufgabe der syntaktischen Analyse. Er untersucht, ob das Quellprogramm syntaktisch korrekt ist, also der zugrundeliegenden Grammatik entspricht, meldet gegebenenfalls Fehler und gibt den dazugehörigen Ableitungsbaum aus.

Parser

Ein Parse-Algorithmus muß möglichst effizient sein, also ein lineares Zeitverhalten haben. Dies läßt sich im allgemeinen für kontextfreie Sprachen nicht erreichen. *Nichtdeterministische Kellerautomaten* (siehe Kapitel 18) erkennen kontextfreie Sprachen, haben aber ein exponentielles Zeitverhalten. Für Programmiersprachen verwendet man *eingeschränkte kontextfreie Grammatiken*, welche durch *deterministische Kellerautomaten* in Linearzeit erkannt werden können.

eingeschränkte
kontextfreie
Grammatiken

Wir betrachten abschließend kurz das Wortproblem. Es soll festgestellt werden, ob ein gegebenes Wort zur Sprache $L(G)$ gehört. Der einfachste Algorithmus verwendet das Prädikat *ableitung*, um durch die Methode *Generieren und Testen* ein Wort zu erkennen:

```
ableitbar(Wort):-
    terminalwort(Wort),
    startsymbol(Start),
    ableitung([Start], Wort).
```

Lösung des Wort-
Problems

Wenn das in Frage stehende Wort zur Sprache $L(G)$ gehört, so wird es durch *ableitbar* erkannt. Bei nicht zur Sprache gehörenden Wörtern terminiert der Algorithmus nicht, weil immer neue Versuche gemacht werden, das nicht herleitbare Wort herzuleiten. Bei kontextsensitiven, kontextfreien und regulären Sprachen kann der Algorithmus durch Berücksichtigung der Wortlänge so verbessert werden, daß aus der Semi-Entscheidbarkeit eine Entscheidbarkeit wird. Es müssen nur die endlich vielen Ableitungen untersucht werden, die Wörter erzeugen, welche höchstens so lang wie das zu untersuchende Wort sind.

Semi-
Entscheidbarkeit

Die Arbeitsweise effizienterer Algorithmen kann erst dann sinnvoll verstanden und theoretisch reflektiert werden, wenn die zugehörigen Maschinenmodelle zur Verfügung stehen. Wir behandeln deshalb in den beiden nächsten Kapitel die *Automaten* und *Kellerautomaten* und kommen in Kapitel 21 auf die Lösung des Wortproblems kontextfreier Sprachen mittels nichtdeterministischer Kellerautomaten zurück.

16.8 Aufgaben

1. Stellen Sie die Grammatik für arithmetische Ausdrücke aus Beispiel 2 in Prolog dar.
2. Wie viele und welche Ableitungsbäume gibt es für das 0-1-Wort 110010 aus Beispiel 4?
3. Die arithmetischen Ausdrücke wurden mittels einer Infix-Grammatik definiert. Geben Sie eine Präfix-Grammatik an.
- 4a) Zeichnen Sie in Schleifenform ein Syntaxdiagramm für eine ein- oder mehrmalige Wiederholung, wie sie bei Repeat-Schleifen auftritt.
- b) Erzählen Sie in einer Bildergeschichte, wie diese Iteration in eine Rekursion umgewandelt werden kann.
5. Entwerfen Sie ein Prädikat, das für
 - a) kontextfreie Sprachen
 - b) kontextsensitive Sprachendas Wortproblem entscheidet.

17 Automaten

Schon vor Jahrhunderten hat der Mensch Maschinen gebaut, die seine *körperliche Arbeit* unterstützten und erleichterten. Diese Maschinen verarbeiteten Energie oder Materie. Die Dampfmaschine und der Flaschenzug sind Beispiele dafür.

körperliche Arbeit

Die Informatik setzt sich mit *informationsverarbeitenden Maschinen* auseinander. Informationsverarbeitende Maschinen unterstützen und erleichtern die *geistige Arbeit* von Menschen. Die Technische Informatik geht der Frage nach, wie informationsverarbeitende Maschinen gebaut werden können, wie die Hardware solcher Maschinen aussieht. Die Theoretische Informatik setzt sich mit der Leistungsfähigkeit und den Grenzen informationsverarbeitender Maschinen auseinander.

geistige Arbeit

17.1 Schaltnetze

Zum Verständnis der Wirkungsweise informationsverarbeitender Maschinen gehört die Einsicht, wie mittels Hardware geistige Grundoperationen nachgebildet werden können. Sie läßt sich im Unterricht schon früh vermitteln, wenn man altersgemäße Hardware benutzt. Mit einfachen Schaltern kann die logische Und-Verknüpfung als Reihenschaltung und die logische Oder-Verknüpfung als Parallelschaltung realisiert werden. Die logische Negation erreicht man mit einem relaisgesteuerten Schalter.

geistige Grundoperationen

Auf einem höheren Abstraktionsniveau arbeitet man mit Gatterschaltungen. Logische Gatter sind Bauteile, welche die logischen Verknüpfungen durchführen. Man kann Sie hardwaremäßig in Form von integrierten Schaltungen kaufen. Alternativ werden Baukästen mit Gattern und Steckbrettern angeboten, mit denen man dann Gatterschaltungen aufbauen kann. Baukästen gibt es auch als Softwaresimulation.

Gatterschaltungen

Ein Gatter ordnet binären Eingangssignalen genau ein binäres Ausgangssignal zu. Wir benutzen für die binären Signale die Ziffern 0 und 1. Die logischen Grundschaltungen werden dann durch folgende Schaltwerttabellen beschrieben:

a	b	und
0	0	0
0	1	0
1	0	0
1	1	1

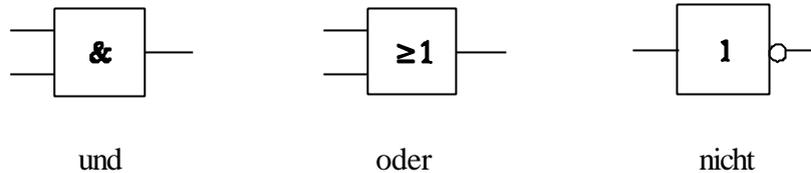
a	b	oder
0	0	0
0	1	1
1	0	1
1	1	1

a	nicht
0	1
1	0

Tabelle -117-1
Schaltwerttabellen
der logischen
Grundschaltungen

Die Gatter stellt man DIN-gerecht durch diese Symbole dar:

DIN-Symbole logischer Grundgatter



Mit Hilfe der logischen Grundgatter lassen sich typische geistige Fähigkeiten maschinell nachbilden. Dazu betrachten wir drei Beispiele:

17.1.1 Logisches Schließen

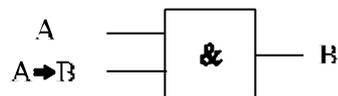
Eine häufig benutzte logische Schlußregel ist der sogenannte *Modus ponens*, mit dem aus einer wahren Voraussetzung A und einer Implikation $A \Rightarrow B$ eine wahre Konsequenz geschlossen wird. Angenommen die Implikation *Wenn der Schalter betätigt wird, dann brennt das Licht.* ($A \Rightarrow B$) stimmt und die Voraussetzung *Der Schalter wurde betätigt.* (A) ist gegeben, dann können wir schließen, daß *Das Licht brennt.* (B)

Modus ponens

$$A \wedge (A \Rightarrow B) \Rightarrow B$$

Mit einer einfachen Und-Schaltung läßt sich der logische Schluß vornehmen:

Gatterschaltung für den Modus ponens



17.1.2 Addieren

Das maschinelle Nachbilden der schriftlichen Addition ist einfach, wenn man im Binärsystem rechnet. Bei der Addition zweier Dualziffern sind nur vier Fälle zu unterscheiden. Die folgende Tabelle gibt jeweils die Summe S und den Übertrag \ddot{U} an:

A	B	S	\ddot{U}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Tabelle 17-2
Addition zweier
Dualziffern

Den Übertrag \ddot{U} erhält man durch die Konjunktion $A \wedge B$, die Summe S durch ein Entweder-Oder in der Form $(\neg A \wedge B) \vee (A \wedge \neg B)$. Daraus ergibt sich das Schaltnetz:

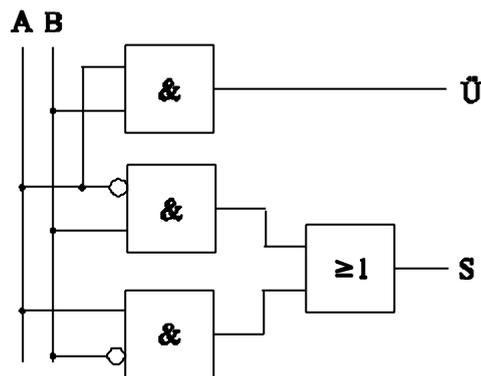


Abb. 17-1
Schaltnetz für einen Halbaddierer

Zwei solche *Halbaddierer* benutzt man zum Aufbau eines *Volladdierers*, bei dem neben den zwei Summanden noch ein Übertrag berücksichtigt wird. Mehrere Volladdierer kann man zusammenschalten, um ein Schaltnetz für einen 4-Bit oder 8-Bit-Paralleladdierer zusammenzubauen. Näheres findet man in [Mod2] oder [Bau3].

17.1.3 Prüfung binär codierter Dezimalziffern

Die Numerikmodule der ersten Pentium-Prozessoren sind so fehlerhaft, daß beim Dividieren von Real-Zahlen Fehler in der Größenordnung 10^{-5} auftreten. Für finanzmathematische Anwendungen ist das natürlich untragbar.

Generell treten Fehler dadurch auf, daß Real-Zahlen intern auf 8 oder 16 Stellen gerundet dargestellt werden. Rundungsfehler kann man dadurch umgehen, daß man alle Ziffern einer Zahl binär codiert im Speicher ablegt. Dazu benötigt man pro Ziffer 4 Bit, wobei man von den 16 möglichen Bitkombinationen, den sogenannten *Tetraden*, nur 10 benötigt. Die restlichen sechs Tetraden stellen ungültige Ziffern dar.

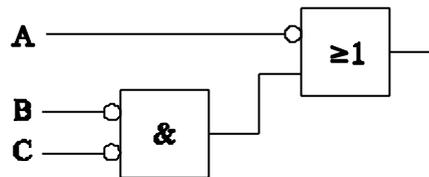
Tabelle 17-3
Codierung von Ziffern durch Tetraden

Tetrade	Ziffer
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7

Tetrade	Ziffer
1000	8
1001	9
1010	ungültig
1011	ungültig
1100	ungültig
1101	ungültig
1110	ungültig
1111	ungültig

Mit einer Prüfschaltung soll festgestellt werden, ob eine gültige oder ungültige Tetrade vorliegt. Dazu bezeichnen wir die vier Bits der Reihe nach mit A, B, C und D. Für $A = 0$ oder für $B=0$ und $C=0$ haben wir eine gültige Ziffer. Daraus ergibt sich dieses Schaltnetz:

Abb. 17-2
Prüfschaltung für gültige Tetraden



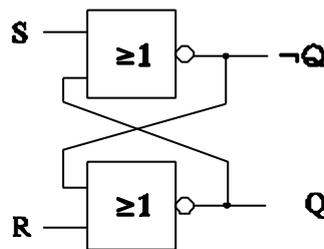
17.2 Speicher

Ein Schaltnetz ordnet jedem Eingangssignal genau ein Ausgangssignal zu, sein Verhalten ist also vollständig durch die Eingangssignale bestimmt. Dies reicht zur Lösung einiger einfacher Probleme aus, komplexere Probleme können mit Schaltnetzen meist nicht gelöst werden.

Bei vielen Probleme benötigt man mehrere Schritte, um eine Lösung zu erhalten. Dazu muß man sich bei jedem Schritt Informationen speichern, welche in nachfolgenden Schritten weiter verwendet werden. Leistungsfähigere informationsverarbeitende Maschinen benötigen also einen *Speicher*.

Speicherelemente können ebenfalls mit Schaltnetzen aufgebaut werden. Damit ein Ergebnis für den nächsten Schritt wieder zur Verfügung steht, muß es auf einen Eingang zurückgeführt werden. Speicherelemente können also grundsätzlich durch *rückgekoppelte Schaltungen* realisiert werden.

Ein Speicherelement zum Speichern eines Bits läßt sich wie folgt realisieren:



Informations-
speicherung

rückgekoppelte
Schaltungen

Abb. 17-3
Gatterschaltung
eines RS-Flipflops

Über den S(et)-Eingang setzt man den Speicher auf $Q=1$, über den R(eset)-Eingang setzt man ihn auf 0 zurück. Diesen 1-Bit-Speicherbaustein nennt man RS-Flipflop. In [Bau3] und [Mod2] findet man weiterführendes über Speicher.

Mit Schaltnetzen und Speichern lassen sich leistungsfähigere Maschinen, die sogenannte *Automaten* bauen. Mit Automaten werden wir uns jetzt ausführlich befassen.

Automaten

Unser Getränkeautomat wird durch folgenden Zustandsgraphen beschrieben (vgl. [Bur1]):

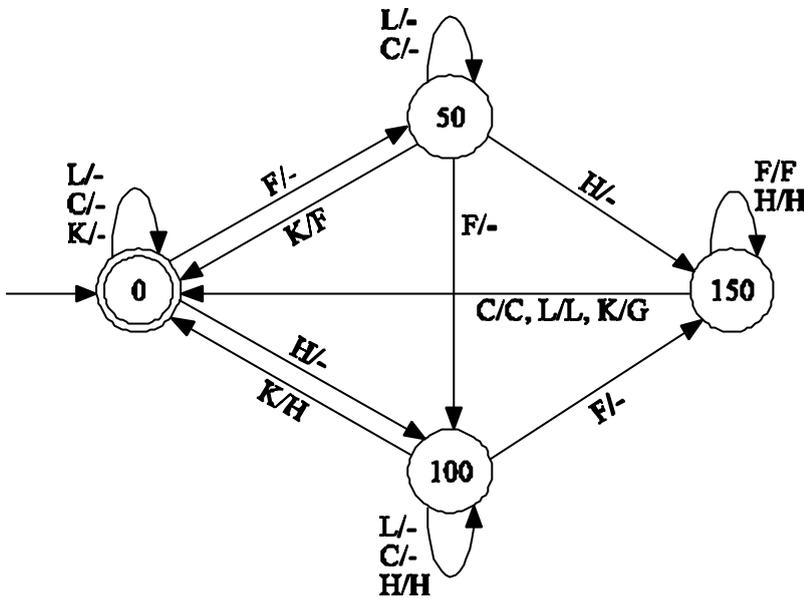


Abb. 17-4
Zustandsgraph
eines Getränke-
automaten

Vereinfachend wurden Kanten, die die selben Zustandsknoten verbinden, zusammengefaßt und mehrfach beschriftet.

17.3.2 Automatensteuerung eines Aufzugs

Ein Aufzug bedient Erdgeschoß, 1. und 2. Obergeschoß. Der Aufzug kann von jedem Stockwerk aus gerufen werden, Passagiere können im Aufzug ein Fahrziel auswählen. Priorität haben Anforderungen, bei der die momentane Fahrtrichtung erhalten bleibt. Weitere Details, wie zum Beispiel Öffnen und Schließen der Tür, berücksichtigen wir nicht.

Eingaben sind die acht verschiedenen möglichen Anforderungskombinationen, die wir wie folgt zusammenfassen: $A=\{\}$, $B=\{E\}$, $C=\{1\}$, $D=\{2\}$, $E=\{E, 1\}$, $F=\{E, 2\}$, $G=\{1, 2\}$, $H=\{E, 1, 2\}$ wobei E für Erdgeschoß, 1 für 1. Obergeschoß und 2 für 2. Obergeschoß steht.

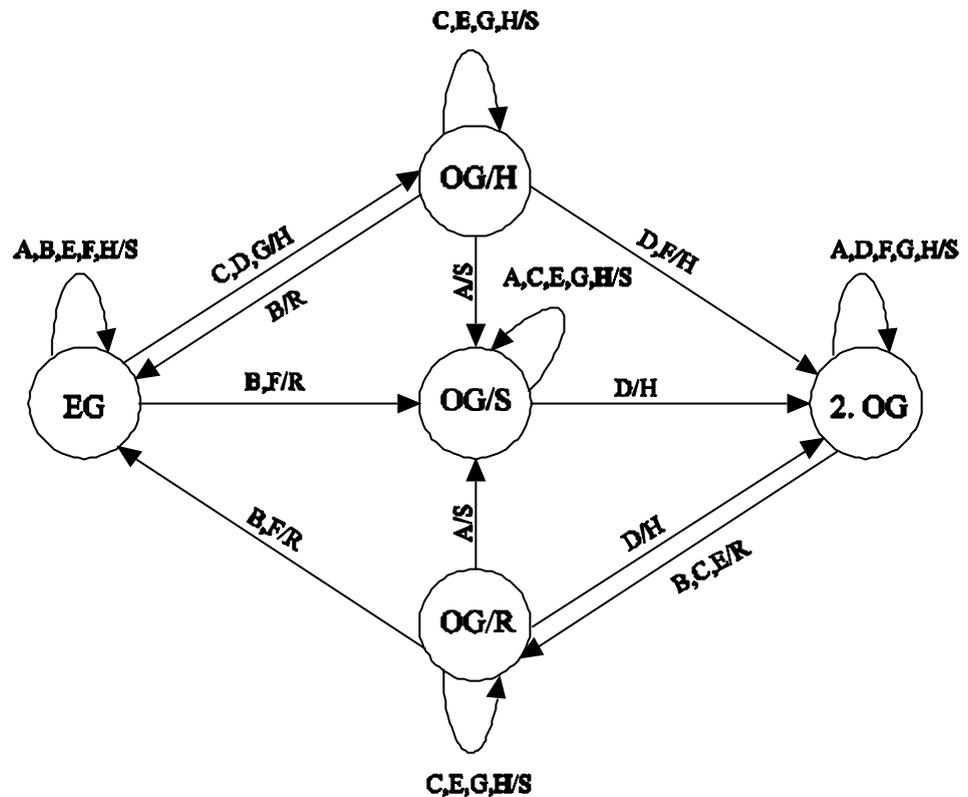
Als Ausgaben legen wir fest: H für Hochfahren, R für Runterfahren und S für Stillstand.

Im Erdgeschoß und im 2. Obergeschoß spielt die Fahrtrichtung keine Rolle. Der Zustand des Aufzugs ist durch das Geschoß bestimmt. Beim 1. Obergeschoß wird er Zustand des Aufzugs auch durch die aktuelle Fahrtrichtung bestimmt, welche wie bei den Ausgaben durch die drei Richtungen H, R und S bestimmt ist.

Ein- und Ausgaben
einer Aufzugs-
steuerung

Im Zustandsgraph legen wir nun das Verhalten des Aufzugs fest:

Abb. 17-5
Zustandsgraph
einer Aufzugs-
steuerung



lexikalische
Analyse

Automaten setzt man im Interpreter- und Compilerbau für die lexikalische Analyse ein. Deren Aufgabe besteht darin, aus den einzelnen Zeichen eines Quelltextes die dargestellten Symbole zu erkennen. Uns macht es keine Schwierigkeiten in den 18 Zeichen *Brutto:=Netto*1.15* die 5 Symbole *Bezeichner Brutto*, Wertzuweisung *:=*, Bezeichner *Netto*, Multiplikationsoperator *** und Realzahl *1.15* zu erkennen. Für die maschinelle Analyse benötigt man Automaten, die in der Lage sind, die jeweiligen Symbole zu erkennen.

übersetzende
und erkennende
Automaten

Solche Automaten müssen nicht wie bisher ständig eine Ausgabe produzieren. Es reicht, wenn Sie sich nach Abarbeitung der Eingabefolge in einem Endzustand befinden, der angibt, daß die verarbeitete Zeichenfolge dem zugehörigen Symbol entspricht. Solche *erkennenden Automaten* nennt man auch *Akzeptoren*. Die zuvor betrachteten *übersetzenden Automaten*, welche jedes Eingabezeichen in ein Ausgabezeichen übersetzen, nennt man *Transduktoren*.

Akzeptoren und
Transduktoren

17.3.3 Akzeptor für Bezeichner

In vielen Programmiersprachen gilt die Regel, daß Bezeichnernamen mit einem Buchstaben beginnen, auf den beliebig viele Buchstaben und Ziffern folgen können. Im Syntaxdiagramm stellt man diesen Sachverhalt wie folgt dar:

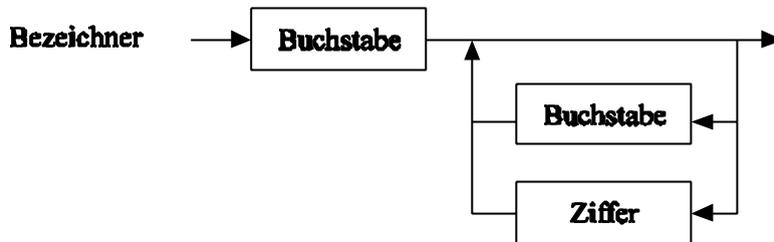


Abb. 17-6
Syntaxdiagramm
für Bezeichner

Ein Automat, der Bezeichner erkennt, erhält als Eingabe Zeichen. Im Anfangszustand hat er noch kein Zeichen gelesen, also auch noch keinen Bezeichner erkannt. Der Endzustand steht für einen erkannten Bezeichner, der Fehlerzustand signalisiert einen falsch gebildeten Bezeichner. Weitere Zustände sind nicht erforderlich, wie folgender Zustandsgraph zeigt.

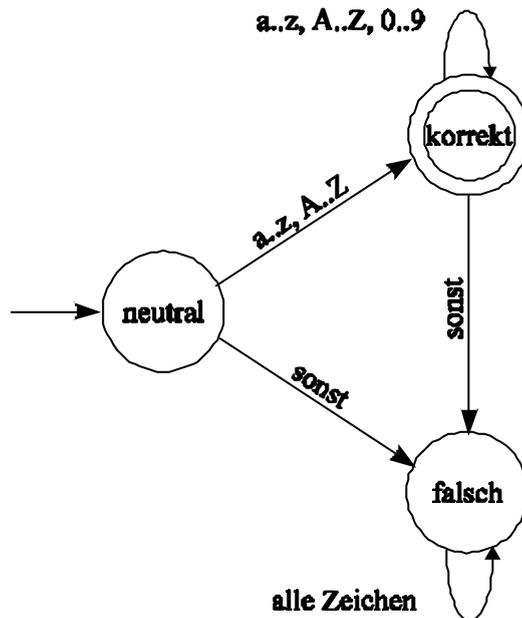
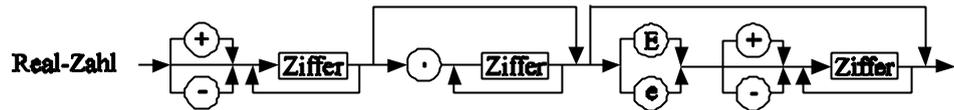


Abb. 17-7
Zustandsgraph
eines Akzeptors für
Bezeichner

17.3.4 Akzeptor für Real-Zahlen

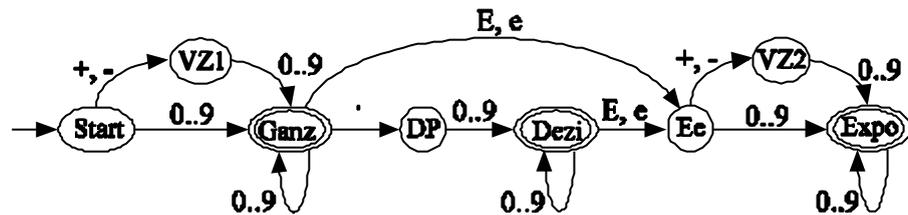
Als weiteres Beispiel für einen erkennenden Automaten betrachten wir einen Akzeptor für Real-Zahlen. Es ist gar nicht so einfach, die korrekte Schreibweise von Real-Zahlen in Worten anzugeben. Ein Syntaxdiagramm klärt den Sachverhalt:

Abb. 17-8
Syntaxdiagramm
für Real-Zahlen



Die Umsetzung in einen erkennenden Automaten ist hier schon etwas schwieriger. Damit die Übersicht erhalten bleibt, lassen wir generell im Zustandsgraph Übergänge in den Fehlerzustand weg. Weiterhin benutzen wir die Abkürzungen VZ für Vorzeichen, DP für Dezimalpunkt und Ee für die Einleitung des Exponenten.

Abb. 17-9
Zustandsgraph
eines Akzeptors für
Real-Zahlen



Der Zustandsgraph weist drei Endzustände auf. Im Endzustand *Ganz* wird eine ganze Zahl erkannt, im Endzustand *Dezi* eine Dezimalzahl und im Endzustand *Expo* eine Zahl in wissenschaftlicher Schreibweise. In allen drei Fällen handelt es sich um eine Real-Zahl.

Dieses Beispiel macht schön deutlich, wie ein Automat mit seinen Zuständen Informationen über die bereits gelesene Eingabe zusammenfassen und zur Festlegung seines Verhaltens auf nachfolgende Eingabezeichen verwenden kann.

17.4 Definition des endlichen Automaten

Nachdem wir anhand von vier Beispielen Automaten kennengelernt haben, generalisieren wir nun und gehen damit zur mathematischen Modellierung von Automaten über. Wir definieren den *endlichen Automaten* wie folgt:

Ein Endlicher Automat besteht aus sechs Komponenten:

- Einer endlichen Menge von *Zeichen*, dem *Eingabealphabet*
- und einem *Ausgabealphabet*.
- Einer endlichen Menge von *Zuständen* mit
- einem ausgezeichneten *Anfangszustand* und
- einer Menge von *Endzuständen*.
- Einer *Übergangsfunktion*, die zu jedem Eingabezeichen und Zustand den Folgezustand sowie ein Ausgabezeichen angibt.

Definition des
endlichen
Automaten

Ein Spezialfall sind die erkennenden endlichen Automaten, die sogenannten *Akzeptoren*. Bei diesen entfällt das Ausgabealphabet und die Überföhrungsfunktion gibt nur den jeweiligen Folgezustand an. Ein Akzeptor akzeptiert alle Eingabewörter, die den Automaten vom Anfangszustand in einen Endzustand überföhren.

Spezialfall
Akzeptor

17.5 Modellierung endlicher Automaten mit Prolog

Die Klärung des Automatenbegriffs erleichtert die Modellierung von Automaten in Prolog. Jede Komponente wird durch ein zugehöriges Prädikat repräsentiert. Bis auf die Übergangsfunktion ist alles sehr einfach. Der Getränkeautomat wird beschrieben durch:

```
eingabe(X):- member(X, [f, h, c, l, k]).
ausgabe(X):- member(X, [-, f, h, g, c, l]).
zustand(X):- member(X, [0, 50, 100, 150]).
anfangszustand(0).
endzustand(0).
```

Modellierung der
Komponenten des
Getränke-
automaten

Da in jedem der vier Zustände fünf verschiedene Eingabezeichen gelesen werden können, muß die Übergangsfunktion für zwanzig Wertepaare definiert werden. Einige lassen sich zusammenfassen. Im folgenden ist nach Eingabezeichen geordnet die Übergangsfunktion definiert.

Übergangsfunktion
des Getränke-
automaten

```
/*uebergang(+Zustand,+Eingabe,-Ausgabe,-NeuerZustand)*/
uebergang(Zustand, f, -, NeuerZustand):-
    Zustand =< 100,
    NeuerZustand is Zustand + 50, !.
uebergang(150, f, f, 150):- !.

uebergang(Zustand, h, -, NeuerZustand):-
    Zustand =< 50,
    NeuerZustand is Zustand + 100, !.
uebergang(Zustand, h, h, Zustand):- !.

uebergang(Zustand, c, -, Zustand):-
    Zustand =< 100, !.
uebergang(150, c, c, 0):- !.

uebergang(Zustand, l, -, Zustand):-
    Zustand =< 100, !.
uebergang(150, l, l, 0):- !.

uebergang( 0, k, -, 0):- !.
uebergang( 50, k, f, 0):- !.
uebergang(100, k, h, 0):- !.
uebergang(150, k, g, 0):- !.
```

Arbeitsweise eines
Automaten

Die in der Definition angegebenen Komponenten bestimmen einen speziellen endlichen Automaten. Die Arbeitsweise hingegen ist für alle Automaten gleich. Der Automat wird in den Anfangszustand gesetzt. Dann liest er ein Zeichen, führt den durch Zustand und gelesenes Zeichen bestimmten Zustandsübergang durch und gibt dabei das Ausgabezeichen aus. Dies macht er solange, wie noch Eingabezeichen da sind. Wenn sich nach Abarbeitung der Eingabe der Automat in einem Endzustand befindet, so hat er die Eingabe akzeptiert.

Die Arbeitsweise endlicher Automaten modellieren wir durch das Prädikat *automat*. Eine Klausel startet den Automaten, eine zweite Klausel führt die Arbeitsschritte aus und mit der dritten Klausel beendet der Automat seine Arbeit:

Implementierung
der Arbeitsweise
eines endlichen
Automaten

```
automat(Eingabe):-
    anfangszustand(Zustand),
    automat(Zustand, Eingabe).
automat(Zustand, [Eingabe|Rest]):-
    uebergang(Zustand, Eingabe, Ausgabe, NeuerZustand),
    write(Zustand), tab(2), write(Eingabe), write(' -> '),
    write(Ausgabe), tab(2), write(NeuerZustand), nl,
    automat(NeuerZustand, Rest).
automat(Zustand, []):-
```

```
endzustand(Zustand).
```

Zum Testen kann man noch programmieren:

Test des Getränkeautomaten

```
test:- automat([f, f, k, f, h, f, l, k]).
```

Die hier gezeigte Modellierung läßt sich problemlos auf das Beispiel der Aufzugssteuerung übertragen. Lediglich die Übergangsfunktion macht etwas Mühe, weil bei fünf Zuständen und acht Eingabezeichen insgesamt vierzig Übergänge zu berücksichtigen sind.

Interessanter ist die Spezialisierung hinsichtlich Akzeptoren. Bei unserem dritten Beispiel geht es um das Erkennen von Bezeichnern. Dabei muß der erkennende Automat zwischen Buchstaben und Ziffern unterscheiden können. Dies geht in Prolog nur umständlich über die entsprechenden ASCII-Codes, weswegen wir zunächst den Sprachumfang erweitern.

17.6 Zeichen und Strings in Prolog

Man hat im Informatikunterricht mit Zeichen und Strings am wenigsten Pro-

Chr und Ord

bleme, wenn man wie in Pascal mit diesen Objekten arbeiten kann. Da Prolog

zerlegen und

zusammensetzen

mit ASCII-Codes arbeitet, brauchen wir die Konvertierungsprädikate *chr* und *ord*. Für die Arbeit mit Strings könnte man gleichfalls die in Pascal verfügbaren String-Prozeduren nachbilden. Wir beschränken uns auf das Nötigste: Strings in Zeichen *zerlegen* und Zeichen zu Strings *zusammensetzen*.

Die Realisierung dieser Anforderungen setzt vertiefte Prolog-Kenntnisse voraus, weswegen man die folgende Lösung einfach zur Verfügung stellen sollte. In *fiæ*-Prolog und TV-SWI-Prolog sind Systemprädikate unterschiedlich implementiert, was zu entsprechenden Hinweisen im folgenden Quelltext führt. Zudem lassen sich in TV-SWI-Prolog Anfragen in den Quelltext aufnehmen. In *fiæ*-Prolog ist etwas mehr Handarbeit erforderlich.

Spracherweiterung CHARSTR

```

/* zerlegen(+Wort, -Zeichenliste)                                */
/* Zerlegt eine Wort in eine Liste von Zeichen.                */
/* Beispiel: zerlegen('aB+l', X). X = [a, 'B', +, 'l'] */
zerlegen(Wort, Zeichenliste):-
    atomic(Wort),
    name(Wort, Liste),
    chr(Liste, Zeichenliste), !.

/* zusammensetzen(+Zeichenliste, -Wort)                        */
/* Setzt einen Liste von Zeichen zu einem Wort zusammen.     */
/* Hinweis: Umkehrung von zerlegen                             */
/* Beispiel:                                                   */
/*     zusammensetzen([a,'X','l',b], X). X = 'aXlb'          */
zusammensetzen(Zeichenliste, Wort):-
    ord(Zeichenliste, Liste),
    name(Wort, Liste), !.

/* chr(+Ordnung, -Zeichen).                                    */
/* Bestimmt zur Ordnungszahl das Zeichen.                    */
/* Hinweis: Auch auf Listen anwendbar.                         */
chr([], []).
chr([K|R], [K1|R1]):-
    chr(K, K1),
    chr(R, R1), !.
chr(K, K1):-
    name(K2, [K]),          /* K2 ist in fix stets ein atom */
    make_atom(K2, K1).     /* in SWI kann es Integer sein */

make_atom(K2, K2):-
    atom(K2), !.
make_atom(0, '0'):- !.    /* nur für SWI-Prolog */
make_atom(K1, K2):-      /* nur für SWI-Prolog */
    int_to_atom(K1, K2).

/* ord(+Zeichen, -Ordnung).                                    */
/* Bestimmt zum Zeichen die Ordnungszahl.                    */
/* Hinweis: Auch auf Listen anwendbar.                         */
ord([], []).
ord([K|R], [K1|R1]):-
    ord(K, K1),
    ord(R, R1), !.
ord(K, K1):-

```

`name(κ , [κ_1]).`

17.7 Modellierung von Akzeptoren

Bei erkennenden endlichen Automaten entfällt das Ausgabealphabet. Die Übergangsfunktion liefert zu einem Zustand und einem Eingabezeichen nur den Folgezustand. Demnach fällt das Prädikat *ausgabe* weg und *uebergang* wird dreistellig.

Wir betrachten Beispiel 3, mit dem Akzeptor für Bezeichner. Zunächst geben wir kein Eingabealphabet an und lassen damit alle Zeichen als Eingabezeichen zu.

```
zustand(X):- member(X, [neutral, korrekt, falsch]).
anfangszustand(neutral).
endzustand(korrekt).

/* uebergang(+Zustand, +Eingabe, -FolgeZustand) */

uebergang(neutral, Eingabe, korrekt):-
    buchstabe(Eingabe), !.
uebergang(korrekt, Eingabe, korrekt):-
    (buchstabe(Eingabe); ziffer(Eingabe)), !.
uebergang(_Zustand, _Eingabe, falsch):- !.
```

Modellierung des
Akzeptors für
Bezeichner

Mit Hilfe der Spracherweiterung CHARSTR lassen sich die Klassifikationsprädikate einfach implementieren, alternativ benutzt man die vordefinierten Prädikate *is_alpha* und *is_digit* der Systembibliothek CTYPES.

Systembibliothek
CTYPES

```
/* Hilfsroutinen */
/* für SWI-Prolog im Quelltext: */ ?- consult(charstr).
/* für fix-Prolog als Anfrage:    ?- consult(charstr) */

/* Klassifikation */
buchstabe(X):-
    ('A' <= X), (X <= 'Z');
    ('a' <= X), (X <= 'z').

ziffer(X):-
    ('0' <= X), (X <= '9').
```

Klassifikations-
prädikate *buch-*
stabe und *ziffer*

Die Eingabe wird jetzt einfacher, da wir im Gegensatz zu Zeichenlisten nun Strings benutzen können, die beim Start des Automaten in Zeichenlisten zerlegt werden.

Implementierung
eines Akzeptors

```
/* automat(+Eingabeliste) */
automat(Eingabe):-
    anfangszustand(Zustand),
    zerlegen(Eingabe, Liste),
    automat(Zustand, Liste).
automat(Zustand, [Eingabe|Rest]):-
    uebergang(Zustand, Eingabe, NeuerZustand),
    write(Zustand), tab(2), write(Eingabe), write(' -> '),
    write(NeuerZustand), nl,
    automat(NeuerZustand, Rest).
automat(Zustand, []):-
    endzustand(Zustand).
```

Aufruf des Akzeptors
für Bezeichner

```
test:- automat('A2dX3 Y2nd').
```

Unser viertes Beispiel, der Akzeptor für Real-Zahlen, kann wie folgt modelliert werden:

Akzeptor für
Real-Zahlen

```
zustand(X):-
    member(X,[start,vz1,ganz,dp,dezi,ee,vz2,expo, fehler]).
anfangszustand(start).
endzustand(X):- member(X,[ganz, dez, expo]).

/* uebergang(+Zustand, +Eingabe, -FolgeZustand) */
uebergang(start, Eingabe, vz1) :- vorzeichen(Eingabe).
uebergang(vz1, Eingabe, ganz) :- ziffer(Eingabe).
uebergang(start, Eingabe, ganz):- ziffer(Eingabe).
uebergang(ganz, Eingabe, ganz) :- ziffer(Eingabe).
uebergang(ganz, '.', dp).
uebergang(dp, Eingabe, dez) :- ziffer(Eingabe).
uebergang(dez, Eingabe, dez) :- ziffer(Eingabe).
uebergang(dez, Eingabe, ee) :- e(Eingabe).
uebergang(ee, Eingabe, vz2) :- vorzeichen(Eingabe).
uebergang(vz2, Eingabe, expo) :- ziffer(Eingabe).
uebergang(ee, Eingabe, expo) :- ziffer(Eingabe).
uebergang(expo, Eingabe,expo) :- ziffer(Eingabe).

/* Klassifikation */

?- consult(charstr).
vorzeichen('+').
vorzeichen('-').
ziffer(X):- ('0' <= X), (X <= '9').
e('e').
e('E').

test:- automat('+275.443E-13').
```

17.8 Erzeugte Sprache - ein Graphenproblem

Die Menge der von einem Akzeptor A erkannten Zeichenketten nennt man die vom Akzeptor A akzeptierte Sprache $L(A)$. Eine Sprache heißt *regulär*, wenn sie von einem endlichen Automaten akzeptiert wird. In unserem 3. Beispiel ist $L(A)$ die Menge aller Bezeichner, im 4. Beispiel die Menge aller Real-Zahlen.

reguläre Sprache

Die Menge der Real-Zahlen entspricht nicht der Menge \mathbb{R} der reellen Zahlen. Erstere ist aufzählbar, wir geben einen entsprechenden Automaten an, letztere ist nicht aufzählbar. Beispielsweise kann unser Automat keine periodischen Dezimalbrüche wie $1/3=0,333\dots$ aufzählen.

Die Bestimmung der von einem Automaten erzeugte Sprache kann als graphentheoretisches Problem gedeutet werden: Gesucht sind alle Wege die im Zustandsgraph des Automaten vom Anfangszustand zu einem Endzustand führen. Jeder Weg bestimmt ein Wort der Sprache und umgekehrt gehört zu jedem akzeptierten Wort ein Weg im Zustandsgraph.

Wörter als Wege
im Zustandsgraph

Nun ist die Bestimmung aller Pfade vom Anfangszustand zum Endzustand keineswegs trivial, denn typischerweise weisen Zustandsgraphen von Automaten Zyklen auf. Insbesondere kommen öfters Selbstzyklen vor. Graphenalgorithmen müssen sich vor Zyklen in acht nehmen, um sich nicht in einem Zyklus zu verfangen.

Zyklen und
Selbstzyklen

Bei der bislang diskutierten Frage, ob ein bestimmtes Wort von einem Automaten erkannt wird oder nicht, hatten wir auch Zustandsgraphen mit Zyklen. Hier konnten wir problemlos mit *Tiefensuche* feststellen, ob ein Wort akzeptiert wird, denn die maximale Suchtiefe war von vorne herein durch die Länge des eingegebenen Wortes bestimmt. Die Tiefensuche konnte sich deshalb nicht in einem Endlos-Zyklus verfangen.

Bei der Lösung des umgekehrten Problems versagt die Tiefensuche, da wir keine Längenbegrenzung einbauen können. Potentiell erkennt ein Automat unbegrenzt lange Zeichenketten. Auch das oft bei der Tiefensuche benutzte Verfahren, sich die schon besuchten Knoten zu merken, um sie nicht ein zweites Mal zu besuchen, ist für unsere Problemstellung ungeeignet. Bei der Bestimmung der akzeptierten Worte müssen manche Knoten mehrmals besucht werden, beispielsweise wird bei der Zahl +4532.45 der Knoten *Ganz* viermal besucht.

Die Zyklenproblematik bekommt man durch *Breitensuche* in den Griff. Ausgehend vom Anfangszustand sucht man Pfade der Länge 1, die zum Endzustand führen. Dann sucht man Pfade der Länge 2 vom Anfangs- zum Endzustand. Die Suche geht mit Pfaden der Länge 3, 4, 5... weiter. Jeder gefundene Pfad bestimmt ein Wort der vom Automaten akzeptierten Sprache.

