Peter Zöller-Greer

Künstliche Intelligenz
Grundlagen
und
Anwendungen

Künstliche Intelligenz

ist die Zusammenfassung einer ganzen Palette von Arbeitsgebieten. In diesem Kompaktkurs werden in die Aussagen- und Prädikatenlogik eingeführt, die KI-Sprache PROLOG erläutert sowie die wichtigen Anwendungen Expertensysteme, Fuzzy-Systeme, Neuronale Netze und genetische Algorithmen besprochen. Eine kurze Diskussion philosophischer Probleme der KI runden das Buch ab. Nach jedem Kapitel sind Übungen zum Selbsttest vorhanden (mit Musterlösungen).

In der gleichen Reihe erschienen:

Prof. Dr. rer. nat. Peter Zöller-Greer
ist Mathematiker und unterrichtet die Fächer Künstliche Intelligenz, Software-Engineering und Multi Media Systeme an der FH Frankfurt am Main-University of Applied Sciences.

9 783981 163902
Preis: 22,- Euro

ISBN 978-3-9811639-0-2

Die Reihe *Wissen & Praxis >kompakt<* nimmt sich komplexer Themen an und versucht diese so einfach wie möglich, beschränkt auf das Wesentliche, für Studium und Praxis gleichermaßen geeignet darzustellen.

Composia Verlag
www.composia.de

Künstliche Intelligenz
Zöller-Greer
Composia Verlag

Peter Zöller-Greer

Künstliche Intelligenz

Grundlagen und Anwendungen

Über 1 Jahr fast ununterbrochen Platz 1 auf amazon.de-Bestsellerliste, z.T. in mehreren Kategorien gleichzeitig!

Mit einer Einführung in die Aussagen- und Prädikatenlogik, PROLOG, Expertensysteme Fuzzy-Systeme, Neuronale Netze und Genetische Algorithmen

Reihe *Wissen & Praxis >kompakt<*

Composia Verlag

Peter Zöller-Greer

Künstliche Intelligenz

Grundlagen und Anwendungen

Mit einer Einführung in die Logik, PROLOG, Expertensysteme, Fuzzy-Systeme, Neuronale Netze und Genetische Algorithmen

Prof. Dr. Peter Zöller-Greer
studierte nach einer Berufsausbildung als Physiklaborant (BASF AG, Ludwigshafen/Rh.) von 1975-1981 Mathematik (Diplom) und Theoretische Physik an den Universitäten Siegen und Heidelberg. Er promovierte an der Universität Mannheim über eine approximationstheoretische Lösung eines Problems aus der Quantenmechanik und arbeitete zunächst als Systemanalytiker bei Brown Boveri Reaktor GmbH und danach als DV-Referent bei ABB Mannheim, bevor er 1989 die Geschäftsführung der Firma Composia GmbH in Mannheim übernahm. Bereits während dieser Tätigkeiten arbeitete er als freier Dozent an verschiedenen Hochschulen. Seit 1993 ist er Professor am Fachbereich Informatik und Ingenieurwissenschaften der FH Frankfurt am Main – University of Applied Sciences. Seine Lehr- und Arbeitsgebiete sind Künstliche Intelligenz, Software-Engineering und Multimedia-Systeme.

Bibliographische Information der Deutschen Bibliothek:
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliographie; detaillierte bibliographische Daten sind im Internet über http://dnb.ddb.de abrufbar.

2. Auflage

ISBN 978-3-9811639-0-2

Inhalt:

Vorwort zur 2. Auflage

Nach dem erfreulichen Erfolg der 1. Auflage liegt nun eine überarbeitete Version des Buchs vor. Es wurden in erster Linie Druckfehler beseitig, die sich in die erste Auflage eingeschlichen hatten.
Bleibt zu hoffen, dass auch der zweiten Auflage der gleiche Erfolg wie der ersten beschienen ist.

Im Februar 2010
Peter Zöller-Greer

Vorwort zur 1. Auflage

Es gab Zeiten, da war Künstliche Intelligenz (KI) ein Topthema der Informatik (und Philosophie). Doch die anfängliche Euphorie wich der Realität. Eigenständig denkende Roboter, die vielleicht sogar einmal die Welt beherrschen, sind in weite Ferne gerückt, an Computer mit „Bewusstsein“ glaubt kaum noch jemand. Doch langsam aber sicher nähern wir uns Geschwindigkeits- und Speicherkapazitäten im Bereich der Computerhardware, welche denen des menschlichen Gehirns durchaus vergleichbar sind. So ist es nur noch eine Frage der Zeit, wann ein ausreichend „großes“ künstliches Neuronales Netz auf so einer Hardware zum Laufen gebracht wird, mit der gleichen Anzahl Neuronen wie in einem Gehirn, und damit trainiert werden kann so wie Menschen in Schulen auch. Dies hat aber auch wichtige philosophische Konsequenzen: „Benimmt“ sich so ein künstliches neuronale Netz nämlich so, als hätte es Persönlichkeit, so stellt sich die Frage, ob der Konditional überhaupt angebracht ist. Im letzten Kapitel dieses Buches (Philosophische Probleme der KI) wird dieser Frage nachgegangen.
Doch zuvor muss ein grundlegendes Verständnis der Künstlichen Intelligenz erarbeitet werden. Nach einleitenden Definitionen und kurzen historischen Betrachtungen (Kapitel 1) ist es erforderlich, zumindest die Grundzüge der Aussagen- und Prädikatenlogik zu kennen (Kapitel 2 und 3). Danach kann die Arbeitsweise der KI-Sprache PROLOG verstanden werden (Kapitel 4) sowie der Aufbau von damit entwickelten Expertensystemen (Kapitel 5). Dann wird in die Fuzzy-Logik eingeführt und an einem Beispiel ein Fuzzy-Expertensystem entwickelt (Kapitel 6). Kapitel 7 beschäftigt sich etwas ausgiebiger mit Neuronalen Netzen und Kapitel 8 mit genetischen Algorithmen, wo Programme „gezüchtet“ werden. Kapitel 9 wird –wie schon angekündigt- einige prominente philosophische Positionen zur KI durchleuchten.
Zu jedem dieser Kapitel gibt es eigene, z.T. sehr umfangreiche Lehrbücher, daher können die hier präsentierten Kapitel das jeweilige Gebiet natürlich nur einführend behandeln.
Ich möchte an dieser Stelle meiner lieben Frau Diana für ihre unermüdliche Geduld mit mir danken. Dank auch meinen „Cratchits“, die mir einen neuen Blick auf das Wesentliche gaben.

Im März 2007
Peter Zöller-Greer

1. Menschliche Intelligenz – Künstliche Intelligenz

Eine Definition des Begriffs „Intelligenz“ erweist sich als sehr schwierig. Intelligenz (lat.: *intelligentia* „Einsicht, Erkenntnisvermögen“, *intellegere* „verstehen“) bezeichnet im weitesten Sinne die Fähigkeit zum Erkennen von Zusammenhängen und zum Finden optimaler Problemlösungen. Der US-amerikanische Psychologe und Ingenieur Louis Leon Thurstone (1887-1955) definiert Intelligenz über den Besitz folgender Fähigkeiten:

- räumliches Vorstellungsvermögen
- Rechenfähigkeit
- Sprachverständnis
- Wortflüssigkeit
- Gedächtnis
- Wahrnehmungsgeschwindigkeit
- logisches Denken

Andere halten die Fähigkeit, aus Beobachtungen abstrakte Modelle bilden zu können, für das wichtigste Merkmal von Intelligenz. Wieder andere glauben, Intelligenz ist schlicht das Ergebnis eines Intelligenztests (IQ).

Dies alles zeigt, dass eine allgemeinverbindliche Definition dieses Begriffs nicht geleistet werden kann. Zweifellos sind kognitive Fähigkeiten und Abstraktionsvermögen, Erkenntnisfähigkeit einschl. der Fähigkeit, diese Erkenntnisse zu kommunizieren, wichtige Merkmale von Intelligenz. In welchem Maß die Fähigkeit zur Kreativität hier zugerechnet werden kann, ist umstritten. Es gibt sicher recht kreative Musiker oder Maler, die aber dennoch nicht über einen sehr hohen IQ verfügen. Außerdem gilt es als erwiesen, dass hohe Intelligenz nicht automatisch zu einem „erfolgreichen“ Leben führt. Viele erfolgreiche Menschen sind nur mittelmäßig intelligent. Deswegen hat der Psychologe Daniel Golemann den Begriff der „Emotionalen Intelligenz“ geprägt, welche diesen Aspekt berücksichtigt. Zur Emotionalen Intelligenz zählen Fähigkeiten wie reale Selbsteinschätzung, Selbstkontrolle, Erkennen eigener Stärken und Schwächen, Empathie, Selbstmanagement etc.; alles Dinge, die nicht unter den „normalen“ Intelligenzbegriff fallen, aber für Erfolg im Beruf ausschlaggebend sind.

Was nun die *Künstliche* Intelligenz (KI oder AI für Artificial Intelligence) betrifft, so gibt es hier zwei wichtige Richtungen, welche auf den *Zweck* bezogen sind:

1. Die **KI als Kognitionswissenschaft** verfolgt das Ziel, die Funktionen des Gehirns zu entschlüsseln. Sie will die natürliche Intelligenz verstehen, nachbauen und auch ersetzen. Der Computer ist dazu das wichtigste Werkzeug.
2. Die **KI als Informatikdisziplin** strebt nach intelligenteren Computer-Systemen. Sie will Software mit Problemlösungsfähigkeiten entwickeln, als Ergänzung der natürlichen Intelligenz. Ob dies nach den Mechanismen der natürlichen Intelligenz erreicht wird, spielt dabei keine Rolle.

In diesem Buch werden Verfahren vorgestellt, welche beiden Strömungen gerecht werden. Aus diesem Grund definieren wir wie folgt:

Definition 1.1 (**Künstliche Intelligenz**)
Unter dem Begriff *Künstliche Intelligenz* verstehen wir die zusammenfassende Bezeichnung für folgende Arbeitsgebiete der Informatik:

- Wissensbasierte Systeme (Expertensysteme)
- Lernfähige und erkennende Systeme (Neuronale Netze)
- Sprachverstehen
- deskriptive und funktionale Sprachen (z.B. PROLOG)
- Robotik
- Fuzzy-Systeme
- Genetische Algorithmen
- Kreative Maschinen
- Philosophische Probleme (z.B. Turing-Test, Chin. Zimmer, Computer-Bewusstsein)

Es ist klar, dass es sich bei so einer Definition um eine „dynamische“ Liste handelt, d.h. in Zukunft wird sich diese Liste erweitern.
Die Behandlung all dieser Arbeitsgebiete würde den Rahmen dieses Buches sprengen. Die Gebiete Robotik und Kreative Maschinen sind z.B. hier kaum vertreten, und auch andere Teilgebiete können nur oberflächlich gestreift werden. Es wird vielmehr Wert auf eine grundlegende Basis gelegt, von der aus auch die anderen, hier nur marginal behandelten Gebiete, selbst erarbeitet werden können.
Bevor wir dies tun, möchte ich allerdings noch einen kurzen historischen Überblick über die bisherige Entwicklung der Künstlichen Intelligenz geben:

- Aristoteles (384–322 v. Chr.): Gesetze der Natur und des Denkens (*Metaphysik*)
- René Descartes (1596–1650): Verstand als physikalisches System; Dualismus: Geist (freier Wille) - Materie
- Gottfried W. Leibniz (1646–1716): Monaden = Bausteine von einfachen Substanzen, Wahrnehmung und Seele
- Immanuel Kant (1724–1804): *Der Verstand schöpft seine Gesetze nicht aus der Natur, sondern schreibt sie dieser vor.* [Idealismus]
- Charles Babbage (1792–1871): *Difference Engine:* Logarithmen-Tafeln
- David Hilbert (1862–1943): Vortrag 1900 in Paris- 23 Probleme. 23. Problem: E*ntscheidungsproblem*
- Kurt Gödel (1906–1978) Sätze über die Vollständigkeit und Unvollständigkeit formaler Systeme (1930)
- Alan Turing (1912–1954): Im Jahr 2000 kann ein Computer mit Speichereinheiten so programmiert werden, dass er sich 5 Minuten mit einem Menschen unterhalten kann und zu 30% den Turing-Test besteht, d.h. für einen Menschen gehalten wird (ist mehr als eingetroffen)
- Alan Turing (1940): erster Computer *Heath Robinson* zum Dechiffrieren deutscher Codes
- Konrad Zuse (1941) *Z 3* mit *Plankalkül* – erster frei programmierbarer Computer
- Marvin Minsky & D. Edwards (1951) bauen den ersten Computer, der auf Neuronalen Netzen basiert (Princeton)
- Arthur Samuel (1952) entwickelt *Dame*-Programme, die lernen, ihre Spielstärke zu verbessern
- 2monatiger Workshop in Dartmouth (1956),10 Teilnehmer: McCarthy, Minsky, Shannon, Rochester u.a.; Idee: Zusammenhänge von Automatentheorie, Neuronalen Netzen und Intelligenzforschung untersuchen
- Der Begriff Artificial Intelligence wird als Name der neuen Disziplin vorgeschlagen und etabliert sich. J. McCarthy (1958) geht ans MIT und entwickelt die Programmiersprache *Lisp*, das *Time-Sharing* für Betriebssysteme und „*Programs with Common-Sense*"; Konzentration auf Wissensrepräsentation und Schließen mittels Logik-Formalismen
- H.A. Simon 1958: *In 10 Jahren wird der Schachweltmeister ein Computer sein.* (1997 IBM-Rechner *Deep Blue*, der 250 Mio. Stellungen in einer Sekunde berechnen konnte, gewinnt gegen Garri Kasparow in 6 Spielen)
- 1966 Einstellen aller US-Funds für Automatische Übersetzung Englisch-Russisch. Den letzten Ausschlag dafür soll angeblich folgendes gegeben haben: Engl. Original: "The spirit is willing but the flesh is

weak." Programm machte daraus: "The vodka is good but the meat is rotten." Spielzeugprobleme seien lösbar, ernsthafte nicht. Hoffnung: schnellere Hardware, größerer Speicher

- Erstes Expertensystem *MYCIN* (1969) zur Diagnose von Blut-Infektionen
- *PROLOG* – "Programmation en Logique" *Constraint Logic Programming* (1989)

Seither gibt es eine Reihe von Softwaren und Verfahren, die z.T. in diesem Buch beschrieben werden, welche z.B. in Suchmaschinen im Internet, Computer-Games und in der Beweistheorie mit Erfolg eingesetzt werden.

Die nächsten Kapitel beschäftigen sich nun mit den logischen Grundlagen, ohne die Künstliche Intelligenz nicht möglich wäre.

2. Aussagenlogik

Dieses Kaptitel versucht in einer Art „Crash-Kurs“ die wichtigsten Begriffe der Aussagenlogik zu präsentieren. Zu diesem Zweck werden manchmal vereinfachte Definitionen herangezogen, die eher auf „Common Sense“ als auf axiomatischem Formalismus basieren. In einer Mathematikveranstaltung würde man dies so nicht machen, aber mit Rücksicht auf den Anwendungsaspekt dieser Dinge kann dies hier wohl vertreten werden.

Zunächst braucht man einen Wahrheitsbegriff. In der Philosophie unterscheidet man zwei „Sorten“ Wahrheiten:

1. Notwendige Wahrheiten.
Darunter versteht man Wahrheiten, deren Gegenteil logisch unmöglich ist. Dies ist nun auch keine präzise Definition, denn es ist unklar, was es heißt, "logisch unmöglich" zu sein. Man setzt hier wieder auf ein gewisses "Metawissen" über die Wahrheit. In der Logik definiert man gewisse mathematische Regeln als wahr (vgl. TARSKI's Wahrheitsdefinition), wie z.B. die Schlussfolgerungsregel. Danach ist das Geschlussfolgerte wahr, falls die Voraussetzung wahr ist. Der Verstoß solcher Regeln ist dann die besagte logische Unmöglichkeit.
Beispiel: "Alle Menschen sind sterblich. Sokrates war ein Mensch. Daraus folgt: Sokrates ist sterblich." Basierend auf dem Wahrheitsbegriff der Logik ist diese Aussage eine notwendige Wahrheit, denn das Gegenteil davon ist logisch unmöglich. Ein anderes Beispiel ist 2 + 2 = 4.

2. Kontingente Wahrheiten.
Dies sind Aussagen, die zwar auch wahr sind, aber deren Gegenteil logisch möglich wäre. Eine Aussage wie "1986 war Ronald Reagan Präsident der USA" zählt hierzu.

Beide Wahrheiten sind jedoch immer absolute Wahrheiten, also Wahrheiten, die unabhängig von Raum und Zeit gelten. Sie sind also zu jeder Zeit an jedem Ort wahr. 2 + 2 = 4 ist so eine absolute Wahrheit, da sie unabhängig von einem Ort oder einem Zeitpunkt ist. Eine Aussage wie "Es gibt keine absolute Wahrheit" ist demnach logisch falsch, denn diese Aussage wäre, wenn sie denn stimmte, selbst auch keine absolute Wahrheit. Damit könnte sie auch falsch sein. Ist es aber falsch, dass es keine absolute Wahrheit gibt, so muss es also doch eine solche geben. Damit leitet sich aus der Aussage "Es gibt keine absolute Wahrheit" logisch her, dass es eben doch eine absolute Wahrheit geben kann. Die Aussage müsste also mit ihrem Gegenteil gleichzeitig stimmen, und das kann nicht sein. Daher ist die Annahme, nämlich "Es gibt keine absolute Wahrheit"

offenbar falsch, und deren Gegenteil, also "Es gibt eine absolute Wahrheit" richtig. Nachfolgend werden wir nun die wichtigsten Begriffe der Aussagenlogik definieren.
Zunächst sei der Begriff einer Definition selbst erläutert. Unter einer *Definition* verstehen wir die Zuordnung eines Begriffes (Ansammlung von Buchstaben) zu einem Objekt der Anschauung oder der Realität.
Es macht daher z.B. keinen Sinn zu fragen, ob eine Definition wahr oder falsch ist. Eine Definition hat „demokratischen" Charakter, d.h. es wird schlicht mehrheitlich beschlossen, wie man etwas nennt.

Damit können wir jetzt festlegen:

Definition 2.1 (**Aussage**)
Unter einer *Aussage* verstehen wir eine Behauptung, die wahr oder falsch sein kann.

Es wird bei Aussagen also vorausgesetzt, dass deren Wahrheit prinzipiell entscheidbar ist. Dies ist eine Einschränkung, die allgemein nicht gilt, denn wie der Mathematiker Gödel gezeigt hat, gibt es auch nicht entscheidbare Behauptungen innerhalb eines logischen Systems. Wenn wir aber hier von Aussagen reden, beschränken wir uns also immer auf prinzipiell entscheidbare Behauptungen.
In der Mathematischen Logik wird häufig einer wahren Aussage der binäre Wert „1" und für falsche Aussagen der binäre Wert „0" zugewiesen.

Definition 2.2 (**Axiom**)
Unter einem *Axiom* verstehen wir eine Aussage, deren Wahrheit definiert wird.

Axiome braucht man also nicht beweisen (man darf es auch gar nicht können), sie stellen die Grundannahmen eines logischen Systems dar, auf denen alles andere aufbaut. Bei der Wahl von Axiomen muss man also vorsichtig sein: Handelt es sich um eine „beweisbare" (also aus etwas anderem herleitbare) Aussage, stellt sie definitionsgemäß kein Axiom dar. Axiome sollten „evidente Wahrheiten" sein, die man zwar nicht beweisen kann, deren Wahrheit aber so offensichtlich ist, dass man sie eben als *wahr definiert*, um eine solide Ausgangsbasis zu haben. Dieser Gedanke wurde schon in der Antike aufgegriffen, z.B. durch die Euklid'schen Axiome. Eines davon ist:

„Ein (endliches) Ganzes ist immer größer als eines seiner Teile".

Diese Einsicht erscheint evident, und ist doch nicht beweisbar. Also definiert man eben, dass diese Aussage wahr ist.

Meistens erstellt man gleich ein ganze Menge von Axiomen, die dann folgerichtig „Axiomensystem“ genannt wird. Dabei muss auch hier die Forderung gelten, dass keine der Aussagen aus einem Axiomensystem aus den anderen Axiomen hergeleitet werden kann. Axiomensysteme sind in diesem Sinne also immer auch minimal.
Im Gegensatz zu anderen Wissenschaften haben die Axiome der Mathematik „Ewigkeitswert“. Das heißt, dass Axiome nicht „veralten“ oder irgend wann einmal nicht mehr „richtig“ sind. Da die Mathematik ein rein „geistiges“ Gebäude ist und sich nicht darum schert, ob die Aussagen der Mathematik in der „wirklichen Welt“ eine Entsprechung haben, kann niemand die Mathematiker zwingen, einmal als wahr definierte Axiome wieder aufzugeben. Solange die Mehrheit der Mathematiker ein Axiom als wahr definiert , bleibt es wahr, ungeachtet der „realen“ Welt.
Da haben es z.B. Physiker oder selbst Philosophen nicht so einfach. Emanuel Kant stellte in seiner „Kritik der reinen Vernunft“ z.B. das Axiom auf, dass Raum und Zeit absolut seien. Doch Einstein konnte zunächst theoretisch und später auch experimentell belegen, dass dem in „Wirklichkeit“ gar nicht so ist. Daraus lernen wir, dass Axiome „über die Welt“ sich irgend wann einmal als falsch herausstellen können. Da die Mathematik keine Aussagen „über die Welt“ macht, hat sie dieses Problem nicht. Dass man die Aussagen der Mathematik „in der Welt“ anwenden und damit Brücken und Hochhäuser bauen kann, sehen die reinen Mathematiker eher als einen günstigen Zufall an.
So gibt es in der Mathematik vieles, was es im Universum gar nicht gibt: Eine (wirklich ganz flache) Ebene, eine Gerade, Kreise, Kugeln etc.; dies sind alles Gebilde, die in Realität allenfalls näherungsweise existieren. Deshalb sprach Plato auch von der Erde als eine Schattenwelt im Gegensatz zu der „Ideenwelt“. In der Ideenwelt gibt es wirklich Geraden, Kreise und Kugeln, doch deren imperfekte Projektionen auf unsere reale Welt liefern eben immer nur unvollkommene „Schatten“ wie reale Kugeln, die nicht „wirklich rund“ sind. Für Plato war z.B. auch die „Zeit“ eine unvollkommene Projektion der Ewigkeit aus der Ideenwelt.
Wir werden uns aber auf die Axiome der Logik beschränken und verbleiben damit in der Mathematik und brauchen keine Angst zu haben, dass diese Axiome einmal „Out of Date“ sind.

Definition 2.3 (**Theorem**)
Ein *Theorem* (oder *Satz*) ist eine Aussage, deren Wahrheit abhängig von gewissen Voraussetzungen „bewiesen“ werden kann (d.h. die Wahrheit ist „herleitbar“). Ein *Lemma* ist ein Hilfssatz, der meistens von einem „wichtigen“ Satz gefolgt wird. Ein *Korrolar* ist ein Nachsatz, meistens ein Spezialfall eines „wichtigen“ Satzes.

Theoreme sind also auch wahre Aussagen, aber welche, die man beweisen kann. Zum Beweis darf man Axiome benutzen oder andere bereits bewiesene Sätze (Theoreme). Letztendlich fußt also alles auf die Wahrheit der Axiome, zumindest indirekt. Deswegen ist das Gebäude der Mathematik auch so stabil.

Definition 2.4 (**Tautologie, Kontradiktion**)
Unter einer *Tautologie* verstehen wir eine Aussage, deren Wahrheit beweisbar ist und die unabhängig vom Wahrheitsgehalt evtl. beteiligter Teilaussagen gilt. Für eine *Kontradiktion* gilt das Gegenteil: Dies ist eine Aussage, welche unabhängig von dem Wahrheitsgehalt evtl. beteiligter Teilaussagen als falsch bewiesen werden kann.

Tautologien sind also „immer wahre" Aussagen, Kontradiktionen „immer falsche" Aussagen. Beispiele für Tautologien sind: „Liegt der Bauer tot im Zimmer, lebt er nimmer" oder: „Wenn der Hahn kräht auf dem Mist, dann ändert sich das Wetter oder es bleibt wie es ist". Wir werden später genauer sehen, warum das Tautologien sind, doch das „Gefühl" sagt einem jetzt schon, dass dem so ist. Ein Beispiel für eine Kontradiktion ist die schon zitierte Aussage: „Es gibt keine absoluten Wahrheiten". Ein anderes Beispiel wäre: „Über Gott kann man keine Aussage machen" (denn dieser Satz ist selbst bereits eine Aussage über Gott). Da die letzten beiden Aussagen eine Kontradiktion sind, weil sie sich selbst widersprechen, nennt man solche Aussagen auch *selbstwidersprüchlich* oder *selbstzerstörend*. Es gibt natürlich auch andere Kontradiktionen wie z.B. „Wenn 2+2=4 ist, dann ist 2+2=5".
Wir sahen, dass Aussagen Wahrheitswerte zugeordnet werden können, und zwar die Werte „wahr" (oder „1") und „falsch" (oder „0"). Wenn man jetzt Aussagen miteinander verknüpft, so müssen auch die Wahrheitswerte verknüpft werden. Dies geschieht über sog. *Wahrheitstafeln*. Üblicherweise werden die Aussagen in die Spaltenüberschriften und die Wahrheitswerte in die Zeilen geschrieben. Über die Wahrheitstafeln kann man jetzt wieder axiomatisch die Wahrheitswerte definieren. Man unterscheidet dabei zwischen *elementaren* logischen Operatoren und *zusammengesetzten* logischen Operatoren. Solche Operatoren werden manchmal auch *Junktoren* genannt.
In nachfolgender Definition werden die Verknüpfungen *nicht*, *und* und *oder* definiert. Wohlgemerkt, es handelt sich dabei wieder um Axiome, also *definierte Wahrheiten*. Wie man aber leicht einsieht, entsprechen diese Definitionen unserer Alltagserfahrung und sind daher als „offensichtliche Wahrheiten" die besten Kandidaten für solch eine Definition.

Definition 2.5 (**elementare logische Operatoren**)
Es seien zwei Aussagen A und B gegeben. Jede dieser Aussagen kann den Wahrheitswert *w* (für wahr) oder *f* (für falsch) annehmen. Wir definieren die Gültigkeit folgender Wahrheitstafel:

A	**B**	**¬A**	**A∧B**	**A∨B**
w	w	f	w	w
w	f	f	f	w
f	w	w	f	w
f	f	w	f	f

Wir nennen ¬A die *Negation* von A, A∧B die *Konjunktion* („Ver-undung“) von A und B und A∨B die *Disjunktion* (Ver-oderung) von A und B.

Die Disjunktion wird auch manchmal *Adjunktion* genannt. Damit haben wir die Operatoren *nicht* (¬), *und* (∧) bzw. *oder* (∨) definiert. Wir werden die Symbole der Aussagen (also A, B etc.) später *Literale* nennen und die aktuellen Wahrheitswerte dann eine *Belegung*.
Die Definition 2.5 zeigt uns, dass es sich bei dem dort eingeführten „oder“ um das sog. „einschließende oder“ handelt, d.h. die Ver-oderung schließt eine Verundung mit ein. Zu unterscheiden davon ist das „entweder-oder“, manchmal auch XOR genannt. Dessen Wahrheitstafel hat in der ersten Zeile der Oder-Verknüpfung ein „f“ stehen, der Rest bleibt gleich.
Neben den elementaren Operatoren gibt es zusammengesetzte Operatoren. Die wichtigsten davon sind wie folgt definiert:

Definition 2.6 (**zusammengesetzte Operatoren**)
Es seien zwei Aussagen A und B gegeben. Wir definieren die Gültigkeit folgender Wahrheitstafel:

A	**B**	**A→B**	**A↔B**
w	w	w	w
w	f	f	f
f	w	w	f
f	f	w	w

Wir nennen A→B eine *Subjunktion* (aus A folgt B) und A↔B eine *Bijunktion* (A genau dann, wenn B).

Diese Operatoren heißen zusammengesetzt, weil, wie man leicht über eine Wahrheitstafel feststellen kann, gilt:

$$A \rightarrow B = (\neg A) \vee B$$
und
$$A \leftrightarrow B = (A \rightarrow B) \wedge (B \rightarrow A)$$

wobei das Gleichheitszeichen hier als "...hat die gleichen Wahrheitswerte wie…" zu interpretieren ist.
Es ist üblich, eine „Rangordnung" von Verknüpfungen zu definieren, so wie das bei der Addition und Multiplikation von Zahlen auch ist. Dort bindet z.B. die Multiplikation stärker als die Addition, weshalb man vereinbart, dass man statt $(x \cdot y)+z$ einfach $x \cdot y+z$ schreibt, also die Klammern weglässt. Man kommt damit überein, dass immer zuerst die Multiplikation und danach die Addition ausgeführt wird. Wenn man es anders möchte, dann muss man entsprechende Klammerungen setzen. Ähnliches gilt auch für logische Operatoren: Die Negation bindet am stärksten, d.h. für $(\neg A) \vee B$ kann man einfach schreiben: $\neg A \vee B$. Die Konjunktion und die Disjunktion sind aber untereinander nicht bevorzugt, hier muss man also immer Klammern setzen (obwohl in der Literatur manchmal zu finden ist, dass die Klammern bei der Konjunktion weggelassen werden und man da dann übereinkommt, dass diese –zumindest schreibtechnisch, um Klammern zu sparen- zuerst ausgeführt wird)! Wer sich nicht sicher ist, dem sei angeraten: Lieber zuviel Klammen setzen als zu wenig.
Wir sahen, dass sich die Subjunktion und die Bijunktion durch Negation, Konjunktion bzw. Disjunktion ausdrücken lassen. Es sei an der Stelle bemerkt, dass bereits die Negation und die Konjunktion (oder auch die Negation und die Disjunktion) ausreichen, um alle anderen Operatoren auszudrücken. Dies kann man durch die Anwendung der (wieder über Wahrheitstafeln leicht selbst nachzuvollziehenden) sog. *de Morgan'schen Regeln* sehen:

$$\neg(A \wedge B) = \neg A \vee \neg B \quad \text{bzw.} \quad \neg(A \vee B) = \neg A \wedge \neg B$$

Damit kann man also z.B. $A \wedge B$ ausdrücken durch $\neg(\neg A \vee \neg B)$, wobei wir aufgrund der Definition 2.5 leicht einsehen, dass $\neg(\neg A) = A$ ist („doppelte Verneinung").
Es gelten noch eine ganze Reihe von algebraischen Gesetzen, wie z.B. das Distributive Gesetz etc., doch darauf kommen wir erst später zurück, wenn wir die algebraischen Eigenschaften logischer Aussagen im Rahmen der Bool'schen Algebra genauer untersuchen.

Definition 2.7 (**Implikation, Äquivalenz**)
Eine tautologische Subjunktion nennen wir eine *Implikation* und eine tautologische Bijunktion nennen wir eine *Äquivalenz*.

Eine Implikation (manchmal auch *Schlussregel* genannt) ist also ein Spezialfall der Subjunktion. „Immer wahre“ Subjunktionen sind Implikationen und „immer wahre“ Bijunktionen sind Äquivalenzen. Dies unterscheidet man auch in der Schreibweise:
$A \Rightarrow B$ stellt eine Implikation dar, man sagt daher auch „A impliziert B“, und $A \Leftrightarrow B$ stellt eine Äquivalenz dar und man sagt auch „A ist äquivalent zu B“.
Im „praktischen Umgang“ werden (z.B. innerhalb von Wahrheitstafeln) die Implikation zunächst aber wie die Subjunktion und die Äquivalenz zunächst wie die Bijunktion behandelt. Wenn man also z.B. beweisen soll, dass irgendwo eine Implikation vorliegt, dann beginnt man ganz normal mit der Wahrheitstafel und stellt die zu zeigende Implikation zunächst als Subjunktion formal in der Tabelle dar. Erst wenn alle Wahrheitswerte eingetragen sind und man dann feststellt, dass in der Spalte der Subjunktion in jeder Zeile nur ein „w“ steht, dann handelt es sich offenbar um eine Tautologie und damit für die Subjunktion um eine Implikation. Entsprechendes gilt für die Äquivalenz. Einander äquivalente logische Ausdrücke haben also immer auch gleiche Wahrheitswerte. Das kennen wir alle ja schon seit unserer Schulzeit, wo z.B. das Umformen von Gleichungen manchmal mit dem Zeichen $\Leftrightarrow$ dargestellt wird. Z.B. ist

$$3x - 12 = 0 \Leftrightarrow$$
$$x = 4$$

Die beiden Zeilen sind nicht gleich (dann müsste ja das selbe dastehen), sondern „nur“ äquivalent. Anders ausgedrückt: die erste Zeile ist genau dann wahr (bzw. falsch) wenn auch die zweite Zeile wahr (bzw. falsch) ist.
Hier nun ein kleines Beispiel, das zeigen soll, dass die Wahrheit einer Implikation nicht unbedingt etwas mit der Wahrheit der implizierten Aussage zu tun haben muss.

Beispiel
Es seien folgende Aussagen gegeben:
1. Ist London nicht in Dänemark, dann ist Paris nicht in Frankreich.
2. Paris ist in Frankreich.

Frage: Folgt daraus, dass London in Dänemark liegt?
Um diese Frage zu beantworten, ist es zunächst erforderlich, diese verbalen Aussagen in aussagenlogische Form zu bringen. Hierzu sind gewisse Konventionen von Nutzen. Eine Liste von Aussagen wird immer als eine Konjunktion der Teilaussagen aufgefasst, d.h. man geht davon aus, dass alle Teile (hier: die Aussagen 1. und 2.) gleichzeitig gelten. Ein „daraus folgt“ wird als Implikation verstanden. Dann müssen wir den Aussagen noch Literale zuordnen. Dabei gilt folgende Faustregel: So wenig Literale wie möglich und keine Literale für zusammengesetzte Aussagen benutzten. Damit legen wir fest:

A = London ist in Dänemark
B = Paris ist in Frankreich
Unser Beispiel lässt sich damit dann so formulieren:

$$[\ (\neg A \rightarrow \neg B) \wedge B\ ] \rightarrow A$$

Die Frage ist jetzt, ob man aus dem $\rightarrow$ ganz rechts ein $\Rightarrow$ machen darf, d.h. ob die Subjunktion tautologisch ist und damit zu einer Implikation wird.
Um dies herauszufinden, machen wir eine Wahrheitstafel.

A	**B**	**¬A**	**¬B**	**¬A → ¬B**	**(¬A → ¬B) ∧ B**	**[(¬A → ¬B) ∧ B] → A**
w	w	f	f	w	w	w
w	f	f	w	w	f	w
f	w	w	f	f	f	w
f	f	w	w	w	f	w

Wir sehen, dass in der letzten Spalte nur noch "w" stehen. Damit handelt es sich hier um eine Tautologie. Und da diese letzte Spalte auf einer Subjunktion basiert, ist diese eine Implikation. Wir dürfen also schrieben:

$$[\ (\neg A \rightarrow \neg B) \wedge B\ ] \Rightarrow A$$

Damit scheinen wir bewiesen zu haben, dass London in Dänemark liegt. Dem ist natürlich nicht so, und deswegen lernen wir hier etwas sehr wichtiges: *Aus der Wahrheit einer Schlussfolgerung dürfen wir nicht allgemein auf die Wahrheit des Geschlussfolgerten schließen!* Manch ein guter Redner verführt leider allzu oft andere Menschen mit genau dieser Taktik. Es wird eine Argumentationskette aufgebaut (die logisch auch gar nicht zu beanstanden ist) und am Schluss das geschlussfolgert, was der Redner gerne darlegen will. Es kann sogar sein, dass es sich dabei wirklich um eine Implikation handelt. Das Problem bei der ganzen Sache ist nur, dass unsere Definition der Subjunktion es zulässt, dass aus etwas Falschem logisch richtig wieder etwas Falsches geschlussfolgert werden kann. Wenn wir z.B. die falsche Aussage 3=5 rechts und links mit der Zahl 2 multiplizieren, so kommt die (ebenfalls falsche) Aussage 6=10 heraus. Man darf daher schreiben:

$$3 = 5 \Rightarrow 6 = 10$$

(man dürfte hier sogar das $\Leftrightarrow$ benutzen). Die Schlussfolgerung ist zwar wahr, doch das Geschlussfolgerte in diesem Beispiel nicht.
Jetzt kann man mit Recht die Frage stellen, was denn dann der Nutzen dieser ganzen Angelegenheit ist. Nun, das Problem stellt sich glücklicher Weise nur, wenn die *Voraussetzung* einer Schlussfolgerung (also die linke Seite vor dem

„⇒“) falsch ist. Ist die Voraussetzung aber wahr, und ist die Schlussfolgerung ebenfalls wahr, dann muss auch das Geschlussfolgerte wahr sein. *Im Falle einer wahren Voraussetzung (Wenn-Teil) kann also aus der Wahrheit der Schlussfolgerung (⇒) auf die Wahrheit des Geschlussfolgerten (Dann-Teil) geschlossen werden.* Oder in Kurzform: Gilt A⇒B, und A ist wahr, dann ist auch B wahr. Wenn A aber falsch ist, dann sagt die Implikation uns praktisch gar nichts, denn dann kann B sowohl wahr oder auch falsch sein. Auf diesem Prinzip basiert alle Naturwissenschaft inklusive der Mathematik. Hier wird immer nur von wahren Voraussetzungen ausgegangen, sodass bei fehlerfreiem Schließen auch immer wieder nur etwas Wahres herauskommt.
Im Rahmen dieser Verfahren, etwas zu „schließen“, haben wir nun einige Regeln benutzt, die in der Logik eigene Namen besitzen. Der Vollständigkeit halber sollen diese Schlussfolgerungsregeln hier mit ihrer „offiziellen“ Namensgebung und z.T. noch mal mit erläuternden Beispielen angegebenen werden.
Zunächst ein Wort zu den Schreibweisen. Es ist in der Aussagenlogik üblich, dass man die Prämissen einer Implikation übereinander schriebt und die Schlussfolgerung darunter, durch einen Strich getrennt. Zum Beispiel für

$$[(A\rightarrow B)\wedge(B\rightarrow C)] \Rightarrow (A\rightarrow C)$$

schreibt man dann übersichtlicher

$$\begin{array}{l} A\rightarrow B \\ \underline{B\rightarrow C} \\ A\rightarrow C \end{array}$$

Unser Beispiel mit London als der Hauptstadt von Dänemark würde man also so schrieben: Statt $[\ (\neg A \rightarrow \neg B) \wedge B\ ] \Rightarrow A$ heißt es dann

$$\begin{array}{l} \neg A \rightarrow \neg B \\ \underline{\quad B \quad} \\ A \end{array}$$

Man darf auch direkt die “verbalen” Aussagen entsprechend hinschrieben, also:

Wenn London nicht in Dänemark liegt, liegt Paris nicht in Frankreich
Paris liegt in Frankreich
London liegt in Dänemark

Man kann auch nicht oft genug wiederholen, dass die Wahrheit der Schlussfolgerung nicht automatisch bedeutet, dass das Geschlussfolgerte wahr ist, wie wir

vorhin schon sahen. Letzteres ist nur dann so (und dann immer!), wenn alle Prämissen wahr sind. In dem letztgenannten Beispiel ist das aber nicht der Fall, denn die erste Prämisse ist nicht wahr (wieso soll aus dem Umstand, wenn London nicht Dänemark liegt, folgen, dass dann auch Paris nicht in Frankreich liegt?). Dennoch ist die Schlussfolgerung als solche (aber nicht das Geschlussfolgerte!) wahr, also eine Implikation. Somit zeigt diese Aussage allenfalls, dass *wenn* wir annehmen, dass die Prämissen stimmen, *dann* auch das Geschlussfolgerte stimmt.
Hier nun die bereits angedeutete „offizielle“ Bezeichnung einiger Schlussregeln, die man leicht über eine Wahrheitstafel verifizieren kann.

Modus ponendo ponens (oder einfach nur Modus ponens):

$$\begin{array}{l} A \rightarrow B \\ \underline{A\quad\quad} \\ B \end{array}$$

Modus tollendo tollens (oder einfach nur Modus tollens):

$$\begin{array}{l} A \rightarrow B \\ \underline{\neg B\quad} \\ \neg A \end{array}$$

Modus tollendo ponens (oder Disjunktiver Syllogismus):

$$\begin{array}{l} A \vee B \\ \underline{\neg A\quad} \\ B \end{array}$$

Modus ponendo tollens:

$$\begin{array}{l} \neg(A \wedge B) \\ \underline{A\quad\quad} \\ \neg B \end{array}$$

Wir haben nun einige Grunddefinitionen der Aussagenlogik kennen gelernt. Mit diesem Wissen wollen wir jetzt eine Abstraktionsstufe höher klettern und uns so ein wenig mit *formaler Logik* beschäftigen. Dies machen wir aus zweierlei Gründen: zum einen vertieft es das Verständnis der Aussagenlogik und zum anderen bereitet es uns auf den Formalismus der Prädikatenlogik für Kapitel 3

vor, wo die entsprechenden Begriffe und Definitionen nur über diesen Formalismus dargestellt werden können.

Definition 2.8 (**Formel**)

(1) Eine Aussage, die nicht weiter mit $\neg$, $\wedge$ bzw. $\vee$ zerlegt werden kann, heißt *Atom* oder auch *irrreduzible* bzw. *atomare Formel*.

(2) Formeln sind rekursiv definiert durch:
- (a) alle Atome sind Formeln
- (b) für alle Formeln F und G sind auch $F \wedge G$ bzw. $F \vee G$ Formeln
- (c) ist F eine Formel, so ist auch $\neg F$ eine Formel
- (d) Formeln sind nur durch (a), (b) oder (c) definiert

Der zweite Teil der Definition mutet vielleicht etwas seltsam an, doch es ist in der Mathematik durchaus üblich, dass Definitionen rekursiv durchgeführt werden. Wir sehen in dieser Definition, dass Formeln im Prinzip Aussagen sind, die aus anderen Aussagen zusammengesetzt sein können. Da Aussagen ja Wahrheitswerte zugeordnet werden können, gilt gleiches für Formeln.

