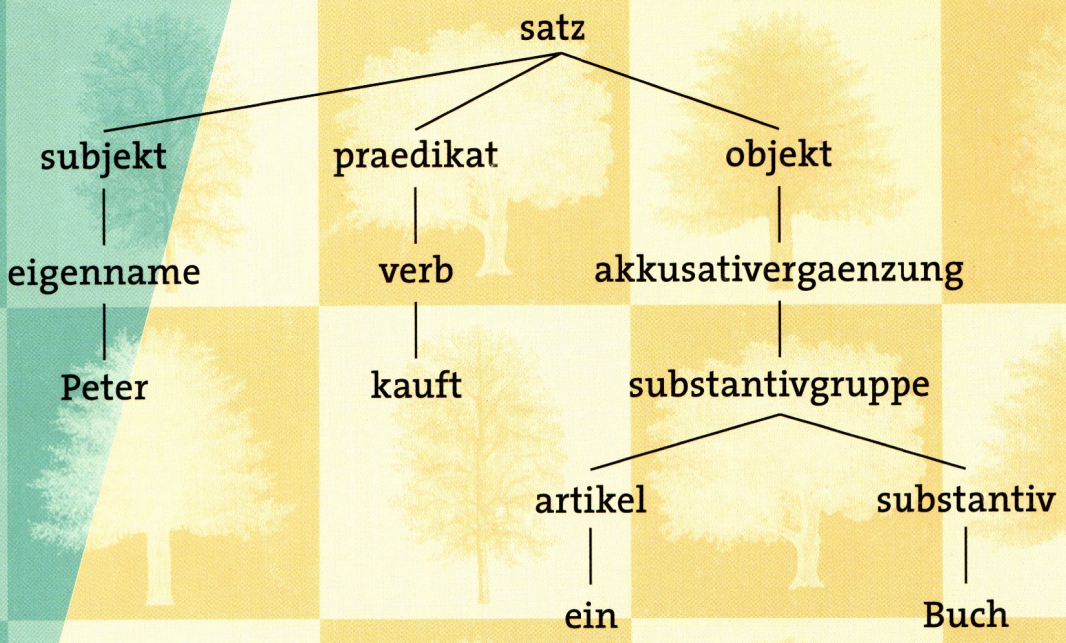
Hessisches Kultusministerium
Amt für Lehrerbildung

HESSEN

Informatik mit Prolog

Sekundarstufe II

Bildungsland Hessen



Materialien zum Unterricht

Materialien zum Unterricht, Sekundarstufe II, Informatik

Informatik mit Prolog

Autor:

Gerhard Röhner

Best.-Nr. 06000

Amt für Lehrerbildung (AfL), Frankfurt 2007

Herausgeber: Amt für Lehrerbildung (AfL)
Publikationsmanagement
Stuttgarter Str. 18-24
60329 Frankfurt am Main

Diese Veröffentlichung wird im Auftrag des Hessischen Kultusministeriums (www.kultusministerium.hessen.de) herausgegeben; sie stellt jedoch keine verbindliche, amtliche Verlautbarung der Hessischen Kultusministerin dar; sie will vielmehr die Diskussion um die behandelten Themen anregen und zur Weiterentwicklung des hessischen Schulwesens beitragen.

Digitale Fassung

Gerhard Röhner

ISBN: 3-88327-372-4
3. vollständig neu überarbeitete Auflage 2007
Best.-Nr.: 06000

Titelgestaltung: Sibylle Tietze • Frankfurt am Main
Satz/Umbruch : Gerhard Röhner • Dieburg
Druck- und Bindearbeiten: Druckerei des Amts für Lehrerbildung, Fuldatal

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

Vorwort zur ersten Auflage

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 allgemein bildenden 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 allgemein bildenden 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“ (V. 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, 1995

Vorwort zur zweiten Auflage

Die erste Auflage des 1995 erschienen ursprünglich zweibändigen Buches „Informatik mit Prolog“ ist mittlerweile vergriffen. In den vergangenen Jahren hat sich die Informatik stürmisch weiter entwickelt. Das Internet und seine Programmiersprache Java haben erheblichen Einfluss auch auf die Schulinformatik genommen. Die Bedeutung von Prolog hat abgenommen, wohl auch weil die Versprechungen der KI nur sehr zögerlich eingelöst werden, während andere Bereiche sich dynamisch entfalten. Der Bedeutungsverlust drückt sich unter anderem darin aus, dass zurzeit nur sehr wenige aktuelle Buchtitel zu Prolog auf dem Markt sind. Neuerscheinungen von Prolog-Büchern für die Schule gibt es seit Jahren nicht mehr. Wieso also diese aktualisierte zweite Auflage von „Informatik mit Prolog“?

Die Nachfrage nach diesen Materialien besteht weiterhin. Sie ergibt sich unter anderem daraus, dass in einigen Bundesländern die Arbeit mit der deklarativen Programmiersprache Prolog verbindlich im Lehrplan vorgegeben ist. Auch

im neuen hessischen Lehrplan Informatik gehört die Auseinandersetzung mit Prolog wieder zu den verbindlichen Inhalten für den Leistungskurs. In Kapitel 1 finden Sie die relevanten Gründe, die für diese Vorgabe sprechen. Daran hat sich seit der ersten Auflage nichts geändert.

Seit letztem Jahr gibt es wieder eine Lehrerweiterbildung Informatik in Hessen. Die Teilnehmer setzen sich in den Ausbildungsveranstaltungen auch mit Prolog auseinander. Das ist natürlich ein guter Grund, eine neue Auflage herauszubringen und den Schulen ein erprobtes Buch für den Informatikunterricht anzubieten.

In der zweiten Auflage sind die bekannten Fehler der ersten Auflage beseitigt und hoffentlich nicht zu viele neue hinzugekommen. Bezüge zu den alten Prolog-Systemen für DOS wurden durch konsequenten Bezug auf die aktuelle SWI-Prolog Version für Windows ersetzt. Soweit möglich wurden auch die Verbesserungen, die SWI-Prolog in den letzten Jahren erfahren hat, in den Beispielen genutzt. So konnten einige in der ersten Auflage genutzte Hilfsmittel durch neuere Systemprädikate ersetzt werden, was eine erhebliche Erleichterung beim Testen von Lösungen mit sich bringt. Zum Beispiel wurde bei den Automatenmodellen von der klassischen Art Zeichenketten als Liste von ASCII-Codes zu bearbeiten auf die viel näher liegende Weise dies mit Listen von Zeichen zu erledigen umgestellt.

Inhaltliche Erweiterungen gibt es besonders in den Kapiteln zur theoretischen Informatik, die vom Autor mehrmals mit Erfolg im Unterricht eingesetzt wurden. Bei den Grammatiken wird neben den Linksableitungen jetzt auch die Arbeit mit Ableitungsbäumen gut unterstützt. Im Automatenkapitel wurden einige Verweise auf Ergebnisse der theoretischen Informatik durch explizite Angabe der Verfahren ersetzt. So wird beispielsweise die Umwandlung nichtdeterministischer in deterministische Automaten dargestellt. Bei den Kellerautomaten wurde ein neues Verfahren ergänzt, bei dem aus Syntaxdiagrammen durch Entrekursivierung mittels push/pop-Operationen Kellerautomaten konstruiert werden können. Der Beweis des Halteproblems wurde didaktisch auf das spezielle Halteproblem reduziert. Beim umfangreichen mini-Pascal-Projekt gibt es als Erweiterung die Ausführung des compilierten Programms durch Simulation in VISIS (Visualisierung eines Intel-Systems). Über diesen Weg kommt man ohne Brüche von der theoretischen zur technischen Informatik, die im neuen Lehrplan als Wahlthema vertreten ist.

Im Aufgabenteil der wurden einige umfangreichere Aufgaben zur theoretischen Informatik ergänzt, die teilweise Klausuren und Abituraufgaben entstammen.

In den letzten Jahren hat der Autor auch einige Software weiter entwickelt, um bessere Rahmenbedingungen für die Arbeit mit Prolog zu schaffen. Zur Verfügung stehen Prolog-Module zum Darstellen von Prolog-Termen (zeichne_term), eine Turtle für Prolog und die didaktische Reduktion des trace-Modus in Form des spur-Prädikats.

Am aufwändigsten war allerdings die Neuentwicklung des SWI-Prolog-Editors, mit dem in einer zeitgemäßen Entwicklungsumgebung Prolog-Programme geschrieben, getestet und genutzt werden können. Wie beim Vorgängersystem Swing-Prolog wird auch hier SWI-Prolog als Subsystem genutzt.

Aus dem ursprünglich zweibändigen Werk ist jetzt ein einbändiges entstanden, was für den Einsatz in der Schule einfacher ist. Bis auf die Kapitel zu Expertensystemen ist aber der inhaltliche Umfang erhalten geblieben. Eingespart wurde der aufwändige Satz mit Marginalspalte. Der Zweispaltensatz bei kleinerer Schrift führt zu einer erheblichen Reduktion der Seitenzahl und spart damit Ressourcen.

Gerhard Röhner, im Juli 2002

Vorwort zur dritten Auflage

Die zweite Auflage von Informatik mit Prolog war innerhalb weniger Jahre vergriffen. Die Anfragen von Lehrerinnen und Lehrern, sowie aus dem Hochschulbereich machten aber deutlich, dass weiterhin Bedarf an diesem Buch besteht. Die dritte Auflage bot die Gelegenheit, die zwischenzeitlich festgestellten Fehler zu korrigieren und Erweiterungen vorzunehmen.

Mit dem neuen Kapitel zu Logikrätseln wird ein Anspruch von *Programming in Logic* eingelöst. Viele Beispiele unterschiedlichster Art machen deutlich, dass sich logische Rätsel gut mit Prolog lösen lassen. Dabei kommt ganz bewusst der deklarative Ansatz von Prolog zum Zuge, und Effizienzverbesserungen durch Einsatz imperativer Methoden bleiben außen vor.

Ein weiteres neues Kapitel thematisiert die GUI-Programmierung mit Prolog auf der Basis von XPCE. Anhand einfacher Beispiele wird gezeigt, wie man eine GUI-Oberfläche entwickeln kann und wie der Austausch zwischen einer GUI-Oberfläche und dem Prolog-Untergrund stattfindet. Das Kapitel soll nur einen Einstieg in die Nutzung von GUI-Oberflächen mit Prologprogrammen ermöglichen und bei Bedarf die Realisierung einfacher XPCE/Prolog-Pro-

gramme ermöglichen. Die Entwicklung komplexerer Anwendungen ist nur durch intensives Studium der XCPE-Dokumentation möglich, denn das objektorientierte XPCE-System umfasst zurzeit 160 Klassen mit über 2700 Methoden.

Bei der Überarbeitung für die dritte Auflage wurden in vielen Kapitel Detailverbesserungen durchgeführt. Bei den Listen wurde beispielsweise ein Unterkapitel zur Akkumulatortechnik ergänzt. Das Kapitel zur Ein- und Ausgabe beschreibt jetzt alternativ zum alten Edinburgh-Standard auch die Nutzung von Streams für den Dateizugriff, und in das Kapitel zum Cut wurden bessere Beispiele aufgenommen. Darüber hinaus wurden viele Kapitel um zusätzliche Aufgaben ergänzt.

Seit der letzten Aufgabe hat sich auch der SWI-Prolog-Editor weiter entwickelt. Automatische Updates, Installation als portable Anwendung auf USB-Sticks, Einbindung des GUI-Debuggers und Spachunterstützung seien hier exemplarisch genannt. Erwähnenswert ist auch, dass Sie nun im didaktischen Werkzeug *zeichne_term* die Knoten von Bäumen interaktiv verschieben können. Ein Doppelklick auf den Knoten ordnet dann seine Nachfolgerknoten neu an.

Gerhard Röhner, im Juli 2007

Inhaltsverzeichnis

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.
- 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.
- 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.

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

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 Problembedingungen 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]:

- 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.
- Wegen des vorwiegend prädikativen Denkstils und seiner Möglichkeiten zur Wissensrepräsentation lassen sich zahlreiche Querverbindungen zu anderen Schulfächern herstellen.
- 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 (siehe Tabelle 1-1). 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.

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.

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		

Tabelle 1-1 Auswertung eines Fragebogens zu Prolog

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 allgemein

bildende 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.

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 kostenloser 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:

Clocksin/Mellish in [Clo1]:

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

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.

Bratko in [Bra1]:

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

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:

```
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:

```
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 erhebliche 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.

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 kennen gelernt hat.

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 Schlüsselworte 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 liegen gebliebenes 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 nicht in ausreichender Vielfalt.

Das vorliegende Buch soll zusammen mit der Visualisierungs-Software ProVisor, dem Prolog-Interpreter SWI-Prolog und dem SWI-Prolog-Editor 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 des didaktischen Tools *zeichne_term* auf der Basis der grafischen Erweiterung XPCE vorgenommen. Das Vierportmodell stellt eine weitere 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 imperativen Stil geprägt zu echten Prolog-Lösungen kommen kann. Über diese Brücke ist ein einfacher Weg von Pascal bzw. Java nach Prolog möglich.

1.3 Hinweise für den Unterricht

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.

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.

Kapitel 4 bringt einfache Datenbankkonzepte mit Prolog und ist aufgrund des geringen Anforderungsniveaus wie Kapitel 2 für die Einführung in Prolog geeignet. Das vorgestellte Auskunfts- und Reisebuchungssystem wird in Kapitel 14 mit einer Benutzungsschnittstelle und Verwaltungsprogrammen ergänzt. Als Alternative oder Ergänzung steht das ICE-Auskunftssystem aus Kapitel 13 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 zugrunde liegende Baumstruktur und verweist auf die Punktnotation für Listen. Sofern auf Strukturuntersuchungen wie in Kapitel 16 und darauf aufbauenden Programmiertechniken im weiteren Verlauf des Unterrichts nicht eingegangen werden soll, kann auf die Thematisierung der Punktnotation verzichtet werden. Die Unterkapitel 5.3 *Listenoperationen – aus imperativ wird deklarativ* und 5.4 *Kopf-Rest-Methode* sind zentrale Bausteine dieses Buches, 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.

Kapitel 7 beschäftigt sich mit Logikrätseln unterschiedlichster Art. Von einer Programmiersprache, die die Logik im Namen führt, kann man erwarten, dass logische Probleme leicht gelöst werden können. Dies wird auch gezeigt, wobei in diesem Kapitel besonders Wert auf einen deklarativen Ansatz gelegt wird.

In Kapitel 8 werden grundlegende Ein- und Ausgabeprädikate vorgestellt, welche für die programmgesteuerte Ein- und Ausgabe genutzt werden können. Zudem wird thematisiert, wie man System-Bibliotheken nutzen und mit Dateien arbeiten kann.

Der Cut ist Thema von Kapitel 9. 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. Für die Programmierpraxis wird der „->“Operator eingeführt, mit dem die If-Then-Else-Struktur bereitgestellt wird.

Kapitel 10 befasst sich ausführlich mit der Unifikation als einem wesentlichen Verfahren, auf dem ein Prolog-Interpreter 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 11 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.

In Kapitel 12 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 *findall*-Prädikat. *findall* wird intensiv in Kapitel 15 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 ein Regelsystem zur Empfehlung von Arzneimitteln ergänzt wird. Diese Regelsysteme können als vereinfachte Modelle für Expertensysteme betrachtet werden.

Ein ICE-Auskunftssystem ist Inhalt von Kapitel 13, Kapitel 14 bringt auf vergleichbarem Niveau ein Auskunfts- und Reisebuchungssystem. Zunächst wird eine datenbanktechnische Modellierung mit einem Entity-Relationship-Diagramm durchgeführt. Die Überführung in das Relationenmodell liefert Fakten, 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 12.2 über Informations- und Dateiverwaltungssysteme.

Kapitel 15 setzt sich mit typischen Suchverfahren auseinander. Sie werden am Beispiel eines bewerteten Graphen eingeführt und später auf Zustandsdiagrame angewendet. Dabei wird deutlich, wie Probleme mit klar definierten Problemzuständen und Zustandsübergängen sich in einheitlicher Weise durch die vorgestellten Suchverfahren lösen lassen.

Kapitel 5 brachte die Basis-Programmiertechniken in Prolog, welche für die Arbeit mit linearen Strukturen benötigt werden. Kapitel 16 befasst sich mit fortgeschrittenen Programmiertechniken für die Arbeit mit verzweigten Baumstrukturen. Vermutlich ist Ihnen von Pascal oder Java die recht unterschiedliche algorithmische Verarbeitung von Listen und Binärbäumen bekannt. Operationen auf Listen können mit iterativen Methoden implementiert werden, für Bäume benötigt man auch 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 *univ*-Operators stellt eine kleine Hürde dar, bringt allerdings wegen der damit vorhandenen Verfügbarkeit von Bäumen erhebliche Vorteile mit sich. Mit dem Verzicht auf den *univ*-Operator verzichtet man gleichfalls auf die Nutzung dynamischer Baumstrukturen und muss sich mit Listen begnügen. Die fortgeschrittenen Programmiertechniken dieses 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 17 beginnt der Teil, der sich hauptsächlich mit Aspekten der theoretischen Informatik befasst. Zunächst werden Grammati-

ken 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.

Das umfangreiche Kapitel 18 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.

In Kapitel 19 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.

Mit den Turingmaschinen, den leistungsfähigsten Automaten, setzt sich Kapitel 20 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).

Interessante Anwendungen der theoretischen Konzepte werden in den Kapiteln 21 und 22 gebracht. Schwerpunkt sind Parser und Interpreter für graphische Befehle der Programmiersprache LOGO, Interpreter und Übersetzer für mini-PASCAL und die maschinellen Sprachverarbeitung. Mit Hilfe des Simulationsprogrammes VISIS ist eine anschließende Vertiefung im Bereich technischer Informatik möglich.

Die Arbeit mit einer Konsole ist typisch für die Nutzung eines Interpreters. Sie entspricht aber nicht der Computernutzung mit gängigen grafischen Benutzungsschnittstellen. In Kapitel 23 werden deshalb XPCE und die Schnittstelle zu Prolog dargestellt. Anhand einfacher Beispiele wird dabei gezeigt, wie man mit dem vorhandenen Entwicklungswerkzeug grafische Benutzungsoberflächen für Prolog-Programme realisieren kann.

1.4 Klassifikation und Einsatzmöglichkeiten

Der Gesamtumfang macht es nötig, die Kapitel zu klassifizieren und Einsatzmöglichkeiten aufzuzeigen, um verschiedene unterrichtliche Intentionen verwirklichen zu können.

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 das Buch nicht als Sprachlehrbuch versteht, 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 SWI-Prolog verwiesen.

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 Sekundarstufe II gehalten werden, wobei die Schülerinnen und Schüler am Beispiel der Familienbeziehungen den Computer als logisch schließende Maschine kennen lernen.

Für einen Oberstufenkurs *Algorithmen und Datenstrukturen* geben die Kapitel 2 bis 16 reichlich Material. Schwerpunkte bilden die Standardalgorithmen auf Listen in Kapitel 5 und Termen in Kapitel 16, sowie die Suchverfahren in Kapitel 15.

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

Ab Kapitel 17 gibt es reichhaltig Material zu einem Kurs *Grundlagen der Theoretischen Informatik* mit den Schwerpunkten Automaten, Formale Sprachen und Interpreter. Der Umfang des gebotenen Materials macht eine Auswahl von Themen insbesondere bei den Automaten nötig.

Legende:
Anspruchsniveau:
1 = gering
2 = mittel
3 = hoch

Neue Sprachelemente:
N = Neu

Einsatzmöglichkeit:
S = Stammkapitel
E = Erweiterung

Nr	Kapitel	Anspruchsniveau	Neue Sprachelemente	Einführungskurs	Algorithmen und Datenstrukturen	Datenbanken	Grundlagen der theo. Informatik	Künstliche Intelligenz	Maschinelle Sprachverarbeitung
1	Prolog in der Schule	-							
2	Fakten, Regeln und Anfragen.	1	N	S	S	S	S	S	S
3	Ablaufverfolgung und Veranschaulichung	2		E	S	S	S	S	S
4	Datenbanken	1		S	E	S			
5	Listen	2	N	E	S	S	S	S	S
6	Arithmetik	1	N		S	S			
7	Logikrätsel	2		E	E			S	
8	Ein- und Ausgabe	2	N		E	S	E	S	S
9	Der Cut!	2	N		S		E	S	S
10	Unifikation	2	N		S		E	S	S
11	Symbolisches Differenzieren	2			S			S	
12	Wissensbasis und Regelsysteme	2	N		S	S		S	
13	ICE-Auskunftssystem	2				S			
14	Auskunfts- und Buchungsystem	2				S			
15	Suchverfahren	2			S			S	
16	Terme	3	N		S		E	S	
17	Grammatiken	2					S	S	S
18	Automaten	2					S		
19	Kellerautomaten	2					S		
20	Turingmaschinen	2					E		
21	Parser und Interpreter	2-3					S		
22	Maschinelle Sprachverarbeitung	2-3						S	S
23	GUI-Programmierung mit XPCE	2-3	N		E		E		

Tabelle 1-2 Klassifikation und Einsatzmöglichkeiten der Kapitel

Die Kapitel müssen nicht unbedingt in der Reihenfolge, wie sie im Buch bzw. 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 16.1 und 16.2 dargestellt werden, schon relativ früh behandeln und die viel anspruchsvolleren Unterkapitel 16.3 und 16.4 auslassen. Ähnliches gilt für Kapitel 12 über *Wissensbasis und Regelsysteme*. Hier kann man die Unterkapitel 12.1 *Hinzufügen und Löschen von Klauseln* und 12.3 *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 SWI-Prolog-Editor

SWI-Prolog ist ein freies und professionelles Prolog-System, das seit 1987 von Jan Wielemaker an der Universität von Amsterdam weiter entwickelt und gepflegt wird. Es eignet sich sehr gut für Unterrichtszwecke und kann kostenlos über www.swi-prolog.org in der aktuellsten Fassung bezogen werden.

Da SWI-Prolog derzeit über keine geeignete integrierte Entwicklungsumgebung (IDE) verfügt, stellt der Autor mit dem SWI-Prolog-Editor eine schülerorientierte und unterrichtsgeeignete Windows-Entwicklungsumgebung zur Arbeit mit Prolog zur Verfügung. Im Editor schreibt man ganz komfortabel seine eigenen Prolog-Programme, wobei man die unter Windows üblichen Datei-, Editor- und Fensteroperationen zur Verfügung hat. Zum Ausführen eines Programms übergibt der SWI-Prolog-Editor den Quelltext an SWI-Prolog, das in einem Fenster des SWI-Prolog-Editors läuft. In diesem Fenster können dann interaktiv die Prolog-Programme ausgeführt und getestet werden.

Eine detaillierte Beschreibung des SWI-Prolog-Editors, sowie Hinweise zur Installation und Konfiguration befinden sich auf dem Hessischen Bildungsserver unter http://lernen.bildung.hessen.de/informatik/swiprolog/index.htm.

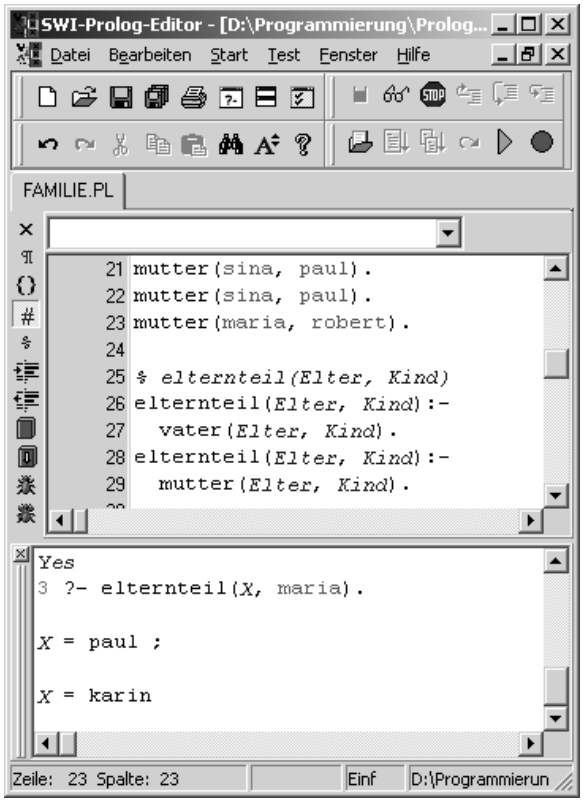
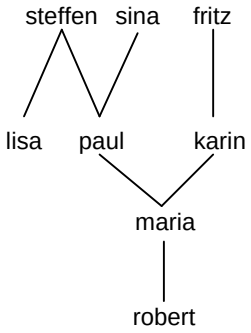


Abb. 2-1: SWI-Prolog-Editor

2.2 Fakten

Ein Prolog-Programm besteht aus Fakten, Regeln und Anfragen. Im Editor schreiben Sie Fakten und Regeln eines Prolog-Programms, das Sie dann durch Konsultieren in die Wissensbasis des Prolog-Interpreters laden. Mit Anfragen im SWI-Prolog-Fenster führen Sie das Programm aus.

Ein Faktum beschreibt eine Eigenschaft eines Objekts oder eine Beziehung zwischen mehreren Objekten. Der im Bild dargestellte Stammbaum lässt sich wie folgt durch Fakten repräsentieren.




```
% maennlich-Fakten
maennlich(paul).
maennlich(fritz).
maennlich(steffen).
maennlich(robert).

% weiblich-Fakten
weiblich(karin).
weiblich(lisa).
weiblich(maria).
weiblich(sina).

% vater(Vater, Kind)-Fakten
vater(steffen, paul).
vater(fritz, karin).
vater(steffen, lisa).
vater(paul, maria).

% mutter(Mutter, Kind)-Fakten
mutter(karin, maria).
mutter(sina, paul).
mutter(maria, robert).
```

Jedes Faktum beginnt mit einem Namen, dem sogenannten *Funktor*. Auf den Funktor folgen in Klammern die einzelnen *Argumente,* welche mit Kommas getrennt werden. Eine Faktendefinition schließt mit einem Punkt.

Da ein Bezeichner, der mit einem Großbuchstaben beginnt, in Prolog grundsätzlich eine *Variable* ist, darf der Funktor nicht mit einem Großbuchstaben beginnen.

Ein in einfache Anführungsstriche gesetzter Bezeichner wird nicht als Variable interpretiert.

So kann man auch die Vornamen mit einem Großbuchstaben schreiben:

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

Die fünfzehn aufgeführten Fakten gehören zu vier verschiedenen *Prädikaten*: maennlich, weiblich, vater und mutter. Jedes Prädikat wird durch seinen *Funktor* und seine *Stelligkeit* (Arität, Anzahl der Argumente) charakterisiert. Die Prädikate können daher auch in der prägnanten Kurzform *maennlich/1*, *weiblich/1*, *vater/2* bzw. *mutter/2* angegeben werden.

Die vier Prädikate unterscheiden sich in der Anzahl der *Klauseln* (Fakten und Regeln), welche für jeweils ein Prädikat definiert sind. Zu den Prädikaten *maennlich*, *weiblich* und *vater* gibt es je vier Klauseln, zum Prädikat *mutter* lediglich drei.

2.3 Regeln

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

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

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

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

grossmutter(Oma, Enkel):-
  mutter(Oma, Elter),
  elternteil(Elter, Enkel).

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

Jede Regel besteht aus einem *Regelkopf*, dem als umgekehrten Schlussfolgerungspfeil zu interpretierenden *Regeloperator* ':-' und einem *Regelrumpf*. Der Regeloperator kann als *wenn* gelesen werden. Er stellt die logische Schlussfolgerung (Konklusion) dar, welche sich aus dem Regelrumpf ergibt. Der Regelrumpf enthält die logische Und-Verknüpfung (Konjunktion) aller Bedingungen (Prämissen), aus welcher sich die Schlussfolgerung ergibt. Die Bedingungen sind Teilziele, die alle erfüllt sein müssen, damit der Regelkopf erfüllt ist.

Syntaktisch wird der Regelkopf wie eine sogenannte *Struktur* geschrieben, die aus einem Funktor und den in Klammern folgenden Argumenten besteht. Die logische Und-Verknüpfung der Teilziele wird durch die trennenden Kommas dargestellt.

Gibt es für eine Regel mehrere Alternativen, wie bei den Prädikaten *elternteil/2* und *vorfahr/2*, so verteilt man diese üblicherweise auf mehrere Klauseln. Stattdessen könnte man die logische Oder-Verknüpfung (Disjunktion) der Prämissen syntaktisch auch durch Semikola ausdrücken.

Die Argumente von Fakten sind in der Regel alles Konstanten. Im Unterschied werden in Regeln meist nur Variablen benutzt. Der Gültigkeitsbereich einer Variablen bezieht sich lediglich auf die Regel, in der sie auftritt. Der Variablenbezeichner *Kind* tritt in obigen Regeln neunmal auf. Tatsächlich werden damit entsprechend den vier verschiedenen Regeln wo Kind auftritt vier verschiedene Variablen bezeichnet.

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 *instanziert* werden.

2.4 Rekursive Prädikate

Interessiert man sich für die 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).
```

Es ist natürlich müßig, für jede weitere Generation von Vorfahren eine neue Regel anzugeben, zumal das Bildungsschema der Regeln klar ist. Dieses 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).
```

Die erste *vorfahr*-Klausel ist nicht-rekursiv und sorgt für die Terminierung der Rekursion, die

zweite *vorfahr*-Klausel enthält den rekursiven Aufruf.

2.5 Anfragen

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

```
?- maennlich(paul).                Yes.
?- weiblich(gerhard).              No.
?- maennlich(gerhard).             No.
?- mutter(karin, maria).           Yes.
?- elternteil(sina, paul).         Yes.
?- sohn(karin).                    No.
?- weiblich(maria), sohn(paul).    No.
```

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 zweite und dritte Anfrage zeigt, hat *no* keinesfalls die Bedeutung von nicht, sondern von *nicht mit der Wissenbasis beweisbar*.

Kommen in der Anfrage Variablen vor, so sucht Prolog Variablenwerte, die die Anfrage erfüllen. Mit einem Semikolon (logisches Oder) fordert man die nächste Lösung an.

```
?- maennlich(Mann).
Mann = paul;
Mann = fritz;
Mann = steffen;
Mann = robert;
No.

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

?- elternteil(Elter, maria).
Elter = paul;
Elter = karin;
No.

?- weiblich(Tochter), vater(V,Tochter).
Tochter = karin, V = fritz;
Tochter = maria, V = paul;
Tochter = lisa, V = steffen;
No.

?- weiblich(Tochter), vater(_,Tochter).
Tochter = karin;
Tochter = maria;
Tochter = lisa;
No.

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

In der vorletzten Anfrage kommt eine anonyme Variable vor, welche mit dem Unterstrich „_“ notiert wird. Mit anonymen Variablen kann man die Ausgabe unerwünschter Information in Anfragen unterdrücken, wie es der Vergleich mit der vorherigen Anfrage zeigt. Die letzte Anfrage macht deutlich, dass der winzige syntaktische Unterschied zwischen Komma und Semikolon einen großen semantischen Unterschied ausmacht: Logische Und-Verknüpfung (Konjunktion) mit Komma und logische Oder-Verknüpfung (Disjunktion) mit Semikolon.

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

ProVisor ist zwar keine Windows-Software verfügt aber über eine leicht zu bedienende Benutzungsoberfläche mit Dateischnittstelle, Editor, Fensterverwaltung und einem eigenen Prolog-Interpreter namens Zwerg-Prolog.

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.

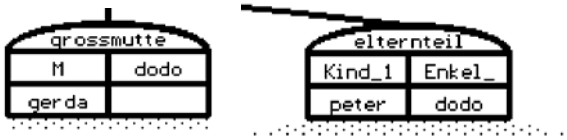


Abb. 2-2: Anfrageknoten und Teilzielknoten

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.

Lässt sich der Klauselkopf eines Lösungskandidaten mit dem Anfrage- oder Teilzielknoten durch geeignete Variablenbelegungen gleichmachen (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.

Das Prolog-Programm aus unseren Fakten, Regeln und der Anfrage *?- tochter(maria, Elter).* setzt ProVisor in Abb. 2-4 dargestellten Und-Oder-Beweisbaum um. 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 *weiblich(Kind)* und *elternteil(Elter, Kind)* besteht. Die Teilzeile werden als Unterknoten durch Strecken mit dem übergeordneten Und-Knoten verbunden.

Abb. 2-3: Oder-Knoten - Trapez

Abb. 2-4: Und-Knoten - Kanten

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

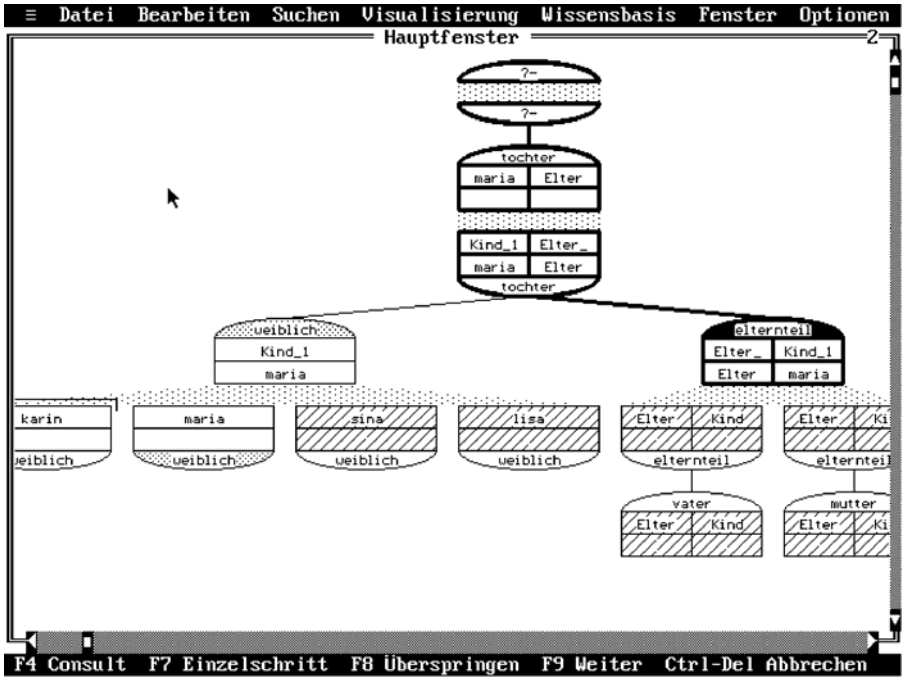


Abb. 2-5
Und-Oder-Beweisbaum

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.

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.

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.

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

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.

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. Oder-Verknüpfungen erhalten zusätzlich einen Verbindungsbogen. In dieser vereinfachten Symbolik sieht unser Beispiel wie folgt aus:

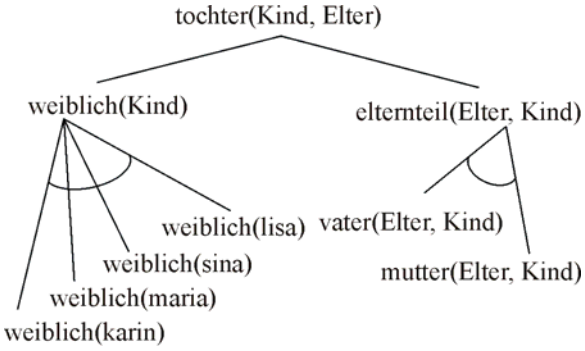


Abb. 2-6: vereinfachter Und-Oder-Beweisbaum

2.7 Maschinelles logisches Schließen

Lässt man mit ProVisor den Prolog-Interpreter schrittweise arbeiten, so wird sukzessive nach Maßgabe des zugrunde liegenden Resolutionsverfahrens der Beweisbaum aufgebaut. Aus der Wissenbasis 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 *instanziert* d. h. belegt werden.

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.

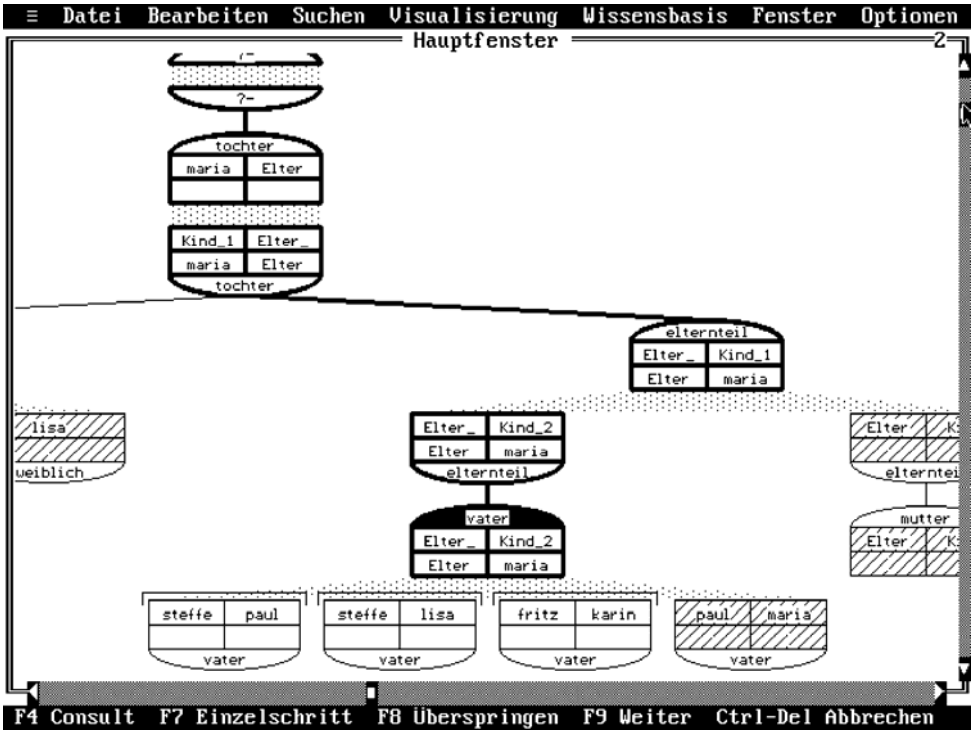
Abbildung 2-6 zeigt die entscheidende Stelle zur Lösung der Anfrage

```
?- tochter(maria, Elter).
```

Zum aktuellen Knoten *vater(Elter, maria)* wird im nächsten Schritt das passende Faktum *vater(paul, maria)* gefunden, wobei *Elter* mit *paul* instanziert wird. Bei der nachfolgenden Suche nach offenen Teilzielen von Und-Knoten erreicht der Prolog-Interpreter schließlich den Anfrageknoten, womit sichergestellt ist, dass keine weiteren Teilziele erfüllt werden müssen. Damit ist eine Lösung gefunden und die instanzierten Anfragevariablen werden als Lösung ausgegeben.

Lösung: `Elter = paul.`

Abb. 2-7: Anfrage tochter(maria,Elter).



Sollen weitere Lösungen gefunden werden, so setzt Backtracking ein. Die bisherige Lösung wird verworfen und es geht im Beweisbaum zum letzten Alternativpunkt zurück. Von dort wird die Suche nach der Methode *Ausprobieren aller Alternativen mit eventuellem Rücksetzen* fortgesetzt.

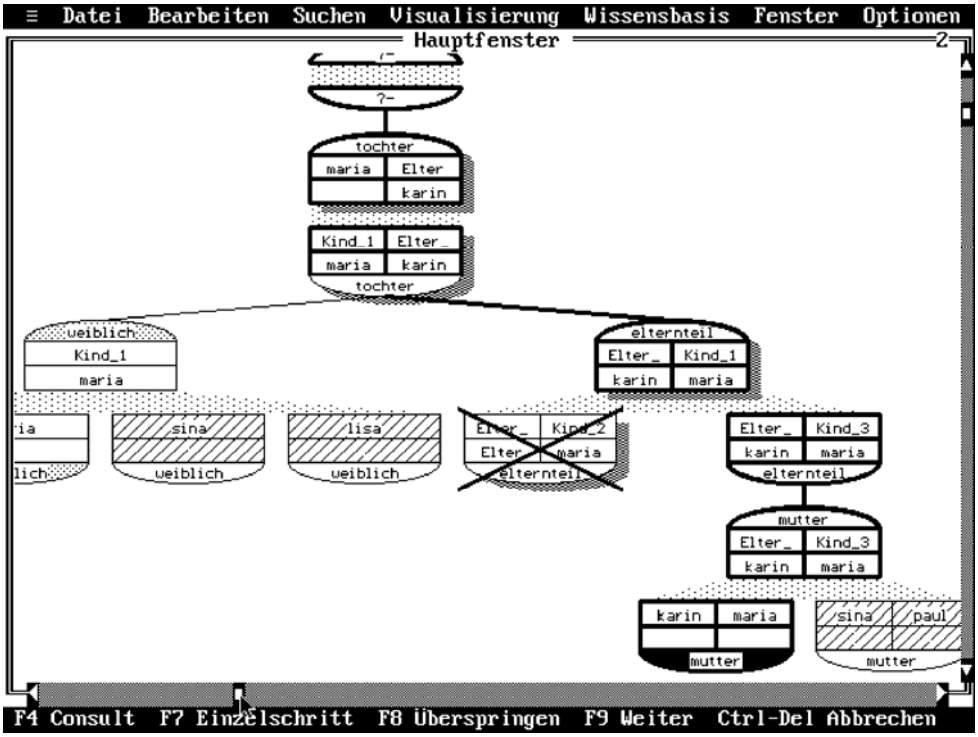


Abb. 2-8: Backtracking im Und-Oder-Beweisbaum

Sind Teilziele nicht weiter erfüllbar, so werden sie durchgestrichen. Knoten, die ein weiteres Mal erfüllt werden, über die also Backtracking stattfindet, werden schattiert dargestellt.

In Abbildung 2-7 wurde gerade die zweite Lösung der Anfrage *?- tochter(maria, Elter)* gefunden.

Die Mechanismen, nach denen wissensbasierte Systeme Schlussfolgerungen durchführen, beruhen also auf *Resolution, Unifikation* und *Backtracking*. Für alle drei Mechanismen lassen sich einfache Algorithmen angeben. Damit sind die Denkfähigkeiten eines Prolog-Interpreters entmystifiziert und auf algorithmische Grundmuster zurückgeführt. Freilich ist die Frage, ob Maschinen denken können, damit nicht beantwortet.

2.8 Aufgaben

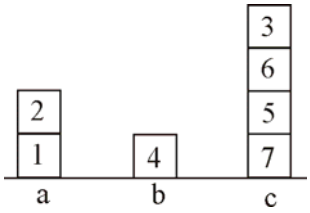
1a) Starten Sie den SWI-Prolog-Editor, legen Sie eine neue Datei an und geben Sie im Editor die Familienbeziehungen als Fakten und Regeln ein.
b) Überprüfen Sie die Anfragen aus Kapitel 2.3.
c) Formulieren Sie Anfragen zu: Eltern von Paul, Bruder von Lisa, Kinder von Karin, Großmutter von Maria, Großvater von Maria, alle Großmütter, alle Väter, alle Eltern-Kind-Beziehungen, sind Lisa und Paul Geschwister, Vorfahren von Maria.

2. 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).`

3. Verfolgen Sie die Arbeitsweise des Prolog-Interpreters mit ProVisor für Beispiele aus Aufgabe 1c.

4. An einem runden Tisch sitzen symmetrisch sechs Personen:
Alfred, Anton, Antonia, Annemarie, Anna und August.
a) Erstellen Sie eine Wissenbasis 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)`

5. Auf den Plätzen a, b und c liegen die Blöcke 1, 2, 3, 4, 5, 6 und 7 in der gezeichneten Anordnung.



a) Stellen Sie die Plätze mit Hilfe von Fakten dar.
b) Stellen Sie die Türme mit Hilfe von auf/2-Fakten dar: auf(1, a), auf(2, 1),…
c) 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.

6. 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

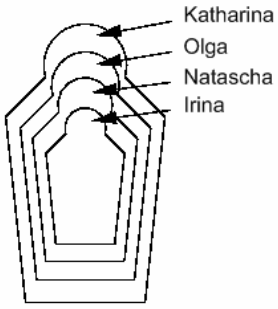
7. Fahrzeugtypen sind durch bestimmte Merkmale charakterisiert, so z. B. Pkw dadurch, dass sie vier Räder haben, motorbetrieben sind, zum Personentransport dienen usw., während Fahrräder zwei Räder haben, einen Pedalantrieb haben, zum Personentransport dienen usw. Entwerfen Sie eine Datenbank, die man befragen kann, welche Merkmale ein bestimmter Fahrzeugtyp, z.B. ein Fahrrad, hat, aber auch, welcher Fahrzeugtyp bestimmte Merkmale hat, z.B. vier Räder zu haben, motorbetrieben zu sein u. a.

8. Entwerfen Sie eine Datenbasis mit folgenden Informationen:
a) Apfelbaum, Stachelbeer- und Himbeerstrauch, Kaktee und Hundsrose sind Pflanzen.
b) Apfel ist die Frucht eines Apfelbaums, Stachelbeere die eines Stachelbeerstrauchs, Himbeere die eines Himbeerstrauchs, Feige die einer Kaktee, Hagebutte die einer Hundsrose.
c) Apfel, Stachelbeeren, Himbeeren und Feigen sind essbar.
d) Etwas kann als Obst bezeichnet werden, wenn es eine Pflanze gibt, dessen Frucht es ist, und wenn es essbar ist.
Stellen Sie Fragen wie:
e) Was ist die Frucht der Kaktee?
f) Welche Früchte sind essbar?
g) Die Früchte welcher Pflanzen werden als Obst bezeichnet?

9. Modellieren Sie die folgenden Aussagen in Prolog und stellen Sie dann geeignete Anfragen.

   Aristoteles ist ein Mensch.
   Sokrates ist ein Mensch.
   Die Schlange ist ein Tier.
   Zeus ist ein Gott.
   Apollo ist ein Gott.
   Alle Menschen sind sterblich.
   Alle Tiere sind sterblich.
   Götter sind unsterblich.

10. Matroschkas sind aus Holz gefertigte ineinander schachtelbare russische Puppen.



a) Modelliere die Matroschkas durch geeignete Fakten.
b) Implementieren Sie ein Prädikat enthaelt/2, mit dem geprüft werden kann, ob eine Matroschka eine andere enthält.

11. In der Wissenbasis stehen die folgenden Direktverbindungen:

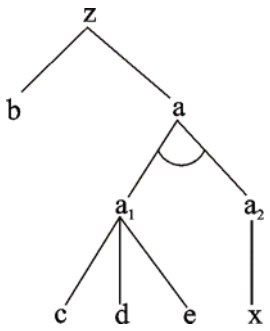
```
direkterZug(forbach, saarbruecken).
direkterZug(freyming, forbach).
direkterZug(fahlquemont, stAvold).
direkterZug(stAvold, forbach).
direkterZug(saarbruecken, dudweiler).
direkterZug(metz, fahlquemont).
direkterZug(nancy, metz).
```

a) Implementieren Sie ein Prädikat zugverbindung(Von, Nach), dass mögliche Zugverbindungen ermittelt.
b) Wenn ein direkter Zug von A nach B fährt, so gibt es auch einen direkten Zug von B nach A.

12. 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.
```

13. 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).
```

Die schrittweise Abarbeitung eines Prolog-Programms wird auch in üblichen Prolog-Interpretern unterstützt, allerdings in rein textlicher Form.

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

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).
```

Die Anfrage *?- satz(der, Nomen, schiesst).* 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.

```
?- satz(der, Nomen, schiesst).
CALL: satz(der, Nomen, schiesst)
CALL: nominalphrase(der, Nomen)
CALL: artikel(der)
EXIT: artikel(der)
CALL: nomen(Nomen)
EXIT: nomen(hund)
EXIT: nominalphrase(der, hund)
CALL: verbalphrase(schiesst)
CALL: verb(schiesst)
EXIT: verb(schiesst)
EXIT: verbalphrase(schiesst)
EXIT: satz(der, hund, schiesst)
satz(der, Nomen, schiesst)
Nomen = hund
yes
REDO: satz(der, hund, schiesst)
REDO: verbalphrase(schiesst)
REDO: verb(schiesst)
FAIL: verb(schiesst)
FAIL: verbalphrase(schiesst)
REDO: nominalphrase(der, hund)
REDO: nomen(hund)
EXIT: nomen(jaeger)
EXIT: nominalphrase(der, jaeger)
CALL: verbalphrase(schiesst)
CALL: verb(schiesst)
EXIT: verb(schiesst)
EXIT: verbalphrase(schiesst)
EXIT: satz(der, jaeger, schiesst)
satz(der, Nomen, schiesst)
Nomen = jaeger
```

SWI-Prolog zeigt beim Backtracking nicht alle REDO-Ports explizit an.

3.2 Das Vierport- oder Boxen-Modell

Die Ausgaben des Trace-Protokolls lassen sich mit dem Vierport-Modell verstehen und veranschaulichen, das jedem Prädikat ein Rechteck (eine Box) mit zwei Eingängen und zwei Ausgängen zuordnet:

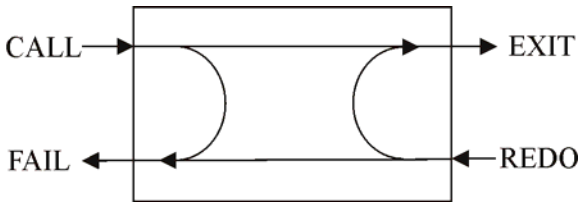


Abb. 3-1 Grundstruktur des Vierport-Modells

Über den CALL-Eingang wird ein Teilziel aufgerufen. Im Erfolgsfall wird das Rechteck über den EXIT-Ausgang verlassen, falls das Ziel scheitert, geht es über den FAIL-Ausgang heraus. Alternative Lösungen können über den Backtracking-Eingang REDO gesucht werden.

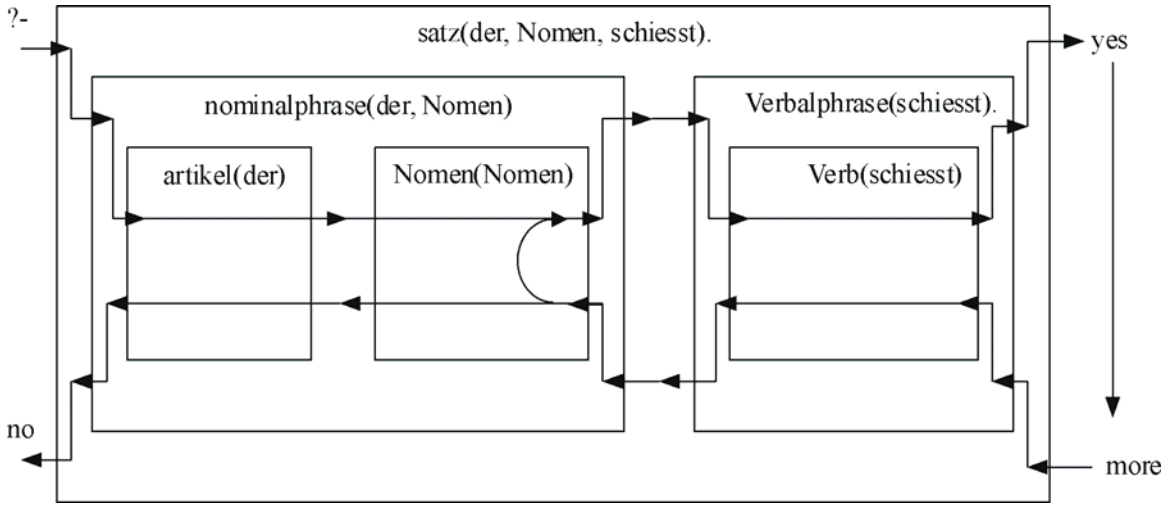


Abb. 3-2 die Anfrage *?-satz(der, Nomen, schiesst)* im Vierport-Modell

Und-Verknüpfungen (Konjunktionen) von Teilzielen werden durch eine Reihenschaltung *Rechtecke* dargestellt, die Verwendung von Prädikaten durch *ineinandergeschachtelte Rechtecke*. Das obige Trace-Protokoll lässt sich im Boxen-Modell so erklären, dass immer dann, wenn ein Rechteck während des Trace betreten oder verlassen wird, der entsprechende Port samt Rechteck und Variablenbelegung angegeben wird.