Lösung der
Zyklenproblematik
durch Breiten-
suche

Die Breitensuche ist im Prädikat *mehrfachuebergang* implementiert. Die erste Klausel bestimmt Pfade der Länge 1, die zweite Klausel Pfade, die um 1 länger sind als die bisherigen Pfade.

Aufzählen der erzeugten Sprache durch Breitensuche

```
/* erzeugte Sprache */
erzeugteSprache(Wort):-
    anfangszustand(ZustandA),
    mehrfachuebergang(ZustandA, Liste, ZustandZ),
    endzustand(ZustandZ),
    zusammensetzen(Liste, Wort).

einfachuebergang(Zustand, Eingabe, Neuzustand):-
    alphabet(Eingabe),
    uebergang(Zustand, Eingabe, Neuzustand).

mehrfachuebergang(Zustand, [Eingabe], Neuzustand):-
    einfachuebergang(Zustand, Eingabe, Neuzustand).
mehrfachuebergang(Zustand, [Kopf|Rest], Neuzustand):-
    mehrfachuebergang(Zwischenzustand, Rest, Neuzustand),
    einfachuebergang(Zustand, Kopf, Zwischenzustand).
```

Die Breitensuche kann nur dann $L(A)$ aufzählen, wenn die Cuts in den *uebergang*-Klauseln entfernt werden. Anderenfalls werden alternative Lösungen im Suchbaum abgeschnitten.

Das Prädikat *einfachuebergang* greift auf das Eingabealphabet zu. Für überzeugende Demonstrationen muß man dieses Alphabet stark einschränken. Beim Akzeptor für Bezeichner erhält man für das Alphabet {a, b, 1, 2} nacheinander die Lösungen:

Sprache der Bezeichner

a, b, aa, ba, ab, bb, a1, b1, a2, b2, aaa, baa, aba, bba, a1a, b1a,...

Beim Akzeptor für Realzahlen, eingeschränkt auf die Ziffern 1 und 2, zählt der Automat nacheinander auf:

Sprache der Real-Zahlen

1, 2, +1, -1, 11, 21, +2, -2, 12, 22, +11, -11, 111, 211, +21, -21, 121, 221, 1.1, 2.1, +12, -12...

Als erste Zahl in Exponentialschreibweise wird 1.1e1 ausgegeben.

17.9 Nichtdeterministischer endlicher Automat

Ein typische Anwendung erkennender Automaten ist die Suche nach Zeichenketten in Texten. Dabei lassen sich leicht Fehler machen. Angenommen in einem Text soll die Zeichenkette *pen* gesucht werden. Dann könnten wir dazu den Automaten mit folgendem Zustandsgraphen konstruieren, wobei die Bezeichnung $x \setminus p$ vereinfachend für alle Zeichen außer *p* steht.

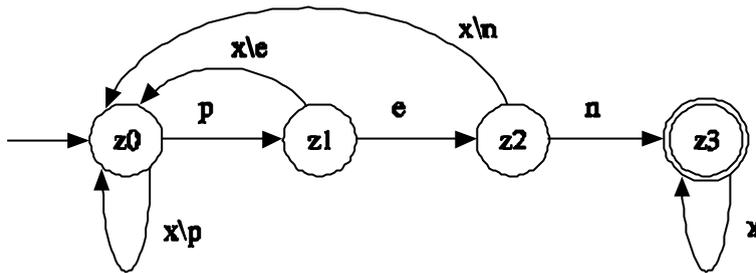


Abb. 17-10 fehlerhafter Akzeptor für die Zeichenkette *pen*

Der Automat arbeitet einwandfrei für Worte wie *Pentagon*, *Doppelpendel* und *Epen*, aber bei *Pappenstiel* findet er *pen* nicht. Beim ersten *p* nach dem Buchstaben *a* wechselt er in den Zustand *z1*, dann liest er kein *e* sondern ein *p*, weswegen er in den Zustand *z0* zurückkehrt.

Mit einem *nichtdeterministischen endlichen Automaten* läßt sich das Problem leicht lösen.

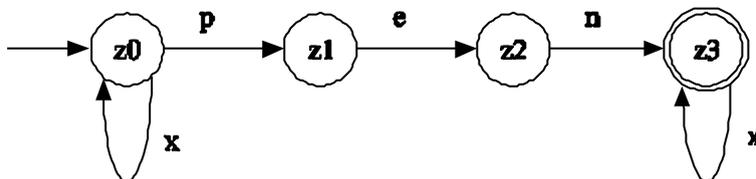


Abb. 17-11 Beispiel eines nichtdeterministischen endlichen Automaten

Der wesentliche Unterschied zwischen beiden Automatenformen besteht darin, daß beim nichtdeterministischen Automaten von einem Zustand aus mehrere Pfeile ausgehen können, die mit dem gleichen Eingabezeichen beschriftet sind. Dies ist im Beispiel für den Zustand *z0* gegeben, bei dem man mit dem Eingabezeichen *p* nach *z0* aber auch nach *z1* gehen kann.

Der nichtdeterministische Automat hat also in einem solchen Fall eine Wahlmöglichkeit für den Übergang. In Erweiterung des bisherigen Akzeptierens legt man fest, daß ein nichtdeterministischer Automat ein Wort *w* dann akzeptiert, wenn er beim Verarbeiten von *w* eine Zustandsfolge durchlaufen kann, die in einem Endzustand endet.

Da Prolog automatisches Backtracking macht, ist es überhaupt kein Problem, nichtdeterministische Automaten zu programmieren. Man läßt lediglich bei den *uebergang*-Klauseln die Cuts weg, damit nach alternativen Lösungen gesucht werden kann:

nichtdeterministischer Akzeptor für *pen*

```
zustand(X) :- member(X, [z0, z1, z2, z3]).
anfangszustand(z0).
endzustand(z3).

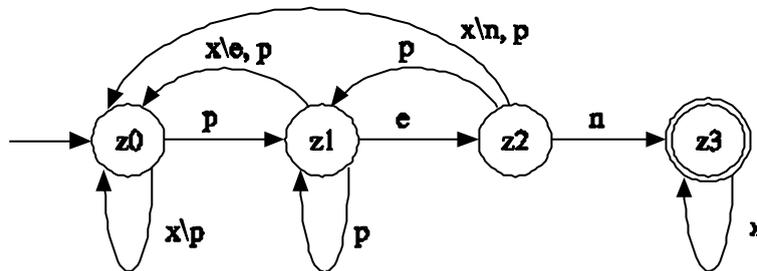
/* uebergang(+Zustand, +Eingabe, -FolgeZustand) */
uebergang(z0, Ein, z0) :- buchstabe(Ein).
uebergang(z0, 'p', z1).
uebergang(z1, 'e', z2).
uebergang(z2, 'n', z3).
uebergang(z3, Ein, z3) :- buchstabe(Ein).

test :- akzeptiere('Pappenstiel').
```

Umwandlung nicht deterministischer in deterministische Automaten

Im Rahmen der Automatentheorie wird gezeigt, daß man jeden nichtdeterministischen Automaten durch ein konstruktives Verfahren in einen deterministischen Automaten umwandeln kann. Grundsätzlich sind also nichtdeterministischen Automaten nicht leistungsfähiger als deterministische Automaten. Das Umwandlungsverfahren liefert für den *pen*-Akzeptor die Lösung:

Abb. 17-12 deterministischer Akzeptor für *pen*



17.10 Automaten mit e-Übergängen

Wir haben einen Akzeptor A_1 für Bezeichner und einen Akzeptor A_2 für Real-Zahlen. Wie kann man damit Akzeptoren bauen, die einen Bezeichner oder eine Real-Zahl, erst einen Bezeichner und dann eine Real-Zahl oder eine beliebige Folge von Real-Zahlen erkennen? Gesucht ist also ein Konstruktionsmechanismus, mit dem aus einfachen Automaten komplexere Automaten zusammengebaut werden können.

Dies geht recht einfach, wenn man bei Automaten zusätzlich ϵ -Übergänge zulässt. Bei einem ϵ -Übergang macht der Automat einen Zustandsübergang, ohne ein Eingabezeichen zu lesen. ϵ -Übergang

Zunächst betrachten wir ein Beispiel und kommen dann auf die Eingangsfrage zurück.

Beispiel: Akzeptor für $0^i 1^j 2^k$, $0 \leq i, j, k$

Der gesuchte Akzeptor soll Worte aus den Ziffern 0, 1 und 2 erkennen, die mit beliebig vielen 0 beginnen, in der Mitte beliebig viele 1er haben und mit beliebig vielen 2ern endet. Beliebiger heißt insbesondere auch null Ziffern. Der Automat soll beispielsweise 00002 erkennen.

Ein Zustandsgraph mit ϵ -Übergängen löst dieses Problem sofort:

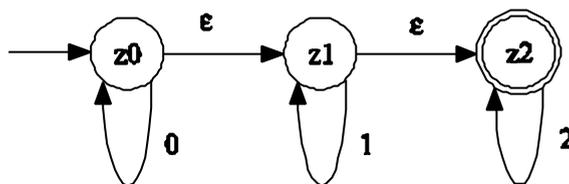


Abb. 17-13
Zustandsgraph mit ϵ -Übergängen

Ein Ergebnis der Automatentheorie ist, daß nichtdeterministische Automaten mit ϵ -Übergängen äquivalent sind zu nichtdeterministischen Automaten ohne ϵ -Übergänge. Die Lösung ohne ϵ -Übergänge ist in der Regel komplizierter:

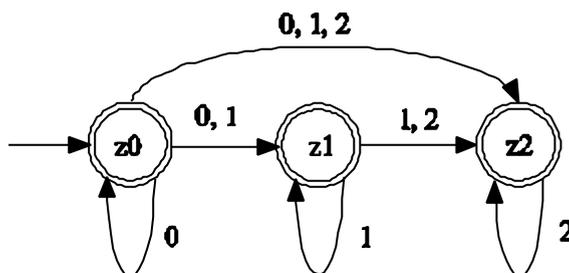


Abb. 17-14
äquivalenter
Zustandsgraph
ohne ϵ -Übergänge

ϵ -Übergänge müssen bei der Modellierung in Prolog gesondert behandelt werden, weil sie keine Eingabezeichen konsumieren. Daher müssen die beiden bisherigen *automat/2*-Klauseln um ϵ -Varianten ergänzt werden:

```

automat(Zustand, [Eingabe|Rest]):-
    eps_uebergang(Zustand, NeuerZustand),
    write(Zustand), tab(2), write(eps), write(' -> '),
    write(NeuerZustand), nl,

```

Implementierung
von ϵ -Übergängen

```
automat(NeuerZustand, [Eingabe|Rest]).
```

```
automat(Zustand,[]):- /*ε-Übergang in den Endzustand */
eps_uebergang(Zustand, NeuerZustand),
write(Zustand), tab(2), write(eps), write(' -> '),
write(NeuerZustand), nl,
automat(NeuerZustand, []).
```

Die ϵ -Übergänge unseres Beispiels können wie folgt programmiert werden:

Modellierung der ϵ -Übergänge

```
eps_uebergang(z0, z1).
eps_uebergang(z1, z2).
```

Wie kann man aus gegebenen Akzeptoren mit Hilfe von ϵ -Übergängen neue Akzeptoren zusammenbauen? Als Antwort auf diese Frage sind im folgenden Baupläne für Verkettung, Vereinigung und beliebige Wiederholung angegeben. Dabei sind a_0, a_1 und a_2 Anfangszustände, e_0, e_1 und e_2 Endzustände der beteiligten Akzeptoren A_0 und A_1 .

Abb. 17-15
Verkettung (Sequenz) von Akzeptoren



Abb. 17-16
Vereinigung (Selektion) von Akzeptoren

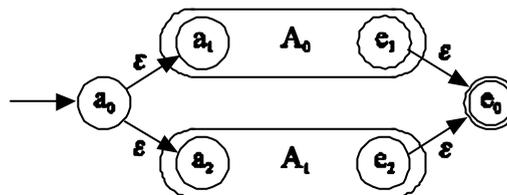
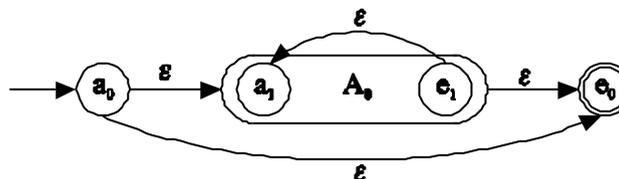


Abb. 17-17
beliebige Wiederholung (Iteration) von Akzeptoren



17.11 Reguläre Ausdrücke

Akzeptiert A_1 das Wort x und A_2 das Wort y so können wir uns nach dem Bauplan der Abbildung 17-15 einen Akzeptor für das zusammengesetzte Wort xy zusammenbauen. Mit dem Bauplan für die Vereinigung können wir uns einen Akzeptor für das Erkennen von x oder y bauen. Im folgenden bezeichnen wir das durch $x+y$. Der Bauplan für die beliebige Wiederholung gibt uns die Möglichkeit mehrfaches Auftreten eines Wortes zu erkennen, zum Beispiel xxx oder $yyyy$. Dafür schreibt man vereinfachend x^3 beziehungsweise y^4 . Die beliebige Wiederholung wird durch x^* bezeichnet.

Verkettung xy Vereinigung $x+y$

beliebige

Wiederholung x^*

Nachdem wir nun wissen, wie wir Automaten zusammenbauen können, klären wir noch, womit gebaut werden darf: zugelassen sind alle Zeichen des zugrundeliegenden Alphabets und ϵ .

Die Baupläne und das Baumaterial gestatten uns den Zusammenbau komplexer Automaten, die wir sehr einfach durch *reguläre Ausdrücke* bezeichnen können. Außer den bisher eingeführten Operatoren brauchen wir allerdings noch Klammern, um Baugruppen kennzeichnen zu können.

reguläre
Ausdrücke

Beispiele für reguläre Ausdrücke:

- $a(a+b)^*b$ entspricht einem Automaten, der alle Worte aus den Zeichen a und b erkennt, die mit a anfangen und mit b aufhören. Man kann auch sagen, daß der reguläre Ausdruck die Sprache mit den angegebenen Wörtern beschreibt.
- $0+(0+1)^*00$ ist die Sprache aus dem Wort 0 und allen Wörtern aus 0 und 1 , die auf 00 enden.
- $(r+s)^*s(r+s)$ ist die Sprache über dem Alphabet $\{r, s\}$, deren Wörter als vorletztes Zeichen ein s enthalten.

 $a(a+b)^*b$ $0+(0+1)^*00$ $(r+s)^*s(r+s)$

Zu jedem regulären Ausdruck können wir nach den Bauplänen den entsprechenden Akzeptor konstruieren. Sprachen, die durch reguläre Ausdrücke beschrieben werden, können also durch endliche Automaten erkannt werden. Die Umkehrung gilt ebenfalls: eine Sprache, die von einem endlichen Automaten erkannt wird, kann durch einen regulären Ausdruck beschrieben werden.

Damit ist die Leistungsfähigkeit endlicher erkennender Automaten ausgelottet: Endliche erkennende Automaten können genau die durch reguläre Ausdrücke beschreibbaren Sprachen erkennen.

Leistungsfähigkeit
endlicher
Automaten

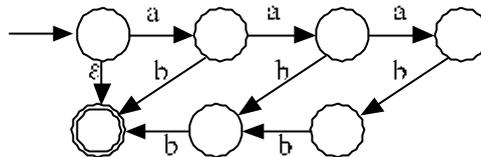
17.12 Die Grenzen endlicher Automaten

die Sprache $a^n b^n$

Die Sprache aus den a-b-Wörtern, die mit beliebig vielen a beginnen und mit beliebig vielen b enden ist regulär und wird durch den regulären Ausdruck a^*b^* beschrieben. Eine Teilsprache dieser Sprache besteht aus allen Wörtern mit gleich vielen a und b. Zugehörige Wörter sind also von der Form $a^n b^n$. Ist diese Teilsprache regulär, wird sie durch einen endlichen Automaten erkannt?

Die Baupläne geben nichts her. Also versuchen wir eine individuelle Konstruktion:

Abb. 17-18
Zustandsgraph für
Akzeptor bis $n=3$



Dieser Automat schafft ϵ , ab, aabb, aaabbb. Wir könnten ihn problemlos erweitern, so daß er alle Wörter bis sagen wir $a^{100}b^{100}$ akzeptiert. Da aber ein endlicher Automat nur eine endliche Anzahl z von Zuständen hat kann ein endlicher Automat nach obigem Prinzip nur Wörter bis $a^m b^m$ mit $m = z \text{ div } 2$ erkennen. Um $a^n b^n$ zu erkennen, muß sich ein Automat irgendwie merken, daß er n-mal der Buchstaben a gelesen hat. Da er sich nur durch Zustände etwas merken kann, benötigt er mindestens n Zustände. n ist aber unbegrenzt, weswegen der Automat letztlich unbegrenzt viele Zustände haben müßte. Dann wäre er nicht mehr endlich.

17.13 Alternative Zugänge

Analyse mit
automat
Synthese mit
erzeugteSprache

Wir haben bei der Modellierung von Automaten in Prolog einen allgemeinen Ansatz gewählt, der es sehr einfach macht, konkrete Automaten zu realisieren. Nach der Festlegung des Anfangszustands, der Endzustände und des Alphabets muß man lediglich etwas Mühe für die Übergänge aufwenden, welche den Zustandsgraphen beschreiben. Zur Analyse von Zeichenketten haben wir das Prädikat *automat*, zur Synthese von Zeichenketten das Prädikat *erzeugteSprache*. Beide Prädikate sind unabhängig vom speziellen Automaten definiert.

Alternativ kann man das Thema Automaten auch mit speziellen Ansätzen angehen. In [Süt1] wird beispielsweise ein Automat für alle Wörter aus den Buchstaben a und b wie folgt modelliert:

```

wort([a]).
wort([b]).
wort([Kopf|Rumpf]):-
    wort(Rumpf), wort([Kopf]).

```

Akzeptor für
a-b-Wörter

Dieser Automat kann korrekte Wörter erkennen, falsche Wörter ablehnen und die Sprache des Automaten erzeugen. Ein weiteres analoges Beispiel ist ein Akzeptor für Dualzahlen.

```

dualziffer(0).
dualziffer(1).
dualzahl([X]):- dualziffer(X).
dualzahl([Kopf|Rest]):-
    dualzahl(Rest),
    dualziffer(Kopf).

```

Akzeptor für Dual-
zahlen

Spezielle Automaten lassen sich leichter hinsichtlich des Themas Interpreter und Compiler erweitern. Die Interpretation einer Dualzahl kann zum Beispiel wie folgt realisiert werden:

```

dezimalwert([X], X, 1).
dezimalwert([Kopf|Rest], Wert, Stelle):-
    dezimalwert(Rest, Wert1, Stelle1),
    Stelle is Stelle1*2,
    Wert is Kopf*Stelle + Wert1.
interpretiere(Dualzahl, Wert):-
    dualzahl(Dualzahl),
    dezimalwert(Dualzahl, Wert, _).

```

Interpreter für Dual-
zahlen

17.14 Spezialisieren durch Entfalten

Am Beispiel des Akzeptors für Bezeichner betrachten wir nun die Technik des *Entfaltens*, mit welcher aus einer allgemeinen Modellierung eines Automaten eine spezielle Modellierung konstruiert werden kann. Die Technik des *Entfaltens* besteht darin, daß man untergeordnete Prädikate in übergeordnete Prädikate einsetzt. Auf diese Weise entledigt man sich einiger Hilfsprädikate und schafft sich die Möglichkeit, übergeordneten Prädikate zu vereinfachen. Diese Vereinfachungen führen dann letztlich zu einem speziellen Automaten.

Technik des
Entfaltens

Den Akzeptor für Bezeichner haben wir bislang wie folgt modelliert:

```

zustand(X):- member(X, [neutral, korrekt, falsch]).
anfangszustand(neutral).
endzustand(korrekt).

```

Akzeptor für
Bezeichner

```

uebergang(neutral, Eingabe, korrekt):-
    buchstabe(Eingabe).
uebergang(korrekt, Eingabe, korrekt):-
    (buchstabe(Eingabe); ziffer(Eingabe)).
uebergang(_Zustand, _Eingabe, falsch).

automat(Eingabe):-
    anfangszustand(Zustand),
    zerlegen(Eingabe, Liste),
    automat(Zustand, Liste).
automat(Zustand, [Eingabe|Rest]):-
    uebergang(Zustand, Eingabe, NeuerZustand),
    write(Zustand),tab(2),write(Eingabe), write(' -> '),
    write(NeuerZustand), nl,
    automat(NeuerZustand, Rest).
automat(Zustand, []):-
    endzustand(Zustand).
    
```

Das Prädikat *zustand* wird explizit nicht benutzt und kann somit entfallen. Die Ausgabe-Anweisungen der zweiten *automat*-Klausel und das Zerlegen des Eingabestrings sind zunächst entbehrlich. Mit den untergeordneten Prädikaten *anfangszustand* und *endzustand* kann die erste Entfaltung vorgenommen werden:

Entfalten von
anfangszustand
und *endzustand*

```

uebergang(neutral, Eingabe, korrekt):-
    buchstabe(Eingabe).
uebergang(korrekt, Eingabe, korrekt):-
    (buchstabe(Eingabe); ziffer(Eingabe)).

automat(Eingabe):-
    automat(neutral, Eingabe).
automat(Zustand, [Eingabe|Rest]):-
    uebergang(Zustand, Eingabe, NeuerZustand),
    automat(NeuerZustand, Rest).
automat(korrekt, []).
    
```

Im zweiten Entfaltungsschritt wird das untergeordnete Prädikat *uebergang* in das übergeordnete Prädikat *automat* eingesetzt. Davon ist lediglich die zweite *automat*-Klausel betroffen. Da es zwei *uebergang*-Klauseln gibt, entstehen beim Einsetzen auch zwei *automat*-Klauseln. Sie unterscheiden sich durch den Zustand, *neutral* oder *korrekt*.

Entfalten von
uebergang

```

automat(Eingabe):-
    automat(neutral, Eingabe).
automat(neutral, [Eingabe|Rest]):-
    buchstabe(Eingabe),
    automat(korrekt, Rest).
automat(korrekt, [Eingabe|Rest]):-
    
```

```
(buchstabe(Eingabe); ziffer(Eingabe)),
  automat(korrekt, Rest).
automat(korrekt, []).
```

Die Entfaltung ist nunmehr abgeschlossen. Eine weitere Spezialisierung ist durch Übergang von der *expliziten* zur *impliziten* Zustandsverwaltung möglich. Dazu übernimmt man den Zustand im ersten Argument in den Namen des Prädikats:

```
automat(Eingabe):-
  automatneutral(Eingabe).
automatneutral([Eingabe|Rest]):-
  buchstabe(Eingabe),
  automatkorrekt(Rest).
automatkorrekt([Eingabe|Rest]):-
  (buchstabe(Eingabe); ziffer(Eingabe)),
  automatkorrekt(Rest).
automatkorrekt([]).
```

implizite statt
explizite Zustands-
verwaltung

Das Prädikat *automat/1* kann entfallen, da man *automatneutral* direkt aufrufen kann. Die neuen Prädikate sollte man inhaltlich deuten, um zu besseren Bezeichnungen zu kommen. *automatneutral* akzeptiert einen kompletten Bezeichner, *automatkorrekt* eine Bezeichnerendung:

```
akzeptiere_bezeichner([Eingabe|Rest]):-
  buchstabe(Eingabe),
  akzeptiere_bezeichnerendung(Rest).
akzeptiere_bezeichnerendung([Eingabe|Rest]):-
  (buchstabe(Eingabe); ziffer(Eingabe)),
  akzeptiere_bezeichnerendung(Rest).
akzeptiere_bezeichnerendung([]).
```

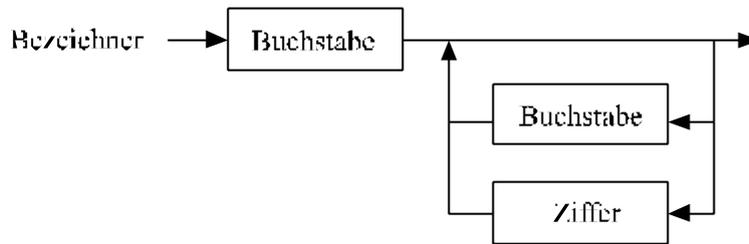
Spezial-Akzeptor
für Bezeichner

17.15 Syntaxdiagramme und Automaten

Wir haben in diesem Kapitel einen systematischen Weg zur Umsetzung des Zustandsgraphen eines Automaten in eine Prolog-Modellierung behandelt. Weitgehend unbeachtet blieb bislang der Übergang vom Syntaxdiagramm zum Zustandsgraphen. Würde man diesen Übergang systematisieren, so könnte man einen klar strukturierten Weg vom Syntaxdiagramm über den Zustandsgraphen und die allgemeine Modellierung zur speziellen Prolog-Modellierung eines Akzeptors angeben.

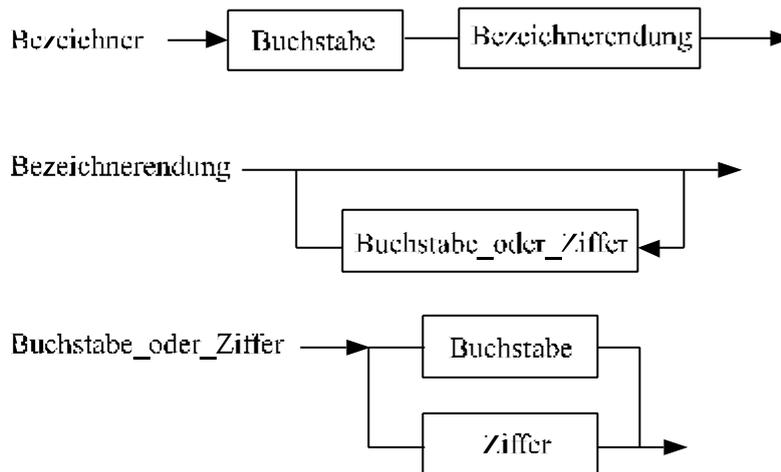
Wir betrachten im folgenden nicht den Übergang von Syntaxdiagrammen zu Zustandsgraphen, sondern widmen uns gleich der umfassenderen Aufgabe, zu einem Syntaxdiagramm einen speziellen Akzeptor zu konstruieren. Dabei orientieren wir uns am Akzeptor für Bezeichner, mit folgendem Syntaxdiagramm.

Abb. 17-19
Syntaxdiagramm
für Bezeichner



Dieses Syntaxdiagramm enthält die drei relevanten Grundstrukturen Sequenz, Iteration und Selektion. Durch Aufteilen des Syntaxdiagramms in mehrere Syntaxdiagramme wird dies deutlicher:

Abb. 17-20
Grundstrukturen im
Syntaxdiagramm
für
Bezeichner



Zum Zwecke des Erkenntnisgewinns abstrahieren wir von den Details, legen die Tabelle aus Kapitel 16 zugrunde und ergänzen eine Spalte mit einem Prolog-Akzeptor für die jeweilige Grundstruktur.

Kontrollstruktur	Grammatikregel	Syntaxdiagramm	Akzeptor in Prolog
Sequenz	$A \rightarrow B C$		akzeptiere_A:- akzeptiere_B, akzeptiere_C.
Selektion	$A \rightarrow B C$		akzeptiere_A:- akzeptiere_B. akzeptiere_A:- akzeptiere_C.
Rekursion	$A \rightarrow B A$		akzeptiere_A:- akzeptiere_B, akzeptiere_A.
Iteration	$A \rightarrow \{ B \}$	iterativ oder besser rekursiv 	akzeptiere_A:- akzeptiere_B, akzeptiere_A. akzeptiere_A.

Tabelle 17-4
Syntaxdiagramme
und Akzeptoren

Aus den Syntaxdiagrammen von Abbildung 17-20 läßt sich nun problemlos ein Akzeptor für Bezeichner konstruieren:

```
akzeptiere_bezeichner([Eingabe|Rest]):-
    akzeptiere_buchstabe(Eingabe),
    akzeptiere_bezeichnerendung(Rest).
```

```
akzeptiere_bezeichnerendung([Eingabe|Rest]):-
    akzeptiere_buchstabe_oder_ziffer(Eingabe),
    akzeptiere_bezeichnerendung(Rest).
akzeptiere_bezeichnerendung([]).
```

```
akzeptiere_buchstabe_oder_ziffer(Eingabe):-
    akzeptiere_buchstabe(Eingabe).
akzeptiere_buchstabe_oder_ziffer(Eingabe):-
    akzeptiere_ziffer(Eingabe).
```

Konstruktion eines
Akzeptors aus
Syntaxdiagrammen

Die so erhaltene Lösung entspricht bis auf Umbenennungen und Zusammenfassungen der Lösung, die wir durch Entfalten im vorangegangenen Abschnitt erhalten haben.

17.16 Aufgaben

1. $X = \{a, b\}$ sei das Alphabet eines Akzeptor.
 - a) Geben Sie das Zustandsgraph eines nichtdeterministischen Automaten an, der alle Wörter mit der Endung *aba* akzeptiert.
 - b) Wie lautet der reguläre Ausdruck für diesen Automaten?
 - c) Modellieren Sie diesen Automaten in Prolog.
 - d) Welcher deterministische Automat leistet das Gleiche?

2. Geben Sie das Zustandsdiagramm eines Automaten an, der gültige Tetraden akzeptiert.

3. Einem übersetzenden Automaten werden wechselweise die Bits zweier Summanden zugeführt.
 - a) Der Automat soll als Ausgabe die Summe produzieren.
 - b) Modellieren Sie diesen Automaten in Prolog.

4. Auf einer seriellen Leitung werden ASCII-Zeichen im folgenden Format übertragen:
 - 1 Startbit (Signallevel 0)
 - 7 Datenbits
 - 1 Paritätsbit (gerade Parität)
 - 2 Stop-Bits (Signallevel 1)
 Beispiel für einen Zeichenblock: 01010110011
 Es ist ein endlicher Automat gesucht, der den Datenstrom auf Fehler überwacht.

5. Zur formalen Sprache L gehören alle 0,1-Folgen, die gleich viele Nullen und Einsen enthalten, wobei zwei benachbarte Zeichen nie gleich sind. Ist L regulär?

6. Skizzieren Sie einen Automaten, der den regulären Ausdruck $(a+b)^*ba$ akzeptiert.

7. Entwickeln Sie einen Automaten, der alle Wörter über dem Alphabet $\{a, b, c\}$ erkennt, bei denen unmittelbar nach jedem a genau ein b kommt. Beispiele: $abc, cb, bcab, babcb$.
- 8a) Konstruieren Sie zur regulären Grammatik $S \rightarrow a \mid aA, A \rightarrow a \mid 1 \mid aA \mid 1A$ für Bezeichner über dem Alphabet $\{1, a\}$ den Zustandsgraphen eines Akzeptors.
- b) Geben Sie ein Verfahren an, mit dem eine reguläre Grammatik in einen Zustandsgraphen überführt werden kann.
9. Zur Suche nach bestimmten Textstellen in Quellprogrammen gibt es unter dem Betriebssystem UNIX, aber auch als Zusatz zu Turbo-Pascal, das Programm GREP (= Global Regular Expression Print). Wie der Name schon andeutet, kann das Suchmuster durch einen regulären Ausdruck beschrieben werden. Für Details beziehen wir uns jetzt auf die sicherlich verfügbare GREP-Variante von Turbo-Pascal.

Suchmuster können durch Hintereinanderschreiben *verkettet* werden. Zur *Vereinigung* benutzt man eckige Klammern. Den regulären Ausdruck $a+b+c$ übersetzt man für GREP in $[abc]$. Die *beliebige Wiederholung*, ausgedrückt durch $*$, bezieht sich stets auf das vorangehende Zeichen: ha^* sucht nach $h, ha, haa, haaa$, usw. Der Punkt „.“ steht für ein beliebiges Zeichen.

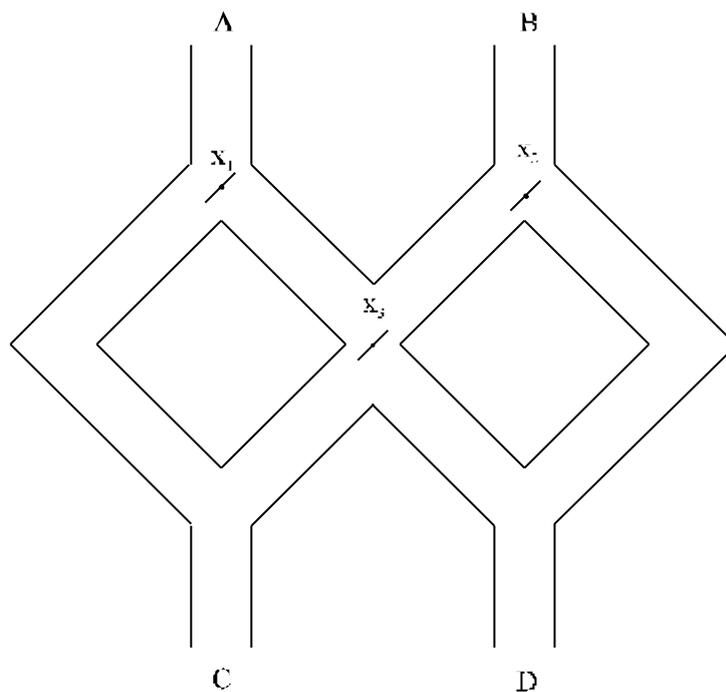
```
GREP -i function.*).*real *.PAS
```

findet in den PAS-Dateien alle Textstellen, die mit *function* beginnen, worauf ein beliebiger String mit null oder mehr Zeichen ($.*$), eine schließende Klammer, ein weiterer String mit null oder mehr Zeichen und das Wort *real* folgen. Wegen der Option $-i$ (= ignore) kommt es auf Groß- und Kleinschreibung nicht an. Damit sucht GREP alle Funktionen, die eine Real-Wert zurückliefern.

Lesen Sie die Anleitung UTILS.DOC zu GREP und geben Sie Suchausdrücke an für:

- a) alle Funktionen,
 b) Prozeduren, die mindestens einen VAR-Parameter haben,
 c) Deklarationen von i als Integer-Variable.

10. Betrachten Sie das in der Abbildung dargestellte Spielzeug. Eine Murmel wird in A oder B fallen gelassen. Die Hebel x_1 , x_2 und x_3 lassen die Murmel entweder nach rechts oder nach links fallen. Immer wenn eine Murmel auf einen Hebel trifft, veranlaßt sie den Hebel, seinen Zustand zu wechseln, so daß die nächste Murmel, die auf den Hebel trifft, den entgegengesetzten Weg nehmen wird.
- Modellieren Sie dieses Spielzeug durch einen endlichen Automaten. Sie brauchen dazu acht Zustände, wobei jeder Zustand eine Stellungskombination der drei Hebel entspricht. Bezeichnen Sie eine Murmel in A durch eine a-Eingabe und eine Murmel in B durch eine b-Eingabe.
 - Eine Eingabefolge wird akzeptiert, wenn die letzte Murmel bei D herauskommt. Beschreiben Sie die von diesem endlichen Automaten akzeptierte Sprache.
 - Modellieren Sie das Spielzeug, als einen übersetzenden Automaten, dessen Ausgabe die Folge der Cs und Ds ist, aus denen die Murmeln hintereinander herausfallen.



18 Kellerautomaten

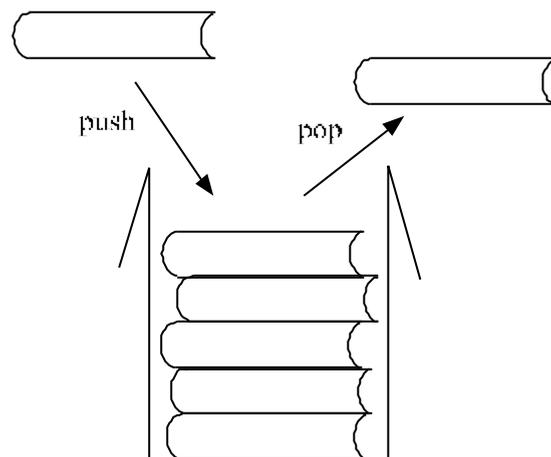
18.1 Konzeption des Kellerautomaten

Im letzten Kapitel haben wir die Grenzen endlicher Automaten am Beispiel der Sprache $L = \{a^n b^n \mid n \in \mathbb{N}\}$ kennengelernt. Die Ursache bestand in der begrenzten Speicherfähigkeit aufgrund der endlichen Anzahl von Zuständen.

Lassen wir unendlich viele Zustände zu, arbeiten wir also mit unendlichen Automaten, so kann die Sprache L erkannt werden. In Theoriebüchern findet man nichts über solche unendlichen Automaten. Ein Grund dürfte darin bestehen, daß unendliche Automaten unpraktisch sind. Es ist sinnvoller, die Speichereinheit von der Steuereinheit des Automaten zu trennen. Die Steuereinheit kann dann endlich ausgeführt werden, während sich das Unendliche lediglich auf den Speicher bezieht.

Damit der Automat nicht zu kompliziert wird, muß man sich Gedanken über die Ausführung des Speichers machen. Ein Ansatz über wahlfreien Zugriff mittels Adressen wäre denkbar, da wir aber die Leistungsfähigkeit von Automaten ausloten wollen, sind wir an einfacheren Speicherstrukturen interessiert. Einfacher ist es gewiß, wenn man statt wahlfreiem Zugriff nur festen Zugriff auf den Speicher erlaubt. Das kann natürlich nicht eine feste Speicherstelle sein, da wir einen unendlichen Speicher zulassen müssen. Aber es kann der feste Zugriff auf das zuletzt gespeicherte Zeichen sein. Dies ist die primitivste Form einer unendlichen Speicherstruktur. Sie wird *Kellerspeicher* genannt.

Anschaulich kann man sich einen Keller als eine Bücherkiste vorstellen, in die man ein Buch legen oder auch herausnehmen kann.



unendlichen
Automaten

Trennung von Spei-
cher- und Steuer-
einheit

Kellerspeicher

Abb. 18-1
Prinzip des Keller-
speichers

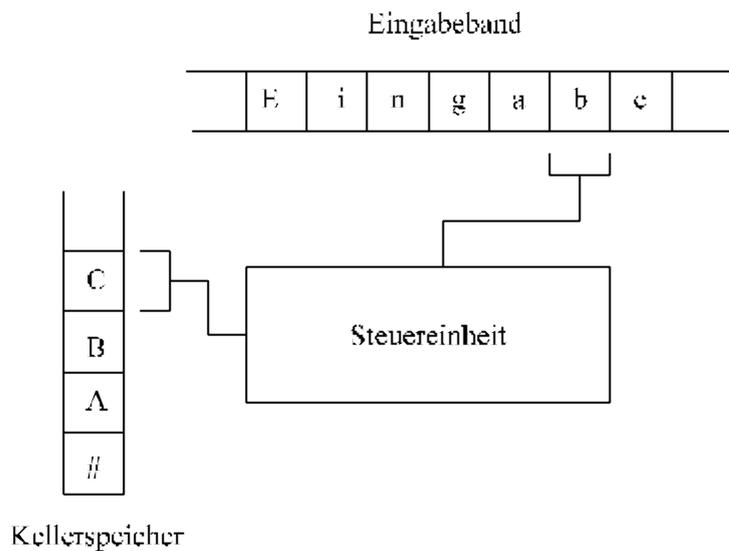
Das Prinzip eines Kellers besteht darin, daß stets das zuletzt eingefügte Element eines Kellers als erstes wieder entfernt werden muß. (LIFO-Prinzip,

engl. Last In First Out). Zur Verwaltung eines Kellerspeichers braucht man folgende Speicheroperationen:

- Kelleroperationen
- push: Legt ein Element auf dem Keller ab.
 - pop: Holt ein Element aus dem Keller.
 - top: Schaut nach, welches Element im Keller obenauf liegt.
 - init: Initialisiert den Keller.

Im Modell stellt sich nun ein Kellerautomat wie in der Skizze dar. Er hat fast alles, was ein Computer auch hat: eine Eingabeschnittstelle, einen Speicher und eine Steuereinheit.