Definition 2.9 (**Belegung**)
Es sei D eine Formel. Wenn eine Funktion $ß: D \rightarrow \{0,1\}$ existiert (wobei 0 für „falsch“ und 1 für „wahr“ steht), dann nennt man $ß$ eine *Belegung* von D.

Die Belegung einer Formel kann also von der Belegung der in ihr enthaltenen Atome abhängen und sich damit auch ändern (wenn sich nämlich die Belegung der Atome ändert).

Beispiel:
Es sei eine Formel gegeben der Form $D = (A \wedge \neg B) \rightarrow C$. Weiter $ß(A)=0$, $ß(B)=1$ und $ß(C)=1$. Wie lautet die Belegung von D? Antwort: $ß(D)=1$.

Definition 2.10 (**Modell**)
Sei F eine Formel und $ß$ eine Belegung. Falls $ß$ für alle in F vorkommenden Atome definiert ist, so heißt $ß$ zu F *passend*. Falls $ß$ passend zu F ist und außerdem gilt: $ß(F)=1$, so heißt $ß$ ein *Modell* für F. Man schreibt dann: $ß \models F$

Ein Modell ist also eine Belegung, die eine Formel „wahr macht“.

Definition 2.11 (**Erfüllbarkeit**)
Eine Formel heißt *erfüllbar*, wenn sie mindestens ein Modell besitzt.

Für eine Formel muss also wenigstens einmal in einer Wahrheitstabelle der Wert „w“ vorkommen. Damit sind Kontradiktionen beispielsweise nicht erfüllbar.

Definition 2.12 (**Gültigkeit**)
Eine Formel F heißt *gültig*, wenn jede passende Belegung ein Modell darstellt. Man schreibt dann: $\models F$

Dies bedeutet, dass eine gültige Formel eine Tautologie darstellt.

Mit diesen Definitionen lässt sich folgender wichtige Satz beweisen:

Satz 2.13
Eine Formel F ist genau dann gültig, wenn $\neg F$ unerfüllbar ist.

Beweis:
Nach Def. 2.12 ist eine Formel genau dann gültig, wenn jede passende Belegung ein Modell ist. Damit ist jede zu $\neg F$ passende Belegung kein Modell. Daher besitzt $\neg F$ kein Modell, was heißt, dass $\neg F$ unerfüllbar ist. (qed)

Eine bekannte Anwendung dieses Satzes sind Widerspruchsbeweise. Ein Beispiel dafür ist die Behauptung, dass die Menge aller Primzahlen P unendlich viele Elemente besitzt, d.h. $|P| = \infty$. Die Formel $|P| = \infty$ ist demnach genau dann gültig, wenn deren Gegenteil, also $|P| < \infty$, unerfüllbar ist. Um letzteres zu zeigen, nimmt man zunächst an, dass $|P| = n < \infty$ ist, wobei nach wie vor P alle Primzahlen enthalten soll. Da P nur endlich ist, kann man schreiben:

$$P = \{p_1, p_2, p_3, ..., p_n\}.$$

Wobei wir o.B.d.A. annehmen, dass die Zahlen geordnet sind.
Jetzt bildet man eine Zahl der Art

$$z = p_1 \cdot p_2 \cdot p_3 \cdot \ldots \cdot p_n + 1$$

und stellt die Frage, ob z eine Primzahl ist. Sicherlich ist z nicht in P enthalten (da größer ist als die „letzte“, größte Primzahl p_n). Andererseits sind Primzahlen ja Zahlen, die nur durch sich selbst oder durch 1 ohne Rest teilbar sind. Die Division von z durch jede andere Primzahl p_i, i=1...n, liefert aber immer den Rest 1, gleiches gilt auch für jedes beliebige Produkt von Primzahlen aus P. Da aber jede Nicht-Primzahl sich mindestens durch eine Primzahl teilen lassen muss (ohne Rest), und dies für z nicht der Fall ist, muss z selbst Primzahl sein.

Damit haben wir eine Primzahl z gefunden, die nicht in P enthalten ist, wo doch aber P die Menge aller Primzahlen war! Die Behauptung also, dass P endlich ist, ist unerfüllbar. Deswegen ist die gegenteilige Behauptung, dass also P unendlich ist, „gültig“ im Sinne der Definition 2.12.

Satz 2.14 (**Substitutionstheorem**)
Es seien F und G äquivalente Formeln. Weiter sei H eine Formel die F als Teilformel enthält. Dann ist H äquivalent zu einer Formel $\hat{H}$, in der jedes Vorkommen von F durch G ersetzt wurde.

Beweis:
Durch Induktion nach Fromelaufbau von H.

Der genaue Beweis wurde hier weggelassen, aber die Idee ist, dass man die Behauptung zeigt, in dem man mit einer Negation beginnt, dann eine Konjunktion und Disjunktion betrachtet und schließt, dass jede Formel ja aus diesen drei Operatoren aufgebaut ist. Das „Gefühl“ sagt einem sowieso, dass es klar ist, dass wenn man innerhalb einer Formel einen Teil durch etwas ersetzt, das die gleichen Wahrheitswerte hat, sich auch die Wahrheitswerte der ganzen Formel dadurch nicht ändern.

Definition 2.15 (**Literal**)
Ein *Literal* ist ein Atom oder die Negation eines Atoms.

Wir hatten bisher atomare Aussagevariable schon als Literale bezeichnet. Def. 2.15 sagt aus, dass auch deren Negationen als Literale bezeichnet werden.

Definition 2.16 (**Normalformen**)
(i) Eine Formel F ist in *konjunktiver Form (KF)*, auch *unvollständige konjunktive Normalform* genannt, wenn sie aus einer Konjunktion von Disjunktionen von Literalen aufgebaut ist
(ii) Eine Formel F ist in *disjunktiver Form (DF)*, auch *unvollständige disjunktive Normalform* genannt, wenn sie aus einer Disjunktion von Konjunktionen von Literalen aufgebaut ist

Formal ausgedrückt heißt dies:

(i) KF: $F = \bigwedge_{i=1}^{n} \left(\bigvee_{j=1}^{m_i} L_{ij} \right)$, wobei $L_{ij} \in \{A_1, A_2 \ldots\} \cup \{\neg A_1, \neg A_2 \ldots\}$

und $A_1, A_2, \ldots$ Atome bezeichnen

(ii) DF: $F = \bigvee_{i=1}^{n} (\bigwedge_{j=1}^{m_i} L_{ij})$, wobei $L_{ij} \in \{A_1, A_2 ...\} \cup \{\neg A_1, \neg A_2 ...\}$

und $A_1, A_2, ...$ Atome bezeichnen

Wenn nun in einer Formel m_i = konstant = Maximalzahl aller vorkommenden Literale ist, so nennen wir die KF auch eine *vollständige konjunktive Normalform* (KNF) und die DF entsprechend eine *vollständige disjunktive Normalform* (DNF). Das Attribut „vollständig" deutet dann nämlich an, dass z.B. bei der KNF in *jedem* der konjungierten Disjunktionsterme immer *alle* Atome der Formel entweder negiert oder unnegiert vorkommen müssen.

Beispiele: (A, B, C und D seien Atome)

1. $F=(A \vee B) \wedge (A \vee \neg B)$ ist eine KNF
2. $F=(\neg A \wedge B \wedge \neg C) \vee (A \wedge D)$ ist eine DF (aber keine DNF)
3. $F=A \vee B$ ist eine KF, eine DF und eine KNF

Allgemein ist jede KNF auch eine KF und jede DNF eine DF (aber natürlich nicht umgekehrt) oder anders ausgedrückt: Die KNF ist ein Spezialfall der KF und die DNF ein Spezialfall der DF.

Satz 2.17
Zu jede Formel F existiert eine äquivalente KF, KNF, DF und DNF.

Beweis:
Durch Induktion nach Formelaufbau von F.

Durch Satz 2.17 ist gewährleistet, dass man jede Formel sozusagen „umformen" kann in eine KNF oder DNF. Dies spielt in der Schaltalgebra eine große Rolle, wie wir gleich sehen werden.

Mit unserer bisherigen Kenntnis ist es nun problemlos möglich, für eine aussagenlogische Formel eine Wahrheitstafel zu erstellen. Dazu müssen wir ja nur alle enthaltenen Atome in eigene Spalten packen und dann sämtliche Teilformeln sukzessive mit den Wahrheitswerten belegen. In der Praxis stellt sich aber oft das umgekehrte Problem: Nicht die Formel ist vorgegeben, sondern eine Wahrheitstafel, die bestimmte schalttechnische Zustände abbildet (z.B. eine Ampelschaltung). Soll so eine Schaltung dann z.B. in einen Chip gebrannt werden, dann ist aber eine aussagenlogische Formel von Nöten, denn die Schaltelemente bestehen ja aus Negations-, Und- bzw. Oder-Gattern (oder Derivaten

davon wie XOR etc.). Die Frage ist also: wie kommt man von einer Wahrheitstafel zu einer diese beschreibende aussagenlogischen Formel? Und wenn so eine logische Schaltung dann noch in hoher Auflage in Produktion gehen soll, ist man natürlich daran interessiert, mit möglich wenigen Schaltbausteinen auszukommen, d.h. man hätte gerne nicht nur einfach eine Formel, sondern auch eine *minimale* Formel. Dies alles sind Probleme, die man mit der sog. *Bool'schen Algebra*, oder, wenn in der Technik angewendet, auch *Schaltalgebra* genannt, lösen kann. Daher wollen wir uns zum Abschluss dieses Kapitel damit etwas beschäftigen.

Wenn man auf die Elemente einer Menge mittels Operatoren algebraische Verknüpfungen definiert (z.B. die Menge der ganzen Zahlen mit Addition und Multiplikation), so nennt man das ganze eine *algebraische Struktur*. Nachfolgend seien spezielle algebraische Strukturen betrachtet, die für uns nützlich sind.

Definition 2.18 (**Verband**)
Eine algebraische Struktur, die aus einer Menge V sowie zwei zweistelligen Verknüpfungen $\sqcap$ und $\sqcup$ auf V besteht, heißt *Verband*, wenn für $a,b,c \in V$ folgende Axiome gelten:

1. Kommutativgesetze:

   $a \sqcap b = b \sqcap a$ und $a \sqcup b = b \sqcup a$

2. Assoziativgesetze:

   $(a \sqcap b) \sqcap c = a \sqcap (b \sqcap c)$ und $(a \sqcup b) \sqcup c = a \sqcup (b \sqcup c)$

3. Absorptionsgesetze:

   $a \sqcap (a \sqcup b) = a$ und $a \sqcup (a \sqcap b) = a$

Für den Verband schreiben wir $(V, \sqcap, \sqcup)$.

Welche konkreten Verknüpfungszeichen man für $\sqcap$ und $\sqcup$ wählt, ist zunächst ohne Belang. Wichtig ist allein, dass die obigen Axiomen gelten, um von einem Verband zu sprechen. Man sieht z.B. schnell ein, dass die „normale" Addition und Multiplikation, also + und $\bullet$, nicht für $\sqcap$ und $\sqcup$ verwendet werden können, wenn V eine Zahlenmenge ist, da dann offenbar das Absorptionsgesetz nicht gilt (während die ersten beiden Axiome gelten). Benutzt man aber beispielsweise für V eine Menge, die selbst aus Mengen besteht und für $\sqcap$ und $\sqcup$ die üblichen Verknüpfungen $\cap$ und $\cup$ für Durchschnitt und Vereinigung, so handelt es sich dabei offensichtlich um einen Verband $(V, \cap, \cup)$. Eine wieder andere Möglichkeit ist, die Menge $V=\{w, f\}$ oder $V=\{0,1\}$ zu definieren, also als Menge der

beiden Wahrheitswerte wahr (w oder 1) und falsch (f oder 0). Wenn man für $\sqcap$ und $\sqcup$ dann $\wedge$ und $\vee$ wählt, so erhält man ebenfalls einen Verband $(\{w, f\}, \wedge, \vee)$.

Definition 2.19 (**Distributiver Verband**)
Ein Verband $(V,\sqcap, \sqcup)$ heißt *distributiv*, wenn für ihn zusätzlich folgende Axiome für $a,b,c \in V$ gelten:

$$a\sqcap(b\sqcup c) = (a\sqcap b)\sqcup(a\sqcap c) \qquad \text{und} \qquad a\sqcup(b\sqcap c) = (a\sqcup b)\sqcap(a\sqcup c)$$

Man sieht an den Distributivgesetzen (wie zuvor auch schon in der Definition der Verbandsaxiome), dass $\sqcap$ und $\sqcup$ völlig gleichberechtigt sind. Anders als bei der Multiplikation und Addition von Zahlen, wo die Multiplikation stärker bindet als die Addition, liegt hier eine Dualität zwischen den beiden Verknüpfungsstrukturen vor.
Die vorher als Beispiele zitierten Mengenverbände und Aussagenverbände sind auch distributive Verbände.

Definition 2.20 (**Komplementär distributiver Verband, Boolesche Algebra**)
Ein distributiver Verband $(V,\sqcap, \sqcup)$ heißt *komplementär* oder auch *Boolescher Verband* oder auch *Boolesche Algebra*, wenn zusätzlich folgende Axiome gelten:

1. Es existiert für alle $a \in V$ ein neutrales Element $e \in V$ („Einselement“) bezüglich der Verknüpfung $\sqcap$ und ein neutrales Element $n \in V$ („Nullelement“) bezüglich der Verknüpfung $\sqcup$ mit den Eigenschaften

   $$a\sqcap e = a \qquad \text{und} \qquad a\sqcup n = a$$

2. Zu jedem $a \in V$ existiert ein $\bar{a} \in V$ („Komplement“ von a) mit den Eigenschaften

   $$a\sqcap \bar{a} = n \qquad \text{und} \qquad a\sqcup \bar{a} = e$$

Für einen Booleschen Verband schreiben wir dann $V(,\sqcap, \sqcup, \bar{})$.

Boolesche Verbände spielen eine ganz wichtige Rolle in der Schaltalgebra. Wir erwähnten bereits, dass dort häufig die Frage nach einer möglichst minimalen Schaltung für ein Problem gesucht wird, für das eine Wahrheitstafel vorliegt. Es soll also möglich sein, ausgehend von einer Wahrheitstafel „rückwärts“ eine dazu passende (und noch möglichst kleine) logische Formel zu finden. Die Ausdrucksweise „...eine...“ deutet an, dass das Ergebnis im Allgemeinen nicht eindeutig ist, d.h. zu der gleichen Wahrheitstafel kann es viele äquivalente Formeln

geben, die diese hervorbringen können. In der Praxis ist man natürlich schon froh, wenn man eine davon gefunden hat. Wie das funktioniert, soll an einem Beispiel demonstriert werden. Dabei wird auch der Nutzen von DNF und KNF sichtbar.
Zu diesem Zweck vereinbaren wir rein schreibtechnisch folgendes: Im Booleschen Verband ordnen wir $\sqcap$ das Symbol $\cdot$ und $\sqcup$ das Symbol + zu. Für die Wahrheitswerte nehmen wir 0 und 1, sodass der Boolesche Verband dann so aussieht: $(\{0,1\}, \cdot, +, \bar{})$. Dabei steht das Produktzeichnen für das logische *und* ($\wedge$) und das Additionszeichen für das logische *oder* ($\vee$). Das hat den Vorteil, dass man „fast" so rechnen kann wie mit normalen Zahlen. Man muss nur immer bedenken, dass die beiden Verknüpfungen dual sind, d.h. hier ist die „Multiplikation" und die „Summe" strukturell gleichwertig. Z.B. für die beiden Distributiven Gesetzte heißt das:

$a(b+c) = ab+ac$ (so wie immer), **aber** auch:

$a+bc = (a+b)(a+c)$.

Dann gibt es hier z.B. noch die Absorptionsgesetzte, die dann so aussehen:

$a(a+b) = a$ und $a+ab = a$

Dann haben wir ja auch die komplementären Elemente, also

$a\,\bar{a} = 0$ und $a+\bar{a} = 1$.

Das Einselement wird einfach 1 genannt und das Nullelement 0.
Sicherheitshalber setzt man lieber zuviel Klammern als zu wenig!
Nun also zu einem konkreten Beispiel, wo die Problembeschreibung zu einer Wahrheitstafel führt, für die dann eine minimale aussagenlogische Formel gefunden werden soll.

Beispiel:
An einem Belüftungssystem einer Industrieanlage soll eine Warnleuchte angebracht werden, die rot aufleuchtet, wenn die Räume mangelhaft belüftet sind, ansonsten leuchtet sie grün. Zur Belüftung stehen vier unabhängige, jedoch in der Kapazität verschiedene Gebläse zur Verfügung, die im Verbundsystem arbeiten (d.h. jedes System ist mit einem Raum verbunden). Über die Kapazität der Größen ist folgendes bekannt:
a) Es müssen mindestens zwei Gebläse arbeiten

b) Sobald Gebläse A ausfällt, muss Gebläse B arbeiten, ansonsten reicht die Belüftung nicht aus.

Konstruieren Sie eine möglichst einfache Schaltung, die die Warnleuchte auf „rot“ umschaltet, wenn der verbotene Zustand eintritt

Lösung:

Die Gebläse seien A, B, C und D genannt. Wenn in Betrieb, dann wird der Wert 1, sonst 0 zugewiesen. Die Kontrolllampe nennen wir L. Wenn die Warnleuchte grün ist, soll sie den Wert 1 erhalten, wenn sie rot aufleuchtet, bekommt sie den Wert 0. Aus den Bedingungen a) und b) ergibt sich folgende Wahrheitstafel:

A	**B**	**C**	**D**	**L**
1	1	1	1	1
1	1	1	0	1
1	1	0	1	1
1	1	0	0	1
1	0	1	1	1
1	0	1	0	1
1	0	0	1	1
1	0	0	0	0
0	1	1	1	1
0	1	1	0	1
0	1	0	1	1
0	1	0	0	0
0	0	1	1	0
0	0	1	0	0
0	0	0	1	0
0	0	0	0	0

Es stellt sich also das Problem, für L eine (minimale) Formel in Abhängigkeit von A, B, C und D zu finden. Um die Minimalität kümmern wir uns später, zunächst soll erst einmal überhaupt eine Formel gefunden werden. Hierzu kommen uns die KNF und DNF zu Hilfe, und zwar aus folgendem Grund:

In vier „Variablen“ hat eine vollständig konjunktive Normalform immer die Gestalt der Art, dass jeder konjungierte Disjunktionsterm alle 4 Variablen (negiert oder unnegiert) enthalten muss. Z.B. für 4 beliebige Variablen r, s, t, und p könnte dies vielleicht so aussehen:

$$(p+\bar{s}+\bar{r}+q)(\bar{p}+s+\bar{r}+q)(p+\bar{s}+r+q)...$$

Hat man jetzt Werte der Variablen derart, dass immer einer der Klammerterme gleich 0 wird, z.B. hier für p=1, s=0, r=1 und p=0 (dann wird die zweite Klammer gleich Null), so ist das Produkt (die Konjunktion) aller Klammern auch immer gleich Null. In unserem Beispiel könnte man jetzt also folgendes machen: Mann könnte diejenigen Zeilen der Tabelle betrachten, für die L den Wert 0 bekommt und damit eine KNF ansetzen, in dem man die 4 Variablen A, B, C, D so hinschreibt, dass immer, wenn die Variable = 0 ist, man sie belässt, und wenn sie =1 ist, man sie mit einem Querstrich darüber (Negation) versieht. Setzt man dann nämlich die aktuelle Belegung dieser „Null-Zeile" ein, so wird der Term selbst gleich Null. Multipliziert man alle solche Null-Zeilen miteinander, hat man die gesuchte Formel. Konkret sieht unsere so konstruierte Formel dann folgendermaßen aus:

$$L = (\overline{A} + B + C + D)(A + \overline{B} + C + D)(A + B + \overline{C} + \overline{D}) \bullet$$
$$(A + B + \overline{C} + D)(A + B + C + \overline{D})(A + B + C + D)$$

Für die erste Zeile, wo L=0 ist, hat man die Wahrheitswerte aus der Tabelle für A, B, C, und D also dem ersten Klammerausdruck der obigen Formel zugewiesen. Wenn man diese Wahrheitswerte da einsetzt, kommt für den ersten Klammerausdruck der Wert 0 heraus und damit für ganz L=0. Entsprechend geht es dann weiter mit der zweiten „Null-Zeile" von L, so dass wir folgerichtig für die sechs Zeilen aus der Wahrheitstafel, wo L=0 wird, auch sechs Klammerterme in der Formel für L erhalten. Somit haben wir mittels der KNF also eine Formel für L gefunden. Es sei dem Leser überlassen, für diese Formel durch Ausmultiplizieren und Anwenden der Axiome (wie Absorptionsgesetze etc.) für Boolesche Verbände eine Minimalform für L zu finden. Die Lösung jedenfalls dafür ist:

$$L = AB + AD + AC + BC + BD$$

Man hätte statt der KNF auch die DNF benutzen können, denn diese verhält sich „spiegelbildlich" zur KNF: Bei der DNF hat man ja eine Disjunktion von Konjunktionstermen. In der Wahrheitstafel kann man dafür jetzt statt der „Null-Zeilen" die „Eins-Zeilen" nehmen, denn die DNF enthält ja in unserer Schreibweise eine Summe von Produkten, wobei jeder Summand immer jede der vier Variablen A, B, C, und D (wieder negiert oder unnegiert) enthält. Ist der den Summand bildende Produktterm der 4 Variablen jetzt so gewählt, dass die Belegung einer „Eins-Zeile" aus der Wahrheitstafel den Wert 1 ergibt, so liefert dieser Term als Summand zusammen mit den anderen immer den Gesamtwert 1 für L. In unserer Wahrheitstafel würde man so z.B. für die erste Zeile, wo L=1 ist, für den ersten Summanden von L den Produktterm erhalten:

$$ABCD$$

Für die nächste Zeile mit L=1 erhalten wir entsprechend als Term

$$AB\overline{CD}$$

usw., sodass schließlich die DNF die Form erhält:

$$L = ABCD + AB\overline{CD} + ...$$

Wenn hier einer der Summanden =1 ist, wird L=1. Die Abbildungsvorschrift ist hier also die: Wenn in einer „Eins-Zeile" die Belegung einer Variablen = 0 ist, so nimmt diese negiert als Faktor in den Produktterm auf. Ist die Belegung = 1, so lässt man die Variable unnegiert. Dadurch ist gewährleistet, dass diese Zeile bei der entsprechenden Variablenbelegung den Wert 1 erhält.
In der Praxis geht man meistens so vor, dass man immer dann, wenn weniger „Eins-Zeilen" als „Null-Zeilen" vorkommen, die DNF benutzt und ansonsten die KNF. Damit sind die „Start-Terme" für die anschließende Minimierung bereits die kleineren.
Es gibt übrigens eine Möglichkeit, eine Minimalform aus einer KNF oder DNF auch „grafisch" recht elegant ganz ohne Rechnerei zu bestimmen, und zwar über die sogenannten KV-Diagramme (steht für Karnaugh-Veitch-Diagramme). Es würde allerdings den Rahmen dieses Buches sprengen, darauf näher einzugehen.
Nachdem wir nun die Grundlagen der Aussagenlogik behandelt haben, wenden wir uns im folgenden Kapitel der nächsten Abstraktionsstufe zu: der Prädikatenlogik.

Übungen zum Selbsttest:

1. Man beweise oder gebe ein Gegenbeispiel:
   a) Falls F -> G gültig ist und F gültig ist, so ist auch G gültig.
   b) Falls F -> G gültig ist und F erfüllbar, so ist G erfüllbar.
   c) |= ([|= (F∧G)] -> [|= (F -> G)])

2. Zeigen Sie, warum folgende Aussagen logische Probleme machen und worin diese Probleme bestehen:
   a) Alles ist relativ (d.h. es gibt keine absolute Wahrheit)
   b) Weil es keinen absoluten Maßstab für Gut und Böse gibt, ist die Welt schlecht.
   c) Wenn Gott die Logik transzendiert, dann braucht sie für Gott nicht zu gelten. Wenn also die Logik nicht gilt, dann muss auch nicht alles wahr sein was er uns offenbart.

3. Zeigen Sie, dass folgende Aussage eine Kontradiktion darstellt:
   „Weil es keine absoluten moralischen Gesetze gibt, soll man die (relativen) moralischen Gesetze anderer Kulturen tolerieren."

4. In der Philosophie gibt es das sog. „Theodizee-Problem". Hierbei geht es um die logische Vereinbarkeit folgender zweier Aussagen:
   a) Es gibt einen allmächtigen und allgütigen Gott
   b) Es gibt (moralisch) Böses in der Welt
   Für Atheisten sind diese beiden Aussagen widersprüchlich, denn sie argumentieren, dass ein allmächtiger Gott in der Lage sein müsste, das Böse in der Welt abzuschaffen, und dass er es wegen seiner unendlichen Güte auch tun müsste. Da aber Böses in der Welt existiert, sehen Atheisten darin den Beweis, dass es so einen Gott nicht geben kann. Zeigen Sie, dass diese beiden Aussagen in Wirklichkeit keinen logischen Widerspruch darstellen, denn es ist möglich, diese beiden Aussagen logisch zu „harmonisieren", in dem weitere Annahmen, die keiner der beiden Aussagen widersprechen, hinzugenommen werden können und damit das ganze logisch konsistent bleibt.
   (Hinweis: Die beiden zunächst widersprüchlich erscheinenden Aussagen
   a) Es gibt keine Strafen
   b) Es kommen keine Diebstähle vor
   kann man dadurch harmonisieren, in dem man eine weitere Annahme trifft, nämlich:
   c) Alle haben genug und sind anständige Menschen
   Leider leben wir nicht in dieser Welt, doch sie ist logisch möglich.)

5. Zur Bergung des Wracks "Charles the Lion" werden drei Hebewerke eingesetzt, ein viertes und fünftes dient dazu, extreme Belastungen auszugleichen. Hierbei soll folgendermaßen vorgegangen werden:
   - Überschreitet die Belastung einen gewissen Wert (Sollwert) an mindestens einem Hebewerk, so wird das vierte Hebewerk hinzugeschaltet.
   - Wird der Sollwert an genau zwei Hebewerken überschritten, so wird auch noch das fünfte Hebewerk hinzugeschaltet.
   - Sind alle drei Ausgangshebewerke überlastet, so ist die Bergung abzubrechen.

   Entwickeln Sie für das vierte und fünfte Hebewerk eine geeignete Schaltung.

6. Von zwei Krankheiten C und D und zwei Krankheitssymptomen S und T ist folgendes bekannt:
   a) Wenn der Patient an Krankheit C leidet und nicht an D, so muss er Symptom T aufweisen.
   b) Wenn mindestens eines der beiden Symptome auftritt, so leidet der Patient an mindestens einer der beiden Krankheiten.
   c) Tritt Symptom S nicht auf, so kann die Krankheit D nicht vorliegen.
   d) Leidet der Patient an Krankheit D, aber nicht an C, so kann das Symptom T nicht auftreten.

   Welche Diagnose kann aufgrund dieser medizinischen Kenntnisse gegeben werden, wenn der Patient (i) beide Symptome, (ii) nur Symptom S, (iii) nur Symptom T, (iv) keines der beiden Symptome aufweist?

3. Prädikatenlogik

Nachdem wir nun die Grundprinzipien der Aussagenlogik kennen, kann diese nun erweitert werden. Bisher waren Aussagen letztendlich immer durch Atome zusammensetzbar. Die Prädikatenlogik „zoomt" nun gewissermaßen in das Innere der aussagenlogischen Atome hinein, in dem sie ihnen eine innere Struktur zuweist.

Definition 3.1 (**Prädikat**)
Unter einem n-stelligen *Prädikat* $p(x_1, ..., x_N)$ mit den Individuen $x_1, ..., x_N$ versteht man eine Aussage, welche bei geeigneter Individuenbelegung wahr wird.

Bei Prädikaten handelt es sich im Prinzip um binärwertige Funktionen. Der Rückgabewert eines Prädikats ist also immer nur *wahr* oder *falsch*. Die Individuen sind Parameter, welche Variablen oder Konstanten sein können. Im Sinne der Aussagenlogik sind Prädikate immer atomare Aussagen (die also nicht weiter logisch durch *und, oder* bzw. *nicht* zerlegt werden können). Beispiel für ein Prädikat wäre:

isst_gerne(karl, schwein).

Hier haben wir ein Prädikat *isst_gerne* und zwei Individuenkonstanten, nämlich *karl* und *schwein*. Wenn Karl tatsächlich gerne Schwein isst, dann erhält dieses Prädikat den Wahrheitswert „wahr" (w oder 1), ansonsten „falsch" (f oder 0). Es ist in der Praxis üblich, Individuenkonstanten klein und Individuenvariable groß zu schreiben. Benutzt man Variable, so stellen diese wirklich Variable im mathematischen Sinn dar. Der Sprachgebrauch „Variable" in einem Computerprogramm ist ansonsten nämlich nicht ganz richtig, da die sog. Variablen dort eigentlich Pseudo-Namen für Speicherzellen darstellen und deren „Wert" zu jeder Zeit bekannt ist (da der Wert ja dem Inhalt der Speicherzelle entspricht, und da muss immer etwas drinstehen).
Die Benutzung von Individuenvariablen in einem Prädikat könnte z.B. so aussehen:

isst_gerne(karl,Y).
isst_gerne(X, schwein).
isst_gerne(X, Y).

Hier haben wir also die Individuenvariablen *X* und *Y*, welche bei geeigneter Belegung das Prädikat *wahr* machen.

Als Variablen lassen wir hier aber keine Prädikate zu, also es gibt keine Prädikate von Prädikaten. Diese Einschränkung ist nicht zwingend, doch wenn man sie trifft (so wie wir), so spricht man von Prädikatenlogik *erster Stufe*.
Ein weiteres Merkmal der Prädikatenlogik ist die Tatsache, dass wir jetzt nicht nur Individuen haben, sondern auch quantitative Aussagen über diese Individuen treffen können. Neben den aus der Aussagenlogik schon bekannten Junktoren $\neg, \wedge, \vee, \rightarrow, \leftrightarrow, \Rightarrow$ und $\Leftrightarrow$ führen wir noch zwei sog. "Quantoren" ein. Sie lauten:

$\forall$ ("für alle") und

$\exists$ ("es gibt").

Definition 3.2 (**Term**)
1. Jede Individuenkonstante ist ein Term.
2. Jede Individuenvariable ist ein Term.
3. $F(t_1, ..., t_N)$ ist ein Term, wenn $t_1, ..., t_N$ Terme und F ein n-stelliges Funktionensymbol sind.
4. Terme sind immer gemäß 1.-3. gebildet.

Die Definition des Begriffs Term ist wieder rekursiv. Wir sagten ja schon, dass ein Prädikat eigentlich eine binärwertige Funktion von Individuen ist. Die in der Term-Definition eingeführte Funktion F kann z.B. konkrete Werte (Individuenkonstanten) für gewisse Individuenvariable liefern. Dies ist insbesondere für die Programmierung von Expertensystemen wichtig (vgl. Kapitel 4 und 5), denn dort will man meistens konkrete Lösungen für ein Problem, d.h. es sollen Individuenvariable durch konkrete Objekte z.B. aus einer Datenbank oder Wissensbasis belegt werden (wobei das zugehörige Prädikat natürlich wahr bleiben muss). Wichtig ist allerdings, dass diese Funktionen selbst keine Prädikate sein dürfen, sondern immer nur wieder Individuen als „Rückgabewert“ besitzen.
Mit Hilfe von Termen lassen sich dann die Prädikate etwas verallgemeinern.

Definition 3.3 (**Atom**)
$p(t_1, ..., t_N)$ ist ein *Atom*, falls p ein n-stelliges Prädikatensymbol ist und t_1 bis t_N Terme sind.

Die einfachste Version eines Atoms ist natürlich das in Def. 3.1 eingeführte Prädikat selbst. In besagter Definition waren aber nur Individuen als Parameter zugelassen. In Definition 3.3 können aber auch Terme als Parameter möglich sein.

Es sei noch angemerkt, dass wir in der Aussagenlogik auch schon Atome definierten. Die Gemeinsamkeit mit der Prädikatenlogik besteht u.a. darin, dass hier wie da Atome nicht weiter (mittels Junktoren) zerlegbar sind, d.h. sie sind irreduzibel.
Damit können wir jetzt wie schon in der Aussagenlogik den Begriff einer Formel auch hier für die Prädikatenlogik einführen.

Definition 3.4 (**Formel**)
1. Atome sind Formeln.
2. $\neg$A ist ein eine Formel, falls A eine Formel ist.
3. (A$\wedge$B) bzw. (A$\vee$B) sind Formeln, falls A und B Formeln sind.
4. $\exists$x(A) bzw. $\forall$y(A) sind Formeln, wenn A und B Formeln und x und y Individuenvariablen sind.
5. Formel werden immer nur nach 1.-4. gebildet.

Betrachten wir ein paar Beispiele für Formeln:

a) $F = \forall x(\neg p(x))$
b) $F = \forall x((\exists y\ p(x,y)) \wedge (\exists z\ q(x,z)))$
c) $F = \forall x\ (gerade(x) \wedge \neg ungerade(x))$

p(...), q(...), gerade(...) und ungerade(...) sind hier offenbar Prädikate. Wir sehen, dass in a) und b) das gleiche Prädikat p(...) mit einer verschiedenen Anzahl von Individuen benutzt wird. So etwas wird in der Praxis jedoch vermieden, da in logischen Programmiersprachen wie z.B. PROLOG oder LISP derartiges nicht zulässig ist.

Als nächstes wenden wir uns einem wichtigen Begriff aus der Prädikatenlogik zu, der so genannten Struktur. In der Aussagenlogik hat man normalerweise Aussagevariablen wie z.B.
A = „Es regnet“ oder
B = „Die Strassen sind nass“.
Die Variablen A und B können nun jeweils die Wahrheitswerte *wahr* oder *falsch* annehmen. Die aktuellen Werte der Aussagevariablen heißen eine Belegung. Bei zwei Aussagevariablen gibt es *4* verschiedene Belegungsmöglichkeiten. Hat man *n* Aussagevariablen, so gibt es 2^n Möglichkeiten, also immer nur endlich viele. In der Prädikatenlogik allerdings ist der Sachverhalt nicht ganz so einfach. Ein Prädikat kann zwar selbst auch immer nur *wahr(=1)* oder *falsch(=0)* sein, jedoch kann es bis zu unendlich viele Möglichkeiten für die im Prädikat vorhanden Individuen geben, und jede dieser Kombinationen kann das Prädikat wahr oder falsch machen. So kann z.B. ein Prädikat ein oder mehrere Individuenvari-

ablen aus der Menge der natürlichen oder auch reellen Zahlen beinhalten. Da es unendlich viele solcher Zahlen gibt, ist der Begriff der Belegung hier nicht so einfach fassbar. Das prädikatenlogische Äquivalent zu der aussagenlogischen Belegung heißt eine Struktur und ist wie folgt definiert:

Definition 3.5 (**Struktur**)
Eine *Struktur* ist eine binäre Funktion *A* über das Paar (U_A, I_A), wobei U_A eine beliebige, nichtleere Menge ist, welche auch die Grundmenge oder das Universum von *A* genannt wird, und I_A eine Abbildung bezeichnet, die

- jedem k-stelligen Prädikatensymbol ein k-stelliges Prädikat über U_A zuordnet
- jedem k-stelligen Funktionensymbol eine k-stellige Funktion über U_A zuordnet
- jeder Individuenvariablen ein Element aus U_A zuordnet.

Betrachten wir z.B. U_A = Menge der Natürlichen Zahlen und sei I_A diejenige Funktion, welche dem Prädikatensymbol "gerade" dasjenige Prädikat zuordnet, welches genau dann wahr ist, wenn eine Zahl aus U_A durch *2* ohne Rest teilbar ist. Dann ist $A(U_A, I_A)$ eine Struktur für die Formel

$$\exists x \, \forall y \, [(\text{gerade}(x) \wedge x=y^2) \text{ -> } \text{gerade}(y)]$$

Wir lernten in der Aussagenlogik den Begriff des Modells kennen. Auch hier gibt es das prädikatenlogische Äquivalent:

Definition 3.6 (**Modell**)
Eine Struktur *A* mit *A(F) = 1* für eine Formel *F* heißt *Modell* für *F*. Man sagt: *A* ist Modell für *F* und schreibt: $A \models F$.

Auch das prädikatenlogische Äquivalent für den Begriff der Gültigkeit gibt es:

Definition 3.7 (**Gültigkeit**)
Wir nennen eine Formel F *gültig*, wenn jede Struktur ein Modell für F ist und schreiben: $\models F$.

Der Begriff des Modells ist von zentraler Bedeutung, denn Formeln, die kein Modell besitzen, sind offenbar nicht erfüllbar. Die Formel $F = X \wedge \neg X$ ist nicht erfüllbar und besitzt daher kein Modell. So eine Formel nennt man –wie in der Aussagenlogik auch- auch eine Kontradiktion. Es kann nichts gleichzeitig mit seinem Gegenteil existieren. Im Gegensatz dazu ist die Formel $F = X \vee \neg X$

immer erfüllbar, also gültig. Gültige Formeln nennt man wie gehabt Tautologien.

Im Zusammenhang mit Formeln *F* und *G* gelten einige wichtige Rechenregeln:

1.) $\neg (\forall x F) \Leftrightarrow \exists x \neg F$
$\neg (\exists x F) \Leftrightarrow \forall x \neg F$
2.) $\forall x \forall y F \Leftrightarrow \forall y \forall x F$
$\exists x \exists y F \Leftrightarrow \exists y \exists x F$
3.) $\forall x F \wedge \forall x G \Leftrightarrow \forall x (F \wedge G)$
$\exists x F \vee \exists x G \Leftrightarrow \exists x (F \vee G)$

Eine wichtige Anwendung der Prädikatenlogik ist die Erstellung von sog. Expertensystemen. Das sind Softwaren, die selbständig logische Schlussfolgerungen ziehen können. Ein „Wissensingenieur“ ist dabei eine Person, die das zu lösende Problem analysiert und die für die programmtechnische Realisierung erforderliche Information einholt. Die Aufgabe des Wissensengineering ist es u.a., dass die Fakten und Regeln in eine Form gebracht werden, die von einer entsprechenden logischen Programmiersprache verstanden wird. Zu diesem Zweck ist eine Überführung umgangsprachlicher Aussagen in die formale Prädikatenlogik notwendig. Dies kann durch Anwendung der bisherigen Regeln geschehen, doch es sind noch weitere Umformungen nötig, damit eine logische Sprache wie z.B. PROLOG oder LISP dies versteht. Hierzu definieren wir also weiter:

Definition 3.8 (**Substitution**)
Sei *F* eine Formel, *t* ein Term und *x* eine Variable. Unter der *Substitution F[x/t]* versteht man diejenige Formel, welche durch Ersetzung der Variablen *x* in jedem Vorkommen durch *t* entsteht.

Wir werden sehen, dass solche Substitutionen später einer wichtige Rolle spielen.

Definition 3.9 (**Bereinigung**)
Eine Formel *F* heißt *bereinigt*, wenn es keine Variable mehr gibt, die in *F* sowohl gebunden (durch ∀,∃) als auch frei vorkommt und hinter allen vorkommenden Quantoren verschiedene Variablen stehen.

Ein erfahrener Wissensingenieur wird von vornherein dafür sorgen, dass die erzeugten Aussagen bereits bereinigt sind.

Ein Beispiel für eine zu bereinigende Formel wäre

$$F = \forall x \exists y P (x, f(y)) \wedge \forall y (Q(x, y) \vee P(x))$$

Hier kommt die Variable y an zwei verschiedene Quantoren gebunden vor. Die Variable x kommt im vorderen Teil gebunden vor, im hinteren dagegen frei. Durch geeignetes Umbenennen kann man diese Formel bereinigen:

$$F= \forall x \exists y P (x, f(y)) \wedge \forall z (Q (w, z) \vee P(w))$$

Die Aussage selbst ist dabei unverändert geblieben.

Definition 3.10 (**Pränex-Form**)
Eine Formel *F* heißt *pränex* (oder in Pränexform), falls sie von der Bauart $Q_1y_1Q_2y_2 \ldots Q_ny_nG$ ist, wobei $Q_i \in \{\exists, \forall\}$ Quantoren und y_i Variablen sind, und in der Formel *G* keine weiteren Quantoren mehr vorkommen (alle Quantoren stehen damit ganz links).