3.3 SWI-Prolog-Debugger

SWI-Prolog bietet einen grafischen Debugger an, mit dem der Ablauf eines Programms visuell sehr gut verfolgt werden kann. Dieser Debugger wird automatisch gestartet, wenn man eine Anfrage traced und im Test-Menü des SWI-Prolog-Editors die Option *GUI-Tracer* eingeschaltet ist.

Im Bereich *Bindings* werden die aktuellen Variablenbindungen angezeigt. Bei komplexen Strukturen können Sie sich Details durch Doppelklick anzeigen.

Im Bereich *Call Stack* wird der aktuelle Aufrufstack dargestellt, bei dem das zu untersuchende Hauptziel ganz oben steht und neue Teilziele am Fuß eingefügt werden. Rechts vom Aufrufstack werden Teilziele dargestellt, in denen noch alternative Lösungen möglich sind. Im Bild wurde das Nomen Y = hund gefunden, womit die Nominalphrase erfüllt ist. Dass es noch weitere Lösungen für nomen(Nomen) gibt, wird durch den Verweis auf nomen/1 mit dem vorangestellten Verzweigungspfeil dargestellt.

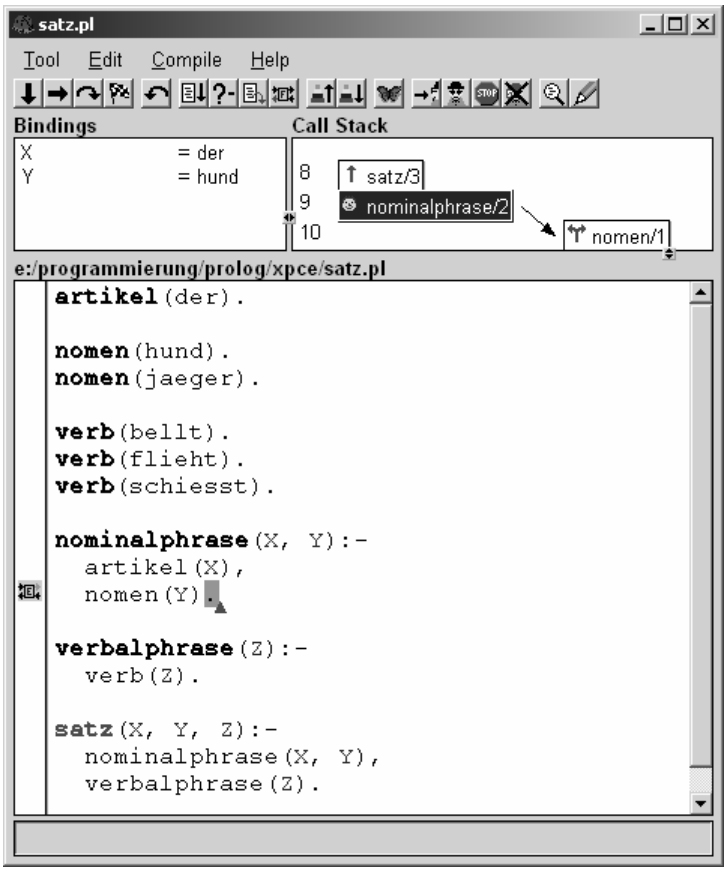


Abb. 3-1 Grafischer Debugger

Bei der schrittweisen Abarbeitung wird im Quelltextbereich die jeweils ausgeführte Zeile farblich hervorgehoben, wobei zwischen CALL und EXIT deutlich unterschieden wird. Klickt man auf ein Element des Aufrufstacks, wird die betreffende Quelltextzeile synchronisiert.

3.4 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 Protokollen die Information zu finden, für die man sich eigentlich interessiert. Das Trace-Protokoll wird zur Trace-Katastrophe.

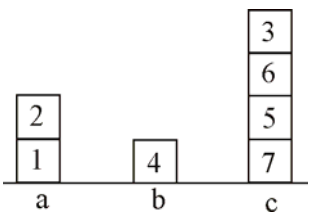
Der Autor hat daher der *Spur*-Konzeption aus [Ste1] folgend mit einer in [Fuc1] beschriebenen Methode einen Metainterpreter programmiert, welcher in Form von Spuren, die Lösungssuche eines Prolog-Interpreters darstellt. Der Metainterpreter liegt in Form des spur/1-Prädikats vor. Die Datei spur.pl speichert man in das SWI-Unterverzeichnis *library* und ruft dann im SWI-Prolog-Fenster *make.* auf. Damit steht spur/1 wie ein Systemprädikat zur Verfügung.

Die Benutzerschnittstelle des Metainterpreters ist das *spur*-Prädikat. Die zu untersuchende Anfrage geben Sie dem *spur*-Prädikat als Argument mit.

```
?- spur(satz(der, Nomen, schiesst)).
satz(der, _G158, schiesst)
   nominalphrase(der, _G158)
      artikel(der)
      nomen(hund)
   verbalphrase(schiesst)
      verb(schiesst)
Nomen = hund ;
      nomen(jaeger)
   verbalphrase(schiesst)
      verb(schiesst)
Nomen = jaeger ;
```

Im obigen Beispiel sieht man, dass der Metainterpreter spur/1 nur die CALL-Ports anzeigt und entsprechend der Aufruftiefe die zu lösenden Teilziele einrückt. Dies ist insbesondere bei Rekursion sehr hilfreich, da sich Rekursionstiefen an den zugehörigen Einrücktiefen ablesen lassen.

Als weiteres Beispiel schauen wir uns die Spur des ueber-Prädikat aus dem Klötzchen-Beispiel an.




```
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
```

3.5 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 Und-Verknüpfung von Teilzielen wird im Boxen-Modell durch Reihenschaltung von Boxen dargestellt.

a) Wie kann die Disjunktion von Teilzielen dargestellt werden?

b) Zeichnen Sie für die Anfrage `?- a.` bei diesen beiden Regeln

   ```
   a:- b, c.
   a:- d, e, f.
   ```

   einen Vierport-Graphen.

5. Schalten Sie im SWI-Prolog-Editor über das Test-Menü den GUI-Tracer ein. Tracen Sie damit eine Anfrage.

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(HCode, Name, Sterne, Gebiet).
```

Die nachstehenden Beispiele aus dem Hotelbestand zeigen, dass Namen auch mit Großbuchstaben geschrieben werden können, ohne gleich als Variable missverstanden zu werden. Man setzt dazu Textkonstanten 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(gsgh03n, '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').
```

Auf diesem Datenbestand sind die beiden folgenden relationalen Grundoperationen möglich:

Selektion - Auswahl bestimmter Datensätze

Beispiel: Hotels auf Djerba

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

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(HCode, Saison, Wochen, Preis).
```

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

```
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 - Verknüpfung zweier Tabellen

Beispiel: Hotelpreise

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

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

```
hotelpreise(HCode, 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, Abflug-Flughafen, Mo-
nat, Saison-Buchstabe).
```

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).
```

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,- € 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(KNr, Name, StrasseNr, PLZ, Ort).
```

Beispiele:

```
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(BNr, HCode, 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 nachfolgenden Aufgaben zeigen, welche Probleme mit diesem Buchungssystem bearbeitet werden können. In Kapitel 14 wird das Auskunfts- und Reisebuchungssystems 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 dem 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. *Skrpycak?*

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 mit Prolog ein
a) Bibliothekssystem
b) Video-Center
c) Fahrplan-Auskunftssystem
d) einen Cocktail-Berater.

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 (künstliche Intelligenz) unterschiedliche Datentypen aufweisen.

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

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

Damit Sie sich von Prolog-Termen ein Bild machen können, speichern Sie die Systemerweiterung *zeichneterm.pl* in das SWI-Unterverzeichnis *library* und rufen dann im SWI-Prolog-Fenster *make.* auf. Damit stehen Ihnen die beiden Prädikate *zeichne_term/0* und *zeichne_term/1* zur Verfügung. Beim ersten Aufruf von *zeichne_ term/0* wird das XPCE-Grafiksystem initialisiert und dann das Fenster Termdarstellung aus Abbildung 5-1 aufgebaut. Gibt man zeichne_term/1 beim Aufruf einen Term mit, so wird dieser im Fenster gleich dargestellt.

Die obige 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. Die inneren Knoten sind mit einem Punkt "." bezeichnet, da der Punkt der Listenfunktor ist.

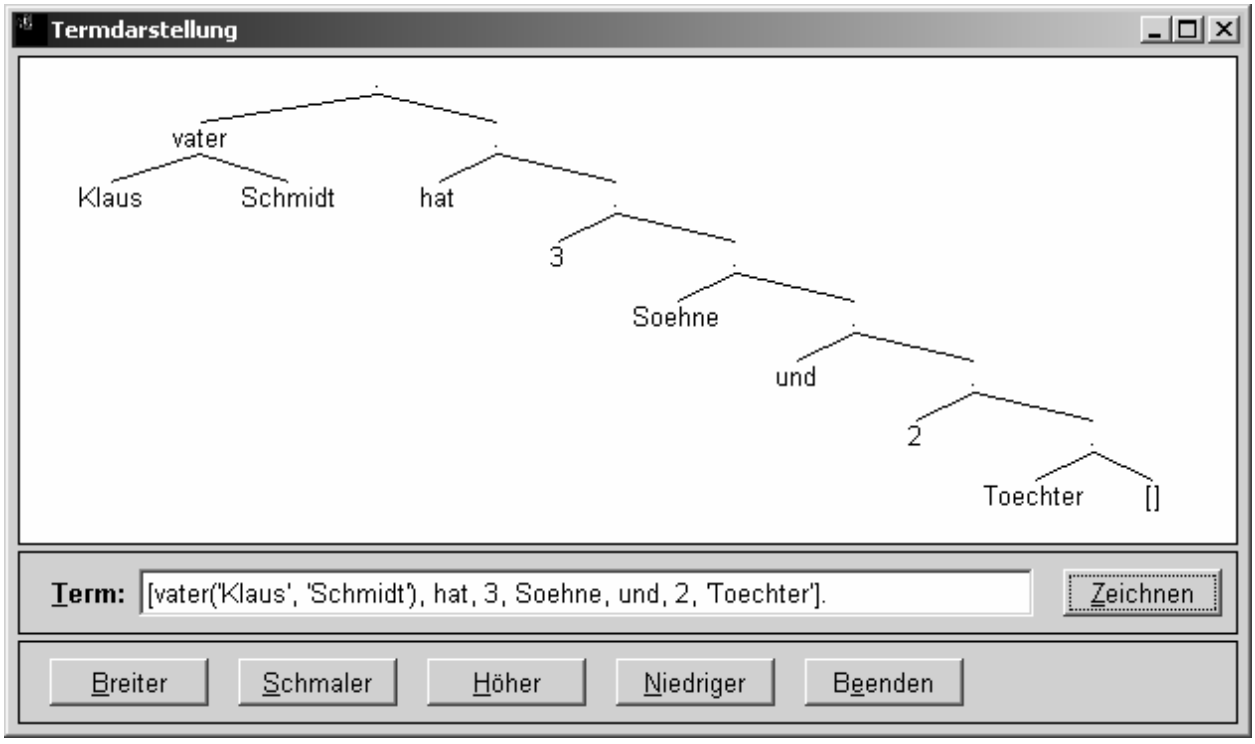


Abb. 5-1 Eine Liste mit sieben Elementen

Die Liste ist eine grundlegende dynamische Datenstruktur. Im Gegensatz zur statischen Datenstruktur Feld ändert sich während des Programmlaufs meist die Anzahl Elemente in einer Liste. In imperativen Programmiersprachen nutzt man den Heap, um zur Laufzeit Speicher für weitere Listenelemente zu erhalten. Mit Zeigervariablen verwaltet man die Adressen, die zum Listenaufbau nötig sind. Eine typische Datenstruktur für eine lineare Liste sieht in Pascal so aus:

```
type ListePtrTyp = ^ListeRecTyp;
     ListeRecTyp = record
       Element: ElementTyp;
```

```
    ToNext : ListePtrTyp;
  end;
```

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.

Mit der Listenschreibweise [Element1, Element2,…, ElementN] lässt sich die Dynamik nicht beschreiben, nur Listen fester Länge können wir so notieren. Prolog bietet daher den Listen-Operator „|“ an. Mit ihm kann man aus einem Kopfelement und einer Restliste eine neue Liste zusammensetzen oder eine bestehende Liste in Kopfelement und Restliste aufteilen:

Eine Liste: [Element | Restliste]

Grafisch dargestellt sieht das wie folgt aus:

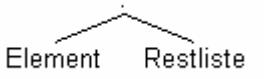


Abb. 5-2 Visualisierung der Listenstruktur [Element|Restliste]

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 daher mit dem Listenoperator eine Möglichkeit, Listen beliebiger Länge zu beschreiben.

Achten Sie darauf, dass der Listen-Operator „|“ unterschiedliche Objekte verbindet. Links vom „|“ steht ein einzelnes Listenelement, rechts davon eine komplette Liste. Typischerweise verbinden binäre Operatoren Operanden gleichen Typs. Da dies beim Listenoperator anders ist, entstehen beim Programmieren schnell Fehler.

Beispiele zur Verwendung des Listen-Operators:

- [a | [b, c]]

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

- [gelb | []]

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

- ?- Y = [b, c, d], X = [a | Y]

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

- ?- [X | 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]] = [Element | Restliste]

  liefert Element=[a, b] und 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.

- ?- NeuerStack =
  [name('Klaus', 'Schmidt') | Stack]

  setzt das strukturierte Element *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.

- ?- [_ | NeuerStack] = Stack

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

5.2 Punktschreibweise für Listen

Die obigen Beispiele zeigen, dass man in Prolog weitaus einfacher mit Listen arbeiten kann, als dies in Pascal oder Java der Fall ist. Dies liegt unter anderem daran, dass eine viel einfachere Notation zur Verfügung steht und keine Zeiger verwendet werden müssen. 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 Listenoperator „|“.

Man kann den Listenoperator auch in erweiterter Form benutzen, in dem man nicht nur ein, sondern gleich mehrere Elemente als Kopf der Liste angibt. Also zum Beispiel

[Ele1, Ele2, Ele3 | Restliste]

Diese Liste besteht aus den drei Elementen Ele1, Ele2 und Ele3, auf die die Restliste folgt.

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

Beispiele:

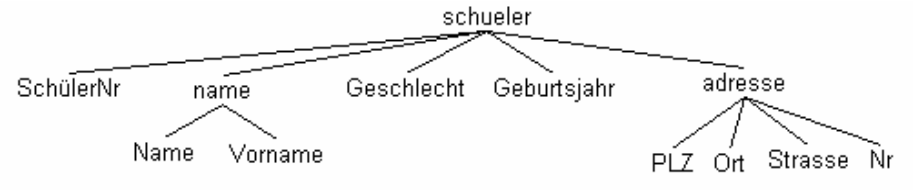


Abb. 5-3 Struktur als Baum

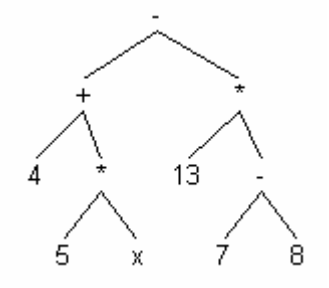


Abb. 5-4 Rechenterm 4+5*x-13*(7-8) als Baum

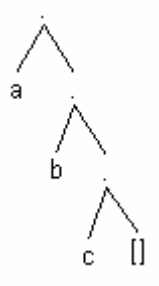


Abb. 5-5 Liste [a, b, c] als Baum

Wie man sieht, befinden sich die eigentlichen Daten in den Blättern der Bäume. Die inneren Knoten bilden das Gerüst für den Aufbau der jeweiligen Datenstruktur. Das schueler-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:

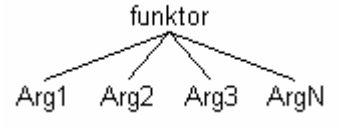


Abb. 5-6 Funktor und Argumente in der Baumdarstellung

Der Term aus Abbildung Abb. 5-6 schreibt sich in Präfix-Schreibweise als

```
funktor(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 *kunde* und *buchung*, schreibt man üblicherweise in Präfix-Notation. Für zweistellige Strukturen ist die Umwandlung von der Präfix- in die Infix-Schreibweise und umgekehrt möglich.

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
```

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, [])                Liste [a]
.(a, .(b, []))          Liste [a, b]
.(a, .(b, .(c, [])))    Liste [a, b, c]
```

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))`

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 – aus imperativ wird deklarativ

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 oder Java 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 Systemprädikate 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.

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
```

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)).
```

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).
```

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).
```

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).
```

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).
```

Im Prolog-Stil ist die imperative Sicht verschwunden. Man sieht nicht mehr, wie das Ergebnis berechnet wird. Stattdessen greift jetzt

die deklarative Sicht. Aus dieser neuen Sicht muss zum Abschluss das Ergebnis interpretiert werden. Die erste Klausel sagt aus, dass das Element E ein Element der Liste [E|_] mit dem Kopfelement E ist. Mit anderen Worten, E ist Element jeder Liste, die E als Kopfelement enthält. Die zweite Klausel berücksichtigt den Fall, dass E nicht das Kopfelement ist. Dabei gilt die Aussage, dass E Element einer Liste [_|Y] ist, wenn E Element der Restliste Y ist. Dieses Beispiel zeigt schön die deklarative und logische Sicht von Prolog-Klauseln.

Die gefundene Lösung leistet mehr, als ursprünglich beabsichtigt war. Vorgesehen 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:

`member(+E, +L).`	Prüft, ob E in L vorkommt.
`member(-E, +L).`	Liefert ein E aus L.
`member(+E, -L).`	Liefert variables L mit E.
`member(-E, -L).`	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).
```

Da member ein nicht überschreibbares Systemprädikat ist, können Sie die entwickelte Implementierung nur unter einem anderen Namen, z. B. *mitglied* ausprobieren.

Lange Listen gibt SWI-Prolog gekürzt aus. Um die komplette Liste zu sehen, gibt man „w" ein.

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 diese Genese aus der imperativen Sicht: die Liste L2 soll an die Liste L1 angehängt werden, um die Resultatliste L3 zu erhalten. Der Pascal/Java-Programmierer sieht somit eine Prozedur mit zwei Wert- und einem Variablenparameter und macht daher den Ansatz:

```
% append(+L1, +L2, -L3)     Prozedurkopf
```

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 imperative 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].
```

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).
```

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

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

Die deklarative Interpretation sieht wie folgt aus. Hängt man an die leere Liste eine Liste L2 an, so erhält man die Liste L2. Hängt man hingegen an die nicht leere Liste [X|R] eine Liste L2 an, so hängt man zunächst an die Restliste R die Liste L2 an. Setzt man vor die Ergebnisliste R3 das Kopfelement X der ersten Liste, so ist [X|R3] die aus beiden Ausgangslisten zusammen gesetzte Liste.

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

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

Da append ein nicht überschreibbares Systemprädikat ist, können Sie die entwickelte Implementierung nur unter einem anderen Namen, z. B. *anhaengen* ausprobieren.

5.4 Kopf-Rest-Methode

Für die algorithmische Bearbeitung von Binärbäumen gibt es einige bekannte Grundmuster. Die drei wichtigsten von insgesamt sechs möglichen sind:

WLR – Wurzel-Links-Rechts (Preorder)
LWR – Links-Wurzel-Rechts (Inorder)
LRW – Links-Rechts-Wurzel (Postorder).

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.

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.

Beispiel 1: Länge einer Liste

Mit dem zweistelligen Prädikat *laenge(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 *laenge(Rest, N1).*

```
laenge([], 0).
laenge(L, N):-
  L = [Kopf | Rest],
  laenge(Rest, N1),
  N is 1 + N1.          (siehe 6.1)
```

Die Spur zeigt sehr schön, wie die Länge einer Liste berechnet wird:

```
?- spur(laenge([a,b,c], N)).
laenge([a, b, c], _G170)
   [a, b, c]=[a, b, c]
   laenge([b, c], _G251)
      [b, c]=[b, c]
      laenge([c], _G282)
         [c]=[c]
         laenge([], 0)
         1 is 1+0
      2 is 1+1
   3 is 1+2
N = 3
```

Beispiel 2: Liste umkehren

Mit dem Prädikate *umdrehen(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]

```
umdrehen(Liste1, Liste2):-
  Liste1 = [Kopf|Rest1],
  umdrehen(Rest1, Rest2),
  append(Rest2, [Kopf], Liste2).
umdrehen([], []).
```

Wegen der besonderen Rolle der *Kopf-Rest-Methode* bei Problemlösungen in Prolog werden im anschließenden Aufgabenteil dazu sehr viele passende Übungsaufgaben angeboten. Sie dienen dem Ziel, die Kopf-Rest-Methode einzuschleifen.

5.5 Akkumulatortechnik

Die Elemente einer Liste kann man wie folgt aufsummieren:

```
summiere(Liste, Summe):-
  Liste = [Kopf | Rest],
  summiere(Rest, SummeR),
  Summe is Kopf + SummeR.
summiere([], 0),
```

Bei diesem Ansatz muss man also erst zum Ende der Liste gehen, um mit dem Addieren anfangen zu können. Mit der *Akkumulatortechnik* kann man vom Anfang her mit dem Aufsummieren beginnen. Dazu braucht man als Akkumulator ein weiteres Argument, in dem die bisherige Teilsumme übergeben werden kann. Sie muss mit 0 initialisiert werden.

```
summiere(Liste, Summe):-
  summiere(Liste, 0, Summe).
```

Jetzt kann die Summation vom Listenanfang her erfolgen:

```
summiere(Liste, Akku, Summe):-
  Liste = [Kopf | Rest],
  AkkuN is Akku + Kopf,
  summiere(Rest, AkkuN, Summe).
```

Beim Erreichen des Listenendes steht das Ergebnis fest:

```
summiere([], Akku, Summe):-
  Summe = Akku.
```

Die Akkumulatortechnik ist manchmal viel effizienter, wie wir uns das am Beispiel des Umdrehens einer Liste verdeutlichen können. Wie beim Summieren gehen wir in drei Schritten vor.

1. Einführen eines weiteren Arguments zur Aufnahme des Zwischenergebnisses:

```
umdrehen(Liste1, Liste2):-
  umdrehen(Liste, [], Liste2).
```

2. Umdrehen für das Kopfelement:

```
umdrehen(Liste1, Akku, Liste2):-
  Liste1 = [Kopf | Rest],
  AkkuN = [Kopf | Akku],
  umdrehen(Rest, AkkuN, Liste2).
```

3. Beim Erreichen des Listenendes das Ergebnis festsetzen:

```
umdrehen([], Akku, Liste2):-
  Liste2 = Akku.
```

Nach der Vereinfachung erhält man:

```
umdrehen(Liste1, Liste2):-
  umdrehen(Liste1, [], Liste2).
umdrehen([Kopf|Rest], Akku, Liste2):-
  umdrehen(Rest, [Kopf|Akku], Liste2).
umdrehen([], Liste2, Liste2).
```

Im Vergleich zur Lösung aus Beispiel 2 ist diese Implementierung deutlich besser, weil nicht bei jedem Listenelement *append* aufgerufen werden muss.

Durch Einsatz der Akkumulatortechnik wurde in beiden Beispielen die Rekursion in die besondere Form der *Endrekursion* gebracht werden, bei der also im Regelrumpf der rekursive Aufruf am Ende steht. Der Vorteil der Endrekursion ist, dass zusätzlicher Speicher zur Verwaltung der Rekursion entfällt.

5.6 Aufgaben

1. Welche Antworten liefern folgende Anfragen?

a) `[X | Y] = [rhein,elbe,weser,mosel].`
b) `[X | [weser, mosel]] = [elbe, weser, mosel].`
c) `[X | [weser, mosel]] = [rhein, elbe, weser, 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, geben Sie gegebenenfalls die Liste in Klammernotation an und lassen Sie sich die Struktur mittels *zeichne_term* anzeigen.

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 *writeln*-Prädikat zum Schreiben.

6. Das Prädikat *gleich(L1, L2)* 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. Das Prädikat *sortiert(L)* soll feststellen, ob eine Liste L aufsteigend sortiert ist.
   ```
   ?- sortiert([3, 6, 9, 15]).
   ```
   Antwort: `Yes.`

10. Definieren Sie ein Prädikat *gezaehlt*, mit gezählt wird, wie oft ein Element in einer Liste vorkommt.
    ```
    ?- 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).
    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. Gesucht ist ein Prädikat *liste2menge*, das aus einer Liste alle Elemente entfernt, die mehrfach vorkommen
    ```
    ?- liste2menge([2,3,2,4,3,4,3], X).
    ```
    Antwort: `X = [2, 3, 4].`

20. Implementieren Sie ein Prüfprädikat für Palindrome, also für Listen, die von vorn und hinten gelesen gleich sind:
    ```
    ?- palindrom([r,e,n,t,n,e,r]).
    ```
    Antwort: `Yes.`

21. Ein Prädikat *permutation* ist gesucht, das die Permutationen einer Liste erzeugen kann.
    Ein Ansatz besteht darin, dass man ein Element aus der Liste auswählt, welches das erste Element der permutierten Liste sein soll, und dann das rekursive Permutieren des Rests der Liste.

a) Implementieren Sie das Hilfsprädikat auswaehlen(-X, +L1, -L2).

b) Implementieren Sie damit permutation(+L1, -L2).

22. Vektoren kann man als Listen darstellen.

a) Entwickeln Sie ein Prädikat mit dem ein Vektor mit einer Zahl multipliziert werden kann.

b) Entwickeln Sie ein Prädikat, mit dem das Skalarprodukt zweier Vektoren berechnet werden kann.

23. 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.

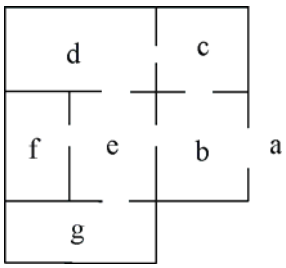
    Beispiel:
    ```
    ?- plaetten([1, 2, [3,4], [],
                 [[5, 6], 7, []]], L2)
    liefert L2 = [1, 2, 3, 4, 5, 6, 7]
    ```

    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.

24. 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, nows,
                    prolog, well], L).
    ```
    liefert
    ```
    L = [der, frau, kennt, prolog, gut]
    ```

25. 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 Merkliste 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. SWI-Prolog stellt aber außer Integer-Arithmetik auch Real-Arithmetik und die üblichen mathematischen Funktionen, wie zum Beispiel *sqrt, log* und *sin*, zur Verfügung.

Die Grundrechenarten schreibt man in gewohnter Weise:

`X + Y`	Addition
`X - Y`	Subtraktion
`X * Y`	Multiplikation
`X / Y`	Real-Divison
`X // Y`	Integer-Division (div)
`X mod Y`	Rest der Integer-Divison

Bei den Vergleichsoperatoren müssen Sie auf die unüblichen Schreibweisen achten:

`X =:= Y`	numerisch gleich
`X =\= 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
`X >= Y`	X ist größer als oder gleich Y

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

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

Neben der automatischen Auswertung arithmetischer Ausdrücke durch die Vergleichsoperatoren gibt es die Auswertung über den *is*-Operator. Dem *is* entspricht in Pascal/Java die Wertzuweisung mit := bzw. =.

```
Variable is <numerischer Term>
```

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

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*3.`
f) `?- X = 5 + 6*3.`
g) `?- 6 * -7 is Y.`
h) `?- 4*5 < 21`
i) `?- 4*5 is 20.`
j) `?- 4*5 = 20.`
k) `?- 20 is 4*5.`
l) `?- 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) * ... * 1.

   Beispiel:
   6! = 6*5! = 6*5* 4! = 6*5*4*3*2*1 = 720

7. Definieren Sie ein Prädikat *maximum(+L, -Max)* derart, dass das Ergebnis *Max* die größte Zahl der Liste L ist.

8. Informieren Sie sich in der SWI-Prolog-Dokumentation, wie Sie Zufallszahlen (engl. random number) erhalten.

9. Ermitteln Sie für ein Hauptgericht in der *Pizzeria Lucania* (vgl. 2.8, Aufgabe 6) den Gesamtpreis.

7 Logikrätsel

Viele Logikrätsel können mit Rechnereinsatz gelöst werden. Doch warum sollte man dies überhaupt machen? Rätsel sollen doch dem Zeitvertreib und der Unterhaltung dienen. Sie sollen durch Denken, logisches Kombinieren und manchmal ausdauerndens Ausprobieren gelöst werden.

Wenn wir hier Rätsel mit Hilfe von Prolog-Programmen lösen, so geht es natürlich um andere Aspekte. Zentral geht es immer um die Modellierung des Problems mit Prolog, wobei wir uns auf die deklarativen Aspekte konzentrieren. Die eigentliche Ermittlung der Lösung überlassen wir dem Prolog-Interpreter. Da wir unterschiedliche Rätselarten behandeln, werden natürlich auch diverse Prolog-Techniken vorgestellt und angewendet.

7.1 Alphametics

Alphametics leitet sich von *Alphabet* und *Arithmetics* ab. In einem Alphametic steht jeder Buchstabe für eine der Ziffern von 0 bis 9, unterschiedliche Buchstaben bedeuten unterschiedliche Ziffern und keine Zahl beginnt mit einer 0. Ein klassisches Beispiel ist:

SEND + MORE = MONEY

Als Lösungstechnik benutzen wir die Methode *Erzeugen und Testen.* Dabei werden der Reihe nach Lösungskandidaten erzeugt und dann getestet, ob der Lösungskandidat tatsächlich eine Lösung ist. Diese Methode lässt sich ganz einfach als Prolog-Programm umsetzen:

```
loese(X):-
  erzeuge(X),
  teste(X).
```

Mögliche Lösungskandidaten finden wir durch Permutation der Ziffern 0 bis 9 und anschließender Zuordnung der Ziffern zu den Buchstaben. Dafür benutzen wir das Systemprädikat *permutation*/2.

Beim Test, ob die erzeugte Permutation eine Lösung ist, prüfen wir ob S und M größer 0 sind und ob die Rechnung aufgeht. Zum Schluss geben wird die Lösung als Liste an.

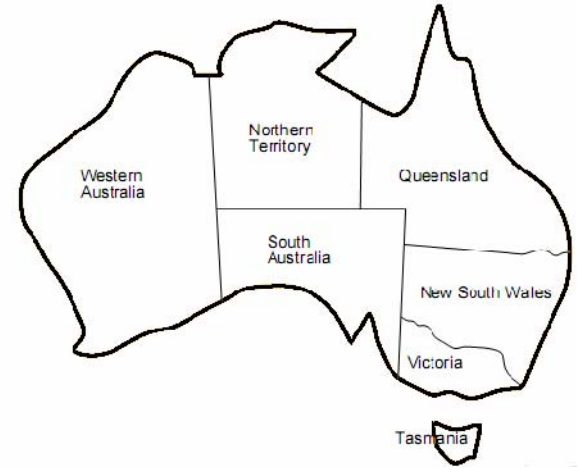
```
loeseRaetsel(L):-
  permutation([0,1,2,3,4,5,6,7,8,9],
              [S,E,N,D,M,O,R,Y,_,_]),
  S > 0,
  M > 0,
            S*1000 + E*100 + N*10 + D
          + M*1000 + O*100 + R*10 + E
                                    =:=
  M*10000 + O*1000 + N*100 + E*10 + Y,
  L = [S,E,N,D,+,M,O,R,E,=,M,O,N,E,Y].
```

Der deklarative Charakter unserer Lösung tritt deutlich hervor. Während der menschliche Rätsellöser aufgrund seiner Erfahrungen mit der schriftlichen Addition logisch ableitet, dass die fünfstellige Summe mit der Ziffer M=1 beginnen muss, sind in der Prolog-Lösung nur die Eigenschaften des Rätsels beschrieben, die eigentliche Lösung ermittelt uns Prolog.

7.2 Färbeproblem

Der Vier-Farben-Satz war das erste große mathematische Problem, das 1977 mit Hilfe von Computern gelöst wurde. Der Satz besagt, dass immer vier Farben ausreichen, um eine beliebige Landkarte so einzufärben, dass keine zwei benachbarten Länder die gleiche Farbe bekommen.

Wir betrachten zunächst ein einfaches Färbeproblem. Die Territorien von Australien sollen mit den drei Farben Rot, Grün und Blau eingefärbt werden. Wie geht das?



Mit einem einfachen Ansatz lässt sich das Problem lösen. Wir haben die drei Grundfarben:

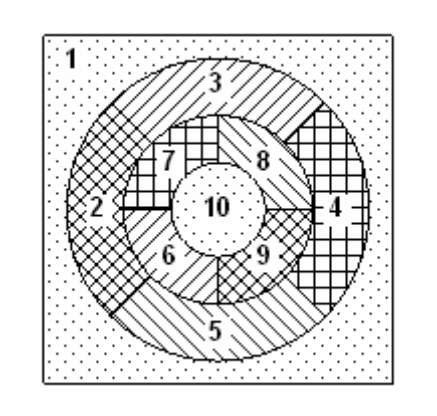
```
farbe(rot).
farbe(gruen).
farbe(blau).
```

Für eine Färbung braucht jedes Territorium eine der zulässigen Farben und die Farben benachbarter Territorien müssen verschieden sein:

```
faerbung(WA, NT, SA, QL, NSW, VIC, TAS):-
  farbe(WA), farbe(NT), farbe(SA),
  farbe(QL), farbe(NSW), farbe(VIC),
  farbe(TAS), WA\=NT, WA\=SA, NT\=SA,
  NT\=QL, SA\=QL, SA\=NSW, SA\=VIC,
  QL\=NSW, NSW\=VIC.
```

Obige Lösung ist auf das spezielle Problem zugeschnitten. Wir betrachten daher im nächsten Beispiel einen allgemeinen Ansatz zur Lösung von Färbeproblemen.

Im Bild wurden fünf Farben (dargestellt durch die Muster) benutzt, um die zehn Flächen einzufärben, vier Farben in den Kreisringen und die fünfte Farbe außen und innen. Wie müssen die Flächen eingefärbt werden, um mit vier Farben auszukommen?



Beim allgemeinen Ansatz trennen wir die Modellierung der Karte vom Lösungsalgorithmus. Der Lösungsalgorithmus kann somit für jede Karte benutz werden.

Bei der Modellierung einer Karte abstrahieren wir von den geometrischen Details, reduzieren auf die Nachbarschaftsrelation und verwenden dafür nachbar-Fakten:

```
nachbar(1, 2).
nachbar(1, 3),
nachbar(1, 4).
nachbar(1, 5).
nachbar(2, 3).
usw.
```

Zum Einfärben nehmen wir die vier Farben Rot, Grün, Blau und Weiß. Um Lösungskandidaten zu erzeugen, müssen wir jedem Land eine der vier Farben zuordnen. Das ist dann jeweils eine sogenannte *Variation* der vier Farben mit der Länge zehn. Nach der Lösungstechnik *Erzeugen und Testen* muss jeder Lösungskandidat noch überprüft werden.

```
loeseFaerbung(Faerbung):-
  variation(10, [rot, gruen, blau,
weiss],
            Faerbung),
  not(hatGleicheFarben(Faerbung)).
```

Die Variationen erstellen wir wie folgt:

```
variation(0, _, []).
variation(N, Farben, Liste):-
  N1 is N - 1,
  variation(N1, Farben, L1),
  member(Farbe, Farben),
  Liste = [N=Farbe | L1].
```

In der zweiten Klausel wird per Rekursion eine Variation mit um 1 kleinerer Länge ermittelt. Am Kopf wird dann ein Term der Form N=Farbe ergänzt, wobei Farbe eine der vier Farben ist. Anstelle des Gleichheitszeichens hätte man auch ein anderes Zeichen, das sich als binärer Operator eignet, nehmen können, z. B. /, % oder :. Wichtig ist nur, das wir einen binären Term haben, mittels dem der Landnummer eine Farbe zugeordnet werden kann. Das Prädikat erzeugt Färbungslisten der Art:

```
[10=gruen, 9=blau, 8=gruen, 7=gruen,
 6=rot, 5=rot, 4=weiss, 3=blau,
 2=gruen, 1=rot]
```

Zum Testen der Lösung erstellen wir ein Prädikat *hatGleicheFarben*, das für alle benachbarten Länder die Farben vergleicht.

```
hatGleicheFarben(Faerbung):-
  nachbar(L1, L2),
  member(L1=F1, Faerbung),
  member(L2=F2, Faerbung),
  F1 == F2.
```

Mit Hilfe von member und Einsatz von Unifikation werden die beiden Farben F1 und F2 für die beiden benachbarten Länder L1 und L2 bestimmt und verglichen.

Ist dieses Prädikat nicht erfüllbar, sind also bei allen benachbarten Ländern die Farben verschieden, so haben wir eine Lösung gefunden.

Diese Lösung des Färbungsproblems ist in hohem Maße deklarativ. Das Prädikat variation beschreibt, dass jedes der zehn Länder eine der vier Farben haben muss und mittels not(hatGleicheFarben) wird beschrieben, dass benachbarte Länder verschiedene Farben haben müssen.

7.3 Mathematische Rätsel

Peter hat drei Briefe, die er dringend sofort versenden möchte. Die Briefmarken, die er noch hat, reichen zwar insgesamt vom Wert genau, aber es sind welche mit ganz krummen Beträgen, solche wie man sie als "Wechselgeld" an Briefmarkenautomaten erhält.

Wie kann er die Marken auf die Briefe verteilen, damit alle Briefe richtig frankiert sind? Es handelt sich um einen normalen Brief für 55 Cent, einen DIN-A4-Brief für 1,44 Euro und eine Büchersendung für 77 Cent. Er hat jeweils eine Briefmarke zu 6, 18, 19, 20, 29, 32 und 40 Cent und zwei Briefmarken zu je 56 Cent.

Zur Lösung dieses Rätsels nach der Methode *Erzeuge und Teste* erstellen wir eine Permutation der Briefmarkenwerte. Dann zerlegen wir mittels *append* die Permutation in Teile, wobei jeder Teil die Briefmarken einer der drei Briefe mit der jeweils geforderten Summe haben soll.

```
loeseMarken(Loesung):-
  permutation([6, 18, 19, 20, 29, 32,
               40, 56, 56], Marken),
  append(Brief1, Brief23, Marken),
  append(Brief2, Brief3, Brief23),
  summe(Brief1, 55),  % vgl. Kap. 5
  summe(Brief2, 144),
  summe(Brief3, 77),
  Loesung = [Brief1, Brief2, Brief3].
```

Diese Lösung ist rein deklarativ, denn es wird nur ausgedrückt, dass bei einer Lösung alle Marken auf drei Briefe verteilt sind und dass die jeweiligen Summen den geforderten Werten entsprechen müssen.

Sehr effizient ist diese Lösung allerdings nicht. Um sie effizienter zu machen, muss man imperativ in den Lösungsweg eingreifen. Das kann beispielsweise so aussehen:

```
loeseBriefmarken(L):-
  L = [B1, B2, B3],
  M = [6,18,19,20,29,32,40,56,56],
  frankiere(B1, 55, M, M1),
  frankiere(B2, 144, M1, M2),
  frankiere(B3, 77, M2, []).
frankiere(B, W, M1, M2):-
  frankiere(B, 0, W, M1, M2).
frankiere(B, Akku, W, M1, M2):-
  auswaehlen(X, M1, Mr),
  X =< W,
  Akku1 = Akku + X,
  W1 is W - X,
  frankiere(B, Akku1, W1, Mr, M2).
frankiere(Akku, Akku, 0, M, M).
```

Hierbei frankieren wir mittels Akkumulatortechnik jeden Brief, wobei die Briefmarken so ausgewählt werden, dass richtig frankiert wird.

7.4 Logische Rätsel

Logik-Rätsel sollten eigentlich mit Prolog einfach zu lösen sein. Das stimmt auch, wenn man die Methode kennt. Sie werden elegant durch Belegung der Werte einer Datenstruktur und Auslesen der gesuchten Werte gelöst. Jeder Hinweis wird in einen Wert über die Datenstruktur übersetzt. Betrachten wir dazu das folgende Beispiel:

Drei Freunde erzielten in einem Wettbewerb die ersten drei Plätze. Jeder der drei hat einen anderen Vornamen, mag eine andere Sportart und gehört einer anderen Nation an. Michael mag Basketball und war besser als der Amerikaner. Simon, der Israeli, war besser als der Tennisspieler. Der Kricketspieler gewann. Wer ist der Australier? Welchen Sport betreibt Richard?

Als Datenstruktur nehmen wir eine Liste mit dem Namen Freunde, die drei Elemente hat, wobei jedes Element für einen der drei Freunde steht. Die Listenelemente sind selbst wieder Listen mit den Werten für Vorname, Sportart, Nationalität und Platzierung.

Die grundlegenden Angaben, wie z. B. dass Michael Basketball spielt, tragen wir direkt in diese Datenstruktur ein. Die Platzierung P1 wird für die Umsetzung der Tatsache, dass Michael besser als der Amerikaner war (P1 < Pa) benötigt.

Mit den member-Teilzielen im Regelrumpf von *loeseLogik* werden weitere Informationen in die Freunde-Datenstruktur eingetragen. Beispielsweise sorgt *member([_,_,_,1], Freunde)*, dass es in der Datenstruktur eine Platzierung mit Platz 1 gibt, und mit *member([_,_,amerikaner,Pa])* drücken wir aus, dass es unter den Freunden einen Amerikaner gibt, der die Platzierung Pa hat.

Ganz im Sinne der deklarativen Programmierung modellieren wir nur die gegebenen Informationen und überlassen es ganz dem Prolog-Interpreter die gesuchte Lösung zu finden. Dieser versucht mittels Backtracking und Unifikation für die gegebenen Bedingungen eine Lösung zu finden.

```
loeseLogik(Freunde):-
  Freunde = [[michael,basketball,_,P1],
             [simon,_,israeli,P2],
             [richard,_,_,_]],
  member([_,_,_,1], Freunde),
  member([_,_,_,2], Freunde),
  member([_,_,_,3], Freunde),
  member([_,_,amerikaner,Pa], Freunde),
  P1 < Pa,
  member([_,tennis,_,Pb], Freunde),
  P2 < Pb,
  member([_,kricket,_,1], Freunde),
  member([_,_,australier,_], Freunde).
```

7.5 Aussagenlogik

Aussagenlogische Rätsel werden durch eine Reihe von Aussagen beschrieben. Alle Aussagen müssen bei einer Lösung erfüllt sein. Oft haben die einzelnen Aussagen einen recht komplizierten Aufbau, in dem logische Negation (¬), logisches Und (∧), Oder (∨), Entweder-Oder (xor), Folgerung (wenn-dann, Implikation, ⇒), Äquivalenz (genau dann-wenn, ⇔) sowie Verknüpfungen davon vorkommen. Das Lösen aussagenlogischer Rätsel ist daher nicht ganz so einfach. Wir entwickeln eine Strategie anhand eines Beispiels.

Beispiel 1: Besuchsabend.
Meiers werden heute Abend zu Besuch kommen und zwar in folgender Konstellation:

Aussage 1: Wenn Herr Meier kommt, bringt er Frau Meier mit. Aussage 2: Mindestens eines der beiden Kinder Walter und Katrin kommt. Aussage 3: Entweder kommt Frau Meier oder Franziska, aber nicht beide. Aussage 4: Entweder kommen Franziska und Katrin oder beide nicht. Aussage 5: Wenn Walter kommt, dann auch Katrin und Herr Meier.

Zur Lösung des Rätsels machen wir folgenden Ansatz:

```
loesung(L):-
  L = [HM, FM, W, K, F],
  wenn_dann(HM, FM),
  oder(W, K),
  entweder_oder(FM, F),
  entweder_oder(und(F,K),
              und(nicht(F), nicht(K))),
  wenn_dann(W, und(K, HM)).
```

Dabei stehen die Variablen HM für Herr Meier, FM für Frau Meier und W, K, F sind die Kinder. Hat eine Variable den Wert w (wahr) so kommt die betreffende Person, beim Wert f (falsch) nicht.

Da die Argumente der aussagenlogischen Prädikate komplexe Ausdrücke sein können, müssen erst die Wahrheitswerte dieser Ausdrücke berechnet werden, um sie dann gemäß dem Prädikat zu verknüpfen. Beim Prädikat entweder_oder sieht das wie folgt aus:

```
entweder_oder(A, B):-
  berechne(A, AE),
  berechne(B, BE),
  entweder_oder(AE, BE, w).
```

Mit *berechne(A, AE)* und *berechne(B, BE)* werden zunächst die Ergebnisse AE und BE der aussagenlogischen Terme A und B berechnet. Die Wahrheitswerte von AE und BE können dann gemäß der Wahrheitstafel von *entweder oder* verknüft werden, wobei das Ergebnis wahr sein muss.

A	B	¬A	A∧B	A∨B	AxorB	A⇒B	A⇔B
w	w	f	w	w	f	w	w
w	f	f	f	w	w	f	f
f	w	w	f	w	w	w	f
f	f	w	f	f	f	w	w

Tabelle: Wahrheitstafeln der aussagenlogischen Verknüpfungen

Die Wahrheitstafeln lassen sich als Fakten modellieren:

```
entweder_oder(w, w, f).
entweder_oder(w, f, w).
entweder_oder(f, w, w).
entweder_oder(f, f, f).
```

Das berechne-Prädikat erhält einen aussagenlogischen Term, der zu einem Wahrweitswert w bzw. f vereinfacht werden muss:

```
berechne(w, w).
berechne(f, f):- !.
berechne(nicht(A), B):-
  berechne(A, AE),
  nicht(AE, B).
berechne(und(A, B), C):-
  berechne(A, AE),
  berechne(B, BE),
  und(AE, BE, C).
berechne(oder(A, B), C):-
  berechne(A, AE),
  berechne(B, BE),
  oder(AE, BE, C).
berechne(entweder_oder(A, B), C):-
  berechne(A, AE),
  berechne(B, BE),
  entweder_oder(AE, BE, C).
berechne(wenn_dann(A, B), C):-
  berechne(A, AE),
  berechne(B, BE),
  wenn_dann(AE, BE, C).
berechne(genau_dann(A, B), C):-
  berechne(A, AE),
  berechne(B, BE),
  genau_dann(AE, BE, C).
```

Da die berechne-Klauseln aber auch zum Erzeugen der Wahrheitswerte benutzt werden, braucht die zweite Klausel einen Cut, damit nicht außer den Wahrheitswerten auch noch komplexe Ausdrücke erzeugt werden.

Die anderen aussagenlogischen Prädikate werden ganz analog programmiert. Zum Beispiel:

```
wenn_dann(A, B):-
  berechne(A, AE),
  berechne(B, BE),
  wenn_dann(AE, BE, w).
wenn_dann(w, w, w).
wenn_dann(w, f, f).
```

```
wenn_dann(f, w, w).
wenn_dann(f, f, w).

oder(A, B):-
  berechne(A, AE),
  berechne(B, BE),
  oder(AE, BE, w).
oder(w, w, w).
oder(w, f, w).
oder(f, w, w).
oder(f, f, f).
```

Des Rätsels Lösung ergibt sich aus

```
?- loesung(L).
   L = [f, f, f, w, w]
```

also kommen Katrin und Franziska.

Beispiel 2: Krimi oder Western?
Was soll heute im Kino angeschaut werden – der Krimi oder der Western? Die 7 Freunde treffen je eine Aussage. Wir sollen herausfinden, wer was sehen möchte.

Bernd: "Gitte oder Ingrid wollen den Western sehen." Gitte: "Frank oder Jens schwärmen für den Krimi." Frank: "Entweder will Bernd oder Ingrid den Western sehen." Ingrid: "Gitte und Vera sind entweder beide für den Western, oder beide für den Krimi." Jens: "Entweder entscheiden sich Frank für den Western und Vera für den Krimi, oder es stimmen Ingrid für den Western und Petra für den Krimi." Petra: "Bernd und Franks Interesse gilt dem Western, oder Ingrid und Jens wollen nicht den Krimi." Vera: "Petra wählt den Western, oder Bernd und Gitte sehen lieber den Krimi."

Für jeden Freund benutzen wir den Anfangsbuchstaben seines Namens als logische Variable. Der Wahrheitswert w stehe für Western und f für den Krimi. Wenn ein Freund den Western sehen will, wird dafür einfach seine Variable geschrieben, wenn er lieber den Krimi sehen will, müssen wir den nicht-Operator benutzen. Bernds Aussag wird also zu: oder(G, J).

Gittes Aussage „Frank oder Jens schwärmen für den Krimi“ bedeutet: Frank schwärmt für den Krimi oder Jens schwärmt für den Krimi. Das wird umgesetzt zu: oder(nicht(F), nicht(J)).

Insgesamt erhalten wir die Lösung durch:

```
loesung(L):-
  L = [B, G, F, I, J, P, V],
  oder(G, I),
  oder(nicht(F), nicht(J)),
  entweder_oder(B, I),
  entweder_oder(und(G, V),
              und(nicht(G), nicht(V))),
  entweder_oder(und(F, nicht(V)),
                und(I, nicht(P))),
  oder(und(B, F), und(I, J)),
  oder(P, und(nicht(B), nicht(G))).

?- loesung(L).
   L = [f, f, f, w, w, f, f]
```

Ingrid und Jens wollen also in den Western, die fünf anderen in den Krimi.

Etwas komplizierter wird es, wenn Lügner mit im Spiel sind, denn dann können Aussagen wahr oder falsch sein. Zwei Fälle müssen berücksichtigt werden: sagt Person A die Wahrheit, so ist dessen Aussage B auch wahr, ist Person A hingegen ein Lügner, so ist dessen Aussage B falsch. Also haben wir $A \Rightarrow B$ und $\neg A \Rightarrow \neg B$. Beide Folgerungen können wir zu einer Äquivalenz zusammenfassen $A \Leftrightarrow B$. Betrachten wir als Beispiel für ein Wahrheit/Lügner-Problem folgende Logelei.

Beispiel 3: Logelei
Abianer sagen die Wahrheit, Bebianer lügen. Folgende Aussagen werden gemacht:

1. Knasi: Knisi ist Abianer.
2. Knesi: Wenn Knösi Bebianer, dann ist auch Knusi ein Abianer.
3. Knisi: Wenn Knusi Abianer, dann ist Knesi Bebianer.
4. Knosi: Knesi und Knüsi sind beide Abianer.
5. Knusi: Wenn Knüsi Abianer ist, dann ist auch Knisi Abianer.
6. Knösi: Entweder ist Knasi oder Knisi Abianer.
7. Knüsi: Knosi ist Abianer.

Wer ist Abianer und wer Bebianer?

Die sieben Personen kürzen wir mit Vokalen bzw. Umlauten ab. Eine Lösung des Problems erhalten wird dann wie folgt:

```
loesung(L):-
  L = [A, E, I, O, U, Oe, Ue],
  genau_dann(A, I),
  genau_dann(E,wenn_dann(nicht(Oe),U)),
  genau_dann(I, wenn_dann(U,nicht(E))),
  genau_dann(O, und(E, Ue)),
  genau_dann(U, wenn_dann(Ue, I)),
  genau_dann(Oe, entweder_oder(A, I)),
  genau_dann(Ue, O).

?- loesung(L).
   L = [f, w, f, f, w, f, f]
```

Also sind Knesi und Knusi Abianer.

Beispiel 4: Knobelmeister
Ein Knobelmeister und 4 Knobler machen Aussagen. Knobelmeister lügen nicht. Zwei der fünf Personen lügen. Folgende Aussagen werden gemacht:

Knobler A: Wenn ich kein Knobelmeister bin, dann spricht entweder C oder E die Wahrheit.
Knobler B: Die Aussage von A ist falsch, wenn D der Knobelmeister ist.
Knobler C: Wenn B oder E der Knobelmeister ist, dann Antwortet D nicht ehrlich.
Knobler D: Falls ich Knobelmeister bin, dann lügt weder A noch C.
Knobler E: Wenn A oder C die Wahrheit sagt bzw. sagen, dann bin ich Knobelmeister.

In diesem Beispiel soll nicht nur festgestellt werden wer die Wahrheit sagt und wer lügt, zusätzlich muss auch noch der Knobelmeister bestimmt werden. Wir setzen die Lösung daher mit zwei Listen an. In der ersten Liste halten wir für jeden Knobler fest, ob der die Wahrheit sagt oder lügt. In der zweiten Liste, ob er Knobelmeister ist oder nicht.