Abb. 18-2
Modell eines
Kellerautomaten



Arbeitsweise eines
Kellerautomaten

Die Steuereinheit des Kellerautomaten wird zu Beginn in den Anfangszustand versetzt und die Speichereinheit mit dem Zeichen # initialisiert. In jedem Arbeitsschritt liest der Kellerautomat ein Eingabezeichen und das aktuelle Speicherzeichen (top). In Abhängigkeit von den drei Größen aktueller Zustand, Eingabezeichen und Speicherzeichen macht der Kellerautomat einen Zustandsübergang und zusätzlich immer ein Speicheroperation. Er kann ein Zeichen auf seinem Keller speichern, ein Zeichen vom Keller löschen oder den Keller unverändert lassen. Dies sind die Operationen *push*(Zeichen), *pop* und *nop* (no Operation).

18.1.1 Die Sprache $L = \{a^n b^n \mid n \in \mathbb{N}\}$

Zur Konkretisierung illustrieren wir die Arbeitsweise am Beispiel eines Kellerautomaten für die Sprache $L = \{a^n b^n \mid n \in \mathbb{N}\}$. Zu Beginn wird der Kellerautomat in den Anfangszustand z_0 versetzt. Wenn er die Eingabe a und das Kellersymbol $\#$ liest, so speichert er a und verbleibt im Zustand z_0 . Er macht das Gleiche, wenn er die Eingabe a und das Kellerzeichen a liest. Liest er stattdessen im Zustand z_0 ein b , so wird seine Reaktion vom Kellerzeichen bestimmt: Bei leerem Keller - Kellerzeichen $\#$ - macht er einen Übergang in den Fehlerzustand z_f , anderenfalls wechselt er in den Zustand z_1 und löscht ein Zeichen vom Keller.

Arbeitsweise eines Kellerautomaten für die Sprache $a^n b^n$

Im Zustand z_1 hat er also mindestens ein b gelesen, weswegen weitere a unabhängig vom Kellerzeichen in den Fehlerzustand z_f führen. Liest er stattdessen ein b und als Kellerzeichen ein a , so wird gelöscht er das Kellerzeichen und verbleibt im Zustand z_1 . Auf diese Weise werden die zuvor gespeicherten a durch eine entsprechende Anzahl von b neutralisiert.

Hatte die Eingabe zu viele b , dann liest der Kellerautomat irgendwann ein b zusammen mit dem Kellerzeichen $\#$. Dies führt in den Fehlerzustand. Hat hingegen die Eingabe zu wenig b , dann ist nach Abarbeitung der Eingabe der Keller nicht leer. Wörter der Sprache L werden also genau dann akzeptiert, wenn sich der Kellerautomat im Endzustand z_2 befindet und der Kellerspeicher leer ist.

Übersichtlich kann man die Schaltung der Steuereinheit in einer Tabelle darstellen, die zu jeder Kombination aus Zustand, Eingabezeichen und Kellerzeichen den Folgezustand und die Kelleroperation angibt. Da wir drei Zustände, zwei Eingabezeichen und zwei Kellerzeichen haben, hat diese Tabelle 12 Einträge.

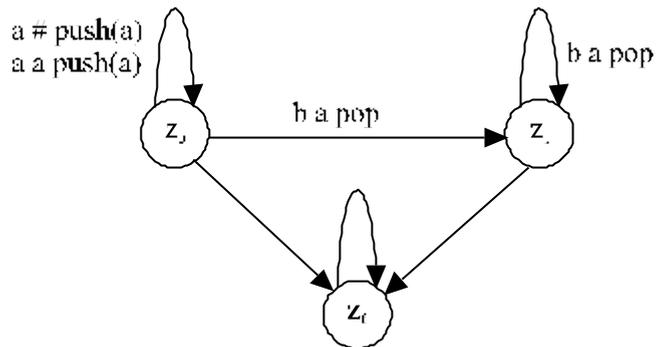
Zustand	Eingabe	Kellerzeichen	neuer Zustand	Operation
z_0	a	$\#$	z_0	push(a)
z_0	a	a	z_0	push(a)
z_0	b	$\#$	z_f	nop
z_0	b	a	z_1	pop
z_1	a	$\#$	z_f	nop
z_1	a	a	z_f	nop
z_1	b	$\#$	z_f	nop
z_1	b	a	z_1	pop
z_f	a	$\#$	z_f	nop
z_f	a	a	z_f	nop
z_f	b	$\#$	z_f	nop

Tabelle 18-1
Kellerautomat für die Sprache $a^n b^n$

z_f	b	a	z_f	nop
-------	---	---	-------	-----

Man kann analog zu den Zustandsgraphen bei Automaten auch einen Zustandsgraphen für Kellerautomaten angeben. Dabei beschränken wir uns auf die vier Zustandsübergänge, die nicht in den Fehlerzustand führen. Die Zustandsübergänge in den Fehlerzustand geben wir wegen der Übersichtlichkeit nur verkürzt an.

Abb. 18-3
Zustandsgraph
eines Kellerauto-
maten für die Spra-
che $a^n b^n$



Blättert man in Theoriebüchern, so findet man nur selten Zustandsgraphen für Kellerautomaten. Der Grund ist einfach: Die Funktionsweise des Kellerautomaten wird im wesentlichen durch den zielgerichteten Einsatz seines externen Kellerspeichers bestimmt. Der interne Speicher in Form der verbleibenden Zustände hat nur eine untergeordnete Bedeutung. Es werden nur wenige Zustände benötigt, weil die wesentlichen Speicheroperationen auf dem Kellerspeicher stattfinden.

Zustandsgraphen
von Automaten und
Kellerauto-
maten

Der Blick auf den Zustandsgraphen eines Kellerautomaten ist weniger aussagekräftig als bei Automaten, weil das Verhalten des Kellerautomaten hinsichtlich seines Speichers durch den Zustandsgraphen nicht dargestellt wird.

Automaten benutzen Zustände als endlichen Speicher. Je leistungsfähiger endliche Automaten sein sollen, um so mehr Zustände brauchen sie. Ein Zustandsgraph veranschaulicht daher gut den Aufbau der Steuereinheit eines Automaten.

18.1.2 Arithmetische Ausdrücke

Als zweites Beispiel betrachten wir die syntaktische Analyse von arithmetischen Ausdrücken, welche aus Zahlen, Klammern und den Operatoren + und - aufgebaut sind. Präzisieren läßt sich dies durch Syntaxdiagramme, wobei das Nichtterminal *Zahl* für eine Folge von Ziffern steht.

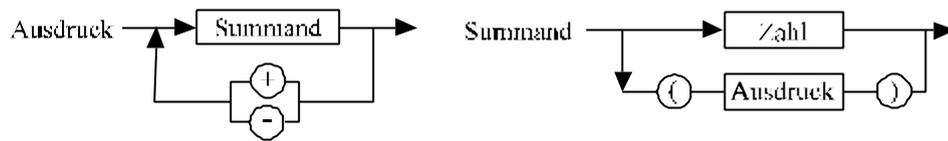


Abb. 18-4
Syntaxdiagramme für arithmetische Ausdrücke

Die Konstruktion eines Kellerautomaten ist längst nicht so einfach, wie bei den Automatenbeispielen. Die Einfachheit der folgenden Lösung täuscht über den Aufwand hinweg, sie zu finden. Eine Lösungsmöglichkeit besteht darin, zunächst die Wiederholung von Summanden wegzulassen. Die zulässigen Ausdrücke sind dann von der Form: *Zahl*, (*Zahl*), ((*Zahl*)), (((*Zahl*))) und so weiter. Daraus läßt sich der folgende Zustandsgraph ableiten:

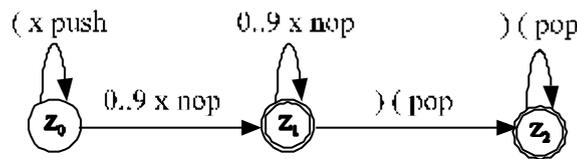


Abb. 18-5
Zustandsgraph eines Kellerautomaten für geklammerte Zahlen

Abkürzend steht hier *x* für ein beliebiges Eingabezeichen und 0..9 für eine Ziffer. Übergänge in den Fehlerzustand sind generell nicht dargestellt. Der Endzustand *z1* kennzeichnet reine Zahlen, der Endzustand *z2* einen Klammersausdruck. An die beiden Endzustände können sich Operatoren anschließen, welche die Wiederholung eines Summanden zulassen. Dazu wird der Zustandsgraph um Übergänge erweitert:

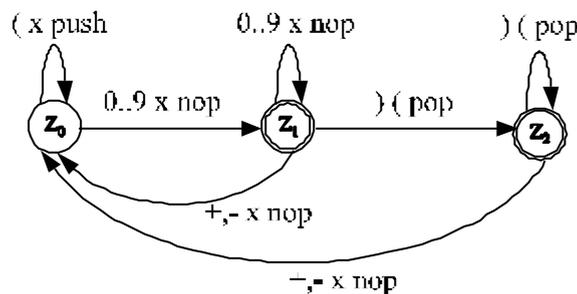


Abb. 18-6
Zustandsgraph eines Kellerautomaten für arithmetische Ausdrücke

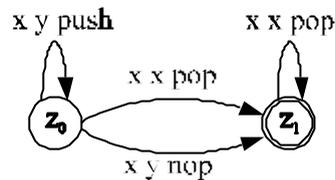
18.1.3 Palindrome

nicht-deterministischer Kellerautomat für Palindrome

Ein *Palindrom* ist eine nicht-leere Zeichenkette, die sich von vorne und hinten gleich liest, zum Beispiel *abba* oder *abaabaaba*. Ein Kellerautomat, der Palindrome erkennt, benötigt zwei Zustände. Im Zustand z_0 schreibt er gelesene Zeichen auf den Keller und im Zustand z_1 vergleicht er gelesene Zeichen mit Kellerzeichen. Wann vom Schreiben zum Lesen übergegangen wird, entscheidet er nichtdeterministisch.

Beim Übergang von z_0 nach z_1 sind zwei wesentlich verschiedene Fälle zu unterscheiden: Hat das Palindrom eine gerade Zahl von Zeichen, so muß zu jedem Zeichen der ersten Hälfte ein gleiches Zeichen an entsprechender Stelle in der zweiten Hälfte existieren. Bei ungeradzahligem Zeichenzahl gibt es für das mittlere Zeichen keinen Partner.

Abb. 18-7
Zustandsgraph eines Kellerautomaten für Palindrome



Ein deterministischer Kellerautomat ist nicht in der Lage, Palindrome zu erkennen. Im Gegensatz zur analogen Situation bei Automaten gibt es keine Möglichkeit, nichtdeterministische in deterministische Kellerautomaten umzuwandeln. Nichtdeterministische Kellerautomaten sind daher leistungsfähigere Automaten als deterministische Kellerautomaten. Die deterministischen Kellerautomaten spielen in der Praxis dennoch eine wichtige Rolle, weil ihre Zeitkomplexität linear ist.

18.2 Definition des Kellerautomaten

Ausgehend von den drei Beispielen und der Erkenntnis, daß eine Definition die anschließende Modellierung in Prolog unterstützt, definieren wir:

Ein deterministischer Kellerautomat besteht aus sieben Komponenten:

- Einer endlichen Menge von Zeichen, dem *Eingabealphabet*
- und einem *Kelleralphabet*
- mit einem ausgezeichneten Anfangszeichen #.
- Einer endlichen Menge von *Zuständen* mit
- einem ausgezeichneten *Anfangszustand* und
- einer Menge von *Endzuständen*.
- Einer *Übergangsfunktion*, die zu jedem Tripel aus Eingabezeichen, Kellerzeichen und Zustand den Folgezustand sowie eine Kelleroperation angibt.

Definition des
deterministischen
Kellerautomaten

Bei nichtdeterministischen Kellerautomaten werden die Übergänge durch eine Übergangsrelation beschrieben, das heißt, daß es zu jedem Tripel aus Eingabezeichen, Kellerzeichen und Zustand verschiedene Paare aus Folgezustand sowie Kelleroperation gibt.

In der Literatur gibt es diverse Definitionsvarianten für Kellerautomaten. Die Standarddefinition bezieht sich auf einen nichtdeterministischen Kellerautomaten, weil dieser genau die kontextfreien Sprachen erkennt. Oft wird mit allgemeineren Kelleroperationen gearbeitet. Dabei können ganze Zeichenfolgen auf den Keller gelegt werden. Solche Kellerautomaten benutzen wir im Kapitel 21 über *Maschinelle Sprachverarbeitung*. In obiger Definition lassen wir didaktisch reduziert nur einfache Kelleroperationen zu. Öfters wird auch auf die Angabe von Endzuständen verzichtet. Solche Kellerautomaten akzeptieren genau dann, wenn der Keller nach Abarbeitung der Eingabe leer ist. Man findet auch übersetzende Kellerautomaten.

Varianten von Kel-
lerautomaten

Die diversen Varianten dürften weitgehend äquivalent zueinander sein. Relevant ist wohl nur der Unterschied zwischen deterministischen und nichtdeterministischen Kellerautomaten.

18.3 Modellierung von Kellerautomaten in Prolog

Die Modellierung der Alphabete und Zustände erfolgt analog der Vorgehensweise bei Automaten. Das Kelleralphabet wird nicht gesondert aufgeführt, weil es in der Regel eine Teilmenge des Eingabealphabets vereinigt mit $\{\#\}$ ist.

Beispiel 1: Die Sprache $L = \{a^n b^n \mid n \in \mathbb{N}\}$.

Modellierung des
Kellerautomaten für
 $a^n b^n$

```
alphabet(X):- member(X, [a, b]).
anfangszustand(z0).
endzustand(z1).
```

Das Prädikat der Übergangsfunktion muß um zwei Parameter ergänzt werden: Oberstes Kellerelement und Kelleroperation:

Übergänge des
Kellerautomaten

```
/* uebergang(+Zustand, +Eingabe, +Kellerelement,
            -Kelleroperation, -FolgeZustand) */
uebergang(z0, a, #, push(a), z0).
uebergang(z0, a, a, push(a), z0).
uebergang(z0, b, a, pop, z1).
uebergang(z1, b, a, pop, z1).
```

Der Kellerautomat wird von *akzeptiere* in den Anfangszustand versetzt und mit der Eingabeliste und dem initialisierten Kellerspeicher gestartet.

Initialisierung des
Kellerautomaten

```
/* akzeptiere(+Wort) */
akzeptiere(Wort):-
    anfangszustand(Zustand),
    zerlegen(Wort, Eingabeliste),
    kellerautomat(Zustand, Eingabeliste, [#]).
```

Anstelle des Prädikats *automat* haben wir nun das analoge Prädikat *kellerautomat*, welches als weiteren Parameter den aktuellen Keller mitführt.

Implementierung
der Arbeitsweise
eines Kellerauto-
maten

```
/* kellerautomat(+Zustand, +Eingabeliste, +Keller) */
kellerautomat(Zustand, [Eingabe|Rest], [Top|Keller]):-
    uebergang(Zustand, Eingabe, Top, KellerOp, NeuerZustand),
    kellern(KellerOp, [Top|Keller], NeuerKeller),
    write(Zustand), tab(2), write([Eingabe|Rest]), tab(2),
    write([Top|Keller]), write(' -> '),
```

```
write(NeuerKeller), tab(2), write(NeuerZustand), nl,
kellerautomat(NeuerZustand, Rest, NeuerKeller).

kellerautomat(EndZustand, [], [#]):-
endzustand(EndZustand).
```

Im Unterschied zu bisherigen Lösung wird nach der Ermittlung eines Übergangs noch die zugehörige Kellerooperation auf dem Keller ausgeführt. Dazu gibt es das neue Prädikat *kellern*.

```
/*kellern(+Kellerooperation,+AlterKeller,-NeuerKeller)*/

kellern(push(Element), Keller, [Element|Keller]).
kellern(pop, [_Element|Keller], Keller).
kellern(nop, Keller, Keller).
```

Implementierung der Kellerooperationen

Beim Aufruf des Kellerautomaten mit *?- akzeptiere('aaabbb')* erhält man folgende Ausgabe:

```
z0 [a,a,a,b,b,b] [#] -> [a,#] z0
z0 [a,a,b,b,b] [a,#] -> [a,a,#] z0
z0 [a,b,b,b] [a,a,#] -> [a,a,a,#] z0
z0 [b,b,b] [a,a,a,#] -> [a,a,#] z1
z1 [b,b] [a,a,#] -> [a,#] z1
z1 [b] [a,#] -> [#] z1
```

der Kellerautomat beim Akzeptieren des Wortes aaabbb

Der erste Wert gibt den aktuellen Zustand, der zweite die noch zu verarbeitende Eingabe und der dritte Wert den aktuellen Keller an. Hinter dem Pfeil stehen der neue Keller und der neue Zustand.

Beispiel 2: Arithmetische Ausdrücke

Der Kellerautomat für die arithmetischen Ausdrücke wird durch folgende Übergänge beschrieben. In den Fällen, bei denen es auf das oberste Kellerelement nicht ankommt, wurde eine anonyme Variable benutzt:

```
uebergang(z0, '(', _, push('(', z0).
uebergang(z0, Zeichen, _, nop, z1):-
ziffer(Zeichen).

uebergang(z1, ')', '(', pop, z2).
uebergang(z1, Zeichen, _, nop, z1):-
ziffer(Zeichen).

uebergang(z1, '+', _, nop, z0).
uebergang(z1, '-', _, nop, z0).
```

Zustandsübergänge des Kellerautomaten für arithmetische Ausdrücke

```
uebergang(z2, ')', '(', pop, z2).
uebergang(z2, '+', _, nop, z0).
uebergang(z2, '-', _, nop, z0).
```

Beispiel 3: Palindrome

Bei den Übergängen des Kellerautomaten für Palindrome benutzt man neben anonymen Variablen auch normale Variablen. Vier Klauseln reichen dann aus:

Zustandsübergänge
beim Kellerautomaten
für Palindrome

```
uebergang(z0, X, _, push(X), z0).
uebergang(z0, _, _, nop, z1).
uebergang(z0, X, X, pop, z1).
uebergang(z1, X, X, pop, z1).
```

18.4 Erzeugte Sprache

Die von einem Kellerautomaten erkannte beziehungsweise erzeugte Sprache erhält man durch Modifikation des entsprechenden Prädikats für Automaten. Der Parameter *Keller* und die Operation *kellern* müssen ergänzt werden

erzeugte Sprache
eines Kellerauto-
maten

```
erzeugteSprache(Wort):-
    anfangszustand(ZustandA),
    mehrfachuebergang(ZustandA, Liste, [#], [#],ZustandZ),
    endzustand(ZustandZ),
    zusammensetzen(Liste, Wort).

einfachuebergang(Zustand,Eingabe,
    [KellerElement|AltKeller],
    NeuKeller, Neuzustand):-
    alphabet(Eingabe),
    uebergang(Zustand, Eingabe, KellerElement,
    KellerOperation, Neuzustand),
    kellern(KellerOperation, [KellerElement|AltKeller],
    NeuKeller).

mehrfachuebergang(Zustand,[Eingabe],AltKeller,
    NeuKeller, Neuzustand):-
    einfachuebergang(Zustand,Eingabe,AltKeller,
    NeuKeller, Neuzustand).

mehrfachuebergang(Zustand, [Kopf|Rest], AltKeller,
    NeuKeller, Neuzustand):-
    mehrfachuebergang(Zwischenzustand,Rest,ZwischenKeller,
    NeuKeller, Neuzustand),
```

```
einfachuebergang(Zustand, Kopf, AltKeller,  
                  ZwischenKeller, Zwischenzustand).
```

Ein überzeugendes Beispiel für die von Kellerautomaten erzeugte Sprache erhält man bei den arithmetischen Ausdrücken. Man sollte dabei das zugrundeliegende Alphabet einschränken, um nicht Zahlen, sondern im wesentlichen Strukturen aufzuzählen.

Kellerautomat für
arithmetische Aus-
drücke

```
alphabet(X):- zerlegen('1+()', Y), member(X, Y).
zustand(X) :- member(X, [z0, z1, z2]).
anfangszustand(z0).
endzustand(z1).
endzustand(z2).
```

Läßt man nur den +-Operator, Klammern und die Zahl 1 zu (Ziffernübergang im Zustand z1 entfernen), so ergibt sich diese Sprache:

die Sprache der
arithmetischen
Ausdrücke

```
1, 1+1, (1), 1+1+1, (1)+1, (1+1), 1+(1). ((1)), 1+1+1+1,
(1)+1+1, (1+1)+1, 1+(1)+1, ((1))+1, (1+1+1), 1+(1+1),...
```

18.5 Direktes Kellern

Wir haben die Kelleroperationen in einem eigenen Prädikat gefaßt und in drei Klausel implementiert. Dies ist eine übersichtliche Lösung. Als Alternative bietet sich an, die Kelleroperation direkt in die Übergangsklauseln einzubauen. Dies hat den kleinen Vorteil, daß der Aufruf von *kellern* unterbleiben kann, verbunden mit dem Nachteil, daß die *uebergang*-Klauseln schwieriger werden. Sinn macht diese Alternative, wenn bei einem Übergang mehrere Zeichen gekellert werden sollen. denn dies läßt sich vergleichsweise einfach bei der alternativen Modellierung realisieren. Beispiel 1 sieht mit eingebauten Kelleroperationen so aus:

```
/* uebergang(+Zustand, +Eingabe, +AlterKeller,
            -NeuerKeller, -FolgeZustand) */
                                                    Implementierung
                                                    des direkten Kel-
                                                    lerns
uebergang(z0, a, [#], [a, #], z0).
uebergang(z0, a, [a|K], [a, a|K], z0).
```

uebergang(z_0 , b , $[a|K]$, K , z_1).
uebergang(z_1 , b , $[a|K]$, K , z_1).

Bei den ersten beiden Klauseln findet eine Push-Operation statt, beiden letzten beiden eine Pop-Operation.

18.6 Spezialisieren durch Entfalten

Wir wenden die bei den Automaten eingeführte Technik des *Entfaltens*, mit der man von allgemeinen Modellierungen zu speziellen Modellierungen von Kellerautomaten kommt, auf das Beispiel der Palindrome an. Die Prädikate *alphabet* und *zustand* werden nicht benötigt, *anfangszustand* und *endzustand* nehmen wir direkt in die betroffenen Klauseln auf:

Initialisierung und
Terminierung des
Kellerautomaten für
Palindrome

```
akzeptiere(Wort):-
    zerlegen(Wort, Eingabeliste),
    kellerautomat(z0, Eingabeliste, [#]).

kellerautomat(z1, [], [#]).
```

Für direktes Kellern schreiben wir die Übergänge um

direktes Kellern

```
uebergang(z0, X, Keller, [X|Keller], z0).
uebergang(z0, _, Keller, Keller, z1).
uebergang(z0, X, [X|Keller], Keller, z1).
uebergang(z1, X, [X|Keller], Keller, z1).
```

und fassen die letzten beiden Klauseln noch zusammen:

```
uebergang(_, X, [X|Keller], Keller, z1).
```

Wir entfernen die Schreibanweisungen aus *kellerautomat* und benutzen direktes Kellern:

Kellerautomat für
Palindrome mit
direktem Kellern

```
kellerautomat(Zustand, [Eingabe|Rest], Keller):-
    uebergang(Zustand, Eingabe, Keller, NeuerKeller,
              NeuerZustand),
    kellerautomat(NeuerZustand, Rest, NeuerKeller).
```

Das Wesentliche des *Entfaltens* haben wir jetzt schon kennengelernt: Man setzt Klauseln in andere Klauseln ein, womit die eingesetzten Klauseln verschwinden, und vereinfacht dann die Klausel, in die eingesetzt wurde. Jetzt entfalten wir *uebergang* in *kellerautomat*. Dadurch entstehen drei *kellerautomat*-Klauseln und das *uebergang*-Prädikat verschwindet:

Entfalten von
uebergang im
Prädikat
kellerautomat

```
kellerautomat(z0, [Zeichen|Rest], Keller):-
    kellerautomat(z0, Rest, [Zeichen|Keller]).

kellerautomat(z0, [Zeichen|Rest], Keller):-
    kellerautomat(z1, Rest, Keller).
```

```
kellerautomat(_, [Zeichen|Rest], [Zeichen|Keller]):-
    kellerautomat(z1, Rest, Keller).
```

Abschließend nennen wir *kellerautomat* und *akzeptiere* in *palindrom* um. Damit erhalten wir einen speziellen Akzeptor für Palindrome:

```
palindrom(Wort):-
    zerlegen(Wort, Eingabeliste),
    palindrom(z0, Eingabeliste, [#]).
palindrom(z0, [Zeichen|Rest], Keller):-
    palindrom(z0, Rest, [Zeichen|Keller]).
palindrom(z0, [Zeichen|Rest], Keller):-
    palindrom(z1, Rest, Keller).
palindrom(_, [Zeichen|Rest], [Zeichen|Keller]):-
    palindrom(z1, Rest, Keller).
palindrom(z1, [], [#]).
```

entfalteter Kellerautomat für Palindrome

18.7 Interpretation arithmetischer Ausdrücke

Als Anwendung werden wir jetzt den Kellerautomaten für arithmetische Ausdrücke in einen Interpreter umbauen. Dies geschieht in zwei Schritten. Zunächst entfalten wir den allgemeinen Kellerautomaten zu einem speziellen Kellerautomaten, dann ergänzen wir die erzeugten Klauseln so, daß die Terme parallel zum Parsen auch interpretiert werden.

Spezialisieren durch Entfalten

Vor dem Entfalten fassen wir einige *uebergang*-Klauseln zusammen, damit hinterher nicht zu viele Klauseln entstehen.

```
uebergang(z0, '(', _, push('('), z0).
uebergang(z0, Zeichen, _, nop, z1):-
    ziffer(Zeichen).
uebergang(Zustand, ')', '(', pop, z2):-
    endzustand(Zustand).
uebergang(z1, Zeichen, _, nop, z1):-
    ziffer(Zeichen).
uebergang(Zustand, Op, _, nop, z0):-
    endzustand(Zustand),
    operator(Op).

endzustand(X):- member(X, [z1, z2]).
```

Zusammenfassung mehrerer uebergang-Klauseln

```
operator(X) :- member(X, ['+', '-']).
```

Entfaltet man den Kellerautomaten für arithmetische Ausdrücke und nennt das entstehende Prädikat *ausdruck* so erhält man:

entfalteter Kellerau-
tomat für
arithmetische Aus-
drücke

```
ausdruck(Wort):-
  zerlegen(Wort, Eingabeliste),
  ausdruck(z0, Eingabeliste, [#]).

ausdruck(Zustand, Eingabe, Keller):- /* Ausgabe */
  write(Zustand), tab(2), write(Eingabe), tab(2),
  write(Keller), nl, fail.

ausdruck(z0, ['(|Rest], Keller):-
  ausdruck(z0, Rest, ['(|Keller]).

ausdruck(z0, [Zeichen|Rest], Keller):-
  ziffer(Zeichen),
  ausdruck(z1, Rest, Keller).

ausdruck(z1, [Zeichen|Rest], Keller):-
  ziffer(Zeichen),
  ausdruck(z1, Rest, Keller).

ausdruck(Zustand, [Op|Rest], Keller):-
  endzustand(Zustand),
  operator(Op),
  ausdruck(z0, Rest, Keller).

ausdruck(Zustand, ['|Rest], ['(|Keller):-
  endzustand(Zustand),
  ausdruck(z2, Rest, Keller).

ausdruck(Zustand, [], [#):-
  endzustand(Zustand).
```

Um die Arbeitsweise des Kellerautomaten verfolgen zu können wurde eine Ausgabeklausel ergänzt, welche nach der Ausgabe mittels *fail* fehlschlägt und somit zur normalen Weiterverarbeitung übergeht.

Ergänzung der Interpretation

Interpretation
arithmetischer
Ausdrücke

Die Interpretation eines arithmetischen Ausdrucks wie zum Beispiel $2+((3-4)+5)$ soll als Ergebnis den Wert 6 des Ausdrucks liefern. Im Laufe der Berechnung können wie im Beispiel Zwischenergebnisse anfallen, welche auch gespeichert werden müssen. Wir ergänzen den Kellerautomaten deshalb um einen zweiten Kellerspeicher namens *Operanden*, auf dem er sich Zwischenergebnisse merken kann und nennen das neue Prädikat nun *interpretiere*. Den ursprüngliche Kellerspeicher *Keller* nennen wir in *Operatoren* um, da auf diesem Kellerspeicher außer öffnenden Klammern zum Interpretieren auch die Operatoren gespeichert werden. Die ersten drei *interpretiere*-Klauseln lauten dann:

Initialisierung der
Interpretation

```
/* Start */
interpretiere(Wort):-
    zerlegen(Wort, Eingabeliste),
    interpretiere(z0, Eingabeliste, [#], [#]).

/* Ausgabe */
interpretiere(Zustand, Eingabe, Operatoren, Operanden):-
    write(Zustand), tab(2), write(Eingabe), tab(2),
    write(Operatoren), tab(2), write(Operanden), nl, fail.

/* Öffnende Klammer */
interpretiere(z0, ['(|Rest], Operatoren, Operanden):-
    interpretiere(z0, Rest, ['(|Operatoren], Operanden).
```

Eine Teilaufgabe ist die Interpretation von Ziffernfolgen als Zahlen. Das erste gelesene Ziffernzeichen wird in eine Zahl konvertiert und als Zwischenergebnis auf dem Operandenkeller gespeichert. Bei nachfolgenden Ziffern wird das Zwischenergebnis vom Operandenkeller geholt, um die weitere Ziffer ergänzt und wieder auf dem Operandenkeller abgelegt:

Interpretation von
Ziffernfolgen

```
/* erste Ziffer */
interpretiere(z0,[Zeichen|Rest],Operatoren,Operanden):-
    ziffer(Zeichen),
    ord(Zeichen, Ziffer),
    Zahl is Ziffer - 48,
    interpretiere(z1, Rest, Operatoren, [Zahl|Operanden]).

/* weitere Ziffern */
interpretiere(z1,[Zeichen|Rest],Operatoren,
    [Zahl1|Operanden]):-
    ziffer(Zeichen),
    ord(Zeichen, Ziffer),
```

```
Zahl2 is Zahl1*10 + Ziffer - 48,  
interpretiere(z1,Rest,Operatoren,[Zahl2|Operanden]).
```

Eine weitere Teilaufgabe besteht in der Interpretation von Summen wie zum Beispiel $16+3-4-11-2$. Zunächst wird 16 gelesen und auf den Operandenkeller gelegt, dann wird der Operator + auf dem Operatorkeller gespeichert, denn vor der Addition muß der zweite Summand bestimmt werden. Als nächstes wird die 3 gelesen und auf dem Operandenkeller gespeichert. Beim nachfolgenden + muß die erste Zwischenrechnung ausgeführt werden: der Operator und die beiden Operanden werden von den Kellern geholt, die Rechnung ausgeführt und das Ergebnis 19 auf dem Operandenkeller gespeichert.

Nach der Zwischenrechnung wird der Operator - und der Operand 4 gekellert. Beim Lesen des nächsten Operators wird klar, daß wiederum eine Zwischenrechnung mit dem Ergebnis $19-4=15$ ausgeführt werden kann. Nach zwei weiteren Zwischenrechnungen steht das Endergebnis auf dem Operandenkeller.

Wie das Beispiel zeigt, sind zwei Fälle beim Lesen eines Operators zu unterscheiden. Befindet sich kein Operator auf dem Operatorkeller, so kann der gelesene Operator sofort gekellert werden, anderenfalls muß zunächst eine Zwischenrechnung ausgeführt werden.

Interpretation von
Summen

```
/* erster Operator */
interpretiere(Zustand, [Op1|Rest], [Op2|Operatoren],
              Operanden):-
    endzustand(Zustand),
    operator(Op1),
    not operator(Op2),
    interpretiere(z0,Rest, [Op1,Op2|Operatoren],Operanden).

/* weiterer Operator, Zwischenrechnung ausführen */
interpretiere(Zustand, [Op1|Rest], [Op2|Operatoren],
              Operanden):-
    endzustand(Zustand),
    operator(Op1),
    berechne(Op2, Operanden, NeuOperanden),
    interpretiere(z0,Rest, [Op1|Operatoren],NeuOperanden).

/* hier wird gerechnet */
berechne(Op,[Zahl2, Zahl1|Operanden],
         [Zahl3|Operanden]):-
    operator(Op),
    (Op = '+', Zahl3 is Zahl1 + Zahl2;
     Op = '-', Zahl3 is Zahl1 - Zahl2).
```

Ist die Eingabe $16+3-4-11-2$ abgearbeitet, so befindet sich noch der letzte Operator auf dem Keller. Kam hingegen in der Eingabe kein Operator vor, so steht das Ergebnis schon im Operandenkeller. Für das Ende der Eingabe müssen wir demnach zwei Fälle unterscheiden:

Endergebnis
bestimmen

```
/* Endergebnis berechnen */
interpretiere(Zustand, [], [Op|Rest], Operanden):-
    endzustand(Zustand),
    berechne(Op, Operanden, NeuOperanden),
    interpretiere(Zustand, [], Rest, NeuOperanden).

/* Endergebnis ausgeben */
interpretiere(Zustand, [], [#], [Ergebnis, #]):-
```

```
endzustand(Zustand),
write(Ergebnis).
```

Zuletzt müssen wir die schließende Klammer interpretieren. Sie bedeutet in aller Regel den Abschluß einer Zwischenrechnung. Es könnte allerdings auch eine einfache Zahl oder ein geklammerter Term in der Klammer gestanden haben.

```
/* Klammer zu mit Summe berechnen */
interpretiere(Zustand, [')'|Rest], [Op|Operatoren],
              Operanden):-
    endzustand(Zustand),
    berechne(Op, Operanden, NeuOperanden),
    interpretiere(z2,[')'|Rest],Operatoren,NeuOperanden).

/* Klammer zu */
interpretiere(Zustand,[')'|Rest], ['('|Operatoren],
              Operanden):-
    endzustand(Zustand),
    interpretiere(z2, Rest, Operatoren, Operanden).
```

Interpretation von
Klammern

Im folgenden Beispiel einer Interpretation werden pro Zeile ausgegeben: der Zustand, die Eingabe, der Operatorenkeller und der Operandenkeller.

```
?- interpretiere('2+(3-4-(4-5))+1').
z0 [2,+, (,3,-,4,-, (,4,-,5,)),),+,1] [#] [#]
z1 [+, (,3,-,4,-, (,4,-,5,)),),+,1] [#] [2,#]
z0 [(,3,-,4,-, (,4,-,5,)),),+,1] [+,#] [2,#]
z0 [3,-,4,-, (,4,-,5,)),),+,1] [(,+,#] [2,#]
z1 [-,4,-, (,4,-,5,)),),+,1] [(,+,#] [3,2,#]
z0 [4,-, (,4,-,5,)),),+,1] [-,(+,#] [3,2,#]
z1 [-,(,4,-,5,)),),+,1] [-,(+,#] [4,3,2,#]
z0 [(,4,-,5,)),),+,1] [-,(+,#] [-1,2,#]
z0 [4,-,5,)),),+,1] [(,-,(+,#] [-1,2,#]
z1 [-,5,)),),+,1] [(,-,(+,#] [4,-1,2,#]
z0 [5,)),),+,1] [-,(,-,(+,#] [4,-1,2,#]
z1 [),),+,1] [-,(,-,(+,#] [5,4,-1,2,#]
z2 [),),+,1] [(,-,(+,#] [-1,-1,2,#]
z2 [),+,1] [-,(+,#] [-1,-1,2,#]
z2 [),+,1] [(,+,#] [0,2,#]
z2 [+ ,1] [+,#] [0,2,#]
z0 [1] [+,#] [2,#]
z1 [] [+,#] [1,2,#]
z1 [] [#] [3,#]
3
```

Interpretation
des Ausdrucks
 $2+(3-4-(4-5))+1$

Die Funktionstüchtigkeit des Interpreters läßt sich am einfachsten mit dem Generator prüfen:

?- erzeugteSprache(X), interpretiere(X).

18.8 Die Grenzen von Kellerautomaten

Die formale Sprache $L_1 = \{a^n \mid n \in \mathbb{N}\}$ ist regulär und wird von einem Automaten erkannt. Automaten können die formale Sprache $L_2 = \{a^n b^n \mid n \in \mathbb{N}\}$ nicht erkennen. Dazu benötigt man einen Kellerautomaten. Nehmen wir einen dritten Faktor c hinzu, so erhalten wir die formale Sprache $L_3 = \{a^n b^n c^n \mid n \in \mathbb{N}\}$. In Kapitel 16 haben wir eine kontextsensitive Grammatik kennengelernt, die L_3 erzeugt. Können Kellerautomaten L_3 erkennen? Die Antwort lautet *Nein!* Wir begnügen uns mit einem inhaltlichen Argument. Nachdem der Kellerautomat $a^n b^n$ abgearbeitet hat, ist der Keller leer, er kann daher nicht mehr entscheiden, ob genau n -mal das Zeichen c folgt.

Grenzen von Kellerautomaten durch beschränkten Speicherzugriff

Man benötigt ein leistungsfähigeres Maschinenmodell, um L_3 zu erkennen. Kellerautomaten können L_3 offenbar deswegen nicht erkennen, weil Sie einer Beschränkung bezüglich des Speicherzugriffs unterliegen. Kellerautomaten können immer nur auf das oberste Element im Speicher zugreifen. Läßt man diese Beschränkung fallen, erlaubt man also den Zugriff auf alle Elemente im Speicher, so erhält man leistungsfähigere Maschinen, die sogenannten *Turingmaschinen*.

Turingmaschinen

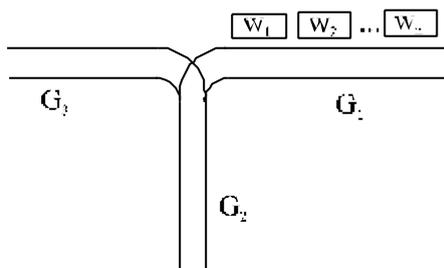
Wir halten die Ergebnisse in einer Tabelle fest, welche zu den verschiedenen Grammatiktypen das jeweilige Maschinenmodell angibt, das zum Erkennen der zugehörigen Sprache benötigt wird:

Tabelle 18-2 Grammatiken und zugehörige Maschinenmodelle

Grammatik	Maschinenmodell
regulär	Automat
kontextfrei	nichtdeterministischer Kellerautomat
kontextsensitiv	Turingmaschine

18.9 Aufgaben

- 1a) Entwerfen Sie einen Kellerautomaten, der Wörter der Form ww^{-1} über dem Alphabet $\{0, 1\}$ akzeptiert, wobei w^{-1} das gespiegelte Wort w ist. Beispiele: 0110, 0110000110.
 - b) Modellieren Sie diesen Automaten in Prolog
 - c) Entwerfen Sie einen zweiten Kellerautomaten, der Wörter der Form wcw^{-1} erkennt, wobei das Zeichen c das Teilwort w von seinem Spiegelbild w^{-1} trennt.
 - d) Vergleichen Sie die beiden Kellerautomaten.
2. Die Grammatik $S \rightarrow (x) \mid (S) \mid SS$ beschreibt *wohlgeformte Klammerausdrücke*.
 - a) Wieso ist diese Grammatik nicht regulär?
 - b) Skizzieren Sie den Zustandsgraphen eines erkennenden deterministischen Kellerautomaten. Hilfe: Beginnen Sie mit $S \rightarrow (x)$ und erweitern Sie den Zustandsgraphen für die beiden anderen Produktionsregeln.
 - c) Modellieren Sie diesen Kellerautomaten in Prolog:
 - 3a) Entwerfen Sie einen Kellerautomaten, der Wörter aus a und b akzeptiert, die gleich viele a und b enthalten.
 - b) Kann der Kellerautomat deterministisch gewählt werden?
 4. Ein Zug aus n unterscheidbaren Wagen w_i ist, wie angedeutet, auf einem Gleis G_1 angeordnet.



Die Gleisanlage ist mit einem Automaten verbunden, der auf unterschiedliche Eingaben folgende Aktionen ausführt.

Eingabe A: Falls G_1 nicht leer: rangiere den ersten Wagen von G_1 nach G_2 , sonst Fehler.

Eingabe B: Falls G_2 nicht leer: rangiere den ersten Wagen von

G_2 nach G_3 , sonst Fehler.

Die Sprache L sei die Menge aller Eingabefolgen über $\{A, B\}$, die zu keinem Fehlerzustand führen und die einen Zug der Länge n vollständig nach G_3 überführen.

Beschreiben Sie L und bilden Sie einen Kellerautomaten, der L akzeptiert.

5. Gegeben ist die Grammatik $S \rightarrow a \mid b \mid (S + S)$. Entwickeln Sie einen Kellerautomaten, der genau die Sätze dieser Sprache akzeptiert.