Pränexformen sind für bereinigte Formeln leicht zu erzeugen: Da jeder Quantor sich immer nur auf genau eine Variable bezieht, kann man diese einfach ganz nach links „ziehen“, wenn man mit Hilfe der obigen Rechenregeln zuerst dafür gesorgt hat, dass alle Negationen ganz nach „innen“ gezogen wurden, also keine Negation mehr auf einen Quantor wirkt. Obiges Beispiel lautet in Pränexform:

$$F = \forall x \exists y \forall z \ [P (x, f(y)) \wedge (Q (w, z) \vee P(w))]$$

Man kann allgemein zeigen, dass es zu jeder prädikatenlogischen Formel eine äquivalente Pränexform gibt.
Der nächste Schritt besteht in der Eliminierung der ∃-Quantoren. Dazu wird angenommen, dass die Formel bereits in bereinigter Pränexform vorliegt. Diese Eliminierung erfolgt sukzessive von links nach rechts. Es wird von links ausgehend bis zum ersten ∃-Quantoren vorgedrungen. Die Variable, die dahinter steht, wird nun in dem rechts davon stehenden Formelteil ersetzt (Substitution) durch eine Funktion, wobei diese Funktion von allen denjenigen Variablen abhängig ist, die links des ∃-Quantors an ∀-Quantoren gebunden sind. Hat man dies getan, geht man weiter von links nach rechts zum nächsten ∃-Quantor und verfährt genau wie zuvor. Dieser Vorgang wird solange wiederholt, bis keine ∃-Quantoren mehr vorhanden sind. Das Ergebnis nennt man dann eine Skolemform.
Genauer:

Definition 3.11 (**Skolemform**)
Sei eine Formel in bereinigter Pränexform. Das Ergebnis folgender Umformungen nennt man *Skolemform* (eliminieren aller ∃-Quantoren von links nach rechts). Algorithmus:
Wiederhole solange, bis *F* keine Existenzquantoren mehr enthält
{
F habe die Form $\forall y_1 \; \forall y_2 \ldots \; \forall y_k \; \exists z \; H$ (*H* kann weitere ∀ und ∃ enthalten), $k \geq 0$.
Bilde mit Hilfe eines neuen in *H* bisher nicht vorkommenden Funktionssymbols *f* folgende Substitution:
$F = \forall y_1 \; \forall y_2 \ldots \; \forall y_k \; H \; [\; z \; / \; f \; (y_1 \ldots y_k) \;]$
}

Der Hintergrund dieser Vorgehensweise liegt auf der Hand: Die Forderung des ∃-Quantors, dass es nämliche eine Variable *gibt* mit der Eigenschaft wie in dem Formelteil beschrieben, korrespondiert mit der realen Möglichkeit, diese Variable auch zu bestimmen. Es *muss* also eine Funktion existieren, welche die Bestimmung des Variablenwerts der ∃-Variablen ermöglicht, wobei im „worse case“ diese Funktion von allen links davor stehenden ∀-Variablen abhängig sein kann. Gerade wenn solche Aussagen in ein Computerprogramm implementiert werden sollen, so muss hier der Algorithmus natürlich eine konkrete Möglichkeit haben, diese ∃-Variablenwerte auch zu bestimmen, sei es durch mathematische Berechnung oder durch Auffinden in einer Tabelle oder ähnliches.
Schließlich werden in der Praxis aus schreibtechnischen Gründen häufig noch alle ∀-Quantoren einschließlich der dahinter stehenden Variablen weggelassen, denn sonst müsste man in der Programmiersprache noch eigene Symbole dafür vorsehen. Die Anwendung dieser Konvention führt dann zu einer quantorenfreien Form der ursprünglichen Formel. Solche Formeln nennt man auch universell quantifiziert. Hier noch ein Beispiel für die Erzeugung einer Skolemform, welche schließlich noch universell quantifiziert wird:

$$F = \forall \, x \, \forall \, y \, \exists \, z \; [\; (p(z) \wedge \; \forall \, y \; q \; (\, x, \, y, \, z) \;) \vee (\neg \, \forall \, z \; r \; (z) \;) \;]$$

Bereinigen:

$$F = \forall \, x \, \forall \, y \, \exists \, z \; [\; (p(z) \wedge \forall \, a \; q \; (\, x, \, a, \, z) \;) \vee (\neg \, \forall \, b \; r \; (b) \;) \;]$$

Alle Negationszeichen nach innen ziehen:

$$F = \forall \, x \, \forall \, y \, \exists \, z \; [\; (\, p(z) \wedge \forall \, a \; q \; (\, x, \, a, \, z) \;) \vee (\exists \, b \neg \, r \; (b) \;) \;]$$

Pränexform (Alle Quantoren nach links außen):

$$F = \forall x \forall y \exists z \forall a \exists b [(p(z) \wedge q (x, a, z)) \vee (\neg r (b))]$$

Skolemform (Existenzquantoren beseitigen):

$$F = \forall x \forall y \forall a \exists b [(p(f (x, y) \wedge q (x, a, f (x, y)))) \vee (\neg r (b))]$$

$$F = \forall x \forall y \forall a [(p(f (x, y) \wedge q (x, a, f (x, y)))) \vee (\neg r (g(x, y, a)))]$$

Universelles Quantifizieren (Allquantoren weglassen):

$$F = (p (f (x, y) \wedge q (x, a, f (x, y))) \vee \neg r (g (x, y, a))$$

Universell quantifizierte Formeln lassen sich fast schon so programmieren wie sie sind. Die meisten Programmiersprachen, die in der Lage sind, prädikatenlogische Formeln zu verarbeiten, stellen jedoch noch weitere Restriktionen an die Formeln. In PROLOG beispielsweise können nur Regeln der Form $A \rightarrow B$ benutzt werden, bei denen B atomar und nicht negiert ist, wobei A sich zusammensetzen kann aus Negationen, Und- bzw. Oder-Termen. Da $A \rightarrow B$ äquivalent zu der Formel $\neg A \vee B$ ist, führt dies zu der Anforderung, dass Regeln möglichst eine Disjunktion von Konjunktionen von Atomen sind, wobei mindestens eines der Atome nicht-negiert sein muss. Dieses nicht-negierte Atom kann man dann als die „rechte Seite“ der daraus-folgt-Regel benutzen; diese ist dann nichtnegiert und atomar, genau so wie PROLOG das fordert (siehe unten).

Definition 3.12 (**Klausel**)
Unter einer *Klausel* verstehen wir eine Disjunktion von Literalen.

Zur Erinnerung: Literale waren negierte oder nichtnegierte Atome. Dies gilt auch für die Prädikatenlogik. Für unsere weiteren Betrachtungen ist ein Spezialfall der Klauseln wichtig:

Definition 3.13 (**Horn-Klausel**)
Eine *Horn-Klausel* ist eine Klausel, die höchstens ein nicht-negiertes Literal enthält. Wenn sie genau ein nicht-negiertes Literal enthält, nennt man sie auch *definite Hornklausel*. Besitzt sie mehr als ein nicht-negiertes Literal, jedoch mind. ein negiertes Literal, nennen wir sie *schwache* Hornklausel.

(Definite) Horn-Klauseln haben also die Form:

$$\neg x_1 \vee \neg x_2 \vee \ldots \vee \neg x_n \vee y$$

Der Vorteil davon ist, dass man aufgrund der de Morgan'schen Regeln dafür schreiben kann:

$$\neg(x_1 \wedge x_2 \wedge \ldots \wedge x_n) \vee y$$
$$\Leftrightarrow$$
$$(x_1 \wedge x_2 \wedge \ldots \wedge x_n) \rightarrow y$$

Horn-Klausen entsprechen also einer Subjunktion (bzw. Implikation, wenn die Voraussetzungen alle wahr sind, was ja im Allgemeinen angenommen wird). Horn-Klauseln kann man im Prinzip immer erzwingen, in dem man die Prädikate geeignet wählt.
Der Sinn dieser ganzen Betrachtungen ist –wie schon mehrfach angesprochen- die Entwicklung von Expertensystemen (vgl. Kapitel 4 und 5). Dazu muss ein Wissensingenieur im Prinzip als „Übersetzer" der „verbalen" Aussagen eines Nicht-Informatikers in die Prädikatenlogik fungieren. KI-Sprachen wie PROLOG können dann die prädikatenlogischen Regeln fast direkt so übernehmen, wie wir sie hier angegeben haben. PROLOG ist nämlich in der Lage, Horn-Klauseln zu verarbeiten. Und Horn-Klauseln lassen sich in der Praxis immer erzwingen, denn Literale können ja negierte oder nicht-negierte Atome (hier dann also Prädikate) sein. Es kann höchstes sein, dass wenn man z.B. nur negierte Atome hat, man eines der Prädikate umbenennen muss. So kann man z.B. aus

¬isst_gerne(karl, schwein)

einfach ein „positives" Prädikat der Form

nicht_isst_gerne(karl, schwwein)

machen. Dann muss man aber natürlich immer dann, wenn Karl doch gerne Schwein essen sollte, hier eine Negation hinzufügen, also z.B. *¬nicht_isst_gerne(karl,schwein).* Das ist zwar nicht unbedingt „ästhetisch", aber manchmal eben programmtechnisch erforderlich, um wenigstens *ein* nicht-negiertes Prädikat in einer Disjunktion sonst nur negierter Prädikate zu haben. Im Kapitel „PROLOG" wird hierzu weiteres gesagt.

Beispiel:
Als Beispiel einer Übersetzung aus der „Umgangssprache" in die Prädikatenlogik sei folgende verbale Regel betrachtet:

„Alle Römer, die Markus kennen, hassen Cäsar, oder denken, dass jeder, der irgendjemanden hasst, verrückt ist“.

Diese Aussagen wollen wir jetzt in eine prädikatenlogische Form bringen, genauer: in eine schwache Hornklausel.
Zunächst muss die Struktur, insbesondere das Universum für die benutzten Individuen, festgelegt werden. Wir definieren, dass das Universum aus der Menge aller Menschen besteht. Nachfolgende Individuen (*X,Y,Z* usw.) sind also Elemente des so definierten Universums. Umsetzten der verbalen Aussage in eine Formel führt also zu:

$$\forall X\,[\,(\,\text{römer}\,(X) \wedge \text{kennen}\,(X, \text{markus}\,)\,) \rightarrow (\,\text{hassen}\,(X, \text{cäsar}\,) \vee (\forall Y\,(\exists Z\ \text{hassen}\,(Y,Z\,) \rightarrow \text{verrückthalten}\,(X,Y\,)\,)\,)\,)\,]$$

Wie man hier sieht, sind bestimmte Prädikate einstellig, andere mehrstellig. Darüber sollte genau nachgedacht werden. Römer zu sein ist ein einstelliges Prädikat. Hassen dagegen zweistellig, denn es muss ein Individuum geben, welches hasst, und ein anderes, auf den sich der Hass bezieht. Das zweistellige Prädikat *verrückthalten* fasst zusammen, dass ein Individuum denkt, dass ein anderes verrückt ist. Auch die Reihenfolge der Individuen innerhalb eines Prädikats ist wichtig. Sie kann zwar zunächst beliebig festgesetzt werden, doch muss dann immer beibehalten und konsistent interpretiert werden. Interpretiert man z.B. das Prädikat *hassen(X,Y)* so, dass *X* der Hasser und *Y* der Gehasste ist, so muss diese Reihenfolge immer durchgehalten werden.
Als nächster Schritt ist es sinnvoll, „$\rightarrow$“ und „$\leftrightarrow$“ zu eliminieren mittels (A $\rightarrow$ B) äquivalent zu ($\neg$ A $\vee$ B):

$$\forall X\,[\neg\,(\,\text{römer}\,(X) \wedge \text{kennen}\,(X, \text{markus}\,)\,) \vee (\,\text{hassen}\,(X, \text{cäsar}\,) \vee (\forall Y \neg(\exists Z\ \text{hassen}\,(Y,Z) \vee \text{verrückthalten}\,(X,Y\,)\,)\,)\,)\,]$$

„$\neg$“ nach innen ziehen:

$$\forall X\,[\,(\neg\,\text{römer}\,(X) \vee \neg\,\text{kennen}\,(X, \text{markus}\,)\,) \vee (\,\text{hassen}\,(X, \text{cäsar}\,) \vee (\forall Y\,(\forall Z\,\neg\,\text{hassen}(Y,Z) \vee \text{verrückthalten}\,(X, Y\,)\,)\,)\,)\,]$$

Pränex-Form:

$$\forall X\ \forall Y\ \forall Z\,[\,\neg\,\text{römer}\,(X) \vee \neg\,\text{kennen}\,(X, \text{markus}\,) \vee \text{hassen}\,(X, \text{cäsar}\,) \vee \neg\,\text{hassen}\,(Y, Z\,) \vee \text{verrückthalten}\,(X, Y\,)\,]$$

Skolem-Form: *dito.*, da keine Existenzquantoren vorhanden sind

Universelles Quantifizieren:

$\neg$ *römer (X)* $\vee \neg$ *kennen (X, markus) $\vee$ hassen (X, cäsar) $\vee \neg$ hassen (Y, Z) $\vee$ verrückthalten (X, Y)*

Die universell quantifizierte Formel ist eine (schwache) HORN-Klausel, da mind. ein nicht-negiertes sowie mind. ein negiertes Literal als Disjunktionsterm vorhanden ist. Daraus kann also eine A→B-Regel gemacht werden. Dazu wählt man einen der nicht-negierten Disjunktionsterme aus, z.B. *verrückthalten(X,Y)*, und konstruiert die Regel:

$\neg$ *römer (X)* $\vee \neg$ *kennen (X, markus) $\vee$ hassen (X, cäsar) $\vee \neg$ hassen (Y, Z) $\vee$ verrückthalten (X, Y)*

$\Leftrightarrow$

$\neg$*(römer (X)* $\wedge$ *kennen (X, markus) $\wedge$ $\neg$ hassen (X, cäsar) $\wedge$ hassen (Y, Z)) $\vee$ verrückthalten (X, Y)*

$\Leftrightarrow$

(römer (X) $\wedge$ kennen (X, markus) $\wedge$ $\neg$ hassen (X, cäsar) $\wedge$ hassen (Y, Z)) → verrückthalten (X, Y)

Definition 3.14 (**Fakten, Regeln, Wissensbasis**)
Enthält ein Prädikat nur Individuenkonstanten, so nennen wir dieses einen *Fakt*. Eine (schwache) Horn-Klausel wird auch eine *Regel* genannt. Fakten und Regeln bilden zusammen die sog. *Wissensbasis*.

Das Prädikat *isst_gerne(karl, schwein)* ist also ein Fakt. Dagegen sind Regeln immer von der Form A→B, wobei B ein (nicht-negiertes) Atom darstellt und A und B Prädikate sind, die Individuenvariablen enthalten können. Es stellt also

(römer (X) $\wedge$ kennen (X, markus) $\wedge$ $\neg$ hassen (X, cäsar) $\wedge$ hassen (Y, Z)) → verrückthalten (X, Y)

eine typische Regel dar.

Expertensysteme sind in der Lage, selbstständig aus einer Wissensbasis Schlussfolgerungen zu ziehen. Um zu verstehen wie das geht, ist ein ganz kurzer Ausflug in die Theorie des automatisierten Beweisens notwendig. Darunter versteht man die Möglichkeit, dass ein Computer ein allgemeines Prinzip kennt, mit dem

mittels einer Menge von gegebenen Fakten und Regeln (also der Wissensbasis) festgestellt werden kann, ob eine eingegebene Aussage wahr oder falsch ist. Die Fakten und Regeln der Wissensbasis stellen dabei per Definition wahre Aussagen dar und werden daher auch als Axiomensystem bezeichnet. Hierzu definieren wir:

Definition 3.15 (**Resolvente**)
Es seien

$$h := h_1 \vee h_2 \vee ... \vee h_N \text{ und}$$

$$k := k_1 \vee k_2 \vee ... \vee k_M$$

zwei universell quantifizierte, unvollständige skolemisierte disjunktive Normalformeln (also Klauseln). Des Weiteren sei o.B.d.A. $h_1 = \neg k_1$. Die Formel

$$Res(h,k) := h_2 \vee ... \vee h_N \vee k_2 \vee ... \vee k_M$$

heißt dann die *Resolvente* der beiden Klauseln *h* und *k*.

Eine Resolvente ist übrigens eine Implikation aus den beteiligten Klauseln, d.h.

$$h \wedge k \Rightarrow Res(h,k)$$

Hierzu ein kleines Beispiel:
Seien $h = A \vee B \vee C \vee D$ und
$k = E \vee A \vee \neg C \vee F$
Nach umsortieren:
$h = C \vee A \vee B \vee D$ und
$k = \neg C \vee E \vee A \vee F$
Die Resolvente lautet dann (da $A \vee A = A$):
$Res(h,k) = A \vee B \vee D \vee E \vee F$

Resolventen sind damit das Ergebnis der “Ver-oderung” zweier Klauseln, wobei Disjunktionsterme gleicher Literale, die in der einen Klausel negiert und in der anderen nicht-negiert vorkommen müssen, weggelassen wurden. Mit Hilfe solcher Resolventen lässt sich nun ein wichtiger Satz formulieren:

Satz 3.16 (**Resolutionstheorem**)
Eine Aussage ist implizierbar (d.h. eine logische Schlussfolgerung) aus einem gegebenen Axiomensystem, wenn aus der Negation der Aussage zusammen mit Formeln des Axiomensystems mittels Bildung von Resolventen eine leere Klausel erzeugt werden kann.

Solch eine leere Klausel wird aus der Negation der Ursprungsbehauptung hergeleitet (impliziert) und entspricht dem binären Wert „0“, also einer „falschen“ Aussage. Nach dem Prinzip des Widerspruchsbeweises muss damit die (nichtnegierte) ursprüngliche Aussage wahr sein.
Mit dem Resolutionstheorem hat man also eine allgemeine Methode zur Verifikation von Aussagen. Dies sei an einem Beispiel vorgeführt.

Beispiel:

Wir nehmen an, dass ein Expertensystem für einen Historiker entwickelt werden soll. Ein Wissensingenieur hat den Historiker befragt und folgendes Axiomensystem aufgestellt:

1. Markus war ein Mensch
2. Markus war ein Pompeianer
3. Alle Pompeianer waren Römer
4. Cäsar war ein Herrscher
5. Alle Römer waren loyal zu Cäsar oder hassten ihn
6. Jeder ist loyal zu einem anderen
7. Wenn Menschen versuchten, Herrscher zu ermorden, so waren sie (die Menschen) nicht loyal zu ihnen (den Herrschern)
8. Markus versuchte, Cäsar zu ermorden

Nun geht der Wissensingenieur daran, dieses verbale Axiomensystem in eine Wissensbasis umzusetzen. Dazu muss zunächst eine prädikatenlogische Formulierung der Axiome erfolgen. Das Universum für die benutzten Individuen sei die Menge aller Lebewesen auf der Erde.

1. Markus war ein Mensch
   mensch(markus)

2. Markus war ein Pompeinaner
   pompeianer(markus)

3. Alle Pompeianer waren Römer
   $\forall X(pompeianer(X) \rightarrow römer(X))$
   universell quantifiziert:
   $\neg pompeianer(X) \vee römer(X))$

4. Cäsar war ein Herrscher
   herrscher(cäsar)

5. Alle Römer waren loyal zu Cäsar oder hassten ihn
$\forall X(römer(X) \rightarrow (loyalzu(X,cäsar) \vee hassen(X,cäsar)))$
universell quantifiziert:
$\neg römer(X) \vee loyalzu(X,cäsar) \vee hassen(X,cäsar)$

6. Jeder ist loyal zu einem anderen
$\forall X \exists Y (loyalzu(X,Y))$
universell quantifiziert:
$loyalzu(X,f(X))$

7. Wenn Menschen versuchten, Herrscher zu ermorden, so waren sie (die Menschen) nicht loyal zu ihnen (den Herrschern)
$\forall X \forall Y ((mensch(X) \wedge herrscher(Y) \wedge ermorden(X,Y)) \rightarrow \neg loyalzu(X,Y))$
universell quantifiziert:
$\neg mensch(X) \vee \neg herrscher(Y) \vee \neg ermorden(X,Y) \vee \neg loyalzu(X,Y)$

8. Markus versuchte, Cäsar zu ermorden
$ermorden(markus, cäsar)$

Axiom Nummer 6 beinhaltet eine Funktion von *X*, die spätestens im Programm, welches das Expertensystem implementiert, genau definiert sein muss. Denkbar ist z.B. eine Tabelle mit zwei Spalten. In der ersten Spalte stehen alle vorkommenden Individuen des Universums und in der zweiten Spalte stehen die jedem der Individuen der ersten Spalten zugeordneten Individuen, denen sie gegenüber loyal sind. Auf jeden Fall liefert die Funktion *f(X)* für ein konkretes Individuum *X* das oder die Individuen aus der Tabelle zurück, denen gegenüber *X* loyal ist.
Angenommen, unser Historiker möchte nun wissen, ob Markus Cäsar gehasst hat. Es ist also das Ziel, die Aussage „Markus hasst Cäsar“ aus dem vorhandenen Axiomensystem heraus zu beweisen. In prädikatenlogischer Form heißt das Beweisziel

$$hassen(markus,cäsar).$$

Axiom 8 stellt zwar fest, dass Markus versuchte, Cäsar zu ermorden, doch das muss nicht notwendiger Weise heißen, dass Markus Cäsar auch hasste. Es gibt z.B. Exekutionskommandos, die Menschen ermorden, um einen Befehl auszuführen, ohne die Menschen, die sie töten, zu hassen. Auch Mord aus Liebe soll schon vorgekommen sein.
Um also den Beweisversuch zu starten, wird zunächst das Gegenteil der zu beweisenden Aussage angenommen und durch sukzessive Resolventenbildung

versucht, gemäß des Resolutionstheorems zusammen mit dem Axiomen 1-8 eine leere Klausel zu erzeugen.

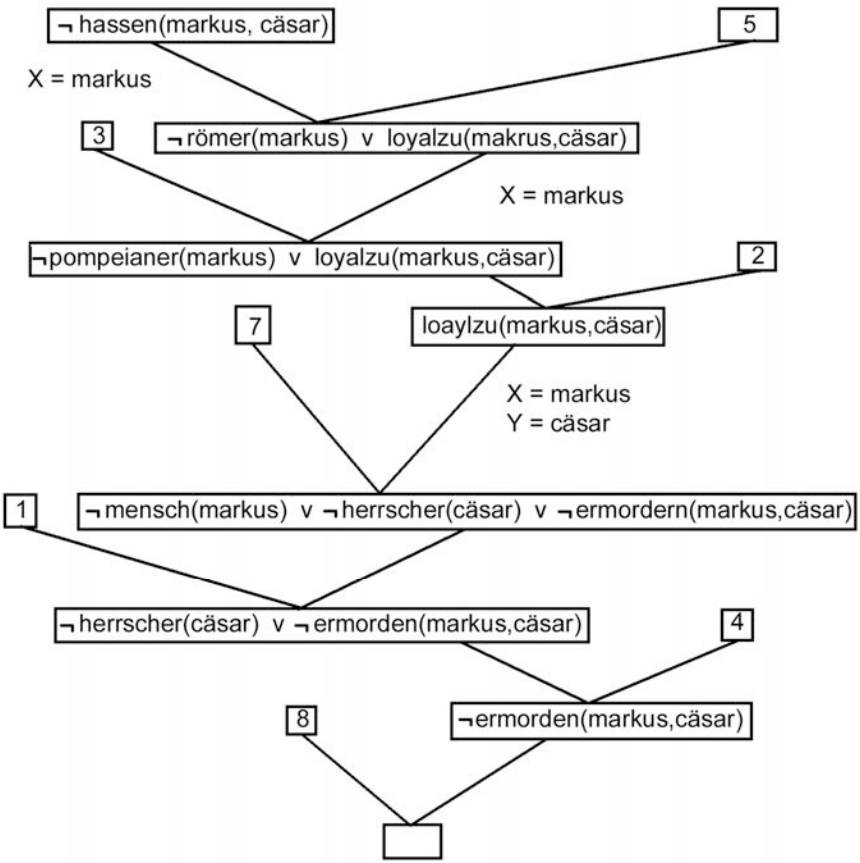


Abb. 3.1 Resolutionsmethode

In Abb.3.1 sieht man die Vorgehensweise. Der Resolutionsalgorithmus sucht zunächst im Axiomensystem eine Klausel, in der die Negation des aktuellen Terms vorkommt. In unserem Fall findet der Algorithmus das Axiom 5, und es wird damit die Resolvente gebildet. Dabei wird die zuvor freie Variable *X* an die Individuenkonstante *markus* gebunden. Nun sucht der Algorithmus erneut die Axiome durch und selektiert dasjenige, in dem wieder das Gegenteil einer der

Disjunktionsterme der aktuellen Klausel vorkommt (die Reihenfolge ist dabei prinzipiell egal, das Ergebnis ist letztendlich das selbe). Jetzt passt Axiom 3 und die dort vorkommende Variable *X* wird während der Resolventenbildung ebenfalls mit *markus* belegt. Dieses Spiel wird solange wiederholt, bis schließlich eine leere Klausel erzeugt wird. Damit ist aufgrund des Resolutionstheorems die ursprüngliche Aussage, also *hassen(markus,cäsar)*, wahr.

Nach dieser kurzen Einführung in die Prädikatenlogik wollen wir uns einer Anwendung derselben zuwenden, nämlich der Erstellung von Expertensystemen. Letztere werden in der Regel mit Hilfe einer Programmiersprache der sogenannten 5. Generation programmiert. PROLOG stellt so eine Programmiersprache dar und hat den Vorteil, dass prädikatenlogische Horn-Klauseln praktisch direkt im Source-Code übernommen werden können. In PROLOG wird im nächsten Kapitel eingeführt.

Übungen zum Selbsttest:

1. Bilden Sie für nachfolgende Formeln jeweils (i) die bereinigte Pränexform und (ii) die Skolemform und (iii) eine universell quantifizierte unvollständige minimale konjunktive Normalform:

   a) $F = \neg\exists r(A(r,t)\vee\forall sB(r,f(s)))\vee\forall sA(g(r,s),t))$

   b) $F = \forall x\forall y(\neg(P(x)\vee\forall zQ(y,z))\wedge\exists k\neg P(k)) \wedge \exists x\forall y\neg(\exists k\neg P(k))$

   c) $F= \exists x\exists y\forall z(P(x,y,z)\vee\neg(\exists a\forall bP(a,b,x)))$

2. Es sei folgendes Axiomensystem (=Wissensbasis) gegeben, wobei das Universum aus der Menge aller PROGRAMMNAMEN besteht (Einträge in der FAT) und zwischen „Programmen“ (eigentlich „Systemprogramme“), „Software“ (die man selbst aufrufen kann) und „Anwendungsprogrammen“ (für „normale“ Benutzer) unterscheiden wird:

   (1) "LAN.EXE" ist ein Programm
   (2) "LAN.EXE" ist eine Software
   (3) Jede Software ist virusinfiziert
   (4) "WORD.EXE" ist ein Anwendungsprogramm
   (5) Alle virusinfizierten Programme ließen "WORD.EXE" in Ruhe oder greifen es an
   (6) Es gibt Programmnamen, die von allen anderen in ruhe gelassen werden
   (7) Wenn Programme versuchen, Anwenderprogramme zu löschen, dann heißt das, das diese Programme die Anwenderprogramme nicht in Ruhe lassen
   (8) "LAN.EXE" versucht, "WORD.EXE" zu löschen

   Zeigen Sie mit Hilfe des Resolutionstheorems unter Bildung von Resolventen die Behauptung:

   (9) "LAN.EXE" greift "WORD.EXE" an.

3. Es sei folgendes Axiomensystem gegeben:

   1. Hugo ist ein bekannter Schauspieler.
   2. Karl ist ein Mensch und er ist auch schön.

3. Leider sind alle schönen Menschen auch eitel.
4. Wenn ein Mensch eitel ist, so will er nicht so sein wie Hugo oder er will ihm alles nachmachen.
5. Menschen wollen so sein wie Schauspieler, falls sie diese mögen.
6. Karl mag Hugo.

Beweisen Sie mit Hilfe des Resolutionstheorems unter Bildung geeigneter Resolventen die Aussage:

7. Karl macht Hugo alles nach.

Hinweis: Es empfiehlt sich grundsätzlich, Axiome, die in konjunktiver Form sind, in einzelne "Teilaxiome" zu zerlegen.

4. Einführung in PROLOG

PROLOG steht für „PROgrammation en LOGique“ und wurde 1972 von Alain Colmerauer an der Universität von Marseille realisiert. Prolog unterscheidet sich grundsätzlich von Programmiersprachen wie Visual Basic, Java oder C++. Letztgenannte Sprachen zählen zu den sog. *imperativen Sprachen*, d.h. sie sind befehlsorientiert. Da gibt es Befehle wie *if...then...else* oder *do...while* oder auch einfach Wertzuweisungen wie *x:=x+1*.
Das alles gibt es in Prolog nicht. Prolog zählt zu den Sprachen der sog. 5. Generation und wird auch als „KI-Sprache“ bezeichnet. Zu dieser Sprachengruppe zählen auch LISP oder LOGO. Man nennt diese Sprachen *funktionale Sprachen*, da sie auf der Basis von Funktionen und Operatoren (wie die logischen Operatoren die wir kennen gelernt haben) basieren. Ein Spezialfall der funktionalen Sprachen sind die *deskriptiven Sprachen*. Sie werden so genannt, weil man hier nicht wie bei imperativen Sprachen eine Problemlösung implementiert (wie das Berechnen eines Kreises), sondern das zu lösende Problem nur *beschreibt* und die Lösung des Problems dann dem System überlässt.
Wie viele andere Programmiersprachen hat auch Prolog sein spezielles Anwendungsgebiet. So wird man selten Kreise mit Prolog berechen (obwohl das geht) oder ein Betriebssystem programmieren. Prolog ist am besten geeignet für Probleme, die sich prädikatenlogisch formulieren lassen, d.h. die durch (Horn-)Klauseln ausgedrückt werden können (vgl. Kapitel 3). Es gibt in Prolog keine Iterationen, wohl aber Rekursionen, die sehr effektiv eingesetzt werden können, wie wir noch sehen werden. Auch gibt es in Prolog eine Datenstruktur, die *Liste* heißt, und ein sehr wichtiges Programmierwerkzeug darstellt. Der eigentliche „Befehlsumfang“ in Prolog ist recht gering, doch die Kombination der wenigen Sprachelemente ist äußerst mächtig. Das wichtigste Anwendungsgebiet von Prolog sind Expertensysteme (vgl. Kapitel 5).
Es würde den Rahmen dieses Buches sprengen, hier einen vollständigen Prolog-Kurs ersetzen zu wollen, dafür gibt es eigenständige Lehrbücher. Wir wollen vielmehr hier die Grundprinzipien der Programmentwicklung mit Prolog veranschaulichen, meistens an einfachen Beispielen und innerhalb des dialektfreien Prolog-Grundstandards. Die hier vorgestellten Prolog-Programme müssten unter jedem Prolog-Interpreter laufen, von denen im Internet einige kostenlos verfügbar sind[1]. Es werden die Prolog-Programme dabei am besten mit einem beliebigen Text-Editor verfasst und als Text-Dateien mit der Endung **.pl* oder **.pro* abgelegt. Hat man einen Prolog-Interpreter einmal installiert und ruft ihn auf, so erscheint der Prompt der From:

[1] Siehe z.B. http://www.swi-prolog.org/

```
?-
```

Um eine Textdatei mit dem Prolog-Quellcode zu laden, gibt man dahinter ein:

```
?- consult(”c:\prolog\test.pro”).
```

Wichtig ist der Punkt hinter der Befehlszeile, dieser schließt in Prolog jede Eingabe ab. Durch obige Eingabe wird also die Textdatei `test.pro` vom Prolog-Interpreter „konsultiert“, d.h. in den Speicher geladen. Aber Achtung: Wiederholt man diese Eingabe, steht das Programm zweimal im Speicher. Normalerweise will man das natürlich nicht (weil man z.B. etwas am Code verändert hat). Dafür schreibt man dann besser

```
?- reconsult(”c:\prolog\test.pro”).
```

Jetzt wird die alte Version des Programm entfernt und durch die neue ersetzt.
Mit der Anweisung

```
?-listing.
```

kann man sich davon überzeugen, dass der Prolog-Code auch geladen wurde.
Natürlich haben wir bisher noch gar kein Prolog-Programm geschrieben, aber wir wissen immerhin schon, wie es in den Prolog-Interpreter „hineinkommt“. Nun zu eineigen Grundlagen und Beispielen für das Erstellen von Prolog-Programmen.
Prolog kennt als Sprachelement eigentlich nur eine einzige Sache: Prädikate (im Sinne von Kapitel 3). Es gilt dabei die Konvention, dass Individuenvariablen groß und Individuenkonstanten klein geschrieben werden. Man kann aber auch Individuenkonstanten schrieben wie man will, wenn man sie in Hochkommata einschließt. Variable müssen aber immer mit einem Großbuchstaben beginnen. Individuenvariable sind Variable im mathematischen Sinn, deswegen machen auch Wertzuweisungen wie etwa $x=x+1$ in Prolog keinen Sinn. Der Interpreter würde dies als eine Gleichung in der Variablen x auffassen und versuchen, diese Gleichung zu lösen (was natürlich unmöglich ist).
Prädikate in Prolog können auch kombiniert werden, aber nur in Form von Horn-Klauseln. Diese werden als Implikationen dargestellt und in Prolog *Regeln* genannt. Normalerweise sind in Regeln Individuenvariable involviert. Neben solchen Regeln gibt es auch *Fakten*. Dabei handelt es sich um einzelne Prädikate, die nur Individuenkonstanten enthalten. Jedes Prolog-Programm ist im Prinzip immer nur aus Regeln und Fakten aufgebaut.
Eigentlich könnte man hier schon das Kapitel über Prolog beenden, denn es gibt keine anderen Sprachkonstrukte in Prolog. Doch da der Mensch praktisch nur an

Beispielen wirklich verstehen lernt, seien dieselbe zum besseren Verständnis nun doch noch aufgeführt. Außerdem wollen wir uns auch mit der internen Verarbeitungsweise von Prolog beschäftigen sowie mit der besonderen Datenstruktur „Liste“, damit wir in die Lage versetzt werden, möglichst effiziente Prolog-Programme zu implementieren.
Wir beginnen mit einem einfachen Beispiel, nämlich der Repräsentation von Mitgliedern eines Sportvereins, zunächst in Form von Fakten:

```
/* Fakten */
mitglied(meier, ernst, hockey).
mitglied(schubert, klaus, fussball).
mitglied(meier, egon, fussball).
```

Hier sehen wir auch gleich, wie Kommentare in ein Prolog-Programm eingefügt werden (durch `/*...kommentare...*/`). Angenommen, dieses Prolog-Programm steht in der Textdatei `sport.pro`, so geben wir ein:

```
?- consult(”sport.pro”).
```

Manche Prolog-Interpreter mögen die „Gänsefüßchen“ nicht, sondern nur das einfache Hochkoma, das muss der jeweiligen Dokumentation entnommen werden.
Wir sehen in dem kleinen Programm, dass hier ein 3-stelliges Prädikat mit Namen *mitglied(...)* definiert wurde (diese Definition geschieht in Prolog implizit, d.h. die Struktur wird einfach durch das erste Fakt festgelegt). Die 3 Individuen (Name, Vorname, Sportart) können in ihrer Anzahl danach nicht mehr geändert werden.
Mit unserem kleinen Programm können nun schon einige Anfragen an das System gemacht werden:

```
?- mitglied(meier, Vorname, Sportart).
```

Ergebnis:

```
Vorname = ernst    Sportart = hockey
Vorname = klaus    Sportart = fussball
Yes
```

Die Eingabe hier wird von Prolog als eine zu beweisende Aussage verstanden. Damit ist gemeint, dass die Eingabe nach dem Prompt als eine Behauptung angesehen wird, die dann mit Hilfe der Resolutionsmethode intern zusammen mit den „Axiomen“ des Programms (das sind unsere drei Fakten aus der Datei

`sport.pro`) benutzt wird um eine leere Klausel (vgl. Kapitel 3) zu erzeugen. Gelingt dies, ist nach dem Resolutionstheorem die Aussage wahr und es erscheint daher am Schluss auch ein `Yes`. In den Beispielen der Resolventenbildung hatten wir Kapitel 3 gesehen, dass dabei ggf. Individuenvariable an Individuenkonstanten gebunden werden. Diese Variablenbindungen liefert jetzt die Prolog-interne „Beweismaschine“ (die auch als *Inferenzmaschine* bezeichnet wird) sozusagen kostenlos mit. Damit sieht man gleich, welche Individuenkonstanten die eingegebene Behauptung „wahr“ machen. Da die Eingabe-Anfrage eine Individuenkonstante (`meier`) und zwei Individuen-Variablen (`Vorname`, `Sportart`) beinhaltet, werden alle diese Anfrage bewahrheitenden Individuenkonstanten zurückgeliefert.
Gibt man dagegen eine Anfrage der Form

```
?- mitglied(meier, ernst, hockey).
```

ein, so erscheint als Ausgabe nur ein schlichtes

```
Yes
```

Da hier keine Individuenvariablen eingegeben wurden, sondern nur Individuenkonstanten, ist auch nichts zu binden und zurückzuliefern (außer der Antwort `Yes` oder `No`). Die Beweisführung endet in diesem Fall auch schon bei der ersten Erfüllung, d.h. es wird nur nach der ersten Verifizierung gesucht. Sind aber Individuenvariablen mit in die Anfrage gepackt, so werden *alle* Fakten durchsucht und erst aufgehört, wenn das letzte Fakt erreicht ist. Dadurch wird erreicht, dass alle möglichen Variablenbelegungen, welche die Anfrage „wahr machen“, ausgegeben werden können.
Es gibt in Prolog auch etwas, das *anonyme Variable* genannt wird. Dabei handelt es sich um eine Individuenvariable, deren Wert „egal“ ist und daher auch nicht angezeigt oder weiter verarbeitet werden muss. Jede anonyme Individuenvariable wir mit _ bezeichnet. Beispiel:

```
?- mitglied(meier,V,_).
```

Das Ergebnis ist hier:

```
V = klaus
Yes
```

Wir wollen jetzt unser kleines Programm etwas interessanter gestalten und erweitern es um einige Fakten, wobei wir die Fakten der Mitglieder um einige Individuen erweitert haben:

```
/* Fakten */
mitglied(schubert,klaus,m,1962,fussball).
mitglied(berger,ursel,w,1968,badminton).
mitglied(meier,egon,m,1955,fussball).
mitglied(mueller,iris,w,1963,badminton).
mitglied(dolittle,elisa,w,1968,badminton).
beitrag(hockey,30.50).
beitrag(fussball,42.30).
beitrag(badminton,10).
```

Jetzt kann man z.B. versuchen, alle weiblichen Mitglieder herauszubekommen, nebst Geburtsjahr und Sportart:

```
?- mitglied(N, V, w, G, S).
```

liefert:

```
N = berger,
V = ursel,
G = 1968,
S = badminton ;

N = mueller,
V = iris,
G = 1963,
S = badminton ;

N = dolittle,
V = elisa,
G = 1968,
S = badminton ;

No.
```

In manchen Prolog-Interpretern (wie hier, da wurde SWI-Prolog benutzt) wird nach der Eingabe zunächst nur die „erste“ Lösung angezeigt. Durch Eingabe des Semikolons (`;`) wird die nächste Lösung gesucht. Da es nur 3 Lösungen gibt und danach wieder ein Semikolon eingegeben wurde, liefert das System am Schluss die Antwort `no` (da keine weitere Lösung gefunden wurde).
Wie schon erwähnt, kann man in Prolog auch Regeln verarbeiten. Eine Regel muss eine Hornklausel darstellen, wobei mind. eines der beteiligten Prädikate nicht negiert sein darf. Dieses nicht-negierte Prädikat muss das „Geschlussfolgerte“ der Implikation sein (gibt es mehrere nicht-negierte Prädikate, kann man sich eines aussuchen).

Regeln haben also in Prolog immer die Form:

$$a \rightarrow b$$

wobei a aus negierten und/oder nicht-negierten Prädikaten durch Konjunktionen (Prolog erlaubt auch Disjunktionen) zusammen gesetzt sein kann, und b immer ein einziges, nicht-negiertes Prädikat sein muss. Die Prolog-Schreibweise ist dafür allerdings:

```
b :- a
```

(lies: „b gilt, wenn a gilt“)

Die Konjunktion in Prolog wird durch ein Komma (`,`) die Disjunktion durch ein Semikolon (`;`) und die Negation durch `not(...)` dargestellt.
Wir erweitern nun unser Prolog-Programm um zwei Regeln:

```
/* Fakten */
mitglied(schubert,klaus,m,1962,fussball).
mitglied(berger,ursel,w,1968,badminton).
mitglied(meier,egon,m,1955,fussball).
mitglied(mueller,iris,w,1963,badminton).
mitglied(dolittle,elisa,w,1968,badminton).
beitrag(hockey,30.50).
beitrag(fussball,42.30).
beitrag(badminton,10).

/* Regeln */
spielpaarung(Sportart,Name1,Vorn1,Name2,Vorn2) :-
       mitglied(Name1,Vorn1,_,_,Sportart),
       mitglied(Name2,Vorn2,_,_,Sportart),
       not (Name1 = Name2).
alter(Name,Vorname) :-
       mitglied(Name,Vorname,_,Geb_jahr,_) , date(Jahr,_,_),
       Erg is Jahr-Geb_jahr , write(Erg).
```

Die Regel `spielpaarung` soll Kandidaten für ein Turnier ermitteln (wer spielt gegen wen in der gleichen Sportart) und die Regel `alter` liefert das Alter eines Mitglieds zurück. Dabei fällt folgendes auf: In der Regel `spielpaarung` wird am Schluss ein Gleichheitszeichen benutzt, während in der Regel `alter` das Wörtchen `is` benutzt wird. Dieser feine Unterschied resultiert aus der Tatsache, dass ein Gleichheitszeichen in Prolog immer mit einer Gleichung assoziiert ist, die also wahr oder falsch sein kann. So wird in der Regel `spielpaarung` er-

zwungen, dass die Gleichung `name1=name2` falsch sein muss, d.h. es soll niemand gegen sich selbst spielen. In der Regel `alter` wird jedoch eine Berechnung durchgeführt, d.h. die Differenz zwischen dem aktuellen Jahr und dem Geburtsjahr wird ermittelt und dann der Variablen `Erg` zugewiesen. Dennoch handelt es sich um keine Wertzuweisung im klassischen Sinn, denn so was wie `erg is erg + 1` geht in Prolog nach wie vor nicht. Aber als „Eselsbrücke" kann man sich hier eine Wertzuweisung (mit der genannten Einschränkung) denken, um vom Gleichheitszeichen unterscheiden zu können.

Dann haben wir in den Regeln die beiden Prädikate `date(...)` und `write(...)`. Während bei imperativen Programmiersprachen so was eine „Funktion" genannt wird, ist dem hier nicht so! In Visual Basic beispielsweise liefert eine Funktion date(...) das Datum als Funktionswert zurück. Aber nicht so bei Prolog! Hier ist `date(J, M, T)` ein dreistelliges Prädikat, das –wie alle Prädikate- nur den Wert *wahr* oder *falsch* zurückgibt. Solche „Standardprädikate" sind in Prolog fest eingebaut und werden immer dann wahr, wenn sie „ausgeführt" sind. Für `date(J, M, T)`heißt das, dass wenn die Individuenvariablen J, M und T mit dem aktuellen Jahr, dem Monat und dem Tag belegt werden konnten, das Prädikat den Wert „wahr" zurückliefert. In unserer Regel wurden Monat und Tag unterdrückt (durch die anonyme Variable) und nur das aktuelle Kalenderjahr an die Individuenvariable `Jahr` abgegeben. Analog verhält es sich mit dem Standardprädikat `write(Erg)` aus der letzten Regel. Es wird wahr, wenn der Wert der Individuenvariablen `Erg` auf den Bildschirm ausgegeben werden konnte.