```
loesung(L1, L2):-
  L1 = [A, B, C, D, E],
  L2 = [F, G, H, I, J],
  genau_dann(A,
    wenn_dann(nicht(F), oder(C, E))),
  genau_dann(B,
    wenn_dann(I,
      nicht(wenn_dann(nicht(F),
              oder(C, E))))),
  genau_dann(C,
    wenn_dann(oder(G, J), nicht(D))),
  genau_dann(D,
    wenn_dann(I, und(A, C))),
  genau_dann(E,
    wenn_dann(oder(A, C), J)),
  gezaehlt(w, L1, 3),
  gezaehlt(w, L2, 1).
```

Mit dem Prädikat *gezaehlt/3* (vgl. Kapitel 5.5) wird gezählt wie oft der Wahrheitswert in den Listen L1 bzw. L2 vorkommt. Alternativ hätte man auch mit permutation/2 arbeiten können, um die beiden Eingangsbedingungen umzusetzen.

7.6 Suche im Zustandsgraph

Die Lösung mancher Rätsel kann dadurch gefunden werden, dass man sie als Zustandsgraph modelliert und dann einen Lösungspfad vom Start- zum Endzustand sucht.

Wir betrachten das klassische Bauer-Wolf-Ziege-Kohl Problem: Ein Bauer kommt mit einen Kohlkopf, einer Ziege und einem Wolf an das Nordufer eines Flusses, den er überqueren muss. Sein Boot ist aber so klein, dass er entweder nur den Kohlkopf oder nur die Ziege oder nur den Wolf hinüberfahren kann. Wenn Ziege und Kohlkopf unbeaufsichtigt zusammen sind, so frisst die Ziege den Kohl. Ebenso wird der Wolf die Ziege fressen, wenn der Bauer nicht aufpasst. Wie kommen alle ungefährdet an das Südufer?

Ein Zustand der Reisegesellschaft lässt sich durch eine vierelementige Liste mit den Positionen von Bauer, Wolf, Ziege und Kohlkopf darstellen. Als Positionen benutzen wir n für das Nordufer und s für das Südufer. Beispielsweise befinden sich im Zustand [n, s, n, s] Bauer und Ziege am Nordufer und Wolf und Kohlkopf am Südufer. Start- und Endzustand werden als Fakten definiert:

```
startzustand([n, n, n, n]).
endzustand([s, s, s, s]).
```

Bei jeder Überfahrt findet ein Zustandswechsel statt. Jeder Passagier kann von Nord nach Süd und umgekehrt wechseln, allerdings muss der neue Zustand zulässig sein, d. h. die Ziege befindet sich beim Bauer, oder Wolf und Kohlkopf sind bei ihm.

```
wechsel(n, s).
wechsel(s, n).

zulaessig([B, _W, Z, _K]):-
  B = Z.
zulaessig([B, W, _Z, K]):-
  B = W, B = K.
```

Der Bauer kann eine Überfahrt alleine machen oder einen Passagier mitnehmen:

```
% ueberfahrt nur Bauer
ueberfahrt([B1,W,Z,K], [B2,W,Z,K]):-
  wechsel(B1, B2),
  zulaessig([B2, W, Z, K]).
% ueberfahrt Bauer und Wolf
ueberfahrt([B1,W1,Z,K], [B2,W2,Z,K]):-
  wechsel(B1, B2),
  wechsel(W1, W2),
  zulaessig([B2, W2, Z, K]).

% ueberfahrt Bauer und Ziege
ueberfahrt([B1,W,Z1,K], [B2,W,Z2,K]):-
  wechsel(B1, B2),
  wechsel(Z1, Z2),
  zulaessig([B2, W, Z2, K]).

% ueberfahrt Bauer und Kohl
ueberfahrt([B1,W,Z,K1], [B2,W,Z,K2]):-
  wechsel(B1, B2),
  wechsel(K1, K2),
  zulaessig([B2, W, Z, K2]).
```

Eine Lösung besteht aus einem Pfad im Zustandsgraphen vom Start- zum Endzustand. Ein solcher Pfad wird als Liste aufeinanderfolgender Zustände dargestellt und mittels Akkumulatortechnik erzeugt. Da ein Graph Zyklen enthalten kann, sorgen wir dafür, dass sich Zustände nicht wiederholen.

```
loeseProblem(L):-
  startzustand(Start),
  loeseProblem(Start, [Start], L).
loeseProblem(Z, L1, L2):-
  endzustand(Z),
  reverse(L1, L2).
loeseProblem(Z1, L1, L2):-
  ueberfahrt(Z1, Z2),
  not(member(Z2, L1)),
  loeseProblem(Z2, [Z2|L1], L2).
```

Als zweites Beispiel für die Lösungstechnik *Suche im Zustandsgraphen* betrachten wie das Wasserkrugproblem. Zwei Wasserkrüge ohne Füllstandsanzeige fassen acht bzw. fünf Liter. Zu Beginn sind beide Krüge leer. Durch geeignetes Einfüllen von Wasser aus einem Wasserhahn, Umfüllen zwischen den Krügen bzw. Ausleeren eines Kruges soll erreicht werden, dass ein Krug vier Liter enthält.

Ein Zustand ist durch die Füllmengen der beiden Krüge bestimmt. Wir modellieren daher einen Zustand durch eine zweielementige Liste mit den Füllmengen des großen und kleinen Kruges. Der Start- und die beiden Endzustände können daher so angegeben werden:

```
startzustand([8, 0]).
endzustand([4, _]).
endzustand([_, 4]).
```

Auffüllen und Ausleeren eines Kruges wird wie folgt erledigt.

```
% kleinen Krug füllen
umfuellen([X, _], [X, 5]).

% großen Krug füllen
umfuellen([_, Y], [8, Y]).

% kleinen Krug ausleeren
umfuellen([X, _], [X, 0]).

% großen Krug ausleeren
umfuellen([_, Y], [0, Y]).
```

Einen Krug kann man aus dem zweiten Krug auffüllen bis der erste voll oder der zweite leer ist. Die Umfüllmenge wird einfach mit Hilfe des Systemprädikats *min* berechnet.

```
% großen Krug auffüllen
umfuellen([X1, Y1], [X2, Y2]):-
  Umfuellen is min(8 - X1, Y1),
  X2 is X1 + Umfuellen,
  Y2 is Y1 - Umfuellen.

% kleinen Krug auffüllen
umfuellen([X1, Y1], [X2, Y2]):-
  Umfuellen is min(5 - Y1, X1),
  X2 is X1 - Umfuellen,
  Y2 is Y1 + Umfuellen.
```

Mit dem Prädikat *loeseProblem* können nach Umbenennung von *ueberfahrt* in *umfuellen* die Lösungen ermittelt werden.

Eine Vertiefung des Themas Suchverfahren finden Sie in Kapitel 15. Mit der dort behandelten Breitensuche kann für das Wasserkrugproblem die kürzeste Lösung als erste Lösung gefunden werden.

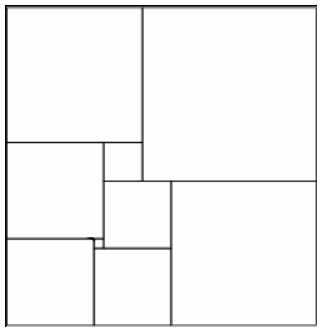
7.7 Aufgaben

1. Lösen Sie die Alphametics

a) THIS + ISA + GREAT + TIME = WASTER

b)
```
ABC + DDE =  FCF
  +     +      +
 CB + BGE =  BCA
  =     =      =
AEE + CBG = DGAE
```

2. Färben Sie diese Landkarte mit vier Farben:



3. Ein Bauer übergibt seinem Sohn 100 Taler. Dafür soll er auf dem Markt 100 Tiere kaufen und das gesamte Geld investieren, aber auch keine Schulden machen. Der Sohn soll jeweils mindestens 1 Rebhuhn, 1 Karnickel und 1 Schaf mitbringen. Auf dem Markt werden die Tiere zu folgenden Preisen angeboten: Rebhühner zu je 1/2 Taler, Karnickel zu je 3 Taler, Schafe zu je 10 Taler. Wie viele Tiere von jeder Gattung muss der Sohn mitbringen, um die Vorgaben des Vaters zu erfüllen?

4. Es gibt genau eine zehnstellige Zahl, die folgenden Anforderungen genügt: Jeder der Ziffern von 0 bis 9 kommt genau einmal vor Die Zahl, die aus den ersten n Ziffern gebildet wird, ist jeweils durch n ohne Rest teilbar Welche Zahl ist das?

5. In einer Strasse stehen fünf Häuser unterschiedlicher Farbe. Sie werden von fünf Männern verschiedener Nationalität bewohnt. Jeder der Männer hat in Bezug auf Rauchen, Trinken und Haustiere andere Gewohnheiten:

1. Der Engländer lebt in dem Haus mit der roten Türe.
2. Der Spanier hat einen Hund
3. Kaffee wird in dem Haus mit der grünen Türe getrunken.
4. Der Holländer trinkt Tee.
5. Das Haus mit der grünen Türe befindet sich direkt neben dem Haus mit der weißen Türe.
6. Im Haus mit der gelben Türe werden Zigaretten geraucht.
7. Der Zigarilloraucher hält sich Schnecken.
8. Der Norweger lebt neben dem Haus mit der blauen Türe.
9. Milch wird im mittleren Haus getrunken.
10. Der Norweger lebt im ersten Haus links.
11. Der Mann, der die Filterzigaretten raucht, lebt in dem Haus, welches neben dem Mann mit dem Fuchs liegt.
12. Zigarette wird geraucht im Haus, das neben dem Haus liegt, in dem man das Pferd hat.
13. Der Zigarrenraucher trinkt Orangensaft.
14. Der Japaner raucht Pfeife.

Wer trinkt Wasser und wem gehört das Zebra?

6. Peter, Thomas und Frank spielen Fußball. Dabei zerbricht eine Glasscheibe. Als sie verhört werden, beschuldigt jeder den anderen: Peter sagt: „Thomas lügt“. Thomas sagt: „Frank lügt“. Frank sagt: „Peter und Thomas lügen beide“. Wer sagt die Wahrheit und wer lügt?

7. Über das Frühstücksbuffet im Hotel konnte man hören:

Martin: Wenn Wurst oder Tee fehlten, dann war auch kein Kaffee da.
Norbert: Wenn keine Semmeln da waren, gab's weder Tee noch Fruchtsaft.
Robert: Wenn Käse vorhanden war, gab es auch genug Marmelade.
Stefan: Wenn die Butter fehlte, war jedenfalls Kaffee erhältlich.
Tobias: Gab es keinen Käse, dann wartete man auch vergeblich auf Wurst.
Uwe: Wenn man Fruchtsaft bekam, musste man auf Marmelade verzichten.
Volker: Wenn Kaffee fehlte, dann war auch keine Butter da.

Das hört sich schlimm an – war es wirklich so schrecklich im Hotel?

8. Auf der Insel *WaFa* leben *Wa's*, die immer die Wahrheit sagen, und *Fa's*, die immer lügen. Außer diesen beiden Gruppen gibt es keine Einheimische auf der Insel. Während unserer Inseldurchquerung fragen wir eine am Straßenrand stehende Gruppe Einheimischer nach dem Weg. Wir erhalten folgende Aussagen der Personen A, B, C, D, E, und G:

A Sie müssen den rechten Weg nehmen, und D lügt.
B Falls D lügt, sagen E und G die Wahrheit.
C Falls D die Wahrheit sagt, müssen Sie den linken Weg nehmen.
D Falls B oder G die Wahrheit sagt, dann lügt E.
E A oder D sagen die Wahrheit.
G B lügt, oder A sagt die Wahrheit.

9. Drei Kannibalen und drei Missionare stehen vor einem Urwaldfluss und wollen ihn überqueren. Sie haben nur ein Boot, das höchstens zwei Personen trägt. An und für sich wären die Kannibalen freundliche Gesellen, doch wenn sich an irgendeiner Stelle, sei es nur für einen Augenblick, mehr Kannibalen als Missionare befinden, so übermannt die Kannibalen ihre Lust nach Menschenfleisch und die Missionare würden blitzschnell aufgefressen. Wie kommen alle sechs ans gegenüberliegende Ufer?

8 Ein- und Ausgabe

8.1 Standard-Prädikate zur Ein- und Ausgabe

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

nl
Gibt das Steuerzeichen für den Zeilenvorschub aus.

tab(+Anzahl)
Gibt Anzahl Leerzeichen aus.

put(Char/Ascii)
Gibt das Zeichen Char bzw. das Zeichen mit dem betreffenden ASCII-Code aus.

write(+Term)
Gibt den Term in normaler Schreibweise aus.

writeln(+Term)
Wie write/1 aber mit zusätzlichem *nl*.

display(+Term)
Gibt Term in Präfix-Notation aus.

Für die Eingabe haben wir diese Standard-Prädikate:

read(-Term)
Liest einen beliebigen Prolog-Term ein.

get0(-Ascii)
Liest ein einzelnes Zeichen ein und gibt dessen ASCII-Code an.

get(-Ascii)
Liefert den ASCII-Code des eingelesenen Zeichens. Zeichen mit ASCII-Code <= 32 werden überlesen.

get_char(-Char)
Liest das nächste Zeichen ein.

get_single_char(-Ascii)
Liest ASCII-Code ohne auf Return zu warten ein

skip (+Ascii)
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 mit den klassischen Standardprädikaten *get* und *put*. Ohne ASCII-Tabelle wird es schwierig, aber die Sequenz 0'<Zeichen> kann anstelle des ASCII-Codes benutzt werden. Zum Beispiel steht 0'f für den ASCII-Code des Kleinbuchstabens f.

Etwas mehr Komfort liefert *get_char* bei der Eingabe, weil damit tatsächlich ein einzelnes Zeichen und nicht dessen ASCII-Code geliefert wird. Wenn die Eingabe nicht mit der Returntaste abgeschlossen werden soll, benutzt man *get_single_ char(-Ascii)*.

8.2 Menüsystem als Anwendung

Als Anwendung der Zeichenprädikate realisieren wir ein Menüsystem für die Hoteldatenbank aus Kapitel 4. Zum Austesten müssen Sie die Dateien frhotel.pl, frkunden.pl und frbuchen.pl konsultieren.

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

auswahl('1'):-
  listing(hotel),
  weiter.
auswahl('2'):-
  listing(kunde),
  weiter.
auswahl('3'):-
  listing(buchung),
  weiter.
auswahl('0'):-
  fail.

weiter:-
  % beseitigt LineFeed der letzten Eingabe
  skip(10),
  write('Bitte Returntaste drücken. '),
  skip(10),
  nl.
```

Was man in Pascal mit einer Repeat-Schleife macht, erledigt man in Prolog mit End-Rekursion. Für jeden Schleifendurchlauf ruft sich *start* selbst 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.

8.3 Lesen und Scheiben in Dateien

SWI-Prolog unterstützt das moderne Stream-Konzept zur Arbeit mit Dateien.

Beispiel 1: CSV-Export

Wir können beispielsweise die Kundendaten aus Kapitel 4 in eine CSV-Datei exportieren, um diese in einer Tabellenkalkulation weiterverarbeiten zu können. Eine CSV-Datei ist eine in Zeilen gegliederte Textdatei, bei der die einzel-

nen Datenelemente durch Semikola getrennt werden. Wir können dieses Format leicht erzeugen.

Mit dem Prädikat *open*/3 öffnen Sie eine Datei zum Lesen oder Schreiben. Über die Variable *Datei* hat man Zugriff auf den geöffneten Stream.

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

Die in der Wissensbasis gefundenen Kundendaten werden mit *write* in der für CSV erforderlichen Form in den Stream geschrieben. Das Systemprädikat *fail* schlägt immer fehl, wodurch Backtracking erzwungen wird und somit alle Kundendaten gespeichert werden. Die zweite Klausel sorgt dafür, dass *kundenExportieren* erfolgreich abgeschlossen wird.

```
kundenExportieren(Datei):-
  kunde(KNr,Name, StrasseNr, PLZ, Ort),
  write(Datei, KNr), write(Datei,';'),
  write(Datei, Name), write(Datei,';'),
  write(Datei, StrasseNr),
  write(Datei, ';'),
  write(Datei, PLZ), write(Datei,';'),
  write(Datei, Ort), nl(Datei),
  fail.
kundenExportieren(_).
```

Beispiel 2: CSV-Import

Den Import von Daten aus Dateien beginnen wir so:

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

Man kann mit *read(Datei, Term)* bequem aus Dateien einlesen, wenn die Daten in der Datei auch als Prolog-Terme zur Verfügung stehen. Das ist bei der CSV-Datei aber nicht der Fall. Die Rohdaten müssen erst eingelesen und dann in Prolog-Terme transformiert werden.

Mit *read_lines_to_codes* lesen wir eine Zeile als Liste von ASCII-Codes ein. Diese wandeln wir mit *atom_codes* in ein Atom. Dann zerlegen wir mit *concat_atom* das Atom am Trennsymbol ; in eine Liste.

```
kundenImportieren(Datei):-
  at_end_of_stream(Datei).
kundenImportieren(Datei):-
  read_line_to_codes(Datei, Codes),
  atom_codes(Atom, Codes),
  concat_atom(Liste, ';', Atom),
  Liste = [KNr, Name, StrNr, PLZ, Ort],
  assert(kunde(KNr, Name,
               StrasseNr, PLZ, Ort)),
  kundenImportieren(Datei).
```

Die so ermittelten Daten fügen wir mit *assert* (vgl. Kapitel 12) der Wissensbasis als kunde-Fakten hinzu. Rekursion sorgt dafür, dass alle kunde-Daten in die Datei geschrieben werden. Die Rekursion terminiert mit *at_end_of_stream*.

8.4 Nutzung der Systembibliotheken von SWI-Prolog

Das Systemprädikat *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 SWI-Prolog zugreifen. Sie besteht aus allen pl-Dateien im \library-Unterverzeichnis.

Die Bibliothek *readln.pl* bietet drei *readln*-Prädikate zum einfachen und flexiblen Eingeben von Texten und Zahlen an. Einzelheiten können Sie der Dokumentation in der Datei readln.pl entnehmen.

Mit dem Prädikat *lese_string* lesen Sie einen beliebigen String ein, ohne ihn in Anführungsstriche setzen und mit einem Punkt beenden zu müssen:

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

Mit dem Prädikat *lese_zahl* lesen Sie bequem Zahlen ein:

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

Readln/5 funktioniert wie ein Scanner, kann durch Optionen konfiguriert werden und eine ganze Liste von Symbolen liefern. Kommt die Eingabe nicht über die Tastatur oder eine Datei sondern aus einem berechneten Atom, so kann dieses mit *tokenize_atom/2* aus der Bibliothek *porter_stem* in Symbole zerlegt werden.

8.5 Formatierte Ausgaben

Für formatierte Ausgaben stellt SWI-Prolog das *format/2*-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]).

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

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

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

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

8.6 Aufgaben

1a) Wieso ist in Beispiel 1 die Klausel *auswahl('0'):- 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:  Steuerzeichen: 3
Zeichen eingeben:  Zeichen: &
Zeichen eingeben:  No
```

3. Implementieren Sie Prädikate für den Import und Export der Kundendaten im XML-Format.

4. Definieren Sie ein Prädikat *plural(+Wort, -WortImPlural)*, das englische Substantive in ihren Plural konvertiert.

   Anfrage: `plural(table, X).`
   Antwort: `X = tables.`

9 Der Cut !

9.1 So wirkt der Cut

Der Prolog-Interpreter arbeitet nach einem festen Verfahren, das auf Resolution, Backtracking und Unifikation aufbaut. Auf diese 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.

Der Cut ist ein nullstelliges Prädikat, das stets erfüllt ist und wie fast alle anderen Systemprädikate *deterministisch* ist, also nicht mehrfach erfüllt 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!

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 q1 erfüllt wird. Das Teilziel „!“ wird stets erfüllt. Es hat als Seiteneffekt die Wirkung, dass es Prolog auf alle Entscheidungen seit dem Aufruf von p festlegt, d. h. dass kein Backtracking mehr im Teilziel q1 sowie in der zweiten p-Regel stattfindet, wohl aber im Teilziel q2. Der Cut wirkt also lokal und global.

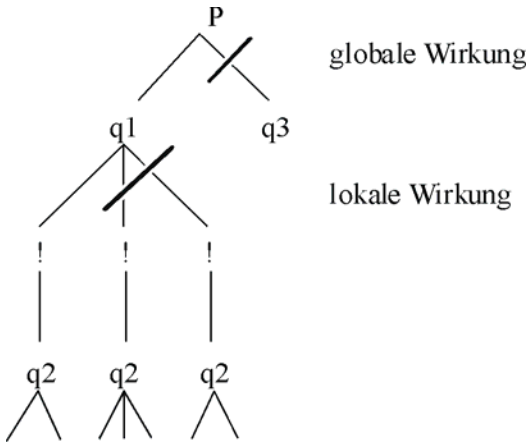


Abb. 9-8 Lokale und globale Wirkung des Cut im Suchbaum

Die *lokale Wirkung* bezieht sich auf die Klausel, in der er vorkommt. Er schneidet im Beweisbaum alle noch nicht untersuchten Zweige des aktuellen Knotens ab.

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

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, ✂, q2). Zum anderen stellt Sie die Operation des Abschneidens nachfolgender Klauseln dar (✂ (p:- q3)).

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

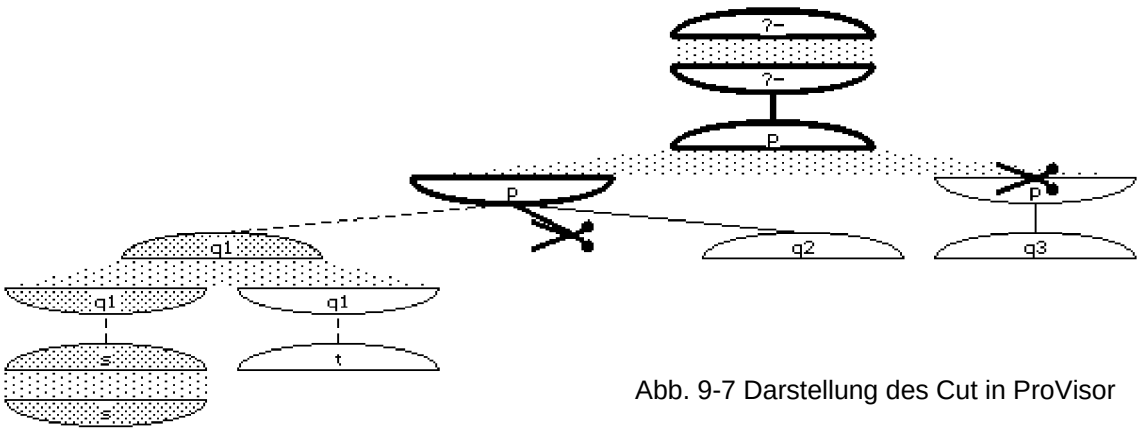


Abb. 9-7 Darstellung des Cut in ProVisor

Veranschaulichen wir den Cut noch 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 mehrfach erfüllt werden kann.

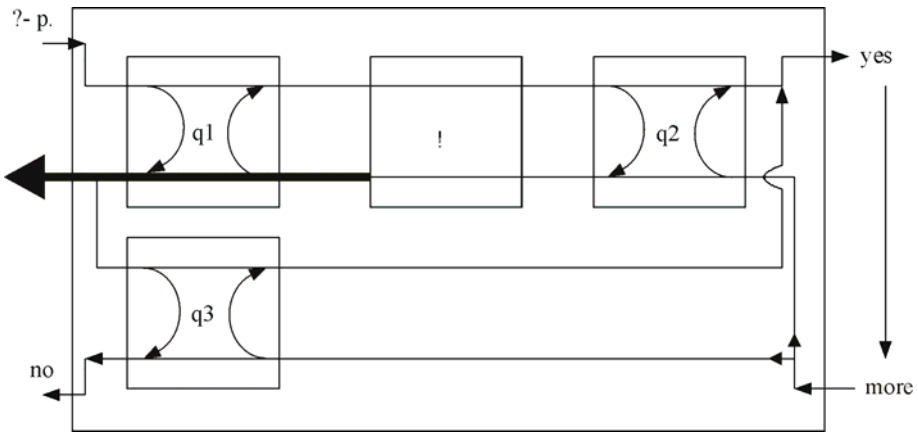


Abb. 9-9 Visualisierung des Cut im Vierport-Modell

- 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.

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

- Wir wissen, dass sich Gegebenheiten der Gegenwart nur auf die Zukunft, aber nicht auf die Vergangenheit auswirken. Zudem ordnen wir unbewusst einer Sequenz q1, !, q3 auch immer eine zeitliche Reihenfolge zu: erst q1, dann ! und schließlich q3. Der Cut wirkt auf die *Vergangenheit* q1.
- 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.
- Grundsätzlich gehen Menschen davon aus, dass Ursachen nur lokale Wirkungen haben. Das trifft beim Cut nicht zu.

9.2 Beispiele zum Cut

Beispiel 1: Vollständige Fallunterscheidung

In einem Land wird das Briefporto nach folgenden Regeln berechnet. Briefe bis 20 g kosten 0,55 €, bis 50 g 1,00 € und über 50 g kosten sie 1,45 €. Mit Prolog-Regeln schreiben wir:

```
porto(Gewicht, '0,55 Euro'):-
  Gewicht =< 20.
porto(Gewicht, '1,00 Euro'):-
  20 < Gewicht, Gewicht =< 50.
porto(Gewicht, '1,45 Euro'):-
  50 < Gewicht.
```

Eine mögliche Anfrage wäre:

```
?- porto(17, Porto).
```

mit dem Ergebnis: `Porto = '0,55 Euro'`

Fordert man weitere Lösungen an, so setzt Prolog die Suche mit der zweiten und dritten Klausel fort. Wir, aber nicht Prolog, wissen, dass es nach der ersten Lösung keinen Zweck hat, weiterzusuchen. Wenn einer der drei Fälle eingetreten ist, sind wir sicher, dass keine weiteren Lösungen existieren, da es sich um eine vollständige Fallunterscheidung handelt. Dieses Wissen können wir Prolog durch Setzen von Cuts mitteilen.

```
porto(Gewicht, '0,55 Euro'):-
  Gewicht =< 20, !.
porto(Gewicht, '1,00 Euro'):-
  Gewicht =< 50, !.
porto(Gewicht, '1,45 Euro'):-
  50 < Gewicht.
```

Da der Cut in der ersten Klausel für Gewichte bis 20 Gramm erfüllt wird und damit verhindert, dass die zweite Klausel benutzt wird, kann in der zweiten Klausel der Test 20 < Gewicht entfallen. Auf Kosten der Verständlichkeit könnte die dritte Klausel zu einem Fakt vereinfacht werden.

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.

Beispiel 2: Familienbeziehungen

Wir betrachten das folgende Programm

```
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).
```

Die erste kind-Klausel wurde um weiblich(E) ergänzt, um einen Cut setzen zu können, denn bei weiblichem Elternteil kann es sich nur um die Mutter handeln.

Bei der Anfrage `?- kind(max, karin).` wird der *Cut*, weil Karin *weiblich* ist, 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.

Aber der Cut schneidet in diesem Beispiel-Lösungen ab. Lässt man sich alle kind-Beziehungen mit ?- kind(K, E) berechnen, so findet man mit dem Cut nur eine Lösung, ohne den Cut sind es aber fünf Lösungen. Im Gegensatz zum ersten Beispiel liegt keine vollständige Fallunterscheidung vor, ein Kind kann eine Mutter und einen Vater haben.

Beispiel 3: Mischen

Zwei sortierte Listen L1 und L2 sollen zu einer sortierten Liste L3 zusammenfasst werden:

```
mische([K1|R1], [K2|R2], [K1|R3]):-
  K1 =< K2,
  mische(R1, [K2|R2], R3).
mische([K1|R1], [K2|R2], [K2|R3]):-
  K1 > K2,
  mische([K1|R1], R2, R3).
mische(L, [], L).
mische([], L, L).
```

Ist K1 =< K2 so können die weiteren Klauseln keine weiteren Lösungen liefern. Also kann man hinter den Vergleich einen Cut setzen:

```
mische([K1|R1], [K2|R2], [K1|R3]):-
  K1 =< K2, !,
  mische(R1, [K2|R2], R3).
```

Dieser Cut ist grün, es werden keine Lösungen abgeschnitten. Das Teilziel K1 =< K2 vor dem Cut hat nur eine Lösung, durch die lokale Wirkung des Cut können also keine Lösungen abgeschnitten werden. Die globale Wirkung bezieht sich auf die zweite bis vierte Klausel, die im Falle K1 =< K2 aber keine Lösungen liefern können.

Setzt man diesen Cut, so kann man in der zweiten Klausel den Vergleich K1 > K2 entfernen, denn der Cut verhindert im Falle K1 =< K2, dass die zweite Klausel zur Anwendung kommt.

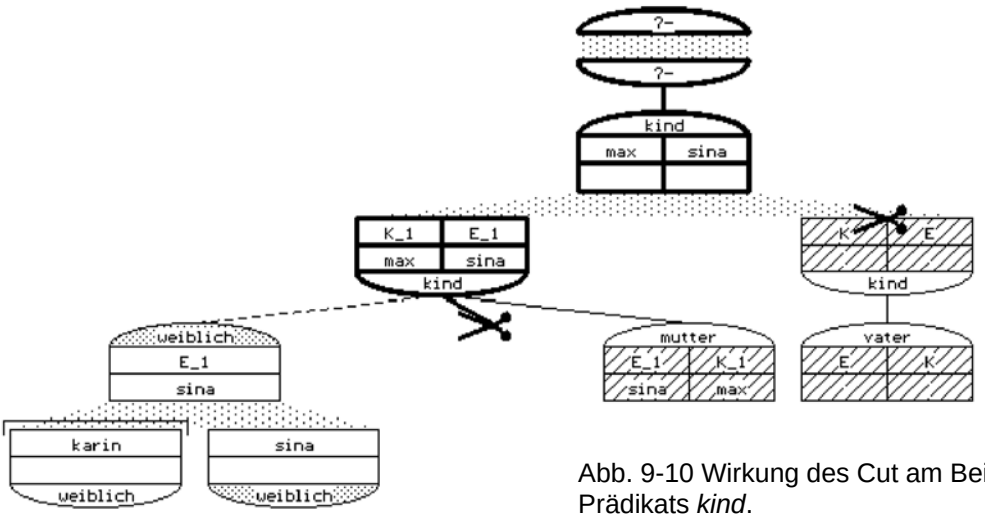


Abb. 9-10 Wirkung des Cut am Beispiel des Prädikats *kind*.

Beispiel 4: Roter Cut

Das member-Prädikat hatten wir in 5.3.1 wie folgt programmiert:

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

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.

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!

Beim Setzen eines Cuts muss man also auch bedenken, ob das Prädikat nur zum Prüfen oder auch zum Generieren von Lösungen eingesetzt werden kann. Rote Cuts können nur mit größter Vorsicht eingesetzt werden. Grüne Cuts sind dagegen harmlos und tragen zur Effizienzsteigerung bei.

Beispiel 5: Backtracking verhindern oder erzwingen

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.
all(_).
```

Beispiel 6: Repeat-Schleife

Mit dem Systemprädikat *repeat* lassen sich Schleifen programmieren. Beispiel:

```
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),
    !.
```

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,
   !.
```

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

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

Beispiel 7: If-Then-Else

Die in imperativen Programmiersprachen vorhandene If-Then-Else-Struktur lässt sich mit Hilfe des Cuts in Prolog nachbilden.

```
if_then_else(Bedingung, Anw1, Anw2):-
  Bedingung, !,
  Anw1.
if_then_else(Bedingung, Anw1, Anw2):-
  Anw2.
```

Doch dieser Ansatz mit zwei Klauseln für die beiden Fälle ist in der Programmierpraxis zu umständlich. Benötigt man in einer Klausel eine Fallunterscheidung, so ist der von SWI-Prolog bereitgestellte „->“ Operator das Mittel der Wahl.

Standardmäßig wird er wie folgt benutzt:

```
  (  Bedingung
  -> Anweisung1
  ;  Anweisung2
  )

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

Ein Prädikat zum Entfernen negativer Zahlen aus einer Liste (vgl. 5.5) kann wie folgt implementiert werden:

```
negative_raus([], []).
negative_raus([X|Rest1], Rest2):-
  X =< 0,
  negative_raus(Rest1, Rest2).
negative_raus([X|Rest1], [X|Rest2]):-
  X >= 0,
  negative_raus(Rest1, Rest2).
```

Mit Hilfe des „->“ Operators ergibt sich diese Lösung:

```
negative_raus([], []).
negative_raus([X|Rest1], Rest2):-
  (  X =< 0
  -> negative_raus(Rest1, Rest2)
  ;  negative_raus(Rest1, Rest3),
     Rest2 = [X | Rest3]
  ).
```

Beispiel 8: Cut-Fail-Kombination

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*.

9.3 Aufgaben

1a) Testen Sie das folgende fehlerhafte *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. 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. 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.
```

5. Implementieren Sie Lösungen der Prädikate gezaehlt/3, remove/3 und subst/4 aus Kapitel 5.5 mit Hilfe des „->“ Operators.

10 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.

10.1 Unifikation als Teil des Prolog-Beweisers

Zunächst betrachten wir die Unifikation anhand der beiden bekannten Beispiele *Familienbeziehungen* und *append*.

10.1.1 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).
```

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

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

Die Anfrage *elternteil(X, paul)* lässt sich mit den ersten vier Klauseln des Programms nicht unifizieren (gleichmachen), da sich die Funktoren *vater* bzw. *mutter* 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.

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 10-1 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* instanziert wurde.

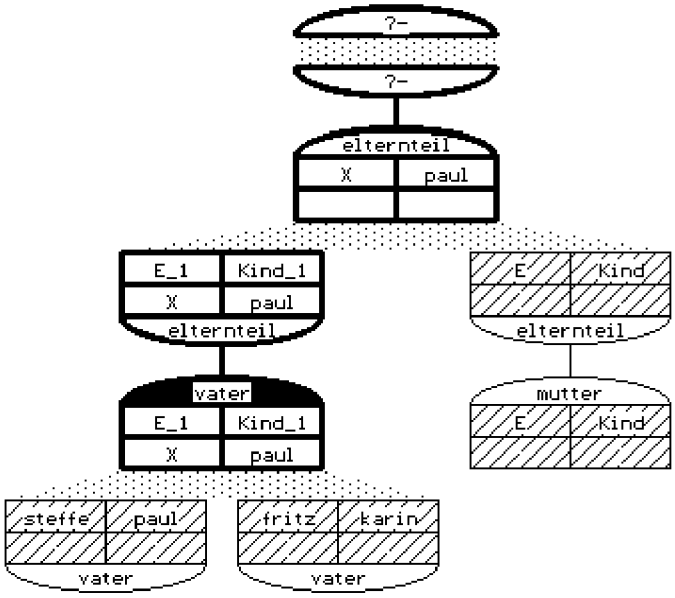


Abb. 10-1 Instanzierung von Variablen bei einer Unifikation

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

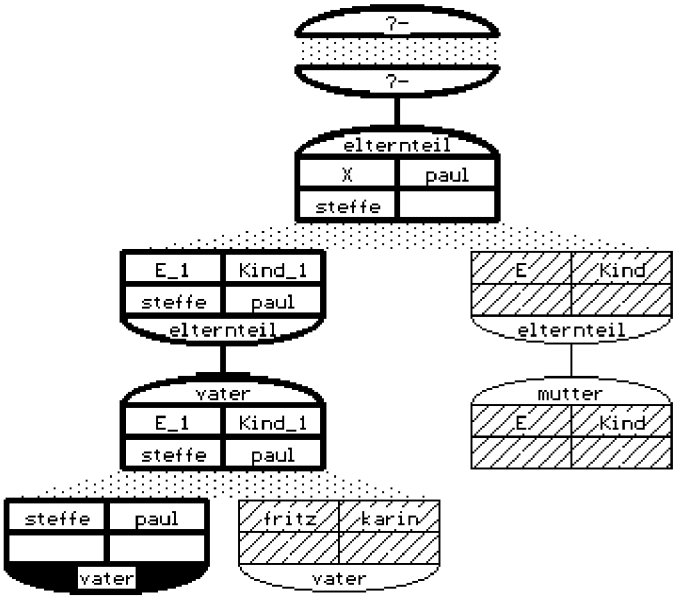


Abb. 10-2 Ein Unifikationsschritt

10.1.2 Listenzerlegung mittels append

Das Prädikat *append* kann man wie folgt implementieren:

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

Bei der Anfrage *?- 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?

```
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.

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

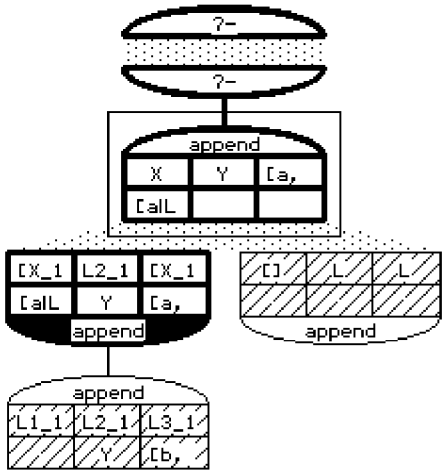
Die beiden Klauseln lassen sich dann durch folgende Substitution unifizieren:

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

Man erhält durch die Substitution den Term:

```
append([a|L1_1], L2_1, [a|[b, c, d]])
```

Im oberen Teilbild von Abbildung 10-3 sehen wir den Beweisbaum nach der ersten Unifikation. Im unteren 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.



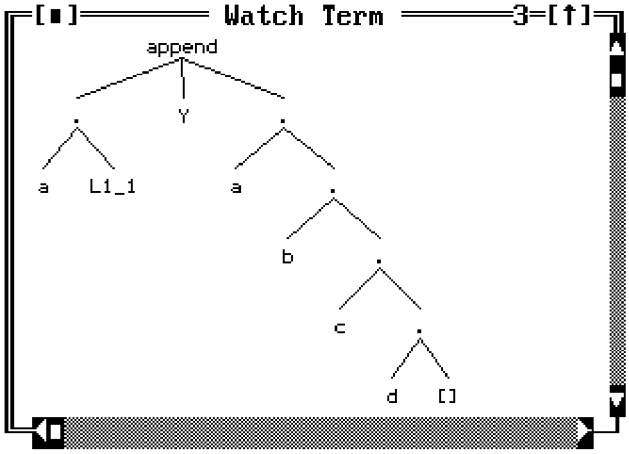


Abb. 10-3 Dynamische Watch-Terme in ProVisor

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:

```
append(L1_1, L2_1, [b, c, d])
```

Mit dem neuen Teilziel wird in der gleichen Weise wie eben beschrieben verfahren. Die Unifikation mit *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 10-4:

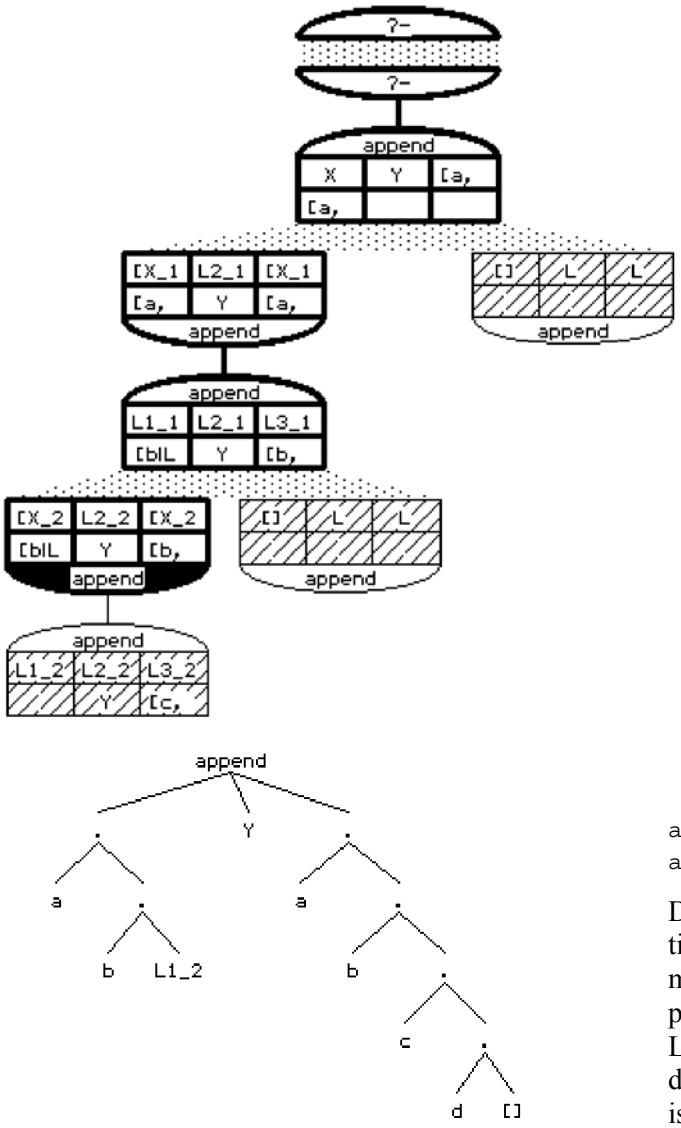


Abb. 10-4 Termgenerierung durch mehrfaches Unifizieren

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

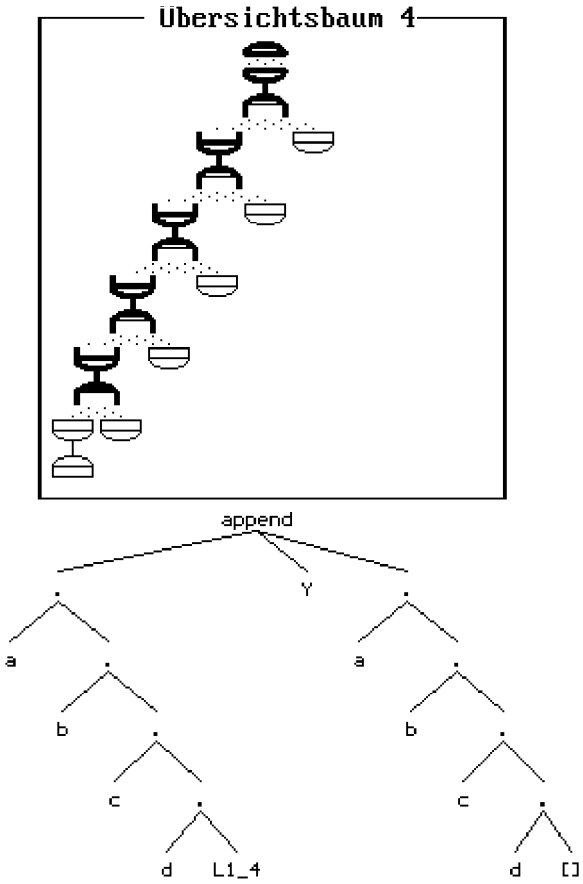


Abb. 10-5 Visualisierung von append kurz vor der ersten Lösung

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.

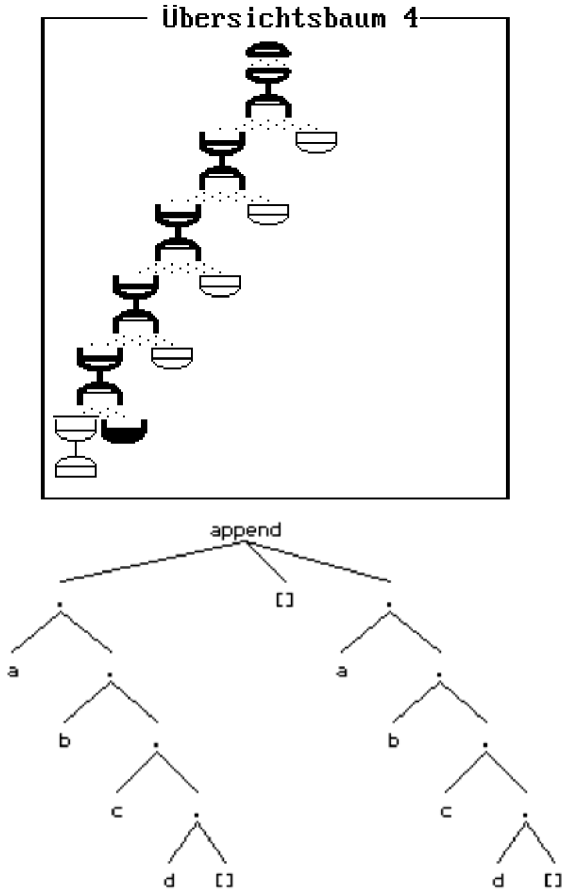
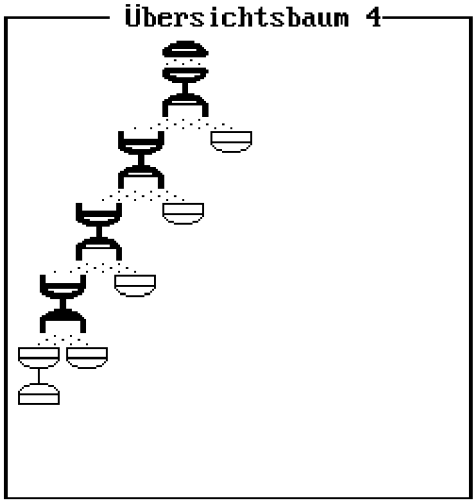


Abb. 10-6 Visualisierung von append 1. Lösung

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.



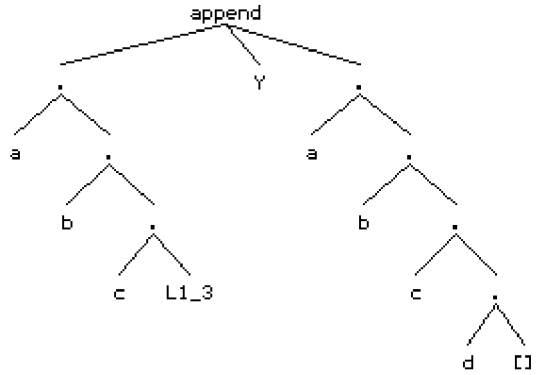


Abb. 10-7 Visualisierung von append kurz vor der zweiten Lösung

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].

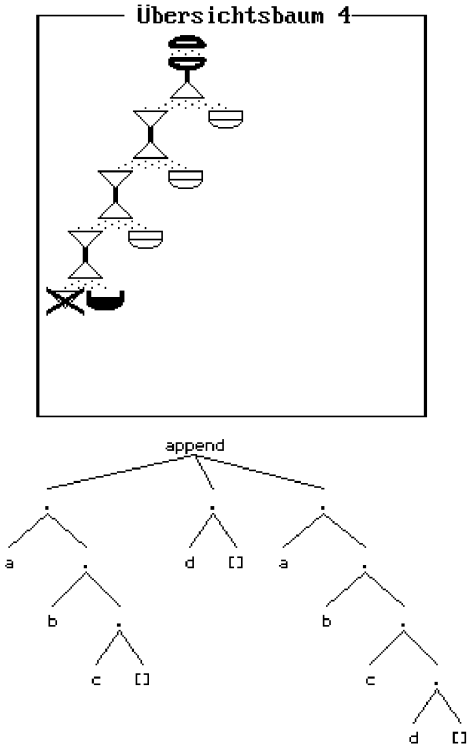


Abb. 10-8 Visualisierung von append 2. Lösung

Backtracking liefert drei weitere Lösungen:

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

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

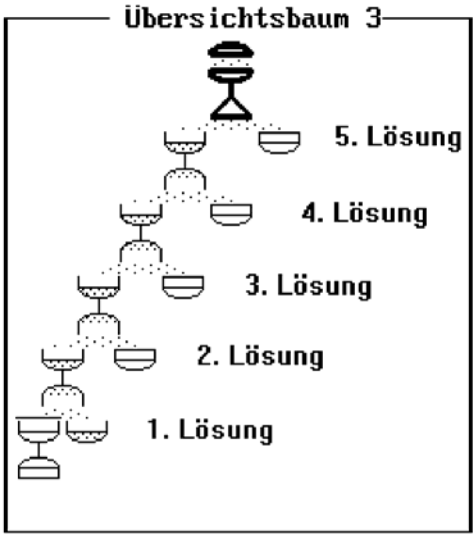


Abb. 10-9 Visualisierung von append, Übersicht zu allen Lösungen

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

Die erste Lösung entsteht dadurch, dass viermal die erste und dann die zweite Klausel ausgewählt werden. 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.

```
?- 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
```

10.2 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:

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.

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

Tabelle 10-1 Zusammenfassung möglicher Unifikationspartner

10.3 Ein Unifikationsalgorithmus

Der folgende Unifikationsalgorithmus 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

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
  1. X eine Variable ist, die nicht in Y vorkommt
     ersetze X im Keller und in U durch Y
     füge X = Y zu U hinzu
  2. Y eine Variable ist, die nicht in X vorkommt
     ersetze Y im Keller und in U durch X
     füge Y = X zu U hinzu
  3. X und Y identische Konstanten oder Variablen sind
     fahre fort
  4. X gleich f(X1,...,Xn) und Y gleich f(Y1, ...,Yn) für einen Funktor f und n > 0 ist
     schreibe Xi = Yi, i = n,..,1
     auf den Keller
  5. ansonsten
     Scheitern:= wahr
wenn Scheitern,
  dann gebe Scheitern aus,
  ansonsten gebe U aus.

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 n=3 haben, schreiben wir drei Gleichungen für die Argumente auf den Keller:

```
[a,b] = [X|Xs]
[c, d] = Ys
Ls = [X|Zs]
```

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 Unifikationsalgorithmus 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 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.

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 hinzugefügt, 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:

```
append([a, b], [c, d], [a|Zs]).
```

10.4 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 15.4 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 =.. [Funktor|Argliste1],
  Term2 =.. [Funktor|Argliste2],
  unify_list(Argliste1, Argliste2).

unify_list([], []).
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 10.3 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)`

11 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 ertragreichen Anlass, über das Verhältnis Mensch - Maschine und über natürliche sowie künstliche Intelligenz zu reflektieren.

11.1 Ableitungsregeln

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

```
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:

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

% Ableitung von X
ableitung(X, X, 1):- !.

% Vorzeichenregel
ableitung(-T, X, -Ta):- !,
  ableitung(T, X, Ta).

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

% Differenzenregel
ableitung(T1 - T2, X, T1a - T2a):- !,
  ableitung(T1, X, T1a),
  ableitung(T2, X, T2a).

% Faktorregel
ableitung(Konst*T, X, Konst*Ta):-
  atomic(Konst),
  Konst \== X, !,
  ableitung(T, X, Ta).

% Produktregel
ableitung(T1*T2, X, T1*T2a + T2*T1a):- !,
  ableitung(T1, X, T1a),
  ableitung(T2, X, T2a).

% Quotientenregel
ableitung(T1 / T2, X,
  (T1a * T2 - T1 * T2a)/(T2*T2)):- !,
  ableitung(T1, X, T1a),
  ableitung(T2, X, T2a).

% Ableitung von Grundfunktionen
ableitung(X^N, X, N*X^(N - 1)):-
  atomic(N),
  N \== X, !.
ableitung(sin(X), X, cos(X)).
ableitung(cos(X), X, -sin(X)).
ableitung(ln(X), X, 1/X).
ableitung(e(X), X, e(X)).
```

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(4*x^3*sin(x), x, A).
```

Lösung:

```
A = 4*x^3*cos(x)+sin(x)*(4*(3*x^(3-1)*1))
```

Die Lösung kommt zwar nicht in der Form, wie wir es gewohnt sind, aber sie ist richtig. Wenn man bedenkt, welche Schwierigkeiten 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!

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 ersten Mal bei der Konstantenregel zu. Da der Term x*(sin(x)/ln(x)) aber 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).

Das Unifikationsverfahren wird zur Mustererkennung eingesetzt, mit dem die auf einen Ableitungsterm anwendbaren Ableitungsregeln er-

mittelt werden. Backtracking sorgt dafür, dass bei der Suche nach anwendbaren Ableitungsregeln bei Bedarf alle Alternativen systematisch in Betracht gezogen werden.