Hinweis: Es ist nach dem bisherigen Verfahren sehr schwierig, einen solchen Kellerautomaten zu konstruieren. Wählen Sie daher ein besseres Verfahren, daß bei beliebigen kontextfreien Grammatiken zum Ziel führt.

Modifizieren Sie den Kellerautomaten so, daß zunächst das Startsymbol S auf den Keller gelegt wird. Befindet sich während der Abarbeitung S oben im Keller, so wird S durch die rechte Seite einer Produktionsregel im Keller ersetzt. Die Position des Lesekopfes auf dem Eingabeband bleibt unverändert. Ist das oberste Kellersymbol dagegen ein Terminal, so wird es mit dem aktuellen Zeichen unter dem Lesekopf verglichen. Falls beide Zeichen übereinstimmen, geht der Lesekopf auf dem Eingabeband um eine Position nach rechts, und das Zeichen wird vom Keller gelöscht. Da dieser Kellerautomat mit einem Zustand auskommt, kann auf die explizite Zustandsverwaltung verzichtet werden.

19 Turingmaschinen

19.1 Konzeption der Turingmaschine

Wir haben die Grenzen von endlichen Automaten und von Kellerautomaten kennengelernt. Also suchen wir leistungsfähigere Maschinenmodelle, mit denen die bisherigen Grenzen überschritten werden können.

Die abschließende Diskussion im Kapitel über Kellerautomaten macht deutlich, daß der restriktive Speicherzugriff die Leistungsfähigkeit von Kellerautomaten begrenzt. Für eine neue Maschine brauchen wir eine bessere Speicherkonzeption, die dennoch möglichst einfach ist, um das Studium der Möglichkeiten und Grenzen der neuen Maschine nicht unnötig kompliziert zu machen.

Orientieren wir uns am Versagen der Kellerautomaten beim Erkennen der Sprache $L_3 = \{a^n b^n c^n \mid n \in \mathbb{N}\}$. Ein Kellerautomat kann zwar a^n speichern, aber wenn der b^n mit a^n vergleicht, also den Speicherinhalt wieder liest, so verliert er die gespeicherte Information. Er benutzt destruktives Lesen des Speichers. Zum Erkennen von L_3 benötigt man *nichtdestruktives* Lesen. Die Beweglichkeit des Schreib-/Lesekopfes muß nicht erweitert werden: wie bei *push*, *nop* und *pop* reicht es aus, den Schreib-/Lesekopf in beide Richtungen bewegen zu können. Im Grunde reicht ein einseitig unbegrenzter Speicher aus. Mit einem zweiseitig unbegrenzten Speicher haben wir es aber einfacher. Die beim Kellerautomaten notwendige Trennung zwischen Speichereinheit und Eingabeband kann dadurch aufgehoben werden, daß man die Eingabe direkt in den Speicher schreibt.

nichtdestruktives
Lesen des Spei-
chers

Wir kommen so zu folgendem Modell der Turingmaschine, das 1936 von dem Mathematiker Alan Turing (1912-1954) vorgestellt wurde. Eine Turingmaschine besteht aus drei Einheiten: dem beidseitig unendlichen *Arbeitsband*, der *Steuereinheit* und dem *Schreib-/Lesekopf*.

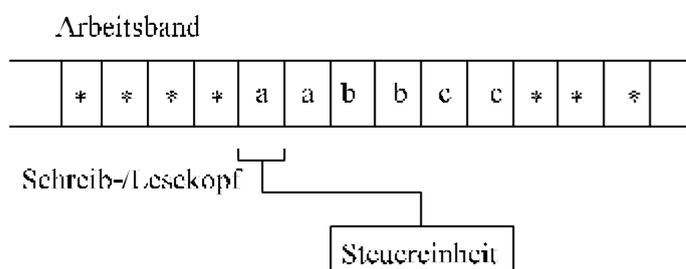


Abb. 19-1
Konzeption einer
Turingmaschine

Das *Arbeitsband* ist in einzelne Zellen unterteilt. Jede Zelle kann genau ein Zeichen speichern. Das *Leerzeichen* „*“ kennzeichnet eine leere Zelle.

Arbeitsband

Schreib-/Lesekopf	Der <i>Schreib-/Lesekopf</i> kann den Inhalt einer Zelle lesen und sie neu beschreiben. Im Arbeitsrythmus einer Turingmaschine ist festgelegt, daß der Schreib-/Lesekopf abwechselnd liest und schreibt. Der Schreib-/Lesekopf kann schrittweise um eine Zelle nach links oder rechts bewegt werden, aber auch auf einer Zelle stehen bleiben.
Steuereinheit	Die <i>Steuereinheit</i> ist ein Automat, der endlich viele verschiedene Zustände annehmen kann. Zwei Zustände haben eine besondere Bedeutung:
Anfangszustand	A ist der Anfangszustand, in dem die Turingmaschine ihre Arbeit beginnt
Endzustand	E ist der Endzustand, bei dessen Erreichen die Turingmaschine ihre Arbeit beendet.

Sonstige Zustände numeriert man gewöhnlich durch. Die Arbeit der Turingmaschine wird durch eine Zustandübergangstabelle gesteuert. Diese Tabelle legt fest, was die Turingmaschine tun soll. Die auszuführende Aktion ist vom gegenwärtigen Zustand und vom Zeichen unter dem Schreib-/Lesekopf abhängig. Zustand und Zeichen bestimmen also den nächsten Befehl. Jeder Befehl ist durch drei Bestandteile gekennzeichnet:

- | | |
|-----------------------------|---|
| Bestandteile eines Befehles | <ol style="list-style-type: none"> 1. Das Zeichen, das in die aktuelle Zelle geschrieben werden soll. 2. Die danach auszuführende Kopfbewegung:
 R eine Zelle nach rechts,
 K keine Kopfbewegung,
 L eine Zelle nach links. 3. Der Zustand, in den gewechselt werden soll. |
|-----------------------------|---|

Jeder Befehl hat somit die Struktur:

Befehlsstruktur einer Turingmaschine	Zustand Zeichen → Zeichen Kopfbewegung Zustand
--------------------------------------	--

Die *Arbeitsweise* einer Turingmaschine ist wie folgt festgelegt: Nachdem man die Eingabe auf das Band eingetragen hat, wird der Schreib-/Lesekopf auf das am weitesten links stehende Eingabezeichen gestellt. Sodann wird die Maschine in den Anfangszustand z_a gebracht und losgelassen. Sie arbeitet entsprechend ihrer Zustandübergangstabelle, bis sie einen Endzustand erreicht hat. Das folgende Struktogramm beschreibt die Arbeitsweise einer Turingmaschine:

Schreib-/Lesekopf auf erstes Eingabezeichen einstellen	
Anfangszustand einnehmen	
	Eingabezeichen lesen
	Ausgabezeichen schreiben
	Kopfbewegung durchführen
	neuen Zustand einnehmen
bis Endzustand erreicht ist	

Abb. 19-2
Struktogramm zur
Arbeitsweise einer
Turingmaschine

19.1.1 Akzeptor für die Sprache $L_2 = \{a^n b^n \mid n \in \mathbb{N}\}$

Eine die Sprache L_2 akzeptierende Turingmaschine kann so arbeiten, daß Sie paarweise ein a und ein b auf dem Arbeitsband durch Leerzeichen löscht. Ist das Band schließlich leer, so wird das Wort akzeptiert.

Liest die Turingmaschine im Anfangszustand ein Leerzeichen, so handelt es sich um das leere Wort, das sofort akzeptiert wird:

$A * \rightarrow * K E$

eine Turingmaschine für die Sprache $a^n b^n$

Anderenfalls muß ein a gelesen und gelöscht werden, wobei in den Zustand 1 gewechselt wird:

$A a \rightarrow * R 1$

Im Zustand 1 geht es an das Ende der Eingabe:

$1 a \rightarrow a R 1 \quad 1 b \rightarrow b R 1$

Ist das Ende der Eingabe erreicht, wechselt die Turingmaschine in den Zustand 2, in dem sie sich merkt, daß ein b gelöscht werden muß:

$1 * \rightarrow * L 2$

Im Zustand 2 löscht die Turingmaschine ein b, wechselt in den Zustand 3 und geht zum Anfang der Eingabe zurück:

$2 b \rightarrow * L 3 \quad 3 b \rightarrow b L 3 \quad 3 a \rightarrow a L 3$

Ist der Anfang der Eingabe erreicht, wechselt die Turingmaschine wieder in den Anfangszustand:

$3 * \rightarrow * R A$

19.1.2 Turingmaschine zum Addieren

Unär-Darstellung von Zahlen Bei geeigneter Interpretation der Bandbelegung können mit Turingmaschinen Berechnungen ausgeführt werden. Zahlen werden grundsätzlich in Unär-Darstellung als Strichliste auf das Arbeitsband geschrieben, wobei mehrere Zahlen durch das Nummernzeichen # voneinander zu trennen sind.

Endergebnis einer Berechnung Erreicht die Turingmaschine den Endzustand, so steht das Ergebnis ab der Position des Schreib-/Lesekopfes auf dem Arbeitsband. Es besteht aus allen Strichen, bis zum ersten von „|“ verschiedenen Zeichen.

Betrachten wir als Beispiel eine Turingmaschine, die addieren kann. Wir orientieren uns an der Summe 3+5. Als Eingabe ist |||#||| auf das leere Arbeitsband zu schreiben. Zur Berechnung der Summe verschiebt die Turingmaschine den ersten Strich | an die Position des Nummernzeichens und geht an den Anfang zurück.

eine Turingmaschine zum Addieren Im Anfangszustand löscht die Turingmaschine einen Strich und wechselt in den Zustand 1:

$$A \quad | \quad \rightarrow \quad * \quad R \quad 1$$

Im Zustand 1 geht es nach rechts, bis das Nummernzeichen erreicht ist:

$$1 \quad | \quad \rightarrow \quad | \quad R \quad 1$$

Das Nummernzeichen wird durch einen Strich ersetzt. Im Zustand 2 geht es an den Anfang zurück:

$$1 \quad \# \quad \rightarrow \quad | \quad L \quad 2$$

$$2 \quad | \quad \rightarrow \quad | \quad L \quad 2$$

Den Schreib-/Lesekopf positioniert die Turingmaschine auf den ersten Strich:

$$2 \quad * \quad \rightarrow \quad * \quad R \quad E$$

Der Sonderfall, daß der erste Summand 0 ist, muß berücksichtigt werden:

$$A \quad \# \quad \rightarrow \quad * \quad R \quad E$$

19.1.3 Algorithmische Grundstrukturen

An einfachen Beispielen soll verdeutlicht werden, wie mit Turingmaschinen algorithmische Grundstrukturen realisiert werden können.

Fallunterscheidung

Die Fallunterscheidung setzen wir zur Berechnung der Vorgängerfunktion ein:

$$\text{pred}(n) = (\text{if } n = 0 \text{ then } 0 \text{ else } n-1)$$

Fallunterscheidung am Beispiel der Vorgängerfunktion

Die Turingmaschine zur Berechnung von *pred* löscht einen Strich, geht nach rechts und hält an:

A | → * R E

Ist kein Strich vorhanden, geht sie sofort in den Endzustand über:

A * → * K E

Sequenzen und Schleifen

Eine Turingmaschine soll den Ausdruck $2(n+m)$ berechnen. Da wir schon eine Turingmaschine TM_1 zur Berechnung einer Summe haben, überlegen wir uns eine Turingmaschine TM_2 zur Multiplikation mit 2. Die Kopplung der beiden Maschinen geschieht so, daß der Endzustand von TM_1 zum Anfangszustand von TM_2 gemacht wird.

Sequenzen am Beispiel Kopplung von Turingmaschinen

Eine Möglichkeit zur Verdopplung besteht darin, das Bandalphabet um die beiden Symbole M und K zu erweitern. M wird als Markierung benutzt, K für eine kopierte Eins. Ausgehend von beispielsweise der Eingabe ||| wird der erste Strich markiert und an das rechte Ende kopiert. Danach wird der zweite Strich markiert und kopiert, bis schließlich alle Striche kopiert sind. Anschließend werden die Markier- und Kopiersymbole wieder durch Striche ersetzt. Mögliche Bandbelegungen sind:

Schleifen am Beispiel der Verdopplung

||| → M|K → |M|KK → ||MKKK → |||

Das Programm der verdoppelnden Turingmaschine lautet:

Programm der
verdoppelnden
Turingmaschine

A | * M R 1 ; ersten Strich markieren
 A * * L 3 ; im Zustand 3 K durch | ersetzen
 A K * K R A ; Kopieren fertig, nach rechts laufen
 1 | * | R 1 ; nach rechts laufen
 1 K * K R 1 ; nach rechts laufen
 1 * * K L 2 ; rechts angekommen, K kopieren
 2 K * K L 2 ; nach links laufen
 2 | * | L 2 ; nach links laufen
 2 M * | R A ; links angekommen, Marke löschen,
 ; Vorgang wiederholen
 3 | * | L 3 ; nach links laufen
 3 K * | L 3 ; K durch 1 ersetzen
 3 * * * R E ; fertig

19.2 Definition der Turingmaschine

Wir fassen die Konzeption der Turingmaschine in folgender Definition zusammen.

Definition der
Turingmaschine

Eine Turingmaschine besteht aus fünf Komponenten:

- Einer endlichen Menge von Zeichen, dem Bandalphabet, das als spezielles Zeichen das Leerzeichen „*“ enthält.
- Einer endlichen Menge von Zuständen mit
- einem ausgezeichneten Anfangszustand und
- einem ausgezeichneten Endzustand.
- Einer Überföhrungsfunktion, die zu jedem Paar aus Zustand und Bandzeichen das zu schreibende Zeichen, eine Kopfbewegung L, K, R und einen neuen Zustand angibt.

Nichtdeterministische Turingmaschinen haben anstelle einer Überföhrungsfunktion eine Überföhrungsrelation.

Interessanterweise sind nichtdeterministische Turingmaschinen genauso leistungsfähig wie deterministische Turingmaschinen. Das ist bei Kellerautomaten anders, während bei Automaten ebenfalls deterministische und nichtdeterministische Varianten ebenbürtig sind.

19.3 Modellierung von Turingmaschinen in Prolog

Analog zum Vorgehen bei Automaten und Kellerautomaten deklarieren wir die Komponenten einer Turingmaschine. Als Beispiel betrachten wir den Akzeptor der Sprache $L_2 = \{a^n b^n \mid n \in \mathbb{N}\}$:

```

alphabet(X):- member(X, [a,b,*]).
zustand(X):- member(X, [a,1,2,3,e]).
anfangszustand(a).
endzustand(e).

/* uebergang(+Zustand, +Eingabe, -Ausgabe,
             -Kopfbewegung, -NeuerZustand). */
uebergang(a, *, *, k, e).
uebergang(a, a, *, r, 1).
uebergang(1, a, a, r, 1).
uebergang(1, b, b, r, 1).
uebergang(1, *, *, l, 2).
uebergang(2, b, *, l, 3).
uebergang(3, b, b, l, 3).
uebergang(3, a, a, l, 3).
uebergang(3, *, *, r, a).

```

Modellierung einer
Turingmaschine für
die Sprache $a^n b^n$

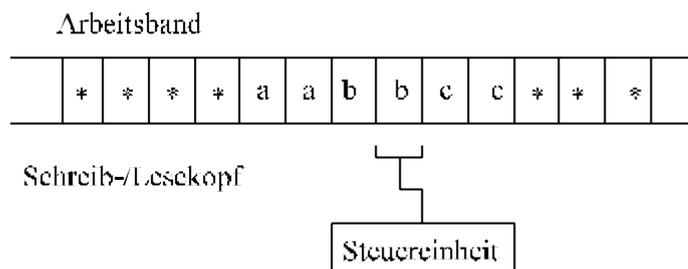
Die fünf Komponenten bestimmen eine spezifische Turingmaschine. Die Arbeitsweise von Turingmaschinen kann dagegen allgemeingültig beschrieben werden. Dazu definieren wir das Prädikat *turingmaschine* (+Zustand, +LinkerSpeicher, +RechterSpeicher), welches im ersten Argument den aktuellen Zustand und in den beiden weiteren Argumenten das Arbeitsband verwaltet.

Wenn man eine Turingmaschine in Pascal simuliert, so modelliert man das Arbeitsband als Reihung (array) und die aktuelle Kopfposition als Index in die

Modellierung des
Arbeitsbandes
durch zwei Listen

Reihung. In Prolog steht anstelle der Reihung die Liste zur Verfügung, welche allerdings keinen Direktzugriff über einen Index erlaubt. Daher modelliert man das Arbeitsband am besten durch zwei Listen, wobei der Kopf der zweiten Liste das aktuelle Zeichen unter dem Schreib-/Lesekopf enthält.

Abb. 19-3
Modellierung des
Arbeitsbandes
einer Turing-
maschine in
Prolog



Das in Abb. 19-3 dargestellte Arbeitsband wird durch folgenden beiden Listen repräsentiert:

```
LinkerSpeicher = [b, a, a]
RechterSpeicher = [b, c, c]
```

wobei die Elemente der Liste *LinkerSpeicher* in der Reihenfolge gespeichert sind, wie sie der Schreib-/Lesekopf der Reihe nach lesen kann. Das so modellierte Arbeitsband ist nur sechs Zellen breit. Um es nach beiden Seiten unbegrenzt zu erweitern, führen wir zwei Klauseln ein, die das Arbeitsband vergrößern, wenn die Turingmaschine dabei ist, den durch *LinkerSpeicher* und *RechterSpeicher* definierten Bereich zu verlassen:

Modellierung des
beidseitig
unbegrenzten
Arbeitsbandes

```
/* turingmaschine(+Zustand,+LinkerSpeicher,
                 +RechterSpeicher) */
turingmaschine(Zustand, [], RechterSpeicher):-
    turingmaschine(Zustand, [*], RechterSpeicher).
turingmaschine(Zustand, LinkerSpeicher, []):-
    turingmaschine(Zustand, LinkerSpeicher, [*]).
```

Ein Arbeitsschritt der Turingmaschine besteht in einem Übergang, der durch den aktuellen Zustand und das aktuelle Zeichen bestimmt wird. Das dabei

geschriebene Zeichen und die ausgeführte Kopfbewegung ändern das Arbeitsband. Für jede Kopfbewegung schreibt man eine eigene Klausel:

Modellierung der
Arbeitsweise einer
Turingmaschine

```
/* Uebergang nach links */
turingmaschine(Zustand, [KopfL|RestL], [KopfR|RestR]):-
    uebergang(Zustand, KopfR, KopfNeu, l, NeuerZustand),
    turingmaschine(NeuerZustand, RestL, [KopfL, KopfNeu|RestR]).

/* Uebergang nach rechts */
turingmaschine(Zustand, [KopfL|RestL], [KopfR|RestR]):-
    uebergang(Zustand, KopfR, KopfNeu, r, NeuerZustand),
    turingmaschine(NeuerZustand, [KopfNeu, KopfL|RestL], RestR).
```

```

/* keine Bewegung */
turingmaschine(Zustand, LinkerSpeicher, [KopfR|RestR]):-
    uebergang(Zustand, KopfR, KopfNeu, k, NeuerZustand),
    turingmaschine(NeuerZustand, LinkerSpeicher,
        [KopfNeu|RestR]).

/* Terminierung */
turingmaschine(Zustand, _, _):-
    endzustand(Zustand).
    
```

Zum Betrieb einer Turingmaschine als Akzeptor definieren wir das Prädikat *akzeptiere*:

```

akzeptiere(Wort):-
    anfangszustand(Zustand),
    zerlegen(Wort, Eingabeliste),
    turingmaschine(Zustand, [], Eingabeliste).
    
```

Initialisierung einer
Turingmaschine

Die Veranschaulichung der Arbeit einer Turingmaschine übernimmt die folgende Ausgabeklausel, welche vor allen anderen Klauseln des Prädikats *turingmaschine* stehen muß:

```

turingmaschine(Zustand, LinkerSpeicher, RechterSpeicher):-
    write(Zustand),
    tab(2), zeigelinks(LinkerSpeicher),
    tab(1), zeigerechts(RechterSpeicher), nl,
    fail.

zeigelinks([]).
zeigelinks([Kopf|Rechts]):-
    zeigelinks(Rechts),
    write(Kopf).
zeigerechts([]).
zeigerechts([Kopf|Rechts]):-
    write(Kopf),
    zeigerechts(Rechts).
    
```

Darstellung der
Arbeitsweise

Die Anfrage `?- akzeptiere('aabb')`. führt zu folgender Ausgabe, wobei zuerst der aktuelle Zustand und dann die Belegung des Arbeitsbandes angezeigt wird. Rechts vom Leerzeichen zwischen linkem und rechtem Speicher steht das Zeichen unter dem Schreib-/Lesekopf:

```

a   aabb
a   * aabb
1   ** abb
1   **a bb
1   **ab b
1   **abb
    
```

Turingmaschine
beim Akzeptieren
des Wortes aabb

```

1  **abb *
2  **ab b*
3  **a b**
3  ** ab**
3  * *ab**
a  ** ab**
1  *** b**
1  ***b **
2  *** b**
3  ** ****
a  *** **
e  *** **

```

Bei rechnenden Turingmaschinen muß nach Erreichen des Endzustandes das Ergebnis der Rechnung vom Arbeitsband abgelesen werden. Es ist Konvention, daß das Ergebnis einer Berechnung an der Position des Schreib-/Lesekopfes beginnt und alle direkt aufeinanderfolgenden Striche umfaßt:

Ergebnis einer
rechnenden
Turingmaschine

```

turingmaschine(Zustand, _Links, Rechts):-
    endzustand(Zustand),
    zaehle_einer(Rchts, Einer),
    write('Ergebnis: '), write(Einer), nl.

zaehle_einer([], 0).
zaehle_einer([Kopf|_Rest], 0):-
    Kopf \== '|'.
zaehle_einer(['|'|Rest], N):-
    zaehle_einer(Rest, N1),
    N is N1 + 1.

```

Die Turingmaschine zur Addition liefert für die Aufgabe 2 + 3 folgendes Ergebnis:

addierende
Turingmaschine

```

a  ||#|||
a  * ||#|||
1  **|#|||
1  **|#|||
2  **|||
2  * *|||
e  **|||
Ergebnis: 5

```

19.4 Berechenbarkeit und Turingmaschine

Turingmaschinen sind leistungsfähiger als Kellerautomaten. Sie können beispielsweise die Sprache $L_3 = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ erkennen. Zum Beweis konstruieren wir eine Turingmaschine, die $a^n b^n c^n$ in zwei Schritten erkennt. Der erste Schritt besteht im Erkennen von $a^n b^n$, wobei auf dem Band als Resultat x^n stehen bleiben soll, im zweiten Schritt wird $x^n c^n$ nach der Methode von Beispiel 19.1.1 erkannt. Wir müssen uns deshalb nur noch um den ersten Schritt kümmern.

Im Zustand a streichen wir ein a und gehen in Zustand 1 über, gegebenenfalls erfolgt der Übergang in den Endzustand. Im Zustand 1 geht die Turingmaschine an das rechte Ende von $a^n b^n$. Im Zustand 2 ersetzt sie ein b durch x . Im Zustand 3 geht sie an den linken Anfang zurück:

```
uebergang(a, a, *, r, 1).
uebergang(a, x, x, k, e).
uebergang(a, *, *, k, e).
uebergang(1, a, a, r, 1).
uebergang(1, b, b, r, 1).
uebergang(1, c, c, l, 2).
uebergang(1, x, x, l, 2).
uebergang(2, b, x, l, 3).
uebergang(3, b, b, l, 3).
uebergang(3, a, a, l, 3).
uebergang(3, *, *, r, a).
```

Nach geeigneter Umbenennung von Zuständen und Alphabetzeichen kann man die Turingmaschine TM_1 aus Beispiel 19.1.1 an obige Turingmaschine TM_2 ankoppeln und die restliche Erkennungstätigkeit durchführen lassen. Beispielsweise muß der Endzustand von TM_2 zum Anfangszustand von TM_1 werden und die Alphabetzeichen a und b von TM_1 müssen in x und c umbenannt werden. Die durch die Verkettung der Turingmaschinen TM_2 und TM_1 entstehende Turingmaschine erkennt die Sprache L_3 .

Beispiel 19.1.3 zeigt, daß mit Turingmaschinen algorithmische Grundstrukturen realisiert werden können. Interpretiert man die Übergänge einer Turingmaschine graphisch, so stellt sich die Fallunterscheidung als Verzweigung, die Wiederholung als Zyklus und die Sequenz als Pfad im Zustandsgraphen dar. Damit ist intuitiv klar, daß Turingmaschinen alles berechnen können, was auch mit Computern berechnet werden kann. Ein Beweis dieser Aussage ist möglich, wenn man Turingmaschinen und Computer stärker formalisiert. Man spricht dann von der Turing-Berechenbarkeit und der RAM-Berechenbarkeit

eine
Turingmaschine für
die Sprache $a^n b^n c^n$

Kopplung von
Turingmaschinen

Turingmaschinen
für algorithmische
Grundstrukturen

(RAM, engl. = Random Access Machine) und weist nach, daß sich beide Maschinen wechselseitig simulieren können.

Berechenbarkeit
 μ -rekursiven Funktionen
 In der Mathematik hat man einen eigenen, auf der Betrachtung von Funktionen beruhenden, Berechenbarkeitsbegriff studiert. Ebenfalls durch wechselseitige Simulation konnte nachgewiesen werden, daß die Klasse der *m*-rekursiven Funktionen mit der durch Turing- und RAM-Maschinen berechenbaren Funktionen übereinstimmt. Welchen Ansatz man zur Definition des Begriffs *Berechenbarkeit* bislang machte, stets konnte der Nachweis geführt werden, daß kein mächtigerer Berechenbarkeitsbegriff gefunden werden konnte, als der der Turing-Berechenbarkeit. Dies drückt sich in der *Churchschen-These* wie folgt aus:

Churchsche-These
 Jede im intuitiven Sinn berechenbare Funktion ist auch Turing-berechenbar.

Mit der Turingmaschine ist somit ein leistungsfähigstes Maschinenmodell gefunden.

19.5 Grenzen der Turingmaschine

algorithmisch unlösbare Probleme
 Wir haben zwar mit der Turingmaschine ein leistungsfähigstes Maschinenmodell gefunden, dennoch hat auch dieses Modell Grenzen. Es gibt Probleme, die nicht algorithmisch gelöst werden können oder anders ausgedrückt, die nicht berechenbar sind. Ein nicht-konstruktiver Nachweis ist über den Vergleich der Mächtigkeit der Menge der Funktionen $f: \mathbb{N} \rightarrow \mathbb{N}$ mit der Menge der Turingmaschinen möglich.

Die Menge der Turingmaschinen über einem endlichen Alphabet ist abzählbar. Da Turingmaschinen letztlich durch die Übergänge definiert sind, kann man die endliche Anzahl der Turingmaschinen mit einem Übergang lexikographisch geordnet aufschreiben. So erhält man die Turingmaschinen TM_1, TM_2, \dots, TM_n . Nun betrachtet man die endliche Anzahl der Turingmaschinen mit zwei Übergängen, ordnet diese ebenfalls lexikographisch an und numeriert weiter: $TM_{n+1}, TM_{n+2}, \dots, TM_m$. Die Aufzählung setzt sich in gleicher Weise mit Turingmaschinen aus drei, vier,... Übergängen fort.

Cantorsches Diagonalisierungsverfahren
 Die Menge M_f der Funktionen $f: \mathbb{N} \rightarrow \mathbb{N}$, deren Definitions- und Wertebereich die natürlichen Zahlen sind, ist nicht abzählbar. Zum Beweis setzt man das Cantorsche Diagonalisierungsverfahren ein. Nimmt man an, daß die Men-

ge M_f dieser Funktionen abzählbar sei, so kann man die Funktionen in der Art f_1, f_2, f_3, \dots aufzählen.

Die Funktion h , mit $h(n) = f_n(n) + 1$, definieren wir nun über die Diagonale folgender zweidimensionalen Wertetabelle:

	1	2	3	4	5	...
f_1	$f_1(1)$	$f_1(2)$	$f_1(3)$	$f_1(4)$	$f_1(5)$...
f_2	$f_2(1)$	$f_2(2)$	$f_2(3)$	$f_2(4)$	$f_2(5)$...
f_3	$f_3(1)$	$f_3(2)$	$f_3(3)$	$f_3(4)$	$f_3(5)$...
...

Tabelle 19-1
Diagonalisierung
der Funktionen
 $f: \mathbb{N} \rightarrow \mathbb{N}$

Durch diese Diagonalkonstruktion der Funktion h wird sichergestellt, daß h einerseits zu M_f gehört, denn Definitions- und Wertebereich von h ist die Menge der natürlichen Zahlen, andererseits sich von allen Funktionen f_i zumindest an der Diagonalstelle unterscheidet. Daher kann es kein j geben mit $f = h$. Obwohl also h zur Menge M_f gehört kommt h nicht in der Aufzählung f_1, f_2, f_3, \dots vor. Dies steht im Widerspruch zur Annahme, daß M_f abzählbar wäre. Also ist M_f nicht abzählbar.

Die abzählbar vielen Turingmaschinen können nicht die überabzählbar vielen Funktionen der Menge M_f berechnen. Also gibt es *nicht berechenbare* Funktionen!

nicht berechenbare
Funktionen

Den *konstruktiven* Nachweis der Grenzen von Turingmaschinen führt man über das Halteproblem:

Gibt es einen Algorithmus, der von allen Prolog- (Pascal-, Eiffel-, Turing-) Programmen entscheidet, ob sie angesetzt auf eine beliebige Eingabe anhalten oder nicht?

Halteproblem

Die Antwort lautet *Nein*. Angenommen wir hätten ein Prolog-Prädikat *haelt_an(Programm, Eingabe)*, das von einem Prolog-Programm feststellt, ob es bei gegebener Eingabe anhält. Das Programm könnte dabei als Liste seiner Klauseln im ersten Parameter übergeben werden.

Wir könnten dann das Prädikat *seltsam(+Programm)* wie folgt implementieren:

```
seltsam(Programm):-
    haelt_an(Programm, Programm),
    write('Das Programm hält an.').
    seltsam(Programm).

seltsam(Programm):-
    not haelt_an(Programm, Programm),
```

das Programm
seltsam

```
write('Das Programm hält nicht an.').
```

das seltsame
seltsam-Programm

Welche Antwort liefert die Anfrage $?- \textit{seltsam}(\textit{seltsam}).?$ Angenommen, das Programm *seltsam* hält bei der Eingabe *seltsam* an. Dann sorgt die Endrekursion in der ersten *seltsam*-Klausel dafür, daß *seltsam* nicht anhält. Nehmen wir hingegen an, daß *seltsam* nicht anhält, so wird das Ziel *not haelt_an(Programm, Programm)* in der zweiten Klausel erfüllt. Das nachfolgende *write*-Teilziel wird ebenfalls erfüllt, wodurch *seltsam* anhält.

Beidesmal ergibt sich ein Widerspruch zur Annahme. Da es das *seltsam*-Prädikat wie oben angegeben gibt, kann der Widerspruch nur mit der Voraussetzung, daß es ein *haelt_an*-Prädikat gibt, erklärt werden. Ein solches Prädi-

das *seltsam*-
Programm als Tu-
ringmaschine
kat ist nicht möglich.

Es stellt sich die Frage, ob dieser Gedankengang auf Turingmaschinen übertragbar ist, denn es ist nicht ersichtlich, wie man einer Turingmaschine eine andere Turingmaschine als Eingabe übergeben kann. Dies gelingt durch geeig-

Codierung von
Turingmaschinen
nete Codierung von Turingmaschinen. Man codiert die Komponenten einer Turingmaschine nach folgendem Verfahren: Das verwendete Alphabet A , die auftretenden Zustände Z und die Kopfbewegungen B werden durch $A = \{a_1, a_2, a_3, \dots, a_n\}$, $Z = \{z_1, z_2, z_3, \dots, z_m\}$ und $B = \{b_1, b_2, b_3\}$ numeriert. Zeichen, Zustände und Kopfbewegungen codiert man durch die Binärdarstellung der zugehörigen Indizes: $\text{code}(a_i) = \text{bin}(i)$, $\text{code}(z_j) = \text{bin}(j)$ und $\text{code}(b_k) = \text{bin}(k)$. Ein Übergang wird durch Codierung seiner Komponenten codiert:

$$\text{code}(z_i \ a_j \rightarrow a_k \ b_l \ z_m) = \#\#\text{bin}(i)\#\text{bin}(j)\#\text{bin}(k)\#\text{bin}(l)\#\text{bin}(m)$$

Die Codierung einer Turingmaschine besteht dann in der Folge der Codierungen ihrer Übergänge. Die Codierung der Eingabe ergibt sich aus der Codie-

Konstruktion der
seltsamen
Turingmaschine
rung des Alphabets.

Wir nehmen an, daß es eine Turingmaschine TM_H gibt, welche für eine beliebige Turingmaschine TM und Eingabe w feststellt, ob TM angesetzt auf w anhält oder nicht. Anhalten soll durch eine 1 auf einem sonst leeren Arbeitsband angedeutet werden, Nicht-Anhalten durch das leere Arbeitsband. Dann konstruieren wir wie folgt eine neue Turingmaschine TM_S . Zunächst kopiert TM_S seine Eingabe und arbeitet dann wie TM_H . Den Endzustand E von TM_H machen wir zu einem Zwischenzustand E_H von TM_S und den Endzustand von TM_S nennen wir E_S . Die Übergänge von TM_H ergänzen wir um zwei weitere Übergänge:

$$\begin{aligned} E_H \ 1 &\rightarrow 1 \ K \ E_H \\ E_H \ * &\rightarrow * \ K \ E_S \end{aligned}$$

die unmögliche
seltsam-
Turingmaschine

Wie reagiert die Turingmaschine TM_S , wenn Sie als Eingabe ihre eigene Codierung erhält? Zunächst kopiert Sie die Eingabe $code(TM_S)$ und arbeitet dann wie TM_H mit der Eingabe $TM=TM_S$ und $w=TM_S$. Angenommen TM_S hält an, dann erreicht TM_S aufgrund der Arbeitsweise von TM_H den Zustand E_H mit einer 1 auf dem Arbeitsband. Da TM_S beim nächsten Übergang die 1 wieder hinschreibt, keine Kopfbewegung durchführt und im Zustand E_H verbleibt, gerät TM_S in eine Endlosschleife. Nimmt man hingegen an, daß TM_S nicht anhält, so erreicht TM_S aufgrund der Arbeitsweise von TM_H den Zustand E_H mit einem leeren Arbeitsband. Der nachfolgende Zustandsübergang führt dann in den Endzustand. Die seltsame Turingmaschine TM_S kann es demnach nicht geben. Die Annahme, daß es eine Halte-Turingmaschine TM_H gibt, ist falsch.

19.6 Aufgaben

1. Geben Sie Turingmaschinen für die folgenden drei Grundfunktionen an:
 - a) Nullfunktion: $z(n) = 0, n \in \mathbb{N}$
 - b) Nachfolgerfunktion: $\text{succ}(n) = n + 1, n \in \mathbb{N}$
 - c) Projektion auf das dritte Argument: $p_3(x_1, x_2, x_3, x_4, \dots, x_n) = x_3$

2. Entwickeln Sie eine Turingmaschine, die Wörter über dem Alphabet $\{a, b\}$ erkennt, welche mit $aab\dots$ beginnen.

3. Gesucht ist eine Turingmaschine zur Berechnung von 2^n . Die Arbeitsweise der gesuchten Turingmaschine besteht darin, das Ergebnis E mit 1 zu initialisieren und dann n -mal mit 2 zu multiplizieren. Dazu dekrementiert man in jedem Schleifendurchlauf n , positioniert den Kopf auf den Anfang des bisherigen Ergebnisses, verdoppelt es und geht wieder zu n zurück. Für $n=3$ stellt sich die Arbeit wie folgt dar:

```

| | | # |      ; Initialisierung
| | # | |     ; erste Multiplikation
| # | | |     ; zweite Multiplikation
# | | | | | | | ; dritte Multiplikation
| | | | | | | | ; Ergebnis

```


20 Parser und Interpreter

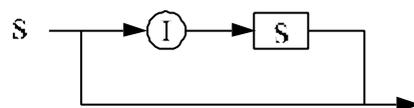
In diesem Kapitel betrachten wir einige Beispiele mit Anwendung der in den letzten Kapiteln dargestellten Theorie. Dabei legen wir besonderen Wert auf die verschiedenen Möglichkeiten, Sprachen zu beschreiben (Grammatiken und Syntaxdiagramme) und syntaktisch (Parser) beziehungsweise semantisch (Interpreter) zu analysieren.

20.1 Strichlisten

Natürliche Zahlen können als Strichlisten dargestellt werden I, II, III..., wobei kein Strich für die Zahl 0 steht (vgl. [Göh1]). Eine Grammatik für die Sprache der Strichlisten ist gegeben durch:

$$S \rightarrow I S \mid \epsilon$$

Im Syntaxdiagramm stellt sich diese Grammatik so dar:



Strichlisten-Grammatik

Abb. 20-1
rekursives
Syntaxdiagramm
für Strichlisten

Wer lieber in Iterationen denkt, der baut sich eher ein äquivalentes Syntaxdiagramm mit einer Schleife:

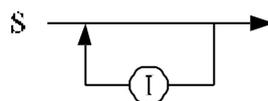


Abb. 20-2
iteratives
Syntaxdiagramm
für Strichlisten

Da die Grammatikregel rekursiv ist und Prolog nur die Rekursion als Wiederholungsstruktur kennt, bleiben wir beim rekursiven Ansatz. Als Parser benötigen wir einen Automaten, da sie Strichlistensprache regulär ist. Dies erkennt man am iterativen Syntaxdiagramm am leichtesten, denn der reguläre Ausdruck I^* beschreibt die Sprache.

Das Syntaxdiagramm kann man direkt in einen Parser umsetzen, der Umweg über einen Automatengraphen ist nicht nötig.

```
parse(['I'|S]):- parse(S).      /* 1. Alternative */
parse([]).                    /* 2. Alternative */
```

Parser für
Strichlisten

Der Interpreter soll eine Strichliste erkennen und gleichzeitig den Wert der Strichliste ermitteln. Dazu erweitern wir einfach den Parser um die Interpretationskomponente und die Parameterliste um den Parameter *Wert*:

```

Interpreter für
Strichlisten      interpretiere(['I'|S], Wert):-
                   interpretiere(S, Wert1),
                   Wert is Wert1 + 1.
                   interpretiere([], 0).
    
```

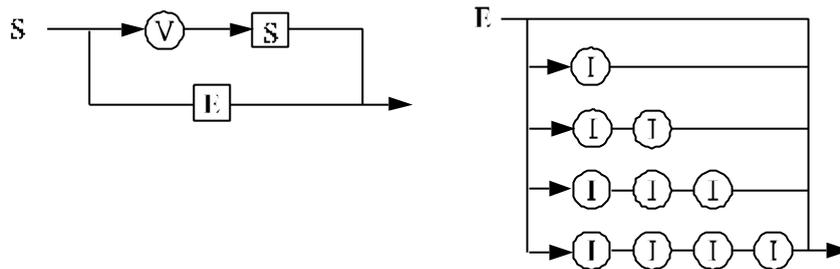
Die Strichlistendarstellung wird übersichtlicher, wenn man fünf Striche stets durch einen Fünferblock ersetzt. Die neue Sprache wird durch folgende Grammatik beschrieben:

```

erweiterte
Strichlisten-
Grammatik      S → V S | E
               E → ∈ | I | II | III | IIII
    
```

Wir bilden die entsprechenden Syntaxdiagramme:

Abb. 20-3
Syntaxdiagramme
der erweiterten
Strichlisten-
Grammatik



Jede Grammatikregel entspricht einem Syntaxdiagramm und jedes Syntaxdiagramm einem *parse*-Prädikat:

```

Parser der
erweiterten
Strichlisten      parse_fuenfer(['V'|S]):- parse_fuenfer(S).
                   parse_fuenfer(E):- parse_einer(E).

                   parse_einer([]).
                   parse_einer(['I']).
                   parse_einer(['I','I']).
                   parse_einer(['I','I','I']).
                   parse_einer(['I','I','I','I']).
    
```

Aus dem Parser lässt sich problemlos der Interpreter entwickeln:

```
interpret_fuenfer(['V'|S], Wert):-
    interpret_fuenfer(S, Wert1),
    Wert is Wert1 + 5.
interpret_fuenfer(E, Wert):-
    interpret_einer(E, Wert).

interpret_einer([], 0).
interpret_einer(['I'], 1).
interpret_einer(['I','I'], 2).
interpret_einer(['I','I','I'], 3).
interpret_einer(['I','I','I','I'], 4).
```

Interpreter der erweiterten Strichlisten

20.2 mini-LOGO

Als weiteres Anwendungsbeispiel betrachten wir einen Parser und Interpreter für einige Turtlegrafik-Befehle der Programmiersprache LOGO. Mit *Forward* (*fd*) und *Backward* (*bk*) kann die Turtle vor und zurück bewegt werden. *Left* (*lt*) und *Right* (*rt*) drehen sie um eine Winkel nach links beziehungsweise rechts. Mittels *Penup* (*pu*) und *Pendown* (*pd*) nimmt man den Zeichenstift der Turtle hoch beziehungsweise runter. Als Kontrollstruktur steht *Repeat* (*rp*) zur Verfügung.