Bei Standard-Prädikaten ist immer etwas Vorsicht geboten, es kann nämlich sein, dass verschiedene Interpreter hier bei der Bezeichnung eines solchen Prädikats von einander abweichen. Kommt also eine Fehler der Art „Prädikat nicht definiert" o.ä., dann sollte in der Dokumentation des Interpreters nachgelesen werden, wie das gewünschte Standardprädikat genau heißt.

Betrachten wir jetzt das Prädikat `spielpaarung(...)` etwas genauer.
Gibt man nach dem Laden des Programms ein

```
?- spielpaarung(badminton, N1,V1,N2,V2).
```

so kommt als Ergebnis folgendes:

```
N1 = berger,
V1 = ursel,
N2 = mueller,
V2 = iris ;

N1 = berger,
V1 = ursel,
N2 = dolittle,
V2 = elisa ;

N1 = mueller,
V1 = iris,
N2 = berger,
V2 = ursel ;

N1 = mueller,
V1 = iris,
N2 = dolittle,
V2 = elisa ;

N1 = dolittle,
V1 = elisa,
N2 = berger,
V2 = ursel ;

N1 = dolittle,
V1 = elisa,
N2 = mueller,
V2 = iris ;

no
```

Wir haben mittels `not(Name1=Name2)`zwar verhindert, dass jemand gegen sich selbst spielt, doch nicht, dass „spiegelbildliche“ Turniere stattfinden wie im 1. und 3. Ergebnis. Wenn man Hin- und Rückspiele haben will, wäre das in Ordnung. Ansonsten müsste man z.B. `not(Name1=Name2)` ersetzen durch `Name1<Name2` (falls der Prolog-Interpreter lexografische String-Vergleiche beherrscht, ansonsten muss man ein Prädikat zum Beispiel der Form `kleiner(Name1, Name2)` selbst programmieren mittels einer geeigneten Regel). Dann kommt jedes Spiel nur einmal, es gibt keine „Rückspiele“ mehr.
Wir bleiben aber bei unserer Originalversion und wollen anhand dieses Beispiels etwas demonstrieren, was in Prolog *Backtracking* genannt. Backtracking geht nach dem Versuch-und-Irrtum-Prinzip (*trial and error*) vor, d.h. es wird versucht, eine erreichte Teillösung schrittweise zu einer Gesamtlösung auszubauen.

Wenn absehbar ist, dass eine Teillösung nicht zu einer endgültigen Lösung führen kann, werden der letzte Schritt bzw. die letzten Schritte zurückgenommen und stattdessen alternative Wege probiert. Auf diese Weise ist sichergestellt, dass alle in Frage kommenden Lösungswege ausprobiert werden können. Mit Backtracking-Algorithmen wird eine vorhandene Lösung entweder gefunden (unter Umständen nach sehr langer Laufzeit), oder es kann definitiv ausgesagt werden, dass keine Lösung existiert. Backtracking wird meistens am einfachsten rekursiv implementiert und ist ein prototypischer Anwendungsfall von Rekursionen. Das Prädikat `spielpaarung` ist offenbar erfüllt, wenn alle drei „Subgoals“

```
Z1 = mitglied(Name1,Vorn1,_,_,badminton)

Z2 = mitglied(Name2,Vorn2,_,_,badminton)

Z3 = not (Name1 = Name2)
```

gleichzeitig im Sinne einer Konjunktion erfüllt ind. Dazu machen wir eine Tabelle und erläutern daran das Backtracking. Ein + hinter dem Subgoal heißt: Die Regel ist erfüllt, ein – bedeutet: nicht erfüllt. Wie man in nachfolgender Tabelle sehen kann, wird zunächst damit begonnen, das erste Mitglied zu untersuchen. Dazu werden die Individuenvariablen `Name1` und `Vorname1` an die Individuenkonstanten `meier` und `ernst` gebunden. Da jedoch die Sportart nicht `badminton`, sondern `hockey` ist, scheitert Z1 sofort. Gleiches passiert mit dem nächsten Mitglied. Das dritte Fakt scheitert jetzt nicht, da die Sportart jetzt `badminton` ist. Dieses Sub-Ziel wird erfüllt und „festgehalten“. Jetzt wird von vorne losgelegt mit Ziel Z2. Auch hier scheitern wieder die ersten beiden Fakten, und erst wenn erneut beim dritten Fakt angelangt wurde, ist auch das Sub-Ziel Z2 erfüllt. Dieses wird (zusammen mit dem „alten“ Sub-Ziel Z1) festgehalten und es wird das dritte Subgoal überprüft (Z3). Hierzu müssen keine Fakten abgeklappert werden, da die Individuenvariablen `Name1` und `Name2` schon bekannt sind und nur noch `not(Name1=Name2)` überprüft werden muss. Da hier aber das gleiche Mitglied sowohl für Z1 als auch für Z2 vorliegt und damit die Namen gleich sind, scheitert das dritte Subgoal. Und jetzt das entscheidende Merkmal des Backtracking: Das letzte Subgoal vor dem Scheitern (also Z2) wird aufgegeben, Z1 aber weiterhin festgehalten und die Suche dort fortgesetzt, wo sie beim „alten“ Z2 aufgehört hatte. In der Tabelle unten kann man selbst verfolgen, wie das ganze weitergeht. Es wird jedenfalls immer das letzte gescheiterte Sub-Ziel „zurückgesetzt“ und dort angeknüpft mit der Suche. Sind alle drei Subgoals erfüllt, wird das Prädikat `spielpaarung` auch bewahrheitet und die

aktuelle Belegung aller eingegebenen Individuenvariablen ausgegeben. So kommen alle Ausgaben der Anfrage zu Stande.

meier	**schubert**	**berger**	**meier**	**müller**	**dolittle**
ernst	**klaus**	**ursel**	**egon**	**iris**	**elisa**
hockey	**fussball**	**badminton**	**fussball**	**badminton**	**badminton**
^Z1-					
	^Z1-				
		^Z1+			
^Z2-		^Z1+			
	^Z2-	^Z1+			
		^Z2+^Z1+			
		^Z3-			
		^Z1+	^Z2-		
		^Z1+		^Z2+	
				^Z3+	
		^Z1+			^Z2+
					^Z3+
			^Z1-		
				^Z1+	
^Z2-				^Z1+	
	^Z2-			^Z1+	
		^Z2+		^Z1+	
		^Z3+			
			^Z2-	^Z1+	
				^Z2+^Z1+	
				^Z1+	^Z2+
					^Z3+
^Z2-					^Z1+
	^Z2-				^Z1+
		^Z2+			^Z1+
		^Z3+			
			^Z2-		^Z1+
				^Z2+	^Z1+
				^Z3+	
					^Z2+^Z1+
					^Z3

Die genaue Kenntnis der Funktionsweise des Backtracking ist sehr wichtig wenn man Prolog-Programme schreibt. Wer bereits mal eine imperative Programmiersprache erlernt hat (wie C oder Java etc.), der tut sich erfahrungsgemäß schwerer Prolog zu erlernen, wie jemand, der bisher gar keine Programmsprache kann. Das Backtracking ist einer der Gründe dafür. Ein „guter" Programmierer muss bekanntlich denken wie der Computer beim Abarbeiten des Programms, und das in Prolog nun mal völlig anders als bei den „klassischen" Programmiersprachen.

Rekursionen:

Als nächstes wollen wir uns mit einer weiteren, wichtigen Möglichkeit in Prolog beschäftigen, der *Rekursion*. Die Idee dahinter ist, dass ein Prädikat des „Wenn-Teils" einer Regel sich selbst im „Dann-Teil" der Regel wieder aufrufen kann. Dabei müssen natürlich verschiedene Individuenvariable involviert sein, sonst würde ja eine Endlosschleife entsehen. Und es muss ein „Abbruchkriterium" existieren, eine Subregel also, die dafür sorgt, dass das Backtracking irgend wann endet. Alle Zwischenergebnisse (Variablenbelegungen) dieser hin- und her-Aufruferei werden dabei in sog. *Stacks*, manchmal auch *Heaps* genannt, abgelegt. Wenn also mal eine Fehlermeldung der Art `Stack-Overflow` oder `Heap-Overflow` entsteht, dann habe Sie vermutlich eine Endlosschleife erzeugt (was heißt, dass kein oder ein falsches Abbruchkriterium vorhanden ist).
Das Prinzip der Rekursion sei an einem einfachen Beispiel erklärt. Hierzu sei ein Prolog-Programm zu schrieben, welches Potenzen der Zahl 2 errechnet. Bevor man so was machen kann, muss man sich die Regeln nebst Abbruchkriterium dafür überlegen. Wir wählen folgendes:

Regel 1: $2^0 = 1$
Regel 2: $2^n = 2 \cdot 2^{n-1}$

Regel1 stellt dabei das Abbruchkriterium dar, d.h. ausgehend von einer konkreten Zahl für n wird zunächst durch Regel2 der Exponent solange reduziert, bis er = 0 ist und damit Regel1 greift. Dies sei nachfolgende genauer untersucht.
Wir betrachten z.B. die Berechnung für n=3.

2^3		8
└> $2 \cdot 2^2$		$2 \cdot 4$ <┘
└> $2 \cdot 2^1$		$2 \cdot 2$ <┘
└> $2 \cdot 2^0$		$2 \cdot 1$ <┘
└>	1	<┘
Regel 2	Regel 1	„zurückspulen"

Die Zeilen der obigen Tabelle repräsentieren die Stacks in Prolog. Von links oben nach unten fortschreitend wird auf jedem Stack eine interne Variable angelegt, die an noch keinen Wert gebunden ist. Erst wenn die Rekursion bei der Regel 1 angekommen ist, wird erstmals einer Variablen ein Wert (die 1) zugewiesen. Dann „klettert“ Prolog die Stacks wieder nach oben und bindet die jeweilige interne Variable des Stacks mit dem aktuellen Zahlenwert der Berechnung. „Ganz oben“ (rechts) angekommen, wird das Ergebnis ausgegeben. Es ist also hier wichtig zu wissen, dass die eigentliche Berechnung erst erfolgt, wenn die Rekursion „zurückgespult“ wird. Dies kann manchmal vorteilhaft ausgenutzt werden. Nun zu dem dazugehörigen Prolog-Programm. Wenn wir eingeben:

```
zweihoch(0,1).
zweihoch(N, Erg) :- N1 is N-1,
                    zweihoch(N1,Resterg),
                    Erg is 2*Resterg.
```

dann scheinen wir alles richtig gemacht zu haben. Da es keine klassische Wertzuweisung in Prolog gibt, kann man in Regel 2 nicht einfach sagen: `N is N-1`, sondern muss eine neue Variable erfinden, die den reduzierten Wert erhält. Mit dieser neuen Variablen wird das gleiche Prädikat rekursiv aufgerufen und am Ende jeweils die Zahl 2 mit dem vorherigen Ergebnis (was erst beim Zurückspulen passiert!) multipliziert.
Wenn wir jetzt eingeben

```
?-zweihoch(3,X).
```

so erwarten wir eine Ausgabe `X=8`. Stattdessen bekommen wir eine lapidare Antwort der Form:

```
?- Error     Stack overflow.
```

Um den Fehler zu suchen, kann man in allen Prolog-Interpretern einen sog. „Trace-Modus“ aktivieren. Dies geschieht häufig mit einem Prädikat wie z.B. `Traceon` an der Stelle im Code, wo der Tracemodus beginnen soll. Mit `traceoff` kann er dann wieder ausgeschaltet werden. Der Tracemodus liefert nun jeden Schritt der internen Verarbeitung und ist beim Auffinden von Fehlern sehr hilfreich. Angewendet auf unser kleines Rekursionsprogramm finden wir schnell heraus, dass das Backtracking bei Regel 1 gar nicht endet. Anstatt zurückzuspulen wird einfach weiter danach wieder Regel 2 angewendet und jetzt für negative Exponenten *n* weitergemacht. Die läuft solange, bis alle Stacks in Prolog vergeben sind und daher der Overflow. Wir haben zwar die Regel 1 als Abbruchkriterium vorgesehen, doch dem Programm nirgends mitgeteilt, dass es

nach Regel 1 das Backtracking wirklich anhalten soll. Bei dem vorherigen Programm mit den Mitgliedern hatten wir dieses Problem deshalb nicht, weil das Backtracking beim Durchsuchen von Fakten automatisch beim letzten Fakt beendet wird. Hier im Rekursionsprogramm haben wir aber keine endliche Anzahl Fakten, sondern neben dem ersten Fakt noch eine rekursive Regel, die brav immer wieder von Neuem aufgerufen wird.
Nun gibt es in Prolog die Möglichkeit, mit einem Standardprädikat namens „cut“ das Backtracking gewaltsam zu beenden. Dieses Standardprädikat wird in Prolog mit einem Ausrufungszeichen bezeichnet, also das Prädikat heißt in Prolog :

```
!
```

Es kann in Prolog überall platziert werden, wo man einen Abbruch wünscht. Wir wünschen ihn bei Regel 1 und schreiben daher:

```
zweihoch(0,1):-!.
zweihoch(N, Erg) :- N1 is N-1,
                    zweihoch(N1,Resterg),
                    Erg is 2*Resterg.
```

Jetzt führt der Aufruf

```
?-zweihoch(3,X).
```

erwartungsgemäß zu dem Ergebnis

```
?- X=8
Yes
```

Bei der Gelegenheit möchte ich gleich noch ein anderes Standardprädikat zur Beeinflussung des Backtracking angeben.
Zu diesem Zweck betrachten wir folgende Variante unseres Mitgliederprogramms:

```
/* Fakten */
mitglied(schubert,klaus,m,1962,fussball).
mitglied(berger,ursel,w,1968,badminton).
mitglied(meier,egon,m,1955,fussball).
mitglied(mueller,iris,w,1963,badminton).
mitglied(dolittle,elisa,w,1968,badminton).
beitrag(hockey,30.50).
beitrag(fussball,42.30).
```

```
beitrag(badminton,10).

/* Regeln */
start :- mitglied(N,V,_,_,S),beitrag(S,G),
         write(N),nl, write(V),nl, write(G).
```

Die Regel `start` enthält im linken Teil keine Individuenvariablen. Im rechten Teil wird der Beitrag für die Mitglieder errechnet, abhängig von deren Sportart. Das Standardprädikat `nl` steht für „new line“ und erzeugt bei der Ausgabe einen Zeilenumbruch. Wenn wir nach dem Laden nun eingeben:

```
?-start.
```

so erscheint:

```
schubert
klaus
42.3

Yes
```

Es kommt also nur das erste Mitglied. Der Grund ist, dass das Backtracking – wie schon erwähnt- bei Eingabe von Prädikaten ohne Individuenvariablen bereits bei der ersten Bewahrheitung aufhört. Das wollen wir hier aber nicht, wir hätten gerne alle Mitglieder nebst deren Beiträge gesehen. Während wir vorhin das Backtracking mittels `cut (!)` künstlich anhielten, möchten wir jetzt das Backtracking gerne künstlich dazu zwingen, weiter zu machen. Das können wir errechen, in dem wir ein weiteres, nicht erfüllbares Subgoal an die Regel (durch Konjunktion) dranhängen. Wir könnten z.B. einfach die Regel so abändern:

```
start :- mitglied(N,V,_,_,S),beitrag(S,G),
   write(N),nl, write(V),nl, write(G), nl, 3=5.
```

Das Subgoal `3=5` ist natürlich nie erfüllbar, d.h. das Backtracking macht weiter bis alle Mitglieder durch sind. Und es werden tatsächlich jetzt alle Mitglieder mit ihren Beiträgen geliefert.
Jetzt braucht man aber nicht als weiteres Subgoal `3=5` hinzuzufügen, denn es gibt ein Standardprädikat dafür: `fail`. Es verhält sich wie `3=5`, scheitert also immer, und mit dieser Regel:

```
start :- mitglied(N,V,_,_,S),beitrag(S,G),
   write(N),nl, write(V),nl, write(G), nl, fail.
```

erhalten wir nach Eingabe von

```
?-start.
```

das Ergebnis:

```
schubert
klaus
42.3
berger
ursel
10
meier
egon
42.3
mueller
iris
10
dolittle
elisa
10

No
```

Die Antwort `no` am Schluss zeigt an, dass das Prädikat `start` nicht erfüllt werden konnte, was folgerichtig ist, denn `fail` scheitert ja immer.

Es sei wegen der wichtigen Bedeutung der Rekursionen noch ein zweites Beispiel angegeben. Diesmal versuchen wir rekursiv die Berechnung der e-Funktion.
Um dies zu bewerkstelligen, brauchen wir wieder (mindestens) zwei Regeln: Eine für das Abbruchkriterium und eine für die Rekursion. Für das Abbruchkriterium nutzten wir aus, dass die e-Funktion in der Nähe des Ursprungs näherungsweise den Wert ihrer Tangente besitzt. Wenn also $y=e^x$ die einfache e-Funktion darstellt, dann lautet bekanntlich die Tangentengleichung im Ursprung: $y=1+x$. Für „kleine“ x nehmen wir sie als Näherung und gleichzeitig als Abbruchkriterium. Für die Rekursionsregel brauchen wir demnach eine Formel, die dafür sorgt, dass das x immer kleiner wird bei jedem rekursiven Aufruf, so dass auch irgendwann mal Regel 1 greifen kann. Wir setzten folgendes an (es gibt auch andere Möglichkeiten):

Regel1:	$e^x = 1 + x$	falls $abs(x) <= 10^{-5}$
Regel2:	$e^x = (e^{x/2})^2$	falls $abs(x) > 10^{-5}$

Regel 2 ist natürlich sehr banal, denn wir ziehen die Wurzel und Quadrieren anschließend wieder, was am Wert nichts ändert. Aber wir schaffen es damit, das x kleiner wird beim Aufruf (nämlich halbiert). In Prolog sieht das Ganze dann so aus:

```
e_hoch(X,Erg):- abs(X)=<0.00001,
                Erg is 1+X,!.

e_hoch(X,Erg):- abs(X)>0.00001,X1 is X/2,
                e_hoch(X1,Resterg),
                Erg is Resterg*Resterg.
```

Wir sehen, dass hier wieder das `cut` zum erzwungenen Beenden der ersten Regel eingesetzt wurde. Die Eingabe der Form:

```
?-e_hoch(1,Ergebnis).
```

Führt erwartungsgemäß zu:

```
Ergebnis = 2.71827

Yes
```

An der Stelle noch eine Warnung. Wir haben uns bisher wenig Gedanken über das „Universum“ unserer Prädikate gemacht. Aus Kapitel 3 wissen wir, dass das Universum diejenige Menge darstellt, aus der die Individuen kommen. Normalerweise ist durch die Fakten das Universum festgelegt. Betrachten wir hierzu folgendes einfache Programm:

```
maennlich(karl).
maennlich(hans).
maennlich(egon).

weiblich(X) :- not(maennlich(X)).
```

Wir machen jetzt folgende Abfragen:

```
?- maennlich(karl).
Yes

?- maennlich(hans).
Yes

?- weiblich(erna).
```

```
Yes

?- weiblich(hans).
No

?- weiblich(hugo).
Yes
```

Bis auf das letzte Resultat sieht das alles gut aus. Doch warum ist Hugo weiblich? Die Antwort liegt im zugrundeliegenden Universum. Hugo zählt nicht dazu. Würden wir nämlich abfragen:

```
?- maennlich(hugo).
```

so wäre die Prolog-Antwort:

```
No
```

denn das Fakt `maennlich(hugo)` existiert nicht. Damit wird `not(maennlich(hugo))` natürlich wahr und somit Hugo als weiblich deklariert. Das Problem ist also deswegen aufgetaucht, weil wir das Prädikat `weiblich` über das „Komplement“ von `maennlich` definierten (also alles, was nicht männlich ist, soll weiblich sein). Und das ist die Warnung: Solche `not(...)`-Konstruktionen sind sehr zu überlegen und zu vermeiden, wenn möglich. Denn wie in unserem Beispiel sehen wir dem Ergebnis nicht an, *warum* es zustande kommt! Ist Hugo nun weiblich, weil er eine Frau ist oder weil er einfach nicht vorkommt in den Fakten der Männer? Hier ist also Vorsicht angebracht!

Listen:

Zum Abschluss diese Kapitels wollen wir uns nun mit der wichtigen Datenstruktur „Liste“ beschäftigen. Sie bezieht sich auf die in Prädikaten benutzten Individuen und ermöglicht es, solchen Individuen noch eine „innere Struktur“ zu geben. Insbesondere können Listen selbst weitere Individuen beinhalten (eine Liste von Individuen, die selbst ein Individuum darstellt). Man kann also Individuen „verschachteln“ mit Hilfe von Listen.
Ein Liste ist in Prolog dabei so definiert, dass sie ein Aufzählung von Individuen ist, durch Komma getrennt, und mit eckigen Klammern umgeben, z.B.:

```
Liste1 = [a, b, c]
Liste2 = [1, 2, 3]
Liste3 = [X,Y,Z]
Liste4 = [[a,b,c], [1,2,3]]
```

Schon aus diesen Beispielen können wir einige Eigenschaften solcher Listen ablesen:

1. Die Anzahl der Elemente einer Liste ist beliebig, insbesondere kann sie auch Null sein ("Leere Liste")
2. Die Elemente einer Liste können selbst wieder Listen sein
3. Die Reihenfolge der Listenelemente ist nicht beliebig (im Gegensatz zu einer Menge)
4. Es können gleiche Elemente mehrfach in einer Liste vorkommen (im Gegensatz zu einer Menge)
5. Die Elemente einer Liste müssen alle vom gleichen Datentyp sein (manche PROLOG-Interpreter lassen allerdings auch gemischte Datentypen innerhalb einer Liste zu)

Das alles wäre nun nichts besonderes, wenn es nicht eine ganz spezielle Möglichkeit gäbe, auf Listen *zuzugreifen*. Man kann nämlich –neben der Darstellung von Listen wie in obigen Beispielen angegeben- sich eine Liste aus zwei Elementen zusammengesetzt denken: einem *Listenkopf* und einem *Listenrumpf*. Der Listenkopf bezeichnet dabei eine Zusammenfassung der ersten *n>0* Listenelemente. Der Listenrumpf ist die Liste der restlichen Listenelemente. Während also der Listenkopf die ersten *Listenelemente* darstellt, ist der Listenrumpf selbst eine *Liste*, manchmal daher auch *Restliste* genannt. Beim Zugriff unterscheidet man Kopf und Restliste dadurch, dass man einen senkrechten Strich dazwischen anbringt (siehe Beispiele unten).
Diese eigentlich simple Zerlegung stellt innerhalb von Prolog ein mächtiges Werkzeug für den Programmierer dar, insbesondere, wenn in Zusammenhang mit Rekursionen eingesetzt. Hier nun ein paar Beispiele solcher Listenzugriffe:

Liste	**Zugriff durch**	**liefert**
`[12,17,4]`	`[X|Y]`	`X = 12` `Y = [17,4]`

`[1,2,3,4]`	`[X,Y|Z]`	`X = 1`
		`Y = 2`
		`Z = [3,4]`

`[1,2]`	`[X,Y|Z]`	`X = 1`
		`Y = 2`
		`Z = [] (leere Liste).`

`[[1,2],[25,6]]`	`[X|Y]`	`X = [1,2]` `Y = [[25,6]]`

Insbesondere sind damit in Prolog folgende Schreibweisen gleichwertig:

```
[12,17,4] = [12|[17,4]]

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

und so weiter.

Erfahrungsgemäß ist das Programmieren mit Listen nicht so einfach und setzt etwas „Gefühl“ dafür voraus. Dieses erwirbt man allerdings nur dadurch, dass man selbst einige Prologprogramme mit Listenverarbeitung schreibt (und aus den Fehlern lernt). Es sollen zum besseren Verständnis nun einige Prolog-Beispiele mit Listenverarbeitung betrachtet werden.
Zu diesem Zweck betrachten wir ein Programm, welches zwei Listen miteinander verkettet:

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

Gibt man jetzt z.B. ein:

```
?-ver([1,2,3],[4,5],E).
```

so erhält man:

```
E = [1, 2, 3, 4, 5]

Yes
```

Offenbar werden die Elemente der zweiten Liste hinter die erste Liste angehängt. Betrachtet man das zugehörige Prolog-Programm, so mag das auf den ersten Blick verwundern. Doch wir wollen uns einmal genau ansehen, was beim Abarbeiten dieses Programms passiert. Hierzu wurde in SWI-Prolog das Programm im sog. „Trace-Modus“ ausgeführt. Dies geschieht dort, in dem man im SWI-Interpreter nach dem Laden des Programm-Files die Angabe macht:

```
?- trace(ver).
```

`ver` war ja der Name des interessierenden Prädikats.
Als Antwort erhält man daraufhin:

```
%         ver/3: [call, redo, exit, fail]

Yes
```

Diese Angabe zeigt, dass `ver` aus 3 Individuen besteht und danach folgt die Beschreibung der 4 Spalten einer getraceten Ausgabe: Spalte 1 beschreibt die Aktion (normalerweise `call` oder `exit`), die 2. Spalte ist eine interne Nummerierung, auf die im Laufe des Abarbeitens verwiesen werden kann, die 3. Spalte beschreibt die (interne) Ausgabe an die Stacks und die 4. Spalte sagt uns `fail` falls was nicht bewahrheitet werden kann. Beim weiteren Aufrufen steht vor dem Fragenzeichen das Wort `[debug]` um anzudeuten, dass wir uns im Trace-Modus befinden. Durch die Eingabe

```
[debug] ?- ver([1,2,3],[4,5],E).
```

erhalten wir folgende Ausgabe:

```
 T Call: (1) ver([1, 2, 3], [4, 5], _G505)
 T Call: (2) ver([2, 3], [4, 5], _G586)
 T Call: (3) ver([3], [4, 5], _G589)
 T Call: (4) ver([], [4, 5], _G592)
 T Exit: (4) ver([], [4, 5], [4, 5])
 T Exit: (3) ver([3], [4, 5], [3, 4, 5])
 T Exit: (2) ver([2, 3], [4, 5], [2, 3, 4, 5])
 T Exit: (1) ver([1, 2, 3], [4, 5], [1, 2, 3, 4, 5])

E = [1, 2, 3, 4, 5]

Yes
```

Wir wollen jetzt diese Ausgabe analysieren, indem wir Zeile für Zeile zusammen mit dem Programm-Code untersuchen, was genau passiert.

Die Eingabe `ver([1,2,3],[4,5],E)` wird zunächst mit der Regel Nr. 2 angewendet (da die Regel 1 nur für eine leere Liste gilt). Diese Regel lautet:

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

Setzt man da jetzt die eingegebenen Listen hinein, sieht es so aus:

```
ver([1|[2,3]],[4,5],[1|R3]):-ver([2,3],[4,5],R3).
```

Man sieht, dass die Restliste R3 (noch) unbekannt ist. Ungeachtet dessen wird das Backtracking „angeworfen“, denn damit die linke Seite wird wahr, muss es die rechte werden. Das Backtracking versucht nun, die rechte Seite zu bewahrheiten. Da sowohl rechts wie auch links die Restliste `R3` noch nicht bestimmt werden kann, wird ein Stack angelegt. Die Stackvariable heißt `_G505`. Es ist

übrigens Prolog-Standard, dass intern vergebene Variable mit einem Underscore beginnen. An der ersten Ausgabe des Trace-Modus sieht man, dass `_G505` der Liste `[1|R3]` zugeordnet wurde. Das ist allgemein so, dass die „eigentliche“ Variable, also `R3`, zusammen mit der Rechenvorschrift, in der sie vorkommt, abgelegt wird.
Um jetzt die rechte Seite der Regel , also `ver([2,3],[4,5],R3)` zu bewahrten, wird erneut die Regel Nr. 2 des Prolog-Programms aufgerufen. Es wird also

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

jetzt mit

```
ver([2,3],[4,5],R3)
```

aufgerufen. Dies bedeutet, dass eingesetzt da steht:

```
ver([2|[3]],[4,5],[2|R3]):-ver([3],[4,5],R3).
```

Das „neue“ R3 hat mit dem des vorherigen Stacks nichts zu tun! Die Variable heißt zwar gleich, wird aber inhaltlich später einen anderen Wert bekommen als das vorherige R3. Auch das gilt allgemein: Bei jedem erneuten (auch rekursiven!) Aufruf einer Regel sind alle vorkommenden Variablen wieder ungebunden. Dies ist auch in der nächsten Ausgabe unseres Trace-Modus zu beobachten: Die ausgegebene Zeile

```
T Call: (2) ver([2, 3], [4, 5], _G586)
```

welche wieder die linke Seite identifiziert, bekommt einen neue Stackvariable, und der Vergleich mit oben zeigt, dass dort jetzt abgelegt ist:

```
_G586 = [2|R3]
```

Um wieder die rechte Seite der Regel zu bewahrheiten, wird diese erneut rekursiv mit der Regel 2 des Programms aufgerufen, also

```
ver([3],[4,5],R3).
```

ist jetzt identifiziert mit

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

So dass eingesetzt folgendes herauskommt:

```
ver([3|[]],[4,5],[3|R3]):-ver([],[4,5],R3).
```

Aus der Trace-Ausgabe sehen wir wieder, dass diezughörige Stack-Variable folgenden Inhalt bekommt:

```
_G589 = [3|R3]
```

Der nächste rekursive Aufruf sieht dann so aus:

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

Es wird damit

```
ver([[]],[4,5],R3):-ver([],[4,5],R3).
```

Auch jetzt wird eine Stackvariable angelegt, und aus der Trace-Ausgabe sehen wir welche:

```
_G592 = R3
```

Der nächste rekursive interne Aufruf ist also `ver([],[4,5],R3).`
Doch diesmal „greift“ die Regel 1, d.h.

```
ver([],L,L).
```

Es können jetzt alle Individuenvariablen bestimmt werden, so dass das Backtracking hier endet. Und damit wird auch zum ersten Mal (das „letzte“ der) `R3` bestimmbar: Es ist gemäß der Regel 1 gleich `L`, also konkret gleich

```
R3 = [4,5]
```

Wie wir auch in der Trace-Ausgabe jetzt beobachten, werden die Stacks nebst deren Berechnungen „zurückgespult“, d.h. von unten nach oben werden die Stackvariablen berechnet. Zur Erinnerung noch mal deren Inhalte (von oben nach unten):

```
_G505 = [1|R3]
_G586 = [2|R3]
_G589 = [3|R3]
_G592 = R3
```

Jedes R3 auf jedem Stack hat –wie schon gesagt- einen anderen Wert. Das Zurückspulen der Stacks liefert dann (von unten nach oben):

```
_G592 = [4,5]
_G589 = [3|[4,5]]= [3,4,5]
_G586 = [2|[3,4,5]]=[2,3,4,5]
_G505 = [1|[2,3,4,5]]=[1,2,3,4,5]
```

Es wurde hier offenbar vom Programmierer die Tatsache ausgenutzt, dass die Ergebnisliste erst beim Zurückspulen entsprechend zusammengesetzt wird, damit die Reihenfolge der Listenelemente am Schluss stimmt. Der Trace-Modus verrät uns übrigens auch, wann das Backtracking endet: Wenn in der zweiten Spalte statt `call` dann `exit` steht. Das heißt nämlich, dass jetzt die Aufrufe des Backtracking zu Ende sind und das Zurückspulen beginnt.
Man sieht einem Prädikat normalerweise nicht immer gleich an, welche Individuen sich auf eine Eingabe und welche sich auf eine Ausgabe beziehen. Zu diesem Zweck gibt es den Begriff des sog. *Fließmusters*. Ein Fließmuster ist ein dokumentarischer Ausdruck, der im Prologprogramm selbst höchstens als Kommentar auftaucht. Es hat sich eingebürgert, dass das oder die Fließmuster eines Prädikates den Namen des Prädikates erhalten und anstatt der Individuen die Buchstaben `i` für Input und `o` für Output hingeschrieben werden. In unserem Fall würde man also schrieben

Fließmuster: `ver(i,i,o)`

Sind mehrere Aufruf-Kombinationen für das gleiche Prädikat möglich, dann besitzt es auch mehrere Fließmuster.

Zum Abschluss des Kapitels werden hier noch ein paar nützliche Listenverarbeitungsprogramme in Prolog angegeben, und es sei dem Leser überlassen, diese selbst auszuprobieren bzw. im Trace-Modus zu untersuchen, was genau dabei geschieht. Letzteres ist insbesondere für Prolog-Programmieranfänger dringend empfohlen!

Das nachfolgende Programm entfernt ein `Wort` aus einer `Liste` und schreibt das Ergebnis in eine Liste namens `Ergliste`:

```
entfernen(_,[],[]).

entfernen(Wort,Liste,Ergliste):-
       Liste=[Wort|Restliste],
       entfernen(Wort,Restliste,Ergliste).
```

```
entfernen(Wort,Liste,Ergliste):-
       Liste=[Kopf|Restliste],
       not(Wort=Kopf),
       Ergliste=[Kopf|Ergrest],
       entfernen(Wort,Restliste,Ergrest).
```

Fließmuster: `entfernt(i,i,o)`

Man sieht also dem Fließmuster den Datentyp nicht an (das erste `i` betrifft ein Listenelement, das zweite `i` ein ganze Liste und das `o` auch eine Liste).
Das nächste Programm ermöglicht es unter Zuhilfenahme des Prädikats `entfernen` des vorherigen Programms, eine Liste von gleichlautenden Listenelementen zu bereinigen:

```
bereinigen([],[]).

bereinigen(Liste,Ergliste) :-
       Liste=[Kopf|Restliste],
       entfernen(Kopf,Restliste,Zwi1),
       bereinigen(Zwi1,Zwi2),
       Ergliste=[Kopf|Zwi2].
```

Zum Beispiel liefert ein Aufruf der Form:

```
?- bereinigen([1,2,2,3,2,4],E).
```

als Ergebnis

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

Yes
```

Es sei abschließend noch erwähnt, dass es auch Standard-Prädikate gibt, welche die im Interpreter geladene Wissensbasis manipulieren können. So kann man „per Programm“ z.B. Fakten hinzufügen oder entfernen, und man kann das Ergebnis davon in eine Datei zurückschreiben. Darauf soll aber im Rahmen dieses Buches nicht näher eingegangen werden, zumal diese Standardprädikate auch für Prolog-Interpreter verschiedener Hersteller sowohl in der Namensgebung als auch im Aufbau voneinander abweichen können.

Im nächsten Kapitel wenden wir uns nun einer Anwendung der Prolog-Programmierung zu: den Expertensystemen.

Übungen zum Selbsttest:

1. Schreiben Sie ein Prolog-Programm zur (rekursiven) Bestimmung von n!.

2. Schreiben Sie ein Prolog-Programm zur rekursiven Bestimmung von *sin(x)* und *cos(x)* **ohne** dass zu deren Berechnung spezielle Funktionen benutzt werden müssen (ANSATZ: Eulers´sche Formeln im Komplexen und trigonometrische Additionstheoreme ausnutzen).

3. Schrieben Sie ein Prolog-Programm zu dem Beispiel auf Seite 43 und verifizieren Sie damit die Wahrheit der Aussage, dass Markus Cäsar hasste.

4. Einstein hat folgendes Rätsel (angeblich) im 19. Jahrhundert verfasst. Er behauptete, 98% der Weltbevölkerung seien nicht in der Lage, es zu lösen. Es lautet:

   1. Es gibt 5 Häuser mit je einer anderen Farbe
   2. In jedem Haus wohnt eine Person anderer Nationalität
   3. Jeder Hausbewohner bevorzugt ein bestimmtes Getränk, raucht eine bestimmte Zigarettenmarke und hält ein bestimmtes Haustier
   4. Keine der 5 Personen trinkt das gleiche Getränk, raucht die gleichen Zigaretten oder hält das gleiche Tier wie seine Nachbarn

   Über die Bewohner ist zusätzlich folgendes bekannt:

   - Der Brite lebt im roten Haus.
   - Der Schwede hält einen Hund.
   - Der Däne trinkt gern Tee.
   - Das grüne Haus steht direkt links neben dem weißen Haus.
   - Der Besitzer des grünen Hauses trinkt Kaffee.
   - Die Person, die Pall Mall raucht, hält einen Vogel.
   - Der Mann, der im mittleren Haus wohnt, trinkt Milch.
   - Der Besitzer des gelben Hauses raucht Dunhill.
   - Der Norweger wohnt im 1. Haus.
   - Der Marlboro-Raucher wohnt neben dem, der eine Katze hält.
   - Der Mann, der ein Pferd hält, wohnt neben dem, der Dunhill raucht.
   - Der Winfield-Raucher trinkt gern Bier.
   - Der Norweger wohnt neben dem blauen Haus.
   - Der Deutsche raucht Rothmans.
   - Der Marlboro-Raucher hat einen Nachbarn, der Wasser trinkt.

Die Frage nun ist: Wem gehört der Fisch?

Schreiben Sie zur Lösung ein entsprechendes Prolog-Programm.

5. Expertensysteme

Ursprünglich waren Expertensysteme Anwendungsprogramme, welche logische Schlussfolgerungen aus einer Wissensbasis ziehen konnten, oder überprüften, ob eine Aussage aus einer vorhandenen Wissensbasis abgeleitet werden kann. Die Übungen zum Selbsttest aus Kapitel 4 (Prolog) haben beide Fälle berücksichtigt: In Aufgabe 3 wird eine einzugebende Aussage verifiziert, während in Aufgabe 4 (Einstein-Rätsel) das Programm selbst eine Lösung sucht. Nun ist es heutzutage Anwendern in der Regel nicht zuzumuten, dass sie selbst Prolog-Programme entwickeln. Und auch für einen Entwickler kann es hilfreich sein, wenn er für immer wiederkehrende, ähnliche Aufgaben, geeignete Werkzeuge zur Verfügung hat. Der Begriff eines Expertensystems hat sich daher erweitert: Es wird damit immer noch ein Anwendungsprogramm im obigen Sinnen bezeichnet, aber auch immer gleichzeitig erwartet, dass ein Expertensystem einem Entwickler hilft, sein Programm zu schreiben (durch spezielle Tools für seine Arbeit); und sie sollen aber auch Software erzeugen, die ein „unbedarfter“ Anwender benutzen kann. Profi-Expertensysteme leisten also beides: sie helfen dem *Wissensingenieur* beim Entwickeln und können ablauffähige Anwenderprogramme mehr oder weniger „auf Knopfdruck“ erzeugen.
Allgemein besteht ein Expertensystem aus folgenden Komponenten:

1. *Wissensbasis*. Das ist die Menge der Fakten und Regeln, formuliert z.B. in einer deskriptiven Sprache wie Prolog.
2. *Inferenzmaschine*. Das ist ein "Schlussfolgerungsprogramm", i.d.R. Bestandteil der benutzen deskriptiven Programmiersprache; ein Mechanismus, der eine gegebene Behauptung auf logische Verträglichkeit mit der Wissensbasis untersucht (z.B. mit Hilfe des Resolutionstheorems).
3. *Dialogkomponente*. Dies ist eine Benutzeroberfläche, welche dem Anwender erlaubt, ohne Kenntnis der im Hintergrund benutzen deskriptiven Sprache mit dem System zu arbeiten.
4. *Trace-Komponente*. Hierbei handelt es sich um eine "Historie", welche die Schlussfolgerungskette, die die Inferenzmaschine zur Verifikation oder Falsifikation einer Aussage benutzt hat, dem Benutzer auf Wunsch anzeigt.
5. *Wissensveränderungskomponente*. Dieser Teil des Expertensystems erlaubt dem Entwickler oder autorisierten Benutzer, weitere Fakten und Regeln zu den vorhandenen hinzuzufügen bzw. zu editieren.

Expertensysteme setzen auch bei dem Anwender Kenntnisse logischer Schlussfolgerungsregeln voraus. Populäre Anwendungen sind z.B. Diagnose-Systeme im Bereich der Medizin. Um die Fakten und Regeln zusammenzustellen, bedarf

es seitens der Entwicklung von Expertensystemen eines speziell dafür ausgebildeten *Wissensingenieurs*, welcher in der Lage ist, die mit einem Fachmann des jeweiligen Wissensgebietes gesammelten Fakten und Regeln (*Wissensakquisition*) in eine aussagenlogische Form zu bringen und sie dann in einer deskriptiven Programmiersprache zu formulieren (ggf. mit Hilfe von entsprechenden Entwicklungstools). Die meisten Softwarekomponenten eines Expertensystems werden nach den üblichen Regeln des Softwareengineerings[2] entwickelt. So kann man z.B. grundsätzlich nach dem Wasserfallmodell oder dem Spiralmodell vorgehen. Wenn es aber um die Entwicklung der Komponente Wissensbasis geht, ist eine Spezialform des Softwareengineerings, nämlich das *Wissensengineering* angesagt.

Der Wissensingenieur geht dabei folgendermaßen vor: Er trifft sich mit dem Fachmann (z.B. einem Arzt) und lässt sich von diesem die „Wenn-Dann-Regeln" in verbaler Form nennen. Der Wissensingenieur braucht dafür normalerweise von dem eigentlichen Fachgebiet (hier: Medizin) nichts zu verstehen. Nach dem „Sammeln" der Regeln und Fakten wird der Wissensingenieur versuchen, diese Regeln in Horn-Klausen umzusetzen. Dazu muss er ausgiebige Kenntnisse über Aussagen- und Prädikatenlogik besitzen (vgl. Kapitel 2 und 3), denn es sind alle Aussagen in bereinigte, skolemisierte universell quantifizierte unvollständige konjunktive Normalformen zu überführen. Dabei kann es sein, dass der Wissensingenieur wieder Rücksprache mit dem Fachmann halten muss, da evtl. manche Regelen nicht ohne weiteres in definite oder wenigstens schwache Horn-Klauselform gebracht werden können (z.B. weil sie kein positives Literal enthalten).