11.2 Cuts in Ableitungsregeln

Grüne Cuts schneiden keine Lösungen ab, verhindern aber die unnötige Suche in Zweigen, von denen wir wissen, dass sie 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.

```
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:

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

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

11.3 Kettenregel

Die Kettenregel f(g(x))´ = f´(g(x))·g´(x) lässt sich einfach umsetzen, wenn man den *univ*-Operator (vgl. 16.4) 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).
```

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

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

11.4 Allgemeine Funktionen

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

Zur Implementierung der Ableitungsregel benötigt man den *univ*-Operator, um den Funktionsnamen vom Argument zu trennen. Mit Hilfe des Systemprädikats *atom_concat* hängt man an den Funktionsnamen ein S an:

```
ableitung(T, X, Ta):-
  T =.. [F, X],
  atom_concat(F, 'S', G),
  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)`

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.

11.5 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 (vgl. 16.4) auskommt, dafür mehr Schreibarbeit macht. Wir definieren ein Prädikat *vereinfachen(+Term, -VereinfachterTerm)* zur Vereinfachung von Summen, Differenzen, Produkten und Quotienten. Beispiel:

```
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:

```
summe(0 + T, T):- !.
summe(T + 0, T):- !.
summe(T1 + T2, T):-
  integer(T1),
  integer(T2),
  T is T1 + T2, !.

summe(0 - T1,  T):-
  integer(T1),
  T is - T1, !.
summe(T - 0,   T):- !.
summe(T1 - T2, T):-
  integer(T1),
  integer(T2),
  T is T1 - T2, !.

summe(A + B, B + A):-
  not(integer(A)),
  integer(B).
summe(T - T, 0):- !.
summe(A*T - T, S):-
  summe(A - 1, A1),
  produkt(A1 * T, S), !.
summe(T - A*T, S):-
  summe(1 - A, A1),
  produkt(A1 * T, S), !.
summe(A*T - B*T, S):-
  summe(A - B, A1),
  produkt(A1 * T, S), !.

summe(T1 + (T2 + T3), T4 + T3):-
  vereinfachen(T1 + T2, T4), !.
```

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

```
summe(T, T).
```

11.6 Aufgaben

1. Entwickeln Sie ein Prädikat zur Berechnung der n-ten Ableitung.
2. Geben Sie in Prolog eine Ableitungsregel 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),
   substituiere(x, 4, X, Y),
   vereinfachen(Y, Z).
```

6. Entwickeln Sie Prädikate zum symbolischen Integrieren.
7. Implementieren Sie ein Prädikat zum Vereinfachen arithmetischer Ausdrücke auf der Basis des univ-Operators. (vgl. 16.4).

12 Wissensbasis und Regelsysteme

12.1 Hinzufügen und Löschen von Klauseln

Fakten und Regeln wurden bislang mittels *consult* 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:

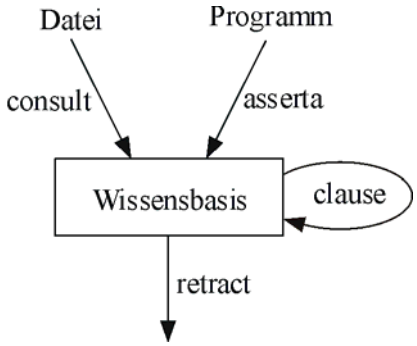


Abb. 12-1 Prädikate zur Manipulation der Wissensbasis

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.

Prädikate, die aus Dateien konsultiert werden, können nur dann um Klauseln ergänzt werden, wenn sie zuvor mit

```
:- dynamic funktor/arität.
```

als dynamisch deklariert wurden. Betrachten wir dazu folgendes Beispiel, bei dem die Wissensbasis zunächst zwei Klauseln zum Prädikat *maennlich* hat:

```
:- dynamic maennlich/1.
maennlich(heinz).
maennlich(manfred).
```

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

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

assertz(maennlich(jens)) ergibt:

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

Mit *assert* können Sie auch *Regeln* der Wissensbasis zufügen.

Beispiele:

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

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).
```

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

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

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

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

Nur *retract*, *repeat* und *clause sind* in Standard-Prolog *backtrackingfähig*. Alle anderen Systemprädikate sind *deterministisch*, also nur ein einziges Mal erfüllbar.

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 (vgl. Kapitel 3.4) ist ein solcher Metainterpreter.

12.2 Einfache Anwendungen

Beispiel 1: Zufallszahlen

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

```
S = (9749 * S) 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:

```
:- dynamic basis/1.
basis(4567).
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
```

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:

```
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 ineffizient, 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.

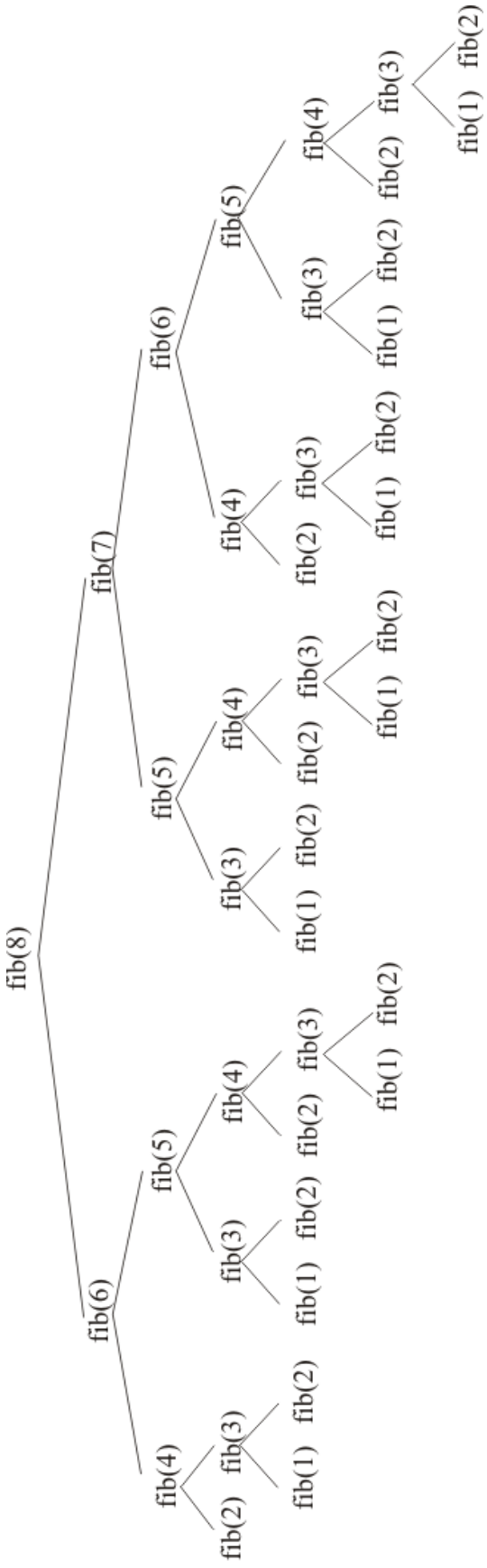


Abb. 12-2 Doppelte Rekursion bei den Fibonacci-Zahlen

Die neue *fib*-Regel lautet:

```
:- dynamic fib/2.
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)), !.
```

12.3 Mengenprädikate

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.

Für diesen Zweck bietet Prolog das Systemprädikat *findall* an, 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. Als Antwort erhält man:

```
Kind = _G465
KindListe = [paul, lisa]
```

In SWI-Prolog stehen neben *findall* auch noch die beiden weiteren Mengenprädikate *bagof* und *setof* als Systemprädikate zu Verfügung.

12.4 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:

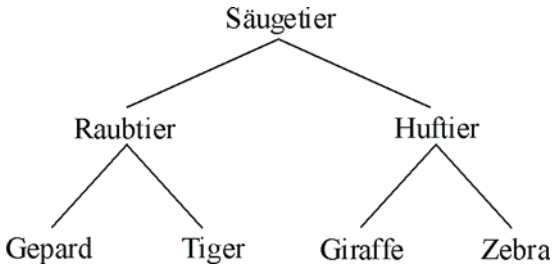


Abb. 12-3 Begriffshierarchie für Säugetierarten

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.

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:

```
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.

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

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.

12.5 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. Wie sammeln Sie alle Nachbarräume von Raum e im Labyrinth von Kapitel 5.5 in einer Liste?

5. 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.

6. Definieren Sie ein Prädikat *delete(+Funktor, +Arität),* das alle Klauseln zu einem Funktor gegebener Arität löscht. Verwenden Sie zur Lösung das Systemprädikat *retract*.

7. 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
:- dynamic verlust/1, gewinn/1.

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),
  writeln(' Streichhölzer!').
zug(X):-
  write('Der Gegner kann gewinnen '),
  writeln('nimm beliebig.').
```

8a) Komplettieren Sie das Regelsystem aus 12.4 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')
```

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.

9. 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 Aspirol ist wirksam bei Kopfschmerzen.
F2 Aspirol 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 Aspirol 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.
```

13 ICE-Auskunftssystem

13.1 Modellbildung

Die Deutsche Bundesbahn bietet elektronische Fahrplanauskunft auf CD-ROM und per Internet an. Wir wollen im folgenden selbst ein solches Informationssystem für die Bahnauskunft konzipieren und realisieren, um dabei typische Informatikmethoden wie zum Beispiel Analyse, Modellierung und Konstruktion komplexer Informatiksysteme kennen zu lernen und anzuwenden.

Wir gehen vom *CityFahrplan* aus, in welchem alle ICE-, EC- und IC-Verbindungen aufgeführt sind. Beschränkt man sich auf die ICE-Züge und einige relevante Strecken, so kommt man zum Streckennetz in Abbildung 13-1.

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-Beziehungstyp-Diagramm) modellieren. 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 Beziehungsrelation *liegt auf* integriert werden, weil Linie nur ein einziges Attribut hat.

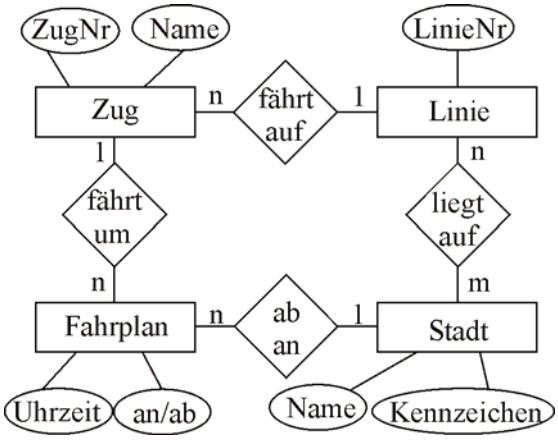


Abb. 13-2 ER-Diagramm des ICE-Auskunftsystems

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

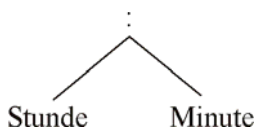
```
stadt(Name,
      Kennzeichen)
linie(LinieNr,
      Kennzeichen)
zug(LinieNr,
    ZugNr, Name)
fp(ZugNr, Uhrzeit,
  Kennzeichen, an/ab)
```

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-Klauselform angegeben:

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

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* @< und @>= (vgl. 16.3) 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.

13.2 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.
```

13.3 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(_).
```

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
----------------------
```

Die Kosmetik wird durch formatierte Ausgaben erreicht, welche wir in Kapitel 8.4 kennen gelernt haben. Für Uhrzeiten benutzen wir eine eigenständige formatierte Ausgabe, um bei Minuten führende Nullen ergänzen zu können:

```
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).
```

13.4 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)).
```

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)*.

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).
```

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:

```
abfahrt(Stadt):-
  stadt(Name, Stadt), nl,
  write('Abfahrt: '),
  writeln(Name), nl,
  writeln('    ab  Zug nach     an'),
  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(Ankunft), nl,
  zeige_abfahrten(Rest).
zeige_abfahrten([]).
```

Damit erhält man beispielsweise folgenden Abfahrtsplan:

```
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
------------------------------
```

13.5 Zugauskunft

Sie wollen morgens, frühestens ab 7^{00} 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).
```

Mit etwas Aufwand

```
zeige_kopf:-
  nl,
  writeln('Reiseverbindung DB'),
  writeln('Bahnhof        Uhr      Zug'),
  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
```

Die Lösung für eine direkte Verbindung lässt sich problemlos auf eine Umsteigeverbindung 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.

```
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:

```
Reiseverbindung    Deutsche Bundesbahn
Bahnhof                 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:

```
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.

13.6 Heuristische Suche im ICE-Netz

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

```
Reiseverbindung    Deutsche Bundesbahn
Bahnhof                 Uhr      Zug
--------------------------------------
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 15 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.

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 Umsteigeverbindungen gesucht. Es werden alle Städte bestimmt, die von dieser Stadt aus direkt erreichbar sind, insgesamt also durch eine Umsteigeverbindung 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.

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.

```
heuristischesuche(Start,Ziel,AbStart):-
  stadt(_, Start),
  stadt(_, Ziel),
  heuristischesuche1([ver(AbStart,
    [Start], [])], Ziel, AbStart).

heuristischesuche1(Pfade,Ziel,AbStart):-
  Pfade = [ver(_, [Ziel|Rest], _)|_],
  reverse([Ziel|Rest], Verbindungen),
  zeige_kopf,
  zeige_verbindungen(Verbindungen,AbStart).

heuristischesuche1([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),
```

```
  heuristischesuche1(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:

```
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    Deutsche Bundesbahn
Bahnhof                 Uhr      Zug
--------------------------------------
München             ab  7:46   ICE 598
Mannheim            an 10:32
Mannheim            ab 10:58   ICE 771
Freiburg            an 12:27
```

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.

13.7 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.

14 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 13 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.

14.1 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 in Abbildung 14-1 zu sehen.

Das Diagramm besteht aus fünf Objekttypen, dargestellt durch Rechtecke, und fünf Beziehungen, dargestellt durch Rauten. In den Ovalen stehen die Attribute der Objekttypen beziehungsweise 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 dem gebuchten Hotel, 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 werden. 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 vier anderen Beziehungen sind vom Typ 1-n: ein Hotel hat mehrere Preise, jeder Preis gehört zu einer Saison, in einem Gebiet liegen

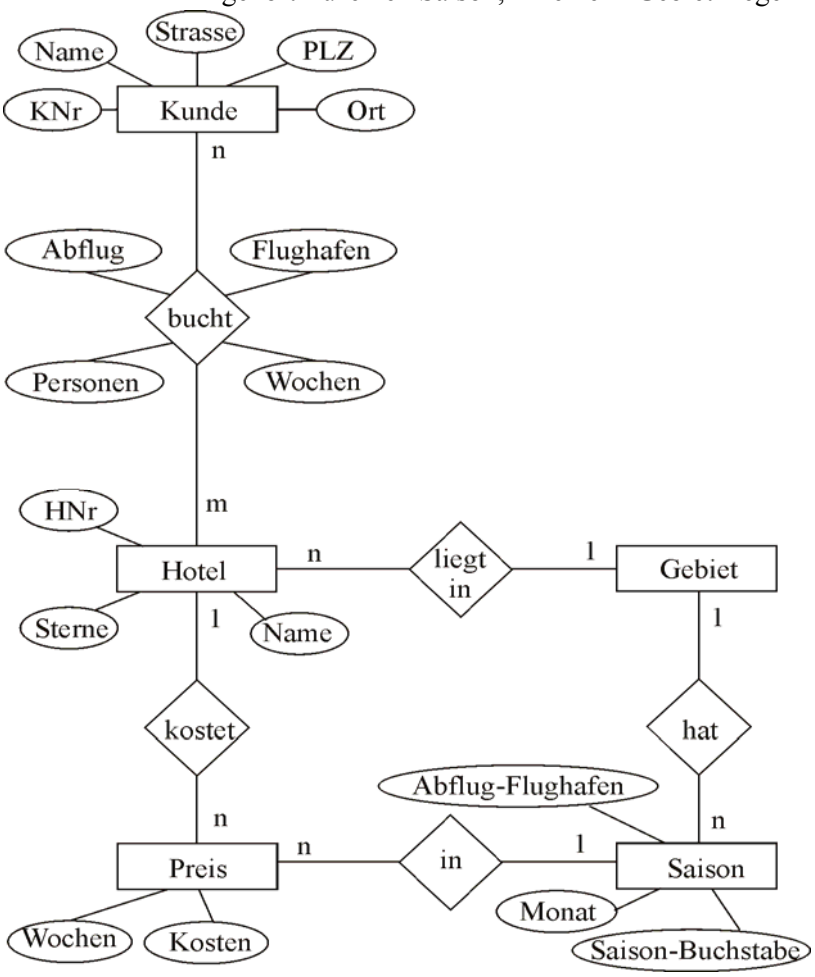


Abb. 14-1 ER-Diagramm des Auskunfts- und Reisebuchungssystems

mehrere Hotels und dieses Gebiet wird von verschiedenen Flughäfen aus in den Urlaubsmonaten angeflogen.

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 braucht man keine Relation, weil keine Attribute gespeichert werden. Das wäre anders, wenn man beispielsweise den Ziel-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.

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 Relationen beschrieben:

```
kunde(KNr, Name, Strasse, PLZ, Ort).
hotel(HNr, Name, Sterne, Gebiet).
preis(HNr, Saison-Buchstabe, Wochen,
      Kosten).
saison(Gebiet, Abflug-Flughafen,
       Monat, Saison-Buchstabe).
buchen(KNr, HNr, Abflug, Personen,
       Abflug-Flughafen, Wochen).
```

14.2 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.

```
start:-
  initialisierung,
  menue,
  terminierung.

initialisierung:-
  titel,
  retractall(wahl(_)),
  asserta(wahl(gebiet(_))),
  asserta(wahl(hotel(_))),
  asserta(wahl(kunde(_))),
  writeln('Lade Datenbank...'),
  consult(frkunden),
  consult(frbuchen),
  consult(frhotel), nl.

titel:-
  writeln('----- Buchungs- und Aus
          kunftssystem ------'),
  writeln('-----  Froh-Reisen GmbH,
          Darmstadt  ------'),
  nl.

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_char(Ch),
  skip(10),
  wahl_ausfuehren(Ch),
  Ch \== '0', !,
  menue.
menue.

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(_)).
```

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('1'):-
  writeln(' Urlaubsgebiete'), nl,
  gebiete_zeigen,
  gebiet_waehlen.

wahl_ausfuehren('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('3'):-
  writeln(' Kunden'), nl,
  write('Name oder Nummer: '),
  lese_string(Kunde),
  bearbeite_kunde(Kunde),
  weiter.

wahl_ausfuehren('4'):-
  writeln(' Buchen'), nl,
  hole_kunde(Kunde),
  hole_hotel(Hotel),
  buchen(Hotel, Kunde).

wahl_ausfuehren('5'):-
  writeln(' neuer Kunde'), nl,
  retractall(wahl(_)),
  asserta(wahl(gebiet(_))),
  asserta(wahl(hotel(_))),
  asserta(wahl(kunde(_))).

wahl_ausfuehren(_).
```

14.3 Gebietsverwaltung

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 SWI-Prolog. Es sortiert eine Liste und eliminiert dabei die Doubletten:

```
% --- Gebietsverwaltung ---------------

gebiete_zeigen:-
  findall(Gebiet, hotel(_, _, _,
          Gebiet), Liste1),
  sort(Liste1, Liste2),
  gebiete_zeigen(Liste2),

gebiete_zeigen([K|R]):-
  tab(2), writeln(K),
  gebiete_zeigen(R).
gebiete_zeigen([]).
```

Nach der Anzeige hat der Benutzer die Möglichkeit, ein Urlaubsgebiet auszuwählen.

```
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))).
```

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.

14.4 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.

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), !.

%-- Hotelnummer
bearbeite_hotel(Hotel):-
  hotel(Hotel, _, _, _),
  retract(wahl(hotel(_))),
  asserta(wahl(hotel(Hotel))),
  zeige_hotel(Hotel), !.
```

Wenn der Benutzer im Hauptmenü die Buchungsverwaltung aufruft, wird unter anderem das ausgewählte Hotel aus der Wissensbasis geholt und zur Kontrolle angezeigt. Wurde noch kein 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).
```

14.5 Kundenverwaltung

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 SWI-Prolog genutzt.

```
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,_,_,_,_),
         Liste1),
  sort(Liste1, Liste2),
  last(KNr1, Liste2),
  KundenNr is KNr1 + 1,
  speicher_kunde(kunde(KundenNr, Kunde,
                 Strasse, PLZ, Ort)),
  !.

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))*.

14.6 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(' Euro'),
  write('Buchen (ja/nein): '),
  lese_string(Buchen),
  Buchen = 'ja',
  asserta(buchung(Kunde, Hotel,
    Personen, datum(Tag, Monat, Jahr),
    Wochen, Flughafen)),
  writeln('Gebucht.').
buchen(_, _).

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), FHaefen),


  (  Haefen = []
  -> write('Abflug nicht möglich.
            Keine Saison in '),
     writeln(Gebiet)
  ;  write('Abflug möglich ab: '),
     writeln(FHaefen)
  ),
  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):-
  preis(Hotel, Saison, Wochen, Kosten),
  Preis is Personen * Kosten, !.
```

14.7 ODBC-Zugriff auf Datenbanken

SWI-Prolog ermöglicht ODBC-Zugriff auf lokale und Web-Datenbanken. Die Dokumentation der Bibliothek odbc.pl findet man auf der SWI-Prolog-Homepage.

In der ODBC-Verwaltung der Systemsteuerung richtet man die Datenquelle (DSN, Data Source Name) ein. Mittels *odbc_connect* stellt man eine Verbindung zur Datenquelle her und mit *odbc_query* kann man SQL-Abfragen stellen.

In unserem einfachen Beispiel hat die Datenquelle den Namen *swi*. Odbc_connect liefert für diese Datenquelle die Verbindung *Connect*, über die die Abfrage verarbeitet wird. Aus der Tabelle Adressen werden Name und Vorname ermittelt und mittels *assert* in der Wissensbasis abgespeichert.

```
odbc_zugriff:-
  odbc_connect(swi, Connect, []),
  abfrage(Con, 'SELECT Name,
          Vorname FROM Adressen').

abfrage(Connect, Abfrage):-
  odbc_query(Connect, Abfrage,
             row(Name, Vorname)),
  assert(adresse(Name, Vorname)),
  fail.
abfrage(_, _).
```

14.8 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. Führen Sie eine eigene Relation für die Gebiete ein, mit Ziel-Flughafen und Gebietsname. Nehmen Sie die notwendigen Anpassungen vor.
7. Verwalten Sie die Daten des Auskunfts- und Reisebuchungssystem in einer Datenbank.

15 Suchverfahren

15.1 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 selbst Schachweltmeister keine Chance mehr haben.

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

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 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.

Im Folgenden beziehen wir uns auf ungerichteten Graphen von Abbildung 15-1. 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:

```
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:

```
verbunden(A, B):- kante(A, B, _).
verbunden(A, B):- kante(B, A, _).
```

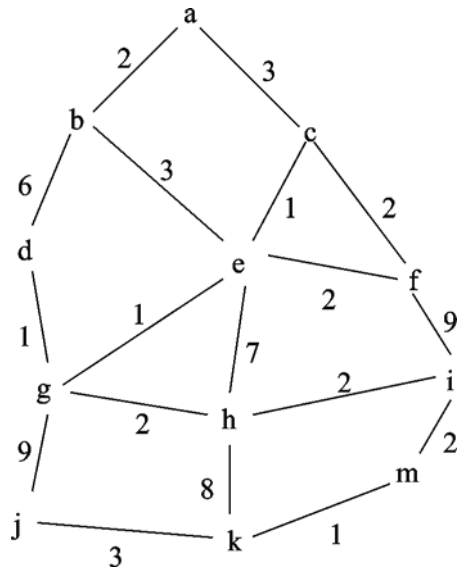


Abb. 15-1 ungerichteter, bewerteter Graph

15.2 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... 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(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).
```

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 mit Hilfe der Akkumulatortechnik wie angegeben.

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 oder mit reverse/2 umdrehen. Die Pfadliste benutzen wir für den Zyklustest und zur Ausgabe der Lösung. Als erste Lösung erhält man:

L = [k, m, i, h, g, d, b, a]

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.

Der zweite Parameter wird als Akkumulator benutzt, in dem der bisherige Pfad gespeichert wird. Stimmt der zuletzt erreichte Knoten mit dem Ziel überein, wird der Akkumulatorinhalt in der ersten tiefensuche/4-Klausel im vierten Parameter als Lösung ausgegeben.

15.3 Breitensuche

Die Tiefensuche ist uns von Prolog und von binären Suchbäumen her wohlbekannt. 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 ausbreitet. Die Wellenfront gibt die aktuellen Knoten an, von denen aus die Suche fortgesetzt wird.

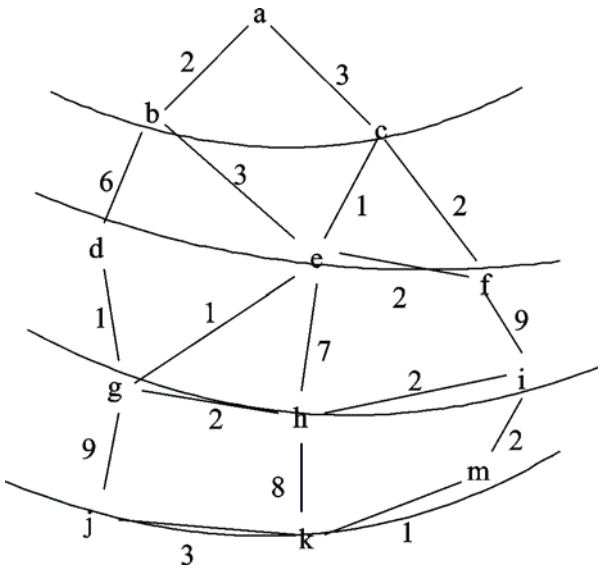


Abb. 15-2 Wellenfronten der Breitensuche

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.

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]].

```
breitensuche(Start, Ziel, Loesung):-
  breitensuche([Start], [], Ziel, Loesung).

breitensuche(Pfad, _, Ziel, Loesung):-
  Pfad = [Ziel|_],
  Loesung = Pfad.

breitensuche(Pfad, Pfade, Ziel, Loesung):-
  Pfad = [Ziel|_], !,
  Pfade = [Pfad|PfadeN],
  breitensuche(Pfad, PfadeN, Ziel, Loesung).

breitensuche(Pfad, Pfade, Ziel, Loesung):-
  Pfad = [KnotenA|_],
  findall([KnotenZ|Pfad],
          (verbunden(KnotenA, KnotenZ),
           not(member(KnotenZ, Pfad))),
          GefundenePfade),
  append(Pfade, GefundenePfade, NeuePfade),
  NeuePfade = [PfadN|RestPfade],
  breitensuche(PfadN, RestPfade,
               Ziel, Loesung).
```

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 wird nicht ein Nachfolger, sondern es werden 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).

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 Erinnerung *findall(+Term, +Ziel, -Liste)* sammelt alle Lösungsterme *Term* eines *Ziels* in einer *Liste*. Damit nur zyklenfreie 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 die Cuts aus den vorangehenden Klauseln entfernen.

15.4 Heuristische Suche

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.

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 die angegebene Lösung.
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.

```
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([KnotenZ, KnotenA|PfadA], KostenZ),
          (verbunden(KnotenA, KnotenZ),
           not(member(KnotenZ, PfadA)),
           bewerte_heuristisch(KnotenA, KostenA,
                               KnotenZ, KostenZ)),
          GefundenePfade),
  sortiere_liste_ein(GefundenePfade, Pfade, NeuePfade),
  NeuePfade = [PfadN|RestPfade],
  heuristischesuche(PfadN, RestPfade, Ziel, Loesung).
```

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.
```

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:

```
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.

15.5 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.

Das Beispiel des Hüpf-Schiebe-Puzzle zeigt, dass man Problemlösen als Pfadsuche im Zustandsraumgraphen auffassen kann. Eine Spielsituation stellt einen Knoten im Zustandsraumgraphen dar. Ein Zug führt von einer Spielsituation zu einer anderen, stellt also eine gerichtete Kante zwischen zwei Knoten im Zustandsraumgraphen dar. Eine Lösung besteht aus einer Folge von Zügen, welche schrittweise die Anfangsposition in die Zielposition transformiert. Im Zustandsraumgraphen 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.

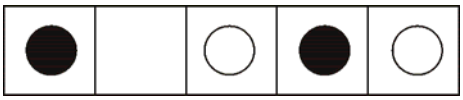


Abb. 15-3 Ein Anfangszustand im Hüpf-Schiebe-Puzzle

Es gibt zwei Hüpf- und zwei Schiebezüge. Bei einem Hüpfzug hüpft 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 15-3 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. 15-4 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.

15.6 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).
```

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).

zug(ZustandA, ZustandN):-
  % l nach rechts hüpfen
  append(X, [l, K1, K2|R], ZustandA),
  append(X, [K2, K1, l|R], ZustandN).

zug(ZustandA, ZustandN):-
  % l nach links schieben
  append(X, [K, l|R], ZustandA),
  append(X, [l, K|R], ZustandN).

zug(ZustandA, ZustandN):-
  % l nach links hüpfen
  append(X, [K1, K2, l|R], ZustandA),
  append(X, [l, K2, K1|R], ZustandN).
```

15.7 Tiefensuche für das Hüpf-Schiebe-Puzzle

Da die Tiefensuche im Zustandsraumgraphen 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 Anfangs- 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:

```
endzustand([w,w,l,s,s]).

zeige(Loesung):-
  writeln('Lösung: '),
  zeige_zuege(Loesung).

zeige_zuege([]).
zeige_zuege([K|R]):-
  zeige_zuege(R),
  writeln(K).

?- tiefensuche([s, l, w, s, w]).
```

```
tiefensuche(Anfangszustand):-
  tiefensuche(Anfangszustand,
              [Anfangszustand]).

tiefensuche(Zustand, Pfad):-
  endzustand(Zustand),
  zeige(Pfad).

tiefensuche(ZustandA, Pfad):-
  zug(ZustandA, ZustandN),
  not(member(ZustandN, Pfad)),
```

15.8 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* übergeben.

```
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],
```

15.9 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.

```
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.

Für die heuristischen 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 $h(n) <= h^*(n)$ 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.

```
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.
```

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*.

Die Berechnung erfolgt mit der Kopf-Rest-Methode. Sind die Köpfe der Positionslisten gleich, so ist der Beitrag zu h(n) gleich 0, anderenfalls 1.

15.10 Bewertung der Suchverfahren für das Hüpf-Schiebe-Puzzle

Das Hüpf-Schiebe-Puzzle ist vergleichsweise einfach. Es sind insgesamt nur 30 verschiedene Zustände möglich. Der Zustandsraumgraph kann daher zum Vergleich der drei Suchverfahren komplett untersucht werden. Die folgende Tabelle enthält Ergebnisse einer Untersuchung:

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
swswl	11	38	0,16	5	58	0,22	17	0,12	10	0,12
swslw	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
wsswl	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

Tabelle 15-1 Untersuchungsergebnisse zur Bewertung der Suchverfahren beim Hüpf-Schiebe-Puzzle

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.

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.

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 $h(n) = 0 <= h^*(n)$ 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.

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.

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.

15.11 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

8	1	3
2		4
7	6	5

Zielzustand

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 wohlbekannten Listen. Die obigen Zustände werden durch die beiden folgenden Listen dargestellt.

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(echts). 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 15-2
Zusammenstellung der Züge im 8er-Puzzle

15.12 Beschränkte Tiefensuche für das 8er-Puzzle

Reine Tiefensuche hat für das 8er-Puzzle keinen Sinn, da der Zustandsraumgraph 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.

```
tiefensuche:-
  write('maximale Tiefe: '),
  read(Tiefe),
  writeln(Tiefe),
  write('Problemnummer: '),
  read(Nummer),
  writeln(Nummer),
  problem(Nummer, Startzustand),
  tiefensuche(Tiefe,[Startzustand],[]).

tiefensuche(_, Pfad, Zuege):-
  Pfad = [Zustand|_],
  endzustand(Zustand),
  write('Lösung: '),
  writeln(Zuege), !.
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:

```
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.

15.13 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),
  writeln(Nummer),
  problem(Nummer, Startzustand),
  breitensuche(
    pfad([Startzustand], []), []).

breitensuche(Pfad, _):-
  Pfad = pfad([Zustand|_], Zuege),
  endzustand(Zustand),
  write('Lösung: '),
  writeln(Zuege), !.
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).
```

15.14 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

```
heuristischesuche:-
  write('Problemnummer: '),
  read(Nummer),
  writeln(Nummer),
  problem(Nummer, Startzustand),
  heuristischesuche(
    pfad([Startzustand], [], 0),  []).

heuristischesuche(Pfad, _):-
  Pfad = pfad([Zustand|_], Zuege, _),
  endzustand(Zustand),
  write('Lösung: '),
  writeln(Zuege), !.
```

```
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]).
```

15.15 Bewertung der Suchverfahren für das 8er-Puzzle

An dieser Stelle möchte ich einige Erfahrungen unter 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, wie viele 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 SWI-Prolog liegen die Suchzeiten für Lösungswege der Länge 20 im Bereich bis einige Sekunden.

Da die Breitensuche alle Pfade speichern muss, gibt es bei Lösungswegen ab der Länge 13 Speicherprobleme. Solche Probleme sind mit Breitensuche nicht mehr lösbar.

Die heuristische Suche löst Probleme bis zu Lösungswegen der Länge 18. Für schwierigere Probleme braucht man eine bessere Heuristik!

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.

15.16 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 ersten 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* aus Kapitel 15-4.

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.*

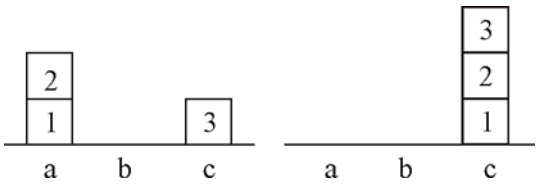
7. Schwierig:

a) 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.

b) Experimentieren Sie mit dieser Heuristik-Funktion. Welche Probleme lassen sich damit lösen?

c) 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.



Startzustand Zielzustand

a) Modellieren Sie die Zustände der Blockwelt durch drei Listen und definieren Sie die sechs möglichen Züge auf diesen Zuständen.

b) Wenden Sie die Tiefen- und Breitensuche auf die Blockwelt an.

c) Konzipieren und implementieren Sie eine Bewertungsfunktion für die heuristische Suche.

16 Terme

16.1 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 Punkt als Funktor (vgl. 5.2) und zwei Argumenten.

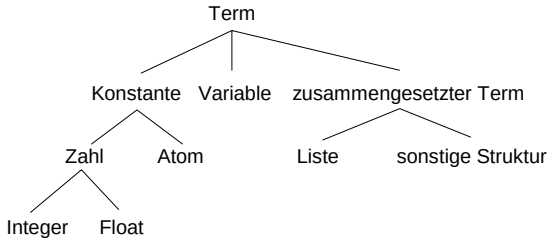


Abb. 16-1 Hierarchie der Prolog-Terme

Zur Termanalyse stellt Prolog folgende Prädikate zur Verfügung:

atomic(X)	prüft, ob X eine Konstante ist.
number(X)	prüft, ob X eine Zahl ist
integer(X)	prüft, ob X eine ganze Zahl ist
float(X)	prüft, ob X eine Dezimalzahl ist
atom(X)	prüft, ob X ein Atom ist
var (X)	prüft, ob X eine freie Variable ist
compound(X)	prüft, ob X ein zusammengesetzter Term ist
is_list(X)	prüft, ob X eine Liste ist

Da in Prolog fast alle Daten zusammengesetzte Terme sind, werden wir uns jetzt mit diesen Termen näher beschäftigen.

16.2 Zusammen gesetzte 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:

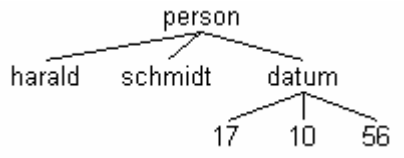


Abb. 16-2 Baumdarstellung des zusammengesetzten Terms
`person(harald,schmidt,datum(17,10,56))`

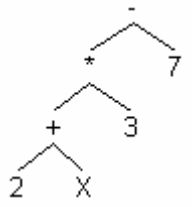


Abb. 16-3 Baumdarstellung des arithmetischen Terms `(2 + X) * 3 - 7`

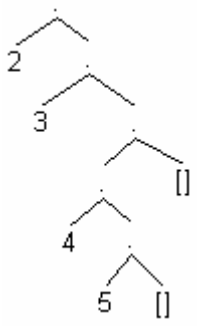


Abb. 16-4 Baumdarstellung der Liste
`[2, 3, [4, 5]].`

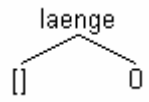


Abb. 16-5
Baumdarstellung des Fakts `laenge([], 0).`

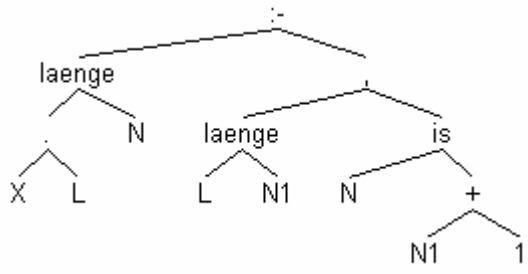


Abb. 16-6 Baumdarstellung der Regel
`laenge([X|L],N):- laenge(L,N1), N` **`is`** `N1+1.`

Prolog sieht keine direkte Möglichkeit vor, Terme als Bäume graphisch auszugeben. Das didaktische Werkzeug *zeichne_term* kann dies (vgl. 5.1). 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.

Das *display*-Prädikat liefert für unsere fünf Beispiele:

```
?- display(person(harald,
    schmidt, datum(17,10,56))).
person(harald,schmidt, datum(17,10,56))

?- display((2 + X) * 3 - 7).
  -(*(+(2,X_1),3),7)

?- 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))))
```

16.3 Term-Vergleichsoperatoren

SWI-Prolog bietet 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 für die Standardordnung irrelevant sind.

Vergleich	Schreibweise
kleiner	Term1 @< Term2
kleiner gleich	Term1 @=< Term2
größer	Term1 @> Term2
größer gleich	Term1 @>= Term2
gleich	Term1 == Term2
ungleich	Term1 \== Term2

Tabelle 16-1 Vergleichsoperatoren für Terme

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.*

Das Systemprädikat *sort(+Liste, -Sortierte-Liste)* sortiert Listen beliebiger Terme nach obiger Standardordnung.

16.4 Strukturoperatoren

Prolog bietet zur Strukturuntersuchung die beiden Prädikate

`functor(?Term, ?Name, ?Stelligkeit)` und
`arg(+ArgNr, +Term, -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)`

Das *functor*-Prädikat greift auf den Funktor zu, das *arg*-Prädikat auf die Argumente. Die Schreibweise *arg(+ArgNr, +Term, -Arg)* zeigt an, dass wegen der +-Zeichen die beiden ersten Argumente instanziert sein müssen und man das Ergebnis, das Argument mit der Nummer ArgNr, im dritten Argument erhält.

```
?- arg(2, liebt(susi, peter), X).
```
Antwort: `X = peter`
```
?- arg(2, 3+4*5, X).
```
Antwort: `X = 4*5`

Der allgemeine Aufbau eines Terms ist

```
funktor(Arg1, Arg2,..., ArgN)
```

wobei die Argumente selbst wieder Terme sein können. Ein Term kann mit dem *univ*-Operator in eine Liste umgewandelt werden, wobei der

Funktor zum Kopf der Liste wird und die Argumente den Rest der Liste bilden:

```
[funktor, Arg1, Arg2,..., ArgN]
```

Graphisch stellt sich die Situation wie folgt dar:

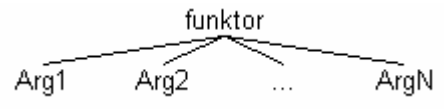


Abb. 16-7 Umwandlung einer Struktur in eine

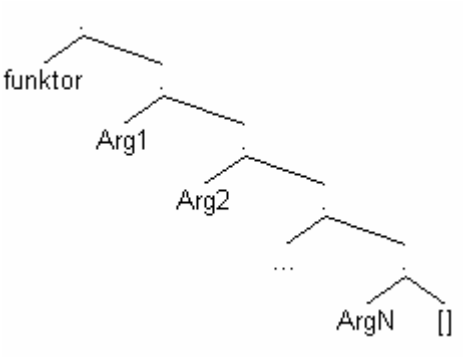


Abb. 16-8 Liste mit Hilfe des univ-Operators

Der *univ*-Operator macht aus einem allgemeinen Baum einen entarteten Binärbaum 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 noch mal unsere Beispiele:

```
?- (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]`

16.5 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.

16.5.1 Zählen von Variablen in Termen

Es soll ein Prädikat *countvar(+Term, -Anzahl)* entwickelt werden, das zählt, wie viele Variablen in einem Term vorkommen.

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).
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):-
  % im Kopf steht der Funktor
  % der keine Variable sein kann
  Term =.. [_Kopf|Liste],
  countvarliste(Liste, Anzahl).
```

3. Die Liste wird nach der Kopf-Rest-Methode behandelt.

```
countvarliste([K|R], N):-
  % Kopf ist eine Konstante
  atomic(K),
  countvarliste(R, N).
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
```

16.5.2 Partielle Auswertung arithmetischer Ausdrücke

Der *is*-Operator kann nur angewendet werden, wenn alle Variablen eines arithmetischen Ausdrucks instanziert 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.

```
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 zusammen gesetzt. 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.

```
auswerten(Term1, Term2):-
  % Term in Liste zerlegen
  Term1 =.. [Funktor|Argumente],
  % Argumente auswerten
  auswerten_liste(Argumente, Werte),
  % Term zusammensetzen
  Term3 =.. [Funktor|Werte],
  vereinfachen(Term3, Term2).
```

3. Das Auswerten einer Liste nehmen wir durch Auswertung aller Argumente nach der Kopf-Rest-Methode für Listen vor.

```
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 beide Operatoren Zahlen sind und der Funktor +, -, *, mod oder Minus ist.

```
vereinfachen(T1 + T2, Wert):-
  integer(T1), integer(T2),
  Wert is T1 + T2.
vereinfachen(T1 - T2, Wert):-
  integer(T1), integer(T2),
  Wert is T1 - T2.
vereinfachen(T1 * T2, Wert):-
  integer(T1), integer(T2),
  Wert is T1 * T2.
vereinfachen(T1 / T2, Wert):-
  integer(T1), integer(T2),
  Wert is T1 / T2.
vereinfachen(T1 mod T2, Wert):-
  integer(T1), integer(T2),
  Wert is T1 mod T2.
vereinfachen(-T, Wert):-
  integer(T),
  Wert is -T.
vereinfachen(T, T).
```

16.6 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-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 imperativen Programmiersprachen 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 problemlos algorithmisch in den Griff bekommt.

Zu einer Wurzel können n verschiedene Nachfolger-Knoten gehören. Aus 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 Beispiel zur partiellen Auswertung arithmetischer Ausdrücke kommt die Wurzel-Knoten-Methode in der Ausprägung KW zum Tra-

gen. 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.

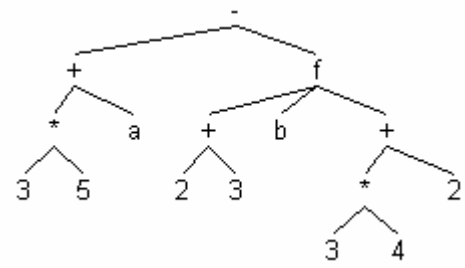


Abb. 16-9 Rechenterm vor der Vereinfachung

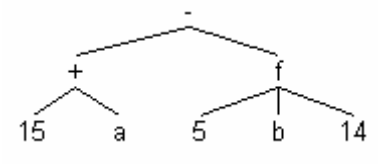


Abb. 16-10 mit Hilfe des univ-Operators vereinfacht

Beim Beispiel zum Zählen von Variablen kommt die WK-Methode zum Einsatz. Da die Wurzel keine Variable sein kann, müssen nur die Variablen in der Knotenliste gezählt werden.

16.7 Aufgaben

1. Wie antwortet Prolog?

```
a) functor(summe(2, 3, 5), X, Y).
b) functor(schueler(paul, schmidt, da-
   tum(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. Implementieren Sie mit Hilfe des *functor*-Prädikats das *compound*-Prädikat (vgl. 16.1).

3. Analysieren Sie

```
berechne:-
  write('X= '), read(X),
  writeln(X),
  write('Operator: '),
  read(Operator), writeln(Operator),
  write('Y= '), read(Y),
  write(Y), nl,
  functor(Term, Operator, 2),
  arg(1, Term, X),
  arg(2, Term, Y),
  Ergebnis is Term,
  write('Ergebnis= '),
  writeln(Ergebnis).
```

4. Definieren Sie ein Prädikat *isin(+Element, +Term)* derart, dass erfüllt ist, wenn *Element* in *Term* vorkommt.

5. Definieren Sie ein Prädikat *grund(+Term)* derart, dass es wahr ist, wenn Term keine uninstanzierten 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, da-
tum(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.

17 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.

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:

Die Raser Jagd macht die auf Polizei.

Erst wenn die Wörter in der richtigen Reihenfolge aufeinander folgen, ergibt sich ein sinnvoller Satz:

Die Polizei macht Jagd auf die Raser.

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 Wörtern 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.

Wir betrachten nun vier Beispiele für Grammatiken, anhand derer dann eine Definition des Begriffs Grammatik erfolgt.

17.1 Grammatiken

17.1.1 Grammatik einfacher deutscher Sätze

Beispiel 1 mit der folgenden Grammatik beschreibt einfache deutsche Sätze:

(1) Satz → Subjekt Prädikat Objekt
(2) Subjekt → Eigenname | 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) Eigenname → 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 Eigenname oder eine Substantivgruppe, welche nach der dritten Regel aus einem Artikel und einem Substantiv besteht.

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.

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:

Satz → (nach Regel 1)
Subjekt Prädikat Objekt → (nach Regel 2)
Eigenname 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

17.1.2 Arithmetische Ausdrücke

Die folgende Grammatik beschreibt die Struktur arithmetischer Ausdrücke, bei denen die vier Grundrechenarten und Klammern vorkommen dürfen. Beispiel 2:

(1) Ausdruck → Term | Term + Ausdruck
| Term - Ausdruck
(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
Term → nach Regel 2
Faktor * Term → nach Regel 3
a * Term → nach Regel 2
a * Faktor → nach Regel 3
a * (Ausdruck) → nach Regel 1
a * (Term - Ausdruck) → nach Regel 2
a * (Faktor - Ausdruck) → nach Regel 3
a * (b - Ausdruck) → nach Regel 1
a * (b - Term) → nach Regel 2
a * (b - 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 *Nichterminale* oder *Variablen* der Grammatik.

17.1.3 Palindrome

Als Beispiel 3 betrachten wir Palindrome aus den Buchstaben a und b. Ein Palindrom ergibt vorwärts wie rückwärts gelesen das gleiche Wort.

(1) S → a | b
(2) S → a S a | b S b

Die Buchstaben a und b sind die Terminale dieser Grammatik. Das Startsymbol S ist zugleich das einzige Nichtterminal. Aus S kann das Wort ababa abgeleitet werden:

S → aSa → abSba → ababa

17.1.4 0-1-Wörter

Als viertes Beispiel betrachten wir eine Grammatik, deren Sprache aus allen 0-1-Wörtern mit gleich vielen Nullen und Einsen besteht.

(1) S → 0 B | 1 A
(2) A → 0 | 0 S | 1 A A
(3) B → 1 | 1 S | 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:

S → 0B → 01S → 010B → 0101S → 01011A → 010110

17.2 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 (Nichtterminalen)*. *Variablen* sind grammatische Konstrukte. Jedes Konstrukt lässt sich gemäß einer Grammatikregel in Terminale oder Variable zerlegen.
- Einem *Startsymbol*. Das Startsymbol ist eine ausgewählte Variable, gewissermaßen das allgemeinste Konstrukt der Sprache.
- Einer Menge von Regeln, oft *Produktionen* genannt. Eine Regel besteht aus einem Kopf, gefolgt von einem Pfeil und einem Rumpf (Kopf → Rumpf). Die Zeichenfolge im Kopf kann durch die Zeichenfolge im Rumpf ersetzt werden.

Die Definition ist so allgemein, dass sie auch das folgende Beispiel einer Grammatik umfasst:

17.2.1 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 Wörter aus Terminalen und Variablen vor. Beispiel 5:

(1) S → a S B C | a B C
(2) C B → B C
(3) a B → a b
(4) b B → b b
(5) b C → b c
(6) c C → c c

Aus dem Startsymbol S können wir aaabbbccc ableiten:

S → aSBC → aaSBCBC → aaaBCBCBC → aaaBBCCBC → aaaBBCBCC → aaaBBBCCC → aaabBBCCC → aaabbBCCC → aaabbbCCC → aaabbbcCC → aaabbbccC → aaabbbccc

17.3 Chomsky-Hierarchie der Grammatiken

Das letzte Beispiel 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.

- Jede Grammatik ist automatisch vom Typ 0. Bei Typ 0 sind den Produktionen keinerlei Einschränkungen auferlegt. Man spricht auch von allgemeinen Phasenstrukturgrammatiken.
- 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.
- 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.
- 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

Grammatik	Typ	Produktionsformen
allgemein	0	$\alpha \rightarrow \beta,\ \alpha \neq \varepsilon$
kontextsensitiv	1	$\alpha A \beta \rightarrow \alpha \gamma \beta, \gamma \neq \varepsilon$
kontextfrei	2	$A \rightarrow \alpha$
regulär	3	A → a | a B

Tabelle 17-1 Produktionsformen der Chomsky-Hierarchie

Die Beispiele 1 bis 4 zeigen kontextfreie Grammatiken. Die Grammatik in Beispiel 5 kann in eine kontextsensitive Grammatik umgebaut werden (vgl. Aufgabe 6).

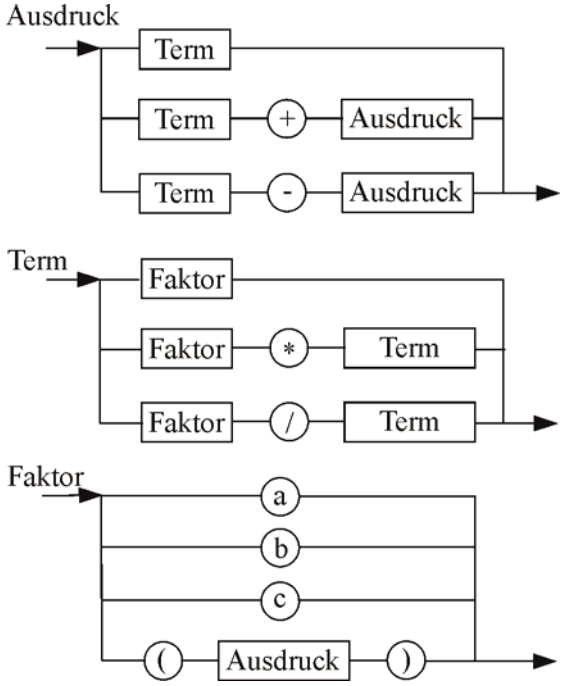
Eine reguläre Grammatik für Bezeichner aus den beiden Terminalen a und 1 ist gegeben durch:

(1) S → a | aA
(2) A → a | 1 | aA | 1A

17.4 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 → ersetzt ist.

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 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.

Die folgende Tabelle stellt für die grundlegenden Kontrollstrukturen den Grammatikregeln die entsprechenden Syntaxdiagramme gegenüber.

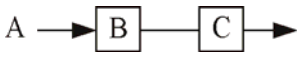
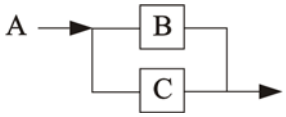
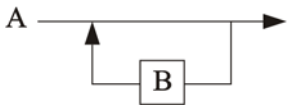
Kontroll-struktur	Grammatik-regel	Syntaxdiagramm
Sequenz	A → B C	
Selektion	A → B | C	
Rekursion	A → B A	
Iteration	A → { B }	

Tabelle 17-2
Gegenüberstellung äquivalenter Kontrollstrukturen, Grammatikregeln und Syntaxdiagramme

Schaut man sich Syntaxdiagramme für Programmiersprachen 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 bzw. Java der While-Schleife.

Da sich in Prolog rekursive Strukturen besser als iterative Strukturen verarbeiten lassen, erzählt folgende Bildergeschichte, wie sich eine Iteration in eine Rekursion umwandeln lässt:

1. Iteration
2. Die Idee!
3. Aufbrechen
4. Umorientieren
5. Einbauen
6. Rekursion

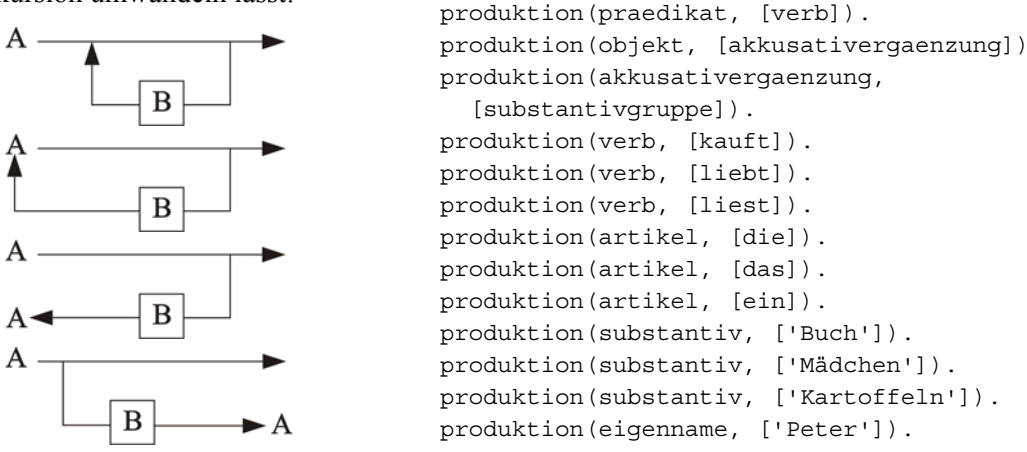


Abb. 17-1 Umwandlung eines iterativen in ein rekursives Syntaxdiagramm

17.5 Modellierung von Grammatiken in Prolog

Zur maschinellen Verarbeitung von Grammatiken bilden wir deren Komponenten in Prolog ab. Jede Komponente modellieren wir als ein Prädikat.