Turtlegrafik-Befehle von LOGO

Die Syntax unserer Sprache *mini-LOGO* legen wir durch Syntaxdiagramme fest:

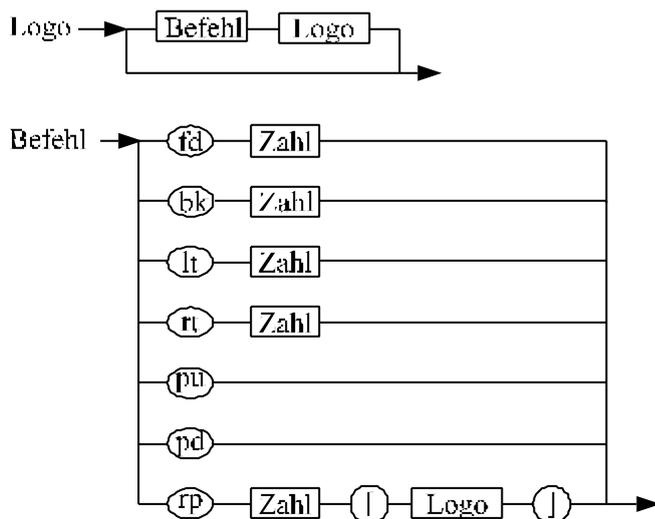


Abb. 20-4
Syntaxdiagramme von mini-LOGO

Parsebaum für mini-LOGO	<p>Im Unterschied zu den bisherigen Parsern soll der Parser für <i>mini-LOGO</i> einen Parsebaum erstellen, damit die anschließende Interpretation einfacher von statten geht. Der Parsebaum ist in diesem Beispiel relativ einfach, weil er im wesentlichen eine Liste von LOGO-Befehlen ist.</p>
Scanner	<p>Die Aufgabe der Symbolerkennung in der Eingabe kann man einem Scanner oder dem Benutzer des Parsers übertragen. Letzteres enthebt von der Pflicht einen Scanner auf der Basis eines erkennenden Automaten zu realisieren, ersteres von der mühsameren Eingabe. Im Folgenden gehen wir davon aus, daß ein Scanner zur Verfügung steht. Beim Beispiel mini-PASCAL wird dann ein Scanner angegeben, der auch für mini-LOGO benutzt werden kann.</p>
Umsetzung der Syntaxdiagramme in entsprechende Prädikate	<p>Die beiden Syntaxdiagramme lassen sich leicht in die beiden zugehörigen Parse-Prädikate <i>parse_logo1</i> und <i>parse_befehl</i> umsetzen. Die Alternativen im Syntaxdiagramm von <i>Befehl</i> werden zu Alternativen in den <i>parse_befehl</i> Klauseln. Etwas Probleme bereitet die Verarbeitung der Eingabe. Im ersten Argument werden die noch nicht verarbeiteten Eingabesymbole übergeben, im zweiten erhält man den nächsten Befehl und im dritten Argument die verbleibenden Eingabesymbole zurück.</p>
Parser für mini-LOGO	<pre> parse_logo(Logo, Parsebaum):- parse_logo1(Logo, Parsebaum, []). parse_logo1(Liste1, [Befehl Liste2], Liste4):- parse_befehl(Liste1, Befehl, Liste3), parse_logo1(Liste3, Liste2, Liste4), !. parse_logo1(Liste, [], Liste). parse_befehl([fd Liste1], fd(Zahl), Liste2):- parse_zahl(Liste1, Zahl, Liste2). parse_befehl([bk Liste1], bk(Zahl), Liste2):- parse_zahl(Liste1, Zahl, Liste2). parse_befehl([lt Liste1], lt(Zahl), Liste2):- parse_zahl(Liste1, Zahl, Liste2). parse_befehl([rt Liste1], rt(Zahl), Liste2):- parse_zahl(Liste1, Zahl, Liste2). parse_befehl([pd Liste], pd, Liste). parse_befehl([pu Liste], pu, Liste). parse_befehl([rp Liste1], rp(Zahl, Befehle), Liste3):- parse_zahl(Liste1, Zahl, ['' Liste2]), Zahl >= 0, parse_logo1(Liste2, Befehle, ['' Liste3]). </pre>

```
parse_zahl([Zahl|Liste], Zahl, Liste):-
integer(Zahl).
```

Als Beispiel betrachten wir den erzeugten Parsebaum zum mini-LOGO Programm: `rp 8 [lt 45 fd 30]`.

```
?- parse_logo([rp, 8, [' ',lt, 45, fd, 30, '']], P).

P = [rp(8, [lt(45),fd(30)])]
```

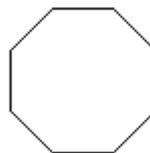
Abb. 20-5
Beispiel eines Parsebaums

Da der Parser einen Parsebaum aufbaut, in dem die Strukturinformation des eingegebenen mini-LOGO-Programms enthalten ist, läßt sich der Interpreter problemlos implementieren. In *fiw*-Prolog ist Grafik möglich, der Interpreter kann die Turtlezeichnungen auf den Bildschirm zeichnen. Die Entwicklung einer grafikfähigen TV-SWI-Prolog ist im Gange.

```
logo(Befehle):-
  parse_logo(Befehle, Liste),
  tell(graphic),
  pixel(X,Y,_F),
  pixel(X,Y,15),
  interpret_logo(Liste),
  skip(13),
  told.

/* LOGO-Interpreter */
interpret_logo([Befehl|Befehle]):-
  interpret_befehl(Befehl),
  interpret_logo(Befehle).
interpret_logo([]).

/* jeden Befehl mit RETURN ausführen */
/*interpret_befehl(_):-
  skip(13), fail. */
```



Interpreter für mini-LOGO

```

/* Forward - fd */
interpret_befehl(fd(Zahl)):-
    forward(Zahl).

/* Back - bk */
interpret_befehl(bk(Zahl)):-
    Zahl1 is -Zahl,
    forward(Zahl1).

/* Left - lt */
interpret_befehl(lt(Zahl)):-
    left(Zahl).

/* Right - rt */
interpret_befehl(rt(Zahl)):-
    Zahl1 is -Zahl,
    left(Zahl1).

/* Penup - pu */
interpret_befehl(pu):-
    graphcolor(0, 0).

/* Pendown - pd */
interpret_befehl(pd):-
    graphcolor(15, 0).

/* Repeat - rp */
interpret_befehl(rp(0, _Befehle)):- !.
interpret_befehl(rp(Zahl, Befehle)):-
    interpret_logo(Befehle),
    Zahl1 is Zahl - 1,
    interpret_befehl(rp(Zahl1, Befehle)).

?- logo([rp, 6, '[', rp, 6, '[', fd, 10, lt, 10, ']', lt,
        60, rp, 12, '[', fd, 10, lt, 10, ']', lt, 60, ']').
    
```

Abb. 20-6
 Interpretation
 eines mini-LOGO
 Programmes

20.3 mini-PASCAL

Im nächsten Beispiel packen wir ein etwa größeres Projekt an. Es sollen ein Scanner, Parser, Interpreter und Compiler für mini-PASCAL realisiert werden. Zunächst legen wir die Syntax von mini-PASCAL mit einer rechtsrekursiven Grammatik fest:

Program	→	program Bezeichner; begin Anweisungen end .	Grammatik von mini-PASCAL
Anweisungen	→	Anweisung ; Anweisungen Anweisung	
Anweisung	→	Variable := Ausdruck	
Anweisung	→	while Ausdruck do Anweisung	
Anweisung	→	begin Anweisungen end write (Ausdruck) ε	
Ausdruck	→	Einfacher_Ausdruck < Einfacher_Ausdruck	
Ausdruck	→	Einfacher_Ausdruck	
Einfacher_Ausdruck	→	Term Term + Einfacher_Ausdruck	
Einfacher_Ausdruck	→	Term - Einfacher_Ausdruck	
Term	→	Faktor Faktor * Term Faktor / Term	
Faktor	→	(Ausdruck) Zahl Variable	
Variable	→	Bezeichner	

Anschaulich wird diese Grammatik durch folgende Syntaxdiagramme - außer Bezeichner und Zahl - beschrieben:

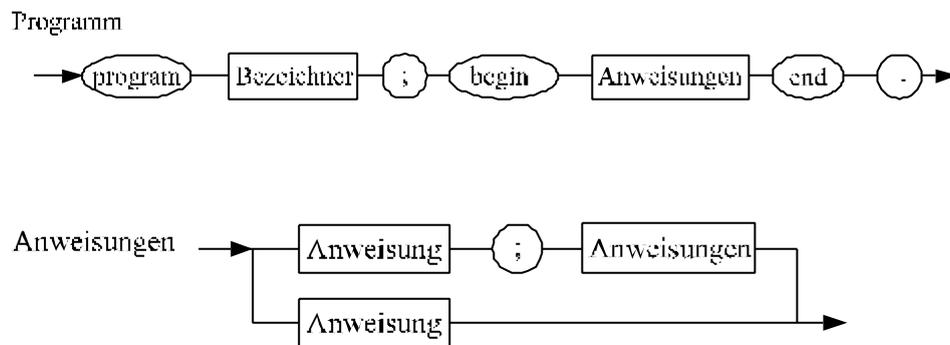
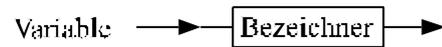
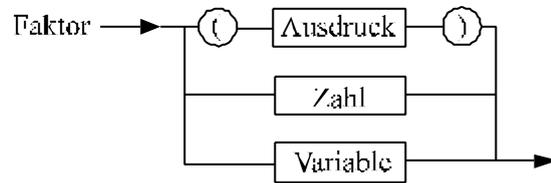
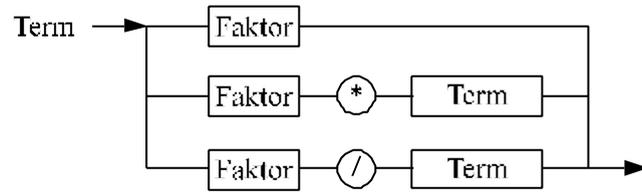
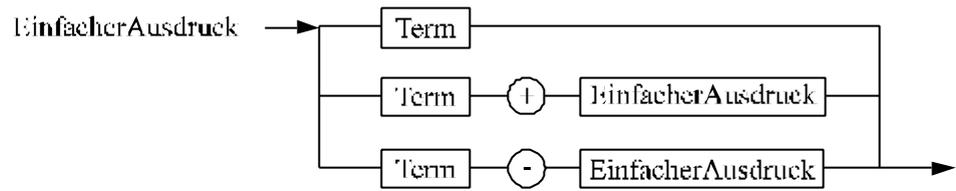
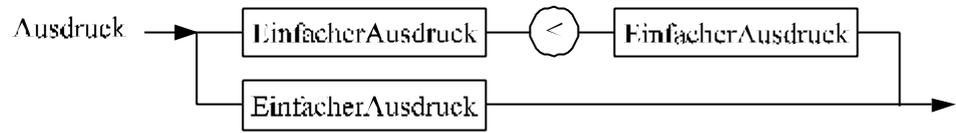
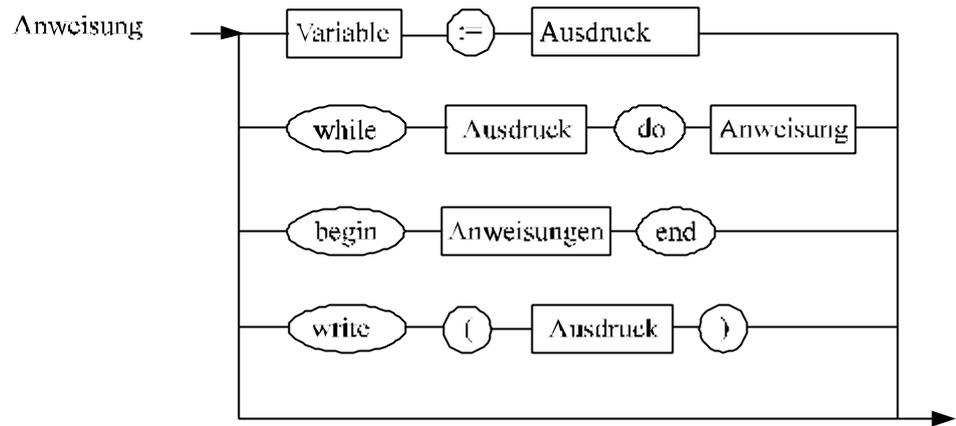


Abb. 20-7
Syntaxdiagramme
von mini-PASCAL



20.3.1 Scanner für mini-PASCAL

Quellprogramme speichert man in Textdateien, auf die der Scanner zugreift. Er liest das gewünschte Quellprogramm und generiert jeweils aus einem oder mehreren aufeinanderfolgenden Zeichen ein Symbol. Spezielle Symbole sind Zahlen, Bezeichner und reservierte Wörter. Daneben gibt es noch Symbole, die als Terminale in den Syntaxdiagrammen auftreten, wie zum Beispiel +, -, :=, und <=. Die Liste der vom Scanner erzeugten Symbole wird später als Eingabe für den Parser benutzt.

Zahlen, Bezeichner, reservierte Wörter und sonstige Symbole

Das mühselige Zusammensetzen von Ziffern zu Zahlen und Buchstaben zu Bezeichnern, delegieren wir an das Prädikat *readln/5* der Systembibliothek `LIB\READLN.PL`. Es liest den kompletten Quelltext ein und bildet eine Liste der darin vorkommenden Zahlen und Atome.

```
init_scanner(Datei):-
    see(Datei),
    readln(Eingabe, _, ".", "0123456789", lowercase),
    seen,
    scan(Eingabe, Symbole),
    retractall(symbole(_)), asserta(symbole(Symbole)).
```

Einlesen des Quelltextes und Bildung der Symbolliste

Einige Vergleichsoperatoren werden anschließend vom *scan*-Prädikat zusammengesetzt, Bezeichner werden zur Unterscheidung von anderen Symbolen in *bez*-Strukturen verpackt:

```
scan([Symbol1, Symbol2|Symbole1], [Symbol3|Symbole3]):-
    zusammensetzen(Symbol1, Symbol2, Symbol3), !,
    scan(Symbole1, Symbole3).
scan([Symbol1|Symbole1], [Symbol2|Symbole2]):-
    verpacken(Symbol1, Symbol2), !,
    scan(Symbole1, Symbole2).
scan([], []).
```

Zusammensetzen und Verpacken von Symbolen

```
zusammensetzen(<, =, <=).
zusammensetzen(<, >, <>).
zusammensetzen(>, =, >=).
zusammensetzen(:, =, :=).
```

```
verpacken(Symbol, bez(Symbol)):-
    name(Symbol, [K|_]), is_alpha(K),
    not reserviert(Symbol).
verpacken(Symbol, Symbol).
```

```
reserviert(X):-
    member(X, [while, do, begin, end, write,
               program, if, then, else]).
```


Gegenüber dem Parser muß der Scanner zwei wesentliche Dienstfunktionen wahrnehmen. Er muß das aktuelle Symbol auf Anforderung dem Parser zur Verarbeitung bereitstellen und zum nächsten Symbol bei Bedarf weiterschalten. Die erste Funktion erledigt das Prädikat *symbol*, das zweite das Prädikat *next_symbol*

Schnittstelle
zwischen Scanner
und Parser

```
symbol(Symbol):-
    symbole([Symbol|_]).
next_symbol:-
    retract(symbole([_|Symbole])),
    asserta(symbole(Symbole)).
```

20.3.2 Parser für mini-PASCAL

Der Parser ist weitgehend eine direkte Umsetzung der Syntaxdiagramme in entsprechende Prolog-Prädikate. Er baut mittels Unifikation den Parsebaum auf, welcher für die Interpretation oder Übersetzung benötigt wird. Anweisungen können leicht geparkt werden:

Parser für
Programm und
Anweisungen von
mini-PASCAL

```
parse_programm(Datei, Parsebaum):-
    init_scanner(Datei),
    parse_programm(Parsebaum), !.

parse_programm(Anweisungen):-
    parse_symbol(program),
    parse_variable(_Name),
    parse_symbol(;),
    parse_anweisung(begin(Anweisungen)),
    parse_symbol(.).

parse_anweisungen([Anweisung|Anweisungen]):-
    parse_anweisung(Anweisung),
    (parse_symbol(;)->
     parse_anweisungen(Anweisungen);
     Anweisungen = []).
```

```

parse_anweisung(zuweisung(Variable, Ausdruck)):-
    parse_variable(Variable), !,
    parse_symbol(:=),
    parse_ausdruck(Ausdruck).

parse_anweisung(while(Ausdruck, Anweisung)):-
    parse_symbol(while), !,
    parse_ausdruck(Ausdruck),
    parse_symbol(do),
    parse_anweisung(Anweisung).

parse_anweisung(begin(Anweisungen)):-
    parse_symbol(begin), !,
    parse_anweisungen(Anweisungen),
    parse_symbol(end).

parse_anweisung(write(Ausdruck)):-
    parse_symbol(write), !,
    parse_symbol('('),
    parse_ausdruck(Ausdruck),
    parse_symbol(')').

parse_anweisung(nil).

```

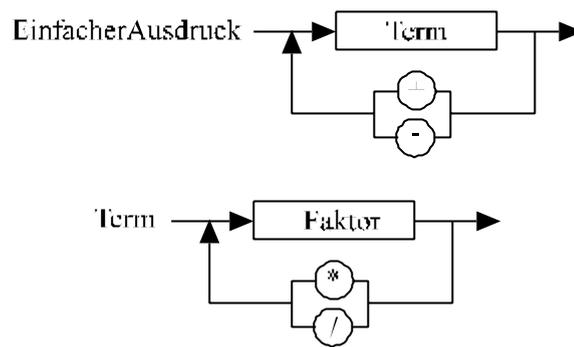
Die Ausdrücke sind schwieriger zu parsen. Zur Illustration betrachten wir den einfachen Ausdruck $a - b - c$. Leiten wir diesen Ausdruck aus unserer Grammatik ab, so erhalten wir wegen *EinfacherAusdruck* \rightarrow *Term* - *EinfacherAusdruck* den Term a und den einfachen Ausdruck $b - c$. Daraus erhält man

rechtsrekursive
Grammatikregeln

$a - (b - c)$ und nicht $(a - b) - c!$

Wir müssen beim Parsen die Regel *von links nach rechts* beachten. Die rechtsrekursiven Grammatikregeln legen allerdings die umgekehrte Reihenfolge nahe. Eine Umformulierung in linksrekursive Grammatikregeln wäre problemlos möglich, doch dadurch handelt man sich linksrekursive Klauseln ein, die leicht zu unendlichen Pfaden im Prolog-Suchbaum führen. Am einfachsten ist es, die rechtsrekursiven Grammatikregeln für *einfacherAusdruck* und *Term* in iterative Form zu bringen. Im Syntaxdiagramm sieht das wie folgt aus:

Abb. 20-8
Iterative Syntax-
diagramme für
einfacher Ausdruck
und Term



Das Lösungsverfahren sei am Beispiel $a - b - c$ erklärt. Der erste Term hat eine Sonderrolle, da er den Ausdruck einleitet. Er wird geparsert und liefert als Teilbaum das Blatt a . Wenn ein Strichoperator folgt, wird der bisherige Term a mit dem Strichoperator und dem nachfolgenden Term b zusammengesetzt. Dies liefert den Teilbaum $a - b$. Wenn ein weiterer Strichoperator folgt, wird der Vorgang wiederholt. Dies liefert dann den Term $(a - b) - c$. Folgt kein Strichoperator, so ist der Vorgang beendet und der gesuchte Parsebaum komplett aufgebaut.

Verfahren zum
Parse arithmetischer
Ausdrücke

Parser für
arithmetische Aus-
drücke

```

parse_ausdruck(Ausdruck):-
    parse_einfacher_ausdruck(Ausdruck1),
    parse_ausdruck(Ausdruck1, Ausdruck).
parse_ausdruck(Ausdruck1, Ausdruck1 < Ausdruck2):-
    parse_symbol(<), !,
    parse_einfacher_ausdruck(Ausdruck2).
parse_ausdruck(Ausdruck1, Ausdruck1 = Ausdruck2):-
    parse_symbol(=), !,
    parse_einfacher_ausdruck(Ausdruck2).
parse_ausdruck(Ausdruck, Ausdruck).

parse_einfacher_ausdruck(Einfacher_Ausdruck):-
    parse_term(Term),
    parse_einfacher_ausdruck(Term, Einfacher_Ausdruck).
parse_einfacher_ausdruck(Term1, Einfacher_Ausdruck):-
    parse_symbol(+), !,
    parse_term(Term2),
    parse_einfacher_ausdruck(Term1+Term2, Einfacher_Ausdruck).
parse_einfacher_ausdruck(Term1, Einfacher_Ausdruck):-
    parse_symbol(-), !,
    parse_term(Term2),
    parse_einfacher_ausdruck(Term1-Term2, Einfacher_Ausdruck).
parse_einfacher_ausdruck(Einf_Ausdruck, Einf_Ausdruck).

parse_term(Term):-
    parse_faktor(Faktor),
    parse_term(Faktor, Term).
parse_term(Faktor1, Term):-
    parse_symbol(*), !,
    parse_faktor(Faktor2),
    parse_term(Faktor1 * Faktor2, Term).
parse_term(Faktor1, Term):-
    parse_symbol(/), !,
    parse_faktor(Faktor2),
    parse_term(Faktor1 / Faktor2, Term).
parse_term(Term, Term).

parse_faktor(Faktor):-
    parse_symbol('('),
    parse_ausdruck(Faktor),
    parse_symbol(')').

parse_faktor(Faktor):-
    parse_zahl(Faktor).

parse_faktor(Variable):-
    parse_variable(Variable).

```



```

parse_zahl(Zahl):-
    symbol(Zahl),
    integer(Zahl),
    next_symbol.

parse_variable(Bezeichner):-
    symbol(bez(Bezeichner)),
    next_symbol.

parse_symbol(Symbol):-
    symbol(Symbol),
    next_symbol.
    
```

20.3.3 Interpreter für mini-PASCAL

Die Interpretation eines Programms in mini-PASCAL ist eine einfache Angelegenheit, da die eigentlich schwierige Arbeit, der Aufbau des Parsebaums schon

Verwaltung des
Variablenspeichers

geleistet ist. Im Rahmen der Interpretation müssen Variablenwerte verwaltet werden. Dies geht durch Eintragung von Variablenwerten in die Prolog-Wissensbasis. Wir benutzen dazu den Funktor *speicher*. Die Klausel *speicher(a, 7)* bedeutet, daß a den Wert 7 hat.

Interpreter für mini-
PASCAL

```

interpret_programm(Datei):-
    parse_programm(Datei, Parsebaum),
    retractall(speicher(_, _)),
    interpret_anweisungen(Parsebaum), !,
    listing(speicher).

interpret_anweisungen([Anweisung|Anweisungen]):-
    interpret_anweisung(Anweisung),
    interpret_anweisungen(Anweisungen).
interpret_anweisungen([]).

interpret_anweisung(zuweisung(Variable, Ausdruck)):-
    interpret_ausdruck(Ausdruck, Wert),
    retractall(speicher(Variable, _)),
    asserta(speicher(Variable, Wert)).

interpret_anweisung(while(Ausdruck, Anweisung)):-
    interpret_ausdruck(Ausdruck, true), !,
    
```

```

    interpret_anweisung(Anweisung),
    interpret_anweisung(while(Ausdruck, Anweisung)).
interpret_anweisung(while(_, _)).

interpret_anweisung(begin(Anweisungen)):-
    interpret_anweisungen(Anweisungen).

interpret_anweisung(write(Ausdruck)):-
    interpret_ausdruck(Ausdruck, Wert),
    write(Wert), nl.

interpret_anweisung(nil).

interpret_ausdruck(Ausdruck1 < Ausdruck2, true):-
    interpret_ausdruck(Ausdruck1, Wert1),
    interpret_ausdruck(Ausdruck2, Wert2),
    Wert1 < Wert2.
interpret_ausdruck(_Ausdruck1 < _Ausdruck2, false).

interpret_ausdruck(Ausdruck1 + Ausdruck2, Wert):-
    interpret_ausdruck(Ausdruck1, Wert1),
    interpret_ausdruck(Ausdruck2, Wert2),
    Wert is Wert1 + Wert2.

interpret_ausdruck(Ausdruck1 - Ausdruck2, Wert):-
    interpret_ausdruck(Ausdruck1, Wert1),
    interpret_ausdruck(Ausdruck2, Wert2),
    Wert is Wert1 - Wert2.

interpret_ausdruck(Ausdruck1 * Ausdruck2, Wert):-
    interpret_ausdruck(Ausdruck1, Wert1),
    interpret_ausdruck(Ausdruck2, Wert2),
    Wert is Wert1 * Wert2.

interpret_ausdruck(Ausdruck1 / Ausdruck2, Wert):-
    interpret_ausdruck(Ausdruck1, Wert1),
    interpret_ausdruck(Ausdruck2, Wert2),
    Wert is Wert1 / Wert2.

interpret_ausdruck(Variable, Wert):-
    speicher(Variable, Wert), !.

interpret_ausdruck(Zahl, Zahl):-
    integer(Zahl), !.

```

20.3.4 Übersetzer für mini-PASCAL

Der Interpreter kann in einen Compiler umgeschrieben werden, der ein mini-PASCAL-Programm in die Assemblersprache von Intel-Prozessoren über-

Ergebnis von
Ausdrücken im
ax-Register

setzt.

Zur Übersetzung von Anweisungen muß man sich Gedanken machen wie man mit Ausdrücken und Kontrollstrukturen umgeht? Wir vereinbaren, daß das Ergebnis übersetzter Ausdrücke stets im ax-Register des Zielprozessors

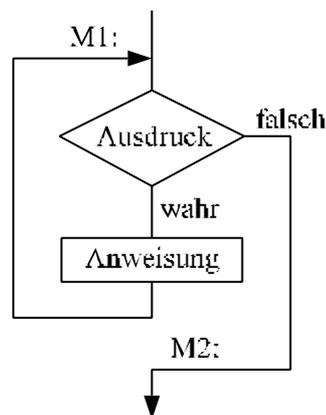
Code-Schablone

steht. Damit kann man die Wertzuweisung übersetzen.

Zur Übersetzung der While-Anweisung benötigt man eine Code-Schablone, Marken und Sprünge. Anschaulich wird der Übergang zur Code-Schablone, wenn man zuerst ein Ablaufdiagramm für die While-Anweisung

Abb. 20-9
Ablaufdiagramm für
eine While-
Anweisung

entwirft:



bedingte und unbedingte Sprünge

Ein Pfeil beginnt am Ausgang *falsch* des Ausdrucks. Dieser Pfeil wird in einen *bedingten* Sprung übersetzt. Der andere Pfeil beginnt bei der Anweisung. Er

wird in einen *unbedingten* Sprung übersetzt. Die Pfeile sind mit Marken versehen, welche die Sprungzeile bezeichnen.

Die Code-Schablone der While-Anweisung sieht so aus:

<pre> M1: <Code für den Ausdruck> cmp ax, true jnz M2 <Code für die Anweisung> jmp M1 M2: </pre>	<p>Code-Schablone der While- Anweisung</p>
--	--

Die Verwaltung der Sprungmarken delegieren wir an die Prädikate *init_compiler* und *gib_marke*:

Verwaltung der
Sprungmarken

```

init_compiler:-
  retractall(marke(_)),
  asserta(marke(1)).

gib_marke(Marke):-
  retract(marke(Marke)),
  Marke2 is Marke + 1,
  asserta(marke(Marke2)).

```

Damit lassen sich die Anweisungen übersetzen:

Übersetzen von
Anweisungen

```

compile_programm(Datei):-
  parse_programm(Datei, Parsebaum),
  retractall(marke(_)),
  asserta(marke(1)),
  compile_anweisungen(Parsebaum).

gib_marke(Marke):-
  retract(marke(Marke)),
  Marke2 is Marke + 1,
  asserta(marke(Marke2)).

compile_anweisungen([Anweisung|Anweisungen]):-
  compile_anweisung(Anweisung), !,
  compile_anweisungen(Anweisungen).

```

```

compile_anweisungen([]).

compile_anweisung(zuweisung(Variable, Ausdruck)):-
    compile_ausdruck(Ausdruck),
    schreibnl('mov ', Variable, ', ax').

compile_anweisung(while(Ausdruck, Anweisung)):-
    gib_marke(Marke1),
    schreib_marke(Marke1),
    compile_ausdruck(Ausdruck),
    schreibnl('cmp ax, true'),
    gib_marke(Marke2),
    schreibnl('jnz M', Marke2),
    compile_anweisung(Anweisung),
    schreibnl('jmp M', Marke1),
    schreib_marke(Marke2).

compile_anweisung(begin(Anweisungen)):-
    compile_anweisungen(Anweisungen).

compile_anweisung(write(Ausdruck)):-
    compile_ausdruck(Ausdruck),
    schreibnl('out ax').

compile_anweisung(nil).

```

Vergleichsausdrücke werden so übersetzt, daß Sie als Ergebnis *True* oder *False* liefern. Dies erleichtert die Übersetzung der Kontrollstrukturen. Kann ein Zwischenergebnis nicht direkt verwendet werden, so legt man es mittels *push* auf den Kellerspeicher ab und holt es mittels *pop* bei Bedarf wieder zurück.

```
compile_ausdruck(Ausdruck1 < Ausdruck2):-
  compile_ausdruck(Ausdruck1),
  schreibnl('push ax'),
  compile_ausdruck(Ausdruck2),
  schreibnl('pop cx'),
  schreibnl('cmp ax, cx'),
  schreibnl('mov ax, true'),
  gib_marke(Marke),
  schreibnl('jl M', Marke),
  schreibnl('mov ax, false'),
  schreib_marke(Marke).
```

Übersetzung von
Ausdrücken

```
compile_ausdruck(Ausdruck1 + Ausdruck2):-
  compile_ausdruck(Ausdruck2),
  schreibnl('push ax'),
  compile_ausdruck(Ausdruck1),
  schreibnl('pop cx'),
  schreibnl('add ax, cx').
```

```
compile_ausdruck(Ausdruck1 - Ausdruck2):-
  compile_ausdruck(Ausdruck2),
  schreibnl('push ax'),
  compile_ausdruck(Ausdruck1),
  schreibnl('pop cx'),
  schreibnl('sub ax, cx').
```

```
compile_ausdruck(Ausdruck1 * Ausdruck2):-
  compile_ausdruck(Ausdruck2),
  schreibnl('push ax'),
  compile_ausdruck(Ausdruck1),
  schreibnl('pop cx'),
  schreibnl('imul ax, cx').
```

```
compile_ausdruck(Ausdruck1 / Ausdruck2):-
  compile_ausdruck(Ausdruck2),
  schreibnl('push ax'),
  compile_ausdruck(Ausdruck1),
  schreibnl('pop cx'),
  schreibnl('idiv ax, cx').
```

```
compile_ausdruck(- Ausdruck):-
  compile_ausdruck(Ausdruck), !,
  schreibnl('neg ax').
```



```
compile_ausdruck(Variable):-  
    schreibnl('mov ax, ', Variable).
```

Einen schönen Ausdruck besorgen das *schreibnl*- und das *schreibmarke*-Prädikat.

Ausgabe- formatierung

```
schreibnl(Ausgabe):-  
    tab(4), write(Ausgabe), nl.
```

```
schreibnl(Ausgabe1, Ausgabe2):-  
    tab(4), write(Ausgabe1), write(Ausgabe2), nl.
```

```
schreibnl(Ausgabe1, Ausgabe2, Ausgabe3):-  
    tab(4), write(Ausgabe1), write(Ausgabe2),  
    write(Ausgabe3), nl.
```

```
schreib_marke(Marke):-  
    write('M'), write(Marke), write(':'), nl.
```

Als Beispiel ist im folgenden die Übersetzung eines Programms zur Fakultätsberechnung angegeben:

Berechnung
der Fakultät

```

program Fakultaet;
begin
  n:= 7;
  i:= 1;
  Fak:= 1;
  while i < n do begin
    i:= i + 1;
    Fak:= Fak * i
  end;
  write(Fak)
end.

```

Übersetzung
des Fakultäts-
programms

```

?- compile_programm('fakultae.pas').
   mov ax, 7
   mov n, ax
   mov ax, 1
   mov i, ax
   mov ax, 1
   mov Fak, ax
M1:
   mov ax, i
   push ax
   mov ax, n
   pop cx
   cmp ax, cx
   mov ax, true
   jl M2
   mov ax, false
M2:
   cmp ax, true
   jnz M3
   mov ax, 1
   push ax
   mov ax, i
   pop cx
   add ax, cx
   mov i, ax
   mov ax, i
   push ax

```

```
    mov ax, Fak
    pop cx
    imul ax, cx
    mov Fak, ax
    jmp M1
M3:
    mov ax, Fak
    out ax
```

20.4 Aufgaben

- 1a) Entwerfen Sie Syntaxdiagramme für römische Zahlen.
- b) Entwickeln Sie auf der Basis der Syntaxdiagramme einen Interpreter für römische Zahlen.

2. Der Sprachumfang von mini-PASCAL kann erweitert, bei Bedarf auch reduziert werden. Zum Reduzieren läßt man einfach einige Syntaxdiagramme weg. Erweitern kann man bei den Ausdrücken und Anweisungen.
 - a) In mini-PASCAL sollen Vorzeichen bei Variablen und Zahlen möglich sein.
 - b) Ergänzen Sie die Fallunterscheidung If-Then-Else.

- 3a) Entwickeln Sie einen Parser für Präfix-Terme aus Ziffern, Buchstaben und den Operatoren +, -, *, /.
- b) Entwickeln Sie einen Übersetzer von Präfix- nach Postfix-Termen.

4. Sofern ein Plotter zur Verfügung steht: Entwickeln Sie einen Parser und Interpreter für *mini-PLOT*.

5. Sofern Fischertechnik-Modelle zur Verfügung stehen: Entwickeln Sie einen Parser, Interpreter und Übersetzer für *mini-FISCH*.

21 Maschinelle Sprachverarbeitung

Wir knüpfen an das in Kapitel 16 eingeführte Beispiel zur Grammatik deutscher Sätze an. Die Frage nach der zu dieser Grammatik gehörenden Sprache haben wir mit dem Prädikat *erzeugteSprache* beantwortet. Die effiziente Lösung des *Wortproblems*, im Beispiel besser *Satzproblem* genannt, ob ein gegebener Satz aus dem Startsymbol abgeleitet werden kann, können wir mit dem Kellerautomaten als theoretischem Rüstzeug neu angehen. Die bisherige Lösung des sturen Durchprobierens mittels Backtracking soll durch einen zielgerichteten Algorithmus auf der Basis eines nichtdeterministischen Kellerautomaten ersetzt werden.

Der Erkennungsalgorithmus wird anschließend zu einem Parser ausgebaut, der zu einem syntaktisch korrekten Satz den zugehörigen Ableitungsbaum erstellt. Auf der Basis des Ableitungsbaums können syntaktische Interpretation eines Satzes stattfinden. Wir werden einen Aussagesatz in einen Fragesatz beziehungsweise Nebensatz umformulieren und in den Übungen weitere syntaktische Umformulierungen kennenlernen.

Generierung des
Ableitungsbaumes

21.1 Wortproblem - nichtdeterministische Kellerautomaten

Ein effizienter Erkennungsalgorithmus wählt bei jedem Ableitungsschritt eine Produktion aus, die aus dem Kopfsymbol der aktuellen Ableitung das Kopfterminal des zu analysierenden Satzes erzeugt. Falls das nicht möglich ist, wird nichtdeterministisch eine andere Produktion für das Kopfsymbol der Ableitung ausgewählt.

effizienter Erkennungsalgorithmus

Wir konzipieren ein zweistelliges Prädikat *ableitbar2(+Ableitung, +Wort)*. Im ersten Argument wird die aktuelle Ableitung übergeben, im zweiten der zu erkennende Satz. Die Implementierung wird nach der Kopf-Rest-Methode vorgenommen, da die Ableitung eine Liste aus Terminalen und Variablen und der Satz eine Liste aus Terminalen ist. Bei jedem Ableitungsschritt sind fünf Fälle zu unterscheiden.

Schnittstelle des
Prädikats
ableitbar2

Fall 1: Stimmt das Kopfsymbol *KopfA* der aktuellen Ableitung mit dem Kopfsymbol *KopfW* des zu erkennenden Satzes überein, so muß nur noch mit den beiden Restlisten weiter analysiert werden. Die aktuelle Ableitung und der Satz werden also jeweils um das erste Symbol gekürzt.

Fall 1: übereinstimmende
Kopfsymbole

```
ableitbar2([KopfW|RestA], [KopfW|RestW]):-  
    !, ableitbar2(RestA, RestW).
```

Fall 2: Wenn die Kopfsymbole KopfA und KopfW nicht übereinstimmen, prüfen wir, ob es eine Produktion gibt, die direkt aus KopfA den Kopf des zu erkennenden Satzes herleitet. Für die weitere Analyse wird nur noch der Restsatz RestW gebraucht. Es darf allerdings nicht mit der Restliste RestA der aktuellen Ableitung weiter gearbeitet werden. Vielmehr muß zunächst der unberücksichtigte Teil Rest1 der Produktion mittels *append* gekellert werden, denn hiermit muß die Ableitung fortgesetzt werden. An dieser Stelle wird deutlich, daß wir einen Kellerautomaten zum Erkennen einer kontextfreien Sprache einsetzen.

```

Fall 2:
Kopf des Satzes
ist ableitbar
ableitbar2([KopfA|RestA], [KopfW|RestW]):-
    produktion(KopfA, [KopfW|Rest1]),
    append(Rest1, RestA, Rest2), /* einkellern! */
    ableitbar2(Rest2, RestW).

```

Fall 3: Läßt sich aus dem Kopf der aktuellen Ableitung KopfW nicht direkt herleiten, so wählen wir eine anwendbare Produktion für KopfA aus. Der Regelrumpf Rest1 wird vor der Restliste RestA eingekellert, und mit dieser neuen Ableitungsliste wird versucht, den alten Satz abzuleiten. Die Auswahl einer Produktion muß nicht zum Ziel führen. In diesem Fall wird mittels Backtracking die nächste anwendbare Produktion ausgewählt. Auf diese Weise wird aus dem deterministischen ein nichtdeterministischer Kellerautomat.

Wer mit dieser Interpretation Probleme hat, möge sich vorstellen, daß der Kellerautomat nicht mehrere Versuche macht, die richtige Produktion zu finden, sondern bei der ersten Auswahl einer Produktion gleich die richtige rät, wie es sich für einen ordentlichen nichtdeterministischen Kellerautomaten gehört.

```

Fall 3:
Auswahl einer
anwendbaren
Produktion
ableitbar2([KopfA|RestA], [KopfW|RestW]):-
    produktion(KopfA, [Kopf1|Rest1]),
    Kopf1 \= KopfW,
    append([Kopf1|Rest1], RestA, Rest2),
    ableitbar2(Rest2, [KopfW|RestW]).