Sind alle Horn-Klauseln zusammengestellt, kann die Wissensbasis programmiert werden. Dazu wird normalerweise eine deskriptive Programmiersprache wie Lisp oder Prolog benutzt (vgl. Kapitel 4).

Nach Erstellung der Wissensbasis mit einer deskriptiven Programmiersprache müssen noch die Benutzerschnittstellen programmiert werden (incl. Trace- und Wissensveränderungskomponente). Dazu muss nicht unbedingt eine deskriptive Programmiersprache benutzt werden, denn die Dialogkomponente fungiert lediglich als Schnittstelle zwischen dem Bediener und der Wissensbasis.

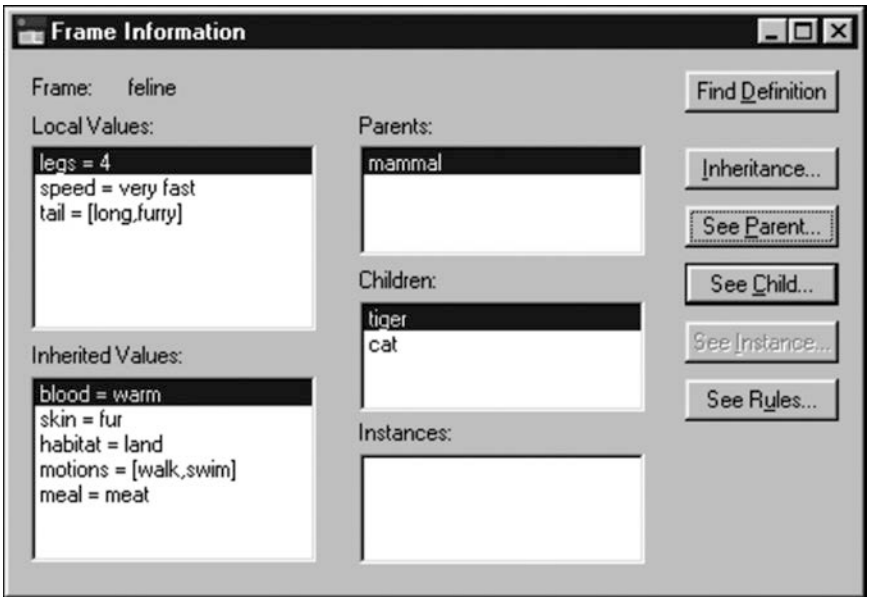
Es gibt mittlerweile eine große Auswahl an professionellen Expertensystemen auf dem Markt. Wir wollen den Umfang und die Werkzeuge eines Profi-Expertensystems an dem Systems *Flint*®, ein Produkt von LPA[3] in England, demonstrieren.

Das Expertensystem setzt ein, wenn der Wissensingenieur wie beschrieben alle Wenn-Dann-Regeln in Form von Horn-Klauseln zusammengestellt hat (wobei

[2] vgl. z.B. Zöller-Greer, Peter: *Software Analyse und Design*, Composia Wächtersbach, 2007.

[3] Siehe http://www.lpa.co.uk

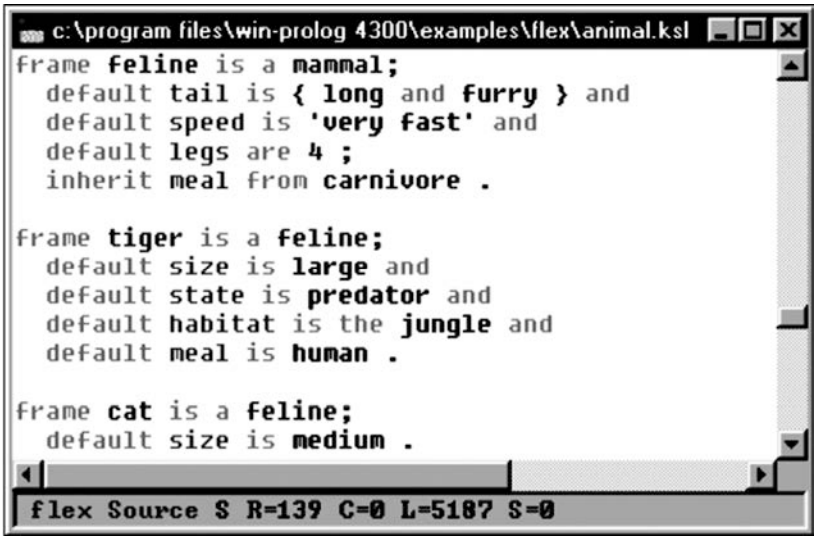
manche Expertensysteme wie Flint hier nicht so pingelig sind, d.h. hier können auch Regeln mit einem Regeleditor bearbeitet werden). Zu Erwähnen ist dabei, dass die Fakten der Wissensbasis in einem erweiterten Modus mit gewissen Datenbank-Features ausgestattet werden können, die dann auch in einer Datenbank abgelegt werden. Diese Möglichkeit ist sehr hilfreich, denn so können z.B. *Frames* zu einem Fakt angelegt werden, deren Instanzen gewisse Attribute anlegen lassen (die dann auch mit abgefragt werden können). Solche Expertensysteme nennt man auch Hybrid-Expertensysteme. Flint stellt die Bearbeitung von Frames so dar[4]:



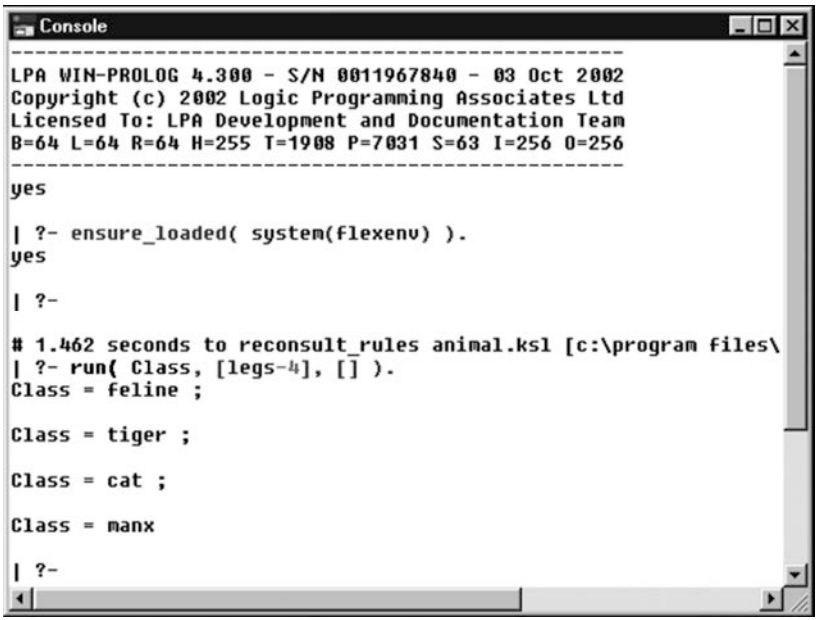
Dieser Frame legt Attribute und weitere Instanzen des Säugetiers „feline“ (=Spezies: Katzen) fest. Frames haben die Eigenschaften objektorientierter Klassen aus dem Softwareengineering und damit vererbbare Eigenschaften.

Die sich daraus ableitende Klassenbeschreibung sehen wir im nächsten Bild. Sie ist hier auch noch weiter editierbar oder kann von vorn herein damit geschrieben werden.

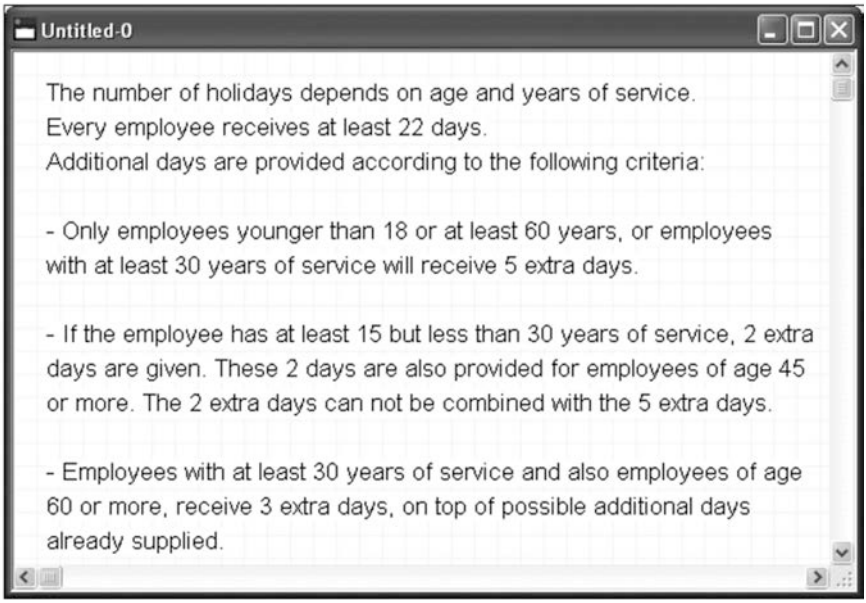
[4] Qullen aller Print-Screens: http://www.lpa.co.uk



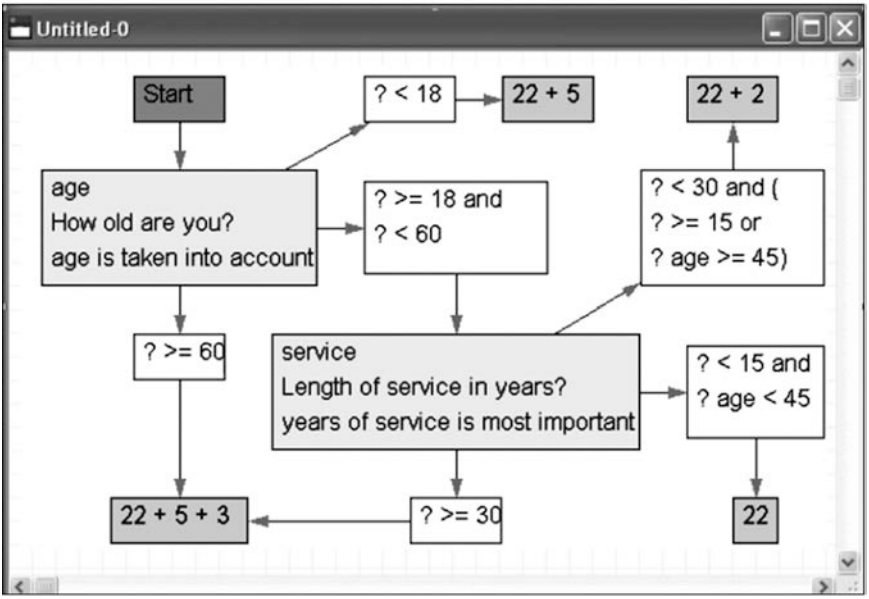
Diese Eigenschaften können im integrierten Prolog-Interpreter abgefragt werden:



Eines der mächtigsten Werkzeuge von Expertensystemen für die Entwicklung sind Regeleditoren. Sie erlauben es, mit grafischen Hilfsmitteln das Regelwerk einzugeben und darzustellen. Ein Generator erzeugt dann im Hintergrund das entsprechende Prolog-Programm. Betrachten wir z.B. folgendes „verbale“ Regelwerk, welches die Herleitung von Urlaubsanspruch für die Mitarbeiter in einer Firma leistet, abhängig von verschiednen Faktoren (Alter, Betriebszugehörigkeit etc.):

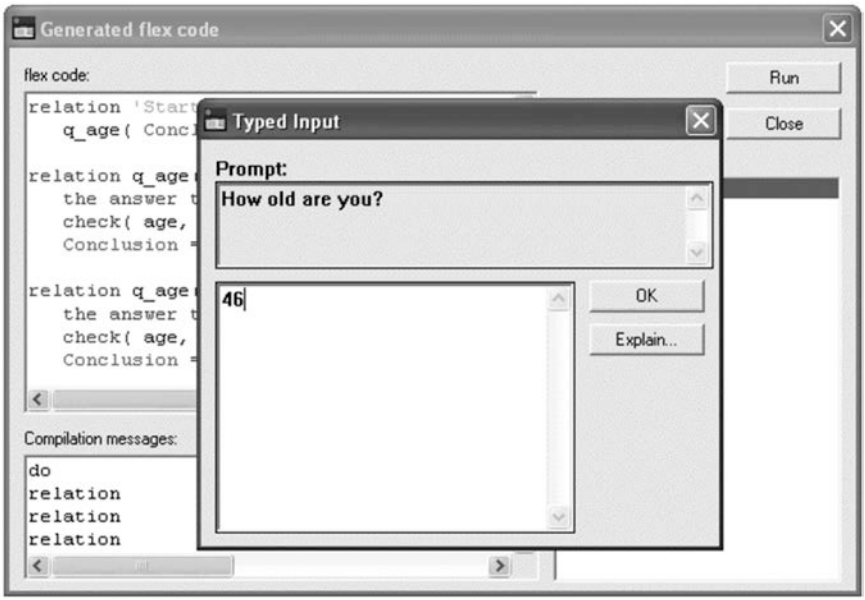


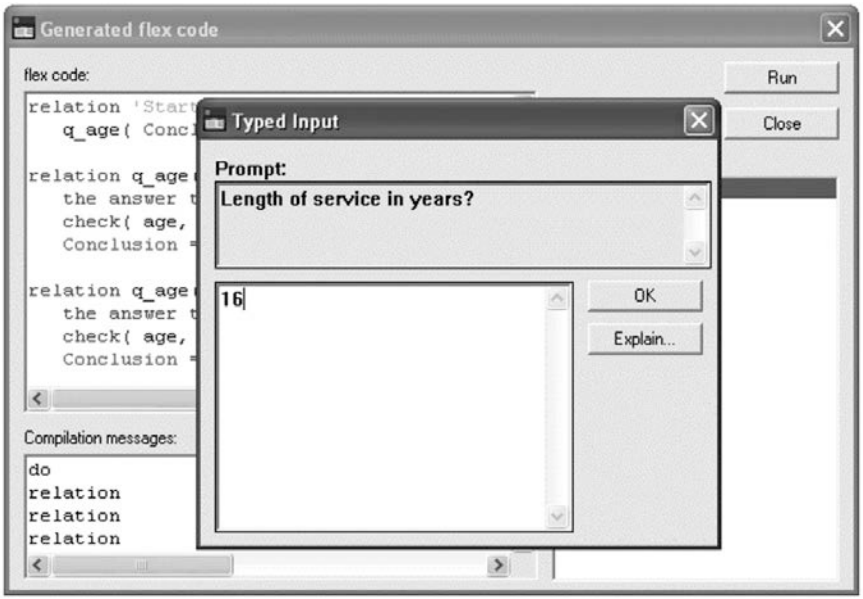
Wenn man alle diese Regeln in den grafischen Regeleditor einzeichnet, so könnte das ganze folgendermaßen Aussehen:

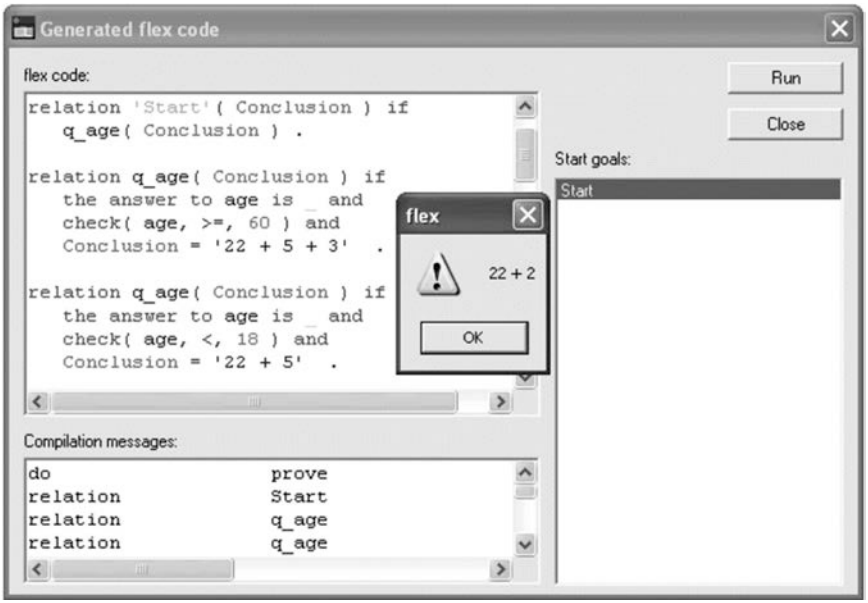


Hierbei sind die hellgrau unterlegten Boxen die Ausgabe an den Benutzer und es wird eine Eingabe erwartet. Nach Eingabe wird abhängig von derselben eine Ausgabe errechnet (dunkelgraue Boxen) oder weitergefragt. Dieses Regelwerk ist jetzt natürlich relativ einfach und wäre auch ohne Regeleditor leicht zu programmieren. Die Stärke dieses Werkzeugs liegt darin, dass sehr komplexe Regelwerke übersichtlich angelegt und dargestellt werden können. In Flint sind diese Boxen durch verschiedene Farben gekennzeichnet.

Ruft ein Nutzer das Programm auf, so bekommt er folgenden Dialog zu sehen:


Generated flex code
flex code:
Run
Close
Typed Input
Prompt:
How old are you?
46
OK
Explain...
Compilation messages:
do
relation
relation
relation


Generated flex code
flex code:
Run
Close
Typed Input
Prompt:
Length of service in years?
16
OK
Explain...
Compilation messages:
do
relation
relation
relation



Dieser kleine Ausschnitt gibt bereits einen Einblick in die Entwicklungsarbeit solcher Expertensysteme. Bei „großen“ Expertensystemen ist natürlich –wie im Softwareengineering immer- eine Arbeitsteilung und damit verstärktes Teamworking gefragt. So kann z.B. eine Trennung erfolgen zwischen der Wissensanalyse und der Programm-Entwicklung.

Das nächste Kapitel beschäftigt sich mit einer Spezialform von Expertensystemen: Den Fuzzy-Expertensystemen. Diese basieren auf Erkenntnissen der Fuzzy-Logik und finden in vielen Geräten das täglichen Bedarfs ihren Einsatz.

Übungen zum Selbsttest:

1. Stellen Sie den Ablauf für die Entwicklung eines Expertensystems unter dem Gesichtspunkt des Wissensengineering grafisch dar.

6. Fuzzy-Systeme

Expertensysteme wie in Kapitel 5 beschrieben besitzen einige Nachteile. Einer davon ist, dass das System der aufgestellten logischen Wenn-Dann-Regeln nur „scharfe“ Erfüllungen kennt: Entweder ist der Wenn-Teil zu 100% erfüllt, und nur dann kommt der Dann-Teil zum tragen, oder nicht. Dies liegt an der binären Struktur der benutzten Logik: es gibt nur die Wahrheitswerte *wahr* oder *falsch*. Die Idee von Fuzzy-Expertensystemen ist dagegen, dass der Wenn-Teil einer Regel z.B. nur zu 60% erfüllt sein braucht und daraus dann trotzdem ein sinnvoller Dann-Teil abgeleitet werden kann. Es gibt also nicht nur die Wahrheitswerte *wahr* und *falsch*, sondern auch „Zwischenwerte“ davon. Um solche unscharfen Regeln formulieren zu können, bedarf es einer Verallgemeinerung der binären Logik.
Im klassischen Fall gilt für beliebige Objekte x einer Objektmenge X und einer Teilmenge $A \subseteq X$ entweder $x \in A$ oder $x \notin A$. Das heißt mit anderen Worten, der Umstand, dass x ein Element von A ist, ist entweder wahr oder falsch. Man könnte auch die Mitgliedschaft von x zu der Menge A durch *0* (kein Mitglied) oder *1* (Mitglied) ausdrücken, also die „Zugehörigkeit“ durch ein Element der binärer Menge {0,1} ausdrücken.
Bei den nachfolgend näher definierten Fuzzy-Mengen wird dieses „Sein oder Nichtsein“ auf ein *Intervall* [0,1] ausgedehnt, d.h. es kann mehr oder weniger Mitgliedschaft geben. Genauer:

Definition 6.1 (**Fuzzy-Menge**)
Sei X eine Sammlung von Objekten. Unter einer *Fuzzy-Menge* A verstehen wir eine Menge geordneter Paare der Form

$$A = \left\{ \left(x, \mu_A(x) \right) \middle| x \in X \right\}$$

wobei $\mu_A(x)$ Zugehörigkeitsfunktion oder auch Zugehörigkeitsgrad von x in A genannt wird.

Die Funktion $\mu_A(x)$ stellt eine Abbildung in eine positive reelle Menge M dar. Meistens wird $M=[0,1]$ gesetzt. Wenn $M=\{0,1\}$ ist, so haben wir den klassischen nicht-Fuzzy-Fall, wie vorher beschrieben.

Beispiele:
Ein Makler möchte seine Häuser, welche er seinen Klienten anbieten will, klassifizieren. Ein Maßstab für den Komfort eines Hauses sei die Anzahl der Zimmer. Sei $X=\{1,2,...,10\}$ die Menge der verfügbaren Typen, wobei die Elemente

$x \in X$ die Anzahl der Zimmer eines Hauses repräsentieren sollen. Dann könnte man die Fuzzy-Menge

„Komfortabler Haustyp für einen 4-Personen-Haushalt“

wie folgt beschreiben:

$$A=\{(1,0.2),(2,0.5),(3,0.8),(4,1.0),(5,0.7),(6,0.3)\}.$$

Die Werte der Zugehörigkeitsfunktionen könnte der Makler aus Erfahrung ermittelt haben. Man kann Fuzzy-Mengen aber auch stetig definieren, wie folgendes Beispiel zeigt. Wir betrachten eine Aussage der Form:

„A sei die Menge reeller Zahlen wesentlich größer als 10“

Dann kann man definieren: $A = \{(x, \mu_A(x)) | x \in X\}$, wobei z.B.

$$\mu_A(x) = \begin{cases} 0, & x \leq 10 \\ 1 - \dfrac{1}{1+(x-10)^2}, & x > 10 \end{cases}$$

Umgekehrt könnte man auch eine Fuzzy-Menge suchen, bei der gilt:

„A sei die Menge reeller Zahlen in der Nähe von 10“

also z.B.: $$A = \left\{ (x, \mu_A(x)) \middle| \mu_A(x) = \frac{1}{1+(x-10)^2} \right\}$$

Wir betrachten einmal eine grafische Darstellung der Zugehörigkeitsfunktion für das letzte Beispiel (Abb. 6.1).

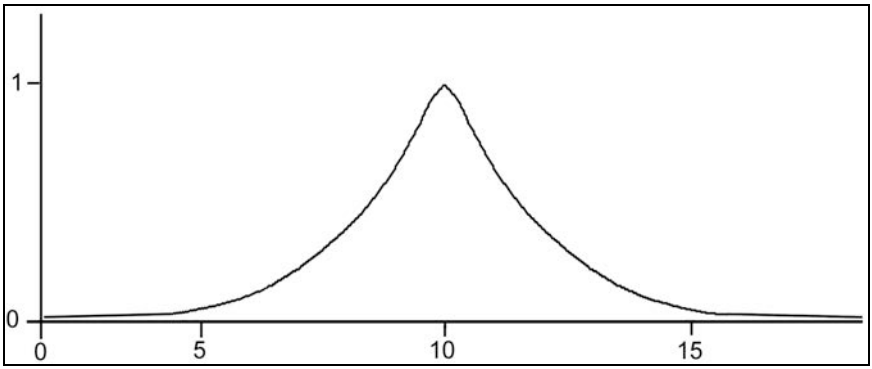


Abb. 6.1 Zugehörigkeitsfunktion: Menge aller reellen Zahlen in der Nähe von 10

Wie aus der Grafik in Abb. 6.1 zu sehen, kann der Umstand, dass sich x in der Nähe von 10 befindet, mehr oder weniger wahr sein. Aussagen wie *„reelle Zahlen in der Nähe von 10“* nennt man auch linguistische Variable. Offensichtlich spielt die Zugehörigkeitsfunktion in der Fuzzy-Logik eine entscheidende Rolle. Wie bei jeder Logik sind nun auch hier logische Verknüpfungen wie *und, oder* etc. zu definieren. Diese Verknüpfungen beziehen sich auf die Zugehörigkeitsfunktionen.

Definition 6.2 (**Fuzzy-Und-Verknüpfung**)
Seien $A \subseteq X$ und $B \subseteq X$. Dann definieren wir die *Konjunktion* der Zugehörigkeitsfunktionen $\mu_A(x)$ und $\mu_B(x)$ durch:
$\mu_A(x) \cap \mu_B(x) := \min\{\mu_A(x), \mu_B(x)\}, x \in X$.

Definition 6.3 (**Fuzzy-Oder-Verknüpfung**)
Seien $A \subseteq X$ und $B \subseteq X$. Dann definieren wir die *Disjunktion* der Zugehörigkeitsfunktionen $\mu_A(x)$ und $\mu_B(x)$ durch
$\mu_A(x) \cup \mu_B(x) := \max\{\mu_A(x), \mu_B(x)\}, x \in X$

Definition 6.4 (**Fuzzy-Nicht-Verknüpfung**)
Sei $A \subseteq X$. Dann definieren wir als *Komplement* bzw. *Negation* der Zugehörigkeitsfunktion $\mu_A(x)$ die Größe $1 - \mu_A(x)$.

Man sieht, dass die Fuzzy-Konjunktion, -Disjunktion und –Negation jeweils für die Grenzfälle $\mu(x) \in \{0,1\}$ die klassischen *und-*, *oder-* und *nicht*-Verknüpfungen der Aussagenlogik darstellen. Es gibt noch eine Reihe weiterer Fuzzy-Operatoren, auf die wir hier aber nicht eingehen. Unsere Definitionen gehen auf Zadeh[5] zurück.

Um nun ein Fuzzy-System zu entwickeln, muss dieses durch ein Regelwerk von linguistischen Variablen beschrieben werden. Dazu ist es zunächst notwendig, diese linguistischen Variablen durch ihre Fuzzy-Menge (d.h. insbesondere durch ihre Zugehörigkeitsfunktion) zu definieren. Die Regelwerke selbst sind dann durch (ggf. mehrere) *wenn...dann*-Beziehungen (wobei im *wenn*-Teil auch *und-* bzw. *oder-* und *nicht*-Verknüpfungen stehen können) beschreibbar. Die Realisierung des *dann*-Teils (Inferenz) führt wieder zu einer Fuzzy-Menge. Die Inferenz bei Fuzzy-Systemen kann auf verschiedene Art und Weise definiert werden. Wie auch in der klassischen Logik besitzt eine Wenn-Dann-Regel die Form *wenn* A *dann* B, d.h. $A \rightarrow B$.

In der Fuzzy-Logik hat man entsprechende Regeln für die Zugehörigkeitsfunktionen der Form *Wenn* $\Psi(x)$ *dann* $\Phi(y)$,d.h. $\Psi(x) \rightarrow \Phi(y)$, wobei $\Psi(x)$ und $\Phi(y)$ bekannte Zugehörigkeitsfunktionen zweier (oder der selben) Fuzzy-Mengen *X* und *Y* sind. $\Psi(x)$ kann dabei selbst eine durch Fuzzy-Operatoren zusammengesetzte Verknüpfung von Zugehörigkeitsfunktionen sein. Die Voraussetzung $\Psi(x)$ beschreibt dabei, zu welchem Grad der *Wenn*-Teil der Regel erfüllt ist, d.h. $\Psi(x)$ ist für ein konkretes $x = x_0 \in X$ zahlenmäßig bekannt. Die Schlussfolgerung $\Phi(y)$ fragt dann i.d.R. nach einem konkreten $y \in Y$, welches zu bestimmen ist. Die Ergebnismenge einer solchen Inferenz ist eine Fuzzy-Menge $\tilde{Y} \subseteq Y$, welche eine aus $\Psi(x)$ und $\Phi(y)$ zusammengesetzte Zugehörigkeitsfunktion $\tilde{\Phi}(y)$ besitzt, die *Konklusionsfunktion* genannt wird. Für diese wird häufig einfach das Produkt benutzt:

Definition 6.5 (**Fuzzy-Produkt-Inferenz***)*
Sei $x_0 \in X$ ein konkreter Wert und damit $\Psi(x_0) \in [0,1]$ ebenfalls zahlenmäßig bekannt. Für eine Zustandsfunktion $\Phi(y) \in Y$, gelte $\Psi(x_0) \rightarrow \Phi(y)$.
Die Konklusionsfunktion errechnet sich dann zu:
$\tilde{\Phi}(y) = \Psi(x_0) \cdot \Phi(y)$.

[5] Zadeh, L.A.: *Fuzzy Sets* in: Information and Control 8, S. 338-353, 1965

Man kann anstatt des Produkts auch eine Inferenz-Regel festlegen, welche die Zielfunktion $\Phi(y)$ in der Höhe des Erfüllungsgrades der Voraussetzung "klippt":

Definition 6.6 (**Fuzzy-Min-Inferenz***)*
Sei $x_0 \in X$ ein konkreter Wert und damit $\Psi(x_0) \in [0,1]$ ebenfalls bekannt. Für eine Zustandsfunktion $\Phi(y) \in Y$, gelte $\Psi(x_0) \rightarrow \Phi(y)$. Die Konklusionsfunktion errechnet sich dann zu: $Min(\Psi(x_0), \Phi(y))$

Welche der beiden Methoden zu bevorzugen ist, hängt vom aktuellen Problem ab. In der Praxis ergibt sich meistens jedoch kein signifikanter Unterschied für beide Methoden. Falls ein Fuzzy-System aus mehreren Regeln besteht, so werden für jede Regel, deren *dann*-Teile sich auf die gleiche linguistische Variable beziehen, die entstehenden Flächen zu einer Gesamtfläche überlagert. Die Beschreibung eines Regelwerks durch Fuzzy-Operationen nennt man auch *Fuzzyfizierung*.
Die aufgrund der Regeln entstandenen Flächen müssen in der Praxis nun wieder einem konkreten *x*-Wert zugeordnet werden, der die tatsächliche Reaktion des Systems vorschreibt (z.B. die Stellung eines Ventils als Ergebnis bestimmter Umgebungsbedingungen). Um dies zu leisten, muss eine Abbildung der aus den Regeln erzeugten Gesamtfläche auf eine reelle Zahl gefunden werden. Diesen Vorgang nennt man *Defuzzyfizierung*. Eine Möglich dafür besteht darin, den Schwerpunkt der Fläche aufzusuchen. Allgemein versteht man dann unter Defuzzyfizierung eine Abbildung, welche der Fläche *Z* der unscharfen Ergebnismenge der Regelbedingung die Abszissenkoordinate X_s des Schwerpunktes zuordnet. Dies kann numerisch erfolgen oder, wenn die entsprechenden mathematischen Voraussetzungen (Integrierbarkeit) gegeben sind, z.B. berechnet werden durch

$$X_S = \frac{\iint x dA}{\iint dA}$$

Der Flächenschwerpunkt ist deswegen als Defuzzyfizierung so geeignet, weil er sozusagen die Fläche am ausgewogensten einer reellen Zahl zuordnet: Würde man die Fläche ausschneiden und auf einem Bleistift zum Balancieren bringen, so liegt dieser Punkt gerade beim Schwerpunkt der Fläche. Zusammenfassend kann man also das Vorgehen zum Erstellen eines Fuzzy-Expertensystems wie folgt beschrieben:

Fuzzyfizierung
Hier werden die Zugehörigkeitsfunktionen, welche über die Eingabevariablen definiert wurden, auf die konkreten Werte angewendet und daraus die "Wahrheitswerte" zwischen *0* und *1* für die Prämissen jeder Fuzzy-Inferenzregel berechnet.

Fuzzy-Inferenz
Hier werden die Zugehörigkeitsfunktionen des "Dann-Teils" durch eine der oben definierten Inferenz-Regeln auf die Werte der Prämissen eingeschränkt (durch Klipping oder Produktbildung); die Ergebnisse sind die *Konklusionsfunktionen.*

Komposition
Existieren mehrere Fuzzy-Regeln für gleiche Zustandsfunktionen im "Dann-Teil" des Regelsystems, so werden all diese Konklusionsfunktionen zusammengefasst. Auch hierfür gibt es mehrere Möglichkeiten: Man kann z.B. alle beteiligten Konklusionsfunktionen einfach addieren (*Summenkomposition*) oder man kann das Maximum (*Maximumkomposition*) bilden.

Defuzzyfizierung
Hier wird die Fuzzy-Menge zu einem konkreten Ausgabewert reduziert. Dies kann z.B. durch Schwerpunktbildung dieser Menge erfolgen.

Beispiel:
Betrachten wir das Ganze an einem konkreten Beispiel, welches 1991 von C. v. Altrock[6] vorgestellt wurde. Es soll ein System entworfen werden, welches die Methanzufuhr in einer Brennkammer regelt. Die Methanzufuhr wird über ein Ventil gesteuert und die Ventilstellung sei abhängig vom Vorkammerdruck und der Brennkammertemperatur. Fuzzysysteme nutzen nun Expertenwissen, welches aus Erfahrung gewonnen wurde. Eine solche aus Erfahrung gewonnene Regel könnte folgendermaßen lauten:

Wenn die Temperatur in der Brennkammer sehr hoch ist und der Vorkammerdruck zumindest über dem Normalwert liegt, dann sollte die Methanzufuhr gedrosselt werden.

Dabei sind Begriffe wie *Temperatur* und *Vorkammerdruck* linguistische Variablen, und Begriffe wie *sehr hoch* oder *Normalwert* die Zugehörigkeitsfunktionen. Grundsätzlich könnte man auch eine Wenn-Dann-Regel für ein klassisches Expertensystem aus ganz konkreten Erfahrungsregeln konstruieren, z.B.:

[6] Altrock, C. v.: *Über den Daumen gepeilt*, in: c't Zeitschrift für Computertechnik, 3/1991

WENN Temperatur ≥ 870 Grad Celcius
UND Vorkammerdruck ≥ 40 bar
DANN Methanventil=0,3 m^3/h

Das Problem bei solchen Regeln ist die harte Grenze zwischen den Zuständen. Eine Temperatur von 870 Grad Celsius ist demnach sehr hoch, aber eine Temperatur von 869,9 Grad Celsius noch normal. Dies zeigt sich z.B. in folgenden beiden Zuständen:

Zustand 1: Temperatur: 871 Grad, Druck 41 bar

Zustand 2: Temperatur: 868 Grad, Druck 62 bar

Der Experte würde erkennen, das der zweite Zustand mit Sicherheit der kritischere ist, aber unsere Regel würde nur beim ersten Zustand greifen. Diese Schwächen eines klassischen Expertensystems können durch ein Fuzzy-System umgangen werden.

Entwurf des Fuzzy-Systems
Zuerst müssen die linguistischen Variablen definiert werden. In unserem Fall sind das *Druck*, *Temperatur* und *Methanventil* (gemeint ist die Reglerstellung des Methanventils). Des Weiteren brauchen wir für jede linguistische Variable noch mehrere Zugehörigkeitsfunktionen. Diese seien für die linguistischen Variablen

Druck: *unter_normal*, *normal* und *über_normal*

Temperatur: *niedrig*, *mittel*, *hoch* und *sehr_hoch*

Methanventil: *gedrosselt*, *halb_offen*, *mittel* und *offen*

Das Zustandekommen der linguistischen Variablen sowie der jeweiligen Zugehörigkeitsfunktionen ist das Ergebnis intensiver Überlegungen und Besprechungen mit dem beteiligten, menschlichen Bediener. In der Praxis ist es so, dass ein Bediener, welcher die entsprechenden Anzeigeinstrumente kontrolliert, abhängig von den abgelesenen Werten die Drosselung oder Öffnung des Methanventils regelt. Dieser Bediener braucht dafür nicht unbedingt die physikalischen Gasgesetze von Boyle-Mariotte oder Gay Lussac zu kennen (mit denen die genauen Werte sicher bestimmbar wären), sondern seine Erfahrung, auf bestimmte Zustände zu reagieren, reicht für eine effiziente Bedienung des Methanventils aus. Dieser Bediener muss daher auch behilflich sein, wenn die Zustandsfunkti-

onen mathematisch modelliert werden. Es gilt genau zu klären, was quantitativ unter Begriffen wie *normal* oder *über_normal* etc. zu verstehen ist. In der Regel wird es sich dabei um Bereiche handeln, d.h. es kann z.B. mehr oder weniger *normal* bzw. *über_normal* geben. Auch können sich die Bereiche natürlich überschneiden, d.h. ein bestimmter Druck könnte gleichzeitig z.B. zu 80% *normal* und zu 12% *über_normal* sein. Und hier liegt die Stärke der Fuzzy-Logik: In klassischen Expertensystemen gibt es z.B. für ein Prädikat *normal* nur die Werte *ja* (100%) oder *nein* (0%), während wir hier jetzt weiche Übergänge haben können. Die Zugehörigkeitsfunktionen können beliebig modelliert werden, jedoch ist es oft ausreichend, diese durch lineare Funktionen darzustellen.

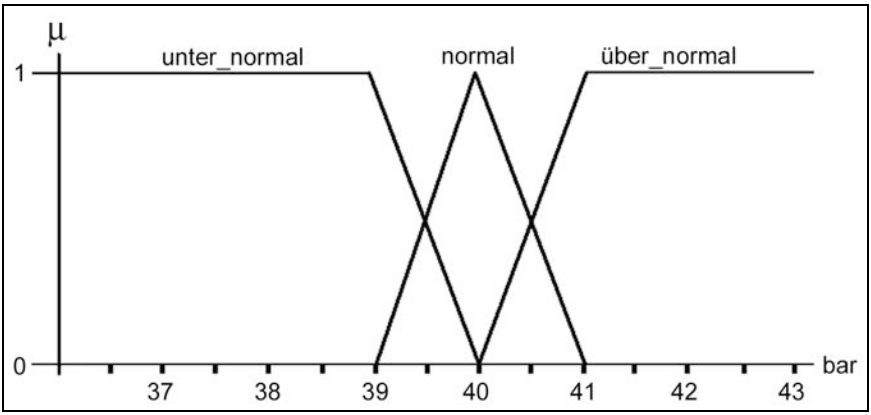


Abb. 6.2 Vorkammerdruck

Abb. 6.2 zeigt die drei Zugehörigkeitsfunktionen des Vorkammerdrucks. Entsprechend lassen sich die Zugehörigkeitsfunktionen der Temperatur modellieren (Abb 6.3).

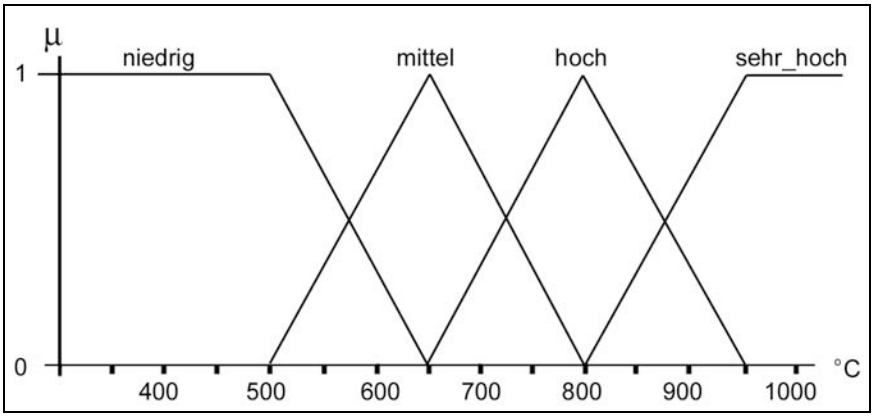


Abb. 6.3 Temperatur

Abb. 6.4 zeigt schließlich die Modellierung der Zugehörigkeitsfunktionen für die Stellung des Methanventils.

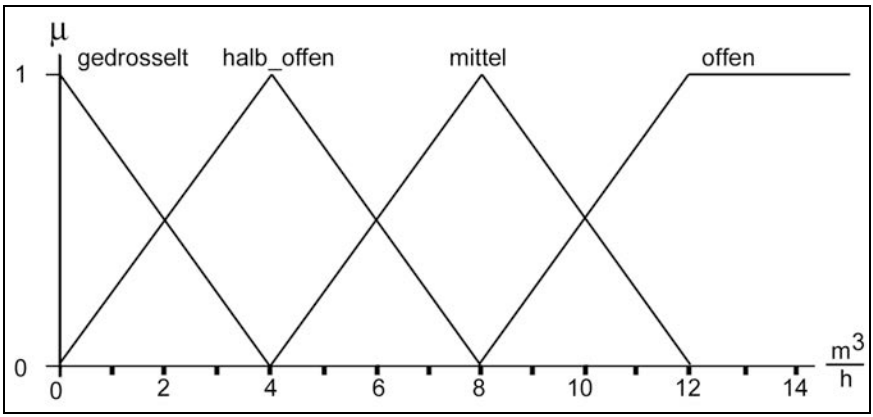


Abb. 6.4 Methanventilstellungen abhängig von der Durchflussrate

Wie man sieht, überschneiden sich die Funktionen. So ist nach Abb. 6.2 z.B. ein Druck von 39,8 bar zu ca. 80% *normal* und zu ca. 20% *unter_normal*.