```
terminal(kauft).
terminal(liebt).
terminal(liest).
terminal(X):-
  member(X, [die, das, ein]).
terminal(X):-
  member(X, ['Buch', 'Mädchen',
             'Kartoffeln']).
terminal('Peter').

variable(X):-
  member(X, [satz, subjekt, praedikat,
    objekt, eigenname, substantivgruppe,
    substantiv, akkusativergaenzung,
    artikel, verb]).

startsymbol(satz).

produktion(satz,
  [subjekt, praedikat, objekt]).
produktion(subjekt, [eigenname]).
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(eigenname, ['Peter']).
```

Die hier benutzten sechs *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 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 muss man den Regelkopf auch als Liste modellieren.

Die Modellierung lässt sich problemlos auf die anderen Beispiele übertragen. Für das vierte Beispiel ergibt sich:

```
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]).
```

17.6 Erzeugte Sprache - Breitensuche in Graphen

Einer Grammatik G lässt sich eine formale Sprache L(G) zuordnen: L(G) ist die Menge aller Wörter aus Terminalen, die sich aus dem Startsymbol durch Anwendung der Produktionen erzeugen lassen.

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*.

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)?

2. Analyse: Lässt sich ein gegebenes Wort aus dem Startsymbol ableiten?

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:

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.

```
einfacheAbleitung(+AlteListe, -NeueListe)
```

Eine Variable aus *AlteListe* wird durch eine Produktion ersetzt. Im Beispiel wurde die Produktion *Artikel* → *das* benutzt. Zur Implementierung des Prädikats gehen wir mit der Kopf-Rest-Methode *AlteListe* soweit durch, bis wir eine Variable gefunden haben.

```
einfacheAbleitung(L1, L2):-
  L1 = [K|R],
  terminal(K),
  einfacheAbleitung(R, L3),
  L2 = [K|L3].
```

Haben wir als Kopfelement von *AlteListe* eine Variable gefunden, so wird für diese eine Produktion gesucht und das Kopfelement durch die Ableitung ersetzt.

```
einfacheAbleitung(L1, L2):-
  L1 = [K|R],
  variable(K),
  produktion(K, L3),
  append(L3, R, L2).
```

Wir ersetzen also immer nur die Variable, die am weitesten links steht, man spricht daher von einer Linksableitung.

Bei kontextfreien Grammatiken ist die Reihenfolge, in der Variablen durch Produktionen ersetzt werden egal, die Linksableitung reicht also aus, bei kontextsensitiven nicht.

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.

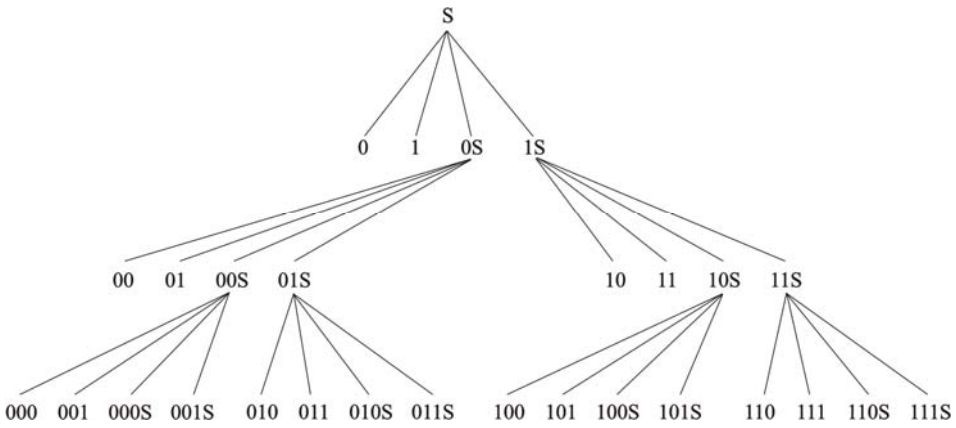


Abb. 17-2 Ableitungsbaum für 0-1-Wörter

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:

```
% linksrekursiv
ableitungL(A, B):-
  einfacheAbleitung(A, B).
ableitungL(A, B):-
  ableitungL(A, C),
  einfacheAbleitung(C, B).

% rechtsrekursiv
ableitungR(A, B):-
  einfacheAbleitung(A, B).
ableitungR(A, B):-
  einfacheAbleitung(A, C),
  ableitungR(C, B).
```

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 | 1 | 0S | 1S gehört der dargestellte Ableitungsbaum von Abbildung 17-2, der alle Ableitungen bis zur dritten Stufe enthält.

- 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,...

Der Vollständigkeit halber sei ergänzt, dass man nicht nur die Ziele in den Regelrümpfen tauschen kann, sondern auch die Klauseln. Damit entstehen zwei weitere logisch äquivalente Implementierungen des ableitung-Prädikats. Beide sind aber wegen Endlosrekursion nicht in der Lage Lösungen zu liefern. Beim vorfahr-Prädikat würde zumindest die rechtsrekursive Variante noch Ergebnisse liefern, denn dort ist der Lösungsraum endlich. Von den vier deklarativ äquivalenten Möglichkeiten, sollte man in der Regel die einfachste benutzen, also erste Klausel ohne Rekursion, zweite Klausel rechtsrekursiv.

Zur Sprache gehören nur Wörter aus Terminalen. Solche Wörter können mittels Kopf-Rest-Methode leicht identifiziert werden.

```
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:

```
erzeugteSprache(Wort):-
  startsymbol(Start),
  ableitung([Start], Wort),
  terminalwort(Wort).
```

17.7 Syntaktische Analyse mit Akzeptoren

Mit den bisher entwickelten Prädikaten können Wörter 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 Wörtern. Die Analyse stellt fest, ob ein gegebenes Wort einer Sprache L(G) angehört. Soll das Wort weiterverarbeitet werden, so muss außerdem die gram-

matische Struktur des Wortes in Form eines Ableitungsbaums bestimmt werden.

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 zugrunde liegenden Grammatik. Die Anwendung einer Produktion erzeugt die Nachfolgeknoten eines inneren Knotens. Für die Grammatik der 0-1-Wörter ist in Abbildung 17-3 der Ableitungsbaum der Ableitung S → 0B → 00BB → 001B → 0011 angegeben.

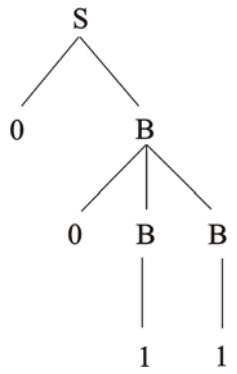


Abb. 17-3 Ableitungsbaum für das Wort 0011

Die Grundstruktur für einen einfachen Generator von Ableitungsbäumen ist im Folgenden dargestellt:

```
ableitung(Ableitungsbaum):-
  startsymbol(Start),
  ableitung(Start, Ableitungsbaum).

% ableitung(+Symbol, -Ableitungsbaum)
ableitung(Terminal, Terminal):-
  terminal(Terminal).
ableitung(Variable, Ableitungsbaum):-
  variable(Variable),
  produktion(Variable, Regelrumpf),
  ableitungListe(Regelrumpf,
                 Ableitungsbaeume),
  Ableitungsbaum =..
    [Variable|Ableitungsbaeume].

% ableitungListe(+RegelrumpfListe,
                 -Ableitungsbaeume)
ableitungListe([K1|R1], [K2|R2]):-
  ableitung(K1, K2),
  ableitungListe(R1, R2).
ableitungListe([], []).
```

Das Prädikat *ableitung/2* erwartet im ersten Argument ein Symbol und gibt im zweiten Argument für dieses Symbol einen Ableitungsbaum aus. Ist das Symbol eine Konstante, dann ist der Ableitungsbaum gleich dieser Konstanten. Handelt es sich bei diesem Symbol um eine Variable, so wird eine Produktion gesucht. Zu jedem Symbol des Produktionsrumpfes wird mit der Kopf-Rest-Methode in *ableitungListe* ein Ableitungsbaum bestimmt. Abschließend werden die Teilergebnisse mit dem univ-Operator zusammengesetzt.

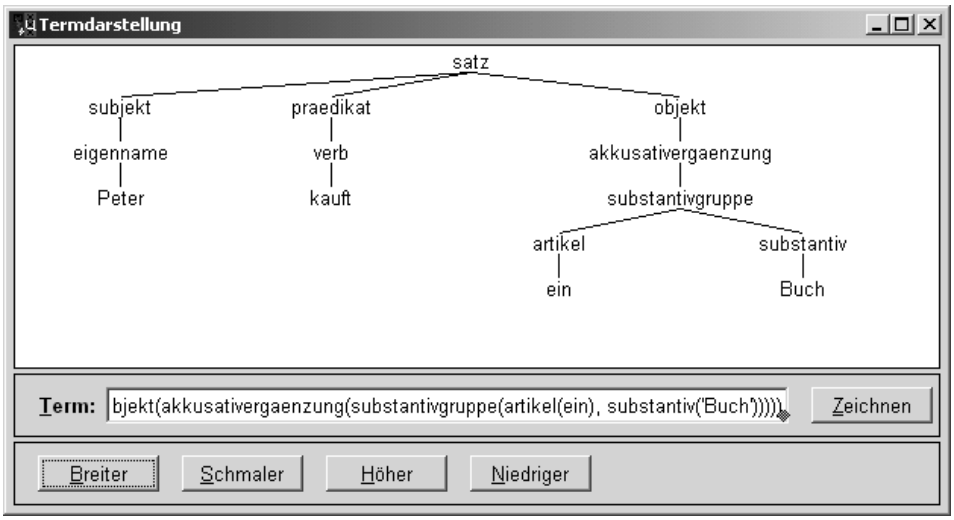


Abb. 17-4 Berechnung eines Ableitungsbaums

Dieser Grundalgorithmus lässt sich so erweitern, dass vorgegebene Sätze analysiert und deren Ableitungsbäume berechnet werden können (siehe *AbleitungsbaumBestimmen.pl*). Abbildung 17-4 zeigt das Ergebnis am einleitenden Beispiel. In einem Compiler übernimmt der *Parser* (dt. = Zerteiler) die Aufgabe der syntaktischen Analyse. Er untersucht, ob das Quellprogramm syntaktisch korrekt ist, also der zugrunde liegenden Grammatik entspricht, meldet gegebenenfalls Fehler und gibt den dazugehörigen Ableitungsbaum aus.

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 19) erkennen kontextfreie Sprachen, haben aber ein exponentielles Zeitverhalten. Für Programmiersprachen verwendet man *eingeschränkte kontextfreie Grammatiken*, welche durch *deterministische Kellerautomaten* in Linearzeit erkannt werden können.

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).
```

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. Dies macht die Angelegenheit technisch schwieriger. Eine Lösung findet man im Programm *AbleitungsbaumBestimmen.pl*.

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 Kapiteln die *Automaten* und *Kellerautomaten* und kommen in Kapitel 22 auf die Lösung des Wortproblems kontextfreier Sprachen mittels nichtdeterministischer Kellerautomaten zurück.

17.8 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 Sprachen

das Wortproblem entscheidet.

6a) Warum ist die Grammatik aus 17.2.1 nicht kontextsensitiv?

b) Wir ersetzen die Produktion CB → BC durch folgende Produktionen:

CB → DB
DB → DC
DC → BC

Begründen Sie, dass die so abgeänderte Grammatik kontextsensitiv ist.

7. Zeichnen Sie Syntaxdiagramme für die Grammatik

a) der einfachen deutschen Sätze

b) der Palindrome

c) der 0-1-Wörter

8. Stellen Sie die behandelten Grammatiken im Formalismus der erweiterten Backus-Naur-Form dar (vgl. Wikipedia).

9. Geben Sie eine Grammatik an für

a) Uhrzeiten der Form 23:14:58

b) Kalenderdaten der Form 2007-07-15

c) internationale Telefonnummer der Form +49 30 1234567

d) Geldbeträge der Form 17,80 €.

10. Entwickeln Sie für die Sprache der Wörter aus den Terminalen a und b, bei denen die Anzahl der a gerade ist, eine Grammatik.

11. Entwickeln Sie eine Grammatik für die durch 3 teilbaren natürlichen Zahlen.

12. Durch folgende Produktionen ist eine Grammatik G gegeben:

Ziffer	→	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
Zahl	→	Ziffer | Ziffer Zahl
Ausdruck	→	Zahl | (Ausdruck) | Ausdruck + Ausdruck | Ausdruck - Ausdruck

a) Geben Sie zwei verschiedene Ableitungsbäume für das Wort 1 – 2 + 3 an. Interpretieren Sie mathematisch die beiden Ableitungsbäume und vergleiche Sie.

b) Ersetzen Sie die Produktionen von Ausdruck durch folgende Produktionen:

Summand	→	Zahl | (Ausdruck)
Ausdruck	→	Ausdruck + Summand | Ausdruck - Summand | Summand

und erläutern Sie die sich daraus ergebenden Konsequenzen.

18 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.

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.

18.1 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.

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.
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:

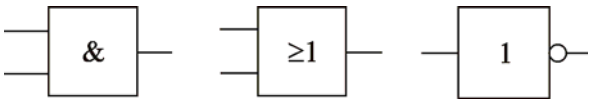
a	b	und
0	0	0
0	1	0
1	0	0
1	1	1

a	b	oder
0	0	0
0	1	1
1	0	1
1	1	1

a	nicht
0	1
1	0

Tabelle 18-1 Schaltwerttabellen der logischen Grundschaltungen

Die Gatter stellt man DIN-gerecht durch diese Symbole dar:



und oder nicht

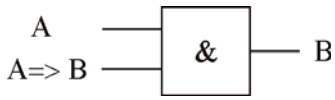
Mit Hilfe der logischen Grundgatter lassen sich typische geistige Fähigkeiten maschinell nachbilden. Dazu betrachten wir drei Beispiele:

18.1.1 Logisches Schließen

Eine häufig benutzte logische Schlussregel ist der sogenannte *Modus ponens,* mit dem aus einer wahren Voraussetzung A und einer Implikation A⇒B eine wahre Konsequenz geschlossen wird. Angenommen die Implikation *Wenn der Schalter betätigt wird, dann brennt das Licht.* (A⇒B) stimmt und die Voraussetzung *Der Schalter wurde betätigt.* (A) ist gegeben, dann können wir schließen, dass *Das Licht brennt*. (B)

$$A \wedge (A \Rightarrow B) \Rightarrow B$$

Mit einer einfachen Und-Schaltung lässt sich der logische Schluss vornehmen:



18.1.2 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 Ü an:

A	B	S	Ü
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Tabelle 18-2 Addition zweier Dualzifffern

Den Übertrag Ü erhält man durch die Konjunktion A∧B, die Summe S durch ein Entweder-Oder in der Form (¬A∧B)v(A∧¬B). Daraus ergibt sich das Schaltnetz:

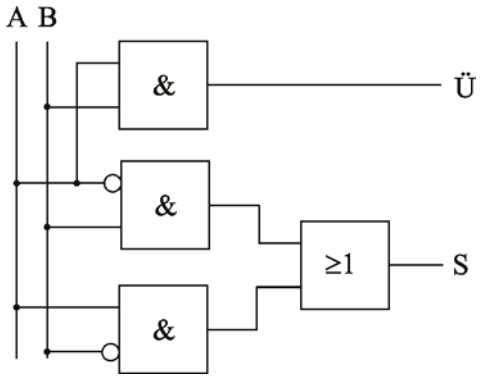


Abb. 18-1 Schaltnetz für einen Halbaddierer

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]. Bei http://www.digitalsimulator.de/ gibt es ein Simulationsprogramm für die Arbeit mit digitalen Schaltungen.

18.1.3 Prüfung binär codierter Dezimalziffern

Die Numerikmodule der ersten Pentium-Prozessoren waren so fehlerhaft, dass beim Dividieren von Real-Zahlen Fehler in der Größenordnung 10^{-5} auftraten. Für finanzmathematische Anwendungen ist so etwas 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.

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

Tabelle 18-3 Codierung von Ziffern durch Tetraden

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:

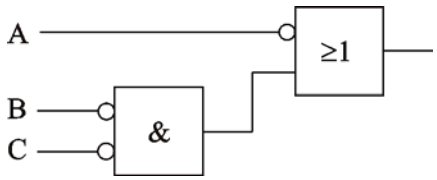


Abb. 18-2 Prüfschaltung für gültige Tetraden

18.2 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 Problemen benötigt man mehrere Schritte, um eine Lösung zu erhalten. Dazu muss man 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:

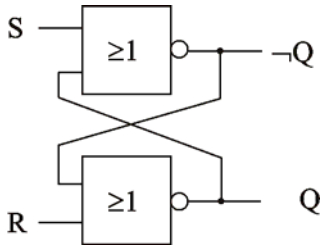


Abb. 18-3 Gatterschaltung eines RS-Flipflops

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.

18.3 Konzeption des endlichen Automaten

Wir betrachten einige konkrete Automaten, um dann im Sinne einer Abstraktion eine formale Definition für Automaten zu motivieren.

18.3.1 Getränkeautomat

Ein Getränkeautomat kann Cola und Limonade ausgeben. Beide Getränke kosten 1,50 €. Es können Eurostücke und 50 Cent-Münzen eingeworfen werden. Wird der Betrag von 1,50 € überschritten, so fällt die Münze ins Geldausgabefach. Bei korrektem Geldeinwurf kann zwischen Cola und Limonade gewählt werden. Zu jedem Zeitpunkt führt das Drücken der Korrekturtaste zur Geldrückgabe. Der Automat soll immer in betriebsbereitem Zustand sein, also alle Getränke vorrätig haben.

Eingaben : 50 Ct (F), 1 € (H), Colataste (C), Limotaste (L), Korrekturtaste (K)
Ausgaben: 50 Ct (F), 1 € (H), 1,50 € (G), Cola (C), Limo (L), nichts (-)

Damit der Automat weiß, wie er auf eine Eingabe reagieren soll, muss er sich merken, wie viel Geld schon eingeworfen wurde. Jeder eingeworfene Geldbetrag steht für einen *Zustand* des Automaten, welcher seine weitere Reaktion bestimmt. Vier Zustände müssen wir unterscheiden:

0 Ct, 50 Ct, 100 Ct und 150 Ct

Das Verhalten des Automaten können wir sehr übersichtlich durch ein Zustandsdiagramm beschreiben, dessen Knoten die Zustände sind. Der Knoten, der dem Anfangszustand entspricht, wird durch einen hineingehenden Pfeil besonders markiert, alle Endknoten werden durch doppelte Kreise gekennzeichnet. Die Kanten sind gerichtet, geben also an, von wo nach wo ein Zustandsübergang stattfinden kann. Zudem sind die Kanten in der Form *E/A* beschriftet. *E* gibt an, bei welcher Eingabe der Zustandsübergang stattfindet, *A* gibt an, welche Ausgabe der Automat bei diesem Zustandsübergang macht.

Unser Getränkeautomat wird durch das Zustandsdiagramm in Abb 18-4 beschrieben (nach [Bur1]). Vereinfachend wurden Kanten, die dieselben Zustandsknoten verbinden, zusammengefasst und mehrfach beschriftet.

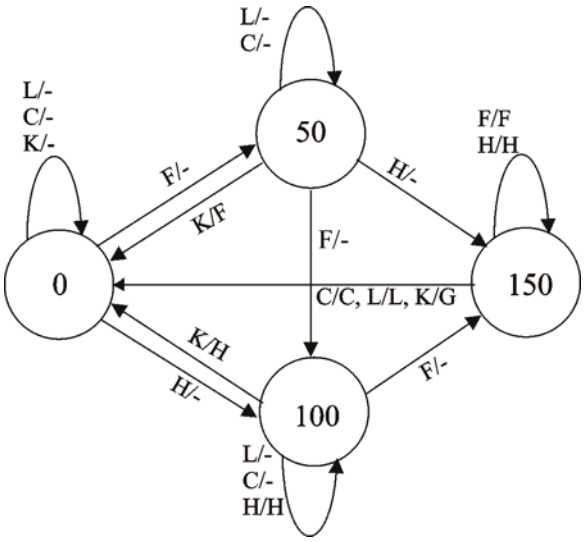


Abb. 18-4 Zustandsdiagramm eines Getränkeautomaten

18.3.2 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. Im Zustandsdiagramm von Abbildung 18-5 legen wir das Verhalten des Aufzugs fest.

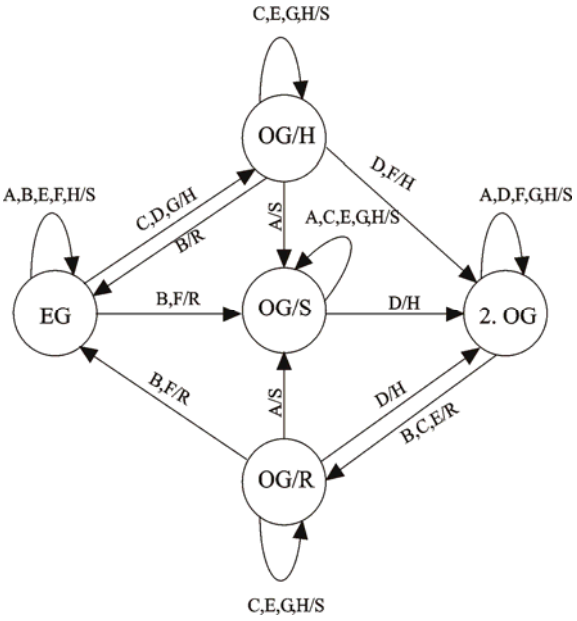


Abb. 18-5 Zustandsdiagramm einer Aufzugssteuerung

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.

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*.

18.3.3 Akzeptor für Bezeichner

In vielen Programmiersprachen gilt die Regel, dass Bezeichner mit einem Buchstaben beginnen, auf den beliebig viele Buchstaben und Ziffern folgen können. Im Syntaxdiagramm stellt man diesen Sachverhalt wie folgt dar:

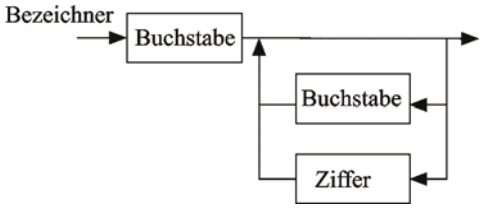


Abb. 18-6 Syntaxdiagramm für Bezeichner

Ein Automat, der Bezeichner erkennt, erhält als Eingabe Zeichen. Im Anfangszustand *neutral* hat er noch kein Zeichen gelesen. Der Endzustand *korrekt* steht für einen erkannten Bezeichner, der Fehlerzustand *falsch* signalisiert einen falsch gebildeten Bezeichner. Weitere Zustände sind nicht erforderlich, wie das Zustandsdiagramm zeigt.

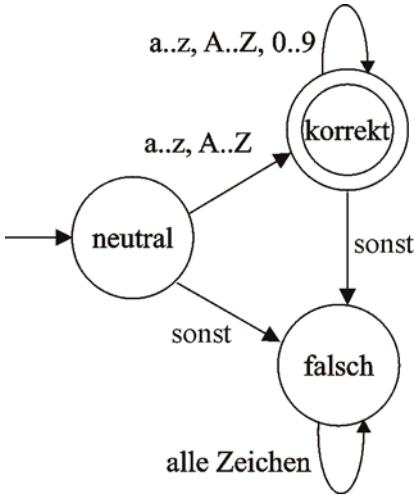


Abb. 18-7 Zustandsdiagramm eines Akzeptors für Bezeichner

18.3.4 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.

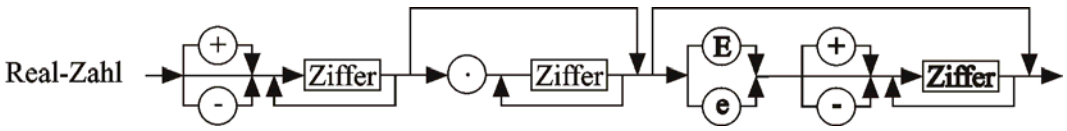


Abb. 18-8 Syntaxdiagramm für Real-Zahlen

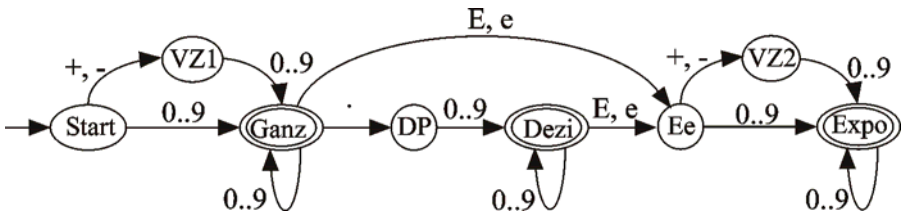


Abb. 18-9 Zustandsdiagramm eines Akzeptors für Real-Zahlen

Die Umsetzung in einen erkennenden Automaten ist hier schon etwas schwieriger. Damit die Übersicht erhalten bleibt, lassen wir generell im Zustandsdiagramm Übergänge in den Fehlerzustand weg. Weiterhin benutzen wir die Abkürzungen VZ für Vorzeichen, DP für Dezimalpunkt und Ee für die Einleitung des Exponenten.

Das Zustandsdiagramm 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.

18.4 Definition des endlichen Automaten

Nachdem wir anhand von vier Beispielen Automaten kennen gelernt 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.

Ein Spezialfall sind die erkennenden endlichen Automaten, die so genannten *Akzeptoren*. Bei diesen entfällt das Ausgabealphabet und die Übergangsfunktion gibt nur den jeweiligen Folgezustand an. Ein Akzeptor akzeptiert alle Eingabewörter, die den Automaten vom Anfangszustand in einen Endzustand überführen.

18.5 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).
```

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.

```
% 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):- !.
```

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. Zur Vereinfachung der Eingabe setzen wir das Systemprädikat *atom_chars/2* ein, das ein eingegebenes Atom in eine Liste der Zeichen zerlegt

```
automat(Eingabe):-
  anfangszustand(Zustand),
  atom_chars(Eingabe, Liste),
  automat(Zustand, Liste).
automat(Zustand, [Eingabe|Rest]):-
  uebergang(Zustand, Eingabe, Ausgabe,
            NeuerZustand),
  write(Zustand), tab(2),
  write(Eingabe), write(' -> '),
  write(Ausgabe), tab(2),
  writeln(NeuerZustand),
  automat(NeuerZustand, Rest).
automat(Zustand, []):-
  endzustand(Zustand).
```

Der Automat kann dann beispielsweise so benutzt werden:

```
?- automat(ffkfhflk).
```

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.

18.6 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 Argument *Ausgabe* weg und das Prädikat *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):-
  is_alpha(Eingabe), !.
uebergang(korrekt, Eingabe, korrekt):-
  is_alnum(Eingabe), !.
uebergang(_Zustand, _Eingabe, falsch).
```

Mit Hilfe der Systemprädikate *is_alpha* und *is_alnum* lässt sich feststellen, ob ein Buchstabe (groß oder klein) bzw. ein alphanumerisches Zeichen (Buchstabe oder Ziffer) vorliegt.

```
% automat(+Eingabe)
automat(Eingabe):-
  anfangszustand(Zustand),
  atom_chars(Eingabe, Liste),
  automat(Zustand, Liste).
automat(Zustand, [Eingabe|Rest]):-
  uebergang(Zustand, Eingabe,
            NeuerZustand),
  write(Zustand), tab(2),
  write(Eingabe), write('  ->  '),
  writeln(NeuerZustand),
  automat(NeuerZustand, Rest).
automat(Zustand, []):-
  endzustand(Zustand).

test:-
  automat('A2dX3 Y2nd').
```

Unser viertes Beispiel, der Akzeptor für Real-Zahlen, kann wie folgt modelliert werden:

```
zustand(X):-
  member(X, [start, vz1, ganz, dp,
        dezi, ee, vz2, expo, fehler]).
anfangszustand(start).
endzustand(X):-
  member(X, [ganz, dezi, expo]).

% uebergang(+Zustand, +Eingabe,
            -FolgeZustand)
uebergang(start, Eingabe, vz1):-
  vorzeichen(Eingabe).
uebergang(vz1, Eingabe, ganz):-
  is_digit(Eingabe).
uebergang(start, Eingabe, ganz):-
  is_digit (Eingabe).
uebergang(ganz, Eingabe, ganz):-
  is_digit (Eingabe).
uebergang(ganz, '.', dp).
uebergang(dp, Eingabe, dezi):-
  is_digit (Eingabe).
uebergang(dezi, Eingabe, dezi):-
  is_digit (Eingabe).
uebergang(dezi, Eingabe, ee):-
  exponent(Eingabe).
uebergang(ee, Eingabe, vz2):-
  vorzeichen(Eingabe).
uebergang(vz2, Eingabe, expo):-
  is_digit (Eingabe).
uebergang(ee, Eingabe, expo):-
  is_digit (Eingabe).
uebergang(expo, Eingabe,expo):-
  is_digit (Eingabe).

% Klassifikation
vorzeichen('+').
vorzeichen('-').
exponent('e').
exponent('E').

test:-
  automat('+275.443E-13').
```

18.7 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 dritten Beispiel ist L(A) die Menge aller Bezeichner, im vierten Beispiel die Menge aller Real-Zahlen.

Die Menge der Real-Zahlen entspricht nicht der Menge 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... aufzählen.

Die Bestimmung der von einem Automaten erzeugte Sprache kann als graphentheoretisches Problem gedeutet werden: Gesucht sind alle Wege die im Zustandsdiagramm 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 Zustandsdiagramm.

Nun ist die Bestimmung aller Pfade vom Anfangszustand zum Endzustand keineswegs trivial, denn typischerweise weisen Zustandsdiagramme 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.

Bei der bislang diskutierten Frage, ob ein bestimmtes Wort von einem Automaten erkannt wird oder nicht, hatten wir auch Zustandsdiagramme 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.

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. 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: 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:

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.

```
%  erzeugte Sprache
erzeugteSprache(Wort):-
  anfangszustand(ZustandA),
  mehrfachuebergang(ZustandA, Liste, ZustandZ),
  endzustand(ZustandZ),
  atom_chars(Wort, Liste).

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).
```

18.8 Nichtdeterministischer endlicher Automat

Eine 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 Zustandsdiagramm konstruieren, wobei die Bezeichnung x\p vereinfachend für alle Zeichen außer p steht.

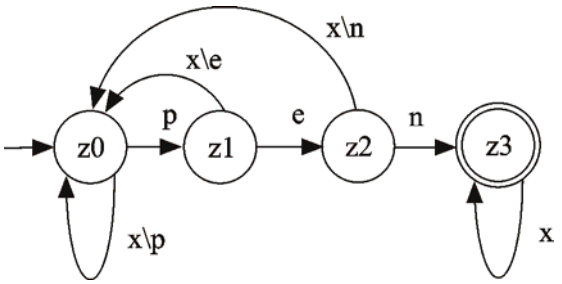


Abb. 18-10 fehlerhafter Akzeptor für die Zeichenkette *pen*

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. 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.

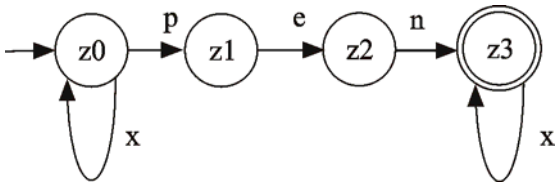


Abb. 18-11 Beispiel eines nichtdeterministischen endlichen Automaten

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 automatisch 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:

```
zustand(X) :-
  member(X, [z0, z1, z2, z3]).
anfangszustand(z0).
endzustand(z3).

% uebergang(+Zustand, +Eingabe,
            -FolgeZustand)
uebergang(z0, Ein, z0):-
  is_alpha(Ein).
uebergang(z0, 'p', z1).
uebergang(z1, 'e', z2).
uebergang(z2, 'n', z3).
uebergang(z3, Ein, z3):-
  is_alpha(Ein).
```

```
test:-
  akzeptiere('Pappenstiel').
```

Jeden nichtdeterministischen Automaten kann man durch das folgende Verfahren in einen deterministischen Automaten umwandeln. Grundsätzlich sind also nichtdeterministischen Automaten nicht leistungsfähiger als deterministische Automaten.

Wir betrachten Mengen von Zuständen und erstellen für den nichtdeterministischen Automaten aus Abbildung 18-11 folgende Übergangstafel:

	p	e	n	s(onst)
z_0	$z_0\, z_1$	z_0	z_0	z_0
$z_0\, z_1$	$z_0\, z_1$	$z_0\, z_2$	z_0	z_0
$z_0\, z_2$	$z_0\, z_1$	z_0	$z_0\, z_1$	z_0
$z_0\, z_3$	$z_0\, z_1\, z_3$	$z_0\, z_3$	$z_0\, z_3$	$z_0\, z_3$
$z_0\, z_1\, z_3$	$z_0\, z_1\, z_3$	$z_0\, z_2\, z_3$	$z_0\, z_3$	$z_0\, z_3$
$z_0\, z_2\, z_3$	$z_0\, z_1\, z_3$	$z_0\, z_3$	$z_0\, z_3$	$z_0\, z_3$

Befindet sich der Automat im Zustand z_0 und liest ein p, so kann er in z_o bleiben oder in z_1 wechseln, er geht somit in den Zustand $z_0\, z_1$ über. Liest er im Zustand z_0 ein anderes Zeichen, so bleibt er z_0. Befindet sich der Automat im Zustand $z_0\, z_1$ und liest ein p so betrachten wir die beiden Teilanfangszustände z_0 und z_1 getrennt. Vom Zustand z_0 aus kommt man zum Zustand $z_0\, z_1$, vom Zustand z_1 aus geht es nicht weiter. Insgesamt verbleibt man als bei Zeichen p im Zustand $z_0\, z_1$. Liest man im Zustand $z_0\, z_1$ ein e, so ergibt der Teilanfangszustand z_0 den Übergang nach z_0 und der Teilanfangszustand z_1 den Übergang nach z_2, insgesamt geht es also nach $z_0\, z_2$. Alle Mengenzustände, die den Endzustand z_3 enthalten, können wir zu einem Mengenendzustand zusammenfassen.

Das Umwandlungsverfahren liefert für den *pen*-Akzeptor die Lösung:

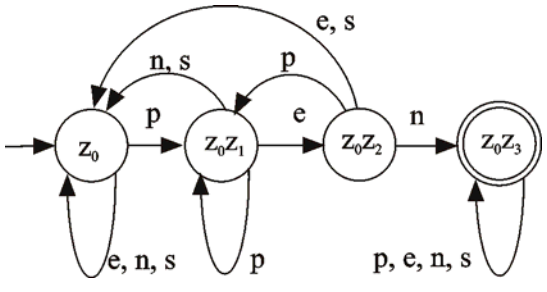


Abb. 18-12 deterministischer Akzeptor für *pen*

18.9 Automaten mit ε-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.

Zunächst betrachten wir ein Beispiel und kommen dann auf die Eingangsfrage zurück.

Beispiel: Akzeptor für $0^i1^j2^k$, $0 \le i, j, k$

Der gesuchte Akzeptor soll Wörter 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. Beliebig heißt insbesondere auch null Ziffern. Der Automat soll beispielsweise 00002 erkennen.

Ein Zustandsdiagramm mit ε-Übergängen löst dieses Problem sofort:

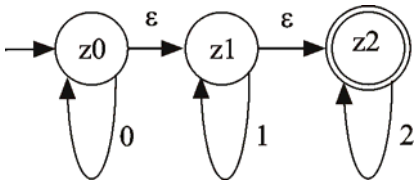


Abb. 18-13 Zustandsdiagramm mit ε-Übergängen

Automaten mit ε-Übergängen sind äquivalent zu Automaten ohne ε-Übergänge, sie sind also nicht leistungsfähiger. Das folgende Verfahren zeigt, wie eine entsprechende Umwandlung vorgenommen werden kann.

Vom Zustand z_0 kommt man bei Eingabe von 0 zum Zustand 0, aber über ein oder zwei weitere spontane ε-Übergänge auch zu den Zuständen z_1 und z_2. Liest der Automat im Zustand z_0 eine 1 so macht er erst einen ε-Übergang, um dann die 1 zu verarbeiten. Anschließend kann er noch einen weiteren ε-Übergang durchführen. Er kommt also in den Mengenzustand $z_1\, z_2$. Wenn er im Zustand z_0 eine 2 bekommt, muss er erst zwei ε-Übergänge machen und kann dann die 2 verarbeiten, erreicht also insgesamt Zustand z_2.

Auf diese Weise erstellt man die vollständige Übergangstabelle. Mengenzustände, die z_2 enthalten, sind Endzustände.

	0	1	2
z_0	$z_0 z_1 z_2$	$z_1 z_2$	z_2
z_1	-	$z_1 z_2$	z_2
z_2	-	-	z_2

Die Lösung ohne ε-Übergänge ist in der Regel komplizierter:

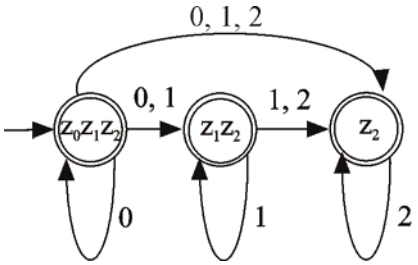


Abb. 18-14 äquivalentes Zustandsdiagramm ohne ε-Übergänge

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:

```
% normaler ε-Übergang
automat(Zustand, Eingabe):-
  eps_uebergang(Zustand, NeuerZustand),
  write(Zustand), tab(2),
  write(eps), write(' -> '),
  writeln(NeuerZustand),
  automat(NeuerZustand, Eingabe).

% ε-Übergang in den Endzustand
automat(Zustand, []):-
  eps_uebergang(Zustand, NeuerZustand),
  write(Zustand), tab(2),
  write(eps), write(' -> '),
  writeln(NeuerZustand),
  automat(NeuerZustand, []).
```

Die ε-Übergänge unseres Beispiels können wie folgt programmiert werden:

```
eps_uebergang(z0, z1).
eps_uebergang(z1, z2).
```

Wir kommen auf die Ausgangsfrage zurück, wie man aus gegebenen Akzeptoren neue Akzeptoren zusammenbauen kann. Als Konstruktionsprinzip benutzt man ε-Übergänge zur Realisierung der Verkettung, Alternative und beliebige Wiederholung. 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.

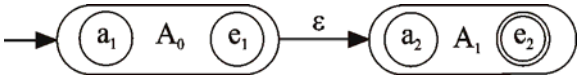


Abb. 18-15 Verkettung (Sequenz) von Akzeptoren

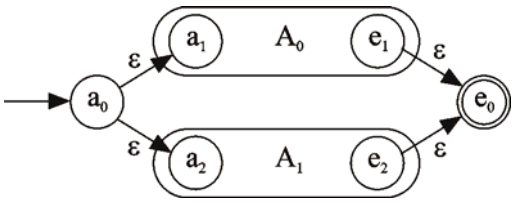


Abb. 18-16 Alternative (Selektion) von Akzeptoren

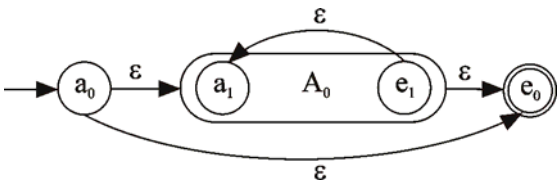


Abb. 18-17 beliebige Wiederholung (Iteration) von Akzeptoren

18.10 Reguläre Ausdrücke

Akzeptiert A_1 das Wort x und A_2 das Wort y so können wir uns nach dem Bauplan der Abbildung 18-15 einen Akzeptor für das zusammengesetzte Wort xy zusammenbauen. Mit dem Bauplan für die Alternative 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.

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* beschreiben können. Außer den bisher eingeführten Operatoren brauchen wir allerdings noch Klammern, um Baugruppen kennzeichnen zu können. Beispiele für reguläre Ausdrücke:

- $a(a|b)^*b$ entspricht einem Automaten, der alle Wörter 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.

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 ausgelotet: Endliche erkennende Automaten können genau die durch reguläre Ausdrücke beschreibbaren Sprachen erkennen.

18.11 Anwendung regulärer Ausdrücke

In vielen Programmiersprachen und Anwendungssystemen werden reguläre Ausdrücke unterstützt. Für deren Syntax gibt es keinen Standard, die nachfolgende Tabelle zeigt, was typischerweise zur Bildung regulärer Ausdrücke angeboten wird.

Zeichen	Bedeutung
.	Der Punkt ist Platzhalter für ein beliebiges Zeichen außer für neue Zeile: `\n`
\s	Erkennt Leerräume, d. h. Leerzeichen und die Steuerzeichen Tabulator, neue Zeile, Return, neue Seite: `[ \t\n\r\f]`
\S	Erkennt Nicht-Leerräume: `[^ \t\n\r\f]`
\b	Erkennt Wortgrenzen, die Position zwischen \w und \W-Zeichen.
\B	Erkennt Nicht-Wortgrenzen
^	Kontextabhängig: Erkennt den Zeilenanfang. Innerhalb einer Zeichenklasse [^...] negiert es diese.
$	Erkennt das Zeilenende
\d	Erkennt Ziffern: [0-9]
\D	Erkennt Nicht-Ziffern: [^0-9]
\w	Erkennt alphanumerische Zeichen und den Unterstrich: [a-z-A-Z0-9_]
\W	Erkennt Nicht-\w-Zeichen: [^a-z-A-Z0-9_]
[...]	Die in den eckigen Klammern stehenden Zeichen werden als Alternative verwendet. Es können Bereiche angeben werden: [a-p]. [^...] negiert die Klasse. Die Zeichenklasse steht für ein Zeichen, kann aber mit Wiederholungszeichen (*, ?, +, {n,m}) vervielfältigt werden.
\	Der Backslash hebt die besondere Bedeutung von Metazeichen auf: +?.*()^$[{|\ um diese als Text suchen zu können, bzw. macht aus Buchstaben Steuerzeichen.
...|...	Erstellt Alternativen für das Suchmuster. Die erste auftretende Alternative im String wird gefunden.
(...)	Dient der Gruppierung von Suchmustern. Das gefundene Muster wird in ein Subpattern für spätere Verwendung gespeichert.
\Zahl	Rückbezüge. Sie verweisen auf ein zuvor in runden Klammern (...) gefundenes Muster. \1 ... \99. Wird insbesondere zum Ersetzen benutzt.
?	Kontextabhängig mit mehreren Bedeutungen: Erkennt vorhergehendes Element 0- oder 1-mal. Hebt die Gierigkeit der anderen Quantifizierer auf: z. B. wird mit (.*?) die minimal nötige Menge gefunden.
*	Erkennt vorhergehendes Element 0-, 1- oder n-mal.
+	Erkennt vorhergehendes Element 1- oder n-mal.
{n,m}	Erkennt vorhergehendes Element n-mal bis höchstens m-mal. ',m' kann entfallen, dann erkennt {n} das Element n-mal und {n,} beliebig oft, aber mindestens n-mal.

Beispiel 1: (Joesef|Sepp) M[ae][iy]e?r

In diesem Beispiel gibt es für den Vornamen die Alternativen Josef und Sepp. Der Nachname darf mit a oder e bzw. i oder y geschrieben werden. Das nachfolgende e ist optional, zum Schluss folgt ein r.

Beispiel 2: [+-]?[0-9]+\.[0-9]+

Dieser reguläre Ausdruck beginnt mit einem optionalen Vorzeichen, auf das ein- oder mehrere Ziffern folgen. Danach folgt ein Dezimalpunkt auf den ein oder mehrere Ziffern folgen.

Beispiel 3: [ab]*abb

Findet Wörter aus den Buchstaben a und b, die auf abb enden.

Beispiel 4:
Zum Einfügen in eine Datenbank soll das Kalenderdatenformat TT.MM.JJJJ in das Format JJJJ-MM-TT geändert werden:

Regulärer Ausdruck zum Suchen:
([0-9]{2})\.([0-9]{2})\.([0-9]{4})

Regulärer Ausdruck zum Ersetzen:
\3-\2-\1

18.12 Die Grenzen endlicher Automaten

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^nb^n. Ist diese Teilsprache regulär, wird sie durch einen endlichen Automaten erkannt?

Die Baupläne geben nichts her. Also versuchen wir eine individuelle Konstruktion:

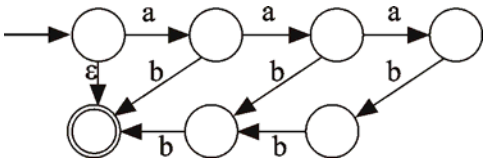


Abb. 18-18 Zustandsdiagramm für Akzeptor bis n=3

Dieser Automat schafft ε, ab, aabb und 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^mb^m mit m = z div 2 erkennen. Um a^nb^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.

18.13 Alternative Zugänge

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 das Zustandsdiagramm 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([X]):-
  is_alpha(X).
wort([Kopf|Rumpf]):-
  wort(Rumpf),
  wort([Kopf]).
```

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]):-
  dualziffer(Kopf),
  dualzahl(Rest).
```

Spezielle Automaten lassen sich leichter zur weiterführenden Interpretation der akzeptierten Eingabe 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, _).
```

18.14 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, übergeordnete Prädikate zu vereinfachen. Diese Vereinfachungen führen dann letztlich zu einem speziellen Automaten.

Den Akzeptor für Bezeichner haben wir bislang wie folgt modelliert:

```
zustand(X):-
  member(X, [neutral,korrekt,falsch]).

anfangszustand(neutral).
endzustand(korrekt).

uebergang(neutral, Eingabe, korrekt):-
  is_alpha(Eingabe).
uebergang(korrekt, Eingabe, korrekt):-
  is_alnum(Eingabe).
uebergang(_Zustand, _Eingabe, falsch).

automat(Eingabe):-
  anfangszustand(Zustand),
  atom_chars(Eingabe, Liste),
  automat(Zustand, Liste).
automat(Zustand, [Eingabe|Rest]):-
  uebergang(Zustand, Eingabe, NeuerZustand),
  write(Zustand), tab(2),
  write(Eingabe), write('  ->  '),
  writeln(NeuerZustand),
  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:

```
uebergang(neutral, Eingabe, korrekt):-
  is_alpha(Eingabe).
uebergang(korrekt, Eingabe, korrekt):-
  is_alnum(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.

```
automat(Eingabe):-
  automat(neutral, Eingabe).
automat(neutral, [Eingabe|Rest]):-
  is_alpha(Eingabe),
  automat(korrekt, Rest).
automat(korrekt, [Eingabe|Rest]):-
  is_alnum(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]):-
  is_alpha(Eingabe),
  automatkorrekt(Rest).
automatkorrekt([Eingabe|Rest]):-
  is_alnum(Eingabe),
  automatkorrekt(Rest).
automatkorrekt([]).
```

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]):-
  is_alpha(Eingabe),
  akzeptiere_bezeichnerendung(Rest).
akzeptiere_bezeichnerendung([Eingabe|Rest]):-
  is_alnum(Eingabe),
  akzeptiere_bezeichnerendung(Rest).
akzeptiere_bezeichnerendung([]).
```

18.15 Syntaxdiagramme und Automaten

Wir haben in diesem Kapitel einen systematischen Weg zur Umsetzung des Zustandsdiagramms eines Automaten in eine Prolog-Modellierung behandelt. Weitgehend unbeachtet blieb bislang der Übergang vom Syntaxdiagramm zum Zustandsdiagramm. Würde man diesen Übergang systematisieren, so könnte man einen klar strukturierten Weg vom Syntaxdiagramm über das Zustandsdiagramm und die allgemeine Modellierung zur speziellen Prolog-Modellierung eines Akzeptors angeben.

Wir betrachten im Folgenden nicht den Übergang von Syntaxdiagrammen zu Zustandsdiagrammen, sondern widmen uns gleich der umfassenderen Aufgabe, zu einem Syntaxdiagramm einen speziellen Akzeptor zu konstruie-

ren. Dabei orientieren wir uns am Akzeptor für Bezeichner, mit folgendem Syntaxdiagramm.

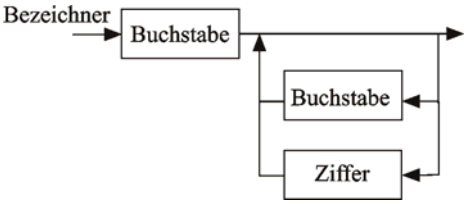


Abb. 18-19 Syntaxdiagramm für Bezeichner

Dieses Syntaxdiagramm enthält die drei relevanten Grundstrukturen Sequenz, Iteration und Selektion. Durch Aufteilen des Syntaxdiagramms in mehrere Syntaxdiagramme wird dies deutlicher:

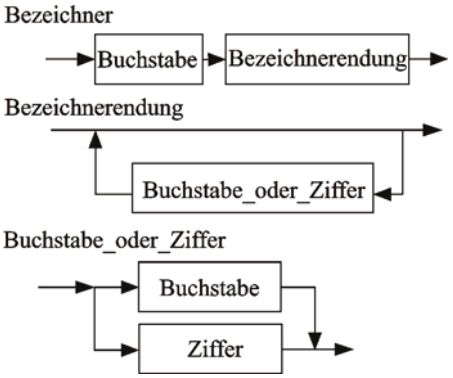


Abb. 18-20 Grundstrukturen im Syntaxdiagramm für Bezeichner

Zum Zwecke des Erkenntnisgewinns abstrahieren wir von den Details, legen die Tabelle aus Kapitel 17 zugrunde und ergänzen eine Spalte mit einem Prolog-Akzeptor für die jeweilige Grundstruktur. Aus den Syntaxdiagrammen von Abbildung 18-20 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).
```

Die so erhaltene Lösung entspricht bis auf Umbenennungen und Zusammenfassungen der Lösung, die wir durch Entfalten im vorangegangenen Abschnitt erhalten haben.