```

Fall 4: Eine ϵ -Produktion erzeugt aus einer Variablen das leere Wort. Hierfür brauchen wir eine eigene Klausel:

```
Fall 4:      ableitbar2([KopfA|RestA], Wort):-  
ε-Produktion  produktion(KopfA, []),      /* Epsilon-Produktion */  
              ableitbar2(RestA, Wort).
```

Fall 5: Gehört ein Satz zur Sprache $L(G)$, so kann durch eine Folge von Ableitungsschritten der zu analysierende Satz nach und nach abgebaut werden. Sind die Wörter des Satzes aufgebraucht und gleichzeitig der Keller leer, so akzeptiert der nichtdeterministische Kellerautomat:

```
ableitbar2([], []).          /* akzeptieren */
```

Fall 5: Terminierung der Analyse

Wir rufen das Prädikat *ableitbar2/2* über das Prädikat *ableitbar2/1* auf:

```
ableitbar2(Wort):-
    terminalwort(Wort),
    startsymbol(Start),
    ableitbar2([Start], Wort).
```

Initialisierung der Analyse

Mit dem neuen Prädikat *ableitbar2* gelingt nun auch der Nachweis, daß ein Satz nicht zu einer Sprache gehört. Zudem arbeitet es wegen der zielgerichteten Auswahl der anzuwendenden Produktionen weitaus effektiver als unsere erste Version. Lediglich bei Linksrekursionen in der zugrundeliegenden Grammatik G mißlingen Ableitungen. Linksableitungen muß man generell durch Umbau der Grammatik vermeiden. Die Theorie beschreibt Verfahren, mit denen dies möglich ist, zum Beispiel Transformation auf Greibach-Normalform [Hop1].

Das Prädikat *ableitbar2* arbeitet als Akzeptor. Als Antwort auf beispielsweise die Anfrage *?- ableitbar2(['Peter', liebt, das, 'Mädchen'])* erhält man lediglich die Antwort *yes*. Um die einzelnen Ableitungsschritte auch verfolgen zu können, ergänzt man am einfachsten die folgende Klausel. Sie muß in der Wissensbasis vor allen anderen *ableitbar2*-Klauseln stehen.

```
ableitbar2(Ableitung,Wort):-
    write(Ableitung), write('->'), write(Wort), nl, fail.
```

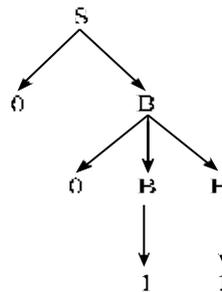
Anzeige der Ableitungsschritte

21.2 Ableitungsbäume und Parser

Ein Akzeptor reicht in vielen Fällen nicht aus. Zur weiteren Verarbeitung eines syntaktisch korrekten Satzes benötigt man dessen grammatische Struktur. Sie wird durch den sogenannten *Ableitungsbaum* beschrieben.

Der Ableitungsbaum zeigt in graphischer Form eine Ableitung an. Die Wurzel des Ableitungsbaumes ist das Startsymbol. Innere Knoten sind durch Variable markiert, die Blätter durch Terminale der zugrundeliegenden Grammatik. Die Anwendung einer Produktion erzeugt die Nachfolgeknoten eines inneren Knotens. Für das Beispiel der 0-1-Wörter aus Kapitel 16.1.4 ist im Bild der Ableitungsbaum für die Ableitung $S \rightarrow 0B \rightarrow 00BB \rightarrow 001B \rightarrow 0011$ angegeben.

Abb. 21-1
Ableitungsbaum für
das Wort 0011



Der bestehende Akzeptor soll nun zu einem Parser ausgebaut werden, der als Ergebnis der syntaktischen Analyse einen Ableitungsbaum liefert. Dazu ergänzen wir die Parameterliste um zwei weitere Parameter:

Schnittstelle
des Prädikats
ableitbar3

```
ableitbar3(+Ableitung,+Wort,-Restwort,-Ableitungsbaum).
```

Die ersten beiden Argumente übernehmen wir aus *ableitbar2*. Im letzten Argument soll als Ergebnis einer Ableitung der Ableitungsbaum geliefert werden. Das dritte Argument wird benötigt, weil wir die rein endrekursive Implementierung von *ableitbar2* durch doppelte Rekursion ersetzen müssen; einen Rekursionsschritt für den Kopf und einen Rekursionsschritt für den Rest der Ableitungsliste. Die rekursive Ableitung des Kopfes verbraucht Terminale des Wortes, was übrigbleibt wird in Restwort zurückgeliefert.

Generierung des
Ableitungsbaums
im Bottom-Up-
Verfahren

Der Ableitungsbaum wird nach dem Bottom-Up-Verfahren sukzessive aus Blättern und Teilbäumen zusammengesetzt. Zur Erläuterung des Verfahrens betrachten wir zwei Beispiele zur deutschen Grammatik.

Durch eine einfache Ableitung kann aus der Variablen *artikel* das Terminal *das* erzeugt werden. Der zugehörige Teilbaum wird in Prolog durch die Struktur *artikel(das)* repräsentiert. Zum Aufbau dieser Struktur benutzen wir den *univ*-Operator (=..), welcher aus der Liste [*artikel*, *das*] die gewünschte Struktur aufbaut.

In der Substantivgruppe *das Mädchen* ist *das* der Artikel und *Mädchen* das Substantiv. Aus den beiden Teilbäumen *artikel(das)* und *substantiv(Mädchen)* wird die Substantivgruppe zusammengesetzt. Der *univ*-Operator wird dazu auf die Liste [*substantivgruppe*, *artikel(das)*, *substantiv(Mädchen)*] angewendet und erzeugt daraus die Struktur *substantivgruppe(artikel(das), substantiv(Mädchen))*.

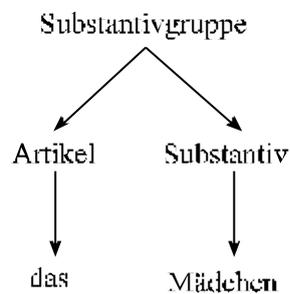
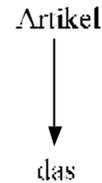


Abb. 21-2
Ableitungsbaum
einer Substantiv-
gruppe

Im folgenden ist die Implementierung des Parsers *ableitbar3* angegeben. Der markanteste Unterschied zu *ableitbar2* besteht darin, daß die Ableitung des Kopfes nicht explizit gekellert, sondern implizit rekursiv behandelt wird. Dies ist notwendig, um die Struktur des Ableitungsbaumes herstellen zu können. Beim Einkellern wird hingegen die grammatische Struktur in eine Liste linearisiert.

Unterschiede
zwischen
ableitbar2 und
ableitbar3

```
ableitbar3(Wort, Ableitungsbaum):-
  terminalwort(Wort),
  startsymbol(Start),
  ableitbar3([Start], Wort, [], [Ableitungsbaum]).
```

Implementierung
eines Parsers, der
Ableitungsbäume
generiert

```
ableitbar3([Kopf|RestA], [Kopf|RestW], RestW1, [Kopf|Baum]):-
  ableitbar3(RestA, RestW, RestW1, Baum).
```

```
ableitbar3([KopfA|RestA], [KopfW|RestW], RestW1, [Kopf|Baum2]):-
  produktion(KopfA, [KopfW|Rest1]),
  ableitbar3(Rest1, RestW, Rest2, Baum1),
  Kopf=..[KopfA, KopfW|Baum1],
  ableitbar3(RestA, Rest2, RestW1, Baum2).
```

```
ableitbar3([KopfA|RestA], [KopfW|RestW], RestW1, [Kopf|Baum2]):-
  produktion(KopfA, [Kopf1|Rest1]),
  Kopf1 \= KopfW,
  ableitbar3([Kopf1|Rest1], [KopfW|RestW], Rest2, Baum1),
```

```
Kopf=..[KopfA|Baum1],  
ableitbar3(RestA, Rest2, RestW1, Baum2).
```

```

ableitbar3([KopfA|RestA], Wort, RestW1, [eps|Baum]):-
    produktion(KopfA, [], /* Epsilon-Produktion */
    ableitbar3(RestA, Wort, RestW1, Baum).

ableitbar3([], Wort, Wort, []).

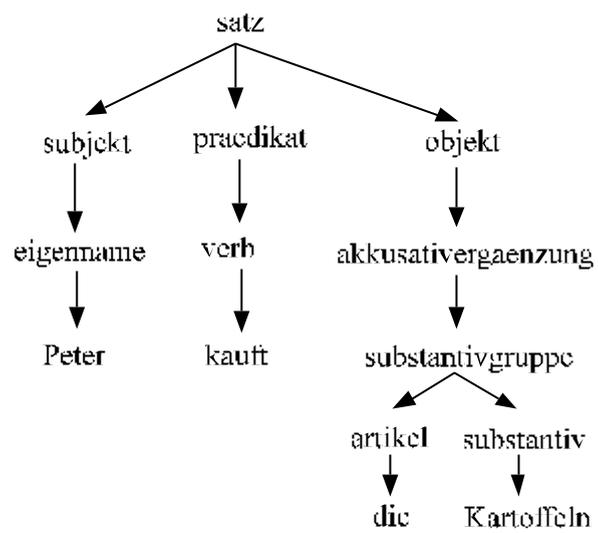
```

Stellen wir mit unserer Grammatik für arithmetische Terme die Anfrage `?- ableitbar3([a,+,b,*,c],B)`, so erhalten wir den folgenden Ableitungsbaum als Lösung: $B = \text{ausdruck}(\text{term}(\text{faktor}(a)), +, \text{ausdruck}(\text{term}(\text{faktor}(b)), *, \text{term}(\text{faktor}(c))))$. Diesen unansehlichen Baum können wir mit der Term-Visualisierung von ProVisor übersichtlich darstellen:

Abb. 21-3
Ableitungsbaum für
den Term $a+b*c$

Die Anfrage `?- ableitbar3(['Peter', kauft, die, 'Kartoffeln'], B)` erzeugt den Ableitungsbaum $B = \text{satz}(\text{subjekt}(\text{eigennamen}(\text{'Peter'})), \text{praedikat}(\text{verb}(\text{kauft})), \text{objekt}(\text{akkusativergaenzung}(\text{substantivgruppe}(\text{artikel}(\text{die}), \text{substantiv}(\text{'Kartoffeln'}))))))$:

Abb. 21-4
Ableitungsbaum für
den Satz Peter
kauft die Kartoffeln



21.3 Verarbeitung natürlicher Sprache

Als kleine Anwendung betrachten wir die maschinelle Verarbeitung natürlicher Sprache. Dem Grammatik-Duden [Gra1] entnehmen wir, daß ein Verb an genau drei Stellen im Satz stehen kann: an erster Stelle bei Frage- und Aufforderungssätzen, an zweiter Stelle bei Aussagesätzen und an dritter Stelle bei Nebensätzen. Ordnen wir Subjekt, Prädikat und Objekt entsprechend um, so können wir aus Aussagesätzen leicht Frage- und Nebensätze machen:

```
fragesatz (Satz):-
    ableitbar3(Satz, satz(S, P, O)),
    schreibestruktur(satz(P, S, O)),
    write('?'), nl.
```

Fragesatz bilden

```
nebensatz(Satz):-
    ableitbar3(Satz, satz(S, P, O)),
    write('... weil '),
    schreibestruktur(satz(S, O, P)),
    nl.
```

Nebensatz bilden

Wir können den in einem Ableitungsbaum gespeicherten Satz dadurch anzeigen, daß wir die Blätter des Ableitungsbaumes ausgeben. Am einfachsten erkennt man sie mit dem Systemprädikat *atom*. Innere Knoten zerlegen wir mit Hilfe des *univ*-Operators in eine Liste. Der Kopf dieser Liste ist der Bezeichner des inneren Knotens, der Rest der Liste enthält alle Nachfolgeknoten, welche der Reihe nach ausgegeben werden.

```
schreibestruktur(X):-
    atom(X), write(X), tab(1).
schreibestruktur(X):-
    X =.. [_|Rest],
    schreibeliste(Rest).

schreibeliste([Kopf|Rest]):-
    schreibestruktur(Kopf),
    schreibeliste(Rest).
schreibeliste([]).
```

Blätter des
Ableitungsbaums
ausgeben

Im Aufgabenkapitel finden Sie Vorschläge, wie Sie die Verarbeitung natürlicher Sprache vertiefen können. An dieser Stelle sollen zunächst noch einige Anmerkungen zu diesem interessanten Themenkreis gemacht werden.

Im Anhang B der GI-Empfehlungen für das Fach Informatik in der Sekundarstufe II allgemeinbildender Schulen [GI1] wird ein Unterrichtsprojekt *Maschinelle Sprachverarbeitung* vorgeschlagen. Die dort angesprochenen

GI-Empfehlungen

Techniken zur Sprachverarbeitung sind durch die obige Darstellung im wesentlichen abgedeckt.

Ein alternativer Ansatz wird im *Arbeitsbuch PROLOG* [Göh1] vorgestellt. Der dortige Parser ist speziell auf die Sprachverarbeitung zugeschnitten. Er arbeitet ineffizient nach der Methode des Generierens und Testens durch Anwendung von `append`-Aufrufen. Dafür hat er den Vorteil, daß man leicht kontextbezogene Abhängigkeiten einbauen kann. Beispielsweise müssen in einer Substantivgruppe Artikel und Substantiv im Geschlecht übereinstimmen. In unsere kontextfreie Grammatik läßt sich dies nur schlecht einbauen. Ein Ansatz wäre die Variable `artikel` durch die drei Artikel `maenn_artikel`, `weib_artikel`, `saech_artikel` zu ersetzen und genauso zwischen männlichen, weiblichen und sächlichen Substantiven zu unterscheiden. Eine Substantivgruppe würde dann aus einem gleichgeschlechtlichen Paar von Artikel und Substantiv bestehen. Es ist leicht ersichtlich, daß damit die Grammatik ziemlich aufgebläht wird. Ein anderer Ansatz besteht im *Entfalten* des Parsers, um den speziellen Bedürfnissen der Sprachverarbeitung besser gerecht werden zu können.

kontextbezogene
Anforderungen

Vor dem gleichen Problem stehen Compiler höherer Programmiersprachen. Die Syntax läßt sich sehr schön durch eine kontextfreie Grammatik beschreiben. Aber es gibt auch kontextbezogene Anforderungen. Beispielsweise darf in einem Ausdruck nur dann eine Variable benutzt werden, wenn sie zuvor deklariert wurde. Grundsätzlich können solche kontextbezogenen Anforderungen in die kontextfreie Grammatik aufgenommen werden, dabei würde allerdings der Umfang der Grammatik gewaltig zunehmen. Zur Lösung des Problems werden die Parser deshalb um die Behandlung kontextbezogener Anforderungen speziell ergänzt.

allgemeiner Parser

Unser Parser ist bewußt so allgemein gehalten, daß er Grammatiken für natürliche Sprachen, Programmiersprachen und formale Sprachen, wie Sie üblicherweise in Beispielen der Theoretischen Informatik vorkommen, verarbeiten kann. Auf diese Weise wird der Zugang zu Fragestellungen der Theoretischen Informatik erleichtert. Dies wird insbesondere dadurch deutlich, daß der Parser als nichtdeterministischer Kellerautomat kontextfreie Sprachen akzeptiert.

Grammatik-
Operator -->

Wollen Sie die Verarbeitung natürlicher Sprache vertiefen, so sollten Sie einen weiteren Zugang in Betracht ziehen. In besseren Prolog-Interpretern, wie zum Beispiel TV-SWI-Prolog, können Sie mit dem Grammatik-Operator --> Produktionen direkt definieren und erhalten automatisch einen Parser für ihre Grammatik. Zudem können die Produktionen um Kontextbedingungen ergänzt werden. Dies sind nahezu ideale Voraussetzungen zur Sprachverarbeitung. Die Nutzung dieser Werkzeuge setzt allerdings das Verständnis von Differenzlisten

voraus. Es ist unklar, ob in Informatik-Grundkursen mit diesem Instrumentarium gearbeitet werden kann.

Als kleines Beispiel hierzu betrachten wir die Grammatik für eine Substantivgruppe in der Schreibweise mit Grammatik-Operator:

```
substantivgruppe --> artikel, substantiv.
artikel --> [das]; [die]; [der].
substantiv --> ['Buch']; ['Mädchen']; ['Kartoffeln'].
```

Beispiel für den Einsatz des Grammatik-Operators

Die Anfrage `?- substantivgruppe([der, 'Buch'],[])` wird mit *yes* beantwortet. Wir ergänzen die Grammatik um die Kontextbedingung, daß das Geschlecht von Artikel und Substantiv übereinstimmen müssen:

```
substantivgruppe --> artikel(G), substantiv(G).
artikel(G) --> [das], {G=saech}; [die], {G=weib};
               [der], {G=maenn}.
substantiv(G) --> ['Buch'], {G=saech}; ['Mädchen'], {G=weib};
                  ['Kartoffeln'], {G=weib}.
```

Grammatik mit Kontextbedingungen

Weil *der* männlich und *Buch* sächlich ist, wird obige Anfrage jetzt mit *no* beantwortet. Damit sollen die Betrachtungen zum Grammatik-Operator beendet sein. Weitergehende Informationen findet man zum Beispiel in [Clo1].

21.4 Computerlinguistik im Unterricht

In [Bay1] stellt K. Bayer das Prolog-Programm SIMPEL vor, das in der Lage ist, mit einem Menschen in deutscher Sprache über eine extrem begrenzte künstliche Welt zu kommunizieren. Mit diesem Programm soll gezeigt werden, wie Schülerinnen und Schüler durch die Entwicklung eines *sprechenden Automaten* Einsichten in die Natur sprachlicher Kommunikation und in die Struktur der Sprache gewinnen können.

Auf einem Tisch befinden sich vier Gegenstände: ein Hut, ein Kasten, ein großer Block und ein kleiner Block. Diese Gegenstände können angehoben werden, so daß sie oberhalb des Tisches schweben. Darüber hinaus können sie mit gewissen Einschränkungen aufeinandergetürmt, in den Kasten gelegt oder auf den Tisch zurückgelegt werden.

Abb. 21-5
Bildschirmmaske
von SIMPEL

Der Benutzer kann Befehle erteilen oder Fragen stellen. Beispiele:

Kommunikation mit
SIMPEL

lege den hut in den kasten !
hebe einen block !
lege etwas auf den tisch !
was liegt auf dem tisch ?
steht der kasten auf dem tisch ?
wo ist der kasten

SIMPEL versucht zunächst die Wörter eines eingegebenen Satzes zu erkennen und erzeugt daraus eine Wortliste (Scanner: lexikalische Analyse). Aus *wo steht der Kasten?* wird *[wo, steht, der, Kasten, ?]*

Auf der Basis einer Grammatik versucht SIMPEL im nächsten Schritt den Satz syntaktisch zu analysieren (Parser: syntaktische Analyse). Typische Grammatikregeln lauten:

```
SATZ      -> PRÄDIKAT + ERGÄNZUNG + ERGÄNZUNG
PRÄDIKAT  -> VERB
ERGÄNZUNG -> ARTIKEL + NOMEN
ERGÄNZUNG -> PRÄPOSITION + ARTIKEL + NOMEN
VERB      -> ist | lege | liegt | steht | stelle | hebe
ARTIKEL   -> der | den | dem | ein | einem | einen
```

Grammatikregeln
von SIMPEL

Die semantische Analyse (Interpreter) ordnet mittels der analysierten Kasus und Präpositionen einem Satz einen syntaktisch-semantischen Rahmen zu:

```
ablegen([def, system, AGENT], [def, hut, OBJEKT],
        [def, Kasten, ZIEL]).
```

syntaktisch-
semantischer
Rahmen

Zur Ausführung des Befehls wird der Rahmen auf *ablegen(system, hut, Kasten)* reduziert. In der Wissensbasis gibt es dann zum Prädikat *ablegen/3* Regeln, welche das Ablegen von Gegenständen beschreiben.

```
ablegen(system,_,hut):-
    print(' REAKTION: auf dem hut kann nichts abgelegt
          werden!'),nl,! ,fail.
```

Ausführung eines
Befehls

Abb. 21-6
 Bearbeitung der
 Frage *Wo ist er?*

Bewertung des
 sprechenden
 Automaten

SIMPEL hat kein Bewußtsein. Für das Programm ist jedes Wissen lediglich eine Menge von Symbolen und Regeln zu ihrer Manipulation. Beantwortet das Programm eine Frage, so überprüft es das Vorhandensein und die Struktur bestimmter Zeichenketten im Speicher; befolgt es einen Befehl, so tilgt es bestimmte Zeichenketten und erzeugt neue. SIMPEL reagiert ausschließlich auf der Basis von Vorschriften und ist unfähig zu einer über die engen Grenzen seiner Welt hinausgehenden Reflexion.

Wichtige Unterschiede zwischen Mensch und Computer springen ins Auge: Der Mensch ist ein soziales Lebewesen mit Gefühlen, Bewußtsein, Intentionen und werthaften Präferenzen. Der Mensch lebt und handelt in einer Welt, die er - anders als SIMPEL - nur zu einem sehr kleinen Teil selbst erschafft und kontrolliert. Der Mensch setzt sich seine Ziele, der Computer wird durch seinen Programmierer gesteuert. Der Mensch handelt auf der Basis eines sich ständig erweiternden, in seinem Gedächtnis gespeicherten Erfahrungsschatzes, SIMPEL dagegen erinnert sich an nichts.

Das Programm SIMPEL ist trotz seines Namens relativ komplex. Zur Demonstration eines *sprechenden Automaten* ist es gut geeignet. Will man Programmiererweiterungen durchführen, muß man allerdings über solide Prolog-Kenntnisse verfügen. Mit weit weniger Aufwand können Schüler ELIZA-Programme analysieren und Programmergänzungen durchführen.

mangelndes
 Alltagswissen

Viele Probleme natürlichsprachlicher Systeme resultieren daher, daß diese Systeme über kein Alltagswissen verfügen. Zum Verstehen des Satzes: *Als ich nach Hause kam, knabberte eine Maus an meiner neuen Maus.* gehört das Erkennen der unterschiedlichen Bedeutung von *Maus*. Mit den bekannten

Verfahren der Syntaxanalyse unter Bezugnahme auf eine Grammatik ist hier nichts zu machen.

Mit dem Projekt *Cyc* [Sull] versucht man seit 1984 das Alltagswissen auf den Rechner zu bekommen. Das Wissen wird in Form von Mikrotheorien gespeichert, zum Beispiel zum Kaufen. Mittlerweile ist die Phase der Gehirnschirurgie vorbei, die Mitarbeiter sind jetzt eher Nachhilfelehrer als Programmierer. *Cyc* versteht jetzt schon 3/4 der *USA Today*. Zukünftige Betriebssysteme sollen unter Ausnutzung von *Cyc* angemessener als heutige Systeme in Dialog mit den Benutzern treten.

Projekt *Cyc*

21.5 ELIZA

1966 schrieb Weizenbaum sein berühmtes ELIZA. Das Programm zielt darauf ab, eine Unterhaltung zu simulieren. Ein Benutzer gibt einen Satz ein, und ELIZA antwortet mit einer entsprechenden Frage oder einer Bemerkung. ELIZA versteht die Frage nicht, es antwortet auf die Frage, indem es Wortmuster erkennt und unter Verwendung von Antwortmustern erwidert. Um die Antworten glaubhafter zu machen, wird eine Sitzung bei einem Psychiater als Anwendungsbereich angenommen. Eine Beispielsitzung mit ELIZA könnte wie folgt aussehen:

Reaktion auf der Basis von Antwortmustern

```
> i am unhappy.
how long have you been unhappy ?

> six month.
please go on.

> can you help me.
what makes you think i can help you ?

> you remind me of my father and brother.
can you tell me more about father ?
```

Beispielsitzung mit ELIZA

Das Verhalten von ELIZA ließ Leute glauben, daß eine echte Unterhaltung mit einem Psychiater stattfinden würde. Weizenbaums Sekretärin forderte ihn auf, den Raum zu verlassen, um ungestört den Dialog fortführen zu können. Psychiater glaubten, den Computer für Therapiezwecke einsetzen zu können. Weizenbaum war von den Reaktionen der Leute auf ELIZA und auf die KI im allgemeinen entsetzt und schrieb eine leidenschaftliche Bitte, das Programm nicht ernst zu nehmen.

Die hier vorgestellte Version ist eine vereinfachte Fassung von ELIZA.

```

/* eliza
   Simuliert eine Unterhaltung mittels Seiteneffekten.
   in Abwandlung von Sterling, Shapiro: Prolog, S. 247ff
*/

ein ELIZA-
Programm  eliza:-
          write('Hallo, ich bin ELIZA. Wo liegt dein Problem?'),
          nl, eingabe(Eingabe), eliza(Eingabe), !.

          eliza([]):-
            write('Ich hoffe, daß ich Dir helfen konnte. ').

          eliza(Eingabe):-
            muster(Stimulus, Erwiderung),
            vergleiche(Stimulus, Eingabe, [], Woerterbuch),
            write(' '),
            antworte(Erwiderung, Woerterbuch),
            !,
            eingabe(Eingabe2),
            eliza(Eingabe2).

/* vergleiche(+Muster, +Satz, +Woerter, -Woerterbuch)
   Vergleicht ein Satzmuster mit dem eingegebenen Satz
   und nimmt die Platzhalter samt zugehöriger
   Satzteile in das Woerterbuch auf.
*/

vergleiche([N|Muster], Ziel, Woerter, Woerterbuch):-
  integer(N),
  append(LinkesZiel, RechtesZiel, Ziel),
  vergleiche(Muster, RechtesZiel,
             [w(N, LinkesZiel)|Woerter], Woerterbuch).
vergleiche([Wort|Muster],[Wort|Ziel],Woerter,Woerterbuch):-
  atom(Wort),
  vergleiche(Muster, Ziel, Woerter, Woerterbuch).
vergleiche([], [], Woerterbuch, Woerterbuch).

/* antworte(+Erwiderung, +Woerterbuch)
   Gibt die Erwiderung aus und ersetzt dabei die Platz-
   halter durch die im Woerterbuch stehenden Satzteile.
*/

antworte([N|Muster], Woerterbuch):-
  integer(N),
  suche(N, Woerterbuch, Wert),
  schreib(Wert),
  antworte(Muster, Woerterbuch).

```

```
antworte([Wort|Muster], Woerterbuch):-  
    schreib(Wort),  
    antworte(Muster, Woerterbuch).  
antworte([], _Woerterbuch):-  
    nl, nl.
```

```

/* suche(+Schlüssel, +Wörterbuch, -Wert)
   Sucht in der Wörterbuchliste nach einem Paar
   w(Schlüssel, Wert) und gibt Wert aus.
*/

suche(Schlüssel,[w(Schlüssel, Wert)|_Woerterbuch], Wert).
suche(Schlüssel,[w(Schlüssel1,_Wert)|Woerterbuch],Wert):-
  Schlüssel \= Schlüssel1,
  suche(Schlüssel, Woerterbuch, Wert).

/* muster(+Stimulus, -Erwiderung)
   Bestimmt ein auf Stimulus anwendbares Erwiderungsmuster.
*/

muster(['Ich', bin, 1, .],
       ['Wie lange bist Du schon', 1, ?]).
muster(['Du',1,'mich',2],
       ['Wieso glaubts Du, dass ich',2,?]).
muster(['Ich', mag, 1, '.'],
       ['Mag jemand in deiner Familie auch', 1, ?]).
muster(['Ich', glaube, 1, '.'], ['Glaubts Du oft', 1, ?]).
muster([1, X, 2],
       ['Kannst Du mir mehr ueber', X, 'erzählen.']):-
  wichtig(X).
muster([1], ['Erzaehle mir bitte mehr von', 1, .]).

wichtig('Vater').
wichtig('Mutter').
wichtig('Sohn').
wichtig('Tochter').
wichtig('Schwester').
wichtig('Bruder').

schreib([K|R]):-
  schreib(K), schreib(R).
schreib([]).
schreib(Wert):-
  atom(Wert), write(' '), write(Wert).

eingabe(Eingabe):-
  write('> '), readln(Eingabe).

```

Stimulus/Antwort-
Paare

Der Kern von ELIZA besteht aus Stimulus/Antwort-Paaren, welche als Fakten in der Form *muster(Stimulus, Antwort)* repräsentiert werden; hierbei sind Stimulus und Antwort Listen von Wörtern und Zahlen. Jede Zahl steht als Platzhalter für einen Satzteil.

Ich bin 1. Wie lange bist Du schon 1?

Unter Verwendung dieses Paares lautet die Antwort des Programms auf die Eingabe *Ich bin unglücklich. Wie lange bist Du schon unglücklich?*

Zur Erzeugung einer passenden Antwort, wird ein Stimulus/Antwort-Paar aus der Wissensbasis entnommen. Das Prädikat *vergleiche* versucht, den Stimulus mit dem Eingabesatz zu unifizieren. Dabei kommt es wesentlich darauf an, die Platzhalter in Form von Zahlen an Teile des Eingabesatzes zu binden. Alle Möglichkeiten werden durch die nichtdeterministische Verwendung von *append* durchprobiert. Dabei gefundene Bindungen von Platzhaltern an Satzteile werden im Wörterbuch, einer Liste aus Paaren *w(Zahl, Satzteil)*, gespeichert.

Zum Aufbau des Wörterbuchs wird die sogenannte Akkumulatortechnik eingesetzt. Man beginnt mit der leeren Liste und ergänzt am Kopf der Liste die Einträge ins Wörterbuch. Auf diese Weise wird sukzessive das Wörterbuch aufgebaut. Die Akkumulation findet im dritten Argument des Prädikats *vergleiche* statt. Das fertige Wörterbuch wird zum Schluß an die Ausgangsvariable *Woerterbuch* gebunden, dem vierten Argument von *vergleiche*.

Konnte ein Stimulus mit dem Eingabesatz unifiziert werden, so wird anschließend vom Prädikat *antworte* die Antwort ausgegeben. Dazu ersetzt *antworte* im zum Stimulus gehörenden Antwortsatz die Platzhalter durch die zugehörigen Einträge im Wörterbuch.

21.6 Aufgaben

- 1a) Geben Sie eine Grammatik für die Sprache der 0-1-Wörter an, bei denen das rechte Halbwort das Spiegelbild des linken Halbwords ist. Lassen Sie sich zur Kontrolle die erzeugte Sprache ausgeben.
 - b) Warum kann ein deterministischer Kellerautomat diese Sprache nicht erkennen?
2. In dieser Aufgabe erweitern wir unsere Deutsche Grammatik.
 - a) Lassen Sie auch einfache Sätze zu, in denen kein Objekt vorkommt.
 - b) Zwischen Artikel und Substantiv können Adjektive stehen. Mehrere Adjektive werden durch Komma voneinander getrennt.
 - c) Bilden Sie zu einem Aussagesatz den zugehörigen Nachfragesatz. Beispiel: Das Mädchen liest das gute Buch.
wird zu: Wer liest das gute Buch?
 - d) Die Akkusativergänzung erfragt man mit „Wen oder Was“. Beispiel: Peter liest das blonde Mädchen.
Erfragen: Wen oder was liest Peter?
Bilden Sie die Frage nach der Akkusativergänzung.
 - e) Ergänzen Sie die Grammatik so, daß Aussage-, Frage- und Nebensätze erkannt werden.
 - f) In Sätzen sollen Satzteile durch andere Satzteile ersetzt werden. Gesucht ist ein Prädikat *ersetze(+AltBaum, +AltTerm, +NeuTerm, -NeuBaum)*, das alle Vorkommen von *AltTerm* in *AltBaum* durch *NeuTerm* ersetzt und das Ergebnis in *NeuTerm* zurückliefert.
Beispiel: ?- ableitbar3(['Peter',kauft,das,'Buch'], Baum),
 ersetze(Baum, artikel(das), artikel(ein), NeuBaum),
 schreibestruktur(NeuBaum).
gibt „Peter kauft ein Buch“ aus.
 - g) Analysieren Sie das folgende Prädikat:

```

alleverben(Satz):-
  ersetze(Satz, verb(_), verb(Verb), SatzX), !,
  produktion(verb, [Verb]), schreibestruktur(SatzX).

```
- 3a) Ergänze im ELIZA-Programm Stimulus/Antwort-Paare.
 - b) Trenne durch Einsatz der Akkumulortechnik die Verarbeitung und Ausgabe innerhalb des Prädikats *antworte*.

22 Expertensysteme

22.1 Architektur eines Expertensystems

Der Begriff Expertensystem artikuliert einen Anspruch, der interdisziplinär in die Anwendungsgebiete solcher Systeme hineinreicht. Im Anwendungsgebiet befinden sich die menschlichen Experten, an denen ein Expertensystem von seinem Namen her beansprucht, gemessen zu werden.

Expertensysteme werden heute in vielen Anwendungsbereichen eingesetzt:

- Unterstützung von Büroarbeiten
Beispiel: integrierte Dokumenten- und Wissensverwaltung
- Unterstützung der Softwareerstellung
Beispiel: wissensbasierte Suche in Projektbibliotheken
- organisatorische Planung
Beispiel: wissensbasierte Fabriklayoutplanung
- technische Konfiguration
Beispiel: Konfiguration lokaler Netze
- technische Diagnose
Beispiel: Überwachung, Steuerung und Regelung industrieller Systeme und Fehlerdiagnose an CNC-Maschinen
- Beratung in Vermögensfragen
Beispiel: Lebensversicherungsberatung
- Anwendung neuer Werkstoffe
Beispiel: Auswahl geeigneter Bauteile im Maschinenbau
- Medizin
Beispiel: wissensbasierte Entscheidungsfindung in der Anästhesie

Anwendungsbereiche von Expertensystemen

Die Grenzen des Anspruchs von Expertensystemen ist durch die heute verfügbare Technologie von Expertensystemen bestimmt. Relevante Konzepte beziehen sich auf die Wissensakquisition und die Wissensrepräsentation.

Die Wissensbasis wird vom Wissensingenieur und Experten aufgebaut. Das Wissen kann beispielsweise in Form von Fakten, Regeln, semantischen Netzen oder Frames dargestellt sein. Zur Verwaltung großer Wissensbasen nutzt man auch Datenbanksysteme. Das vorgegebene Wissen kann durch fallspezifisches Wissen, das direkt vom Anwender erfragt und durch berechnete Zwischenergebnisse ergänzt werden. Die Aufgabe der Wissenrepräsentation beinhaltet auch stets die Frage, wie effizient Wissen von der Inferenzmaschine genutzt werden kann.

Da sich der menschliche Experte nicht zuletzt durch seine Fähigkeit auszeichnet, sein Wissen auch mitteilen zu können, muß der Erklärungsfähigkeit

Wissensakquisition und Wissensrepräsentation

Wissensbasis

fallspezifisches Wissen

Frage- und Erklärungs-komponente

Inferenzmaschine

Benutzer-schnittstelle

Wissensbasis und Expertensystem-shell

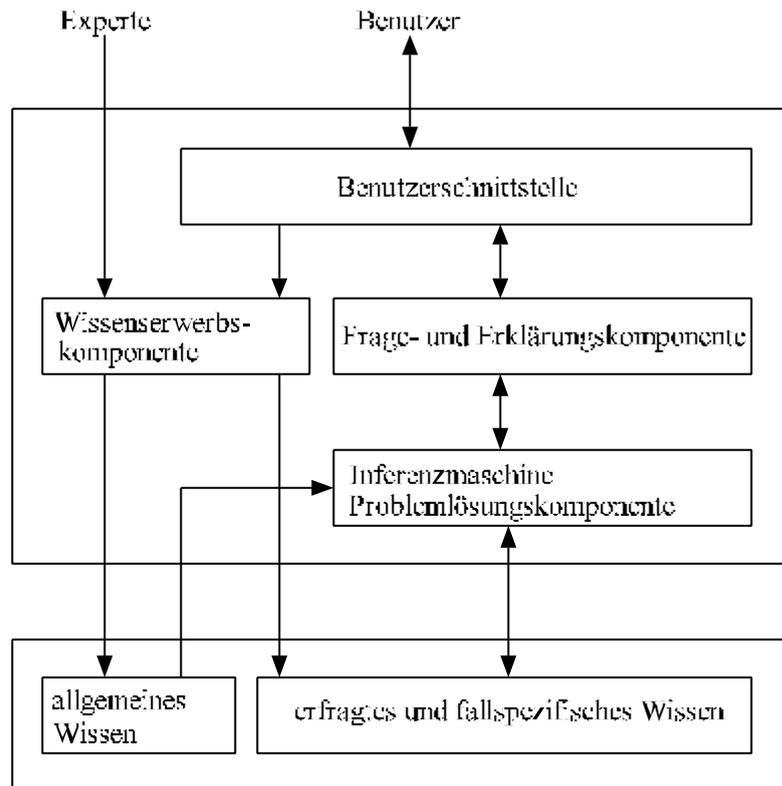
von Expertensystemen definitionsgemäß eine besondere Bedeutung zukommen. Die Frage- und Erklärungskomponente nimmt Fragen des Benutzers an, stellt Rückfragen an den Benutzer und gibt im Sinne einer Erklärung Antwort auf Warum- und Wie-Fragen des Benutzers.

Die Problemlösungskomponente, oft auch Inferenzmaschine genannt, versucht, das vom Benutzer gestellte Problem zu lösen. Dazu greift es auf die Wissensbasis zurück und erfragt bei Bedarf weiteres Wissen beim Benutzer. Mittels Backtracking, Unifikation und Anwendung von Schlußregeln und Regeln der Wissensbasis, sucht die Inferenzmaschine eine Lösung der Anfrage.

Die Kommunikation zwischen Benutzer und Expertensystem wird über die Benutzerschnittstelle, welche oft auch als graphische Benutzerschnittstelle zur Verfügung steht, abgewickelt.

Die Architektur eines Expertensystems ist in Abbildung 22-1 dargestellt. Ein Expertensystem gliedert sich in zwei wesentliche Teile: die Wissensbasis und die Expertensystemshell. Sie strikte Trennung ermöglicht es, durch Austausch der Wissensbasis und Beibehaltung der Expertensystemshell ein neues Expertensystem zu erhalten.

Abb. 22-1
Architektur eines
Expertensystems



Im folgenden sollen Konzepte von Expertensystemen exemplarisch behandelt werden. Wir tun dies bewußt anhand einer Expertensystemshell, die in Prolog realisiert ist. Zum einen arbeiten viele Expertensysteme mit Prolog oder Prolog-ähnlichen Sprachen, zum anderen soll aber auch deutlich werden, wie mit der neuen Programmiersprache größere Programme geschrieben werden können. Und zwar solche Programme, die intelligentes Verhalten auf Maschinen nachbilden, ein gewisses Maß von Sprachverarbeitung aufweisen und von der reinen Informations- zur Wissensverarbeitung übergehen. Sprachverarbeitung, logisches Schließen und Wissensverarbeitung sind Kennzeichen von Computersystemen neuer, nämlich intelligenter Art, welchen sich auch der Informatikunterricht widmen muß.

22.2 Wissensdomäne - Bittere Pillen

Es gibt geeignete und ungeeignete Beispiele für Wissensbasen in Expertensystemen. Zu den ungeeigneten zähle ich die bekannten Beispiele zum Erkennen von Tieren [Bra1] und [Win1], zur Bewertung von Kreditanträgen [Ste1] und zur Empfehlung eines Autotyps [Kle1], weil die gesellschaftliche Relevanz zu kurz kommt. Geeigneter sind gewiß die Beispiele aus [Bel1] zur Beratung für den Transport gefährlicher Güter und [Sav1] mit einem medizinischen Expertensystem, weil hierbei im Unterricht viel eher das Bedürfnis und die Notwendigkeit der Bewertung und Diskussion solcher Systeme entsteht.

Für unsere Übungen verwenden wir die Expertensystemshell aus [Bra1] und statten sie mit einer medizinischen Wissensbasis zur Schmerzbehandlung aus.