Nachdem nun alle Zugehörigkeitsfunktionen mathematisch bekannt sind, müssen die entsprechenden linguistischen Regeln formuliert werden. Der Einfachheit halber beschränken wir uns hier auf zwei Regeln:

Regel 1:

WENN Temperatur=sehr_hoch
ODER Vorkammerdruck=über_normal
DANN Methanventil=gedrosselt

Regel 2:

WENN Temperatur=hoch
UND Vorkammerdruck=normal
DANN Methanventil=halb_offen

Der große Vorteil dieser Regeln besteht darin, dass hier die linguistischen Variablen mit ihren Zustandsfunktionen in Verbindung gebracht werden, wobei keine Zahlen in den Regeln auftauchen. Dennoch gibt es unendlich viele Möglichkeiten für die Erfüllung dieser Regeln, da ja z.B. der erste Teil der Wenn-Bedingung der ersten Regel *Temperatur=sehr_hoch* nicht wie bei klassischen Expertensystemen nur die Werte *ja* oder *nein* annehmen kann, sondern alle denkbaren Werte aus dem Intervall [0,1]. Die Disjunktion der beiden Teile der Wenn-Bedingung der Regeln sowie die Dann-Schlussfolgerung geschehen nach den durch Def. 6.2 bis 6.6 angegebenen Fuzzy-Verknüpfungen.
Die Funktion des Systems wird an einem Beispiel demonstriert. Es sei folgender Zustand gegeben:

Temperatur= 910 Grad, Vorkammerdruck=40,5 bar.

Die Werte der Zugehörigkeitsfunktionen der Wenn-Teile der Regeln können aus den Grafiken abgelesen werden:

Temperatur 910 Grad:

sehr_hoch	0,8
hoch	0,3
mittel	0,0
niedrig	0,0

Druck 40,5 bar:

unter_normal	0,0
normal	0,5
über_normal	0,5

Verbal ausgedrückt bedeutet das, dass eine Temperatur von 910 Grad eher *sehr_hoch* und kaum noch *hoch* ist, und ein Druck von 40,5 bar zwischen *normal* und *über_normal* liegt.

Für die Wenn-Teile der Regeln ergibt sich mit unseren Fuzzy-Definitionen von *und* und *oder*:

Regel 1: *Temperatur=sehr_hoch=0,8*
ODER
Vorkammerdruck = über_normal = 0,5

entspricht: *max(0,8;0,5)=0,8*

Regel 2: *Temperatur=hoch=0,3*
UND
Vorkammerdruck=normal=0,5

entspricht: *min(0,3;0,5)=0,3*

Bei der Schlussfolgerung wird davon ausgegangen, dass der Dann-Teil einer Regel in dem Maße erfüllt ist, wie seine Vorbedingungen. Das Methanventil müsste demnach zu einem Grad von 0,8 gedrosselt und zu einem Grad von 0,3 halb offen sein. Um den tatsächlichen Wert zu erhalten, muss entweder die Produkt- oder die Min-Inferenz durchgeführt werden. Wir entscheiden uns für die Min-Inferenz.

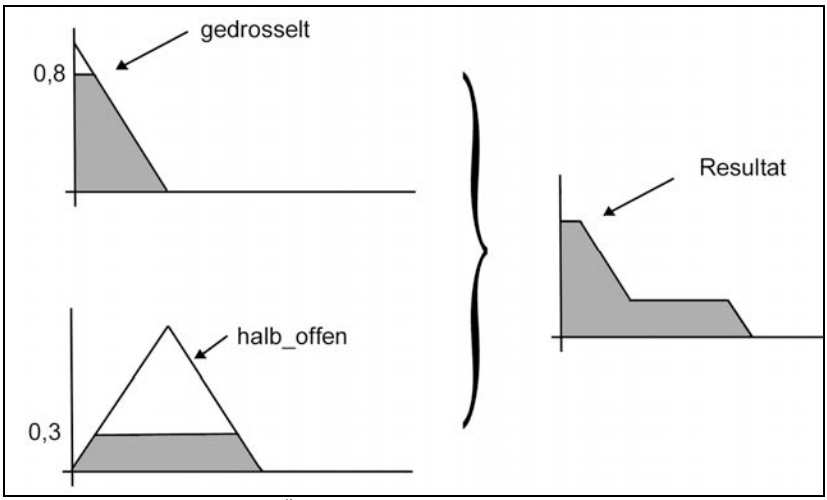


Abb. 6.5 Min-Inferenz und Überlagerung der Konklusionsfunktionen

In Abb. 6.5 sind die Zugehörigkeitsfunktionen des Methanventils aufgrund der Min-Inferenz gemäß des Erfüllungsgrades ihrer Voraussetzungen jeweils gekappt worden (vgl. Def. 6.6). Anschließend bedienten wir uns der erwähnten *Maximumkomposition*, um die resultierende Konklusionsfunktion (die Überlagerung) zu erhalten.

Jetzt beginnt die Defuzzyfizierung, denn es ist ja ein konkreter Wert für die Stellung des Methanventils zu bestimmen. Wie bereits erwähnt, kann man sich dafür der Bestimmung des Flächenschwerpunkts bedienen (Abb. 6.6).

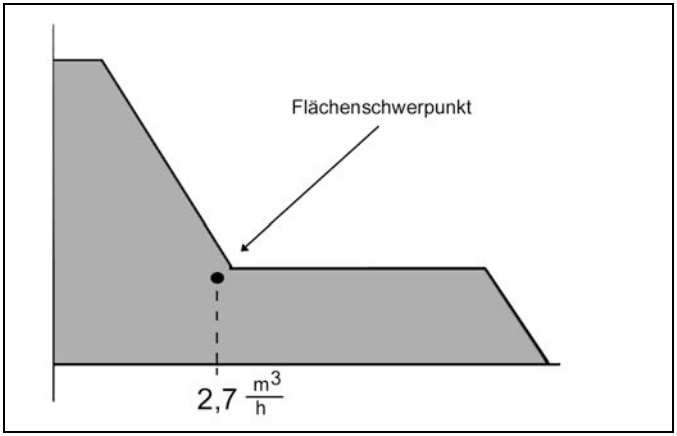
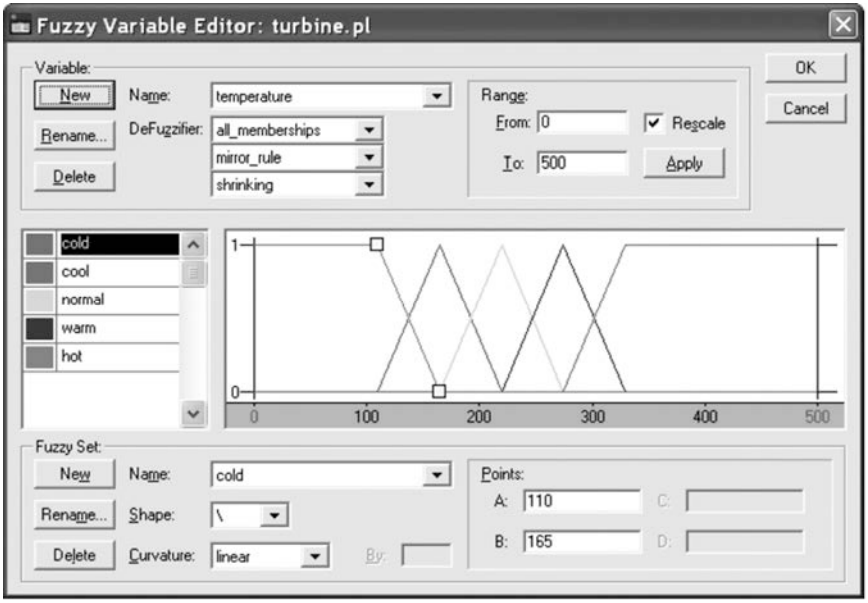


Abb. 6.6 Defuzzyfizierung durch Ermitteln des Flächenschwerpunkts

Zusammenfassend stellen wir fest, dass also tatsächlich ein ganz konkreter Wert (2,7 Kubikmeter pro Stunde) für das Methanventil ermittelt werden konnte, und das ganz ohne mathematische Anwendung irgendwelcher physikalischen Gasgesetze, sondern nur aufgrund des Fuzzy-Regelsystems.
Daraus ergeben sich viele andere praktische Anwendungen wie z.B. das optimierte ruckelfreie Anfahren und Bremsen der U-Bahn von Sendai in Japan oder das Herstellen wackelfreier Videokameras. Im Rahmen dieses Buches konnte natürlich nur das prinzipielle Entwickeln eines Fuzzy-Systems aufgezeigt werden. Darüber hinaus gibt es viele weitere Möglichkeiten, welche der entsprechenden Fachliteratur zu entnehmen sind.
Ich möchte zum Abschluss dieses Kapitels noch einmal das Profi-Expertensystem *Flint*® der englischen Firma LPA heranziehen, welches wir bereits im letzten Kapitel beispielhaft demonstrierten. Dieses System kann nämlich auch Fuzzy-Expertensysteme entwickeln. Es bieten sich dem Entwickler hierfür verschiedene Werkzeuge an. Dazu gehören neben der einfachen Model-

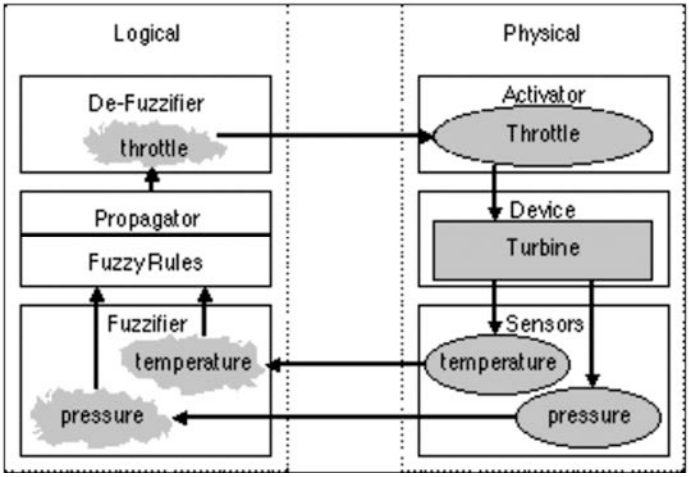
lierung von linearen Zugehörigkeitsfunktionen auch leicht zu bedienende Kontroll-Mechanismen. Nachfolgend ein Bildschirmausdruck für die Modellierung der linearen Zugehörigkeitsfunktionen[7]:



Dieses Beispiel ähnelt unserem vorherigen (zumindest in der Modellierung) und man sieht, wie man den definierten linguistischen Variablen (im Bild: Temperatur) die jeweiligen Zugehörigkeitsfunktionen zuordnet (im Original-Programm durch verschiedene Farben zugewiesen).

Am Beispiel eines Turbinen-Controllers kann man sehen, wie die „reale“ Welt der simulierten gegenübergestellt ist:

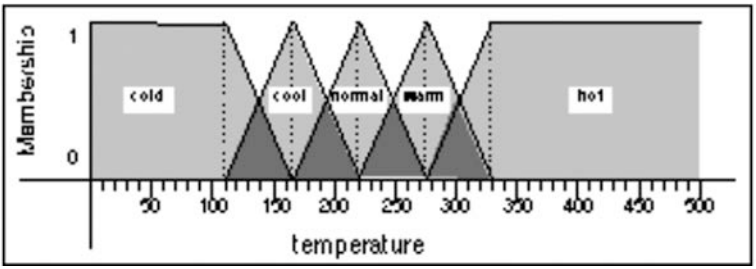
[7] Quellen aller Print-Screens: http://www.lpa.co.uk



Es existieren in beiden „Welten“ dir gleichen Bezeichnungen (wie „throttle“ = Drosselventil, oder Druck und Temperatur), wobei der Propagator hier die Werteveränderung der linguistischen Variablen aufgrund der Parameter berechnet.
Die Definition der Fuzzy-Werte geschieht –wie schon gesehen- entweder grafisch oder auch in Prolog mit Anweisung wie z.B.

```
fuzzy_variable( temperature ) :-
   [ 0 , 500 ] ;
   cold,   \ , linear, [ 110 , 165       ] ;
   cool,   /\, linear, [ 110 , 165 , 220 ] ;
   normal, /\, linear, [ 165 , 220 , 275 ] ;
   warm,   /\, linear, [ 220 , 275 , 330 ] ;
   hot,     /, linear, [       275 , 330 ] .
```

Die entsprechende Zugehörigkeitsfunktion lässt sich in Flint dann direkt anzeigen:



Auch die Regel-Definitionen z.B. für das Drosselventil kann man in einem Regeleditor direkt eingeben (nachdem die entsprechenden linguistischen Variablen wie `cold`, `weak`, `positive_large` etc. definiert wurden):

```
fuzzy_rule(throttle1)
   if temperature is cold
   and pressure is weak
   then throttle is positive_large.

fuzzy_rule(throttle2)
   if temperature is hot
   and pressure is high
   then throttle is negative_large.

fuzzy_rule(throttle3)
   if temperature is normal
   and pressure is ok
   then throttle is zero.
```

Die Regeln lassen sich dann in einer sog. „Fuzzy-Matrix“ übersichtlich darstellen (hier z.B. wieder für das Drosselventil):

```
fuzzy_matrix throttle_value
   temperature  *  pressure  ->  throttle         ;
   cold         *  weak      ->  positive_large   ;
   cold         *  low       ->  positive_medium  ;
   cold         *  ok        ->  positive_small   ;
   cold         *  strong    ->  negative_small   ;
   cold         *  high      ->  negative_medium  ;
   cool         *  weak      ->  positive_large   ;
   cool         *  low       ->  positive_medium  ;
   cool         *  ok        ->  zero             ;
   cool         *  strong    ->  negative_medium  ;
   cool         *  high      ->  negative_medium  ;
   normal       *  weak      ->  positive_medium  ;
   normal       *  low       ->  positive_small   ;
   normal       *  ok        ->  zero             ;
   normal       *  strong    ->  negative_small   ;
   normal       *  high      ->  negative_medium  ;
   warm         *  weak      ->  positive_medium  ;
   warm         *  low       ->  positive_small   ;
   warm         *  ok        ->  negative_small   ;
   warm         *  strong    ->  negative_medium  ;
   warm         *  high      ->  negative_large   ;
   hot          *  weak      ->  positive_small   ;
   hot          *  low       ->  positive_small   ;
```

```
   hot          *  ok        ->  negative_medium  ;
   hot          *  strong    ->  negative_large   ;
   hot          *  high      ->  negative_large   .
```

Die Defuzzyfizierungs-Regeln schließlich können wieder in Prolog angegeben werden durch eine Eingabe wie z.B.

```
find_throttle( Temperature, Pressure, Throttle ) :-
   fuzzy_variable_value( temperature, Temperature ),
   fuzzy_variable_value( pressure,    Pressure    ),
   fuzzy_propagate( minimum, maximum, complement, [t] ),
   fuzzy_variable_value( throttle,    Throttle    ).
```

Sodas am Schluss z.B. ganz konkrete Reglerstellungen in Prolog gefunden werden können:

```
?- find_throttle( 300, 150, Throttle ) .
Throttle = -26.040413058933
```

Im nächsten Kapitel wird in eine weitere wichtige Anwendung der Künstlichen Intelligenz eingeführt: in Neuronale Netze.

Übungen zum Selbsttest:

1. Was ist unter den Begriffen Zugehörigkeitsfunktion, linguistische Variable, Konklusionsfunktion, Fuzzyfizierung und Defuzzyfizierung zu verstehen?

2. Entwickeln Sie eine Zugehörigkeitsfunktion für die Größe eines Menschen unter Zuhilfenahme einer linguistischen Variablen *höhe(x)*. Versuchen Sie die Größe damit als eine stetige Funktion zu definieren. Zeichnen Sie eine Grafik der Zugehörigkeitsfunktion.

7. Neuronale Netze

Der Aufbau neuronaler Netze unterscheidet sich grundlegend von allen anderen Anwendersystemen. Wichtigstes Kennzeichen ist, dass neuronale Netze nicht programmiert, sondern trainiert werden. Ähnlich wie ein Lehrer einem Kind etwas durch häufiges Wiederholen beibringt, werden neuronale Netze mit vorgegebenen Lernmustern trainiert, damit sie später selbständig auf nicht trainierte Situationen sinnvoll reagieren. In diesem Kapitel kann wieder nur auf den prinzipiellen Aufbau neuronaler Netze eingegangen werden, da auch dieses Thema für sich buchfüllend ist. Es werden einige neuronale Netze und deren Modellierung repräsentativ untersucht, so dass der Leser den grundlegenden Aufbau und die Anwendung versteht.
Der Aufbau neuronaler Netze benutzt als Vorbild den Aufbau des menschlichen Gehirns, weshalb wir uns kurz mit demselben beschäftigen wollen.

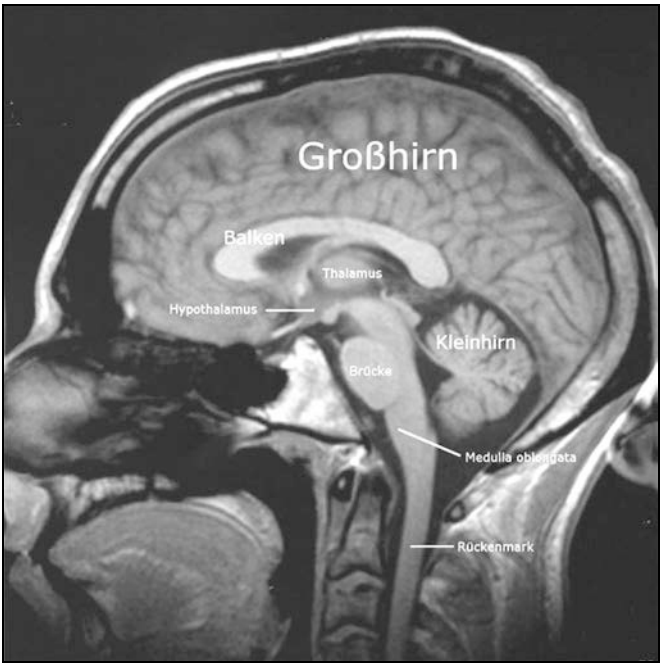


Abb. 7.1 Menschliches Gehirn[8]

[8] Quelle: http://de.wikipedia.org/wiki/Bild:Labeledbrain.jpg

Das menschliche Gehirn besitzt über 100 Milliarden Nervenzellen, welche netzwerkartig miteinander verknüpft sind. Das Kleinhirn verwaltet die motorischen Fähigkeiten, während das verlängerte Rückenmark vegetative Funktionen, wie Atmung, Blutdruck, Verdauung etc. kontrolliert. Das sog. Limbische System ist für das emotionale Verhalten zuständig sowie für Teile des Langzeitgedächtnisses. Zum limbischen System gehören außerdem die Großhirnrinde, der Neocortex, die motorische Rinde, die somato-sensorische Rinde sowie die Sehbahn. Für uns wichtig ist aber der eigentliche Aufbau und die Zusammenhänge der einzelnen Nervenzellen im Gehirn.
Jede Nervenzelle des Gehirns besitzt einen Zellkörper, welcher den Zellkern enthält. Der Zellkörper besitzt viele Verästelungen, von denen eine Axon genannt wird. Das Axon ist der Ausgang der Nervenzelle. Jede Nervenzelle besitzt nur *einen* einzigen Ausgang, welcher sich allerdings am Ende verästeln kann. Die anderen Äste einer Nervenzelle sind die sog. Dendriten, das sind die Eingänge der Nervenzelle; davon hat eine Nervenzelle allerdings sehr viele, bis zu mehreren Tausend. Nun stellt sich die Frage: Was geht überhaupt heraus und hinein bei den Zellen? Das sind schwache elektrische Ströme, die da fließen. Der Ausgang, also das Axon einer Nervenzelle, ist mit den Eingängen, also den Dendriten, anderer (oder auch derselben) Nervenzellen verbunden. Diese Verbindungsstellen, also die Stellen, wo die Axone mit den Dendriten verbunden werden, heißen Synapsen. Dabei handelt es sich um Schaltstellen die festlegen, wie viel des elektrischen Stromes, welcher von dem betreffenden Axon ausgeht, an den jeweiligen Dendriten durchgelassen wird. Das Wort Schaltstelle ist eigentlich nicht ganz richtig, denn es vermittelt den Eindruck, dass hier nur ein- oder ausgeschaltet werden kann. In Wirklichkeit aber gibt es auch alle Zwischenzustände, das heißt, die Synapsen sind eigentlich eher Regler, welche gar nichts, alles oder einen ganz bestimmten Teil des ankommenden elektrischen Stromes durchlassen können. Es kann sogar die Polarität umgekehrt werden (Hemmung). Das Ende eines Axons bei der Synapse ist leicht verdickt uns „schwebt“ über dem Dendrit. Die Weitergabe des elektrischen Stromes geschieht auf elektro-chemischem Wege, in dem chemische Botenstoffe, so genannte Neurotransmitter (wie z.B. Adrenalin), vom Axon-Ende ausgeschüttet werden, welche den synaptischen Spalt überqueren und so in den sog. Ionenkanal der Dendriten gelangen. Dort werden dann diese chemischen Botenstoffe wieder in elektrischen Strom umgewandelt, welcher dann entlang des Dendrits zum zugehörigen Zellkörper weiterfließt (vgl. Abb. 7.2).

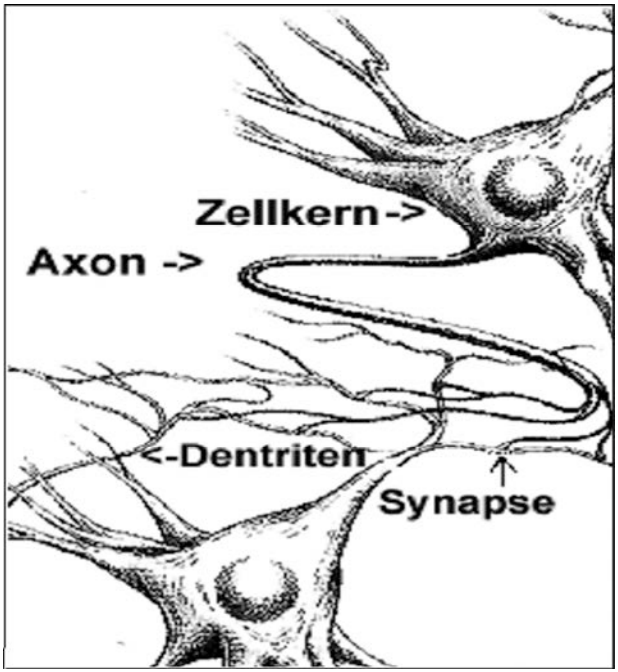


Abb. 7.2 Biologische Neuronen im Gehirn

Doch was passiert nun in der Zelle, wenn elektrischer Strom bei ihr ankommt? Dazu muss man zunächst wissen, dass die Gehirnzellen, wenn sie gerade nicht gereizt werden, eine ständige Oberflächenspannung von ca. -70mV besitzen. Das ist das so genannte Ruhepotential. Kommt jetzt über ein Dendrit einer Zelle weiterer elektrischer Strom an (welcher von einer Synapse durchgelassen wurde), so erhöht sich das Potential an der Zelloberfläche kurzfristig. Alle ankommenden Ströme verschiedener Dendriten der selben Zelle addieren sich, und wenn ein gewisser Schwellwert überschritten wird, dann „zündet“ das Neuron. Damit ist gemeint, dass in diesem Fall die Nervenzelle an ihrem Axon selbst einen kurzen, erhöhten Spannungsimpuls abgibt, welcher weitere Dendriten anderer oder auch der gleichen Zellen speist. Kommt jetzt wieder bei einer Zelle genügend Spannung zusammen, so zündet auch diese und so weiter: Das Gehirn denkt. Mediziner haben nun herausgefunden, dass bei jedem Lernvorgang die synaptischen Regler verstellt werden. In der Tat ist es so (stark vereinfacht), dass unser Wissen durch die Gesamtheit der Kombination aller synaptischer Reglerstellung in unserem Gehirn repräsentiert wird. Neues Wissen lernen heißt also nicht die Zunahme irgend welcher Speicherbelegungen, sondern einfach nur

die Veränderung der Kombination der synaptischen Regler. Damit wird das Gehirn natürlich auch nie „voll“ werden können. Es kann aber sein, dass nach und nach bereits Gelerntes durch die Veränderung der Synapsen wieder unscharf oder sogar ganz vergessen wird, wie wir alle wissen. Diese Art und Weise, Information verteilt in den synaptischen Reglern abzulegen, bildet die Grundlage für künstliche neuronale Netze. Ein künstliches Neuron auf dem Computer wird nach dem biologischen Vorbild modelliert. Dies sei nachfolgend genauer betrachtet.

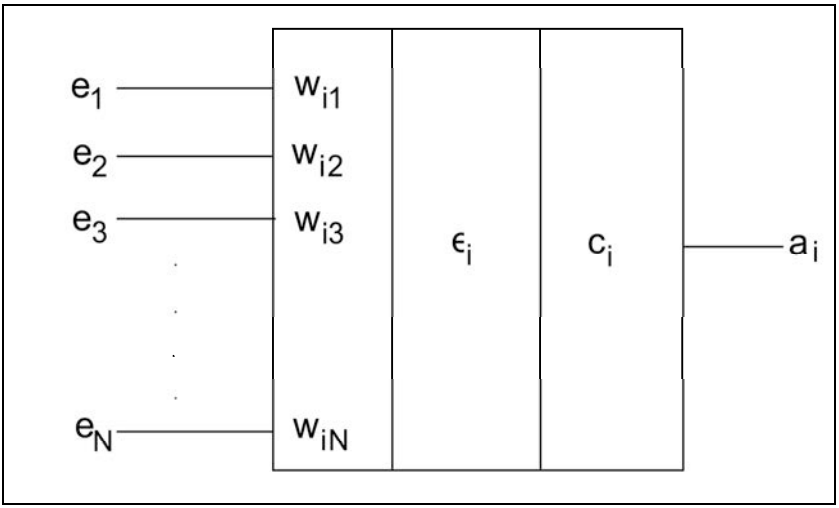


Abb. 7.3 Schema eines künstlichen Neurons

In Abb. 7.3 sehen wir die schematische Darstellung eines Neurons. Der Index *i* identifiziert das Neuron. Wir nehmen hierbei an, dass dieses Neuron zu einer Gruppe von Neuronen gehört, die alle den gleichen Input e_1 bis e_N eingespeist bekommen. Wir folgen hier der Schreibweise von Hoffmann[9]. Im Einzelnen bedeuten:

[9] Hoffmann, N.: Kleines Handbuch Neuronale Netze, Vieweg 1993

Symbol	Bedeutung	Biologische Entsprechung
i	i-tes Neuron	i-te Nervenzellen im Gehirn
j	j-ter Eingang des Neurons	j-tes Dendrit der Nervenzelle
e_j	j-ter Eingangswert des Neurons (i.A. reelle Zahl)	Am j-ten Dendrit ankommende Spannung **vor** der Synapse
w_{ij}	Gewicht des j-ten Eingangs der i-ten Neurons (i.A. reelle Zahl)	Durchlasswert der Synapse, die die ankommende Spannung am j-ten Dendrit der i-ten Nervenzelle regelt
$\varepsilon_i = \varepsilon_i(e_j, w_{ij})$	Effektiver Eingang des i-ten Neurons	Kumulierte Eingangsspannung aller Eingangswerte *nach* den Synapsen
$c_i = c_i(\varepsilon_i)$	Aktivität des i-ten Neurons	Sich einstellendes Membranpotential an der Zelloberfläche
a_i	Ausgangswert	Am Axon ausgegebene Spannung

Neben der Darstellung eines Neurons im Blockschaltbild, wie in Abbildung 7.3 geschehen, findet man in der Literatur auch oft noch die alternative Rezeptoren-Darstellung (Abb. 7.4).

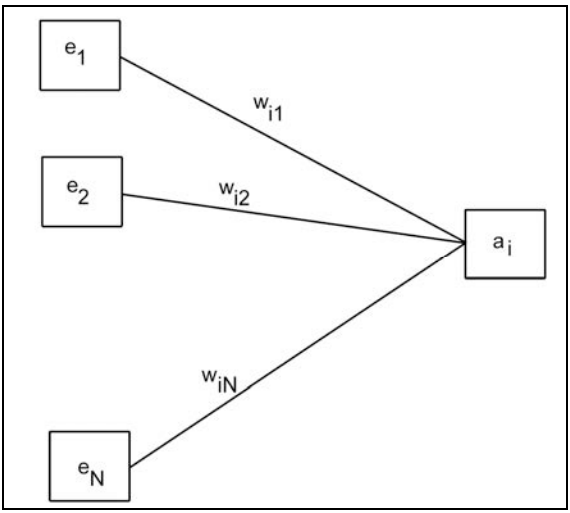


Abb. 7.4 Rezeptorendarstellung eines Neurons

Während also im Blockschaltbild die Kanten die Ein- bzw. Ausgänge und ein Rechteck ein ganzes Neuron darstellen, so sind in der Rezeptorendarstellung Rechtecke jeweils Ein- bzw. Ausgänge, und die Kanten stellen die Gewichte dar. Häufig werden dabei Kanten, die Gewichten $w_{ij} = 0$ entsprechen, ganz weggelassen.
Gelegentlich findet man auch noch die so genannte Hinton-Darstellung (Abb. 7.5):

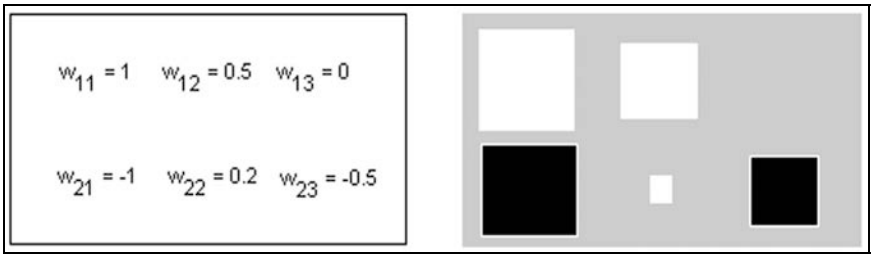


Abb. 7.5 Hinton-Diagramm

Hinton-Diagramme haben den Vorteil, dass man hier die Stärke der das Wissen repräsentierenden Gewichte w_{ij} optisch ablesen kann. Da diese Information jedoch wenig aussagekräftig ist, sind Hinton-Diagramme entsprechend selten anzutreffen.
Die Modellierung der an einem Neuron beteiligten mathematischen Funktionen geschieht auf vielfältige Wiese. So werden verschiedene Neuronentypen unterschieden, die letztlich die verschiedenen mathematischen Ansätze für die beteiligten Funktionen widerspiegeln. Welche mathematische Modellierung dabei zu bevorzugen ist, hängt von dem damit zu lösenden Problemkreis ab. Grundsätzlich wird dabei natürlich versucht, die Funktionen so nah wie möglich an die realen Verhältnisse im menschlichen Gehirn anzupassen. Doch einerseits sind diese (noch) gar nicht vollständig bekannt, und andererseits ist das, was bekannt ist, nur recht aufwändig zu modellieren. Es existieren daher häufig wesentlich einfachere Ansätze, die in der Praxis aber oft ausreichend sind. So seien nachfolgend einige mathematische Funktionen zur Modellierung der beteiligten Komponenten eines künstlichen Neurons angegeben, wobei hier nur eine kleine Auswahl getroffen werden kann. Da wir uns nachfolgend immer nur auf ein einzelnes Neuron beziehen, wird der Neuronen-Index *i* weggelassen.

Berechnung des effektiven Eingangs:
Neuronen erster Ordnung (lineare Neuronen):

$$\varepsilon = \sum_{j=1}^{n} w_j e_j$$

Neuronen höherer Ordnung:

$$\varepsilon = w^0 + \sum_{j=1}^{n} w_j^1 e_j + \sum_{j,k=1}^{n} w_{jk}^2 e_j e_k + \sum_{jkl=1}^{n} w_{jkl}^3 e_j e_k e_l + \ldots$$

Bei linearen Neuronen wird also einfach das Skalarprodukt benutzt, während bei den höheren Neuronen, die allgemein auch als Sigma-Pi-Neuronen bezeichnet werden, alle Kombinationen einfließen.

Berechnung der Aktivität:
Im einfachsten Fall wird die Aktivität einfach als proportional zum effektiven Eingang angesetzt (Lineare Aktivierungsfunktion):

$$c = s\,\varepsilon$$

Wird $s=1$ gesetzt, dann nennt man die Aktivität auch die Identität. Bei der linearen Aktivität ist keine Zeitabhängigkeit vorhanden. Es macht allerdings manchmal Sinn, zeitabhängige Aktivierungsfunktionen zu benutzten. So ein Fall liegt z.B. bei der so genannten BSB-Aktivierungsfunktion (Brain-State-In-The-Box) vor:

$$c(t+1) = c(t) + s\,\varepsilon - d[c(t)-c_0]$$

Dabei bedeuten:

s	Skalierungsfaktor, $s>0$
d	Abklingkonstante (oder Abnahme), $0<d<=1$
c_0	Ruhewert der Aktivität

BSB bildet Nervenzellen genauer nach: Solange die Signale über die Synapsen eintreffen, wächst das Membranpotential kontinuierlich an. Beim Fehlen der Signale nimmt es langsam wieder seinen Ruhezustand ein. Ein weiteres Beispiel

für eine zeitbehaftete Aktivität ist die DMA-Aktivierungsfunktion (Distributed Memory and Amnesia):

$$c(t+1) = \begin{cases} c(t) + s\varepsilon[c(t) - m] - d[c(t) - c_0] & \text{für} \quad \varepsilon < 0 \\ c(t) + s\varepsilon[M - c(t)] - d[c(t) - c_0] & \text{für} \quad \varepsilon \geq 0 \end{cases}$$

Es bedeuten:

s, c_0, d	wie bei BSB
m	Minimum
M	Maximum

Bei BSB kann der Gleichgewichtszustand grundsätzlich beliebig groß werden (hängt von den Parametern ab). DMA begrenzt den Gleichgewichtszustand durch das Intervall *[m,M]*.
Eine sehr einfache zeitabhängige Aktivierungsfunktion stellt die so genannte Hopfield-Aktivität dar:

$$c(t-1) = \begin{cases} m & \text{für} \quad \varepsilon < 0 \\ c(t) & \text{für} \quad \varepsilon = 0 \\ 1 & \text{für} \quad \varepsilon > 0 \end{cases}$$

Hier hängt die Aktivität nur vom Vorzeichen des effektiven Eingangs ab. Falls ε=0, so bleibt die Aktivität unverändert. Je nach Modell ist m=-1 oder m=0.
Neuronen werden häufig nach der benutzten Aktivierungsfunktion benannt; so unterscheidet man also lineare Neuronen, BSB-Neuronen, DMA-Neuronen und Hopfield-Neuronen. Es gibt noch eine Reihe weiterer Neuronentypen.

Berechnung des Ausgangs:
Überschreitet die Aktivität eine bestimmte Schwelle, so "feuert" die Nervenzelle, d.h. am Axon stellt sich ein Ausgangswert *a* als Folge einer Ausgangsfunktion *a(c)* ein. Auch hierfür gibt es diverse Möglichkeiten der Modellierung. Wir wollen hier nur zwei davon betrachten, eine sehr einfache, aber diskrete Funktion sowie eine glatte. Im diskreten Fall leistet die Stufenfunktion gute Dienste:

$$a(c) = \begin{cases} m & \text{falls} \quad c < \vartheta \\ M & \text{falls} \quad c \geq \vartheta \end{cases}$$

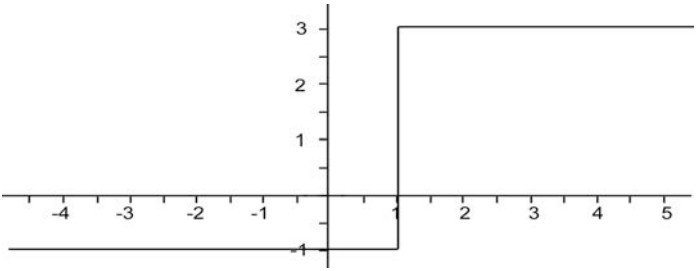


Abb. 7.6 Schwellwertfunktion mit m = -1, M = 3 und $\vartheta = 1$.

Sobald der Schwellwert ϑ von der Aktivität *c* überschritten wird, hüpft der Funktionswert des Ausgangs vom Minimum *m* zum Maximum *M*. Für manche Anwendung erweist sich jedoch die Unstetigkeit der Stufenfunktion als Nachteil. Eine stetige und zudem noch beliebig differenzierbare Variante liefert die allgemeine Fermi-Funktion:

$$a(c) = m + \frac{M - m}{1 + e^{-4\sigma \frac{c - \vartheta}{M - m}}}$$

Der Wertebereich liegt zwischen *[m,M]* und die Schwelle ϑ stellt den Wendepunkt dar. σ ist ein Maß für die Streckung (Steigung). Abb. 7.7 liefert dafür ein Beispiel.

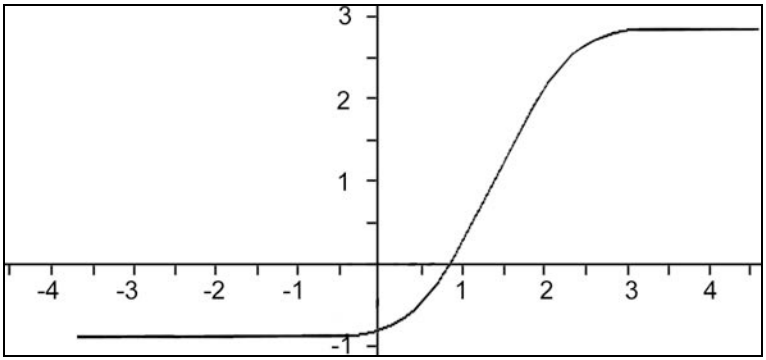


Abb. 7.7 Allgemeine Fermi-Funktion mit m = -1, M = 3, $\vartheta = 1$ und $\sigma = 2$.

Die Aufgabe eines Entwicklers für Neuronale Netze besteht also zunächst einmal darin, eine geeignete Modellierung der benutzten Neuronen vorzunehmen. Dabei bieten sich ihm vielfältige Möglichkeiten, denn es können neben den hier aufgezeigten Funktionen noch viele andere benutzt werden, und diese können untereinander auch noch beliebig kombiniert werden.
Sind die Neuronen einmal modelliert, stellt sich das nächste Problem: wie sollen die Neuronen miteinander verknüpft werden? Eine „totale“ Lösung wäre, eine bestimmte Anzahl Neuronen vorzugeben und einfach jeden Ausgang mit allen Eingängen zu verbinden. So eine völlige Vernetzung existiert tatsächlich, und das neuronale Netz wird dann Hopfield-Netz genannt. Durch geeignetes Setzen der Gewichte kann man damit jede beliebige Netzkonstruktion simulieren; Verbindungen, die nicht vorkommen sollen, kann man durch Setzen der dafür zuständigen Gewichte auf Null erreichen. Der Nachteil aber ist, dass die mathematischen Bedingungen sich exponentiell mit der Anzahl Neuronen und Eingängen vermehren, was in der Praxis das Handling solcher Netze fast unmöglich macht.
Eine andere, sinnvolle Lösung bietet sich an, wenn man der Natur wieder etwas genauer auf die Finger schaut. Hier weist uns ein Phänomen den Weg, das die Mediziner laterale Inhibition (seitliche Hemmung) nennen. Dabei handelt es sich um die bekannte Tatsache, dass das menschliche Gehirn in der Lage ist, die Übergänge von relativ kontrastschwachen Flächen, welche mit dem Auge beobachtet werden, künstlich kontrastreicher zu machen. Wie in Abb. 7.8 zu sehen ist, erscheinen die helleren Flächenteile an den Rändern zu den dunkleren nochmals leicht zusätzlich aufgehellt. Das gleiche geschieht in umgekehrter Richtung: Am Übergang von einer hellen zu einer dunkleren Fläche scheint im dunkleren Teil der Rand zum helleren leicht dunkler. Dies ist jedoch eine optische Täuschung. Das Gehirn rechnet diese zusätzlichen Helligkeitsanteile hinzu, damit die Konturen besser zu erkennen sind.

Abb. 7.8 Scheinbare Kontrasterhöhung an Grauflächenübergängen verschiedener Intensität

Dabei ist es zunächst so, dass die reinen Helligkeitswerte, die auf der Netzhaut unserer Augen ankommen, von einer neuronalen Rezeptorschicht aufgenommen werden (Abb. 7.9).

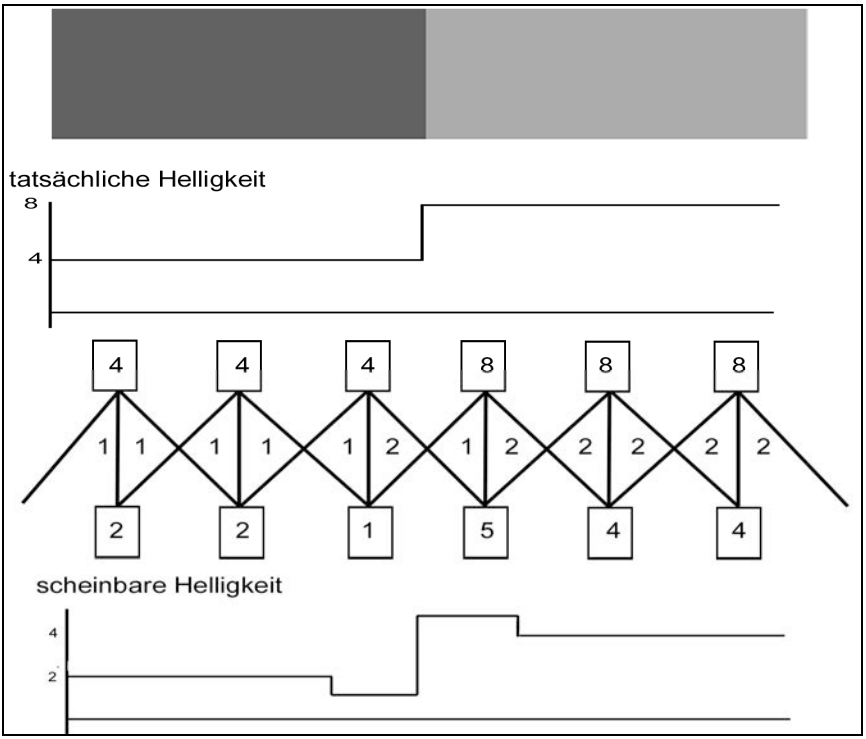


Abb. 7.9 Laterale Inhibition

In Abb. 7.9 sind zwei verschieden helle Grauflächen näher betrachtet. Die tatsächliche Helligkeit ist bis auf den Sprung immer konstant. Der dunkle Teil habe einen Helligkeitswert von 4 Lux, der hellere von 8 Lux. Die erste neuronale Rezeptorschicht bestehe aus 6 Neuronen, welche jeweils die Helligkeitswerte aufnehmen. Die darunter liegende, zweite neuronale Schicht soll die Wahrnehmung in unserem Gehirn repräsentieren. Wie im unteren Teil zu erkennen ist, wurde um den Sprung herum zunächst der dunklere Teil weiter abgesenkt und der hellere weiter angehoben als in den anderen Bereichen. Das Gehirn erreicht dies durch folgenden Trick: Zwischen den beiden Rezeptorschichten herrscht nicht nur eine direkte Verbindung der sich gegenüberliegenden Knoten, sondern es besteht jeweils noch je eine Verbindung mit den rechts und links davon liegenden Knoten. Diese seitlichen Verbindungen hemmen den Durchfluss der direkt gegenüberliegenden Knoten von oben nach unten, und zwar so, dass von

dem oben liegenden Wert die rechts und links vorhandenen Werte subtrahiert werden. Der Wert der jeweiligen Hemmungsverbindung errechnet sich in unserem Beispiel durch Division des aussendenden Knotens (oben) durch die Zahl 4. Dadurch kommt der im unteren Teil in Abb. 7.9 angegebene Helligkeitsverlauf zu Stande.