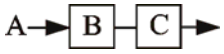
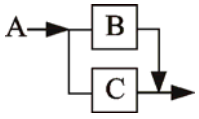
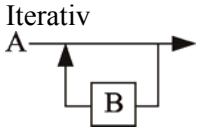
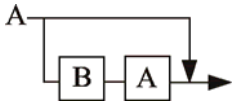
Kontroll-struktur	Grammatik-regel	Syntaxdiagramm	Akzeptor in Prolog
Sequenz	A → B C		akzeptiere_A:- akzeptiere_B, akzeptiere_C.
Selektion	A → B | C		akzeptiere_A:- akzeptiere_B. akzeptiere_A:- akzeptiere_C.
Rekursion	A → B A		akzeptiere_A:- akzeptiere_B, akzeptiere_A.
Iteration	A → { B }	Iterativ  oder besser rekursiv 	akzeptiere_A:- akzeptiere_B, akzeptiere_A. akzeptiere_A.

Tabelle 18-4 Syntaxdiagramme und Akzeptoren

18.16 Aufgaben

1. X = {a, b} sei das Alphabet eines Akzeptor.
a) Geben Sie das Zustandsdiagramm 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. Eine Digitaluhr hat eine Anzeige und zwei Schaltknöpfe A und B. Im Normalzustand zeigt die Uhr die Zeit an, im Einstellzustand kann man die Zeit einstellen. Die Funktionsweise ist im Zustandsdiagramm dargestellt.
a) Erläutern Sie das Zustandsdiagramm.
b) Die Digitaluhr steht auf 9:03 Uhr, tatsächlich ist es schon 10:07 Uhr. Geben Sie an, in welcher Reihenfolge die Schaltknöpfe betätigt werden müssen, um die richtige Zeit einzustellen.
c) Jemand drückt die Schaltknöpfe in folgender Reihenfolge:
1) ABABBBAABBAA
2) ABAABABAA
Erläutern Sie, was in den beiden Fällen passiert.
d) Die Digitaluhr kann auch als Stoppuhr benutzt werden. Drückt man im Normalzustand Knopf B, so wechselt man in den Stoppuhrmodus. Drücken auf Knopf A startet und erneutes Drücken auf Knopf A stoppt die Zeitmessung. Mit Knopf B kommt man wieder in den Normalzustand zurück. Erweitern Sie dementsprechend das Zustandsdiagramm.

5. Modellieren Sie einen Videorekorder als Automaten.

6. 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.

7. 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?

8. Skizzieren Sie einen Automaten, der den regulären Ausdruck $(a|b)^*ba$ akzeptiert.

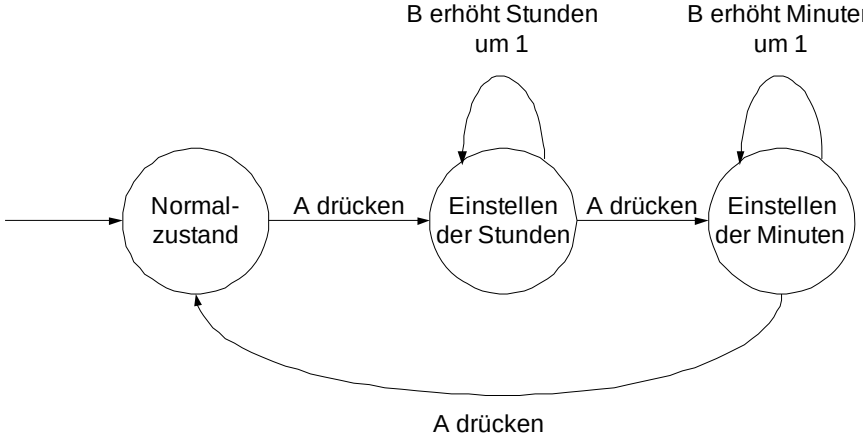
9. Entwickeln Sie einen Automaten, der alle Wörter über dem Alphabet {a, b, c} erkennt, bei denen unmittelbar nach jedem a genau ein b kommt. Beispiele: abc, cb, bcab, babcb.

10a) 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} das Zustandsdiagramm eines Akzeptors.
b) Geben Sie ein Verfahren an, mit dem eine reguläre Grammatik in ein Zustandsdiagramm überführt werden kann.

11. Wir betrachten einen erkennenden Automaten (Akzeptor) für Adressen nach dem http-Protokoll mit folgendem Aufbau:

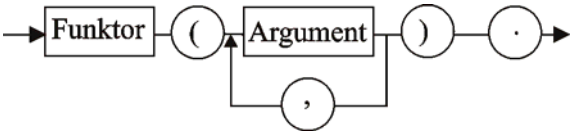
http://Rechnername.Domain.Domain....Domain

wobei für Rechnername und Domainnamen nur Kleinbuchstaben zugelassen sind. Der Automat kann immer nur ein einzelnes Zeichen verarbeiten.
a) Zeichnen Sie ein Syntaxdiagramm für solche Adressen.



b) Überführen Sie das Syntaxdiagramm in ein Zustandsdiagramm.
c) Modellieren Sie das Zustandsdiagramm in Prolog.
d) Überführen Sie das Zustandsdiagramm in eine Grammatik.
e) Erläutern Sie welcher Typ von Grammatik vorliegt.
f) Überführen Sie das Syntaxdiagramm in eine Grammatik. Welchen Typ hat diese Grammatik?

12a) Erläutern Sie das Syntaxdiagramm für Prolog-Fakten:



b) Wir beschränken die Zeichen des *Funktors* auf Kleinbuchstaben. Ein *Argument* darf ein String aus Kleinbuchstaben ohne Anführungszeichen oder eine natürliche Zahl sein. Geben Sie eine Grammatik für Funktor und Argument an.
c) Vervollständigen Sie zu einer Grammatik für Prolog-Fakten. Welcher Typ von Grammatik liegt vor?
d) Modellieren Sie das Syntaxdiagramm als Zustandsdiagramm eines deterministischen endlichen Automaten.
e) Überführen Sie das Zustandsdiagramm in eine reguläre Grammatik.
f) Interpretieren Sie die Ergebnisse aus c) und e).

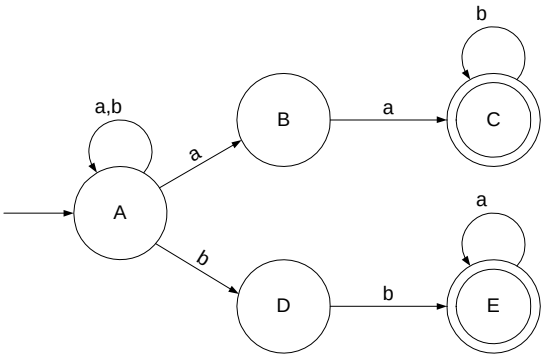
13a) Konstruieren Sie einen regulären Ausdruck für Hunderterzahlen ohne führende Null.
b) Entwickeln Sie das Zustandsdiagramm eines akzeptierenden nichtdeterministischen Automaten.
c) Berechnen Sie den zugehörigen deterministischen Automaten und stellen Sie ihn als Zustandsdiagramm dar.

14. Bei einem Getränkeautomaten kann ein Kunde wahlweise über Tasten Kaffee oder Tee auswählen. Danach kann er mit der Milch-Taste zusätzlich Mich, mit der Zucker-Taste zusätzlich Zucker erhalten. Weiteres Drücken der Milch- bzw. Zucker-Taste schaltet die Milch- bzw. Zuckerzugabe jeweils um. Nach Einwurf einer 1 € Münze wird erst der Becher ausgegeben, dann das gewählte Getränk und falls erforderlich Milch und zuletzt Zucker eingefüllt.
a) Der Getränkeautomat soll als endlicher Automat modelliert werden. Bestimmen Sie das Ein- und Ausgabealphabet.
b) Zeichnen Sie ein Zustandsdiagramm für den Automaten. Benutzen Sie für die Getränkeausgabe ε-Übergänge.
c) Erläutern Sie die Arbeitsweise des Automaten am Beispiel gezuckerter Milchkaffee.
d) Legen Sie eine Zustandsübergangstabelle an und tragen Sie fünf Übergänge ein.
e) Erläutern Sie den Aufbau und die Größe der Tabelle.
f) Modellieren Sie die statische Struktur des Automaten in Prolog.

15. Im Internet-Standarddokument (Request for Comment) RFC 1945 für das HTTP-Protokoll findet man die im Kasten dargestellte Grammatik für erlaubte Datum-Uhrzeit-Formate.
a) Geben Sie das heutige Datum und die aktuelle Uhrzeit in den möglichen Formaten an.
b) Vergleichen und bewerten Sie die Formate.
c) Stellen Sie die Grammatik in Form von Syntaxdiagrammen dar.
d) Entwickeln Sie das Zustandsdiagramm eines Akzeptors für *time*. Nur gültige Zeiten sollten akzeptiert werden.
e) Schreiben Sie einen Akzeptor für *date1*.
f) Erläutern Sie an einem Beispiel die Grenzen dieser Grammatik.

```
HTTP-date     = rfc1123-date | rfc850-date | asctime-date
rfc1123-date  = wkday "," SP date1 SP time SP "GMT"
rfc850-date   = weekday "," SP date2 SP time SP "GMT"
asctime-date  = wkday SP date3 SP time SP 4DIGIT
date1         = 2DIGIT SP month SP 4DIGIT          ; day month year (e.g., 02 Jun 1982)
date2         = 2DIGIT "-" month "-" 2DIGIT        ; day-month-year (e.g., 02-Jun-82)
date3         = month SP ( 2DIGIT | ( SP 1DIGIT ))  ; month day (e.g., Jun  2)
time          = 2DIGIT ":" 2DIGIT ":" 2DIGIT       ; 00:00:00 - 23:59:59
wkday     = "Mon" | "Tue" | "Wed" | "Thu" | "Fri" | "Sat" | "Sun"
weekday   = "Monday" | "Tuesday" | "Wednesday" | "Thursday" | "Friday" | "Saturday" | "Sunday"
month = "Jan" | "Feb" | "Mar" | "Apr" | "May" | "Jun" | "Jul" | "Aug" | "Sep" | "Oct" | "Nov" | "Dec"
```

16. Wandeln Sie diesen nichtdeterministischen endlichen Automaten mit der Potenzmengenkonstruktion in einen endlichen Automaten um.



17. Konstruieren Sie einen endlichen Automaten, für die Sprache der Wörter aus den Terminalen a und b, die auf abb enden. Wandeln Sie diesen Automaten mit der Potenzmengenkonstruktion in einen deterministischen Automaten um.

18. Geben Sie Regeln an, nach denen man das Zustandsdiagramm eines endlichen Automaten in eine reguläre Grammatik transformieren kann.

19. Analysieren Sie diese regulären Ausdrücke:
a) <[^>]*>
b) [0-9]+(,[0-9]*)?
c) ([a-z])\1
d) [\w]+(\.[\w]+)*@[\w-]+(\.[\w]+)+

20. Geben Sie reguläre Ausdrücke an für:
a) deutsche Autokennzeichen
b) vierstellige Hexadezimalzahlen
c) IP-Adressen
d) E-Mail-Adressen

21. Suchen Sie in einem Dokument nach Buchstaben, die direkt vor oder nach Klammern oder unmittelbar nach Punkten oder Doppelpunkten stehen. Fügen Sie an diesen Stellen zusätzlich ein Leerzeichen ein.

22. Gegeben ist die Grammatik G mit den Produktionen S → ABB, A → a | aS und B → b
a) Leiten Sie drei Wörter der Sprache L(G) ab, beschreibe die Sprache L(G) und die Grammatik G.
b) Zeichnen Sie Syntaxdiagramme für die Grammatik G und fassen Sie sie dann zu einem einzigen Syntaxdiagramm zusammen.
c) Transformieren Sie das rekursive in ein iteratives Syntaxdiagramm mit Kelleroperationen.
d) Übersetzen Sie das Syntaxdiagramm in ein Zustandsdiagramm für einen Kellerautomaten.
e) Modellieren Sie das Zustandsdiagramm mit einem Akzeptor in Prolog.

23. Auf der nächsten Seite ist aus dem RFC1738 eine gekürzte Grammatik für Uniform Resource Locators (URLs) gegeben.
a) Erläutern Sie die Metazeichen dieser Grammatik. Wie werden optionale und obligatorische Schleifen unterschieden?
b) Geben Sie eine gültige ftp-URL an, mit der der Benutzer abi2001 mit seinem Passwort geheim eine ftp-Verbindung zum Server www.luo.de in das Unterverzeichnis ftp seines Homeverzeichnisses ~abi2000 herstellen kann.
c) Weisen Sie durch eine Ableitung nach, dass folgende URL korrekt ist: http://138.27.56.1:80/a.htm
d) Entwickeln Sie das Zustandsdiagramm eines endlichen Automaten, der http-URLs ohne den optionalen Anteil /hpath?search akzeptiert.
e) Zeichnen Sie je ein Syntaxdiagramm für die beiden Produktionen *httpurl* und *toplabel*.
f) Erläutern Sie den folgenden Akzeptor:

```
akzeptiere_toplabel([K|R], R):-
  is_alpha(K).
akzeptiere_toplabel(Ein, Aus):-
  Ein = [K|Ein1],
  is_alpha(K),
  akzeptiere_alphadigitminus(Ein1, Aus1),
  Aus1 = [K1|Aus],
  is_alnum(K1).
akzeptiere_alphadigitminus([K|Ein], Aus):-
  (is_alnum(K); K='-'),
  akzeptiere_alphadigitminus(Ein, Aus).
akzeptiere_alphadigitminus(L, L).
```

g) Schreiben Sie in Prolog einen Akzeptor für *hostname*. Die Akzeptoren *akzeptiere_toplabel (+Liste, -Rest)* und *akzeptiere_domainlabel (+Liste, -Rest)* stehen zur Verfügung.

RFC 1738
Uniform Resource Locators (URL) December 1994 Berners-Lee, Masinter & McCahill

```
url            = httpurl | ftpurl | newsurl | nntpurl |
                 telneturl | gopherurl | waisurl | mailtourl | fileurl

; URL schemeparts for ip based protocols:
login          = [ user [ ":" password ] "@" ] hostport
hostport       = host [ ":" port ]
host           = hostname | hostnumber
hostname       = *[ domainlabel "." ] toplabel
domainlabel    = alphadigit | alphadigit *[ alphadigit | "-" ] alphadigit
toplabel       = alpha | alpha *[ alphadigit | "-" ] alphadigit
alphadigit     = alpha | digit
hostnumber     = digits "." digits "." digits "." digits
port           = digits
user           = *[ uchar | ";" | "?" | "&" | "=" ]
password       = *[ uchar | ";" | "?" | "&" | "=" ]
urlpath        = *xchar    ; depends on protocol see section 3.1

; FTP (see also RFC959)
ftpurl         = "ftp://" login [ "/" fpath [ ";type=" ftptype ]]
fpath          = fsegment *[ "/" fsegment ]
fsegment       = *[ uchar | "?" | ":" | "@" | "&" | "=" ]
ftptype        = "A" | "I" | "D" | "a" | "i" | "d"

; HTTP
httpurl        = "http://" hostport [ "/" hpath [ "?" search ]]
hpath          = hsegment *[ "/" hsegment ]
hsegment       = *[ uchar | ";" | ":" | "@" | "&" | "=" ]
search         = *[ uchar | ";" | ":" | "@" | "&" | "=" ]

; Miscellaneous definitions
lowalpha       = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" |
                 "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" |
                 "s" | "t" | "u" | "v" | "w" | "x" |
                 "y" | "z"
hialpha        = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" |
                 "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" |
                 "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z"

alpha          = lowalpha | hialpha
digit          = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
safe           = "$" | "-" | "_" | "." | "+"
extra          = "!" | "*" | "'" | "(" | ")" | ","
national       = "{" | "}" | "|" | "\" | "^" | "~" | "[" | "]" | "`"
punctuation    = "<" | ">" | "#" | "%" | <">

reserved       = ";" | "/" | "?" | ":" | "@" | "&" | "="
hex            = digit | "A" | "B" | "C" | "D" | "E" | "F" |
                 "a" | "b" | "c" | "d" | "e" | "f"
escape         = "%" hex hex

unreserved     = alpha | digit | safe | extra
uchar          = unreserved | escape
xchar          = unreserved | reserved | escape
digits         = 1*digit
```

19 Kellerautomaten

19.1 Konzeption des Kellerautomaten

Im letzten Kapitel haben wir die Grenzen endlicher Automaten am Beispiel der Sprache $L = \{a^n b^n \mid n \in \mathbb{N}\}$ kennen gelernt. 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.

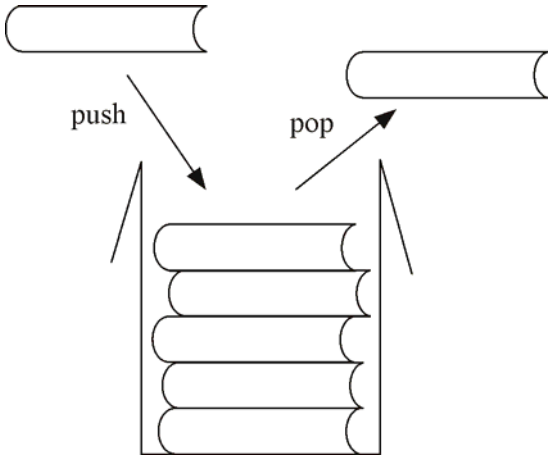


Abb. 19-1 Prinzip des Kellerspeichers

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:

push: Legt ein Element auf dem Keller ab.
pop: Holt ein Element aus dem Keller.
top: Schaut nach, welches Element im Keller obenauf liegt.

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.

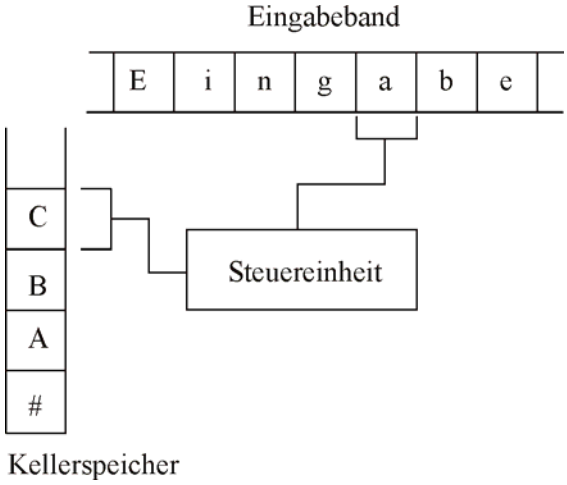


Abb. 19-2 Modell eines Kellerautomaten

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 oberste Zeichen im Keller (top). In Abhängigkeit von den drei Größen aktueller Zustand, Eingabezeichen und Speicherzeichen macht der Kellerautomat einen Zustandsübergang und zusätzlich immer eine 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).

Ist für eine Kombination aus Zustand, Eingabezeichen und Speicherzeichen kein Übergang festgelegt, so hält der Kellerautomat an. Die Eingabe kann dann nicht mehr weiter verarbeitet, insbesondere nicht akzeptiert werden. Durch diese Festlegung erspart man sich den expliziten Umgang mit Fehlerzuständen.

19.1.1 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.

Befindet sich auf dem Keller ein a, so wechselt er in den Zustand z_1 und löscht ein Zeichen vom Keller. Für den Fall, dass kein a auf dem Keller liegt, sondern ein #, wird nichts festgelegt. Bei einer solchen nicht definierten Situation hält der Kellerautomat also nicht akzeptierend an.

Im Zustand z_1 hat er mindestens ein b gelesen. Liest er ein weiteres b und als Kellerzeichen ein a, so löscht er das Kellerzeichen und verbleibt im Zustand z_1. Auf diese Weise werden die zuvor gespeicherten a durch eine entsprechende Anzahl von b neutralisiert. Für den Fall, dass im Zustand z_1 ein a gelesen wird, wird kein Übergang festgelegt, denn nach einem b darf kein a mehr vorkommen.

Hatte die Eingabe zu viele b, dann liest der Kellerautomat irgendwann ein b zusammen mit dem Kellerzeichen #. Hierfür wird kein Übergang festgelegt, weswegen der Kellerautomat anhält. Hat hingegen die Eingabe zu wenig b, dann ist nach Abarbeitung der Eingabe der Keller nicht leer.

Wörter der Sprache L werden genau dann akzeptiert, wenn die Eingabe komplett gelesen und der Kellerspeicher leer ist. Wir brauchen bei dieser Sprache keine Endzustände. Im Zustand z_0 wird bei leerem Keller das leere Wort ε akzeptiert und im Zustand z_1 alle Wörter mit $n \geq 1$. Bei manchen Sprachen reicht allerdings die Akzeptanz durch leeren Keller nicht aus, dann braucht auch der Kellerautomat Endzustände.

Übersichtlich kann man die Schaltung der Steuereinheit in einer Tabelle darstellen, die zu jeder Kombination aus Zustand, Eingabezeichen und Kellerzeichen die Kelleroperation und den Folgezustand angibt.

Zustand	Eingabe	Kellerzeichen	Operation	neuer Zustand
z_0	a	#	push(a)	z_0
z_0	a	a	push(a)	z_0
z_0	b	a	pop	z_1
z_1	b	a	pop	z_1

Tabelle 19-1 Kellerautomat für die Sprache a^nb^n

Man kann analog zu den Zustandsdiagrammen bei Automaten auch Zustandsdiagramme für Kellerautomaten angeben.

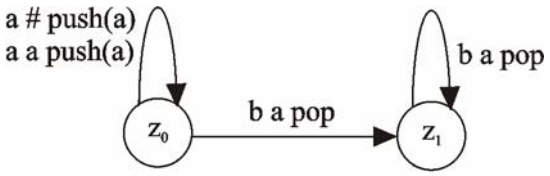


Abb. 19-3 Zustandsdiagramm eines Kellerautomaten für die Sprache a^nb^n

Blättert man in Theoriebüchern, so findet man nur selten Zustandsdiagramme 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.

Der Blick auf das Zustandsdiagramm eines Kellerautomaten ist weniger aussagekräftig als bei Automaten, weil das Verhalten des Kellerautomaten hinsichtlich seines Speichers durch das Zustandsdiagramm nicht dargestellt wird.

Automaten benutzen Zustände als endlichen Speicher. Je leistungsfähiger endliche Automaten sein sollen, umso mehr Zustände brauchen sie. Ein Zustandsdiagramm veranschaulicht daher gut den Aufbau der Steuereinheit eines Automaten.

19.1.2 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.

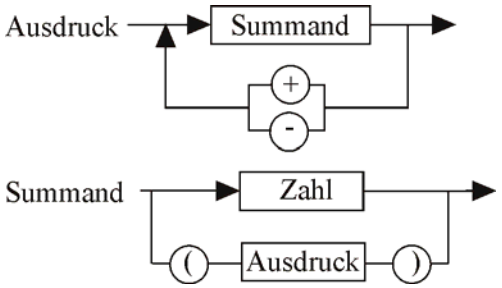


Abb. 19-4 Syntaxdiagramme für arithmetische Ausdrücke

Die Konstruktion eines Kellerautomaten ist längst nicht so einfach, wie bei den Automatenbeispielen. Die Einfachheit der folgenden Lösung für das Erkennen arithmetischer Ausdrücke täuscht über den Aufwand hinweg, sie zu finden. Eine Lösungsmöglichkeit besteht darin, zunächst die Wiederholung von Summanden wegzulas-

sen. Die zulässigen Ausdrücke sind dann von der Form: Zahl, (Zahl), ((Zahl)), (((Zahl))) und so weiter. Daraus lässt sich das folgende Zustandsdiagramm ableiten:

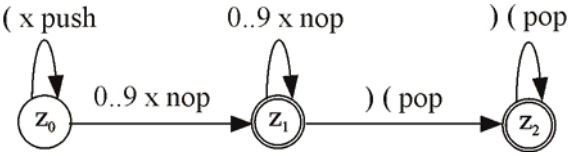


Abb. 19-5 Zustandsdiagramm eines Kellerautomaten für geklammerte Zahlen

Abkürzend steht hier x für ein beliebiges Eingabezeichen und 0..9 für eine Ziffer. Der Endzustand z_1 kennzeichnet reine Zahlen, der Endzustand z_2 einen Klammerausdruck. An die beiden Endzustände können sich Operatoren anschließen, welche die Wiederholung eines Summanden zulassen. Dazu wird das Zustandsdiagramm um Übergänge erweitert:

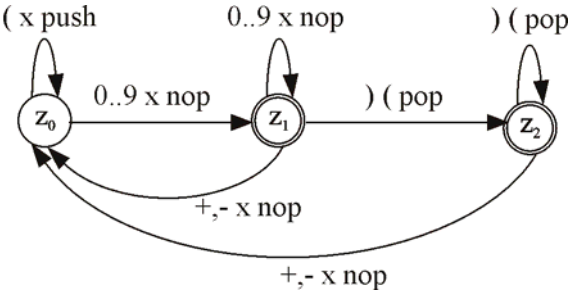


Abb. 19-6 Zustandsdiagramm eines Kellerautomaten für arithmetische Ausdrücke

Bei der Sprache der arithmetischen Ausdrücke reicht die Akzeptanzbedingung leerer Keller bei gelesener Eingabe nicht aus. So wäre beispielsweise nach der Abarbeitung der Eingabe „25+“ zwar der Keller leer, offensichtlich ist aber 25+ kein arithmetischer Ausdruck. Der Kellerautomat für arithmetische Ausdrücke braucht also Endzustände.

19.1.3 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 ungradzahliger Zeichenzahl gibt es für das mittlere Zeichen keinen Partner.

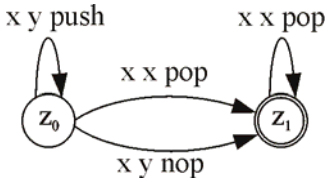


Abb. 19-7 Zustandsdiagramm eines Kellerautomaten für Palindrome

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.

19.2 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* sowie
- einer *Übergangsfunktion*, die zu jedem Tripel aus Eingabezeichen, Kellerzeichen und Zustand den Folgezustand sowie eine Kelleroperation angibt.

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 22 über Maschinelle Sprachverarbeitung. In obiger Definition lassen wir nur einfache Kelleroperationen zu.

19.3 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 $L = \{a^n b^n | n \in \mathbb{N}\}$.

```
alphabet(X):-
  member(X, [a, b]).
anfangszustand(z0).
```

Das Prädikat der Übergangsfunktion muss um zwei Parameter ergänzt werden: oberstes Kellerelement und Kelleroperation:

```
% 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.

```
akzeptiere(Wort):-
  anfangszustand(Zustand),
  atom_chars(Wort, Eingabeliste),
  kellerautomat(Zustand, Eingabeliste, [#]).
```

Anstelle des Prädikats *automat* haben wir nun das analoge Prädikat *kellerautomat*, welches als weiteren Parameter den aktuellen Keller mitführt. Im Unterschied zum Automaten wird nach der Ermittlung eines Übergangs noch die zugehörige Kelleroperation auf dem Keller ausgeführt. Dazu gibt es das neue Prädikat *kellern*.

```
% 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), write(' -> '),
  write(NeuerKeller), tab(2),
  writeln(NeuerZustand),
  kellerautomat(NeuerZustand, Rest, NeuerKeller).
kellerautomat(EndZustand, [], [#]).

% kellern(+Kelleroperation, +AlterKeller,
          -NeuerKeller)
kellern(push(Element), Keller, [Element|Keller]).
kellern(pop, [_Element|Keller], Keller).
kellern(nop, Keller, Keller).
```

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 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):-
  is_digit(Zeichen).

uebergang(z1, ')', '(', pop, z2).
uebergang(z1, Zeichen, _, nop, z1):-
  is_digit(Zeichen).


uebergang(z1, '+', _, nop, z0).
uebergang(z1, '-', _, nop, z0).

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:

```
uebergang(z0, X, _, push(X), z0).
uebergang(z0, _, _, nop, z1).
uebergang(z0, X, X, pop, z1).
uebergang(z1, X, X, pop, z1).
```

19.4 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.

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.

```
alphabet(X):-
  atom_chars('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:

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),...

19.5 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)
uebergang(z0, a, [#], [a, #], z0).
uebergang(z0, a, [a|K], [a, a|K], z0).
uebergang(z0, b, [a|K], K, z1).
uebergang(z1, b, [a|K], K, z1).
```

Bei den ersten beiden Klauseln findet eine Push-Operation statt, bei den letzten beiden eine Pop-Operation.

19.6 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:

```
% erzeugte Sprache
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).
```

```
akzeptiere(Wort):-
  atom_chars(Wort, Eingabeliste),
  kellerautomat(z0, Eingabeliste, [#]).

kellerautomat(z1, [], [#]).
```

Für direktes Kellern schreiben wir die Übergänge um

```
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(Zustand,
              [Eingabe|Rest], Keller):-
  uebergang(Zustand, Eingabe,
            Keller, NeuerKeller,
            NeuerZustand),
  kellerautomat(NeuerZustand,
                Rest, NeuerKeller).
```

Das Wesentliche des *Entfaltens* haben wir jetzt schon kennen gelernt: 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:

```
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):-
  atom_chars(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, [], [#]).
```

19.7 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.

19.7.1 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):-
  is_alpha(Zeichen).
uebergang(Zustand, ')', '(', pop, z2):-
  endzustand(Zustand).
uebergang(z1, Zeichen, _, nop, z1):-
  is_alpha(Zeichen).
uebergang(Zustand, Op, _, nop, z0):-
  endzustand(Zustand),
  operator(Op).

endzustand(X):-
  member(X, [z1, z2]).
operator(X):-
  member(X, ['+', '-']).
```

Entfaltet man den Kellerautomaten für arithmetische Ausdrücke und nennt das entstehende Prädikat *ausdruck* so erhält man:

```
ausdruck(Wort):-
  atom_chars(Wort, Eingabeliste),
  ausdruck(z0, Eingabeliste, [#]).

ausdruck(Zustand, Eingabe, Keller):-
  write(Zustand), tab(2),
  write(Eingabe), tab(2),
  writeln(Keller), fail.

ausdruck(z0, ['('|Rest], Keller):-
  ausdruck(z0, Rest, ['('|Keller]).

ausdruck(z0, [Zeichen|Rest], Keller):-
  is_alpha(Zeichen),
  ausdruck(z1, Rest, Keller).

ausdruck(z1, [Zeichen|Rest], Keller):-
  is_alpha(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.

19.7.2 Ergänzung der Interpretation

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 ursprünglichen 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 sind dann *% start, % Ausgabe und % öffnende Klammer*.

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. (*% erste Ziffer, % weitere Ziffern*).

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 werden 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. (*%erster Operator, %weiterer Operator, %hier wird gerechnet*).

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 berechnen, % Endergebnis ausgeben*).

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, % Klammer zu*).

Im folgenden Beispiel einer Interpretation werden pro Zeile ausgeben: 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
```

Die Funktionstüchtigkeit des Interpreters lässt sich am einfachsten mit dem Generator prüfen:

```
?- erzeugteSprache(X), interpretiere(X).
```

```
% Interpretation arithmetischer Ausdrücke

% Start
interpretiere(Wort):-
  atom_chars(Wort, Eingabeliste),
  interpretiere(z0, Eingabeliste, [#], [#]).

% Ausgabe
interpretiere(Zustand, Eingabe, Operatoren, Operanden):-
  write(Zustand), tab(2), write(Eingabe), tab(2),
  write(Operatoren), tab(2), writeln(Operanden), fail.

% öffnende Klammer
interpretiere(z0, ['('|Rest], Operatoren, Operanden):-
  interpretiere(z0, Rest, ['('|Operatoren], Operanden).

% erste Ziffer
interpretiere(z0, [Zeichen|Rest], Operatoren, Operanden):-
  is_alpha(Zeichen),
  char_code(Zeichen, Ziffer),
  Zahl is Ziffer - 48,
  interpretiere(z1, Rest, Operatoren, [Zahl|Operanden]).

% weitere Ziffern
interpretiere(z1, [Zeichen|Rest], Operatoren, [Zahl1|Operanden]):-
  is_alpha(Zeichen),
  char_code(Zeichen, Ziffer),
  Zahl2 is Zahl1*10 + Ziffer - 48,
  interpretiere(z1, Rest, Operatoren, [Zahl2|Operanden]).

% 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 ausfuehren
interpretiere(Zustand, [Op1|Rest], [Op2|Operatoren], Operanden):-
  endzustand(Zustand),
  operator(Op1),
  berechne(Op2, Operanden, NeuOperanden),
  interpretiere(z0, Rest, [Op1|Operatoren], NeuOperanden).

% 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).

% 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).
```

```
% hier wird gerechnet
berechne(Op, [Zahl2, Zahl1|Operanden],
         [Zahl3|Operanden]):-
  operator(Op),
  (Op = '+', Zahl3 is Zahl1 + Zahl2;
   Op = '-', Zahl3 is Zahl1 - Zahl2).

test:-
  interpretiere('2+(3-4-(4-5))+1').
```

19.8 Kelleroperationen auf Syntaxdiagrammen

In Beispiel 19.1.1 wurde schon auf die große Schwierigkeit hingewiesen, zu gegebenen Grammatiken bzw. Syntaxdiagrammen Kellerautomaten zu konstruieren. Wir schauen uns daher eine vom Autor entwickelte Methode an, mit der diese Problemstellung gelöst werden kann.

Als Ausgangspunkt wählen wir Syntaxdiagramme, weil sich durch den visuellen Zugang Sachverhalte leichter verstehen lassen. Im ersten Schritt setzen wir die Syntaxdiagramme ineinander ein, so dass ein Syntaxdiagramm entsteht.

Greifen wir das einleitende Beispiel der arithmetischen Ausdrücke auf, so sieht dieses zusammengefasste Syntaxdiagramm wie folgt aus:

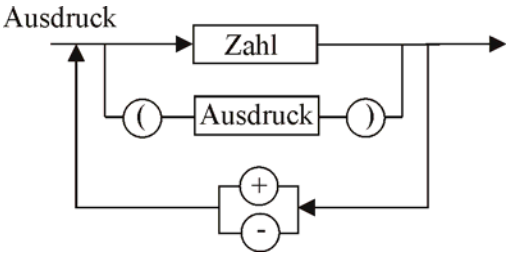


Abb. 19-8 rekursives Syntaxdiagramm

Charakteristisch ist die geklammerte Rekursionsstruktur des entstandenen Syntaxdiagramms. Ohne einen solchen geklammerten rekursiven Anteil würde ein endlicher Automat zum Erkennen ausreichen.

Wir beseitigen die Rekursion jetzt durch Einsatz eines Kellers, in dem wir die öffnende Klammer pushen und nach der Klammer zum Anfang des Syntaxdiagramms gehen. Das entspricht dem bisherigen Syntaxdiagramm, weil dort nach der öffnenden Klammer es auch mit *Ausdruck* weitergeht.

Wenn der innere Ausdruck abgearbeitet ist, geht es zur schließenden Klammer. Dies ersetzen wir durch einen Pfad vom Ausgang des gesamten Ausdruck-Syntaxdiagramms, gehen von dort vor die schließende Klammer und führen bei der schließenden Klammer die pop-Operation aus.

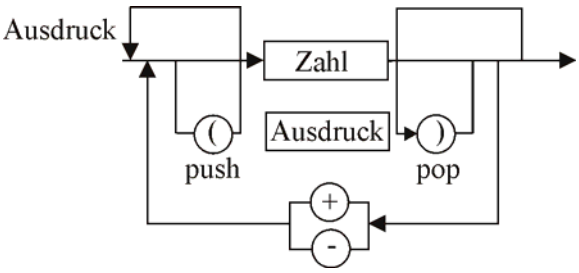


Abb. 19-9 entrekursiviertes Syntaxdiagramm

Durch Einsatz eines Kellers und Ergänzung von push- und pop-Operationen auf dem Syntaxdiagramm wurde die Rekursion beseitigt. Vereinfachungen führen zu folgendem Syntaxdiagramm.

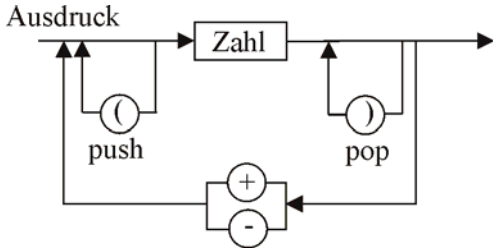


Abb. 19-10 vereinfachtes Syntaxdiagramm

Dieses Syntaxdiagramm kann nun analog zu der in Kapitel 18.4 für Automaten entwickelten Methodik in einen Kellerautomaten transformiert werden. In unserem überschaubaren Beispiel kann man aber durch geeignete Einführung von Zuständen auch direkt den Kellerautomaten ablesen. Mit zwei Zuständen, einem vor und einem hinter *Ziffer* kommt man nicht aus, weil nach einer schließenden Klammer es nicht mit einer Ziffer weitergehen darf. Eine schließende Klammer steht am Schluss oder es folgt ein Rechenzeichen.

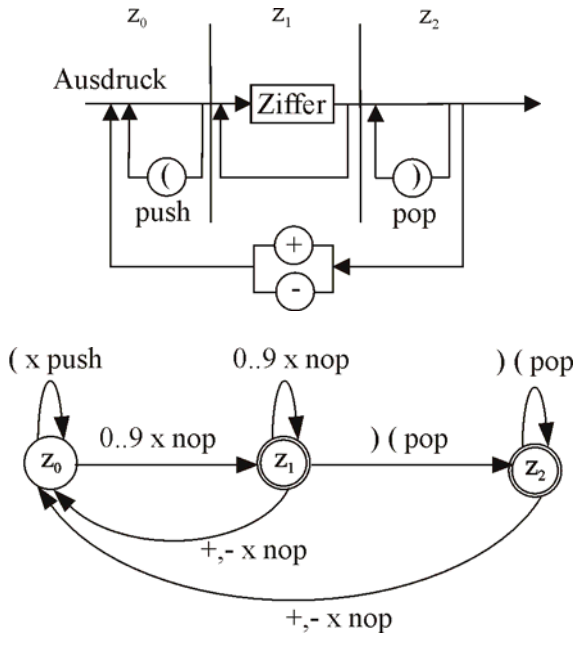


Abb. 19-11 direkt in Kellerautomat umsetzbares Syntaxdiagramm

Bei der Umsetzung achtet man insbesondere auf die Verzweigungen im Syntaxdiagramm, die zu unterschiedlichen Übergangspfeilen im Zustandsdiagramm führen.

19.9 Die Grenzen von Kellerautomaten

Die formale Sprache $L_1 = \{a^n \mid n \in \mathbb{N}\}$ ist regulär und wird von einem Automaten erkannt. Endliche 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 17 haben wir eine kontextsensitive Grammatik kennen gelernt, 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.

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.*

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:

Grammatik	Maschinenmodell
regulär	Automat
kontextfrei	nichtdeterministischer Kellerautomat
kontextsensitiv	Turingmaschine

Tabelle 19-2 Grammatiken und zugehörige Maschinenmodelle

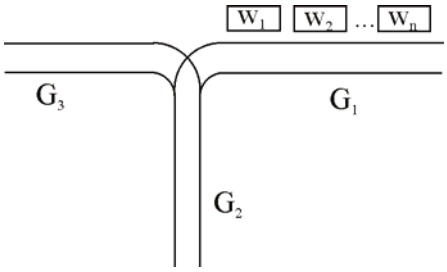
19.10 Aufgaben

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 das Zustandsdiagramm eines erkennenden deterministischen Kellerautomaten. Hilfe: Beginnen Sie mit $S \rightarrow (x)$ und erweitern Sie das Zustandsdiagramm 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.

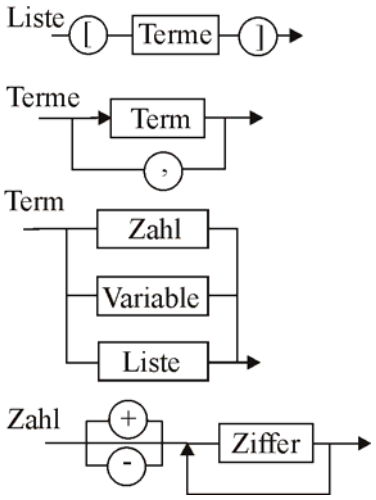
5. Gegeben ist die Grammatik $S \rightarrow a \mid b \mid (S + S)$. Entwickeln Sie einen Kellerautomaten, der genau die Sätze dieser Sprache akzeptiert. Hinweis: Es ist nach dem bisherigen Verfahren sehr schwierig, einen solchen Kellerautomaten zu konstruieren. Wählen Sie daher ein besseres Verfahren, 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.

6. Gegeben ist die Grammatik G mit den Produktionen
   $S \rightarrow ABB$
   $A \rightarrow a \mid aS$
   $B \rightarrow b$

a) Leiten Sie drei Wörter der Sprache L(G) ab, beschreiben Sie die Sprache L(G) und die Grammatik G.
b) Zeichnen Sie Syntaxdiagramme für die Grammatik G und fassen Sie sie dann zu einem einzigen Syntaxdiagramm zusammen.
c) Transformieren Sie das rekursive in ein iteratives Syntaxdiagramm mit Kelleroperationen.
d) Übersetze das Syntaxdiagramm in ein Zustandsdiagramm für einen Kellerautomaten.
e) Modelliere das Zustandsdiagramm mit einem Akzeptor in Prolog.

7a) Erläutern Sie die nachfolgenden Syntaxdiagramme für Listen in Prolog.
b) Begründen Sie mit einem Ableitungsbaum, dass [2, X, [17, 4]] eine Liste ist.
c) Ergänzen Sie die Syntaxdiagramme so, dass auch die leere Liste und der Listenoperator "|" benutzt werden können.
d) Übersetzen Sie die Syntaxdiagramme in eine Grammatik. Benutze Sie bei den Schleifen rekursive Produktionen. Von welchem Typ ist die Grammatik, durch welchen Automaten kann sie erkannt werden?
e) Geben Sie Prolog-Akzeptoren der Art `akzeptiere_liste(Eingabe,Ausgabe)` für die Syntaxdiagramme an.



Syntaxdiagramme zu Aufgabe 6.

8. Eine formale Sprache L sei durch die folgende Grammatik gegeben:

   Nichtterminale = {S, B, C}
   Terminale = {a, b, c, ε} (ε = leeres Wort)
   Startzeichen S
   Produktionen:
   $S \rightarrow aSa$ $S \rightarrow bBb$ $S \rightarrow cC$
   $B \rightarrow bBb$ $B \rightarrow cC$ $C \rightarrow cC$ $C \rightarrow \varepsilon$

a) Bilden Sie einige Worte der Sprache L und beschreiben Sie diese Sprache. Begründen Sie, dass diese Sprache nicht durch einen endlichen Automaten erkannt werden kann und ordnen Sie sie in die Chomsky-Hierarchie ein.
b) Entwickeln Sie einen Kellerautomaten K, der alle Worte der Sprache L erkennt.
c) Die Sprache $T = \{a^n b^n c b^n a^n\}$ mit $n \geq 1$ ist eine Teilmenge von L. Erläutern Sie, warum die Sprache T nicht mit einem Kellerautomaten überprüft werden kann.

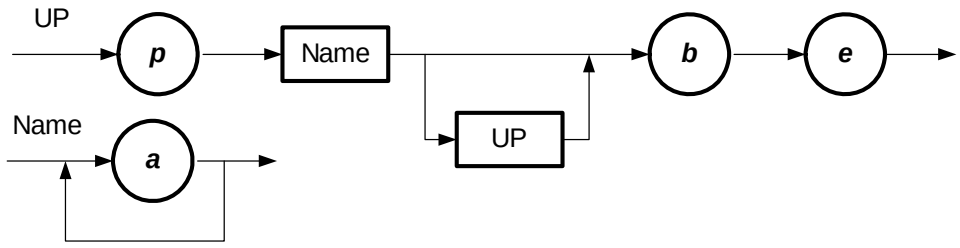
9. Gegeben sind die nachstehenden Syntaxdiagramme der Sprache L(UP), die den Aufbau geschachtelter und für die Zwecke der Aufgabe vereinfachter Unterprogramme UP beschreiben. Dabei wird in Anlehnung an die Syntax von Pascal das Terminal *p* für *procedure*, *b* für *begin* und *e* für *end* verwendet.

a) Beschreiben Sie, wie geschachtelte Unterprogramme der Sprache L(UP) gebildet werden müssen. Geben Sie drei verschiedene Wörter der Sprache L(UP) an.
b) Analysieren Sie, ob die Wörter der Sprache L(UP) von einem Kellerautomaten oder bereits von einem endlichen Automaten akzeptiert werden können. Begründen Sie Ihre Entscheidung.
c) Entwerfen Sie die Übergangsfunktion eines Kellerautomaten, der syntaktisch korrekte

Wörter der Sprache L(UP) akzeptiert. Überprüfen Sie Ihre Lösung anhand der Eingabefolgen *paabe* und *papabe*. Notieren Sie dazu jeweils die Arbeitsschritte des Kellerautomaten in der Form:

restliche Eingabefolge Zustand aktuelle Kellerbelegung

d) Entwerfen Sie unabhängig vom Kellerautomaten eine Grammatik für die Sprache L(UP) mit Startsymbol, Terminalen, Nichtterminalen und Produktionsregeln.
e) Zeigen Sie mit einem Ableitungsbaum, dass sich das Wort paabe aus ihrer Grammatik ableiten lässt.
f) Implementieren Sie den Kellerautomaten in Prolog. Das System soll auf die Anfrage ?- uprg(Wort). mit yes oder no antworten, je nachdem ob das Wort zur Sprache L(UP) gehört oder nicht. Gehen Sie davon aus, dass das Wort bereits eine Liste von Eingabezeichen ist.



Syntaxdiagramm für Aufgabe 9

10. Gegeben sei die formale Sprache
$L = \{ a^n b^m c^r \mid n \text{ und } r \text{ gerade, } m \text{ ungerade und } n, m, r > 0 \}$
a) Konstruieren Sie einen endlichen Automaten, der die Sprache erkennt, und geben Sie eine Grammatik dazu an.
b) Geben Sie einen regulären Ausdruck an, der die Sprache aus a) beschreibt.
c) Begründen Sie anschaulich, warum ein endlicher Automat nicht ausreichen kann, wenn man zusätzlich die folgende Forderung an die Sprache stellt: m > n , n und r gerade, m ungerade und n, m, r > 0 bleibt
Beschreibe die Arbeitsweise und die Akzeptanzbedingungen eines Kellerautomaten, der diese Sprache erkennt, und stelle ihn als Zustandsdiagramm dar. ε-Übergänge dürfen benutzt werden.

11. Gegeben ist die Sprache der vollständig geklammerten Summen über den Zahlen und Variablen {a, b, c}.

Beispiele:
((17+3)+(a+4))
(((5+b)+(6+a))+31)

a) Bilden Sie zwei weitere Beispiele
b) Zeichnen Sie Syntaxdiagramme für diese Sprache.
c) Übersetzen Sie diese in eine Grammatik. Welchen Typ hat sie?
d) Zeichnen Sie das Zustandsdiagramm eines erkennenden Akzeptors.

20 Turingmaschinen

20.1 Konzeption der Turingmaschine

Wir haben die Grenzen von endlichen Automaten und von Kellerautomaten kennen gelernt. 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 er 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 genügt ein einseitig unbegrenzter Speicher. 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. 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*.

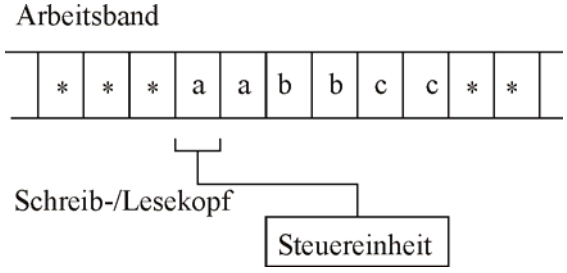


Abb. 20-1 Konzeption einer Turingmaschine

Das *Arbeitsband* ist in einzelne Zellen unterteilt. Jede Zelle kann genau ein Zeichen speichern. Das *Leerzeichen* „*“ kennzeichnet eine leere Zelle.

Der *Schreib-/Lesekopf* kann den Inhalt einer Zelle lesen und sie neu beschreiben. Im Arbeitsrhythmus 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.

Die *Steuereinheit* ist ein Automat, der endlich viele verschiedene Zustände annehmen kann. Zwei Zustände haben eine besondere Bedeutung:

A ist der Anfangszustand, in dem die Turingmaschine ihre Arbeit beginnt
E ist der Endzustand, bei dessen Erreichen die Turingmaschine ihre Arbeit beendet.

Sonstige Zustände nummeriert 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:

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:

Zustand Zeichen $\rightarrow$
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 stellt die Arbeitsweise einer Turingmaschine übersichtlich dar:

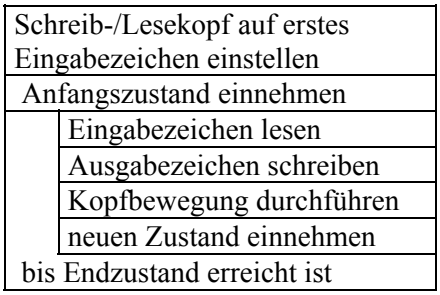


Abb. 20-2 Struktogramm zur Arbeitsweise einer Turingmaschine

20.1.1 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 mit Leerzeichen überschreibt. 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 * → * K E
```

Anderenfalls muss ein a gelesen und gelöscht werden, wobei in den Zustand 1 gewechselt wird:

```
A a → * R 1
```

Im Zustand 1 geht es an das Ende der Eingabe:

```
1 a → a R 1
1 b → 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 * → * L 2
```

Im Zustand 2 löscht die Turingmaschine ein b, wechselt in den Zustand 3 und geht zum Anfang der Eingabe zurück:

```
2 b → * L 3
3 b → b L 3
3 a → a L 3
```

Ist der Anfang der Eingabe erreicht, wechselt die Turingmaschine wieder in den Anfangszustand:

```
3 * → * R A
```

20.1.2 Turingmaschine zum Addieren

Bei geeigneter Interpretation der Bandbelegung können mit Turingmaschinen Berechnungen ausgeführt werden. Zahlen werden grundsätzlich in Unär-Darstellung als Folge von 1-ern auf das Arbeitsband geschrieben, wobei mehrere Zahlen durch das Nummernzeichen # voneinander zu trennen sind.

Erreicht die Turingmaschine den Endzustand, so steht das Ergebnis ab der Position des Schreib-/Lesekopfes auf dem Arbeitsband. Es besteht aus allen Einern, bis zum ersten von 1 verschiedenen Zeichen.

Betrachten wir als Beispiel eine Turingmaschine, die addieren kann. Wir orientieren uns an der Summe 3+5. Als Eingabe ist 111#11111 auf das leere Arbeitsband zu schreiben. Zur Berechnung der Summe löscht die Turingmaschine die erste 1 und ersetzt das Nummernzeichen durch eine 1.

Im Anfangszustand löscht die Turingmaschine eine 1 und wechselt in den Zustand 1:

```
A 1 → * R 1
```

Im Zustand 1 geht es nach rechts, bis das Nummernzeichen erreicht ist:

```
1 1 → 1 R 1
```

Das Nummernzeichen wird durch eine 1 ersetzt. Im Zustand 2 geht es an den Anfang zurück:

```
1 # → 1 L 2
2 1 → 1 L 2
```

Den Schreib-/Lesekopf positioniert die Turingmaschine auf die erste 1:

```
2 * → * R E
```

Der Sonderfall, dass der erste Summand 0 ist, muss berücksichtigt werden:

```
A # → * R E
```

20.1.3 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:

pred(n) = (if n = 0 then 0 else n-1)

Die Turingmaschine zur Berechnung von *pred* löscht eine 1, geht nach rechts und hält an:

```
A 1 → * R E
```

Ist keine 1 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.

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 111 wird die erste 1 markiert und an das rechte Ende kopiert. Danach wird die zweite 1 markiert und kopiert, bis schließlich alle 1er kopiert sind. Anschließend werden die Markier- und Kopiersymbole wieder durch 1er ersetzt. Mögliche Bandbelegungen sind:

```
111 → M11K → 1M1KK → 11MKKK → 111111
```

Das Programm der verdoppelnden Turingmaschine lautet:

```
A 1 → M R 1  ; erste 1 markieren
A * → * L 3  ; im Zustand 3 K
             ; durch 1 ersetzen
A K → K R A  ; Kopieren fertig,
             ; nach rechts laufen
1 1 → 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 1 → 1 L 2  ; nach links laufen
2 M → 1 R A  ; links angekommen,
             ; Marke löschen,
             ; Vorgang wiederholen
3 1 → 1 L 3  ; nach links laufen
3 K → 1 L 3  ; K durch 1 ersetzen
3 * → * R E  ; fertig
```

20.2 Definition der Turingmaschine

Wir fassen die Konzeption der Turingmaschine in folgender Definition zusammen.

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 Übergangsfunktion, 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 Übergangsfunktion eine Übergangsrelation.

Interessanterweise sind nichtdeterministische Turingmaschinen genauso leistungsfähig wie deterministische Turingmaschinen. Das ist bei Kellerautomaten anders, während bei Automaten ebenfalls deterministische und nichtdeterministische Varianten ebenbürtig sind.

20.3 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).
```

Die fünf Komponenten bestimmen eine spezifische Turingmaschine. Die Arbeitsweise von Turingmaschinen kann dagegen allgemeingültig beschrieben werden. Dazu definieren wir das Prädikat *turingmaschine (+Zustand, +Linker-Speicher, +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 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.