Das Expertenwissen entnehmen wir den *Bitteren Pillen* [Lan1]. Dabei beschränken wir uns auf Aspekte des ersten Kapitels, in dem Schmerzmittel behandelt werden. Die Schmerzmittel werden in vier Gruppen eingeteilt:

1. Schmerz- und fiebersenkende Mittel
2. Starke Schmerzmittel
3. Kopfschmerz- und Migränemittel
4. Krampflösende Mittel

Aus jeder Gruppe nehmen wir mehrere Medikamente in unsere Wissensbasis auf. Die Beschreibung eines Medikament gliedert sich *Präparat*, *Wichtigste Nebenwirkungen* und *Empfehlung*. Im folgenden sind Auszüge aus den Bitteren Pillen wiedergegeben:

Beispiele für
Wissensbasen

Wissensbasis zur
Schmerzbehand-
lung

Einteilung der
Schmerzmittel

Tabelle 22-1
Schmerz- und fiebersenkende Mittel

Präparat	Wichtigste Nebenwirkungen	Empfehlung
Aspirin Tabl. Acetylsalicylsäure (ASS)	Magenbeschwerden, kann in seltenen Fällen Asthmaanfälle auslösen. Wegen der Möglichkeit des erhöhten Risikos von Reye-Syndrom durch Acetylsalicylsäure (ASS) bei Kindern und Jugendlichen bis zum Alter von 19 Jahren ist Paracetamol vorzuziehen.	Therapeutisch zweckmäßig Als langbewährtes Mittel gegen Schmerzen, Fieber und rheumatische Entzündungen zu empfehlen. Bei empfindlichen Magen jedoch wenig geeignet.
ASS-Ratiopharm Tabl. Acetylsalicylsäure	Magenbeschwerden, kann in seltenen Fällen Asthmaanfälle auslösen. Wegen der Möglichkeit des erhöhten Risikos von Reye-Syndrom durch Acetylsalicylsäure (ASS) bei Kindern und Jugendlichen bis zum Alter von 19 Jahren ist Paracetamol vorzuziehen	Therapeutisch zweckmäßig Als langbewährtes Mittel gegen Schmerzen, Fieber und rheumatische Entzündungen zu empfehlen. Bei empfindlichen Magen jedoch wenig geeignet.
Ben-u-ron Saft, Tabl. Zäpfchen, Kapseln Paracetamol	Bei sehr häufigem, jahrelangem Gebrauch sind Nierenschäden nicht auszuschließen. Bei Überdosierung Leberschäden.	Therapeutisch zweckmäßig Als langbewährtes Mittel gegen Fieber und Schmerzen zu empfehlen. Bei rheumatischen Entzündungen ist ASS vorzuziehen.
Paracetamol-Ratiopharm Tabl., Zäpfchen Paracetamol	Bei sehr häufigem, jahrelangem Gebrauch sind Nierenschäden nicht auszuschließen. Bei Überdosierung Leberschäden.	Therapeutisch zweckmäßig Als langbewährtes Mittel gegen Fieber und Schmerzen zu empfehlen. Bei rheumatischen Entzündungen ist ASS vorzuziehen.

Präparat	Wichtigste Nebenwirkungen	Empfehlung	Tabelle 22-2 Starke Schmerzmittel
Alodan Amp. Pethidin <i>Rezeptpflichtig</i>	Müdigkeit, Übelkeit, Erbrechen, Atmungsstörungen, Suchtgefahr	Therapeutisch zweckmäßig nur zur Behandlung sehr schwerer Schmerzzustände (z.B. Krebsschmerzen), bei denen andere Schmerzmittel nicht mehr wirksam sind. Suchtgefahr.	
Dilaudid-Atropin Zäpfchen, Injektionslösung schwach und stark Hydromorphon Atropin <i>Rezeptpflichtig</i>	Müdigkeit, Mundtrockenheit, Sehstörungen, Übelkeit, Atmungsstörungen, Suchtgefahr. Erbrechen trotz Atropin möglich.	Therapeutisch zweckmäßig nur zur Behandlung sehr schwerer Schmerzzustände (vor allem Koliken), bei denen andere Schmerzmittel nicht mehr wirksam sind. Kombination eines Morphinderivats mit dem krampflösenden Atropin. Suchtgefahr.	
Heptadon Amp. Methadon <i>Rezeptpflichtig</i>	Müdigkeit, Übelkeit, Erbrechen, Atmungsstörungen, Suchtgefahr	Therapeutisch zweckmäßig nur zur Behandlung sehr schwerer Schmerzzustände (z.B. Krebsschmerzen), bei denen andere Schmerzmittel nicht mehr wirksam sind. Suchtgefahr.	
Vilan Amp., Tabl., Zäpfchen Nicomorphin <i>Rezeptpflichtig</i>	Bei sehr häufigem, jahrelangem Gebrauch sind Nierenschäden nicht auszuschließen. Bei Überdosierung Leberschäden.	Therapeutisch zweckmäßig nur zur Behandlung sehr schwerer Schmerzzustände (z.B. Krebsschmerzen), bei denen andere Schmerzmittel nicht mehr wirksam sind. Suchtgefahr.	

Tabelle 22-3
Kopfschmerz- und
Migränemittel

Präparat	Wichtigste Nebenwirkungen	Empfehlung
Avamigran Filmtabl., Zäpfchen Ergotamin Propyphenazon <i>Rezeptpflichtig</i>	Durchblutungsstörungen, Übelkeit, Erbrechen, Möglichkeit lebensbedrohlicher Schockformen. Lebensgefährliche Abnahme weißer Blutzellen ist nicht auszuschließen.	Möglicherweise zweckmäßig bei rechtzeitiger Einnahme zu Beginn eines Migräneanfalls. Kombination von Migränemittel (Dihydroergotamin) mit Schmerzmittel (Propyphenazon). Nur kurzfristig anwenden. Eine zuverlässige Wirkung ist mit Zäpfchen nicht zu erreichen.
Dociton Tabl. Propranolol <i>Rezeptpflichtig</i>	Langsamer Puls, Verstärkung einer Herzschwäche, Impotenz; Vorsicht bei Asthma, Zuckerkrankheit und Durchblutungsstörungen der Gliedmaßen. Vorsicht: Medikament nicht plötzlich absetzen, weil sonst schwere Herzschädigungen auftreten können	Möglicherweise zweckmäßig zur Vorbeugung von Migräneanfällen
Migril Ergotamin, Cyclizinhydrochlorid, Coffein <i>Rezeptpflichtig</i>	Durchblutungsstörungen, Übelkeit, Erbrechen, Müdigkeit	Möglicherweise zweckmäßig bei rechtzeitiger Einnahme zu Beginn eines Migräneanfalls. Kombination eines Migränemittel (Dihydroergotamin) mit einem Mittel gegen Erbrechen. Nur kurzfristig anwenden.
Sandomigran Drag., Pizotifen <i>Rezeptpflichtig</i>	Müdigkeit, Appetitzunahme, psychische Veränderungen, Mundtrockenheit, Pulsbeschleunigung, Erhöhung des Augeninnendrucks, Blasenentleerungsstörungen.	Möglicherweise zweckmäßig zur Vorbeugung von Migräne. Hemmt die Wirkung der körpereigenen Substanzen (Histamin, Serotonin, Acetylcholin).

Präparat	Wichtigste Nebenwirkungen	Empfehlung
Atropinsulfat Braun Amp. Atropin <i>Rezeptpflichtig</i>	Mundtrockenheit, Herzklopfen, Sehstörungen (Abnahme des Reaktionsvermögens), verminderte Schweißbildung (Wärmestau möglich).	Therapeutisch zweckmäßig bei Verkrampfungen im Magen-Darm-Bereich (z.B. Gallenkoliken, spastische Verstopfung)
Buscopan Amp., N-Butylscopolaminium <i>Rezeptpflichtig</i>	Mundtrockenheit, Herzklopfen, Sehstörungen (Abnahme des Reaktionsvermögens), verminderte Schweißbildung (Wärmestau möglich).	Therapeutisch zweckmäßig bei kolikartigen Krampfzuständen im Magen-Darm-Bereich.
Motillium Filmtabl., Susp., Domperidon <i>Rezeptpflichtig</i>	Müdigkeit, relativ selten Bewegungsstörungen (Dyskinesien), aber besonders bei Kindern möglich	Therapeutisch zweckmäßig bei Übelkeit, Erbrechen und zur Beschleunigung der Entleerung des Magens.
Spasmo-Cibagin Drag., Zäpfchen Propyphenazon Drofenin <i>Rezeptpflichtig</i>	Lebensgefährliche Abnahme weißer Blutzellen ist nicht auszuschließen. Möglichkeit lebensbedrohlicher Schockformen, Mundtrockenheit, Sehstörungen, Verstopfung	Abzuraten Nicht sinnvolle Kombination von Schmerzmittel (Propyphenazon) und krampflösendem Mittel (Drofenin).

Tabelle 22-4
Krampflösende Mittel (Spasmolytika)

22.3 Wissenrepräsentation - Struktur von Fakten und Regeln

Trennung von Wissensbasis und Expertensystemshell

Die Wissensrepräsentation nehmen wir in Form von Fakten und Regeln vor. Zur Unterstützung der Expertensystem-Architektur, insbesondere der Trennung von Wissensbasis und Expertensystemshell und des Entwurfs der Inferenzmaschine und der Erklärungskomponente, werden Fakten und Regeln in der Wissensbasis als Fakten zum Funktor ':= ' gespeichert. Die Inferenzmaschine greift als Metainterpreter auf diese Fakten zu und interpretiert sie, um eine Lösung zu erhalten.

Den Funktor ':= ' definieren wir mit `op(900, xfx, :=)` als Infixoperator. Fakten können dann in folgender Form notiert werden:

Beispiele für Fakten

```
fakt:= medikament('Aspirin','leichte Kopfschmerzen',x,x,
                  'Magenbeschwerden').
fakt:= medikament('ASS-Ratiopharm','leichte Kopfschmerzen',
                  x,x,'Magenbeschwerden').
fakt:= medikament('Ben-u-ron','leichte Kopfschmerzen',
                  x,x,x).
fakt:= medikament('Paracetamol-Ratiopharm','leichte
                  Kopfschmerzen',x,x,x).
```

Regelwissen wird in Form von Wenn-Dann-Regeln repräsentiert. Um solche Regeln möglichst nah in der Sprache des Experten und damit verständlich formulieren zu können, definieren wir weitere Operatoren:

Erweiterung des Sprachschatzes durch Operatoren

```
op(880, fx, wenn).
op(870, xfx, dann).
op(550, xfy, oder).
op(540, xfy, und).
op(300, fx, 'abgeleitet durch').
op(600, xfx, von).
op(600, xfx, durch).
op(500, fx, nicht).
op(100, xfx, ist).
op(100, xfx, sind).
op(100, xfx, gegen).
op(100, fx, keine_nebenwirkung).
op(100, xfx, hat_nebenwirkung).
op(100, xfx, vertraeglich_bei).
op(100, fx, unvertraeglich_bei).
op(100, xfx, wirksam_bei).
op(110, xfx, bei).
op(100, fx, zeige).
```

```
op(100, fx, neues).
op(100, xfx, beruecksichtigt).
```

Durch den konsequenten Einsatz solcher Operatoren können Fragen an das System und Antworten des Systems ohne irgendwelchen Aufwand nah zur natürlichen Sprache ausgedrückt werden.

Regeln können wir mittels Operatoren in folgender allgemeinen Form aufschreiben:

```
Regelname :=
  wenn
    Bedingung
  dann
    Folgerung.
```

Struktur von
Regeln

Zwei Beispiel für Regeln:

```
regel3 :=
  wenn
    Medikament gegen 'leichte Kopfschmerzen'
  dann
    Medikament gegen 'leichte Schmerzen'.

regel5 :=
  wenn
    'Kopfschmerzen' sind 'anfallsartig' und
    Medikament gegen 'Migräne'
  oder
    Medikament gegen 'leichte Kopfschmerzen'
  dann
    Medikament gegen 'Kopfschmerzen'.
```

Beispiele für
Regeln

In Regeln können Variablen und Konstanten genutzt werden. Regel 3 benutzt die Variable *Medikament* und die Konstanten *leichte Kopfschmerzen* und *leichte Schmerzen*. Regel 5 zeigt, daß Bedingungen auch zusammengesetzt sein können.

Man muß sich klar machen, daß obige Regeln des Expertensystems, auch wenn es nicht den Anschein hat, Prolog-Fakten zum Prädikat $:=/2$ sind. Die Darstellung der Regel 3 als Baumstruktur macht dies deutlich.

Abb. 22-2
Regel 3 als Baum-
struktur

In standardmäßiger Präfixnotation schreibt sich das:

```

    Regel 3 des
    Expertensystems
    als Prolog-Fakt
    := '(regel3, wenn(dann(
        gegen(Medikament, 'leichte Kopfschmerzen'),
        gegen(Medikament, 'leichte Schmerzen')))).
  
```

Der Vergleich mit Regel 3 von oben macht den Nutzen von Operatoren deutlich und zeigt, durch welche einfache Mittel natürlichsprachlich wirkende Sätze verarbeitet werden können.

22.4 Fakten über Medikamente

Nachdem nun klar ist, wie wir Wissen repräsentieren, muß Expertenwissen jetzt akquiriert werden. Faktenwissen legen wir als fünfstellige Strukturen ab:

```

Faktenwissen über
  Medikamente
medikament(Medikament, Beschwerde, Umstand,
           Nebenwirkung, Unverträglichkeit).
  
```

```

Medikament:      Medikamentenname
Beschwerde:      Anwendungsgebiet des Medikaments
Umstand:         Umstand der Beschwerde, z.B. bei
                 Anfall, zur Vorbeugung
Nebenwirkung:    Nebenwirkungen des Medikaments
Unverträglichkeit: nicht anwendbar bei z.B. Magen
                 beschwerden
  
```

Sind Umstände, Nebenwirkungen oder Unverträglichkeiten nicht vorhanden, so notieren wir dies mit dem Kleinbuchstaben x. Damit ergeben sich folgende Fakten:

```

    leichte
    Kopfschmerzen

fakt:= medikament('Aspirin', 'leichte Kopfschmer-
                zen',x,x, 'Magenbeschwerden').
fakt:= medikament('ASS-Ratiopharm', 'leichte Kopfschmer-
                zen',x,x, 'Magenbeschwerden').
  
```

```
fakt:= medikament('Ben-u-ron', 'leichte Kopfschmerzen',x,x,x).
```

```
fakt:= medikament('Paracetamol-Ratiopharm', 'leichte Kopfschmerzen',x,x,x).
```

Migränemittel

```
fakt:= medikament('Avamigran','Migräne','bei Anfall',  
'Übelkeit und Erbrechen',x).
```

```
fakt:= medikament('Dociton','Migräne','zur Vorbeu-  
gung','Herzschwäche und Impotenz','Asthma').
```

```
fakt:= medikament('Migril','Migräne','bei An-  
fall','Durchblutungsstörungen, Übelkeit',x).
```

```
fakt:= medikament('Sandomigran', 'Migräne',
  'zur Vorbeugung', 'psychischeVeränderung',
  'Prostatavergrößerung').
```

```
fakt:= medikament('Alodan', 'Krebs', x, 'Suchtgefahr', x).
```

starke Schmerzmittel

```
fakt:= medikament('Dilaudid-Atropin', 'Koliken', x,
  'Suchtgefahr', x).
```

```
fakt:= medikament('Heptadon', 'Krebs', x, 'Suchtgefahr', x).
```

```
fakt:= medikament('Vilan', 'Krebs', x, 'Suchtgefahr', x).
```

```
fakt:= medikament('Atropinsulfat', 'Krampf', 'im Magen-
  Darm-Bereich', 'Sehstörung', x).
```

Krämpfe

```
fakt:= medikament('Buscopan', 'Krampf', 'im Magen-Darm-
  Bereich', 'Sehstörung', x).
```

```
fakt:= medikament('Motilium', 'Krampf', 'bei Übel-
  keit', 'Bewegungsstörung', x).
```

```
fakt:= medikament('Spasmo-Cibalgin comp', 'Krampf', 'bei
  Schmerz', x, x).
```

22.5 Regeln zur Verabreichung von Schmerzmitteln

Regelwissen ist weit schwieriger zu akquirieren als Faktenwissen. Die folgenden Regeln entstanden in langwierigen Versuchen, in deren Verlauf sie mehrfach geändert und ergänzt wurden, bis der jetzige Stand erreicht wurde. Eigentlich sollte noch eine bessere Trennung zwischen Diagnose und Medikation angestrebt werden.

Zunächst gehen wir von der Klassifikation der Schmerztypen aus und stellen fest, welche der vier möglichen Schmerzarten der Patient hat:

```
regel02:=
  wenn
    ('Patient' beklagt 'starke Schmerzen' oder
    'Patient' beklagt 'Kopfschmerzen' oder
    'Patient' beklagt 'Krampf' oder
    'Patient' beklagt 'leichte Schmerzen') und
    'Patient' beklagt Beschwerde
  dann
    'Patient' hat Beschwerde.
```

Klassifikation der Schmerztypen

Der Operator *beklagt* ist als fragbar deklariert. Bei fragbaren Operatoren kann das System beim Benutzer nachfragen, also ob beispielsweise der Patient Kopfschmerzen hat.

fragbaren Operatoren

Ist eine Schmerzart festgestellt, so muß gegebenenfalls eine genauere Schmerzcharakterisierung erfolgen. Bei starken Schmerzen unterscheiden wir zum Beispiel Krebs Schmerzen und Schmerzen durch eine Kolik. Regel 04 fragt, ob Krebs Schmerzen oder Koliken vorliegen und sucht dann ein passendes Medikament.

Kopfschmerzen sind leichte Schmerzen	regel03:= wenn Medikament gegen 'leichte Kopfschmerzen' dann Medikament gegen 'leichte Schmerzen'.
Krebsschmerzen und Koliken sind starke Schmerzen	regel04:= wenn ('Patient' beklagt 'Krebsschmerzen' und Medikament gegen 'Krebs') oder ('Patient' beklagt 'Koliken' und Medikament gegen 'Koliken') dann Medikament gegen 'starke Schmerzen'.
Migränemittel gegen anfallsartige Kopfschmerzen	regel05:= wenn 'Kopfschmerzen' sind 'anfallsartig' und Medikament gegen 'Migräne' oder Medikament gegen 'leichte Kopfschmerzen' dann Medikament gegen 'Kopfschmerzen'.

Ein Medikament kann empfohlen werden, wenn es gegen die Beschwerde wirksam ist, der Umstand berücksichtigt ist und keine Unverträglichkeiten vorliegen:

eine Medikamenten- empfehlung	regel09:= wenn medikament (Medikament, Beschwerde, Umstand, _, Unvertraeglichkeit) und Medikament wirksam_bei Beschwerde und (Medikament und Beschwerde) beruecksichtigt Umstand und Medikament vertraeglich_bei Unvertraeglichkeit dann Medikament gegen Beschwerde.
-------------------------------------	--

Die Wirksamkeit ist schnell festgestellt:

```
regel06:=
  wenn
    medikament(Medikament, Beschwerde, _,_,_)
  dann
    Medikament wirksam_bei Beschwerde.
```

Wirksamkeit von
Medikamenten

Ob bei einem Medikament bei einer konkreten Beschwerde ein Umstand zu berücksichtigen ist, erfassen wir durch Regel 07. *bei* ist als fragbarer Operator ausgelegt.

```
regel07:=
  wenn
    medikament(Medikament, Beschwerde, Umstand,_,_) und
    ( gleich(Umstand, x)
      oder
        ungleich(Umstand, x) und Beschwerde bei Umstand)
  dann
    (Medikament und Beschwerde) beruecksichtigt Umstand.
```

Berücksichtigung
von Umständen

Nach ähnlichem Schema prüfen wir auf Unverträglichkeit *unvertraeglich bei* ist ein fragbarer Operator.

```
regel08:=
  wenn
    medikament(Medikament,_,_,_,Unvertraeglichkeit) und
    ( gleich(Unvertraeglichkeit, x)
      oder
        ungleich(Unvertraeglichkeit, x) und
          nicht unvertraeglich_bei Unvertraeglichkeit)
  dann
    Medikament vertraeglich_bei Unvertraeglichkeit.
```

Berücksichtigung
von Unverträglich-
keiten

Mit Regel 01 kann eine Beratung bestehend aus Diagnose und Therapie durchgeführt werden:

```
regel01:=
  wenn
    'Patient' hat Beschwerde und
    Medikament gegen Beschwerde
  dann
    'Beratung'.
```

Beratung,
Diagnose und
Therapie

22.6 Schnittstelle zu Prolog

Prolog-Regeln als
Fakten des
Expertensystems

Die Inferenzmaschine interpretiert nur den vorgegebenen Sprachumfang aus *wenn, dann, und, oder, nicht...* Wir können also nicht ohne weiteres in den Regeln Systemprädikate von Prolog benutzen. Dies geht nur über den Umweg, daß wir Prolog-Regeln als Fakten des Expertensystems aufnehmen.

Ungleich und *gleich* haben wir in obigen Regeln benutzt, um festzustellen, ob ein Umstand oder eine Unverträglichkeit vorliegt.

ungleich und *gleich*

```
fakt:= ungleich(A, B):- A \== B.
fakt:= gleich(A, B):- A == B.
```

Mit *zeigeMedikament* können wir uns die Daten eines Medikaments anzeigen lassen:

zeigeMedikament
als Fakt des
Expertensystems

```
fakt:=
zeigeMedikament:-
    fakt:= medikament(Medikament, Beschwerde, Umstand,
                    Nebenwirk, Unvertraeg),
    write('Medikament      : '), write(Medikament), nl,
    write('Beschwerde      : '), write(Beschwerde), nl,
    write('Umstand         : '), write(Umstand), nl,
    write('Nebenwirkung     : '), write(Nebenwirk), nl,
    write('Unverträglichkeit: '), write(Unvertraeg), nl.
```

22.7 Ergänzungen zur Wissensrepräsentation

Stellt man beispielsweise die Anfrage *'Novalgin' wirksam_bei 'Kopfschmerzen'* so verhält sich unser Expertensystem sehr zurückhaltend, ganz im Gegensatz zum Verhalten des Prolog-Interpreters, der in Verkennung der Tatsachen glatt *No* antworten würde.

negatives Wissen

Der Grund für die vornehme Zurückhaltung ist darin zu sehen, daß die Anfrage weder verifizierbar noch falsifizierbar ist, denn es existiert in unserer Wissensbasis kein Fakt über das Medikament *Novalgin*. Die Angelegenheit wird aber nicht wesentlich besser, wenn wir ein solches Faktum ergänzen. Das erkennt man bei der Anfrage *Ihre Frage: 'Aspirin' wirksam_bei 'Krampf'*.

Zwar ist ein Fakt über *Aspirin* gespeichert, aber die gestellte Anfrage ist mit unserer Implementierung der Regel06 nicht widerlegbar. Diese Regel kann zu einem gegebenen Medikament die Beschwerde liefern und bestätigen, daß

ein Medikament gegen die Beschwerde wirksam ist. Über die Negation muß man sich separat Gedanken machen.

Auf obige Aspirin-Frage sollte das Expertensystem die Antwort *falsch* liefern, weil in der Wissensbasis als Fakt vorkommt, daß Aspirin gegen leichte Kopfschmerzen wirkt und Krampf und leichte Kopfschmerzen verschiedene Beschwerden sind. Dies können wir umsetzen in die Regel

```
regel06b:=
  wenn
    medikament(Medikament, Beschwerdel,_,_,_) und
    nicht gleich(Beschwerdel, Beschwerde)
  dann
    nicht Medikament wirksam_bei Beschwerde.
```

Da die Anfrage positiv gestellt ist, kann Regel 6b in dieser Form nicht zur Lösung der Frage benutzt werden. Wir müssen das äußere nicht nach innen ziehen:

```
regel06a:=
  wenn
    nicht(medikament(Medikament, Beschwerdel,_,_,_) und
           nicht gleich(Beschwerdel, Beschwerde))
  dann
    Medikament wirksam_bei Beschwerde.
```

Verwendung von
nicht

Eine vergleichbare Situation liegt bei unserer Implementierung von *gleich* vor. Die Fragen *gleich(a,a)* und *nicht gleich(a,a)* werden korrekt beantwortet. Aber zur Frage *gleich(a,b)* und *nicht gleich(a,b)* gibt es keine Antwort. Dazu müssen wir folgende Regel ergänzen:

```
regel81:=
  wenn
    nicht ungleich(A, B)
  dann
    gleich(A, B).
```

22.8 Trennung von Wissensbasis und Expertensystemshell

Da wir die Wissensbasis konsequent von der Expertensystemshell getrennt haben, ist es ganz einfach, bei gleicher Shell mit einer anderen Wissensbasis ein neues Expertensystem zu erhalten. Aus [Bra1] entnehmen wir eine Wissensbasis über elektrische Geräte:

Expertenwissen
über elektrische
Geräte

```
defekt_regel:=
    wenn
        eingeschaltet(Geraet) und
        geraet(Geraet) und
        nicht arbeitet(Geraet) und
        verbunden(Geraet, Sicherung) und
        bewiesen(intakt(Sicherung))
    dann
        bewiesen(defekt(Geraet)).

sicherung_ganz_regel:=
    wenn
        verbunden(Geraet, Sicherung) und
        arbeitet(Geraet)
    dann
        bewiesen(intakt(Sicherung)).

sicherung_defekt_regel:=
    wenn
        verbunden(Geraet1, Sicherung) und
        eingeschaltet(Geraet1) und
        nicht arbeitet(Geraet1) und
        gleiche_sicherung(Geraet1, Geraet2) und
        eingeschaltet(Geraet2) und
        nicht arbeitet(Geraet2)
    dann
        bewiesen(durchgebrannt(Sicherung)).

gleiche_sicherungsregel:=
    wenn
        verbunden(Geraet1, Sicherung) und
        verbunden(Geraet2, Sicherung) und
        verschieden(Geraet1, Geraet2)
    dann
        gleiche_sicherung(Geraet1, Geraet2).

fakt:= verschieden(X, Y):- not(X = Y).
fakt:= geraet(heizluefter).
fakt:= geraet(lampe1).
fakt:= geraet(lampe2).
fakt:= geraet(lampe3).
fakt:= geraet(lampe4).
fakt:= verbunden(lampe1, sicherung1).
fakt:= verbunden(lampe2, sicherung1).
fakt:= verbunden(heizluefter, sicherung1).
fakt:= verbunden(lampe3, sicherung2).
fakt:= verbunden(lampe4, sicherung2).
fragbar(eingeschaltet(G), eingeschaltet('Geraet')).
fragbar(arbeitet(G), arbeitet('Geraet')).
```

22.9 Aufgaben

Ein lauffähiges Expertensystem erhalten Sie durch Konsultieren der Datei SCHMERZ.PL. Starten Sie mit der Anfrage: ?- *experte*.

1. Stellen Sie folgende Fragen. Wird eine Wissensfrage vom Expertensystem gestellt, so können Sie mit der Antwort *warum* nachfragen, weswegen diese Frage gestellt wird.
 - a) 'Aspirin' gegen 'leichte Kopfschmerzen'.
 - b) Medikament gegen 'leichte Kopfschmerzen'.
 - c) Medikament gegen 'Migräne'.
 - d) Medikament wirksam_bei 'Krebs'.
 - e) 'Beratung'.
 - f) zeige 'Migril'.
 - g) nicht 'Vilan' gegen 'Krebs'.
 - h) 'Patient' hat Beschwerde und Medikament gegen Beschwerde.
 - i) Medikament gegen 'Krampf' und zeige Medikament.
 - j) Medikament gegen 'leichte Kopfschmerzen' oder Medikament gegen 'Migräne'.

2. Ergänzen Sie die Wissensbasis um die beiden krampflösenden Medikamente *Dopasse* und *Duspatal*.

Präparat	Wichtigste Nebenwirkungen	Empfehlung
Dolpasse Amp. Metamizol, Orphenadrin, Vitamin B6 <i>Rezeptpflichtig</i>	Mundtrockenheit, Herzklopfen, Sehstörungen (Abnahme des Reaktionsvermögens), verminderte Schweißbildung. Seltene, dann aber lebensgefährliche Abnahme weißer Blutzellen oder lebensbedrohlicher Schockformen (unter anderem Blutdruckabfall) und Hautausschläge (auch schwere Formen möglich).	Abzuraten Wenig sinnvolle Kombination von krampflösenden Mitteln (Orphenadrin) mit Schmerzmitteln (Metamizol). Höchstens vertretbar zur kurzfristigen Behandlung von schwersten kolikartigen Schmerzen in begründeten Ausnahmefällen. Metamizolhaltige Präparate sind in vielen Ländern verboten.
Duspotal Drag., Susp.	Mundtrockenheit, Kopfschmerzen	Möglicherweise zweckmäßig zur kurzfristigen An-

Mebeverin
Rezeptpflichtig

wendung beim sogenannten
Reizkolon

3. Schmerz- und fiebersenkende Mittel enthalten oft den Wirkstoff Acetylsalicylsäure (ASS). Wegen der Möglichkeit des erhöhten Risikos von Reye-Syndrom sollte dieser Wirkstoff bei Kindern und Jugendlichen bis zum Alter von 19 Jahren nicht verabreicht werden. Das Reye-Syndrom ist eine sehr seltene Erkrankung mit schwerem Erbrechen, Fieber, Leberfunktionsstörungen, Orientierungsverlust, Krämpfen und Koma. 20 bis 30 Prozent der Kinder sterben. Ergänzen Sie den *Umstand*, daß der Patient mindestens 20 Jahre alt sein soll, in den betreffenden Fakten.

4. Implementieren Sie Regeln, mit denen Anfragen der Art
 - a) `Medikament hat_nebenwirkung Nebenwirkung.`
 - b) `keine_nebenwirkung Medikament.`
beantwortet werden können.
 - c) Testen Sie ihre Regeln mit dem Anfragen
`'Heptadon' hat_nebenwirkung Nebenwirkung`
`Medikament hat_nebenwirkung 'Sehstörung'`
`keine_nebenwirkung 'Ben-u-ron'`
`keine_nebenwirkung 'Atropinsulfat'`
`Medikament gegen 'Krebs' und`
`keine_nebenwirkung Medikament`

5. Stellen Sie Regel 8 grafisch als Baumstruktur dar.

6. Die Wissensbasis enthält fast nur Medikamente, die als zweckmäßig eingestuft sind, damit nur solche empfohlen werden. Wie müßte die Wissensbasis ergänzt werden, um auch weniger empfehlenswerte Medikamente berücksichtigen zu können.

7. Ergänzen Sie ein Prädikat, mit dem interaktiv neue Medikamente aufgenommen werden können.

23 Komponenten einer Expertensystemshell

Die Expertensystemshell stammt von Bratko und wird in [Bra1] ausführlich entwickelt und dargestellt. Leider ist das gut lesbare Buch vergriffen und eine Neuauflage nicht geplant. Wir werden in Anlehnung an [Bra1] die Komponenten einer Expertensystemshell darstellen.

23.1 Die Inferenzmaschine

Die Inferenzmaschine arbeitet als sogenanntes rückwärts-verkettendes System. Eine an das Expertensystem gestellte Frage wird als Konklusion betrachtet. Das System versucht nun Prämissen abzuleiten, aus denen sich durch Anwendung einer Regel die Konklusion ergibt. Gelingt ein Beweis der Prämissen, so hat man auch einen Beweis der ursprünglichen Frage gefunden.

rückwärts-
verkettende
Inferenzmaschine

Prolog-Interpreter arbeiten ebenfalls als rückwärts-verkettendes Systeme. Dies macht man sich am besten durch die Beziehung zwischen einer Wenn-Dann-Regel des Expertensystems und einer Prolog-Klausel klar. Betrachten wir als Beispiel die Regel 06:

Prolog-Interpreter
als rückwärts-
verkettende
Systeme

```
regel06 :=
  wenn
    medikament(Medikament, Beschwerde, _, _, _)
  dann
    Medikament wirksam_bei Beschwerde.
```

Die Folgerung (Konklusion) wird zum Klausel-Kopf, die Bedingung (Prämisse) der Wenn-Dann-Regel zum Klausel-Rumpf. Als Prolog-Klausel schreibt sich dies in der Form:

```
wirksam_bei(Medikament, Beschwerde):-
  medikament(Medikament, Beschwerde, _, _, _).
```

Die Inferenzmaschine des Expertensystems wird durch den nachfolgenden Inferenz-Algorithmus beschrieben.

Inferenz-Algorithmus Benutze eine der folgenden Regeln, um eine Antwort A auf eine Frage F zu finden:

1. Ist die Frage F ein Fakt der Wissensbasis, dann lautet die Antwort *F ist wahr*.
2. Gibt es in der Wissensbasis eine Regel der Form *Wenn Bedingung dann Frage F*, dann erkunde *Bedingung*, um eine Antwort A zu finden.
3. Ist die Frage F von der Gestalt *F1 und F2*, dann untersuche F1. Ist F1 falsch, dann lautet die Antwort *F ist falsch*, ansonsten erkunde F2 und kombiniere die Antworten auf beiden Fragen geeignet zur Antwort A.
4. Ist F von der Gestalt *F1 oder F2*, dann untersuche F1. Wenn F1 wahr ist, dann lautet die Antwort *F ist wahr*, oder alternativ, erkunde F2 und kombiniere die Antworten auf beiden Fragen zur Antwort A.
5. Ist F von der Gestalt *nicht F1*, dann untersuche F1. Wenn F1 wahr ist, dann lautet die Antwort *F ist falsch*, wenn F1 falsch ist, dann lautet die Antwort *F ist wahr*.
6. Ist F eine *fragbare* Frage, dann frage den Anwender nach F.

Die Realisierung dieses Algorithmus erfolgt durch das Prädikat

Schnittstelle der Inferenzmaschine

```
erforsche(+Frage, +Spur, -Antwort).
```

Im ersten Argument übergibt man die zu untersuchende Frage, im dritten Argument wird die Antwort zusammen mit der Begründung geliefert. Zur Beantwortung von Warum-Fragen dient das Argument *Spur*. Der dargestellte Inferenz-Algorithmus wird wie folgt implementiert:

Implementierung der Inferenzmaschine

```
erforsche(Ziel, Spur, Ziel ist wahr
          wurde 'als Fakt gefunden.):-
fakt:= Ziel.

erforsche(Ziel, Spur, Ziel ist Wahrheitswert wurde
          'abgeleitet durch' Regel von Antwort):-
Regel:= wenn Bedingung dann Ziel,
erforsche(Bedingung,[Ziel durch Regel|Spur], Antwort),
wahrheit(Antwort, Wahrheitswert).
erforsche(Ziel1 und Ziel2, Spur, Antwort):-
```

```

erforsche(Ziell, Spur, Antwort1),
fahre_fort(Antwort1, Ziell und Ziel2, Spur, Antwort).

erforsche(Ziell oder Ziel2, Spur, Antwort):-
erforsche_ja(Ziell, Spur, Antwort);
erforsche_ja(Ziel2, Spur, Antwort).

erforsche(Ziell oder Ziel2,Spur,Antwort1 und Antwort2):-
!,
not erforsche_ja(Ziell, Spur, _),
not erforsche_ja(Ziel2, Spur, _),
erforsche(Ziell, Spur, Antwort1),
erforsche(Ziel2, Spur, Antwort2).

erforsche(nicht Ziel, Spur, Antwort):- !,
erforsche(Ziel, Spur, Antwort1),
invertiere(Antwort1, Antwort).

erforsche(Ziel,Spur,Ziel ist Antwort wurde mitgeteilt):-
benutzerantwort(Ziel, Spur, Antwort).
    
```

Der Algorithmus und die Implementierung machen klar, welche Art von Fragen beantwortet werden können: Fragen, die aufgrund von Fakten oder Regeln lösbar sind, Konjunktionen, Disjunktionen und Negationen solcher Fragen, sowie Fragen, die an den Benutzer gestellt werden können.

Eine genauere Analyse der Expertensystemshell zeigt, daß man es mit einer dreiwertigen Logik zu tun hat. Eine Frage kann mit der Antwort wahr (w), falsch (f) oder unbekannt (u) beantwortet werden. Unbekannt heißt, daß eine Anfrage von der Inferenzmaschine weder mit wahr noch mit falsch beantwortet werden kann. Dabei wirken sich Operatoren wie folgt aus:

dreiwertige Logik

und	w	f	u
w	w	f	u
f	f	f	f
u	u	u	u

oder	w	f	u
w	w	w	w
f	w	f	u
u	w	u	u

nicht	w	f	u
	f	w	u

Tabelle 23-1
Operatoren der dreiwertigen Logik

Die Konjunktion wird im Gegensatz zur Disjunktion nicht symmetrisch behandelt. Bei der Negation muß man darauf achten, daß in der Anfrage alle Variablen instanziiert sind. Zur Erläuterung betrachten wir das Beispiel:

Probleme der Negation

Ihre Frage: nicht 'Ben-u-ron' wirksam_bei Beschwerde.
Die Intention der Frage ist herauszubekommen, bei welchen Beschwerden Ben-u-ron nicht wirksam ist. Das Expertensystem liefert aber die Antwort:

```
Antwort: (nicht Ben-u-ron wirksam_bei leichte
```


wortung der zweiten Frage wendet die Inferenzmaschine gerade Regel 9 an. Diese Regel führt zur Instanzierung *Medikament = Avamigran* und zur Teilfrage (*Avamigran und Migräne*) *beruecksichtigt bei Anfall*. Zur Lösung dieser Teilfrage zieht das System Regel 7 heran. Diese Regel enthält die Teilfrage *Migräne bei Anfall*. Da es keine Regel und kein Fakt für *bei* gibt, aber *bei* fragbar ist, stellt die Inferenzmaschine die Frage *Migräne bei Anfall*, auf die der Benutzer mit *warum* gerade die Warum-Frage gestellt hat.

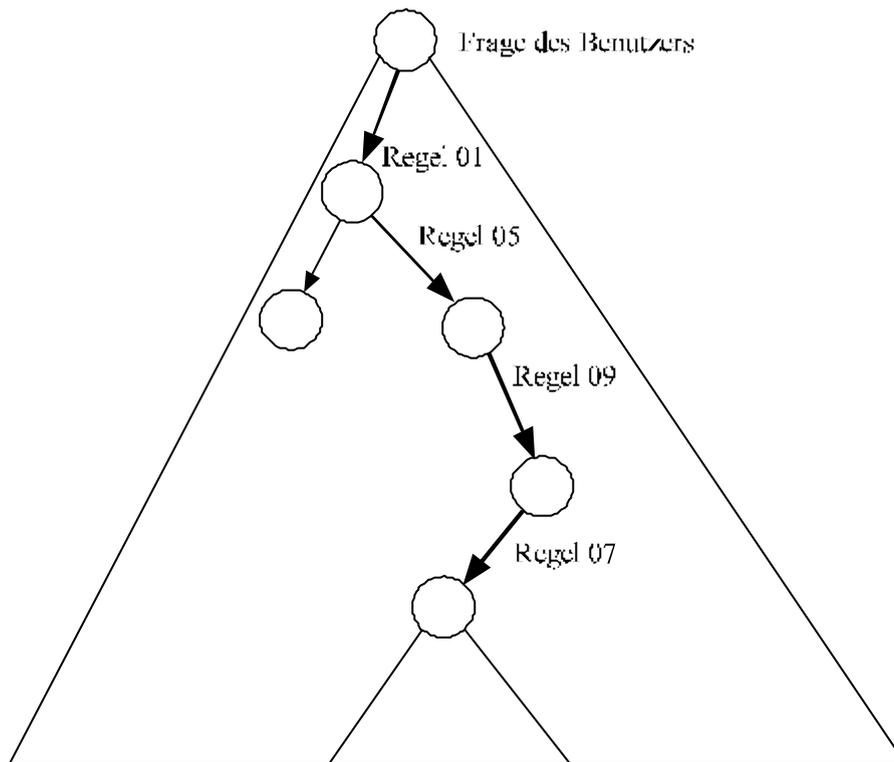


Abb. 23-1
Spur im
Suchbaum

In Abbildung 23-1 wird die Spur als Folge fester Pfeile im Suchraum angegeben. Offensichtlich wird die Spur immer dann erweitert, wenn eine Regel in Anspruch genommen wird. Die zugehörige *erforsche*-Klausel lautet:

```
erforsche(Ziel, Spur, Ziel ist Wahrheitswert wurde
          'abgeleitet durch' Regel von Antwort):-
  Regel:= wenn Bedingung dann Ziel,
  erforsche(Bedingung,[Ziel durch Regel|Spur],Antwort),
  wahrheit(Antwort, Wahrheitswert).
```

Erweiterung der
Spur durch
erforsche-Regel

Die Spur wird als Keller in einer Liste verwaltet, welcher bei Anwendung einer Regel um den Eintrag *Ziel durch Regel* ergänzt wird. Gibt der Benutzer als Reaktion auf eine Frage *warum* ein, so wird mittels *zeige_spur* die Spur ausgegeben:

Ausgabe der Spur

```
zeige_spur([]):-
    nl, write('Das war ihre Frage. '), nl.

zeige_spur([Ziel durch Regel|Spur]):-
    nl, write('Wurde mit '), write(Regel),
    write(' abgeleitet: '), write(Ziel),
    zeige_spur(Spur).
```

23.3 Erklärungskomponente - Wie-Fragen

Wie-Frage Hat das System eine Antwort auf eine Frage des Benutzers gefunden, so kann sich der Benutzer mit einer Frage *wie* erklären lassen, wie das Expertensystem zu seiner Antwort gekommen ist. Eine Wie-Frage wird mit einem Beweis aus Regeln und Unterregeln beantwortet, welche zur Herleitung der Folgerung benutzt werden.

Beweis als
Und-Oder-
Lösungsbaum

Für unsere Regelsprache besteht ein Beweis aus einem Und-Oder-Lösungsbaum. Daher muß unsere Inferenzmaschine einen Und-Oder-Lösungsbaum erzeugen, der die Regeln und Unterziele enthält. Nur die Beantwortung des obersten Ziels würde unzureichend sein.