Für die Praxis lernen wir von der Natur also, dass es sinnvoll ist, nicht einfach wie beim Hopfield-Netz alle Aus- und Eingänge aller Neuronen miteinander zu verbinden, sondern die Neuronen in Schichten anzuordnen. Dadurch ergeben sich erheblich weniger Verbindungen, denn es werden ja nur die Ausgänge der Neuronen einer Schicht mit Eingängen der Neuronen einer anderen Schicht vernetzt. Dies ist also die Motivation der so genannten Neuronenschicht-Modelle.

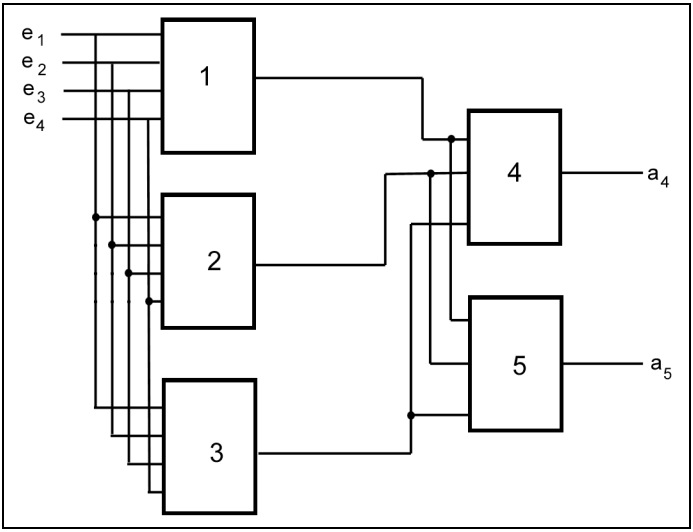


Abb. 7.10 Neuronenschicht-Darstellung

In Abb. 7.10 sind beispielsweise zwei Neuronenschichten vorhanden. Das daraus gebildete Netz besitzt die vier Eingänge e_1, e_2, e_3 und e_4 sowie die zwei Ausgänge a_4 und a_5. Die Ausgänge der Neuronen der ersten Schicht, also a_1, a_2 und a_3 sind mit den Eingängen der Neuronen der zweiten Schicht verbunden (man könnte sie z.B. e_5, e_6 und e_7 nennen). Die Eingänge aller Neuronen einer Neuronenschicht sind miteinander parallel geschaltet. Wie schon bei einzelnen Neuronen gibt es auch für neuronale Netze neben der Blockschaltbild-

Darstellung (wie in Abb. 7.10) noch eine alternative Rezeptorenschicht-Darstellung (Abb. 7.11).

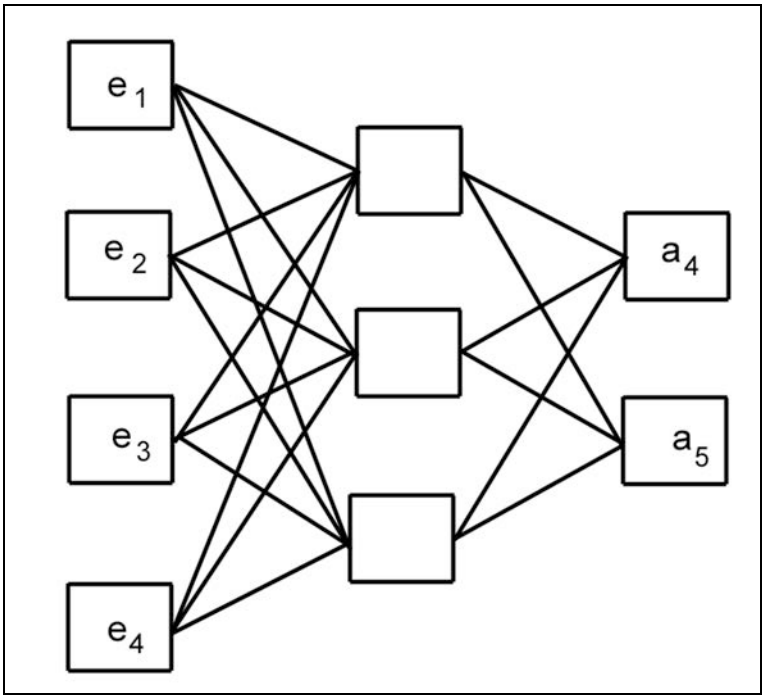


Abb. 7.11 Rezeptorenschicht-Darstellung

In Abb. 7.11 handelt es sich um das gleiche Netz wie in Abb. 7.10. Welche Darstellung man bevorzugt ist Geschmackssache. Es sei nur darauf hingewiesen, dass der Begriff einer „Schicht“ bei den beiden Darstellungen verschiedene Bedeutung hat. In der Blockdarstellung besteht eine Schicht aus ganzen Neuronen, während in der Rezeptorendarstellung eine Schicht jeweils die Ein- oder Ausgänge bezeichnet. Daraus folgt, dass ein Netz, welches in der Blockdarstellung aus *n* (Neuronen-)Schichten besteht, in der Rezeptorenschichtdarstellung immer aus *n+1* (Rezeptoren-)Schichten gebildet ist. In beiden Darstellungen redet man in dem Fall, dass es außer den Eingangs- und den Ausgangsschichten noch innere Schichten gibt, auch von Zwischenschichten.
Es kann auch vorkommen, dass Ausgänge mancher Neuronen wieder mit Eingängen der selben oder Neuronen anderer, davor liegender Schichten verbunden

sind. In so einem Fall redet man von rückgekoppelten Netzen, ansonsten von vorwärtsgekoppelten Netzen.
Man kann Neuronen auch räumlich organisieren, wie das in unserem Gehirn der Fall ist. Dabei kann es sinnvoll sein, nahe beieinander liegende Neuronen stärker zu koppeln aus weiter entfernte. Dies geschieht dann einfach durch Hinzunahme eines Ortsvektors, der den Wert der Gewichte beeinflussen kann.
Hat man schließlich die Topologie der Vernetzung festgelegt, so ist das Netz konfiguriert. In der nächsten Phase wird nun noch festgelegt, welche Lernregeln benutzt werden. Das Lernen ist fest verbunden mit der so genannten Trainingsphase. Die Idee dabei ist folgende: Es werden dem neuronalen Netz Ein- und Ausgabewerte vorgegeben und die Gewichte w_{ij} als variabel betrachtet. Durch geeignete Lernregeln wird nun erreicht, dass die w_{ij} so „justiert“ werden, dass bei Eingabe des trainierten Inputs der zuvor trainierte Output vom Netz richtig berechnet wird. Die Einordnung des Lernvorgangs ist in Abb. 7.12 zu sehen.

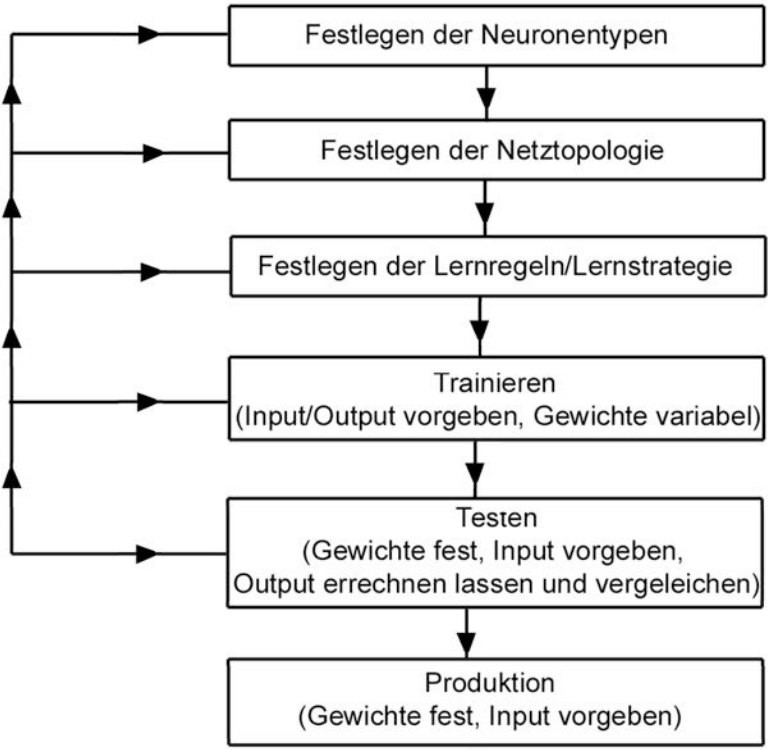


Abb. 7.12 Phasen der Entwicklung neuronaler Netze

Nach den Phasen der Festlegungen der Neuronentypen sowie der Netztopologie ist eine geeignete Lernstrategie zu entwickeln, mit der zusammen die Trainingsphase begonnen werden kann. In der Trainingsphase werden die Gewichte so lange angepasst, bis für alle trainierten Eingaben die zugewiesenen Ausgaben im Rahmen einer vorgegebenen Genauigkeit vom Netz richtig berechnet werden. Ist dies geschehen, so wird in die Testphase eingetreten. Dort werden dem Netz jetzt Eingaben gegeben, deren zugehörige Ausgaben dem Tester bekannt sind, die aber zuvor noch nicht trainiert wurden. Das Netz berechnet seine Ausgaben und es wird dann die Abweichung mit dem tatsächlichen Soll-Output untersucht. Ist auch diese vom Netz berechnete und zuvor nicht trainierte Ausgabe richtig berechnet worden, so gilt die Testphase als abgeschlossen und das Netz kann in die Produktion gehen. Im Prinzip ist dieses Vorgehen das gleiche wie bei der Ausbildung von Menschen in einer Schule: Die Lernphase besteht darin, dass ein Lehrer seinen Schülern einen bestimmten Unterrichtsstoff präsentiert. Dabei stellt der Lehrer einen Zusammenhang zwischen bestimmten Sachverhalten her, meistens in der Form: *Wenn* A *dann* B. A könnte man als Eingabe und B als Ausgabe bezeichnen. Die Synapsen in den Gehirnen der Schüler stellen sich durch diesen Lernprozess (hoffentlich) so ein, dass die Eingabe A mit der Ausgabe B assoziiert wird; diesen Vorgang bezeichnet man auch als Konnektionismus. Der Lehrer wird im Laufe des Unterrichts dabei immer wieder durch Abfragen versuchen festzustellen, ob A und B bei den Schülern auch richtig in Zusammenhang gebracht wurden und dabei ggf. seine Erklärungen wiederholen. Nach Abschluss dieser Lernphase wird der Lehrer in einer Klausur den Lernerfolg testen: Er wird einerseits wieder abfragen, ob das von ihm gelehrte richtig assoziiert wurde, in dem er Fragen stellt wie: Was passiert, wenn A vorliegt? Er wird sich allerdings nicht darauf beschränken, nur das Gelernte abzufragen, sondern er wird auch Dinge fragen, die nicht so zuvor gelehrt wurden, deren Antwort aber aus dem Gelernten vom (guten) Schüler geschlossen werden kann. Der Lehrer gibt in einer Klausur also Eingaben, die so noch nicht trainiert wurden. Der Schüler produziert eine Antwort, und der Lehrer wird diese beim Korrigieren mit einer Musterlösung vergleichen und kann so den Lernerfolg des Schülers testen. Ist dieser Test positiv ausgefallen, kann der Schüler dann die Schule mit Erfolg verlassen und in der „Produktionsphase“ seines Lebens wird darauf vertraut, dass er bisher ungelöste Probleme richtig löst.

Der Lernvorgang besteht aus einzelnen Lernschritten. Bei jedem Lernschritt werden die Gewichte um einen bestimmten Betrag δw_{ij} geändert, also

$$w_{ij}^{neu} = w_{ij}^{alt} + \delta w_{ij}$$

oder in DV-Schreibweise (Wertzuweisung)

$$w_{ij} \rightarrow w_{ij} + \delta w_{ij}\,.$$

Die Zahlenwerte δw_{ij} sind durch die jeweilige "Lernregel" bestimmt. Oft benötigt man sehr viele Lernschritte, bis das Netz das gewünschte Verhalten zeigt. Man fordert im Allgemeinen von einem neuronalen Netz, dass es nach dem Lernen bestimmter Muster weitere Muster lernen kann (Plastizität). Andererseits können dabei Konflikte auftreten derart, dass bereits gelernte Muster wieder verlernt werden (schwache Stabilität). Diese Art von Konflikt bezeichnet man als Stabilitäts-Plastizitäts-Dilemma.

Überwachtes Lernen
Diese Art des Lernens, auch *Lernen mit Lehrer* genannt, trainiert das Netz aufgrund der Vorgabe fester Trainingsmengen in Form von Musterpaaren

$$(E_j^{\mu}, S_i^{\mu}), \mu = 1 \ldots p$$

Es sind hier *p* vorgegebene Input-Output-Assoziationen vorhanden, *E* steht für Eingangswert, *S* für (Ausgangs-) Sollwert. Der Lernvorgang spielt sich dann folgendermaßen ab (pro Lernschritt):

1. Man wählt das m-te Eingangsmuster E_j^m aus, legt es an das Netz an (j-ter Eingang).
2. Das Netz berechnet das tatsächliche Ausgangsmuster A_i^m.
3. Weicht das Ist-Muster A_i^m von dem Sollmuster S_i^m innerhalb einer vorgegebenen Genauigkeit ab, so ändert man die Gewichte so ab (über δw_{ij}), dass die Abweichung kleiner wird (Konvergenz). Wie die Gewichtsänderung δw_{ij} zu wählen ist, hängt von der verwendeten Lernregel ab.

Hebb hat 1949 die Vermutung ausgesprochen, dass dann, wenn zwei verbundene Nervenzellen gleichzeitig feuern, die Verbindungsstärke der sie miteinander verbindenden Synapsen zunimmt. Übertragen auf ein neuronales Netz bedeutet dies, dass das Gewicht w_{ij} eines Neurons sich vergrößert, wenn e_j und a_i gleichzeitig feuern, also positiv sind. Man kann daher versuchsweise eine Proportionalität ansetzen

$$\delta w_{ij} = \eta a_i e_j\,,$$

wobei der Proportionalitätsfaktor η eine positive reelle Zahl ist, Lernrate genannt.
Hier kommen jedoch die *tatsächlichen* Ausgangswerte a_i vor, und nicht die Sollwerte. Man muss daher die Regel entsprechend abändern (Großbuchstaben deuten die Netz-Ein- und Ausgänge an):

$$\delta w_{ij} = \eta S_i E_j .$$

Diese Regel wird Hebb'sche Lernregel genannt. Beachte: Dies kann nur für Netze ohne Zwischenschicht gelten, da nur dort die Eingangssignale der Eingangsschicht $e_j=E_j$ bzw. die Ausgangswerte der Ausgangsschicht $a_i=S_i$ vorhanden sind; die Werte einer evtl. Zwischenschicht fließen nicht in die Lernregel ein, daher lassen sich die Gewichte solcher Zwischenschichten damit auch nicht anpassen. Die Hebb´sche Lernregel besitzt einige interessante Eigenschaften. Initialisiert man die Gewichte $w_{ij} = 0$, so gilt nach Abschluss der Lernphase offenbar

$$w_{ij} = \eta \sum_{\mu=1}^{p} S_i^{\mu} E_j^{\mu} .$$

Man erkennt daraus, dass die Reihenfolge des Lernens keine Rolle spielt. Mehrmaliges Lernen aller Muster vergrößert die Gewichte alle um einen konstanten Faktor, der auch η zugeschlagen werden könnte. Es stellt sich daher keine Verbesserung des Lernerfolges bei mehrmaligem Lernen der Muster ein (es gibt Lernregeln, da ist das anders). Beim Lernen erfolgt auch keine Berechnung der Ausgangswerte (die Eingangs- und Sollwerte werden direkt gesetzt), daher wird auch keine Abweichung von Ist- und Sollwert berücksichtigt. Letzterer Mangel wird durch folgende Erweiterung der Hebb'schen Lernregel behoben:

$$\delta w_{ij} = \eta (S_i - A_i) E_j .$$

Dabei bedeutet *A* der Ist-Wert, *S* der Sollwert und *E* der Eingangswert. Diese Regel wird auch Delta-Lernregel oder Widrow-Hoff-Lernregel genannt. Auch hier sind nur neuronale Netze ohne Zwischenschicht zulässig.
Möchte man nun auch Netze erfolgreich trainieren, die Zwischenschichten besitzen, so bietet sich ein anderer Ansatz an, der auch Lernen durch Lohn und Strafe genannt wird. Er soll hier kurz angedeutet werden.
Es seien wieder zu lernende Musterpaare der Form

$$(E_j^\mu, S_i^\mu), \mu = 1 \ldots p$$

gegeben. Die Ausgangsschicht kann mit der Delta-Lernregel trainiert werden. Die verborgenen Neuronen haben keine Sollwerte; daher wird ein pauschales Fehlersignal definiert der Form:

$$r := \frac{1}{n} \sum_{i=1}^{N_A} (S_i - A_i)^2$$

Dieses Fehlersignal wird den versteckten Neuronen zugeführt. Der Normierungsfaktor *n* ist dabei so zu wählen, dass $r \in [0,1]$ ist. Bei $r=0$ ist der Ausgang korrekt, bei $r=1$ völlig falsch. Die verborgenen Neuronen müssen dann ihre Gewichte so anpassen, dass *r* verringert wird. Dies gelingt mit einer Variante der Hebb´schen Lernregel. Es seien aus Vereinfachungsgründen hier nur solche Netze betrachtet, deren Ausgangswerte lediglich 0 oder 1 sein können. Solche Neuronen nennt man auch McCulloch-Pitts-Neuronen. Der Normierungsfaktor *n* ist dann $n=N_A$ (Anzahl der Neuronen der jeweiligen Schicht). Für die verborgenen Neuronen fordern wir:

1. Der effektive Eingang ist durch das Skalarprodukt gegeben
2. Aktivierungsfunktion ist die einfache Fermi-Funktion, d.h.
   $$c_i = \frac{1}{1 - e^{-\varepsilon_i}}$$ (*i* nummeriert nur die verborgenen Neuronen)
3. Die Ausgangsfunktion ist stochastisch:
   $$P_i(a_i = 1) = c_i$$

Eine stochastische Ausgangsfunktion hat die Eigenschaft, dass die reproduzierten Größen nicht immer einen festen Werte besitzen, sondern nur im statistischen Mittel gegen einen Wert tendieren. Eine häufig verwendete Funktion hierfür ist die Boltzmann-Funktion:

$$P(a_i = 1) = \frac{1}{1 + e^{-(\varepsilon_i - \vartheta_i)/T}}$$

Die Größe T stellt einen vorgebbaren Parameter dar. Um nun die Lernregel zu finden, sei zunächst der Fall $r=0$ betrachtet. Hier sei jetzt die Hebb´sche Lernregel benutzt:

$$\delta w_{ij} = \eta(a_i - c_i)e_j$$

Im korrekten Fall verstärken sich die Synapsen (=Lohn). Im Falle *r=1*, also wenn das Netz falsch reproduziert, sollen sich die Gewichte genau umgekehrt verhalten (Strafe), d.h.

$$\delta w_{ij} = \eta(1 - a_i - c_i)e_j$$

Die beiden Fälle führen dann zu der Lernregel

$$\delta w_{ij} = (1-r)\,\eta(a_i - c_i)e_j + r\eta'(1 - a_i - c_i)e_j$$

Unüberwachtes Lernen
Beim überwachten Lernen war bekannt, welche Ausgangsmuster produziert werden sollten. Oft hat man jedoch nur eine Menge zu analysierender Eingangsmuster, d.h. es soll eine Klassifikation der Muster vom Netz selbst vorgenommen werden. Ist die Klasseneinteilung nicht bekannt, so muss das Netz Ähnlichkeiten bei den Mustern herausfinden und geeignete Klassen selbst festlegen (die Anzahl der Klassen wird aber vorgegeben). Da die Netzausgänge nicht bekannt sind, spricht man hier von unüberwachtem Lernen oder auch Lernen ohne Lehrer. Wir beschränken uns hier allerdings nur auf einen Spezialfall, nämlich das so genannte kompetitive Lernen, auch Wettbewerbslernen genannt. Man geht dabei wieder von der Hebb'schen Lernregel aus:

$$w_{ij} \to w_{ij} + \eta e_j a_i$$

Dies kann aber prinzipiell zu beliebig großen Gewichten führen. Um dies zu vermeiden, kann man diesen Ansatz durch eine geeignete Norm dividieren:

$$w_{ij} \to \frac{w_{ij} + \eta e_j a_i}{\left\| w_{ij} + \eta e_j a_i \right\|}$$

Wie immer ist η>0. Um nicht zu komplizierte Ausdrücke zu erhalten, wie dies z.B. bei der Euklid´schen Norm

$$\left\| \vec{e} \right\| = \sqrt{\sum_k (e_k)^2}$$

der Fall wäre, fordert man folgendes:

1. Die Eingangsvektoren sowie die Gewichtsvektoren seien normiert; benutzt wird allerdings nicht obige Euklid´sche Norm, sondern die einfachere, sog. 1-Norm:

$$\left|\vec{e}\right| = \sum_k \left|e_k\right| = 1 \qquad \text{bzw.} \qquad \left|\vec{w}_i\right| = \sum_k \left|w_{ik}\right| = 1$$

2. Die Eingänge sowie die Gewichte seien nicht-negativ (d.h. die Betragsstriche können weggelassen werden)
3. Die Neuronenausgänge seien wieder binär, d.h. $a_i \in \{0,1\}$.

Unter diesen Voraussetzungen lautet jetzt die Lernregel:

$$w_{ij} \to \frac{w_{ij} + \eta e_j a_i}{\sum_k (w_{ik} + \eta e_k a_i)}$$

Wegen der Normierungsbedingungen gilt offenbar $\sum_k e_k = 1$ und $\sum_k w_{ik} = 1$, so dass

$$w_{ij} \to w_{ij} + \frac{\eta}{1 + \eta a_i}(e_j - w_{ij}) a_i$$

Daraus leitet man ab:

$$\delta w_{ij} = \frac{\eta}{1 + \eta a_i}(e_j - w_{ij}) a_i$$

Mit der Ersetzung

$$\eta \to \frac{\eta}{1 + \eta}$$

wobei berücksichtigt wird, dass a_i nur die Werte *0* oder *1* annehmen kann, erhält man schließlich die endgültige Form der Wettbewerbs-Lernregel:

$$\delta w_{ij} = \eta(e_j - w_{ij})a_i$$

Diese kann man auch schreiben als

$$\delta w_{ij} = \begin{cases} \eta(e_j - w_{ij}) & falls \quad Neuron \quad i \quad aktiv \quad ist \\ 0 & sonst \end{cases}$$

Es lernt also nur dasjenige Neuron, welches bei der Reproduktion den „Wettbewerb" gewonnen hat. Der Lernvorgang lässt die einfache, anschauliche Deutung zu, dass sich der Gewichtsvektor des gewinnenden Neurons zum Eingangsvektor „hindreht". Um dies zu sehen, betrachten wir folgendes:
Für das gewinnende Neuron ist $a_i = 1$. Wegen $0 < \eta < 1$ ist auch $0 < 1 - \eta < 1$ und daher

$$\left|\vec{w}_i - \vec{e}\right| > (1-\eta)\left|\vec{w}_i - \vec{e}\right| = \left|\vec{w}_i + \eta(\vec{e} - \vec{w}_i)a_i - \vec{e}\right|$$

Dies bedeutet also, dass

$$\left|\vec{w}_i^{neu} - \vec{e}\right| < \left|\vec{w}_i - \vec{e}\right|$$

D.h., der neue Gewichtsvektor liegt näher beim Eingangsvektor als der alte. Die eigentliche Klasseneinteilung folgt also durch die vorgegebenen (Ausgangs-) Neuronen (deren Anzahl gleich der Anzahl der möglichen Klassen ist), sie stellen eine Art „Mutterzellen" für die Eingänge dar. Sei z.B. ein Trainingsatz $\vec{E}^{\mu}$ betrachtet, dessen Größe *p* kleiner als die Anzahl *N* der Neuronen ist. Lernt man einen einzelnen Eingangsvektor, so wird das Neuron, dessen Gewichtsvektor ihm am ähnlichsten ist, seinen Gewichtsvektor noch weiter auf den Eingangsvektor zubewegen (das ist das gewinnende Neuron). Am Ende des Lernens wird es zu jedem Eingangsvektor ein ihm zugeordnetes Neuron geben. Legt man jetzt einen nicht gelernten Eingangsvektor an, so wird dasjenige Neuron aktiv werden, dessen Gewichte dem angelegten Vektor am ähnlichsten sind, wobei das Ähnlichkeitskriterium durch den kleinsten vektoriellen Abstand gegeben ist.
Wenn der Trainingsatz ein *p* besitzt, das größer als die (Ausgangs-) Neuronenanzahl *N* ist, dann führt bereits das Training schon zu einer Klasseneinteilung, wobei ähnliche Eingangsvektoren zu einer Klasse zusammengefasst

werden. Jeder Gewichtsvektor kann dann als Prototyp aufgefasst werden und ein nicht gelernter Eingangsvektor wird derjenigen Klasse zugeteilt, deren Prototyp ihm am ähnlichsten ist. Das selbständige Anpassen an die Umgebungsbedingungen beim Wettbewerbslernen wird auch als Selbstorganisation bezeichnet. Auf dieser Eigenschaft basieren Netztypen wie die so genannten selbstorganisierenden Karten.

Es sei noch eine Bemerkung zur Reproduktion, also zur Berechnung der Ausgangswerte in neuronalen Netzen, gemacht. Um das Reproduktionsprinzip besser zu verstehen, betrachten wir einen sehr stark vereinfachten Spezialfall für ein einschichtiges neuronales Netz. Ohne auf die Details weiter einzugehen sei es so beschaffen, dass man eine Matrix-Gleichung der folgenden Form anwenden kann:

$$\vec{A} = W\vec{E},$$

wobei $\vec{E}$ die Netzeingänge, $\vec{A}$ die Netzausgänge und W die Matrix der Gewichte darstellt, genauer:

$$\begin{pmatrix} a_1 \\ \dots \\ a_n \end{pmatrix} = \begin{pmatrix} w_{11} & \dots & w_{1m} \\ \dots & \dots & \dots \\ w_{n1} & \dots & w_{nm} \end{pmatrix} \begin{pmatrix} e_1 \\ \dots \\ e_m \end{pmatrix}$$

In der Lernphase werden A und E vorgegeben und die Elemente der Matrix W sind variabel. Ist das neuronale Netz großzügig genug ausgelegt, so ist das sich dabei ergebende Gleichungssystem stark unterbestimmt, d.h. es gibt viele mögliche w_{ij}, die bei Multiplikation der Matrix W mit dem Eingabevektor E den Ausgabevektor A erzeugen. Dies ist auch wichtig, denn es soll ja nicht nur ein Muster (E,A) trainiert werden. Wichtig ist also auch noch, dass bei Anlegen neuer Muster jetzt eine Lösung in den Variablen w_{ij} gefunden wird, die sowohl das neue als auch das alte Muster richtig reproduziert. Solange die Matrix W groß genug ist und damit genügend viele Lösungsmöglichkeiten existieren, kann dieses Verhalten durch geeignete Lernregeln immer erzwungen werden, es kann also eine gewisse Konvergenz erreicht werden. Werden relativ zur Matrixgröße zu viele Muster trainiert, können alte Muster nach und nach „vergessen“ bzw. nur noch unvollständig reproduziert werden, und die neuen Muster werden auch nicht mehr richtig gelernt. Später, in der Produktionsphase, liegen dann die Matrixelemente (durch das vorangegangene Training) fest und es werden nur noch Ausgänge A zu Eingängen E berechnet (Reproduktion).

Neuronale Netze, die sich durch Matrixgleichungen der oben angegebenen Form beschreiben lassen, nennt man auch Musterassoziatoren. Wenn die Eingangsmuster linear unabhängig sind, konvergiert die Anwendung der Delta-Lernregel zu einer exakten Lösung der Lernaufgabe. Um die Wichtigkeit dieser Aussage zu beschreiben, sei hier das sog. XOR-Problem kurz diskutiert.

Nehmen wir einmal an, es seien die folgenden 4 Musterpaare $(\vec{E}^1, S^1)...(\vec{E}^4, S^4)$ in einem neuronalen Netz zu trainieren:

Musterpaar	E_1	E_2	S
1	0	0	0
2	1	0	1
3	0	1	1
4	1	1	0

Die Eingangsmuster sind nun aber linear abhängig, d.h. es kann keine Konvergenz der Delta-Lernregel garantiert werden. Ein möglicher Musterassoziator könnte aus einem Neuron mit zwei Eingängen bestehen. Wir verwenden die McCulloch-Pitts-Neuronen mit einer Schwelle. Dann gilt mit der einfachen Stufenfunktion $\Theta(x)=0$ falls $x< 0$ und $\Theta(x)=1$ falls $x\geq 0$:

$$A = \Theta(w_1 E_1 + w_2 E_2 - \vartheta)$$

Setzt man die hier zu lernenden Muster ein, so erhält man vier Gleichungen, hat aber nur die drei Parameter w_1, w_2 und Theta als Unbekannte zur Verfügung (d.h. die Schwelle wird auch als variabel betrachtet). Dies führt zu einem widersprüchlichen Gleichungssystem in diesen Parametern. Das lässt sich wie folgt veranschaulichen:

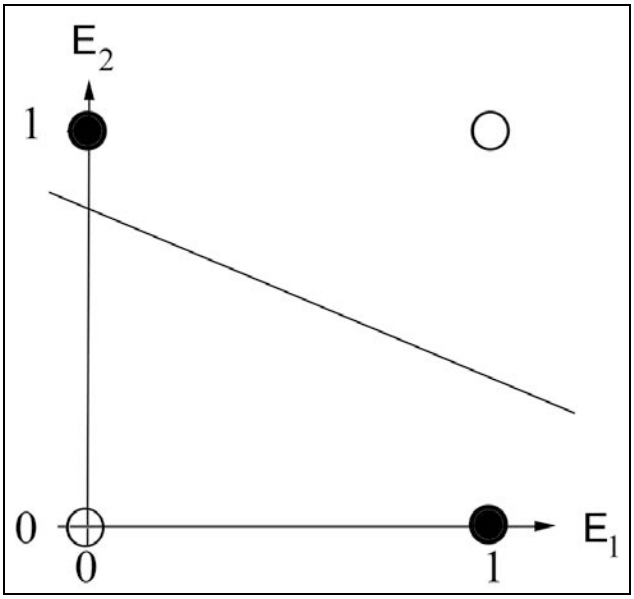


Abb. 7.13 Trennung der Eingangsgrößen durch eine Gerade nicht möglich

Zu den ausgefüllten Kreisen in Abb. 7.13 gehört der Sollwert 1, zu den offenen 0. Es ist außerdem eine mögliche Schwellwert-Gerade eingezeichnet. Die zweikomponentigen Eingangsvektoren kann man als Punkte in der E_1E_2-Ebene darstellen. Die Ausgangsfunktion hat ihren Sprung bei $\theta = 0$, also bei $w_1E_1 + w_2E_2 - \vartheta = 0$. Dabei handelt es sich offensichtlich um eine Geradengleichung. Alle Eingangsmuster oberhalb der Gerade liefern den Ausgang "1", alle anderen den Ausgang "0". Die Gerade trennt also die Ebene in zwei Bereiche mit unterschiedlichem Netzausgang. Anschaulich sieht man, dass es keine Gerade gibt, welche die Ebene so trennt, dass auf der einen Seite nur die offenen, auf der anderen nur die ausgefüllten Kreise liegen. Solche Probleme nennt man *nicht linear separabel* oder *nicht linear trennbar*. Ein etwas allgemeineres Beispiel ist das folgende:

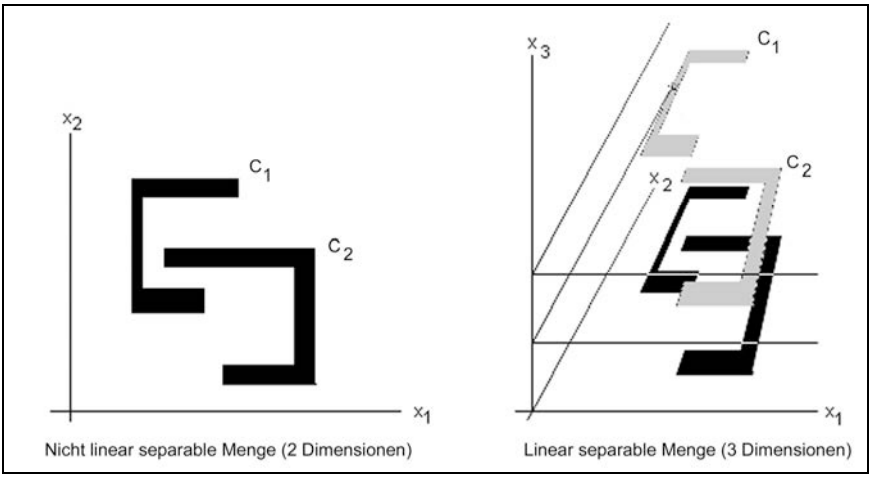


Abb. 7.14 Lineare Separabilität

Es seien die beiden Muster C_1 und C_2 (Abb. 7.14) zu trainieren, also zu klassifizieren bezüglich der Eingänge x_1 und x_2, d.h. es soll später, bei der Reproduktionsphase, erkannt werden, nachdem ein x_1 und x_2 eingegeben wird, ob der Punkt (x_1,x_2) im Muster C_1 oder C_2 oder in keinem von beiden liegt. Ohne das Problem mathematisch zu behandeln ist ersichtlich, dass es auch hier keine Gerade gibt, welche C_1 und C_2 linear separiert (linker Teil von Abb. 7.14).
Eine mögliche Lösung des Problems besteht darin, eine dritte Achse x_3 einzuführen und den beiden Mustern so eine verschiedene Höhe zuzuordnen (rechter Teil von Abb. 7.14). Jetzt gibt es eine lineare Hyperebene, welche die Muster trennt; das Problem ist jetzt linear separabel (in drei Eingangsdimensionen). Grundsätzlich kann man auf diese Art und Weise viele Probleme linear separieren und damit die Muster linear unabhängig machen. Dadurch ist die Konvergenz des Lernprozesses durch die Delta-Lernregel gesichert.
Es sei abschließend ein relativ prominentes neuronales Netz, das so genannte Fehlerrückführungsnetz oder auch Backpropagation-Netz, näher betrachtet. Es besitzt folgende Eigenschaften:

1. Das Netz ist vorwärtsgekoppelt, kann aber mehrschichtig sein.
2. Effektiver Eingang für alle Neuronen ist das Skalarprodukt, Aktivierungsfunktion ist die Identität (d.h. $c_i = \sum_j w_{ij} e_j$).
3. Die Ausgangsfunktion ist nicht linear (häufig die Fermi-Funktion) und differenzierbar.

Für die Lernregel wird eine so genannte Kostenfunktion als zu minimierende Fehlerfunktion benutzt:

$$D = \frac{1}{2} \sum_{\mu} \sum_{\nu} (A_\nu^\mu - S_\nu^\mu)^2$$

Die Idee dabei ist, dass die Gewichte so angepasst werden müssen, dass die Kostenfunktion ein Minimum annimmt. Die Soll-Outputs S hängen ja formelmäßig von den w_{ij} ab, so dass durch Ableiten und Nullsetzen der Kostenfunktion prinzipiell diejenigen w_{ij} gefunden werden können, für die D minimal ist. Die Summe erstreckt sich über alle Ausgangsneuronen ν sowie über die zu lernenden Musterpaare μ. Die Anwendung des Gradientenabstiegsverfahrens zur Minimierung der Kostenfunktion führt schließlich zur sog. Fehlerrückführungs-Lernregel:

$$\delta w_{ij} = \eta \sum_{\mu} \psi_i^\mu e_j^\mu$$

wobei sich das sog. *charakteristische Fehlermaß* Ψ folgendermaßen errechnet: Für die letzte (Ausgangs-)Schicht gilt:

$$\psi_i^\mu = (S_i^\mu - A_i^\mu) a_i'(c_i^\mu)$$

Für die vorletzte Schicht (und sukzessive für alle Zwischenschichten von rechts nach links) gilt:

$$\psi_i^\mu = a_i'(c_i^\mu) \sum_k \psi_k^\mu w_{ki}$$

wobei *k* der Laufindex der Neuronen der folgenden (bereits berechneten) Schicht darstellt. Die Gewichte w_{ki} sind dabei ebenfalls von der Folgeschicht, allerdings *vor* der Gewichtsanpassung, zu nehmen. Da der Fehler (rückwärts) durch die einzelnen Schichten des Netzes zurückgeführt wird, spricht man von Fehlerrückführung.

Ein Anwendungsbeispiel:
Der Autor des vorliegenden Buches hat zusammen mit Studenten der Elektrotechnik an der Fachhochschule Frankfurt am Main im Rahmen einer Diplomarbeit folgende Aufgabe durchgeführt: Es sollte ein neuronales Netz entworfen

und anschließend so trainiert werden, dass es in der Lage ist, zu einer vorgegebenen Melodie passende Harmonien zu komponieren. In der Lernphase sollte das Netz also Melodien als Eingabe-, und die zugehörigen Harmonien als Ausgabenmuster trainiert bekommen, so dass es in der Produktionsphase zu bisher nicht trainierten Melodien sinnvolle, für das menschliche Ohr einigermaßen anhörbare Harmonien erzeugt. Ein besonderer Gesichtspunkt war dabei, dass der Computer gleich das Ergebnis in hörbarer Form vorspielen sollte. Es mussten also neben dem neuronalen Netz noch Prä- und Post-Prozeduren (in C^{++}) geschrieben werden, welche die Eingabemelodien sowie die Ausgabe-Harmonien (incl. der Melodie) auf einer (MIDI-) Klaviatur bzw. einem Synthesizer in vom neuronalen Netz passende Ein- und Ausgaben umsetzen konnten.
Als neuronales Netz wurde ein so genanntes holografisches neuronales Netz gewählt. Diese wurden 1990 von John G. Sutherland zum ersten Mal vorgestellt. Neuartig an diesem Netz ist vor allem, dass die reellen Musterpaare auf die komplexe Ebene abgebildet werden und diese (komplexen) Zahlen zum trainieren und reproduzieren benutzt werden. Aufgrund der mathematischen Eigenschaften der komplexen Zahlen (Faltungssatz) kann man weit mehr Information in so einer holografischen Zelle ablegen als bei einem konventionellen neuronalen Netz. Abb. 7.15 zeigt den Aufbau der holografischen Neuronen.

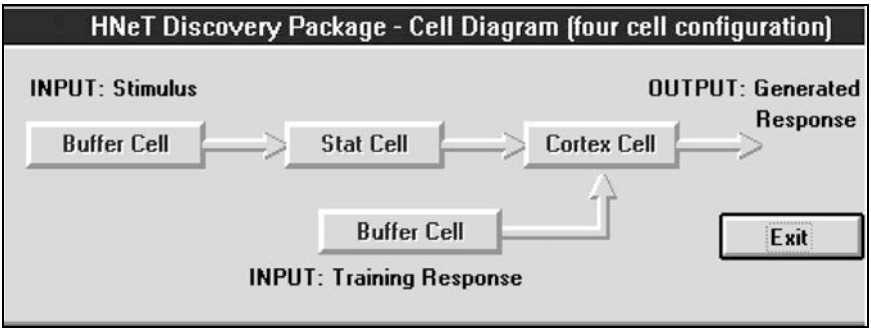


Abb. 7.15 Holografisches Neuron

Wie ersichtlich, gibt es vier Zell-Schichten.

Stimulus-Buffer-Zelle:
Hier werden die Eingangsdaten des zu trainierenden oder reproduzierenden Musters in komplexem Datenformat gespeichert. Die Konvertierungsmethoden werden später erläutert. Diese Daten können nachbearbeitet werden.

Response-Buffer-Zelle:
Hier werden die Ausgabedaten (Solldaten) der Trainings- bzw. Testphase in komplexem Datenformat gespeichert.

Stat-Zelle:
Hier werden über ein mathematisches Verfahren kombinatorische Produkte höherer Ordnung aus den Daten der Stimulus-Buffer-Zelle und der Response-Buffer-Zelle gespeichert. Dabei kann die Ordnung dieser kombinatorischen Produkte spezifiziert werden.

Cortex-Zelle:
Hier werden die eigentlichen Gewichte in holografischer Form in einer sog. *Korrelations-Matrix* gespeichert. Gelernt wird nur in den Cortex-Zellen.