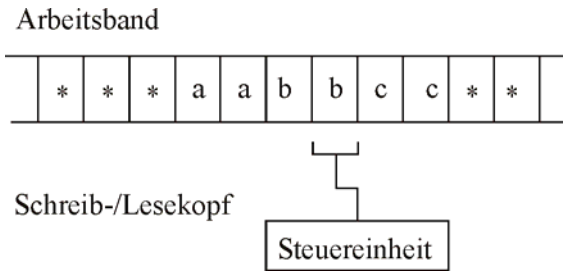


Abb. 20-3 Modellierung des Arbeitsbandes einer Turingmaschine in Prolog

Das in Abbildung 20-3 dargestellte Arbeitsband wird durch die 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.

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.

```
akzeptiere(Wort):-
  anfangszustand(Zustand),
  atom_chars(Wort, Eingabeliste),
  turingmaschine(Zustand, [], Eingabeliste).

% Anzeige der Arbeitsweise
turingmaschine(Zustand, LinkerSpeicher, RechterSpeicher):-
  write(Zustand),
  tab(2), zeigelinks(LinkerSpeicher),
  tab(1), zeigerechts(RechterSpeicher),
  nl, fail.

% Banderweiterung
turingmaschine(Zustand, [], RechterSpeicher):-
  turingmaschine(Zustand, [*], RechterSpeicher).
turingmaschine(Zustand, LinkerSpeicher, []):-
  turingmaschine(Zustand, LinkerSpeicher, [*]).

% 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).

% anzeigen
zeigelinks([]).
zeigelinks([Kopf|Rechts]):-
  zeigelinks(Rechts),
  write(Kopf).
zeigerechts([]).
zeigerechts([Kopf|Rechts]):-
  write(Kopf),
  zeigerechts(Rechts).
```

Zum Betrieb einer Turingmaschine als Akzeptor definieren wir das Prädikat *akzeptiere*. Die Veranschaulichung der Arbeit einer Turingmaschine übernimmt eine Klausel, welche vor allen anderen Klauseln des Prädikats *turingmaschine* stehen muss.

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
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 aufeinander folgenden Striche umfasst:

```
turingmaschine(Zustand, _Links, Rechts):-
  endzustand(Zustand),
  zaehle_einer(Rechts, Einer),
  write('Ergebnis: '),
  writeln(Einer).

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:

```
a   11#111
a  * 11#111
1  ** 1#111
1  **1 #111
2  ** 11111
2  * *11111
e  ** 11111
Ergebnis: 5
```

20.4 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 20.1.1 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 20.1.1 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 20.1.3 zeigt, dass mit Turingmaschinen algorithmische Grundstrukturen realisiert werden können; im Zustandsdiagramm stellt sich die Fallunterscheidung als Verzweigung, die Wiederholung als Zyklus und die Sequenz als Pfad 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 (RAM, engl. = Random Access Machine) und

weist nach, dass sich beide Maschinen wechselseitig simulieren können.

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 *μ-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:

> Jede im intuitiven Sinn berechenbare Funktion ist auch Turing-berechenbar.

Mit der Turingmaschine ist somit ein leistungsfähigstes Maschinenmodell gefunden.

20.5 Grenzen der Turingmaschine

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, ... TM_n. Nun betrachtet man die endliche Anzahl der Turingmaschinen mit zwei Übergängen, ordnet diese ebenfalls lexikographisch an und nummeriert weiter: TM_{n+1}, $TM_{n+2, ...}$ TM_m. Die Aufzählung setzt sich in gleicher Weise mit Turingmaschinen aus drei, vier,... Übergängen fort. Dieses Nummerierungsverfahren wird auch nach dem Mathematiker Kurt Gödel (1906-1978) Gödelisierung genannt.

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 Menge M_f dieser Funktionen abzählbar sei, so kann man die Funktionen in der Art f_1, f_2, f_3,... 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 20-1 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 $f_j = h$. Obwohl also *h* zur Menge M_f gehört, kommt *h* nicht in der Aufzählung f_1, f_2, f_3,... 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!

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 anhalten oder nicht?

Die Antwort lautet *Nein*. Angenommen wir hätten ein Prolog-Prädikat *haelt_an(Programm),* das von Prolog-Programmen feststellt, ob sie anhalten oder nicht. Die Programme könnten dabei als Liste ihrer Klauseln im Parameter übergeben werden. Wir könnten dann das Prädikat *ausgabe(+Programm)* programmieren, das das Ergebnis der *haelt_an*-Berechnung schön ausgibt.

```
ausgabe(Programm):-
  haelt_an(Programm),
  write(´Das Programm hält an.´).

ausgabe(Programm):-
  not(haelt_an(Programm)),
  write(´Das Programm hält nicht an.´).
```

Um die Leistungsfähigkeit des *haelt_an*-Prädikats zu testen erweitern wir nun die erste Klausel des ausgabe-Prädikats.

```
ausgabe(Programm):-
  haelt_an(Programm),
  write(´Das Programm hält an.´),
  ausgabe(Programm).

ausgabe(Programm):-
  not(haelt_an(Programm)),
  write(´Das Programm hält nicht an.´).
```

Welche Antwort liefert die Anfrage ?- *ausgabe(ausgabe-Prädikat),* wobei als Parameter ausgabe-Prädikat eine Liste der ausgabe-Klauseln mitgegeben wird? Angenommen, das Programm *ausgabe* hält an, dann sorgt die Endrekursion in der ersten *ausgabe*-Klausel dafür, dass *ausgabe* nicht anhält. Nehmen wir hingegen an, dass *ausgabe* nicht anhält, so wird das Ziel *not(haelt_an(ausgabe-prädikat))* in der zweiten Klausel erfüllt. Das nachfolgende *write*-Teilziel wird ebenfalls erfüllt, wodurch *ausgabe* anhält.

In beiden Fällen ergibt sich ein logischer Widerspruch. Da es das *ausgabe*-Prädikat wie oben angegeben gibt, kann der Widerspruch nur mit der angenommenen Voraussetzung, dass es ein *haelt_an*-Prädikat gäbe, erklärt werden. Ein solches Prädikat kann es also nicht geben.

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 geeignete 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 = $\{a_1, a_2, a_3,..., a_n\}$, Z = $\{z_1, z_2, z_3,..., z_m\}$ und B = $\{b_1, b_2, b_3\}$ nummeriert. Zeichen, Zustände und Kopfbewegungen codiert man durch die Binärdarstellung der zugehörigen Indizes: code(a_i) = bin(i), code(z_j) = bin(j) und code(b_k) = bin(k). Ein Übergang wird durch Codierung seiner Komponenten codiert:

code(z_i $a_j \rightarrow a_k$ b_l z_m) =
##bin(i)#bin(j)#bin(k)#bin(l)#bin(m)

Die Codierung einer Turingmaschine besteht dann in der Folge der Codierungen ihrer Übergänge. Die Codierung der Eingabe ergibt sich aus der Codierung 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:

E_H 1 $\rightarrow$ 1 K E_H
E_H * $\rightarrow$ * K E_S

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.

20.6 Aufgaben

1. Geben Sie Turingmaschinen für die folgenden drei Grundfunktionen an:

a) Nullfunktion: z(n) = 0, n $\in \mathbb{N}$
b) Nachfolgerfunktion: succ(n) = n + 1, n $\in \mathbb{N}$
c) Projektion auf das dritte Argument:
$p_3(x_1, x_2, x_3, x_4,... x_n) = x_3$

2. Entwickeln Sie eine Turingmaschine, die Wörter über dem Alphabet {a, b} erkennt, welche mit aab... 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:

```
111#1          ; Initialisierung
 11#11         ; erste Multiplikation
```

```
1#1111      ; zweite Multiplikation
 #11111111  ; dritte Multiplikation
  11111111  ; Ergebnis
```

4. Analysiere die folgende Turingmaschine am Beispiel der Bandbelegung 1 1 1 1 1 # 1 1 1

a) A # → * K E
A 1 → * R 1
1 1 → 1 R 1
1 # → # R 1
1 * → * L 2
2 1 → * L 3
2 # → 1 L 9
3 1 → 1 L 3
3 # → # L 3
3 * → * R A
9 1 → 1 L 9
9 * → * R E

b) Konstruiere eine Turing-Maschine, die Zahlen nach folgender Methode von der Unärdarstellung in die Binärdarstellung umrechnet. Nach Eingabe der Bandbelegung, z.B. 1 1 1 1 1 1 1 für die Zahl 7, schreibt die Turingmaschine vor die erste 1 ein # und davor eine 0. Dann entfernt Sie in einer Schleife jeweils die ganz rechts stehende 1 und erhöht dafür die links vom # entstehende Binärzahl um 1. Nach einiger Zeit haben wir also z.B. die Bandbelegung 101 # 1 1. Sind alle 1er abgearbeitet haben wir links vom # die gewünschte Dualzahl.

c) Beschreibe die Arbeitsweise der konstruierten Turingmaschine indem du die Bedeutung der Zustände und der darin stattfindenden Aktionen angibst.

5. Gödelisierung von Turingmaschinen
Die Zustände eines Turingprogramms nummeriert man durch. Der Anfangszustand erhält die Nummer 1, der Endzustand die höchste auftretende Nummer, welche n sei. Die Zustände werden dann wie folgt als Folge von Nullen kodiert:

$z_1 = 0$ $z_1 = z_a,\ z_n = z_e$
$z_2 = 00$
$z_3 = 000$
$z_4 = 0000$
...
$z_i = 0^i$ $1 <= i <= n$

In analoger Weise geht man bei den Alphabetzeichen vor. Man nummeriert sie der Reihe nach durch und kodiert sie ebenfalls als Folge von Nullen.

a_1 = * = 0
a_2 = 1 = 00
a_3 = # = 000
...
$a_j = ... = 0^j$ $1 <= j <= n$

Die drei möglichen Kopfbewegungen werden wie folgt kodiert: L = 0 K = 00 R = 000

Ein einzelner Befehl einer Turingmaschine besteht aus fünf Bestandteilen, die separat zu kodieren sind. Man setzt dann die dabei entstandenen Kodierungen zusammen, wobei je zwei Kodierungen durch eine 1 getrennt werden.

Beispiel: Der Befehl A * -> 1 R A wird durch 010100100010 kodiert.

Die Kodierung einer Turingmaschine setzte sich aus den Kodierungen aller m Befehle zusammen, wobei code(i) für die Kodierung des i-ten Befehls steht.

111 code(1) 11 code(2) 11 ... 11 code(m) 111

Interpretiert man diese Kodierung als Dualzahl und konvertiert die Dualzahl in das Zehnersystem, so erhält man die Gödelnummer der Turingmaschine.

a) Kodieren Sie für die Turingmaschine zur Berechnung der Vorgängerfunktion:
A * → * K E
A 1 → * R E
die Befehlsbestandteile, die Befehle, das Turingprogramm. Drücke die Codenummer des Turingprogramms als Summe von Zweierpotenzen aus und rechnen Sie dann die Codenummer in eine Dezimalzahl um.

b) Durch die Gödelnummer 64500286247495 mit der Binärdarstellung 1110101010100110100 1001010011001010010010000111 wird eine Turingmaschine kodiert, deren Programm dekodiert werden soll. Beschreiben Sie die Wirkungsweise der zur Gödelnummer gehörenden Turingmaschine.

c) Ermitteln Sie die kleinste Dezimalzahl, welche Gödelnummer einer Turingmaschine ist.

d) Schreiben Sie eine TM, die prüft, ob eine Binärzahl Gödelnummer einer TM ist.

21 Parser, Interpreter und Compiler

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.

21.1 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:

$$S \rightarrow I\ S \mid \varepsilon$$

Im Syntaxdiagramm stellt sich diese Grammatik so dar:

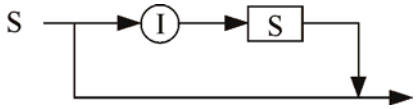


Abb. 21-1 rekursives Syntaxdiagramm für Strichlisten

Wer lieber in Iterationen denkt, der baut sich eher ein äquivalentes Syntaxdiagramm mit einer Schleife:

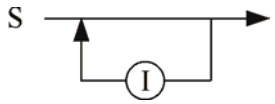


Abb. 21-2 iteratives Syntaxdiagramm für Strichlisten

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 die 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).
parse([]).
```

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*:

```
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:

$$S \rightarrow V\ S \mid E$$
$$E \rightarrow \varepsilon \mid I \mid II \mid III \mid IIII$$

Wir bilden die entsprechenden Syntaxdiagramme:

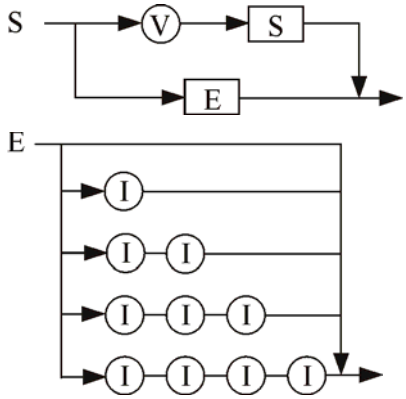


Abb. 21-3 Syntaxdiagramme der erweiterten Strichlisten-Grammatik

Jede Grammatikregel entspricht einem Syntaxdiagramm und jedes Syntaxdiagramm einem *parse*-Prädikat:

```
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).
```

21.2 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.

Die Syntax unserer Sprache *mini-LOGO* definieren wir durch diese beiden Syntaxdiagramme:

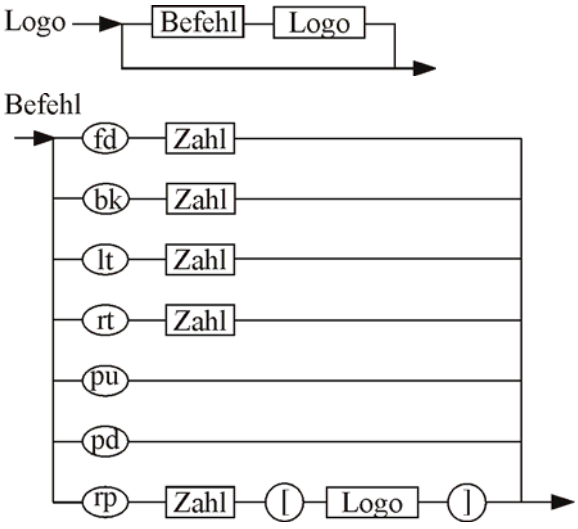


Abb. 21-4 Syntaxdiagramme von mini-LOGO

Im Unterschied zu den bisherigen Parsern soll der Parser für *mini-LOGO* einen Parsebaum für die anschließende Interpretation erstellen. Die Struktur des Parsebaums legen wir wie folgt fest. Ein Parsebaum ist eine Liste aus mini-LOGO-Befehlen. Die Elemente der Liste sind die Atome *pu* und *pd* bei den parameterlosen Befehlen. Befehle mit einem Parameter werden als Strukturen mit dem Parameter als Argument dargestellt, also zum Beispiel *fd(50)* oder *lt(30)*. Beim *repeat*-Befehl werden außer der Anzahl der Wiederholungen noch die zu wiederholenden Befehle benötigt. Diese werden in einem zweiten Argument als Liste angegeben. Ein *rp*-Befehl im Parsebaum wird also in der Form *rp(Anzahl, Befehle)* dargestellt.

Die beiden Syntaxdiagramme lassen sich leicht in die beiden zugehörigen Parse-Prädikate *parse_logo1* und *parse_befehl* umsetzen. *parse_logo1* arbeitet nach der Kopf-Rest-Methode das Quell-Programm ab und baut den Parsebaum auf. Die Alternativen im Syntaxdiagramm von *Befehl* werden zu Alternativen in den *parse_befehl* Klauseln. Etwas Probleme bereitet die Verarbeitung der Eingabe, da der *rp*-Befehl sehr viele Symbole der Eingabeliste umfassen kann. Wir brauchen daher nicht nur die Liste der Eingabesymbole, sondern immer noch eine zweite Liste, welche die restlichen Symbole nach einer Parseoperation enthält. Mit dieser zweiten Liste wird dann der Parsevorgang fortgesetzt. Der Aufruf von *parse_logo1* in parse_logo/2 mit der leeren Liste sorgt dafür, dass nur Logo-Programme akzeptiert werden, die komplett geparst werden können.

```
% LOGO-Parser
parse_logo(Logo, Parsebaum):-
  parse_logo1(Logo, Parsebaum, []).

% parse_logo1(+Eingabe, -Parsebaum,
              -RestlicheEingabe)
parse_logo1(Liste1, [Befehl|Liste2],
            Liste4):-
  parse_befehl(Liste1, Befehl, Liste3),
  parse_logo1(Liste3, Liste2, Liste4).
parse_logo1(Liste, [], Liste).

% parse_befehl(+Eingabe, -Befehl,
                -RestlicheEingabe)
parse_befehl([fd, Zahl|Rest], Befehl,
             Rest):-
  integer(Zahl),
  Befehl = fd(Zahl).

parse_befehl([bk, Zahl|Rest], Befehl,
             Rest):-
  integer(Zahl),
  Befehl = bk(Zahl).

parse_befehl([lt, Zahl|Rest], Befehl,
             Rest):-
  integer(Zahl),
 Befehl = lt(Zahl).

parse_befehl([rt, Zahl|Rest], Befehl,
             Rest):-
  integer(Zahl),
  Befehl = rt(Zahl).

parse_befehl([pd|Rest], pd, Rest).

parse_befehl([pu|Rest], pu, Rest).

parse_befehl([rp, Zahl, '[' | Rest1],
             rp(Zahl,Befehle), Rest3):-
  integer(Zahl),
  Zahl >= 0,
  parse_logo1(Rest1, Befehle, Rest2),
  Rest2 = [']' | Rest3].
```

Als Beispiel betrachten wir den erzeugten Parsebaum zum mini-LOGO Programm:

```
?- parse_logo([rp, 8, '[', rp, 8, [, lt,
45, fd, 30, ], lt, 45, ']'], PB).

PB=[rp(8,[rp(8,[lt(45),fd(30)]), lt(45)])]
```

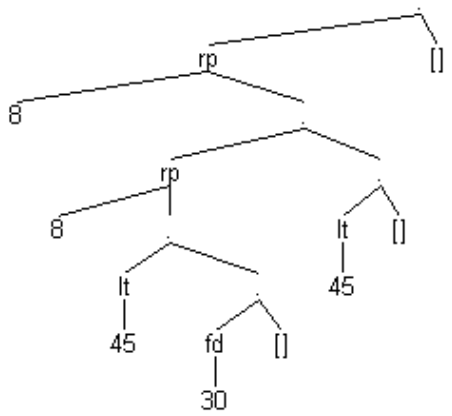


Abb. 21-5 Beispiel eines Parsebaums

Da der Parser einen Parsebaum aufbaut, in dem die Strukturinformation des eingegebenen mini-LOGO-Programms enthalten ist, lässt sich der Interpreter problemlos implementieren. Allerdings braucht man eine grafische Schnittstelle, die Befehle zur Grafikdarstellung bereitstellt. Wir benutzen das Turtle-Modul auf der Basis der grafischen Erweiterung XPCE von SWI-Prolog. Es enthält folgende Prädikate:

% Turtle initialisieren.
`turtle_init`

% Turtle beenden.
`turtle_term`

% zum Punkt P(X/Y) eine Linie zeichnen.
`turtle_drawto(X, Y)`

% eine Strecke der Länge L in die aktuelle Richtung zeichnen.
`turtle_draw(L)`

% um den Winkel W relativ drehen.
`turtle_turn(W)`

% zu einem absoluten Winkel W hindrehen.
`turtle_turnto(W)`

% eine Strecke der Länge L in aktueller Richtung ohne Zeichnen bewegen.
`turtle_move(L)`

% zum Punkt P(x/y) ohne Zeichnen bewegen.
`turtle_moveto(X, Y)`

Um Logo-Programme auf der Basis des Turtle-Moduls interpretieren zu können, merken wir uns im Fakt *penpos*, ob der Zeichenstift unserer Logo-Turtle oben oder unten ist.

```
% Logo-Interpreter
logo(Befehle):-
  parse_logo(Befehle, Parsebaum),
  turtle_init,
  asserta(penpos(down)),
  interpret_logo(Parsebaum).

% Befehlsliste
interpret_logo([Befehl|Befehle]):-
  interpret_befehl(Befehl),
  interpret_logo(Befehle).
interpret_logo([]).

% Forward - fd
interpret_befehl(fd(Zahl)):-
  (  penpos(down)
  -> turtle_draw(Zahl)
  ;  turtle_move(Zahl)
  ).

% Back - bk
interpret_befehl(bk(Zahl)):-
  Zahl1 is -Zahl,
  (  penpos(down)
  -> turtle_draw(Zahl1)
  ;  turtle_move(Zahl1)
  ).

% Left - lt
interpret_befehl(lt(Zahl)):-
  turtle_turn(Zahl).

% Right - rt
interpret_befehl(rt(Zahl)):-
  Zahl1 is -Zahl,
  turtle_turn(Zahl1).

% Penup - pu
interpret_befehl(pu):-
  retract(penpos(_)),
  asserta(penpos(up)).

% Pendown - pd
interpret_befehl(pd):-
  retract(penpos(_)),
  asserta(penpos(down)).

% 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, 8, '[', rp, 8, '[', lt, 45,
fd, 30, ']', lt, 45, ']']).
```

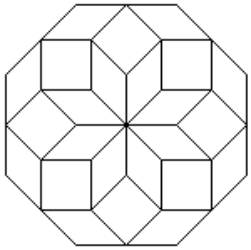


Abb. 21-6 Interpretation eines mini-LOGO-Programms

Die Eingabe eines mini-LOGO-Programms ist etwas umständlich, weil es in Form einer aufbereiteten Liste angegeben werden muss. Ein sogenannter *Scanner* übernimmt die Aufgabe ein mini-LOGO-Programm als Liste für den Parser aufzubereiten. Da mini-LOGO eine sehr einfache Sprache ist, kommen wir mir dem System-Prädikat *readln*/1 aus. Mit readln/1 können wir ein mini-LOGO-Programm über die Tastatur einlesen. Readln liefert die benötigte Liste als Eingabe für den Parser:

```
?- readln(X), logo(X).
```

Wie bei Programmiersprachen üblich können mini-LOGO-Quelltexte auch mit einem Editor geschrieben werden. Um aus einer Datei einzulesen benutzt man den Befehl *see*. Nach *see(Dateiname)* liest *readln* nicht mehr von der Tastatur, sondern aus der Datei. Nach dem Einlesen schließt man mit *seen* wieder die Datei.

```
logodatei(Dateiname):-
  see(Dateiname),
  readln(X),
  logo(X),
  seen.

?- logodatei('logo1.log').
```

21.3 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 folgender Grammatik fest:

Programm →
  program Bezeichner; begin Anweisungen end .

Anweisungen →
  Anweisung | Anweisung ; Anweisungen

Anweisung →
  Variable := Ausdruck |
  while Ausdruck do Anweisung |
  begin Anweisungen end |
  write (Ausdruck) | ε

Ausdruck →
  Einfacher_Ausdruck |
  Einfacher_Ausdruck < Einfacher_Ausdruck

Einfacher_Ausdruck →
  Term |
  Einfacher_Ausdruck + Term |
  Einfacher_Ausdruck – Term

Term →
  Faktor | Term * Faktor | Term / Faktor

Faktor →
  (Ausdruck) | Zahl | Variable

Variable → Bezeichner

Die zweite Produktion drückt aus, dass Anweisungen durch ein Semikolon getrennt werden. Im Unterschied zu C oder Java muss also nicht jede einzelne Anweisung mit einem Semikolon abgeschlossen werden. Mini-Pascal kennt nur fünf Anweisungen, darunter auch die leere Anweisung ε. Es ist eine produktive Aufgabe weitere Anweisungen zu ergänzen. Auch bei den Ausdrücken geht es spartanisch zu, lediglich der Vergleichsoperator < ist zugelassen. Bei den arithmetischen Ausdrücken wird Linksrekursion benutzt, damit die mathematische Auswertungsregel „von links nach rechts“ beachtet wird. Sie sorgt dafür, dass z. B. 5 – 6 – 7 nur als (5 – 6) – 7 abgeleitet werden kann. Die ausführlichen Produktionen für Bezeichner und Zahlen ersparen wir uns hier. Bezeichner müssen mit einem Buchstaben beginnen, als Zahlen sind Integer-Zahlen zugelassen. Mini-Pascal kommt ohne Variablendeklaration aus, ähnlich wie in einigen Basic-Dialekten werden Variablen implizit durch die Benutzung definiert. Das geht unproblematisch, weil wir nur Variablen des Datentyps Integer benutzen.

Anschaulich wird die Grammatik durch folgende Syntaxdiagramme beschrieben:

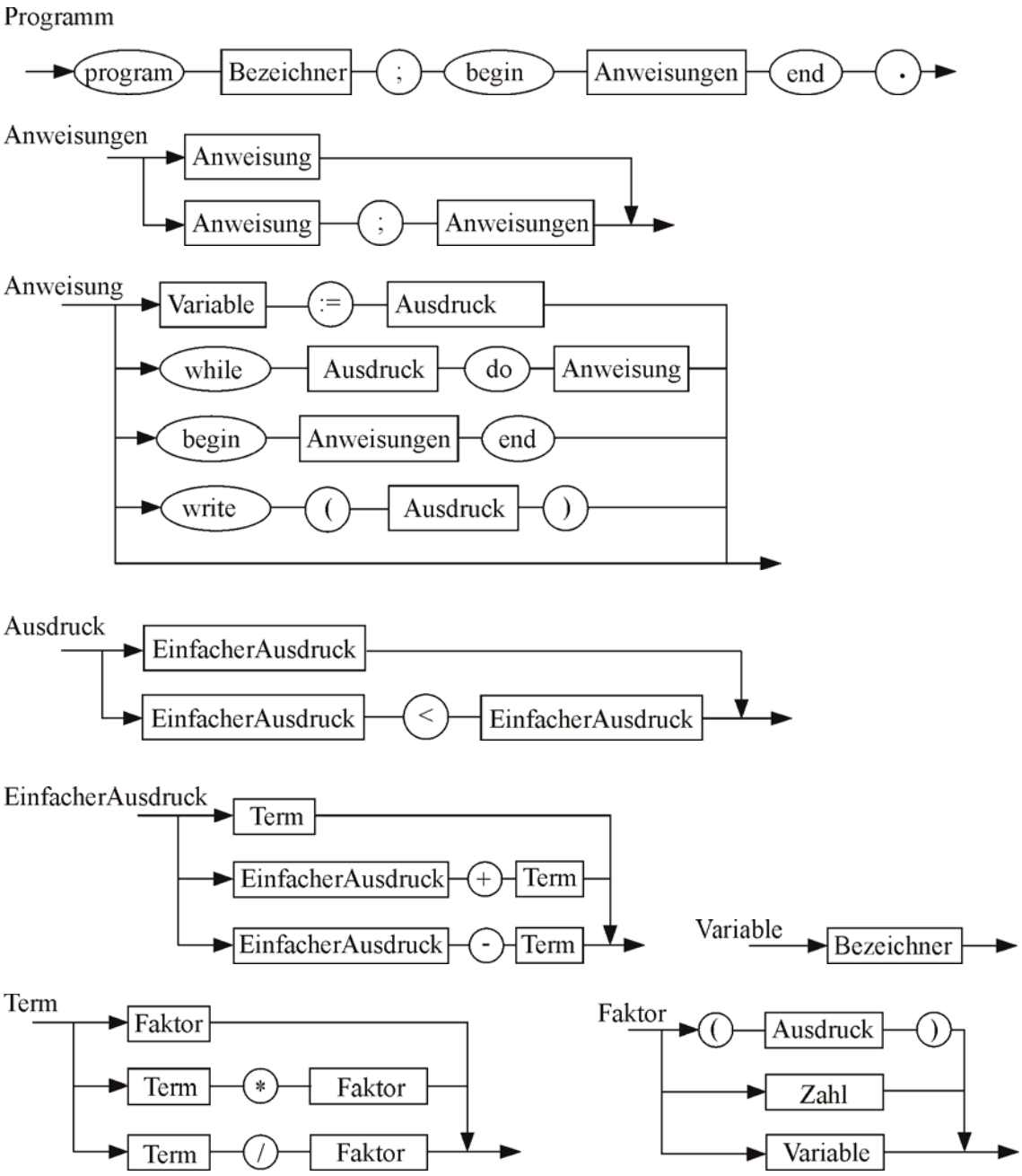


Abb. 21-7 Syntaxdiagramme von mini-PASCAL

Das folgende Programm zur Berechnung von 7! = 1*2*3*4*5*6*7 ist ein mini-Pascal-Programm:

```
program Fakultaet;
begin
  n:= 7;
  i:= 1;
  Fak:= 1;
  while i < n do begin
    i:= i + 1;
    Fak:= Fak * i
  end;
  write(Fak);
end
```

21.3.1 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 aufeinander folgenden 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.

Das mühselige Zusammensetzen von Ziffern zu Zahlen und Buchstaben zu Bezeichnern, de-

legieren wir an das Systemprädikat *readln/5*. 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, GescannteSymbole),
  retractall(symbole(_)),
  assert(symbole(GescannteSymbole)).
```

Einige Vergleichsoperatoren werden anschließend vom *scan*-Prädikat zusammengesetzt, Bezeichner werden zur Unterscheidung von anderen Symbolen und von reservierten Schlüsselwörtern in einstellige *bez*-Strukturen verpackt:

```
% Symbole bilden
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([], []).

% Operatoren zusammen setzen
zusammensetzen(<, =, <=).
zusammensetzen(<, >, <>).
zusammensetzen(>, =, >=).
zusammensetzen(:, =, :=).

% Bezeichner verpacken
verpacken(Symbol, bez(Symbol)):-
  atom_chars(Symbol, [K|_]),
  is_alpha(K),
  not(reserviert(Symbol)).
verpacken(Symbol, Symbol).

% reservierte Schlüsselwörter
reserviert(X):-
  member(X, [program, begin, end,
             while, do, write]).
```

Das Ergebnis wird mit *assert* in der Wissensbasis als Fakt *symbole(GescannteSymbole)* abgelegt. Im Unterschied zur Lösung von mini-LOGO muss in den parse-Klauseln dann nicht mit Eingabe- und Restlisten gearbeitet werden. Stattdessen stellt der Scanner dem Parser auf Anforderung das aktuelle Symbol zur Verarbeitung bereitstellen und erlaubt es zum nächsten Symbol bei Bedarf weiterzuschalten. Die erste Funktion erledigt das Prädikat *symbol*, das zweite das Prädikat *next_symbol*.

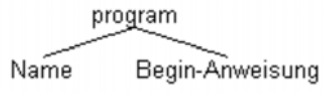
```
symbol(Symbol):-
  symbole([Symbol|_]).
next_symbol:-
  retract(symbole([_|Symbole])),
  assert(symbole(Symbole)).
```

21.3.2 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. Zum Parsen von Anweisungen setzt man Parse-Schablonen ein, welche die Struktur geparster Anweisungen beschreiben. Folgende Parse-Schablonen werden benutzt:

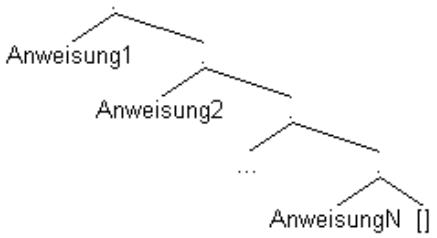
Gesamtes Programm:

```
program(Name, Begin-Anweisung)
```



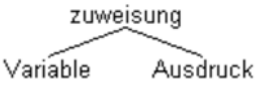
Sequenz von Anweisungen:

```
[Anweisung1, Anweisung2,...]
```



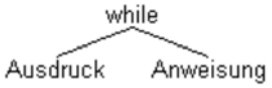
Wertzuweisung:

```
zuweisung(Variable, Ausdruck)
```



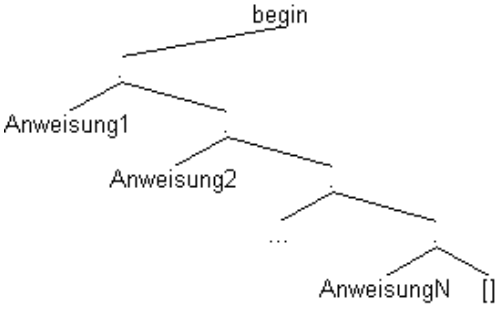
While-Anweisung:

```
while(Ausdruck, Anweisung)
```



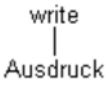
Begin-Anweisung:

```
begin(Liste der Anweisungen)
```



Write-Anweisung:

```
write(Ausdruck)
```

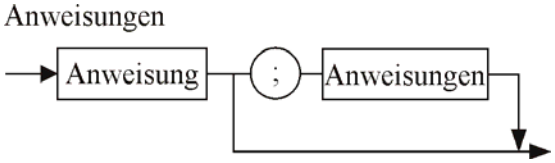


Die leere Anweisung wird durch das Atom *nil* repräsentiert.

Anweisungen können leicht geparst werden:

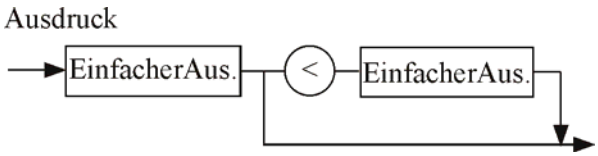
```
parse(Datei, Parsebaum):-
  init_scanner(Datei),
  parse_programm(Parsebaum).

parse_programm(Programm):-
  parse_symbol(program),
  parse_bezeichner(Name),
  parse_symbol(;),
  parse_anweisung(begin(Anweisungen)),
  parse_symbol(.),
  Programm = program(Name,
               begin(Anweisungen)).

parse_anweisungen([Anweisung|Anweisungen]):-
  parse_anweisung(Anweisung),
  (  parse_symbol(;)
  -> parse_anweisungen(Anweisungen)
  ;  Anweisungen = []
  ).

parse_anweisung(Zuweisung):-
  parse_bezeichner(Variable), !,
  parse_symbol(:=),
  parse_ausdruck(Ausdruck),
  Zuweisung = zuweisung(Variable,Ausdruck).

parse_anweisung(WhileAnweisung):-
  parse_symbol(while), !,
  parse_ausdruck(Ausdruck),
  parse_symbol(do),
  parse_anweisung(Anweisung),
  WhileAnweisung=while(Ausdruck,Anweisung).

parse_anweisung(BeginAnweisung):-
  parse_symbol(begin), !,
  parse_anweisungen(Anweisungen),
  parse_symbol(end),
  BeginAnweisung = begin(Anweisungen).

parse_anweisung(WriteAnweisung):-
  parse_symbol(write), !,
  parse_symbol('('),
  parse_ausdruck(Ausdruck),
  parse_symbol(')'),
  WriteAnweisung = write(Ausdruck).

parse_anweisung(nil).
```

Effizientes Parsen erfolgt möglichst ohne Backtracking. Unsere Art der Symbolverarbeitung des Scanners ist darauf schon abgestimmt, denn ein einmal gelesenes Symbol steht nicht mehr für Backtracking zur Verfügung. Aus diesem Grund ist das Parse-Prädikat nicht nach dem bisherigen Syntaxdiagramm, sondern nach diesem äquivalenten Syntaxdiagramm implementiert:



Zunächst wird die erste Anweisung geparst. Wenn dann ein Semikolon folgt werden die restlichen Anweisungen geparst, ansonsten terminiert *parse_anweisungen* mit der bisher geparsten Anweisungsliste.

Beim Parsen eines *Ausdrucks* haben wir die analoge Situation wie bei den Anweisungen. Nach dem Parsen eines *EinfachenAusdrucks* müssen wir zur Vermeidung von Backtracking zuerst prüfen, ob das Kleiner-Symbol kommt. Wir gehen also nach diesem äquivalenten Syntaxdiagramm vor:



Die Fallunterscheidung hinter dem ersten einfachen Ausdruck nehmen wir nicht mit dem if-then-else Operator `->` vor, sondern erledigen dies durch separate Klauseln. Dies erleichtert die spätere Erweiterung auf zusätzliche Operatoren.

```
parse_ausdruck(Ausdruck):-
  parse_einfacher_ausdruck(Ausdruck1),
  parse_ausdruck(Ausdruck1, Ausdruck).
parse_ausdruck(Ausdruck1, Ausdruck):-
  parse_symbol(<), !,
  parse_einfacher_ausdruck(Ausdruck2),
  Ausdruck = (Ausdruck1 < Ausdruck2).
parse_ausdruck(Ausdruck1, Ausdruck):-
  Ausdruck = Ausdruck1.
```

Die einfachen Ausdrücke und Terme sind aufgrund der linksrekursiven Produktionen etwas schwieriger in backtrackingfreie parse-Klauseln umzusetzen. Zunächst formen wir die linksrekursiven Produktionen in äquivalente rechtsrekursive Produktionen um. Wir erläutern das Umformungsverfahren anhand der zwei Produktionen

$$A \rightarrow \alpha \mid A\beta$$

wobei A eine Variable und α sowie β Folgen von Terminalen und Variablen seien, die nicht mit A beginnen. Aus den beiden Produktionen lassen sich die Wörter α, $\alpha\beta$, $\alpha\beta\beta$, $\alpha\beta\beta\beta$,

αβββββ usw. ableiten. Gleiches lässt sich durch die drei folgenden Produktionen erreichen:

$A \rightarrow \alpha R$
$R \rightarrow \beta R \mid \varepsilon$

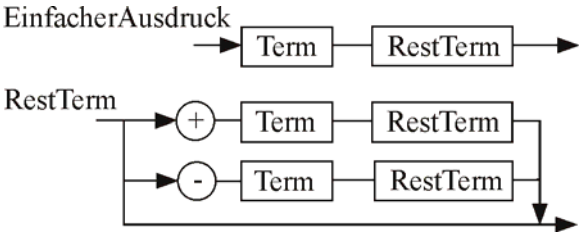
Dabei ist R eine neue Variable und $R \rightarrow \beta R$ eine rechtsrekursive Produktion. Wenden wir diese Idee auf die Produktionen

Einfacher_Ausdruck →
Term | Einfacher_Ausdruck + Term

an, so erhalten wir mit den Substitutionen A = Einfacher_ Ausdruck, α = Term und β = + Term bzw. β = - Term die Produktionen

Einfacher_Ausdruck → Term RestTerm
RestTerm → + Term RestTerm | ε bzw.
RestTerm → - Term RestTerm | ε

Die Syntaxdiagramme der so umgeformten Produktionen sehen wie folgt aus:



Nach diesen Vorbereitungen können die neuen Syntaxdiagramme in äquivalente Parse-Prädikate transformiert werden:

```
parse_einfacher_aus(Einf_Aus):-
  parse_term(Term),
  parse_restterm(Term, Einf_Aus).
parse_restterm(Term1, Einf_Aus):-
  parse_symbol(+), !,
  parse_term(Term2),
  parse_restterm(Term1 + Term2, Einf_Aus).
parse_restterm(Term1, Einf_Aus):-
  parse_symbol(-), !,
  parse_term(Term2),
  parse_restterm(Term1 - Term2, Einf_Aus).
parse_restterm(Einf_Aus, Einf_Aus).
```

Analog dazu muss das Syntaxdiagramm für Terme ungewandelt und dann umgesetzt werden. Wir erhalten:

```
parse_term(Term):-
  parse_faktor(Faktor),
  parse_restfaktor(Faktor, Term).
parse_restfaktor(Faktor1, Term):-
  parse_symbol(*), !,
  parse_faktor(Faktor2),
  parse_restfaktor(Faktor1*Faktor2, Term).
parse_restfaktor(Faktor1, Term):-
  parse_symbol(/), !,
  parse_faktor(Faktor2),
  parse_restfaktor(Faktor1/Faktor2, Term).
parse_restfaktor(Term, Term).
```

Übrig bleiben einige einfache Parse-Klauseln:

```
parse_faktor(Faktor):-
  parse_symbol('('),
  parse_ausdruck(Faktor),
  parse_symbol(')').

parse_faktor(Faktor):-
  parse_zahl(Faktor).

parse_faktor(Variable):-
  parse_bezeichner(Variable).

parse_zahl(Zahl):-
  symbol(Zahl),
  integer(Zahl),
  next_symbol.

parse_bezeichner(Bezeichner):-
  symbol(bez(Bezeichner)),
  next_symbol.

parse_symbol(Symbol):-
  symbol(Symbol),
  next_symbol.
```

21.3.3 Interpreter für mini-PASCAL

Die Interpretation eines mini-Pascal-Programms ist eine einfache Angelegenheit, da die eigentlich schwierige Arbeit, der Aufbau des Parsebaums schon 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.

```
interpretiere(Datei):-
  parse(Datei, Parsebaum),
  retractall(speicher(_, _)),
  interpret_program(Parsebaum), !,
  listing(speicher).

interpret_program(program(Name,
                  BeginAnweisung)):-
  write('Interpretation von: '),
  writeln(Name),
  interpret_anweisung(BeginAnweisung).

interpret_anweisungen([Anwei|Anweis]):-
  interpret_anweisung(Anwei),
  interpret_anweisungen(Anweis).
interpret_anweisungen([]).

interpret_anweisung(zuweisung(Var, Aus)):-
  interpret_ausdruck(Aus, Wert),
  retractall(speicher(Var, _)),
  assert(speicher(Var, Wert)).
```

```
interpret_anweisung(while(Aus, Anwei)):-
  interpret_ausdruck(Aus, true), !,
  interpret_anweisung(Anwei),
  interpret_anweisung(while(Aus, Anwei)).
interpret_anweisung(while(_, _)).

interpret_anweisung(begin(Anweis)):-
  interpret_anweisungen(Anweis).

interpret_anweisung(write(Ausdruck)):-
  interpret_ausdruck(Ausdruck, Wert),
  writeln(Wert).

interpret_anweisung(nil).

interpret_ausdruck(Aus1 < Aus2, true):-
  interpret_ausdruck(Aus1, Wert1),
  interpret_ausdruck(Aus2, Wert2),
  Wert1 < Wert2.
interpret_ausdruck(_Aus1 < _Aus2, false).

interpret_ausdruck(Aus + Aus2, Wert):-
  interpret_ausdruck(Aus1, Wert1),
  interpret_ausdruck(Aus2, Wert2),
  Wert is Wert1 + Wert2.

interpret_ausdruck(Aus1 - Aus2, Wert):-
  interpret_ausdruck(Aus1, Wert1),
  interpret_ausdruck(Aus2, Wert2),
  Wert is Wert1 - Wert2.

interpret_ausdruck(Aus1 * Aus2, Wert):-
  interpret_ausdruck(Aus1, Wert1),
  interpret_ausdruck(Aus2, Wert2),
  Wert is Wert1 * Wert2.

interpret_ausdruck(Aus1 / Aus2, Wert):-
  interpret_ausdruck(Aus1, Wert1),
  interpret_ausdruck(Aus2, Wert2),
  Wert is Wert1 / Wert2.

interpret_ausdruck(Variable, Wert):-
  speicher(Variable, Wert).

interpret_ausdruck(Zahl, Zahl):-
  integer(Zahl).
```

21.3.4 Compiler für mini-PASCAL

Der Interpreter kann in einen Compiler umgeschrieben werden, der ein mini-PASCAL-Programm in die Assemblersprache von Intel-Prozessoren übersetzt. Für die Übersetzung von Anweisungen und Ausdrücken benutzen wir Code-Schablonen. Beispielsweise wird eine Begin-Anweisung in eine Sequenz der compilierten Einzelanweisungen übersetzt. Bei einer write-Anweisung wird zunächst der auszugebende Ausdruck compiliert und dann das Ergebnis ausgegeben. Für Ausdrücke legen wir grundsätzlich fest, dass das Ergebnis im Akkumulator AX abgelegt wird. Ähnlich wie die write-Anweisung funktioniert die Übersetzung der Wertzuweisung. Zunächst wird der Ausdruck compiliert, wodurch nach Vereinbarung das Ergebnis dann im Akkumulator-Register AX steht. Anschließend erhält die Variable im Speicher den Akkumulatorinhalt als Wert zugewiesen.

Für die Übersetzung von Kontrollstrukturen braucht man zusätzlich Sprungmarken und Sprünge. Ein anschauliches Flussdiagramm hilft bei der Konstruktion der Code-Schablone:

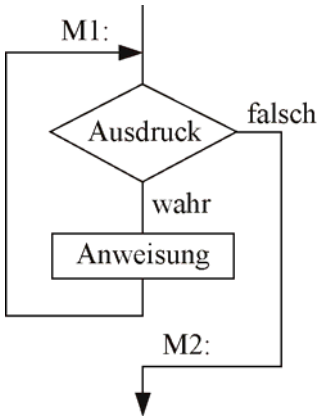


Abb. 21-8 Flussdiagramm für eine While-Anweisung

Zuerst wird der Ausdruck übersetzt und dann mit *true* verglichen. Ist der Ausdruck falsch, erfolgt ein bedingter Sprung zur Marke M2 am Ende der Codeschablone. Anderenfalls wird die zu wiederholende Anweisung compiliert. Zum Abschluss folgt ein unbedingter Sprung zur Marke M1 an den Anfang der While-Anweisung. Daraus ergibt sich folgende Code-Schablone für die While-Anweisung:

```
M1:
    <Code für den Ausdruck>
    cmp  ax, true
    jne  M2   ; bedingter Sprung
    <Code für die Anweisung>
    jmp  M1   ; unbedingter Sprung
M2:
```

Die Verwaltung der Sprungmarken delegieren wir an die Prädikate *compile_programm* und *gib_ marke*:

```
compiliere(Datei):-
  parse(Datei, Parsebaum),
  retractall(marke(_)),
  assert(marke(1)),
  compile_program(Parsebaum).

gib_marke(Marke):-
  retract(marke(Marke)),
  Marke1 is Marke + 1,
  assert(marke(Marke1)).
```

Damit lassen sich die Anweisungen übersetzen:

```
compile_program(program(Name,
                BeginAnweisung)):-
  write('Compiliere das Programm: '),
  writeln(Name),
  compile_anweisung(BeginAnweisung).

compile_anweisungen([Anwei|Anweis]):-
  compile_anweisung(Anwei), !,
  compile_anweisungen(Anweis).
compile_anweisungen([]).

compile_anweisung(zuweisung(Var, Aus)):-
  compile_ausdruck(Aus),
  schreibnl('mov  [', Var, '], ax').

compile_anweisung(while(Aus, Anwei)):-
  gib_marke(Marke1),
  schreib_marke(Marke1),
  compile_ausdruck(Aus),
  schreibnl('cmp  ax, true'),
  gib_marke(Marke2),
  schreibnl('jne  M', Marke2),
  compile_anweisung(Anwei),
  schreibnl('jmp  M', Marke1),
  schreib_marke(Marke2).

compile_anweisung(begin(Anweisungen)):-
  compile_anweisungen(Anweisungen).

compile_anweisung(write(Ausdruck)):-
  compile_ausdruck(Ausdruck),
  schreibnl('out 0, 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. Bei binären Ausdrücken ist es oft sinnvoll erst den zweiten Ausdruck zu berechnen, diesen zu kellern um dann den ersten Ausdruck direkt in das AX-Register compilieren zu können.

```
compile_ausdruck(Aus1 < Aus2):-
  compile_ausdruck(Aus2),
  schreibnl('push ax'),
  compile_ausdruck(Aus1),
  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).

compile_ausdruck(Aus1 + Aus2):-
  compile_ausdruck(Aus2),
  schreibnl('push ax'),
  compile_ausdruck(Aus1),
  schreibnl('pop  cx'),
  schreibnl('add  ax, cx').

compile_ausdruck(Aus1 - Aus2):-
  compile_ausdruck(Aus2),
  schreibnl('push ax'),
  compile_ausdruck(Aus1),
  schreibnl('pop  cx'),
  schreibnl('sub  ax, cx').

compile_ausdruck(Aus1 * Aus2):-
  compile_ausdruck(Aus2),
  schreibnl('push ax'),
  compile_ausdruck(Aus1),
  schreibnl('pop  cx'),
  schreibnl('imul cx').

compile_ausdruck(Aus1 / Aus2):-
  compile_ausdruck(Aus2),
  schreibnl('push ax'),
  compile_ausdruck(Aus1),
  schreibnl('pop  cx'),
  schreibnl('idiv cx').

compile_ausdruck(Variable):-
  integer(Variable),
  schreibnl('mov  ax, ', Variable).

compile_ausdruck(Variable):-
  schreibnl('mov  ax, [', Variable, ']').
```

Einen schönen Ausdruck besorgen die *schreibnl*- und das *schreibmarke*-Prädikate.

```
schreibnl(Ausgabe):-
  tab(4), writeln(Ausgabe).

schreibnl(Ausgabe1, Ausgabe2):-
  tab(4), write(Ausgabe1),
  writeln(Ausgabe2).

schreibnl(Ausgabe1,Ausgabe2,Ausgabe3):-
  tab(4), write(Ausgabe1),
  write(Ausgabe2), writeln(Ausgabe3).

schreib_marke(Marke):-
  write('M'), write(Marke),
  writeln(':').
```

21.3.5 Ausführung mit VISIS

Die compilierten Programme können mit einigen kleinen Ergänzungen im Simulationsprogramm VISIS (Visualisierung eines Intel-Systems) ausgeführt werden. Die Konstanten *false* und *true* müssen definiert und alle Variablen im Datensegment .data deklariert werden. Um die Variablen deklarieren zu können, sammeln wir diese in der Wissensbasis über *variable/1*-Fakten an. Variablen werden bei Wertzuweisungen geprüft, ob sie schon deklariert sind oder noch nicht.

```
% Variable existiert schon
compile_anweisung(zuweisung(Var, Aus)):-
  variable(Var),
  compile_ausdruck(Aus),
  schreibnl('mov  [', Var, '], ax').

% neue Variable
compile_anweisung(zuweisung(Var, Aus)):-
  assertz(variable(Var)),
  compile_ausdruck(Aus),
  schreibnl('mov  [', Var, '], ax').
```

Zum Abschluss der Compilierung können dann die Variablen in das Datensegment geschrieben werden.

```
schreibe_daten_segment:-
  retract(variable(Var)),
  schreib_variable(Var),
  schreibe_daten_segment.
schreibe_daten_segment.

schreib_variable(Variable):-
  write(Variable),
  atom_length(Variable, N),
  N1 is 5 - N,
  tab(N1),
  writeln(' dw ?').
```

Die compile-Klausel für das Hauptprogramm kümmert sich um die Konstantendeklaration, das Code- und das Datensegment:

```
compile_program(program(Name,
                BeginAnweisung)):-
  write('Compiliere das Programm: '),
  writeln(Name),
  writeln('false equ 0'),
  writeln('true  equ 1'), nl,
  retractall(variable(_)),
  schreibnl('.code'),
  compile_anweisung(BeginAnweisung),
  schreibnl('hlt'), nl,
  schreibnl('.data'),
  schreibe_daten_segment,
  schreibnl('end').
```

Das so compilierte Programm kann in VISIS geladen und ausgeführt werden. Als Beispiel ist im Folgenden die Übersetzung des Programms zur Berechnung von 7! angegeben:

```
program Fakultaet;
begin
  n:= 7;  i:= 1; Fak:= 1;
  while i < n do begin
    i:= i + 1;
    Fak:= Fak * i
  end;
  write(Fak)
end.