Der Und-Oder-Baum wird im dritten Argument von *erforsche* aufgebaut. Antwort hat den allgemeinen Aufbau *Folgerung wurde Gefunden*. Dabei ist *Gefunden* eine Rechtfertigung für *Folgerung*. Die *erforsche*-Klauseln zeigen, welche Antworten erzeugt werden:

Aufbau des
Und-Oder-
Lösungsbaums
durch den
wurde-Operator

```
erforsche(Ziel, Spur, Ziel ist wahr wurde
    'als Fakt gefunden.):-
    fakt:= Ziel.

erforsche(Ziel, Spur, Ziel ist Wahrheitswert wurde
    'abgeleitet durch' Regel von Antwort):-
    Regel:= wenn Bedingung dann Ziel,
    erforsche(Bedingung,[Ziel durch Regel|Spur], Antwort),
    wahrheit(Antwort, Wahrheitswert).
```

```

erforsche(Ziell oder Ziel2,Spur,Antwort1 und Antwort2):-
!,
not erforsche_ja(Ziell, Spur, _),
not erforsche_ja(Ziel2, Spur, _),
erforsche(Ziell, Spur, Antwort1),
erforsche(Ziel2, Spur, Antwort2).

erforsche(Ziel,Spur,Ziel ist Antwort wurde mitgeteilt):-
benutzerantwort(Ziel, Spur, Antwort).

```

In unserem Beispiel sieht die Antwort dann so aus:

Beratung

```

wurde abgeleitet durch regel01 von
  Patient hat Kopfschmerzen
  wurde abgeleitet durch regel02 von
    Patient beklagt Kopfschmerzen
    wurde mitgeteilt
  und
  Patient beklagt Kopfschmerzen
  wurde mitgeteilt
und
Avamigran gegen Kopfschmerzen
  wurde abgeleitet durch regel05 von
    Kopfschmerzen sind anfallsartig
    wurde mitgeteilt
  und
  Avamigran gegen Migräne
  wurde abgeleitet durch regel06 von
    medikament(Avamigran,Migräne,_,_,_)
    wurde als Fakt gefunden.
  und
  (Avamigran und Migräne) beruecksichtigt bei
  Anfall
  wurde abgeleitet durch regel07 von
    medikament(Avamigran,Migräne,bei
    Anfall,_,_)
    wurde als Fakt gefunden.
  und
  ungleich(bei Anfall,x)
  wurde als Fakt gefunden.
  und
  Migräne bei bei Anfall
  wurde mitgeteilt
und
Avamigran vertraeglich_bei x
  wurde abgeleitet durch regel08 von
    medikament(Avamigran,_,_,_,x)
    wurde als Fakt gefunden.
  und
  gleich(x,x)
  wurde als Fakt gefunden.

```

Und-Oder-
Lösungsbaum als
Antwort auf die
Frage wie

Das Anzeigen des Und-Oder-Lösungsbaums wird vom Prädikat *zeige* vorgenommen. Durch Einrücken und Beachtung der Antwortmuster wird eine passende Darstellung der Unterbäume erreicht:

Anzeigen des Und-Oder- Lösungsbaums	<pre>zeige(Loesung):- nl, zeige(Loesung, 0), !.</pre>
Konjunktion	<pre>zeige(Antwort1 und Antwort2, H):- !, zeige(Antwort1, H), tab(H), write(und), nl, zeige(Antwort2, H).</pre>
Negation	<pre>zeige(nicht Antwort ist Wahrheit wurde Gefunden, H):- tab(H), write(nicht), nl, H1 is H + 4, zeige(Antwort, H1), tab(H), write('ist '), write(Wahrheit), write(' wurde '), write(Gefunden), nl.</pre>
Konklusion	<pre>zeige(Antwort wurde Gefunden, H):- tab(H), schreibe_ant(Antwort), nl, tab(H), write(' wurde '), zeigel(Gefunden, H).</pre>
Regelanwendung	<pre>zeigel(Abgeleitet von Antwort, H):- !, write(Abgeleitet), write(' von '), nl, H1 is H + 4, zeige(Antwort, H1). zeigel(Gefunden, _):- write(Gefunden), nl. schreibe_ant(Ziel ist wahr):- !, write(Ziel). schreibe_ant(Antwort):- write(Antwort).</pre>

23.4 Aufgaben

- 1a). Laden Sie das Expertensystem und setzen Sie mittels `spy(erforsche)` einen Haltepunkt auf das Prädikat `erforsche`.
 - b). Tracen Sie die Anfrage:


```
?- erforsche('Ben-u-ron' wirksam_bei Medikament,
             [], Loesung)).
```
 - c). Welche Spuren - mittleres Argument von `erforsche` - haben wir jeweils am Exit-Port?
 - d). Welche Und-Oder-Bäume - drittes Argument von `erforsche` - haben wir jeweils am Exit-Port?
2. Untersuchung der Inferenzmaschine.
 - a). Löschen Sie alle Haltepunkte und setzen Sie neue auf die Call-Ports von `erforsche`.
 - b). Tracen Sie die Anfrage `?- experte`, wobei Sie im Expertensystem die Frage `Beratung` stellen und auf `starke Schmerzen` mit `ja` antworten. Notieren Sie bei jedem Call, welche `erforsche`-Klausel die Inferenzmaschine ausgewählt hat.
 3. Untersuchung der Erklärungskomponente - Wie-Fragen
 - a). Ändern Sie die zweite `erforsche`-Klausel zur automatischen Anzeige der Spur wie folgt ab:

```
erforsche(Ziel, Spur, Ziel ist Wahrheitswert wurde
          'abgeleitet durch' Regel von Antwort):-
  Regel:= wenn Bedingung dann Ziel,
  nl, write([Ziel durch Regel|Spur]), nl,      /* neu */
  erforsche(Bedingung,[Ziel durch Regel|Spur],Antwort),
  wahrheit(Antwort, Wahrheitswert).
```

- b). Führen Sie den folgenden Dialog und achten Sie dabei auf die aktuelle Spur, mit welcher Warum-Fragen beantwortet werden können.

----- Schmerzmittel-Berater-----

```
Ihre Frage bitte: 'Beratung'
Patient beklagt starke Schmerzen? warum. dann nein.
Patient beklagt Kopfschmerzen?    warum. dann ja.
Kopfschmerzen sind anfallsartig?  warum. dann ja.
Migräne - bei Anfall?              warum. dann ja.
```

- c) Fügen Sie jetzt in das *erforsche*-Prädikat die weitere Zeile
`nl, zeige_spur([Ziel durch Regel|Spur]), nl`
ein und wiederholen Sie Teil b.
4. Untersuchung der Erklärungskomponente - Wie-Fragen
- a) Machen Sie die Änderungen aus Aufgabe 3 rückgängig.
- b) Stellen Sie die Anfrage
`?- init, erforsche('Ben-u-ron' wirksam_bei Medikament,
[], Antwort).`
wobei *init* dafür sorgt, daß zuvor erfragtes Wissen wieder gelöscht wird.
- c) Ergänzen Sie die obige Anfrage um *zeige(Antwort)*.
- d) Wiederholen Sie b) und c) mit
`?- init, erforsche('Beratung', [], Antwort).`

Anhang A

Glossar

Anfrage	Eine Frage des Benutzers an das Prolog-System. Mit der Anfrage <code>?- vater(Papa, max).</code> soll das Prolog-System den Vater von Max ermitteln.
Anonyme Variable	Eine anonyme Variable benutzt man, wenn man am Wert einer Variablen nicht interessiert ist. Anonyme Variable bezeichnet man mit dem Unterstrich „_“. Kommt der Unterstrich mehrmals innerhalb einer Klausel vor, so steht er jeweils für eine andere Variable.
Argument	Bestandteil eines zusammengesetzten Terms. In <code>vater(peter, hans)</code> sind <code>peter</code> und <code>hans</code> die beiden Argumente.
Arität	Stelligkeit oder Anzahl der Argumente einer Struktur. Im Fakt <code>schueler(hans, schmidt, 9a)</code> hat das Prädikat <code>schueler</code> die Arität 3. Schreibweise: <code>schueler/3</code> .
Atom	Eine Zeichenfolge zur Bezeichnung eines Objekts. Atome werden klein oder in einfachen Anführungszeichen geschrieben. Beispiele: <code>hans</code> , <code>'Müller'</code> . Zahlen und Variablen sind keine Atome.
Backtracking	Verfahren für Suchprozesse, das im Falle des Fehlschlagens einer Suche aufgrund einer Sackgasse zum nächstliegenden Knoten im Suchbaum zurückkehrt, an dem es noch alternative Möglichkeiten gibt und dort die Suche fortsetzt.
Cut	Ein Systemprädikat, welches Backtracking im Prädikat, das den Cut enthält, beim Versuch der Reerfüllung unterbindet. Schreibweise: „! <code>!</code> “

Disjunktion	Logische Oder-Verknüpfung. Sie wird in Prolog durch das Semikolon ausgedrückt.
Fakt	Eine Tatsache über Eigenschaften von Objekten und Beziehungen zwischen diesen. Beispiel: <i>alter('Theo', 27)</i> sagt aus, daß Theo 27 Jahre alt ist. Ein Fakt schreibt man als Klausel ohne Regulator „:-“.
Funktor	Name einer Struktur. Beispiele: In <i>mutter(doris, lea)</i> ist <i>mutter</i> Funktor des Prädikats <i>mutter/2</i> . In $2 = 3 + 7$ ist „= \leq “ und in $[2, 3, 7]$ „ \leq “ der jeweilige Funktor. Stellt man eine Struktur als Baum dar, so findet man den Funktor in der Wurzel.
Instanziierung	Bindung von Variablen an Konstanten oder Strukturen durch Unifikation.
Klausel	Ein Fakt oder eine Regel.
Konjunktion	Logische Und-Verknüpfung. Sie wird in Prolog durch das Komma ausgedrückt.
Konstante	Eine Integer-Zahl, eine Real-Zahl oder ein Atom.
Kopf-Rest-Methode	Standard-Methode zur Listenbearbeitung. Man bearbeitet zunächst den Kopf, dann den Rest der Liste oder erst den Rest und dann den Kopf.
Liste	Ein spezieller zusammengesetzter Term. Zum Beispiel ist $[a, 7, X]$ eine Liste mit drei Elementen. Die leere Liste wird mit $[\]$ bezeichnet.
Listenoperator	Operator zum Zerlegen einer Liste in Kopfelement und Restliste oder zum Zusammensetzen einer Liste aus einem Kopfelement und einer Restliste. Der Listenoperator wird für die Kopf-Rest-Methode benötigt. Schreibweise: „ “.

Objekt	Objekte sind Dinge aus unserer Vorstellungswelt oder Abstraktionen von realen Gegenständen. Sie werden in Prolog durch Zeichenketten repräsentiert. Beispielsweise sind Atome, Zahlen, Fakten, Regeln und Terme Objekte.
Operator	Ein Atom, das als Funktor in Präfix-, Infix- oder Postfix-Schreibweise auftreten kann.
Prädikat	Menge aller Klauseln mit gleichem Funktor und gleicher Stelligkeit. Beispiel: <i>elternteil/2</i> ist das <i>zweistellige</i> Prädikat <i>elternteil</i> .
Regel	Logische Aussage der Form „Wenn A_1 und A_2 und ... und A_n , dann B “, wobei A_1, A_2, \dots, A_n die Prämissen (Voraussetzungen) und B die Konklusion (Folgerung) darstellt. In Prolog schreibt man eine solche Regel in der Form „ $B:- A_1, A_2, \dots, A_n$ “, wobei A_1, A_2, \dots, A_n der Regelrumpf und B der Regelkopf ist.
Resolution	Ein Verfahren zum Beweisen logischer Aussagen. Wenn die zu beweisende Aussage A ein Faktum ist, dann ist sie bewiesen. Wenn eine Regel $B:- A_1, A_2, \dots, A_n$ existiert, deren Kopf B sich mit der Aussage A unifizieren lässt, so ist A beweisbar, wenn alle Teilziele A_1, A_2, \dots, A_n des Regelrumpfes beweisbar sind. Kann eine Aussage in mehreren Schritten unter Verwendung der Resolution auf die leere Aussage zurückgeführt werden, dann ist die Aussage beweisbar.
Stelligkeit	siehe Arität
Struktur	Ein zusammengesetzter Term aus Funktor und mindestens einem Argument.
Term	Das grundlegende Datenobjekt in Prolog. Ein Term kann eine Konstante, Variable oder ein zusammengesetzter Term sein.

Unifikation	Der Prozeß des Gleichmachens von Termen durch Instanzierung von Variablen mit Termen.
univ-Operator	Operator zum Zerlegen eines Baums in eine Liste oder zum Zusammensetzen eines Baums aus einer Liste aus Wurzel und Knoten. der univ-Operator wird für die Wurzel-Knoten-Methode benötigt. Schreibweise: „=..“
Variable	Eine Variable ist der Name für ein Objekt, daß während der Programmausführung durch Unifikation mit Konstanten und Strukturen instanziiert werden kann. Eine Variable kann zudem mit einer anderen Variablen unifiziert werden. Variablennamen beginnen mit einem Großbuchstaben oder dem Unterstrich bei anonymen Variablen.
Wissensbasis	Menge aller Fakten und Regeln, auf die der Prolog-Interpreter zugreifen kann, welche mittels <i>consult</i> oder <i>assert</i> in die Wissensbasis aufgenommen wurden.
Wurzel-Knoten-Methode	Standard-Methode zur Baumbearbeitung. Man bearbeitet zunächst die Wurzel, dann alle damit verbundenen Knoten oder erst alle Knoten und dann die Wurzel. Der Zugriff auf Wurzel und Knoten ist mit dem univ-Operator „=..“ möglich.
Ziel	Ein Term, dessen Beweis durch Resolution und Unifikation gesucht wird.
zusammengesetzter Term	Ein Objekt aus Funktor und einem oder mehreren Argumenten.

Anhang B

Literaturverzeichnis

- [Aho1] Aho, A.: Compilerbau, Teil 1. Bonn: Addison-Wesley, 1988.
- [Bah1] CityFahrplan ICE/EC/IC, 1994/95. Mainz: Deutsche Bahn AG, 1994.
- [Bau1] Baumann, R.: Didaktik der Informatik. Stuttgart: Klett, 1990.
- [Bau2] Baumann, R.: Informatik für die Sekundarstufe II, Band 1. Stuttgart: Klett, 1992.
- [Bau3] Baumann, R.: Informatik für die Sekundarstufe II, Band 2. Stuttgart: Klett, 1992.
- [Bau4] Baumann, R.: Prädikative Denk- und Programmiermethoden im Informatikunterricht, Teil 1. In: LOG IN, 13 (1993), H. 3, S. 55-57. Berlin: LOG IN Verlag, 1993.
- [Bau5] Baumann, R.: Prädikative Denk- und Programmiermethoden im Informatikunterricht, Teil 2. In: LOG IN, 13 (1993), H. 4, S. 56-60. Berlin: LOG IN Verlag, 1993.
- [Bau6] Baumann, R.: Prädikative Denk- und Programmiermethoden im Informatikunterricht, Teil 3. In: LOG IN, 13 (1993), H. 5, S. 52-58. Berlin: LOG IN Verlag, 1993.
- [Bau7] Baumann, R.: Prüfungsfachwahl - Ein Programmierprojekt in PROLOG. In: LOG IN, 13 (1993), H. 4, S. 34-39. Berlin: LOG IN Verlag, 1993.
- [Bau8] Baumann, R.: Programmierung eines kleinen Expertensystems, Teil 2. In: LOG IN, 9 (1989), H. 5, S. 38-42. München: Oldenbourg, 1989.
- [Bay1] Bayer, K.: Computerlinguistik im Unterricht. In: LOG IN, 12 (1992), H. 1, S. 39-49. München: Oldenbourg, 1992.
- [Bel1] Belli, F.: Einführung in die logische Programmierung mit PROLOG. Mannheim: BI-Wissenschaftsverlag, 1988.
- [Bra1] Bratko, I.: Prolog, Programmierung für Künstliche Intelligenz. Bonn: Addison-Wesley, 1987.
- [Bur1] Burkert, J.: Informatik heute, Algorithmen und Datenstrukturen, Band 2. Hannover: Schroedel, Schöningh, 1988.
- [Bus1] Bussmann, H.: Computer und Allgemeinbildung. In: Neue Sammlung, 27 (1987), H. 1, S. 2-39.
- [Clo1] Clocksin, W. F.: Programmieren in PROLOG. Berlin: Springer, 1990.

- [Dei1] Deißler, R., Stamm, K.: Zugauskünfte per Computer - Ein Unterrichtsprojekt, Teil 1. In: LOG IN, 10 (1990), H. 4, S. 30-34. Teil 2: In: LOG IN, 10 (1990), H. 5, S. 52-55. München: Oldenbourg, 1990.
- [Fuc1] Fuchs, N.: Kurs in Logischer Programmierung. Berlin: Springer, 1990.
- [GI1] GI-Empfehlungen für das Fach Informatik in der Sekundarstufe II allgemeinbildender Schulen. Beilage in: LOG IN, 13 (1993), H. 3. Berlin: LOG IN Verlag, 1993.
- [Göh1] Göhner, H., Hafenbrak, B.: Arbeitsbuch PROLOG. Bonn: Dümmler, 1995.
- [Gra1] Duden Band 4, Grammatik. Mannheim: Duden-Verlag, 4. Auflage, 1984.
- [Her1] Hermes, A.: Von der Wissensbasis zum Expertensystem, Eine Einführung mit PROLOG. In: Informatik betrifft uns, Themenband: Aachen: Bergmoser+Höllner.
- [Hop1] Hopcroft, Ullmann: Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie. Bonn: Addison-Wesley, 1990.
- [Kef1] O'Keefe, R.: The Craft of Prolog. Cambridge: The MIT Press, 1990.
- [Kle1] Kleine Büning, H.: PROLOG. Stuttgart: Teubner, 1988.
- [Knü1] Knülle-Wenzel, A.: Handbuch und Tutorial zu fix-PROLOG. Bochum: Selbstverlag, 1991.
- [Lan1] Langbein, K.: Bittere Pillen, Nutzen und Risiken der Arzneimittel. Köln: Kiepenheuer&Witsch, 65. Auflage, 1993.
- [Leh1] Lehmann, G.: Ziele im Informatik-Unterricht. In: LOG IN, 12 (1992), H. 1, S. 26-30. München: Oldenbourg, 1992.
- [Mod1] Modrow, E.: Zur Didaktik des Informatik-Unterrichts, Band 1. Bonn: Dümmler, 1991.
- [Mod2] Modrow, E.: Automaten, Schaltwerke und Sprachen. Bonn: Dümmler, 1986.
- [Ott1] Otto, H. M.: ProLog-Puzzles. Bonn: Dümmler, 1991.
- [Sav1] Savory, S. E.: Grundlagen von Expertensystemen. München: Oldenbourg, 1990.
- [Ste1] Sterlin, L.: PROLOG, Fortgeschrittene Programmieretechniken. Bonn: Addison-Wesley, 1988.
- [Sub1] Schubert, S.: Fachdidaktische Fragen der Schulinformatik und (un)mögliche Antworten. In: Gorny, P. (Hrsg.): Informatik und Schule 1991, Wege zur Vielfalt beim Lehren und Lernen, GI-Fachtagung, Oldenburg 1991. Berlin: Springer, 1991.

- [Süt1] Schütz: PROLOG, Prozedurale Aspekte. In: Praxis der Mathematik, 1/34, Jg. 1992.
- [Sul1] Schult, T. J.: Cyc's Fiction. In: c't, 1994, H. 5, S. 94-103. Hannover: Heise, 1994.
- [Wie1] Wielemaker, J.: SWI-Prolog 1.6, Reference Manual. Amsterdam: University of Amsterdam, 1992.
- [Win1] Winston, P.: Künstliche Intelligenz. Bonn: Addison-Wesley, 1987.

Anhang C

Abbildungsverzeichnis

- Abb. 2-1 Anfrage- und Teilzielknoten 26
- Abb. 2-2 Oder-Knoten 26
- Abb. 2-3 Und-Knoten 27
- Abb. 2-4 Und-Oder-Beweisbaum 27
- Abb. 2-5 vereinfachter Und-Oder-Beweisbaum 29
- Abb. 2-6 Lösungssuche für die Anfrage ?- tochter(maria,Elter). 30
- Abb. 2-7 Backtracking im Und-Oder-Beweisbaum 31
- Abb. 3-1 Grundstruktur des Vierport-Modells 37
- Abb. 3-2 die Anfrage ?-satz(der, jaeger, bellt) im Vierport-Modell 37
- Abb. 5-1 Visualisierung einer Liste mit sieben Elementen 47
- Abb. 5-2 Visualisierung der Listenstruktur [Element|Restliste] 48
- Abb. 5-3 Struktur als Baum 50
- Abb. 5-4 Rechenterm als Baum 50
- Abb. 5-5 Visualisierung der Liste [a, b] 51
- Abb. 5-6 Funktor und Argumente in der Baumdarstellung 51
- Abb. 7-1 Aufbau einer Buchstabenliste mit Hilfe einer unvollständigen Datenstruktur 69
- Abb. 8-1 Visualisierung des Cut mit ProVisor 74
- Abb. 8-2 Visualisierung des Cut im Vierport-Modell 74
- Abb. 8-3 Globale und lokale Wirkung des Cut im konventionellen Suchbaum 75
- Abb. 8-4 Wirkung des Cut am Beispiel des Prädikats kind 76
- Abb. 9-1 Instanziierung von Variablen bei einer Unifikation 84
- Abb. 9-2 Ein Unifikationsschritt 85
- Abb. 9-3 Dynamische Watch-Terme in ProVisor 86
- Abb. 9-4 Termgenerierung durch mehrfaches Unifizieren 87
- Abb. 9-5 Visualisierung von append kurz vor der ersten Lösung 87
- Abb. 9-6 Visualisierung von append 1. Lösung 88
- Abb. 9-7 Visualisierung von append kurz vor der zweiten Lösung 88
- Abb. 9-8 Visualisierung von append 2. Lösung 89
- Abb. 9-9 Visualisierung von append Übersicht zu allen Lösungen 89
- Abb. 11-1 Prädikate zur Manipulation der Wissensbasis 104
- Abb. 11-2 Doppelte Rekursion bei den Fibonacci-Zahlen 107
- Abb. 11-3 Begriffshierarchie für Säugetierarten 111
- Abb. 12-1 ICE-Streckennetz 117
- Abb. 12-2 ER-Diagramm des ICE-Auskunftsystems 118
- Abb. 13-1 ER-Diagramm des Auskunfts- und Reisebuchungssystems 132

- Abb. 14-1 ungerichteter, bewerteter Graph 142
- Abb. 14-2 Wellenfronten der Breitensuche 144
- Abb. 14-3 Ein Anfangszustand im Hüpf-Schiebe-Puzzle 148
- Abb. 14-4 Zielzustand im Hüpf-Schiebe-Puzzle 149
- Abb. 15-1 Hierarchie der Prolog-Terme 165
- Abb. 15-2 Baumdarstellung eines zusammengesetzten Terms 166
- Abb. 15-3 Baumdarstellung eines arithmetischen Terms 166
- Abb. 15-4 Baumdarstellung einer Liste 166
- Abb. 15-5 Baumdarstellung eines Fakts 167
- Abb. 15-6 Baumdarstellung einer Regel 167
- Abb. 15-7 Umwandlung einer Struktur 170
- Abb. 15-8 mit dem univ-Operator in eine Liste 170
- Abb. 16-1 Umwandlung eines iterativen in ein rekursives Syntaxdiagramm 185
- Abb. 16-2 Ableitungsbaum für 0-1-Wörter 189
- Abb. 16-3 Ableitungsbaum für das Wort 0011 190
- Abb. 17-1 Schaltnetz für einen Halbaddierer 195
- Abb. 17-2 Prüfschaltung für gültige Tetraden 196
- Abb. 17-3 Gatterschaltung eines RS-Flipflops 197
- Abb. 17-4 Zustandsgraph eines Getränkeautomaten 199
- Abb. 17-5 Zustandsgraph einer Aufzugssteuerung 200
- Abb. 17-6 Syntaxdiagramm für Bezeichner 201
- Abb. 17-7 Zustandsgraph eines Akzeptors für Bezeichner 201
- Abb. 17-8 Syntaxdiagramm für Real-Zahlen 202
- Abb. 17-9 Zustandsgraph eines Akzeptors für Real-Zahlen 202
- Abb. 17-10 fehlerhafter Akzeptor für die Zeichenkette pen 211
- Abb. 17-11 Beispiel eines nichtdeterministischen endlichen Automaten 211
- Abb. 17-12 deterministischer Akzeptor für pen 212
- Abb. 17-13 Zustandsgraph mit ϵ -Übergängen 213
- Abb. 17-14 äquivalenter Zustandsgraph ohne ϵ -Übergänge 213
- Abb. 17-15 Verkettung (Sequenz) von Akzeptoren 214
- Abb. 17-16 Vereinigung (Selektion) von Akzeptoren 214
- Abb. 17-17 beliebige Wiederholung (Iteration) von Akzeptoren 214
- Abb. 17-18 Zustandsgraph für Akzeptor bis $n=3$ 216
- Abb. 17-19 Syntaxdiagramm für Bezeichner 220
- Abb. 17-20 Grundstrukturen im Syntaxdiagramm für Bezeichner 220
- Abb. 18-1 Prinzip des Kellerspeichers 225
- Abb. 18-2 Modell eines Kellerautomaten 226
- Abb. 18-3 Zustandsgraph eines Kellerautomaten für die Sprache $a^n b^n$ 228
- Abb. 18-4 Syntaxdiagramme für arithmetische Ausdrücke 229
- Abb. 18-5 Zustandsgraph eines Kellerautomaten für geklammerte Zahlen 229

- Abb. 18-6 Zustandsgraph eines Kellerautomaten für arithmetische Ausdrücke 229
- Abb. 18-7 Zustandsgraph eines Kellerautomaten für Palindrome 230
- Abb. 19-1 Konzeption einer Turingmaschine 245
- Abb. 19-2 Struktogramm zur Arbeitsweise einer Turingmaschine 247
- Abb. 19-3 Modellierung des Arbeitsbandes einer Turingmaschine in Prolog 252
- Abb. 20-1 rekursives Syntaxdiagramm für Strichlisten 261
- Abb. 20-2 iteratives Syntaxdiagramm für Strichlisten 261
- Abb. 20-3 Syntaxdiagramme der erweiterten Strichlisten-Grammatik 262
- Abb. 20-4 Syntaxdiagramme von mini-LOGO 263
- Abb. 20-5 Beispiel eines Parsebaums 265
- Abb. 20-6 Interpretation eines mini-LOGO Programmes 266
- Abb. 20-7 Syntaxdiagramme von mini-PASCAL 267
- Abb. 20-8 umgeformte Syntaxdiagramme 272
- Abb. 20-9 Ablaufdiagramm für eine While-Anweisung 275
- Abb. 21-1 Ableitungsbaum für das Wort 0011 284
- Abb. 21-2 Ableitungsbaum einer Substantivgruppe 285
- Abb. 21-3 Ableitungsbaum für den Term $a+b*c$ 286
- Abb. 21-4 Ableitungsbaum für den Satz Peter kauft die Kartoffeln 286
- Abb. 21-5 Bildschirmmaske von SIMPEL 290
- Abb. 21-6 Bearbeitung der Frage Wo ist er? 292
- Abb. 22-1 Architektur eines Expertensystems 300
- Abb. 22-2 Regel 3 als Baumstruktur 307
- Abb. 23-1 Spur im Suchbaum 321

Anhang D

Tabellenverzeichnis

Tab. 1-1	Auswertung eines Fragebogens zu Prolog	2
Tab. 1-2	Klassifikation und Einsatzmöglichkeiten der Kapitel	17
Tab. 6-1	Zusammenstellung von Vergleichsoperatoren	63
Tab. 9-1	Zusammenfassung möglicher Unifikationspartner	91
Tab. 14-1	Untersuchungsergebnisse zur Bewertung der Suchverfahren beim Hüpf-Schiebe-Puzzle	154
Tab. 14-2	Zusammenstellung der Züge im 8er-Puzzle	157
Tab. 15-1	Vergleichsoperatoren für Terme	168
Tab. 16-1	Produktionsformen der Chomsky-Hierarchie	182
Tab. 16-2	Gegenüberstellung äquivalenter Kontrollstrukturen, Grammatikregeln und Syntaxdiagramme	184
Tab. 17-1	Schaltwerttabellen der logischen Grundschaltungen	193
Tab. 17-2	Addition zweier Dualzahlen	195
Tab. 17-3	Codierung von Ziffern durch Tetraden	196
Tab. 17-4	Syntaxdiagramme und Akzeptoren	221
Tab. 18-1	Kellerautomat für die Sprache $anbn$	227
Tab. 18-2	Grammatiken und zugehörige Maschinenmodelle	242
Tab. 19-1	Diagonalisierung der Funktionen $f: N \rightarrow N$	257
Tab. 22-1	Schmerz- und fiebersenkende Mittel	302
Tab. 22-2	Starke Schmerzmittel	303
Tab. 22-3	Kopfschmerz- und Migränemittel	304
Tab. 22-4	Krampflösende Mittel (Spasmolytika)	305
Tab. 23-1	Operatoren der dreiwertigen Logik	319

Anhang E

Index

- 53
 ! 73
 + 53
 , 22
 . 52
 :- 22; 105
 : 119
 ; 23
 = 64
 =. 169
 == 23
 -> 79
 ? 53
 ?- 19
 @ 168
 [47
 ^ 100
 | 48
 0'<Zeichen> 67
 1-n 118; 132

A

Abfahrtsplan 123
 Ablaufverfolgung 35
 ableiten 97
 Ableitung 178
 Ableitungsbaum 190; 284
 Ableitungsregeln 97
 Akkumulatortechnik 296
 Akzeptor 200; 207
 Vereinigung 214
 Verkettung 214

all 78
 Allgemeinbildung 8
 Alltagswissen 292
 Anfangsunterricht 2
 Anfangszustand 203; 231; 250
 Anfragen 24
 anonyme Variable 24
 append 55; 85
 Arbeitsband 245; 252
 arg 169
 Argumente 53; 165
 Arität 22
 Arithmetik 63
 Arithmetische Ausdrücke 229
 Interpretation 237; 274
 Parser 233; 272
 Übersetzung 277
 assert 104
 Atom 165
 atomic 165
 Attribute 131
 Ausgabe 67
 Ausgabealphabet 203
 auswerten 172
 Automat 198
 Definition 203
 erkennender 200
 Grenzen 216
 Modellierung 203
 nichtdeterministisch 211
 sprechender 290
 Syntaxdiagramme für 220
 übersetzender 200
 unendlicher 225

B

Backtracking 30; 73; 98
Backus-Naur-Form 183
Baum 141
Baumstrukturen 50
bedingter Sprung 275
Benutzungsschnittstelle 131; 133
Berechenbarkeit 255
between 64
Bewertungsfunktion 152; 155
Bezeichnernamen 201
Beziehungen 131
Binomialkoeffizienten 113
Blockwelt 33
BNF-Form 183
Boxen-Modell 37
Breitensuche 144; 151; 159; 187
bruder 22

C

Call 35; 80
Cantor 256
CHARSTR 206
Chomsky 181
Chomsky-Hierarchie 182
chr 205
Churchsche-These 256
clause 105; 110
Clipboard 19
Code-Schablone 275
Compiler 191; 275
compoundterm 165
consult 19; 32; 104
countvar 171
CTYPES 207
Cut 73; 99
Cut-Fail-Kombination 80

D

Datenbanken 41
Datenstrukturen 50
deklaratives Programmieren 97
deterministisch 73; 106
Diagonalisierungsverfahren 256
direktes Kellern 235
Disjunktion 23
display 52; 67; 167
div 63
dreiwertige Logik 319
drucke 175
dualzahl 217

E

einfacheableitung 187
einfacherAusdruck 271
Eingabe 67
Eingabealphabet 203
ELIZA 293
elternteil 22
endlicher Automat 198
End-Rekursion 68
Endzustände 203; 231
Entfalten 217; 236
Entity-Relationship-Diagramm
118; 131
Entscheidbarkeit 191
Entscheidungsfragen 24
 ϵ -Übergänge 213
Ergänzungsfragen 24
Erklärungskomponente 300
erzeugte Sprache 187; 209; 234
Exit 35
Expertensystem 299
Expertensystemshell 300; 317

F

Fail 35; 78
 Fakten 21
 Fakultätsberechnung 279
 Familienbeziehungen 22
 Fenster 19
fiæ-Prolog 19
 Fibonaccizahlen 107
 findall 109; 110; 145
 formale Sprache 187
 format 71
 Fragekomponente 300
 functor 169
 Funktor 21; 165

G

Gatter 193
 get_single_char 67
 get0 67
 GI-Empfehlungen 8
 Grammatik 177; 181
 0-1-Wörter 180
 Arithmetische Ausdrücke 179
 Chomsky-Hierarchie 182
 deutsche Sätze 178
 Modellierung in Prolog 185
 Palindrome 180
 Regeln 178
 Umwandlung in Syntaxdiagramme 183
 Graph 141
 bewertet 142; 147
 gerichtet 142
 Zustandsraum 148
 GREP 223
 Gültigkeitsbereich 23

H

haelt_an 257
 Halbaddierer 195
 Halteproblem 257
 Hardware 193
 Heuristik 146
 Heuristische Suche 127; 146;
 152; 160
 Hüpf-Schiebe-Puzzle 148

I

ICE-Auskunftssystem 117
 If-Then-Else 79
 Inferenz-Algorithmus 318
 Inferenzmaschine 300; 317
 Infix-Notation 51
 Infixoperatoren 166
 Inorder 56
 Input-Fenster 19
 Instanziierung 84
 integer 165
 Intellektik 5
 Interpreter 261
 is 64
 is_alpha 71; 207
 is_digit 207

J

Join 42

K

Kanten 142
 Kelleralphabet 231
 Kellerautomat 225
 Arbeitsweise 227
 Definition 231

Grenzen 242
 Modell 226
 Modellierung 232
 nichtdeterministisch 281
 Zustandsgraph 228
 Kellerooperationen 226
 Kellerspeicher 225
 Kettenregel 101
 Klauseln 22
 Konklusion 317
 Konstante 165
 kontextfrei 182
 kontextfreie Grammatik 288
 kontextsensitiv 182
 Kontrollstrukturen 184
 Kopf-Rest-Methode 56
 Künstliche Intelligenz 97; 141

L

L(G) 187
 Labyrinth 61
 last 138
 length 57
 lese_string 70
 lexikalische Analyse 200
 LIFO-Prinzip 226
 linie_zeichnen 71
 linksbuendig 71
 linksrekursiv 188
 Grammatik 27
 liste 165
 Listen 47; 165
 Listen-Operator 48
 Logische Programmierung 10

M

Maschinelle Sprachverarbeitung
 10; 281
 member 53
 Mengenprädikate 109
 Menüsystem 68
 Metainterpreter 38; 306
 mini-FISCH 280
 mini-LOGO 263
 mini-PASCAL 267
 mini-PLOT 280
 mod 63
m-rekursive Funktionen 256

N

name 69
 natürliche Sprache 287
 natürlichsprachliche Systeme 292
 Netzwerk 141
 Nimm 114
 nl 67
 n-m 132
 no 24
 nonvar 165
 not 80

O

Oder-Verknüpfung 193
 once 78
 Operatoren 100
 ord 205
 Output-Fenster 19

P

Palindrome 180; 230; 234

Parallelschaltung 193
 Parsebaum 264; 265
 Parser 191; 261; 270; 284
 Pascal 1
 Pfeil-Auf-Taste 25
 pop 226
 Postorder 56
 potenz 100
 Potenzfunktionen 100
 Prädikat 22
 Präfix-Notation 51
 Präfixoperatoren 166
 Prämissen 317
 Preorder 56
 Prioritätswarteschlange 147
 Problemlösen 148
 Problemlösungskomponente 300
 Produktionen 181
 Projektion 42
 Prolog
 Einführung 12
 Kontrollstrukturen 6
 Vorteile 1; 5
 Protokoll 36
 ProVisor 25
 Punktschreibweise 50
 push 226
 put 67
 Puzzle
 8er-Puzzle 156
 Hüpf-Schiebe-Puzzle 148

R

read 67
 readln 70; 269
 Real-Zahlen 202
 rechtsbuendig 71
 rechtsrekursiv 188
 reconsult 19; 32; 104

Redo 35
 Regel 22
 Regelkopf 22
 Regelsystem 111
 Medikamente 115
 Säugetiere 111
 Regelwissen 309
 regulär 182
 reguläre Ausdrücke 215
 Reihenschaltung 193
 rekursiv 23
 Relationen 118
 repeat 78
 Resolution 29; 73; 98
 retract 105
 retractall 105
 reverse 57
 römische Zahlen 280
 RS-Flipflop 197
 rückgekoppelte Schaltungen 197
 rückwärts-verkettendes System
 317

S

Scanner 269
 Schmerzmittel 301
 Schreib-/Lesekopf 245; 246
 selbstdefinierter Operatoren 100
 Selektion 41
 seltsam 257
 Semantik 6
 Semi-Entscheidbarkeit 191
 SIMPEL 290
 skip 67
 sort 123; 136
 Speicher 197
 Speichereinheit 225
 Sprachverarbeitung 177; 281
 Spuren 38

- Startsymbol 178; 181
 Stelligkeit 22
 Steuereinheit 225
 Stimulus 296
 Strategiespiele 141
 Strichlisten 261
 Strings 207
 Strukturen 165
 Strukturoperatoren 169
 Strukturuntersuchung 169
 Substitution 83
 succ 64
 Suchfront 144
 Suchverfahren 141
 Bewertung 154; 161
 SWI-Prolog 20
 Symbolisches Differenzieren 97
 syntaktische Analyse 177; 190
 Syntax 6
 Syntaxdiagramm
 Umwandlung iterativ-rekursiv
 185
 Syntaxdiagramme 183
 Synthese-Problem 187
 Systembibliothek 70; 207; 269
-
- T**
- tab 67; 175
 Technische Informatik 193
 Teile und Herrsche 98
 Term 165; 271
 Analyse 165
 Klassifikation 165
 Standardordnung 168
 Vergleichsoperatoren 119; 168
 Visualisierung 167
 Terminale 181
 Termuntersuchungen
 Beispiele 171
 Tetraden 196
 Theoretische Informatik 193
 Tiefensuche 143; 150
 beschränkte 158
 top 226
 Trace 35
 Transduktor 200
 Turing 245
 Turing-Berechenbarkeit 255
 Turingmaschine 245
 Arbeitsband 252
 Arbeitsweise 246
 Definition 250
 Grenzen 256
 Konzeption 245
 Kopplung 249
 Modellierung 251
 nichtdeterministisch 250
 rechnende 248
 TV-SWI-Prolog 20
-
- Ü**
- Überföhrungsfunktion 250
 Übergangsfunktion 203; 231
 Übersetzer 275
-
- U**
- Und-Oder-Beweisbaum 25
 abstrakt 34
 aktiver Pfad 28
 Blatt 27
 Kanten 26
 Knoten 26
 vereinfacht 28
 Und-Verknüpfung 193
 Unifikation 29; 83; 91; 98

univ-Operator 101; 170; 285;
287
unlösbare Probleme 256
unvollständigen Datenstruktur 69

V

var 165
Variable 21; 165
Veranschaulichung 35
vereinfachen 102
Vergleichsausdrücke 277
Vergleichsoperatoren 63
Verwaltungsfunktionen 133
Verwaltungskomponenten 131
Vierport-Modell 37; 74
Visualisierung 25
Volladdierer 195
vorfahr 23

W

Warteschlange 145
Warum-Frage 320
Watch-Term 86
Wenn-Dann-Regel 317
While-Anweisung 275
Wie-Frage 322
Wissensakquisition 299
Wissensrepräsentation 299; 306
wohlgeformte Klammerausdrücke
243
Wortproblem 191; 281
Wort-Problem 187
write 67
Wurzel-Knoten-Methode 173

Y

yes 24

Z

zentriert 71
zerlegen 205
Zufallszahlen 106
Zugauskunft 125
Zugbegleiter 121
zusammengesetzter Term 165
zusammensetzen 205
Zustandsgraph 198
Zustandsraumgraph 148
Zustandsübergang 198
Zyklus 143
Zyklustest 143