?- compiliere('fakultae.pas').

false equ 0
true  equ 1

    .code
    mov  ax, 7
    mov  [n], ax
    mov  ax, 1
    mov  [i], ax
    mov  ax, 1
    mov  [fak], ax
M1:
    mov  ax, [n]
    push ax
    mov  ax, [i]
    pop  cx
    cmp  ax, cx
    mov  ax, true
    jl   M2
    mov  ax, false
M2:
    cmp  ax, true
    jne  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 cx
    mov  [fak], ax
    jmp  M1
M3:
    mov  ax, [fak]
    out 1, ax
    hlt

    .data
n     dw ?
i     dw ?
fak   dw ?
    end
```

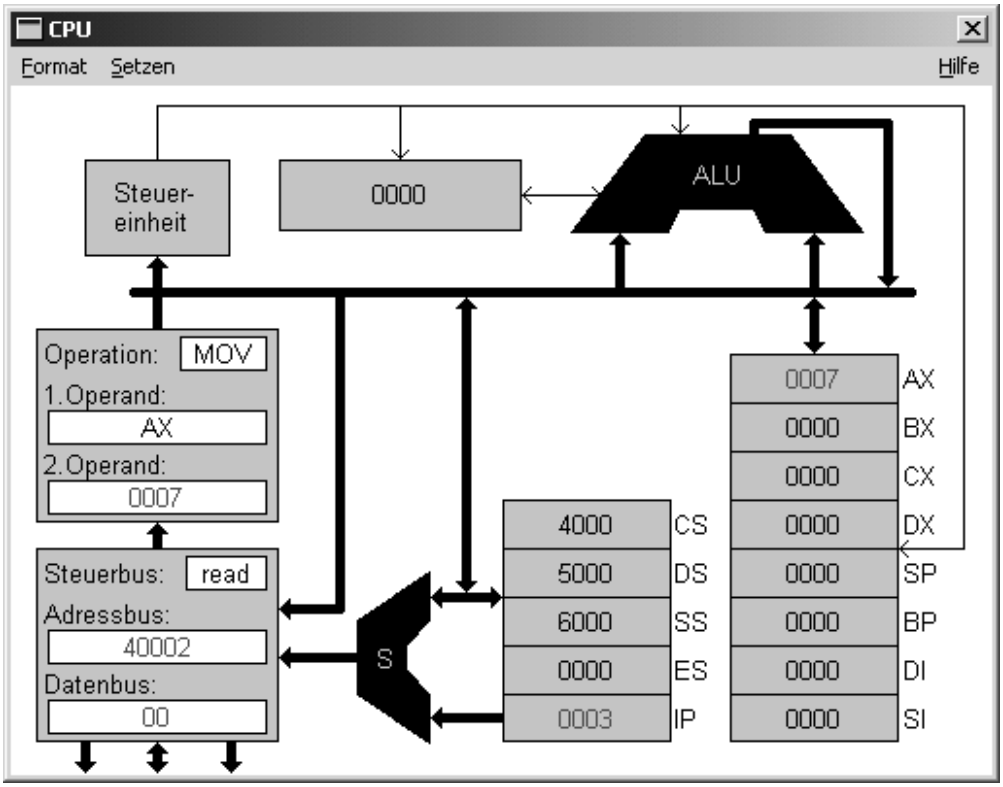


Abb. 21-9 Abarbeitung des Befehls MOV AX, 7 in VISIS

21.4 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 deutlich um weitere Anweisungen und Ausdrücke erweitert werden.
a) In mini-PASCAL sollen Vorzeichen bei Variablen und Zahlen möglich sein.
b) Ergänzen Sie die Fallunterscheidung If-Then-Else.
c) Ergänzen Sie die repeat und for-Schleife.

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. Definieren Sie eine Sprache mini-Java und schreiben Sie dafür einen Parser, Interpreter und Compiler.

5. Sofern ein Plotter zur Verfügung steht: Entwickeln Sie einen Parser und Interpreter für *mini-PLOT*.

6. Sofern Fischertechnik- oder LEGO-Modelle zur Verfügung stehen: Entwickeln Sie einen Parser, Interpreter und Übersetzer für *mini-FISCH* bzw. *mini-LEGO*.

22 Maschinelle Sprachverarbeitung

Wir knüpfen an das in Kapitel 17 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 Interpretationen eines Satzes stattfinden. Wir werden einen Aussagesatz in einen Fragesatz beziehungsweise Nebensatz umformulieren und in den Übungen weitere syntaktische Umformulierungen kennen lernen.

22.1 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.

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.

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.

```
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.

```
ableitbar2([KopfA|RestA],[KopfW|RestW]):-
  produktion(KopfA, [KopfW|Rest1]),
  append(Rest1, RestA, Rest2),
  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.

```
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:

```
ableitbar2([KopfA|RestA], Wort):-
  produktion(KopfA, []),
  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([], []).
```

Wir rufen das Prädikat *ableitbar2/2* über das Prädikat *ableitbar2/1* auf:

```
ableitbar2(Wort):-
  terminalwort(Wort),
  startsymbol(Start),
  ableitbar2([Start], Wort).
```

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 zugrunde liegenden Grammatik G misslingen Ableitungen. Linksrekursionen muss man durch Umformung der Grammatik beseitigen. In Kapitel 21.3.2 haben wir dazu eine einfache Technik kennen gelernt, welche in den meisten Fällen ausreicht.

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('->'),
  writeln(Wort), fail.
```

22.2 Ableitungsbäume und Parser

Ein Akzeptor alleine reicht nicht immer aus. Zur weiteren Verarbeitung eines syntaktisch korrekten Satzes benötigt man zusätzlich dessen grammatische Struktur. Sie wird durch den so genannten *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 zugrunde liegenden Grammatik. Die Anwendung einer Produktion erzeugt die Nachfolgeknoten eines inneren Knotens. Für das Beispiel der 0-1-Wörter aus Kapitel 17.1.4 ist im Bild der Ableitungsbaum für die Ableitung S → 0B → 00BB → 001B → 0011 angegeben.

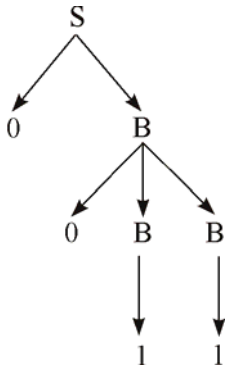


Abb. 22-1 Ableitungsbaum für das Wort 0011

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:

```
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 übrig bleibt wird in Restwort zurückgeliefert.

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)).*

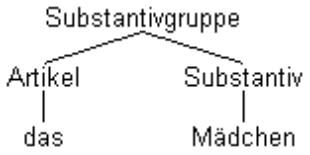


Abb. 22-2 Ableitungsbaum einer Substantivgruppe

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.

Stellen wir mit der Grammatik für arithmetische Terme aus Kapitel 17.1.2 die Anfrage *?- ableitbar3([a,+,b,*,c],B),* so erhalten wir den folgenden Ableitungsbaum als Lösung: B = ausdruck(term(faktor(a)), +, ausdruck (term (faktor(b), *, term(faktor(c))))). Diesen unansehlichen Baum können wir mit *zeichne_term* übersichtlich darstellen:

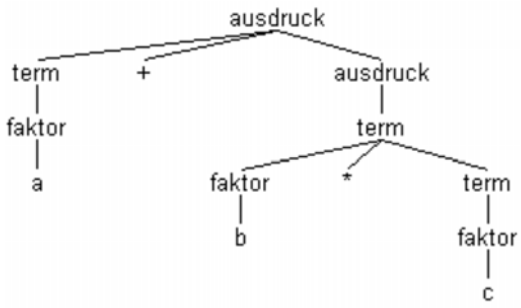


Abb. 22-3 Ableitungsbaum für den Term a+b*c

Die Anfrage *?- ableitbar3(['Peter', kauft, die, 'Kartoffeln'], B)* erzeugt den Ableitungsbaum B = satz(subjekt(eigenname('Peter')), praedikat(verb(kauft)), objekt(akkusativergaenzung (substantivgruppe (artikel (die), substantiv ('Kartoffeln'))))):

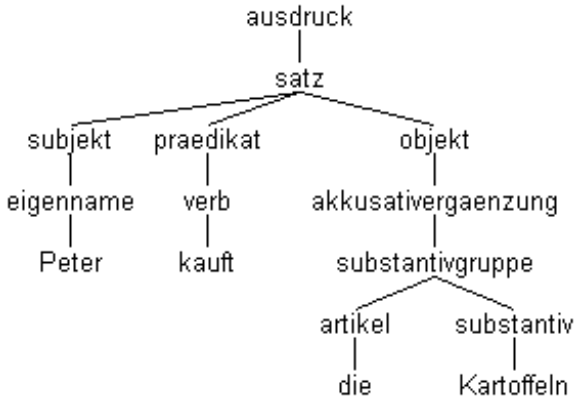


Abb. 22-4 Ableitungsbaum für den Satz *Peter kauft die Kartoffeln*

```
ableitbar3(Wort, Ableitungsbaum):-
  terminalwort(Wort),
  startsymbol(Start),
  ableitbar3([Start], Wort, [], [Ableitungsbaum]).

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, []),
  ableitbar3(RestA, Wort, RestW1, Baum).

ableitbar3([], Wort, Wort, []).
```

22.3 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)),
  writeln('?').

nebensatz(Satz):-
  ableitbar3(Satz, satz(S, P, O)),
  write('..., weil '),
  schreibestruktur(satz(S, O, P)),
  nl.
```

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([]).
```

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 allgemein bildender Schulen [GI1] wird ein Unterrichtsprojekt *Maschinelle Sprachverarbeitung* vorgeschlagen. Die dort angesprochenen 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.

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.

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.

Wollen Sie die Verarbeitung natürlicher Sprache vertiefen, so sollten Sie einen weiteren Zugang in Betracht ziehen. In guten Prolog-Interpretern, wie zum Beispiel 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 unter Verwendung des Grammatik-Operators:

```
substantivgruppe -->
  artikel, substantiv.
artikel -->
  [das]; [die]; [der].
substantiv -->
  ['Buch'];
  ['Mädchen'];
  ['Kartoffeln'].
```

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}.
```

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].

22.4 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 aufeinander getürmt, in den Kasten gelegt oder auf den Tisch zurückgelegt werden. Der Benutzer kann Befehle erteilen oder Fragen stellen.

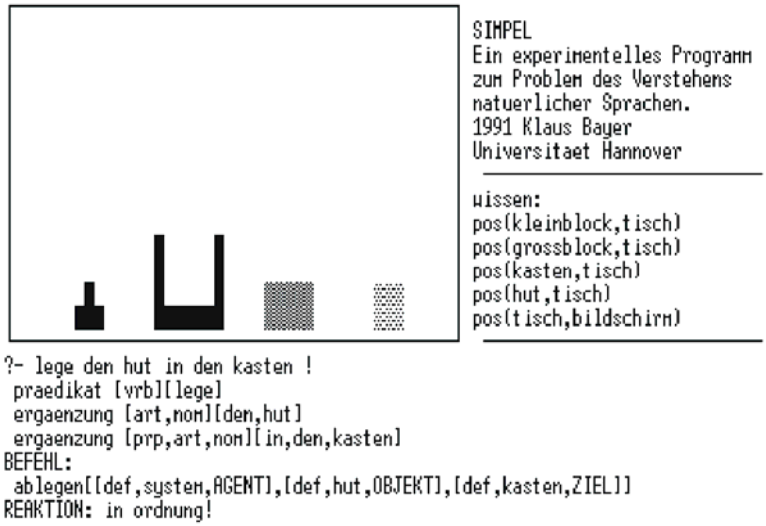


Abb. 22-5 Bildschirmmaske von SIMPEL

Beispiele:

```
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:

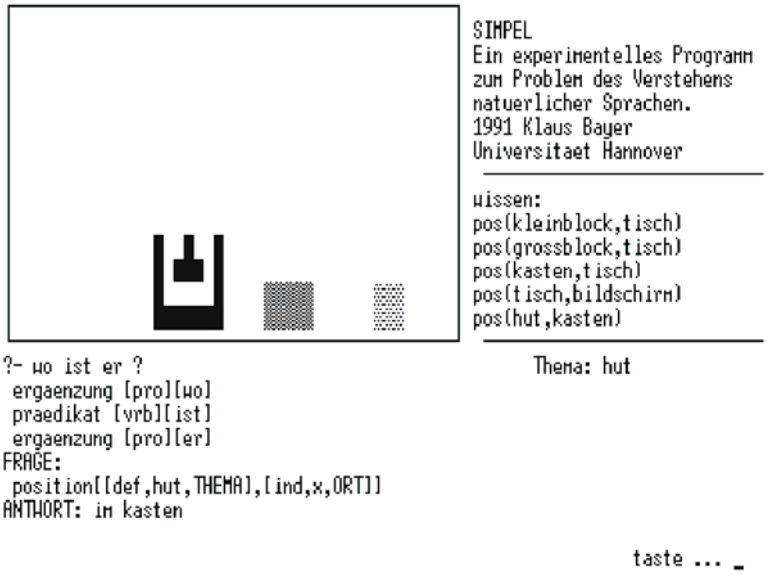
```
SATZ -> PRÄDIKAT + ERGÄNZUNG + ERGÄNZUNG
PRÄDIKAT  -> VERB
ERGÄNZUNG -> ARTIKEL + NOMEN
ERGÄNZUNG -> PRÄPOSITION + ARTIKEL + NOMEN
VERB      -> ist | lege | liegt |
             steht | stelle | hebe
ARTIKEL   -> der | den | dem | ein |
             einem | einen
```

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]).
```

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.
```

Abb. 22-6 Bearbeitung der Frage *Wo ist er?*

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 Programmerweiterungen durchführen, muss man allerdings über solide Prolog-Kenntnisse verfügen. Mit weit weniger Aufwand können Schüler ELIZA-Programme analysieren und Programmergänzungen durchführen.

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.

22.5 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:

```
> 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 bro-
ther.
can you tell me more about father ?
```

Das Verhalten von ELIZA lies 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

eliza:-
  writeln('Hallo, ich bin ELIZA. Wo liegt dein Problem?'),
  eingabe(Eingabe), eliza(Eingabe), !.

eliza([]):-
  writeln('Ich hoffe, dass ich Dir helfen konnte. ').


eliza(Eingabe):-
  muster(Stimulus, Erwiderung),
  vergleiche(Stimulus, Eingabe, [], Woerterbuch),
  write('Eliza: '),
  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,
             [(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(Schluessel, [(Schluessel,  Wert)|_Woerterbuch], Wert).
suche(Schluessel, [(Schluessel1,_Wert)| Woerterbuch], Wert):-
  Schluessel \= Schluessel1,
  suche(Schluessel, Woerterbuch, Wert).
% muster(+Stimulus, -Erwiderung)
% Bestimmt ein auf Stimulus anwendbares Erwiderungsmuster.

muster(['Ich', bin, 1, .], ['Wie lange bist Du schon', 1, ?]).
muster(['Ich', mag, 1, '.'], ['Mag jemand in deiner Familie auch', 1, ?]).
```

```
muster(['Ich', glaube, 1, '.'], ['Glaubts Du oft', 1, ?]).
muster(['Ich', habe, 1, '.'], ['Warum hast Du', 1, ?]).
muster(['Ich', will, 1, '.'], ['Warum willst Du', 1, ?]).
muster(['Du', 1, 'mich', 2], ['Wieso glaubst Du, dass ich', 2, ?]).
muster(['Du', bist, 1, .], ['Warum bin ich', 1, ?]).
muster(['Weil', du, 1, .], ['Das stimmt nicht!']).
muster([1, X, 2], ['Kannst Du mir mehr über', X, 'erzählen.']):-
  wichtig(X).

muster([1], ['Erzähle mir bitte mehr von', 1, .]).
muster(_, Standardantwort):-
  R is random(7),
  standard(R, Standardantwort).
standard(0, ['Erzähle mir bitte mehr.']).
standard(1, ['Bitte fahre fort!']).
standard(2, ['Ich bin nicht sicher, dass ich dich richtig verstehe.']).
standard(3, ['Siehst du dafür bestimmte Gründe?']).
standard(4, ['Was bringt dich zu dieser Vermutung?']).
standard(5, ['Regt es dich auf, über solche Dinge zu sprechen?']).
standard(6, ['Hat das etwas mit deiner Aussage von vorhin zu tun?']).

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).
```

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 über 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 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.

ELIZA Programme wurden immer weiter entwickelt. Heute stehen im Internet so genannte Chat-Roboter zu allen möglichen Themen, wie Doktor, Beichtvater und ... zur Verfügung.

Zur Vertiefung des Themas gibt es auf dem Hessischen Bildungsserver unter http://lern-

archiv.bildung.hessen.de/archiv/sek_ii/informatik/13.2/prolog/ eine theoretische und unterrichtspraktische Arbeit zum Thema „Künstliche Intelligenz“ von K. G. Kristin und Dr. W. Steup.

22.6 Aufgaben

1a) Geben Sie eine Grammatik für die Sprache der 0-1-Wörter an, bei denen das rechte Halbwort das Spiegelbild des linken Halbworts 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 liebt das blonde Maedchen.
Erfragen: Wen oder was liebt 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) Ergänzen Sie im ELIZA-Programm Stimulus/Antwort-Paare.
b) Trennen Sie durch Einsatz der Akkumulatortechnik im Prädikat *antworte* die Verarbeitung von der Ausgabe.

23 GUI-Programmierung mit XPCE

In den vorangegangenen Kapiteln wurde auf der Konsole mit Prolog gearbeitet. Typisch für die Arbeitsweise eines interpretierenden Systems nimmt dabei der Prolog-Interpreter die Anfragen direkt entgegen und liefert sofort die betreffenden Ausgaben. Diese Art der Nutzung entspricht nicht dem üblichen Standard. Heutzutage haben Programme eine grafische Benutzungsoberfläche mit allen daraus sich ergebenden Vorteilen.

SWI-Prolog bietet mit der Erweiterung XPCE die Möglichkeit, Prolog-Programme mit einer grafischen Benutzungsoberfläche zu versehen. Im Unterschied zu Prolog ist XPCE ein objektorientiertes System, das speziell zur Entwicklung grafischer Benutzungsoberflächen entwickelt wurde. Es bietet alle Möglichkeiten, die man von einem GUI-Tool erwartet und noch einiges mehr. Kein Wunder, denn das System umfasst zurzeit 160 Klassen mit über 2700 Methoden. Bei der Entwicklung komplexerer Anwendungen kommt man nicht umhin, sich intensiv mit der XPCE-Dokumentation und seiner Klassenbibliothek auseinander zu setzen.

Im Folgenden geht es nur darum, anhand einfacher Beispiele, einen Einstieg in XPCE zu ermöglichen, um bei Bedarf eigene Prolog-Programme mit einer grafischen Benutzungsschnittstelle versehen zu können. Dabei wird weitgehend auf XPCE-spezifische Möglichkeiten verzichtet, Aktionen besonders effizient oder elegant durchzuführen. Vielmehr wird Wert auf Verständlichkeit gelegt.

23.1 XPCE-Grundlagen

Die Schnittstelle zwischen Prolog und XPCE besteht aus nur vier Prädikaten. Mit *new*/2 bzw. *free*/1 können Objekte erzeugt und wieder vernichtet werden, mit *send*/2 ruft man Methoden von Objekten auf und mit *get*/3 fragt man Attribute oder Werte ab. Die Anwendung dieser Prädikate zeigen wir am Beispiel eines einfachen Body-Mass-Index-Rechners. Dessen Benutzungsoberfläche können wir wie folgt erzeugen:

```
bmi:-
  new(D, dialog('BMI-Rechner')),
  new(Gewicht, int_item('Gewicht')),
  new(Groesse, int_item('Größe')),
  new(Ergebnis, text_item('Ergebnis')),
  send(D, append(Gewicht)),
  send(D, append(Groesse)),
  send(D, append(Ergebnis)),
  new(B1, button(berechne,
          message(@prolog, berechne, D))),
  new(B2, button(beenden,
          message(D, destroy))),
  send(D, append(B1)),
  send(D, append(B2)),
  send(D, default_button(B1)),
  send(D, open).
```

Abb. 23-1 GUI-Oberfläche des BMI-Rechners

Mit den ersten vier new-Anweisungen werden das Dialogfenster, die beiden Integer-Felder für Gewicht und Größe, sowie das Textfeld Ergebnis erzeugt. New ist also ein Konstruktor, mit dem Objekte der XPCE-Klassenbibliothek angelegt werden können.

Mit dem send/2-Prädikat wird anschließend vom Dialogfenster D die Methode append aufgerufen, wobei jeweils ein Eingabefeld als Parameter angegeben wird. Append bezieht sich hier auf XPCE und hat nichts mit dem append/3-Prädikat von Prolog zu tun. In üblicher objektorientierter Punktnotation wäre die Anweisung

```
send(D, append(Gewicht))
```

in dieser Form zu schreiben:

```
D.append(Gewicht);
```

XPCE verfügt über einen leistungsfähigen Layout-Manager, der sich um die Anordnung der GUI-Komponenten kümmert. Das Dialog-Fenster erhält über die append-Aufrufe drei Eingabefelder, welche vom Layout-Manager automatisch horizontal und vertikal ausgerichtet werden.

Anschließend werden mit *new*/2 die zwei Schalter erzeugt und wie die Eingabefelder als weitere Komponenten dem Dialogfenster hinzugefügt. Standardmäßig richtet der Layout-Manager Eingabefelder untereinander und Schalter nebeneinander aus. Die Anweisung

```
send(D, default_button(B1)),
```

ruft für das Dialogfenster D die Methode *default_button* mit dem Parameter B1 auf, wodurch der Schalter „Berechne“ mit der Returntaste gekoppelt wird. Zum Schluss wird die Methode *open* des Dialogfensters aufgerufen, wodurch das Fenster angezeigt wird.

Kommen wir noch mal zu den Schaltern zurück. Beim Konstruktor *button* werden zwei Parameter angegeben. Aus dem ersten macht der Layout-Manager die Schalterbeschriftung, der zweite Parameter definiert die Ereignisbehandlung. Beim Schalter *Berechne* bedeutet

```
message(@prolog, berechne, D).
```

dass XPCE beim Drücken des Schalters das Prolog-Prädikat *berechne* mit D als Parameter aufrufen soll. Beim Schalter *Beenden* bedeutet

```
message(D, destroy)
```

dass XPCE vom Dialogfenster D die Methode *destroy* aufrufen soll, wodurch das Fenster geschlossen und das Objekt D vernichtet wird. Destroy ist eine von free/1 abgeleitete Methode.

Wir halten fest, dass Ereignisse nur in XPCE auftreten können, und die Ereignisbehandlung über @prolog an Prolog weitergeleitet oder innerhalb von XPCE stattfinden kann.

Die BMI-Berechnung erfolgt im Prolog-Prädikat *berechne*/1:

```
berechne(D):-
  % Eingabe
  get(D, member('Gewicht'), Ge),
  get(Ge, selection, Gewicht),
  get(D, member('Größe'), Gr),
  get(Gr, selection, Groesse),

  % Verarbeitung
  BMI is Gewicht/(Groesse/100)^2,
  bmi(BMI, Antwort),

  %Ausgabe
  get(D, member('Ergebnis'), Ergebnis),
  send(Ergebnis, selection(Antwort)).
```

Wir halten uns an das EVA-Prinzip, lesen zuerst die Werte aus dem Dialogfenster ein, verarbeiten dann die Werte und geben das Ergebnis zum Schluss im Dialogfenster aus. Zum Einlesen der Werte brauchen wir das eingangs erwähnte *get*/3-Prädikat. Mit

```
get(D, member('Gewicht'), Ge),
```

wird vom Dialogfenster D die Funktion *member* mit dem Parameter *Gewicht* aufgerufen. Member ist eine Methode der XPCE-Klasse *dialog* und hat nichts mit dem Prolog-Prädikat *member*/2 zu tun. Get hat im Gegensatz zu send drei Parameter, weil im dritten Parameter das Ergebnis des Funktionsaufrufs zurückgegeben wird. In der üblichen Punktnotation objektorientierter Sprachen würde die get-Anweisung wie folgt geschrieben:

```
Ge = D.member('Gewicht');
```

Mit *send*/2 können also Objekteigenschaften gesetzt und Methoden (Prozeduren) ausgeführt werden und mit *get*/3 können Objekteigenschaften erfragt und Funktionswerte berechnet werden.

Die get-Anfrage ermittelt als Ergebnis Ge das Integer-Eingabefeld Gewicht. Der eingegebene Wert steht im Attribut *selection* und kann daher mittels

```
get(Ge, selection, Gewicht),
```

ermittelt und in der Prolog-Variablen Gewicht gespeichert werden. Analog zum Gewicht wird die Größe aus dem GUI-Formular eingelesen. Aus den beiden Werten wird der BMI-Faktor berechnet und mit dem Hilfsprädikat *bmi*/2 die Antwort ermittelt.

```
bmi(BMI, 'Du hast Untergewicht.'):-
  BMI < 19, !.
bmi(BMI, 'Du hast Übergewicht.'):-
  BMI > 25, !.
bmi(_BMI, 'Du hast Normalgewicht.').
```

Zum Ausgeben der Antwort wird mit der nächsten get-Anweisung das Textfeld Ergebnis ermittelt und dann die Antwort an dessen Attribut *selection* geschickt.

23.2 Ableitungen

In Kapitel 11 haben wir die zwei Prädikate *ableitung*/3 und *vereinfachen*/2 zum symbolischen Differenzieren entwickelt. Wir statten dieses Projekt nun mit einem einfachen GUI-Formular aus.

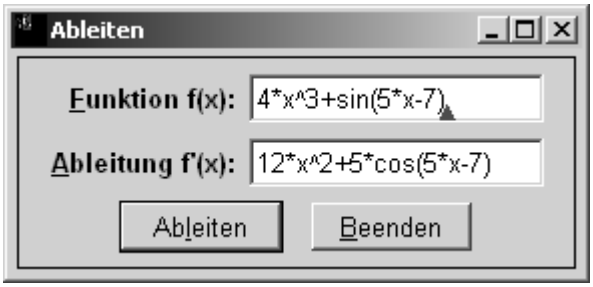


Abb. 23-2 GUI-Formular zum symbolischen Differenzieren

Das Formular ist leicht erstellt:

```
ableiten:-
  new(D, dialog('Ableiten')),
  new(Funktion,
      text_item('Funktion f(x)')),
  new(Ableitung,
      text_item('Ableitung f''(x)')),
  send(D, append(Funktion)),
  send(D, append(Ableitung)),
  new(B1, button(ableiten,
          message(@prolog, ableiten, D))),
  new(B2, button(beenden,
          message(D, destroy))),
  send(D, append(B1)),
  send(D, append(B2)),
  send(D, default_button(B1)),
  send(D, open).
```

Etwas mehr Mühe macht die Verarbeitung der Eingabe, denn XPCE kennt nur elementare Datentypen, aber keine komplexen Datentypen von Prolog. Beim Einlesen und Ausgeben der Daten müssen daher geeignete Konvertierungen vorgenommen werden.

```
ableiten(D):-
  % Eingabe
  get(D, member('Funktion f(x)'), Fu),
  get(Fu, selection, Funktion),

  % Verarbeitung
  term_to_atom(Term, Funktion),
  ableitung(Term, x, Ableitung),
  vereinfachen(Ableitung, Vereinfacht),
  term_to_atom(Vereinfacht, AlsAtom),

  % Ausgabe
  get(D, member('Ableitung f''(x)'), Abl),
  send(Abl, selection(AlsAtom)).
```

Das Attribut selection eines Textfelds liefert einen String, der in Prolog als Atom zur Verfügung gestellt wird. Mit dem Systemprädikat *term_to_atom*/2 kann der als Atom eingelesene Funktionsterm in einen Prolog-Term konvertiert werden. Nach der Ableitung und Vereinfachung wird der Ableitungsterm wieder mit *term_ to_atom*/2 in ein Atom konvertiert, um es im Textfeld der Ableitung anzeigen zu können.

23.3 Endlicher Automat

In Kapitel 18 haben wir endliche Automaten mit Prolog modelliert. Das Prädikat *automat*/1 hat dabei einen Automaten in den Anfangszustand gesetzt und ihn dann die Eingabe verarbeiten lassen. In einer grafischen Benutzungsoberfläche soll die Abarbeitung schrittweise erfolgen können. Bei jedem Schritt soll das nächste Eingabezeichen gelesen und verarbeitet werden.

Das Formular für den Automaten ist ganz einfach erstellt.

Abb. 23-3 GUI-Formular für endlichen Automaten

```
automat:-
  new(D, dialog('Automat')),
  new(TIEingabe, text_item('Eingabe')),
  new(TIZustand, text_item('Zustand')),
  send(D, append(TIEingabe)),
  send(D, append(TIZustand)),
  anfangszustand(Zustand),
  send(TIEingabe, selection(babab)),
  send(TIZustand, selection(Zustand)),
  new(B1, button('Zeichen lesen',
      message(@prolog, lesen, D))),
  new(B2, button(beenden,
      message(D, destroy))),
  send(D, append(B1)),
  send(D, append(B2)),
  send(D, default_button(B1)),
  send(D, open).
```

Die beiden Textfelder sind hier nach dem Programmstart schon vorbelegt. Die Vorbelegung des Textfeldes Eingabe ist statisch. Der Anfangszustand wird hingegen der Wissenbasis entnommen und im Textfeld TIZustand angezeigt.

Auch bei diesem Beispiel muss für die Verarbeitung eine Konvertierung der Daten stattfinden. Der Eingabestring wird mit dem Systemprädikat *atom_chars*/2 in eine Zeichenliste zerlegt, so dass das erste Zeichen verarbeitet werden kann. Die Restliste wird für die Ausgabe wieder in ein Atom konvertiert.

```
lesen(D):-
  % Eingabe
  get(D, member('Zustand'), TIZustand),
  get(D, member('Eingabe'), TIEingabe),
  get(TIEingabe, selection, Eingabe),
  get(TIZustand, selection, Zustand),

  % Verarbeitung
  atom_chars(Eingabe, [Kopf|Rest]),
  uebergang(Zustand, Kopf, NeuerZustand),
  atom_chars(RestEingabe, Rest),

  % Ausgabe
  send(TIEingabe, selection(RestEingabe)),
  send(TIZustand, selection(NeuerZustand)).
```

23.4 LOGO-Interpreter

Im nächsten Beispiel kommen weitere GUI-Komponenten hinzu. Der LOGO-Interpreter aus Kapitel 21.2 soll über eine GUI-Oberfläche bedient werden. Diese erhält im oberen Teil eine *view*- und im unteren eine *dialog*-Komponente. Die view-Komponente besitzt einen vielseitig verwendbaren Editor, mit dem mehrzeiliger Text verarbeitet werden kann. Zur Zusammenfassung der beiden Komponenten braucht man eine *frame*-Komponente.

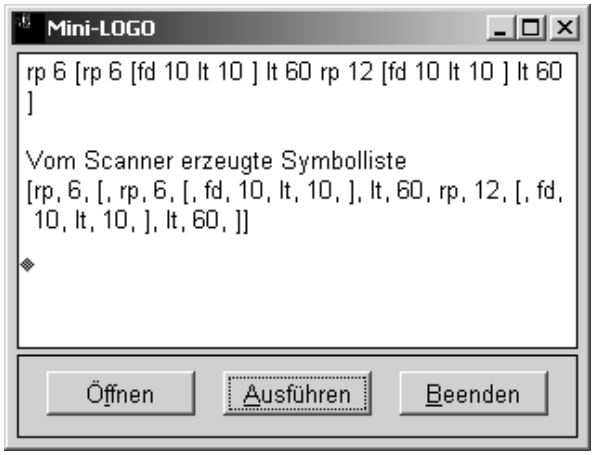


Abb. 23-4 GUI-Formular für Mini-LOGO

```
logo:-
  new(F, frame('Mini-LOGO')),
  new(V, view),
  new(D, dialog),
  send(F, append(V)),
  send(F, append(D)),
  send(D, below(V)),
  send(V, width(30)),
  send(V, height(10)),
  new(B1, button('Öffnen',
      message(@prolog, oeffnen, V))),
  new(B2, button('Ausführen',
      message(@prolog, ausfuehren, V))),
  new(B3, button('Beenden',
      message(F, destroy))),
  send(D, append(B1)),
  send(D, append(B2)),
  send(D, append(B3)),
  send(F, open).
```

Die view- und dialog-Komponente werden der frame-Komponenten mittels *append* hinzugefügt. *Send(D, below(V))* legt fest, dass die Dialogkomponente D unterhalb der view-Komponente V platziert werden soll. Mit *send(V, width(30))* wird die Breite der view-Komponente auf 30 eingestellt.

Beim Öffnen soll eine Datei mit einem mini-LOGO-Programm ausgewählt und in der view-Komponenten angezeigt werden. Die XPCE-Bibliothek *find_file* stellt einen komfortablen Dateiauswahldialog bereit, der wie folgt eingebunden wird:

```
:- use_module(library(find_file)).
:- pce_global(@finder, new(finder)).

oeffnen(V):-
  working_directory(Dir, Dir),
  get(@finder, file(@on, '.log', Dir), Datei),
  send(V, load(Datei)).
```

Mit use_module wird die XPCE-Bibliothek eingebunden und *new(finder)* erstellt einen Dateiauswahldialog als Objekt. Dieses Objekt wird mit dem Namen @finder durch *pce_global* global verfügbar gemacht, muss also nicht jedes Mal neu beim Öffnen erstellt werden.

Die Voreinstellungen des Dateiauswahldialogs werden durch *file(@on, '.log', Dir)* festgelegt. @on bedeutet, dass nur existierende Dateien ausgewählt werden können, .log ist die zu benutzende Dateierweiterung und *Dir* legt das Startverzeichnis für die Auswahl fest, in diesem Falle also das Arbeitsverzeichnis. Die get(@finder…)-Anweisung öffnet den Dateiauswahldialog. Der Name der ausgewählten Datei wird zurückgegeben. Über die load-Methode der view-Komponente wird die Datei dann geöffnet.

Die in 21.2 entwickelten Prädikate werden zum Ausführen eines mini-LOGO-Programms benutzt. Dabei soll die vom Scanner erzeugte Symbolliste in der view-Komponente angezeigt werden, der erzeugte Parsebaum mit zeichne_term und das ausgeführt Programm mit der turtle dargestellt werden.

```
ausfuehren(V):-
  get(V?editor?file, name, Datei),
  see(Datei),
  readln(Eingabe, _, '.',
         '0123456789', lowercase),
  seen,
  send(V?editor, append('\nVom Scanner ')),
  send(V?editor, append
              ('erzeugte Symbolliste\n[')),
  schreib_liste(V?editor, Eingabe),
  send(V?editor, append(']\n')),
  parse_logo(Eingabe, Parsebaum),
  zeichne_term(Parsebaum),
  turtle_init,
  asserta(penpos(down)),
  interpret_logo(Parsebaum).

schreib_liste(E, [Element]):-
  send(E, append(Element)).
schreib_liste(E, [Kopf|Rest]):-
  send(E, append(Kopf)),
     send(E, append(', ')),
     schreib_liste(E, Rest).
```

Die view-Komponente merkt sich den Namen der geöffneten Datei. Man muss allerdings den XPCE-Klassenbrowser bemühen, um das betreffende Attribut zu finden. Den Klassenbrowser öffnet man im SWI-Prolog-Editor über Hilfe | XPCE | Manual und wählt dann im Menü *Browsers* den Eintrag *Class Browser*.

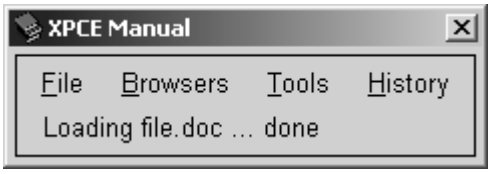


Abb. 23-5 Das XPCE-Manual-Menü

Unter *Class* wählt man zuächst view aus. Dass die view-Klasse über einen Editor verfügt, sieht man rechts oben in der Vererbungshierarchie. Ein Doppelklick auf *editor* öffnet die gezeigte Ansicht. Man sieht, dass dem Editor eine Datei zugeordnet ist. File ist aber die Dateivariable und nicht der Dateiname. Um den ausfindig zu machen, wechselt man zur Klasse *file* und findet dort den Dateinamen unter dem Attribut *name*.

Mit der folgenden get-Anweisung wird der Weg von der view-Komponente, über das editor-Attribut und dessen file-Attribut beschritten und der Dateiname in der Prolog-Variablen Datei gespeichert..

```
get(V?editor?file, name, Datei).
```

In der Punktnotation objektorientierten Sprachen würde man statt V?editor?file einfach V.editor.file schreiben.

Wir fahren mit der Beschreibung des *ausfuehren*-Prädikats fort. Mittels see, readln und seen liest der Scanner das ausgewählte mini-LOGO-Programm ein und erzeugt die Symbolliste. Über die append-Methode des Editors wird diese zusammen mit einer Überschrift ausgegeben, wobei das Prädikat *schreib_liste* behilflich ist. Anschließend erledigen der Parser und der Interpreter die restliche Arbeit.

23.5 Graphen-Layouter

Als abschließendes Beispiel behandeln wir die Arbeit mit Graphen. Graphen sollen grafisch dargestellt und bearbeitet werden können. Auf der nächsten Seite ist ein Bild der Anwendung zu sehen. Das Formular enthält links eine *view*-Komponente und rechts eine *picture*-Komponente, die zur Darstellung grafischer Objekte benutzt wird. Darunter befindet sich noch eine dialog-Komponente, und alle drei Komponenten sind in einer frame-Komponenten enthalten.

Das Formular wird mittels *graph* erzeugt:

```
graph:-
  new(F, frame('Graph')),
  new(V, view),
  send(V, width(25)),
  new(P, picture),
  new(D, dialog),
  send(F, append(V)),
  send(P, right(V)),
  send(D, below(V)),
  new(B1, button('öffnen',
      message(@prolog, oeffnen, V))),
  new(B2, button(zeichnen,
      message(@prolog, zeichnen, P))),
  new(B3, button(ausgeben,
      message(@prolog, ausgeben, P,V))),
  new(B4, button(speichern,
      message(@prolog, speichern, V))),
  new(B5, button(beenden,
      message(F, destroy))),
  send(D, append(B1)),
  send(D, append(B2)),
  send(D, append(B3)),
```

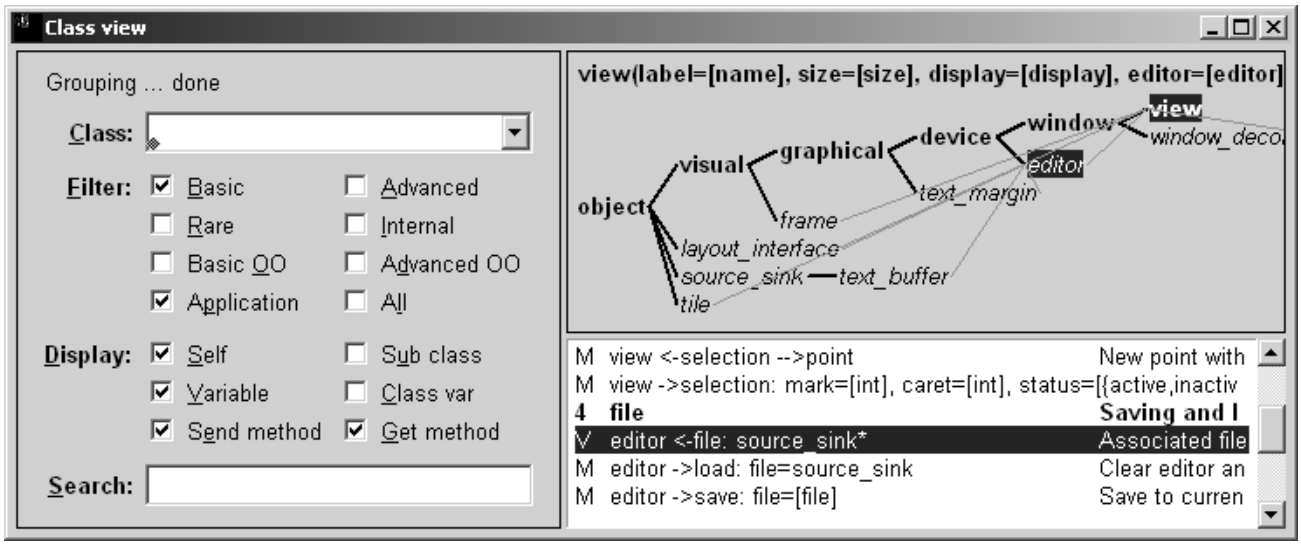


Abb. 23-6 Der XPCE-Klassenbrowser

```
  send(D, append(B4)),
  send(D, append(B5)),
  send(F, open).
```

Beim Öffnen wird eine Prolog-Datei mit Definitionen von Knoten und Kanten in die view-Komponente geladen. Bei Knoten werden Name und x-y-Position, bei Kanten die zwei zu verbindenden Knoten und die Distanz der beiden Knoten angegeben. Geöffnet wird wie beim mini-LOGO-Beispiel, wobei die ausgewählte Datei gleich in die Wissensbasis geladen wird.

```
oeffnen(V):-
  working_directory(Dir, Dir),
  get(@finder, file(@on, '.pl', Dir),
     Datei),
  send(V, load(Datei)),
  consult(Datei).
```

Der Inhalt des Editors kann wie folgt gespeichert werden:

```
speichern(V):-
  get(V?editor?file, name, Datei),
  send(V, save(Datei)).
```

Das Systemprädikat *forall/2* sorgt dafür, dass alle konsultierten Knoten und Kanten gezeichnet werden.

```
zeichnen(P):-
  forall(knoten(Name, X, Y),
         zeichneKnoten(P, Name, X, Y)),
  forall(kante(Von, Nach, D),
         zeichneKante(P, Von, Nach, D)).
```

Da ein Knoten aus zwei grafischen Elementen, einem Kreis und einem Text besteht, brauchen wir ein *device*-Objekt, das diese beiden Elemente zusammenfasst. Um später leichter mit den device-Objekten arbeiten zu können, erhalten sie mit der name-Methode den Namen des Knotens. Ein Kreis-Objekt wird erzeugt und mit der display-Methode in das device-Objekt eingefügt. Dann wird ein Objekt für grafischen Text mit dem Namen des Knotens erzeugt und auch in das device-Objekt eingefügt.

Um die Knoten auf der grafischen Oberfläche später verschieben zu können, erhält jedes display-Objekt ein *recogniser*-Objekt, das eine Mausbewegung bei gedrückter linker Maustaste in eine betreffende Objektbewegung umsetzt.

Zusätzlich werden für jeden Knoten vier Verbindungspunkte (handles) festgelegt, an die später die Verbindungskanten andocken können.

```
zeichneKnoten(P, Name, X, Y):-
  new(Device, device),
  send(Device, name(Name)),
  new(Kreis, circle(17)),
  send(Device, display(Kreis)),
  new(Text, text(Name)),
  send(Device, display(Text, point(5, 0))),
  send(P, display(Device, point(X, Y))),
  new(M, move_gesture(left)),
  send(Device, recogniser(M)),
```

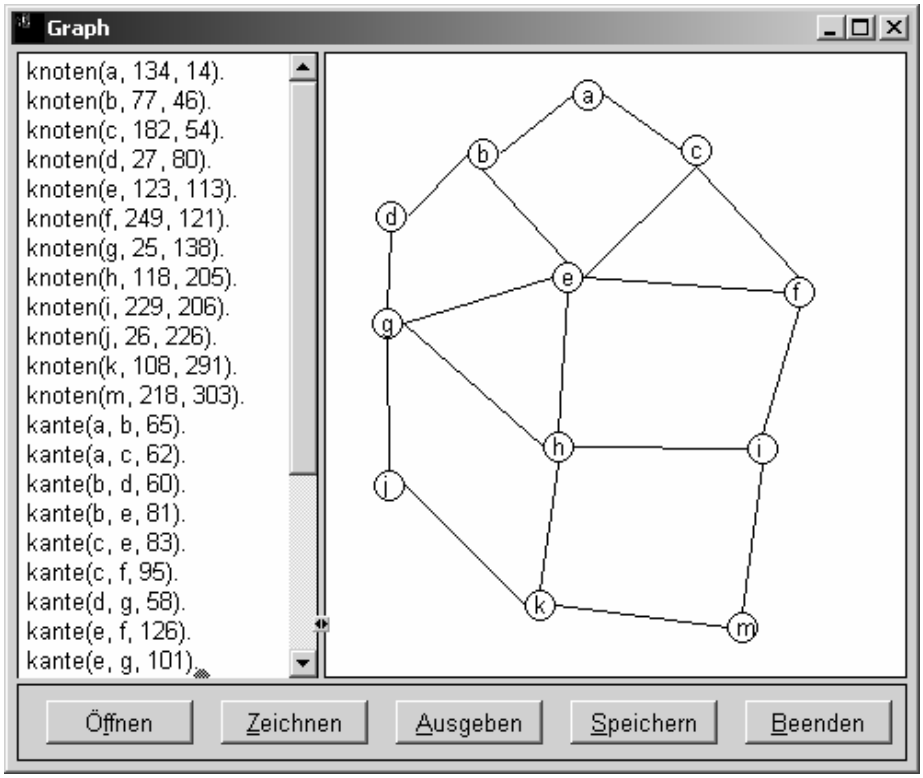


Abb. 23-7 Der Graphen-Layouter

```
  send(Device, handle(
       handle(w/2, 0, link, link))),
  send(Device, handle(
       handle(w/2, h, link, link))),
  send(Device, handle(
       handle(0, h/2, link, link))),
  send(Device, handle(
       handle(w, h/2, link, link))).
```

Da bei den Knoten Verbindungspunkte definiert wurden, können diese nun ganz einfach miteinander verbunden werden.

```
zeichneKante(P, Von, Nach, _D):-
  get(P, member(Von), K1),
  get(P, member(Nach), K2),
  send(K1, connect, K2, @graph_link).
```

Dabei ist *@graph_link* ein global verfügbar definiertes Verbindungsobjekt:

```
:- pce_global(@graph_link, new(link
   (link, link, line(0, 0, 0, 0)))).
```

Hat der Anwender das Layout seines Graphen gestaltet, so will er es auch speichern können. Dazu werden die Daten der Knoten und Kanten abgefragt und zunächst im Editor ausgegeben, dessen Inhalt dann gespeichert werden kann.

Vor dem Ausgeben wird zunächst der Inhalt des Editors mit der Methode *clear* gelöscht. Dann wird der Wissensbasis der Name eines Knoten entnommen und mittels member das zugehörige device ermittelt. Über die offset-Methode wird die Position des Knotens bestimmt. Mit der praktischen String-Methode können alle Informationen zu einem knoten-Fakt zusammengefasst und in den Editor eingefügt werden. Das abschließende fail sorgt dafür, dass alle Knoten dran kommen. Man hätte natürlich auch hier mit forall arbeiten können, hätte dafür aber ein weiteres Prolog-Prädikat gebraucht.

```
ausgeben(P, V):-
  send(V?editor, clear),
  knoten(Knoten, _, _),
  get(P, member(Knoten), Device),
  get(Device, offset, point(X, Y)),
  send(V?editor, append(string
      ('knoten(%s,%d,%d).\n',Knoten,X,Y))),
  fail.
```

Natürlich kennt das Picture-Objekt alle seine Knoten und man könnte ohne Rückgriff auf die knoten-Fakten der Wissensbasis die gewünschten Informationen ermitteln. Allerdings wären dafür weitergehende XPCE-Spezialitäten erforderlich.

Mit einer weiteren *ausgeben*-Klausel werden die Kanten ausgegeben. Über die Wissensbasis werden die verbundenen Knoten ermittelt und über die zuständigen displays die aktuellen Positionen ermittelt. Daraus wird der aktuelle Abstand der Knoten errechnet und schließlich die Angaben als kanten-Fakt im Editor ausgegeben.

```
ausgeben(P, V):-
  kante(Knoten1, Knoten2, _),
  get(P, member(Knoten1), D1),
  get(P, member(Knoten2), D2),
  get(D1, offset, point(X1, Y1)),
  get(D2, offset, point(X2, Y2)),
  Dis is round(sqrt((X1-X2)^2+(Y1-Y2)^2)),
  send(V?editor, append(string('kante
    (%s,%s,%d).\n',Knoten1,Knoten2,Dis))),
  fail.
ausgeben(_, _).
```

23.6 Ausführbare Prolog-Programme

Prolog-Programme können auch als direkt ausführbare Programme erzeugt werden, die ohne eine installierte Prolog-Umgebung lauffähig sind. Für das Beispiel des BMI-Rechners erzeugt man folgende Datei.

```
:- consult('bmi.pl').

main:-
  pce_main_loop(main).

main(_Argv):-
  bmi. % Start-Prädikat des BMI-Rechners

save(Exe):-
  pce_autoload_all,
  qsave_program(Exe,
    [ emulator(swi('bin/xpce-stub.exe')),
      stand_alone(true),
      goal(main) ]
  ).
```

Beim Konsultieren dieses Programms wird der BMI-Rechner gestartet. Anschließend ruft man auf der Konsole save(bmi) auf, wodurch die ausführbare Datei bmi.exe erzeugt wird. Sie funktioniert allerdings nur zusammen mit den DLL-Dateien libpl.dll, pl2xpce.dll und pthreadVC.dll.

23.7 Aufgaben

1. Realisieren Sie das Auskunfts- und Reisebuchungssystem aus Kapitel 14 als GUI-Anwendung.
2. Implementieren Sie das ELIZA-Programm aus Kapitel 22 als GUI-Anwendung. Zum Zerlegen der Eingabe in eine Tokenliste benutzen Sie das Prädikat *tokenize_atom*/2 der Bibliothek *porter_stem*.

Anhang A – Glossar

Anfrage

Eine Frage des Benutzers an das Prolog-System. Mit der Anfrage *?- vater(Papa, max).* 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 bzw. Parameter eines Prädikats. In *vater(peter, hans)* sind *peter* und *hans* die beiden Argumente.

Arität

Stelligkeit oder Anzahl der Argumente einer Struktur. Im Fakt *schueler(hans, schmidt, 9a)* hat das Prädikat *schueler* die Arität 3. Schreibweise: *schueler/3.*

Atom

Eine Zeichenfolge zur Bezeichnung einer Textkonstanten. Atome werden klein oder in einfachen Anführungszeichen geschrieben. Beispiele: *hans*, *‘Müller’*. 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: „!“

Disjunktion

Logische Oder-Verknüpfung. Sie wird in Prolog durch das Semikolon ausgedrückt.

Fakt

Eine Tatsache über Eigenschaften von Objekten oder Beziehungen zwischen diesen. Beispiel: *alter(‘Theo’, 27)* sagt aus, dass Theo 27 Jahre alt ist. Ein Fakt schreibt man als Klausel ohne Regeloperator „:-“.

Funktor

Name einer Struktur bzw. eines Prädikats. Beispiele: In *mutter(doris, lea)* ist *mutter* Funktor des Prädikats mutter/2. In 2 =< 3 + 7 ist „=<“ und in [2, 3, 7] „.“ der jeweilige Funktor. Stellt man eine Struktur als Baum dar, so findet man den Funktor in der Wurzel.

Instanzierung

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] ist 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: *elternteil/2* ist das *zweistellige* Prädikat *elternteil*.

Regel

Logische Aussage der Form „Wenn A_1 und A_2 und ... und A_n , dann B“, wobei A_1, A_2,..., 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,..., A_n.“, wobei A_1, A_2,..., 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,..., A_n. existiert, deren Kopf B sich mit der Aussage A unifizieren lässt, so ist A beweisbar, wenn alle Teilziele A_1, A_2,..., 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 zusammen gesetzter 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, das während der Programmausführung durch Unifikation mit Konstanten und Strukturen instanziert 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 *consult* oder *assert* in die Wissenbasis 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.

zusammen gesetzter Term

Besteht aus einem Funktor und einem oder mehreren Argumenten, auch Struktur genannt.

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+Höller.

[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 Programmiertechniken. 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.

[Wie1] Wielemaker, J.: SWI-Prolog 5.6, Reference Manual. Amsterdam: University of Amsterdam, 2007.

[Win1] Winston, P.: Künstliche Intelligenz. Bonn: Addison-Wesley, 1987.

Anhang C – Index

Zeichen

A

B

C

D

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z