Um reelle Zahlen in Komplexe zu transformieren bieten sich verschiedene Möglichkeiten an. Im Falle holografischer Netze gibt es prinzipiell zwei Möglichkeiten, die lineare (Rampenfunktion) und die sigmoide (Sigma-Funktion) Konvertierung. In beiden Fällen werden komplexe Zahlen in der Euler´schen Darstellung benutzt. Jede komplexe Zahl lässt sich bekanntlich darstellen als $z = \lambda e^{i\theta}$, wobei λ die sog. Magnitude und θ den Phasenwinkel bezeichnet. Holografische Netze interpretieren dabei die Magnitude als Maß für die "Vertraulichkeit" eines Zahlenwertes. Bei den Ein- und Ausgabemustern kann sie auf *1* oder von Hand gesetzt werden (z.B. für physikalischen Messdaten bei bekannter Fehlerabweichung etc.). Die eigentliche Umsetzung der reellen Ein- und Ausgabewerte erfolgt in die Abbildung auf den Phasenwinkel im Einheitskreis.
Günstig erweist sich bei der sigmoiden Transformation, dass, wenn die Eingangsdaten eine Gaußverteilung bilden, man dann nach der Transformation eine Gleichverteilung erhält.
Der Lernvorgang gestaltet sich wie folgt: Es seien wieder p Musterpaare $(\vec{E}^{\mu}, \vec{S}^{\mu})$ zu trainieren. Wir fassen hier jedes Musterpaar als zeitindiziert auf, d.h. wir führen einen Index t_μ ein, μ = 1...p. Wir bezeichnen jetzt

$$\vec{E}^{\mu} = (E_{j,t_\mu}) := (E_{1,t_\mu}, E_{2,t_\mu}, \ldots, E_{N_E,t_\mu})$$

$$\vec{S}^{\mu} = (S_{i,t_\mu}) := (S_{1,t_\mu}, S_{2,t_\mu}, \ldots, S_{N_A,t_\mu})$$

Mittels komplexer Transformation ergeben sich dann folgende Darstellungen:

$$E_{j,t_\mu} = \lambda_{j,t_\mu} e^{i\theta_{j,t_\mu}}$$

$$S_{i,t_\mu} = \gamma_{i,t_\mu} e^{i\phi_{i,t_\mu}}$$

In Matrix-Darstellung lautet das Ganze dann:

$$\mathbf{E} = \begin{pmatrix} \lambda_{1,t_1} e^{i\Theta_{1,t_1}} & \lambda_{2,t_1} e^{i\Theta_{2,t_1}} & \dots & \lambda_{NE,t_1} e^{i\Theta_{NE,t_1}} \\ \lambda_{1,t_2} e^{i\Theta_{1,t_2}} & \lambda_{2,t_2} e^{i\Theta_{2,t_2}} & \dots & \lambda_{NE,t_2} e^{i\Theta_{NE,t_2}} \\ \dots & \dots & \dots & \dots \\ \lambda_{1,t_p} e^{i\Theta_{1,t_p}} & \lambda_{2,t_p} e^{i\Theta_{2,t_p}} & \dots & \lambda_{NE,t_p} e^{i\Theta_{NE,t_p}} \end{pmatrix}$$

bzw.

$$\mathbf{S} = \begin{pmatrix} \gamma_{1,t_1} e^{i\Phi_{1,t_1}} & \gamma_{2,t_1} e^{i\Phi_{2,t_1}} & \dots & \gamma_{NA,t_1} e^{i\Phi_{NA,t_1}} \\ \gamma_{1,t_2} e^{i\Phi_{1,t_2}} & \gamma_{2,t_2} e^{i\Phi_{2,t_2}} & \dots & \gamma_{NA,t_2} e^{i\Phi_{NA,t_2}} \\ \dots & \dots & \dots & \dots \\ \gamma_{1,t_p} e^{i\Phi_{1,t_p}} & \gamma_{2,t_p} e^{i\Phi_{2,t_p}} & \dots & \gamma_{NA,t_p} e^{i\Phi_{NA,t_p}} \end{pmatrix}$$

Die Lernregel lautet dann:

$$\delta w_{ij} = \sum_{\mu=1}^{p} \overline{E}_{j,t_\mu} \cdot S_{i,t_\mu} = \sum_{\mu=1}^{p} \lambda_{j,t_\mu} \gamma_{i,t_\mu} e^{i(\theta_{j,t_\mu} - \phi_{i,t_\mu})}$$

oder in Matrix-Form

$$\left[\delta w_{ij}\right] = \overline{E}^T \cdot S$$

Entsprechend gilt für die Gewichtsmatrix

$$W^{alt} = W^{neu} + \left[\delta w_{ij}\right]$$

Anschaulich bedeutet dies, dass bei jeder späteren Eingabe eine Rotation des Phasenvektors in Richtung des jeweiligen Ausgabewertes erfolgt. Die Repro-

duktion schließlich erfolgt mittels Auflösen der obigen Matrix-Gleichung nach S:

$$A = \frac{1}{\omega} E^* \cdot W$$

wobei E^* den Reproduktions-Input und A den Reproduktions-Output bezeichnet. Der Normalisierungsfaktor ω wird so gewählt, dass die Magnitude der Ergebnisse zwischen 0 und 1 liegt, z.B. durch

$$\omega = \sum_{j=1}^{N_E} \lambda_j^*$$

Man kann übrigens zeigen, dass der gemachte Fehler approximativ liegt bei

$$\phi_{Fehler} \approx \frac{1}{\pi\sqrt{8}} \arctan\sqrt{\frac{p}{N_A}}$$

Um nun mit dem holografischen Netz Musik zu trainieren, war es zunächst erforderlich, diese in ein entsprechendes numerisches Datenformat umzuwandeln, da neuronale Netze immer mit Zahlen arbeiten. Zu diesem Zweck wurde ein Konverter programmiert, der die Melodiewerte sowie die Harmonisierungen in sinnvolle Zahlenwerte umwandelte. Es zeigte sich jedoch schon bald, dass eine diskrete Trainingsmenge der Form, dass einer bestimmten Note eine definierte Harmonie zugeordnet wird, zu keinem befriedigenden Ergebnis führte. Es stellte sich heraus, dass die besten Ergebnisse erzielt wurden, wenn quasi ein Fenster über eine zu trainierende Note gelegt wird, welche die vorherige und die nächste Note zusammen mit den zugehörigen Harmonien berücksichtigt. Durch diese Koppelung konnte schließlich tatsächlich das Netz sinnvoll trainiert und passable Ergebnisse erzielt werden. Es hat sich weiter gezeigt, dass die Art und Weise, wie das Netz seine Harmonien an nicht trainierte Melodien hinzukomponiert, stark von dem Stil der Trainingsmenge abhängig ist. Abb. 7.16 zeigt ein Ergebnis, wo das Lied „Die Gedanken sind frei“, welches nicht zuvor dem holografischen Netz trainiert wurde, harmonisiert wurde. Dem wird das Original gegenübergestellt. Die Bezeichnung T, S und D bedeuten Tonika (z.B. C-Dur), Subdominante (dann F-Dur) und Dominante (dann G-Dur). Ein p hinter der Harmonie steht für eine Quarte und die 7 für den Dominant-Sept-Akkord.

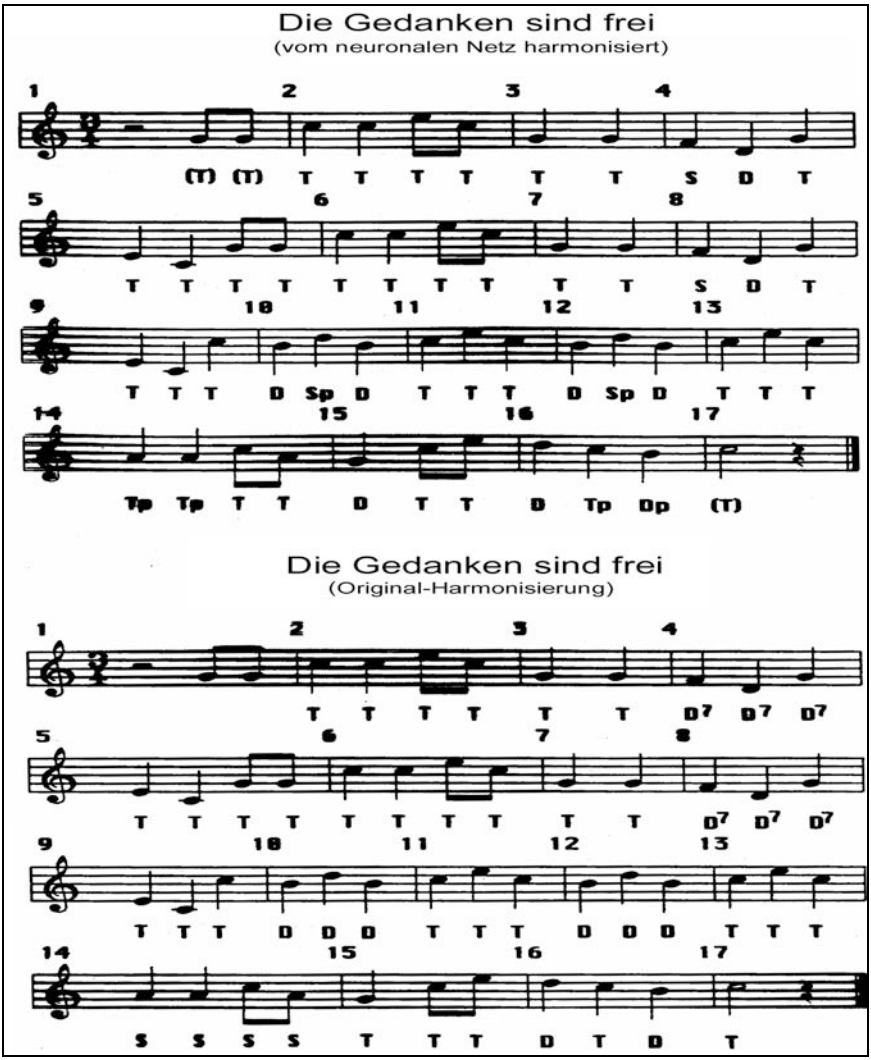


Abb. 7.16 Von einem holografischen neuronalen Netz harmonisiertes Lied

Im nächsten Kapitel wollen wir uns dem Thema „Genetische Algorithmen“ annehmen.

Übungen zum Selbsttest:

1. Es sei ein einfaches neuronales Netz (Musterassoziation) der Form vorgegeben, dass es 2 Eingänge und 2 Ausgänge besitzt. Es sollen folgende 2 Muster trainiert werden:

E_1	E_2	A_1	A_2
0	1	1	0
1	1	0	0

   Benutzen Sie ein einfaches lineares Gleichungssystem zur Berechnung der Gewichtsmatrix und geben Sie diese an!

8. Genetische Algorithmen

Genetische Algorithmen sind Teil des „Evolutionary Computing“ , welches nach Weicker[10] in die Gebiete *Genetische Algorithmen, Evolutionäres Programmieren, Evolutionsstrategien* und *Genetische Programmierung* unterteilt werden kann. Eine Untermenge der Genetischen Algorithmen sind *Classifier Systeme*.

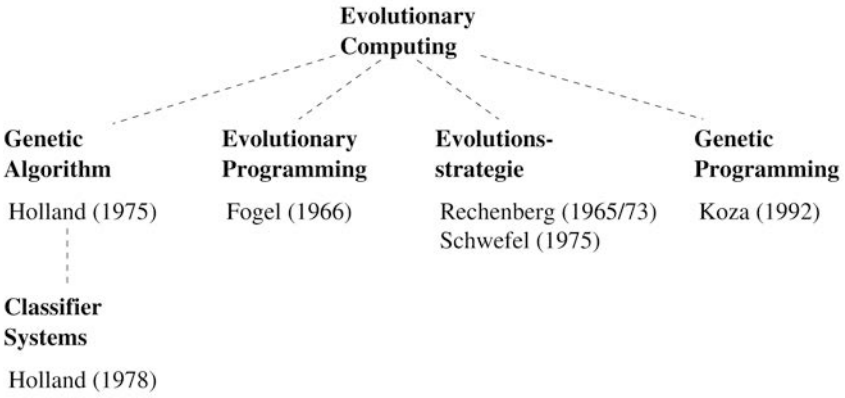


Abb. 8.1 Einordnung: Genetische Algorithmen[11]

Weicker definiert diese Teilgebiete wie folgt[12]:

Genetische Algorithmen (Holland, 1969, 1973, 1992) im klassischen Sinn arbeiten auf einer binären Darstellung des Problemraums. Der Selektionsdruck wird hierbei über eine stochastische, fitnessproportionale Selektion der Eltern aus der Vorgängerpopulation erzeugt. Als Operatoren dienen ein Mutationsoperator, welcher einzelne Bits im Individuum invertiert, und ein Rekombinationsoperator, welcher die Chromosomen der Eltern vermischt (ein sog. Crossover-Operator). In der Theorie der genetischen Algorithmen wird dabei dem Crossover-Operator durch das Schema-Theorem besonders viel Bedeutung beigemessen, welches die starke Verbreitung und Kombination vorteilhafter Informationen in der Population postuliert. Eine Unterform der genetischen Algorithmen bilden die Classifier Systeme (Holland & Reitman, 1978), bei welchen

[10] Weicker, K.: *Evolutionäre Algorithmen*, in: Softcomputing - Tagungsband zum ersten Softcomputing-Treffen, 1999
[11] Quelle: ibid.
[12] ibid.

in der binären Repräsentation Regeln zur Klassifikation oder Steuerung eines Systems erlernt werden.

Evolutionsstrategien (Rechenberg, 1973, 1994; Schwefel, 1981) arbeiten auf einer reellwertigen Darstellung des Problemraums. Die Eltern werden zufällig ausgewählt, und der Selektionsdruck entsteht ausschließlich bei der Umweltselektion, welche deterministisch nur die besten Individuen übernimmt. Bei der klassischen Evolutionsstrategie wurde lediglich die Mutation benutzt, wobei hier additiv gemäß einer Normalverteilung Änderungen an den reellwertigen Elementen eines Individuums vorgenommen werden. Sehr bald hat sich hier jedoch auch die Verwendung von Rekombinationsoperatoren sowie der Einsatz von selbstadaptiven Steuerungen der Parameter der Mutation abgezeichnet.

Evolutionäres Programmieren (Fogel et al., 1966) beschränken sich nicht auf eine bestimmte Repräsentation, sondern arbeiten auf der natürlichen Darstellung des Problems. Evolutionäres Programmieren wurde hierbei zunächst für die Entwicklung von endlichen Automaten zur Zeitserienvorhersage eingesetzt. Charakteristische Merkmale dieses Verfahrens sind ein Fehlen des Rekombinationsoperators sowie eine spezielle Selektion zur Erzeugung der nächsten Generation. Unabhängig von den Evolutionsstrategien wurden auch beim evolutionären Programmieren selbstadaptive Steuerungen der Mutationsparameter entwickelt.

Genetisches Programmieren (Koza, 1989, 1992) arbeitet auf einer dynamischen Darstellung, wie z.B. Baumstrukturen, und wird daher vornehmlich für die evolutionäre Entwicklung von Computerprogrammen u.a. eingesetzt.

Neben diesen strikt getrennten klassischen Formen haben sich inzwischen eine große Anzahl von Mischformen gebildet.
Der Grund, warum ich Genetische Algorithmen für dieses Buch herausgepickt habe ist der, dass Genetische Algorithmen einerseits sehr verbreitet und andererseits recht leicht zu implementieren sind. Zudem sind die anderen Gebiete des Evolutionären Computing so eng damit verwandt, dass sich die hier vorgestellten Konzepte leicht auf die anderen Teilgebiete übertragen lassen.
Bevor wir konkret an Genetische Algorithmen rangehen, möchte ich noch kurz die Stellung derselben im Verhältnis zu den bereits behandelten Kapiteln über Neuronale Netze und Fuzzy-Systeme beschreiben. Es ist nämlich in der Tat so, dass diese beiden Gebiete auch für Entwicklung genetischer Algorithmen nutzbringend eingesetzt werden können (auch wenn wir dies hier nicht näher untersuchen können).
In Abb. 8.2 sehen wir ein mögliches Zusammenspiel der besagten Bereiche:

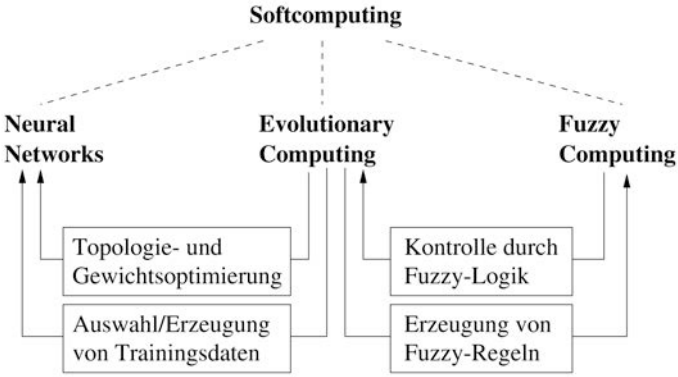


Abb. 8.2 Einordnung von Neuronalen Netzten und Fuzzy-Systemen[13]

Evolutionäre Algorithmen können bei den neuronalen Netzen zur Optimierung unterschiedlicher Aspekte eingesetzt werden, so z.B. als Alternative zu den klassischen Lernverfahren zur Gewichtsoptimierung, wie z.B. Backpropagation, aber auch zur Topologieoptimierung. Auch Fuzzy-Regeln können mit evolutionären Algorithmen erzeugt werden bzw. die Struktur und die Parameter eines Fuzzy-Modells angepasst werden. Eine andere Möglichkeit der Zusammenarbeit stellt die Kontrolle der Parameter eines evolutionären Algorithmus durch Fuzzy-Regeln dar.

Wenn man die Ursprünge und Motivationen der evolutionären Algorithmen betrachtet, stellt man einen gewissen Glauben an die Erschaffung eines „universellen Optimierers“ fest. Dies zeigt sich z.B. bei den genetischen Algorithmen durch die binäre Kodierung und damit einhergehend der Verneinung einer Übernahme jeglichen Problemwissens. Dies zeigt sich auch in den Versuchen vieler Forschungsarbeiten, durch experimentelle Analyse verschiedener Benchmark-Probleme die Überlegenheit eines Algorithmus über einen anderen zu beweisen. Dieser Glaube wurde durch das *No Free Lunch-Theorem* (Wolpert & Macready, 1997) erschüttert, in welchem in einer wahrscheinlichkeitstheoretischen Analyse bewiesen wurde, dass gemittelt über alle möglichen Probleme alle Algorithmen gleich gut bzw. gleich schlecht sind. Dies gilt insbesondere auch für den Algorithmus, der lediglich jedes Mal eine rein zufällig gezogene Lösung aus dem Suchraum betrachtet. Aus diesem Resultat und der Überlegenheit eines Algorithmus über einen anderen bezüglich eines speziellen Problems folgt direkt, dass es Nischen im Raum der Probleme gibt, in denen diese ver-

[13] Quelle: ibid.

schiedenen Algorithmen jeweils überlegen sind. Dies führt zu zwei praktischen Konsequenzen:

- Einerseits stellt sich die Frage, was genau die Nische für jeden einzelnen Algorithmus ist (allerdings fehlen trotz jahrelanger Forschungsaktivitäten noch eindeutige Charakteristika für die Anwendbarkeit eines speziellen Algorithmus).

- Andererseits ergibt sich daraus die Problematik, für ein festes Problem den passenden Algorithmus zu finden. Dies führt zur Anpassung des Algorithmus an das Problem und die Inkorporation von Problemwissen, womit sich nahezu alle Anwender beschäftigen. Dann spricht man auch von hybriden evolutionären Algorithmen, die lokale Suchalgorithmen oder Heuristiken im evolutionären Algorithmus mitbenutzen.

Das No-Free-Lunch-Theorem setzt also dem Traum, ein probleminvariantes Verfahren zu erhalten, ein jähes Ende. Wie immer in der Welt hat auch hier alles seinen Preis.
Wir wollen uns jetzt näher mit den genetischen Algorithmen befassen.
Die Idee genetischer Algorithmen ist die, dass für ein *Optimierungsproblem* eine möglichst „gute“ Lösung auf Basis der drei Vorgänge *Selektion*, *Mutation* und *Rekombination* (Crossover) gefunden wird. Biologen beobachten in der Natur, dass z.B. die Population von Hasen, welche Wölfe auf dem Speiseplan haben, dadurch das Problem das „Nicht-gefressen-werdens“ zu optimieren versuchen, in dem die schnellsten Hasen mit den besten Ohren am besten davon kommen (Selektion). Die anderen werden halt gefressen. Zufällige Mutation in der DNS der Hasen, welche z.B. die Größe der Ohren oder die Länge der Pfoten bestimmt, führen dann der Theorie nach zu den besagten Veränderungen, von denen nur die der Zielfunktion, welche immer mit dem Optimierungsproblem einhergeht, nämlich, nicht gefressen zu werden, am besten genügen, die dem Gefressenwerden entgingen. Diese Glücklichen paaren sich dann untereinander (Rekombination), so dass damit immer schnellere, besser hörende Hasen entstehen. Dabei sind allerdings sog. „elitäre Selektionen“ erfahrungsgemäß zu vermeiden. Damit ist gemeint, dass es für die Optimierung nicht unbedingt von Vorteil ist, wenn es nur „gute“ Lösungen gibt, denn sonst kann das ganze zum Stillstand kommen, es entwickelt sich gar nichts mehr (und die armen Wölfe verhungern).
Die Lösungen eines Optimierungsproblems nennt man im Zusammenhang mit genetischen Algorithmen auch *Bevölkerung* oder *Population*.
Die Frage ist nun, wie lassen sich die Prinzipien dieser biologischen Evolutions-Hypothese auf die Programmierung übertragen? Kann man Programme „züch-

ten“, in dem man diese drei Evolutionsprinzipien z.B. auf ein C-Programm anwendet? Man stelle sich also ein C-Programm vor, das man durch Mutation zufällig im Quellcode verändern lässt. In mehr als 99% der Fälle wird das Programm hinterher wohl einfach nicht mehr lauffähig sein. Oder versuchen wir, zwei verschiedene C-Programme zu rekombinieren, in dem wir beide Programme halbieren und die erste Hälfte des ersten Programms mit der zweiten Hälfte des zweiten zu einem neuen C-Programm vereinen. Auch hier würde man wohl kaum erwarten, dass man jetzt ein „besseres“ Programm erhält als die beiden vorherigen. So kann es also nicht funktionieren.
Was aber funktioniert, ist, wenn man die am Optimierungsproblem beteiligten Individuen (in der Biologie sind das z.B. die DNS-Stränge) binär kodiert, also als Folge von Nullen und Einsen darstellt, und dabei –wie bei der DNS auch- der jeweiligen Position (und natürlich dem binären Wert) eine bestimmte Bedeutung gibt. So könnte man für die Hasen z.B. einen Bit-String erfinden der Form:

0011000101010101010101010101000001111101010101010

in dem man definiert, dass die ersten 5 Bits die Länge der Beine angibt, die nächsten 5 Bits die Größe der Ohren usw.; eine Teilung (zum Zwecke der Rekombination) darf dann natürlich nur immer nach 5er-Blöcken gemacht werden. Hat man einen zweiten Hasen mit

10100100101000101010101010000010101011110101010101010

So kann man eine „Paarung“ (Rekombination) der beiden so machen, dass man per Zufall irgendwo beide Bit-Strings unterteilt und dann dem „Nachwuchs“ z.B. die erste Hälfte des einen und die andere Hälfte des anderen mit gibt:

00110001010101010101 | 0101010101000001111101010101010

1010010010100010101 | **0101000001010101111010101010101010**

führt zum „Nachwuchs“

0011000101010101010110101000001010101111010101010101010

Auch eine zufällige Mutation, d.h. das zufällige Abändern eines oder mehrerer Bits kann man hier vornehmen (hier z.B. das 3 Bit von links invertieren):

000**1000101010101010110101000001010101111010101010101010**

Das so veränderte Tier kann jetzt auf das Wolfsgebiet losgelassen werden und kann entweder besser überleben (wenn es „besser“ ist als seine Eltern) oder es wird gefressen und ist damit „wegselektiert“.
In Anlehnung an die Genetik nennt man die Einheiten (also im Hasen-Beispiel die 5er-Bits) der Bit-Strings auch *Gene*.

Im Rahmen des vorliegenden Buches kann jetzt nicht im Einzelnen auf die sehr umfangreiche Theorie genetischer Algorithmen eingegangen werden. Auch kann diese auf sehr hohem Abstraktionsniveau mathematisch untersucht werden[14].
Um dennoch ein „Gefühl“ für den Ablauf eines genetischen Algorithmus zu bekommnen, möchte ich ein einfaches Optimierungsproblem Schritt für Schritt „von Hand“ durcharbeiten, so dass die erwähnten Prinzipien klar werden und der Leser diese dann selbst weiter vertiefen kann.
Zunächst ist es allgemein so, dass mit einer zufälligen Ausgangspopulation begonnen wird. Dann werden die Zyklen Selektion, Rekombination und ggf. Mutation mehrmals durchlaufen nach von „außen“ festgelegten Randbedingungen (bei den Hasen z.B. die max. Hasenpopulation, die u.a. durch ein Revier vorgegeben ist etc.). Die Einzelheiten werden an nachfolgendem Beispiel näher erläutert.

Beispiel:
Es sei das Maximum der Funktion $y=x^2$ auf dem endlichen Intervall $[0,31]$ zu bestimmen. Will man dies mit einem Genetischen Algorithmus lösen, so benötigten man neben dem formulierten Optimierungsproblem eine Zielfunktion, manchmal auch „Fitness-Funktion“ genannt, sowie einen Ansatz für die Bit-Strings.
Als Arbeitsbedingungen legen wir daher folgendes fest:

1. Die Population besteht aus 4 Bit-Strings ("Gene" genannt); die Populationsanzahl soll konstant gehalten werden in jedem Zyklus
2. Jedes Gen besteht aus 5 Bit
3. Die Fitness-Funktion ist Naheliegenderweise $f(x)= x^2$

Wie in der Evolutionshypothese vorausgesetzt, spielt der Zufall natürlich eine große Rolle. So wie manche Hasen zufällig überlebten (nicht weil sie schon die „besten“ waren, sondern weil sie einfach Glück hatten und nicht gefressen wurden), wählen wir per Zufall eine *Anfangspopulation* aus. Wir starten also unseren genetischen Algorithmus damit, dass wir 4 Zufallszahlen zwischen 1 und 31

[14] Z.B. Rothlauf, F.: *Representations for Genetic and Evolutionary Algorithms*, Springer 2006

bestimmen. Nehmen wir einmal an, diese seien: 13, 24, 8 und 19 (in dieser Reihenfolge).
Zur Demonstration der Vorgehensweise nummerieren wir die entsprechenden „Gene“ und tragen sie in einer Tabelle (siehe Tabelle 1) ein, wo auch die „Fitness“ (=Wert der Zielfunktion) und ihr prozentualer Anteil gemessen an der Summe der Fitnesswerte (=100%).

Tabelle 1 :

String-Nr.	String	x-Wert	Fitness	%-Anteil
1	0 1 1 0 1	13	169	14,4
2	1 1 0 0 0	24	576	49,2
3	0 1 0 0 0	8	64	5,5
4	1 0 0 1 1	19	361	30,9
Gesamt			1170	100,0

Selektion:
Es gibt verschiedene Selektionsoperatoren bei genetischen Algorithmen. Letztlich habe sie alle die Aufgabe, „schlechte“ Individuen zu entfernen und bessere zu bevorzugen („Survival of the Fittest“). Wir bedienen uns dabei einer gängigen Selektionsmethode, dem sog. „Rouletteverfahren“ .
Dafür stelle man sich vor, man hätte einen Roulettetisch, der flächenmäßig so aufgeteilt ist, wie die Prozentzahlen der letzten Spalte aus Tabelle 1 angeben:

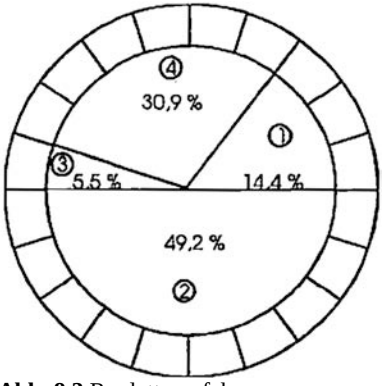


Abb. 8.3 Rouletteverfahren

Das Gen mit der besten Fitness bekommt den größten Bereich zugeordnet. Würde man jetzt eine Kugel zufällig auf den Tisch werfen (bzw. in die Gruben am Rand), so werden natürlich größere Bereiche häufiger getroffen als kleinere (um noch mal das Hasen-Beispiel zu zitieren: gäbe es 4 Hasentypen, die sich in den

Bereichen mit einer Häufigkeit gemäß den Prozentsätzen aufhalten, so hat ein Hase in dem großen Bereich die kleinste Chance, vom Wolf gefressen zu werden, was einer Optimierung seiner „Nicht-gefressen-werden"-Zielfunktion entspricht). Tabelle 2 zeigt jetzt, wie die Trefferrate beim Werfen von 4 Kugeln aussieht (letzte rechte Spalte, ganzzahlig gerundet aus der vorletzten Spalte):

Tabelle 2:

String -Nr.	Anfangs-Population (zufällig ermittelt)	x	$f(x)=x^2$ (Fitness)	%-Anteil bezogen auf die Summe d. Fitness	Treffer Wahrscheinlichkeit bei 1 Wurf.	Erwartete Trefferquote bei 4 Würfen	Tatsächliche Treffer (Roulette)
1	0 1 1 0 1	13	169	14,4	0,14	0,58	1
2	1 1 0 0 0	24	576	49,2	0,49	1,97	2
3	0 1 0 0 0	8	64	5,5	0,06	0,22	0
4	1 0 0 1 1	19	361	30,9	0,31	1,23	1
Summe		64	1170	100,0	1,00	4,00	4
Durchschnitt		16	293	25,0	0,25	1,00	1
Max		24	576	49,2	0,49	1,97	2

Das Gen Nummer 3 wird jetzt eliminiert und eine neue Population gebildet gemäß der Trefferrate des Rouletteverfahrens:

Rekombination:
Tabelle 3 zeigt das Ergebnis der Selektion und anschließenden Rekombination.

Tabelle 3:

String Nr.	Selektierte Gene (mit Anzeige der Crossover-Stelle)	Paare für das Crossover (zufällig ermittelt)	Crossover-Stelle (zufällig ermittelt)	Neue Population	x-Wert	$f(x)=x^2$ Fitness
1	0 1 | 1 0 1	3	2	0 1 0 0 0	8	64
2	1 1 0 0 | 0	4	4	1 1 0 0 1	25	625
3	1 1 | 0 0 0	1	2	1 1 1 0 1	29	841
4	1 0 0 1 | 1	2	4	1 0 0 1 0	18	324
Summe					80	1854
Durchschnitt					20	463,5
Max					29	841

Tabelle 3 ist also so entstanden: Die neuen Gene sind anzahlmäßig entsprechend der letzten Spalte aus Tabelle 2 ermittelt und neu nummeriert worden. Spalte 2 aus Tabelle 3 gibt diese Gene an. Die nächste Spalte rechts davon zeigt die zufällig ermittelten Paare für die Rekombination an. Diese Zufallszahlen müssen für unseren Fall mit der Einschränkung ermittelt werden, dass sich „keiner mit sich selbst" rekombiniert (also nicht die erste Zeile mit der ersten Zeile usw.) und nie die gleiche Zahl zweimal vorkommt. Für die 4. Spalte, wo per Zufall die Crossoverstelle bestimmt wird, gilt als Randbedingung, dass diese für zwei zu rekombinierende Gene gleich ist (sonst würde ja die Genlänge von 5 Bit nicht mehr eingehalten werden können). Für unseren Fall heißt das, dass eigentlich nur 2 statt 4 Zufallszahlen bestimmt werden müssen, und jede Zahl dann für 2 Gene gilt (nämlich die Elterngene). Spalte 5 zeigt die so neu entstandenen (Kinder-)Gene mit ihren dezimalen Entsprechungen (Spalte 6) und dem jeweils zugehörigen Fitnesswert (letzte Spalte).

Nächste Selektion:
Tabelle 4 zeigt die neue Tabelle zur nächsten Selektion (analog Tabelle 2):

Tabelle 4:

String-Nr.	Neue Population	x	$f(x)=x^2$ (Fitness)	%-Anteil bezogen auf die Summe d. Fitness	Treffer Wahrscheinlichkeit bei 1 Wurf.	Erwartete Trefferquote bei 4 Würfen	Tatsächliche Treffer (Roulette)
1	0 1 0 0 0	8	64	3,4	0,034	0,14	0
2	1 1 0 0 1	25	625	33,7	0,337	1,35	1
3	1 1 1 0 1	29	841	45,4	0,454	1,81	2
4	1 0 0 1 0	18	324	17,5	0,175	0,70	1
Summe		80	1854	100,0	1,000	4,00	4
Durchschnitt		20	463,5	25,0	0,250	1,00	1
Max		29	841	45,4	0,454	1,81	2

Diesmal selektiert das Rouletteverfahren Gen Nummer 1 als „untauglich", dafür wird Gen Nummer 3 in der nächsten Population zweimal vorkommen (gemäß der rechten Spalte der Tabelle 4).

In Tabelle 5 sieht man das Selektionsergebnis, versehen wiederum mit den jeweils zufällig ermittelten Crossoverstellen sowie den neuen Paarbildungen.

Nächste Rekombination:
Auch hier wurde die neue Population durch das Rouletteverfahren bestimmt, welches ja die beste Fitness berücksichtig.

Tabelle 5:

String Nr.	Selektierte Gene (mit Anzeige der Crossover-Stelle)	Paare für das Crossover (zufällig ermittelt)	Cross-over-Stelle (zufällig ermittelt)	Neue Populati-on	x-Wert	$f(x)=x^2$ Fitness
1	1 1 | 0 0 1	2	2	1 1 1 0 1	29	841
2	1 1 | 1 0 1	1	2	1 1 0 0 1	25	625
3	1 1 1 0 | 1	4	4	1 1 1 1 0	30	900
4	1 0 0 1 | 0	3	4	1 0 0 0 1	17	289
Sum-me					101	2655
Durch-schnitt					25,25	663,75
Max					30	900

Nächste Selektion:
Mit der neuen Population wird also wieder verfahren wie gehabt, d.h. die beste Fitness bestimmt die Trefferquote im Rouletteverfahren und selektiert so auf's Neue.

Tabelle 6:

String-Nr.	Neue Popula-tion	x	$f(x)=x^2$ (Fit-ness)	%-Anteil bezogen auf die Summe d. Fitness	Treffer Wahr-scheinlich-keit bei 1 Wurf.	Erwartete Treffer-quote bei 4 Würfen	Tatsäch-liche Treffer (Roulet-te)
1	1 1 1 0 1	29	841	31,7	0,317	1,27	1
2	1 1 0 0 1	25	625	23,5	0,235	0,94	1
3	1 1 1 1 0	30	900	33,9	0,339	1,35	2
4	1 0 0 0 1	17	289	10,9	0,109	0,44	0
Sum-me		101	2665	100,0	1,000	4,00	4
Durch-schnitt		25,3	666,25	25,0	0,250	1,00	1
Max		30	900	33,9	0,339	1,35	2

Man sieht, dass das Maximum schon fast erreicht ist. Noch eine weitere Rekombination, und der genetische Algorithmus kann beendet werden.

Nächste Rekombination:
Mit der neu selektierten Population wird jetzt erneut eine Rekombination nach den bekannten Verfahren durchgeführt:

Tabelle 7:

String Nr.	Selektierte Gene (mit Anzeige der Crossover-Stelle)	Paare für das Crossover (zufällig ermittelt)	Cross-over-Stelle (zufällig ermittelt)	Neue Populati-on	x-Wert	$f(x)=x^2$ Fitness
1	1 1 1 0 | 1	4	4	1 1 1 0 1	29	841
2	1 1 0 | 0 1	3	3	1 1 0 1 0	26	676
3	1 1 1 | 1 0	2	3	1 1 1 0 1	29	841
4	1 1 1 1 | 0	1	4	1 1 1 1 1	31	961
Sum-me					101	3319
Durch-schnitt					25,25	829,75
Max					31	961

Man sieht, dass das Maximum jetzt gefunden ist. In der Praxis kann man so einen Algorithmus abbrechen, wenn sich der beste Fitnesswert entweder nicht mehr ändert oder man gibt eine Gesamtanzahl möglicher Zyklen vor und nimmt diejenige Population, die den besten Fitnesswert darunter geliefert hat (dazu muss man diesen natürlich zwischenspeichern).

Wir haben bisher allerdings keine Mutationen durchgeführt. Das hat einen einfachen Grund: Mutationen sind in der Regel eher „zerstörerisch“ als nützlich, denn das zufällige Abändern eines Bits in einem Gen wird ja nicht nach irgendwelchen sinnvollen Kriterien vorgenommen. Sie werden in der Praxis eigentlich hauptsächlich deswegen gemacht, um nicht in eine Endlosschleife zu geraten. So ist z.B. eine Mutation oft erst nach 10.000 Selektions- und Rekombinationszyklen üblich. Mathematisch gesehen ist die Mutation die unwahrscheinlichste Methode, etwas Besseres hervorzubringen, und wird daher recht sparsam eingesetzt.

Damit sind wir inhaltlich am Ende, doch es folgt noch ein äußerst wichtiges Kapitel: Was hat die Künstliche Intelligenz in Zukunft für philosophische Konsequenzen?

Übungen zum Selbsttest:

1. Erklären Sie in eigenen Worten die Begriffe: Gen, Population, Selektion, Crossover und Mutation.

9. Philosophische Probleme mit KI

Auch über dieses Thema kann man dicke Bücher schreiben. Ich möchte daher versuchen, die wichtigsten kontroversen Streitpunkte kurz anzureisen und die jeweiligen Positionen gegenüber zu stellen.
In den Anfängen der Künstlichen Intelligenz war die Diskussion die, ob es jemals Computer geben wird, die den Turing-Test bestehen können. Dieser Test, benannt nach dem englischen Mathematiker Allan Turing, sieht vor, dass ein Mensch an einem Computerterminal sitzt und dort über die Tastatur umgangssprachliche Fragen stellt und seinerseits Fragen, die er am Bildschirm sieht, beantwortet. Ein ganz normale Kommunikation also. Der kommunizierende Mensch weiß dabei jetzt nicht, ob die Antworten (und daraufhin gestellten Fragen des „anderen") von einem anderen Menschen oder einem Computer kommen. Wenn ein Computer einst „so gut" ist, dass man diesen Unterschied nicht feststellen kann, gilt der Turing-Test als bestanden. Wie gesagt, in den 50er Jahren war die Frage, ob es so was jemals geben könnte. Meistens wurde dies verneint. Heute lächelt man darüber, denn der Turing-Test gilt längst als bestanden, wie viele Cyberbots zeigen[15].
Die Frage, die in neuerer Zeit die Philosophen quält, ist die, ob ein Computer ein Bewusstsein entwickeln kann. Das Grundproblem dabei ist, dass man das eigentlich gar nicht feststellen kann. Man kann es streng genommen ja noch nicht einmal bei einem anderen Menschen wirklich 100% feststellen. Woher wissen Sie denn, ob Ihr menschliches Gegenüber *wirklich* ein Bewusstsein hat? Sie können nicht in sein Gehirn schlüpfen um nachzuschauen. Sie können nur durch Analogschlüsse zum eigenen Verhalten dies mit hoher Wahrscheinlichkeit vermuten. Wenn sich ein Computer genau so benimmt, als *hätte* er ein Bewusstsein, und sich praktisch nur noch durch sein Äußeres vom Mensch unterscheidet, wer will dann mit Sicherheit ausschließen, dass dieses Bewusstsein echt ist? In diesem Zusammenhang wurde bereits von Computerrassismus und „Carbon-Based-Chauvinism" geredet und für Computerrechte (im Sinne von Menschenrechten) plädiert[16]. Die Computerhardware jedenfalls könnte es schon in sehr naher Zukunft ermöglichen, ein künstliches neuronales Netz mit 10 Milliarden Neuronen zu implementieren, was der Neuronenanzahl des menschlichen Gehirns entspricht, und dieses über Kameras, Mikrofone und Lautsprecher aus der Welt in Echtzeit „lernen" lassen, gerade so wie ein kleines Baby auch. Sollte sich dann eine Art Persönlichkeit einstellen, könnte schon argumentiert werden, dass nur noch die Äußerlichkeit einen Unterschied zum Menschen macht. Es gibt daher erwartungsgemäß Fürsprecher und Gegner des „Carbon-Based-

[15] siehe z.B. http://www.pandorabots.com/pandora
[16] vgl. z.B. Stork, D.G.: HAL's Legacy, MIT Press, 1997

Chauvinism“, und zwar aus allen möglichen Lagern. Ich möchte den Versuch unternehmen, einige Pro- und Contrapositionen zumindest aufzulisten. Wobei wir folgende Lager unterscheiden:

Pro-KI seien diejenigen, die der Meinung sind, dass ein Computer nicht „diskriminiert“ werden darf wegen seines „Aussehens“. D.h., wenn ein Computer sich so verhält, als hätte er Bewusstsein, als könnte er wirklich verstehen, und keine objektive Messung dies widerlegen kann, dann sollte man die Möglichkeit zumindest offen lassen, dass der Computer eben *wirklich* Bewusstsein und Verständnis haben kann.

Contra-KI sei das Lager derer genannt, welche der Meinung sind, dass ein Computer Bewusstsein und Verständnis immer nur *simuliert*, also eben immer nur so tut, als hätte er Bewusstsein, Verständnis, Gefühle etc.; in Wirklichkeit ist dem nicht so, diese Dinge gibt es nur bei Menschen (und evtl. noch bei Tieren).

Philosophische Argumente
Ein bekannter Vertreter des **Contra-KI**-Lagers, John Searle, stellte zur Unterscheidung zwischen wirklichem und simuliertem Verstehen folgendes Gedankenexperiment an:
Um wirkliche Intelligenz zu besitzen, muss man auch *wirklich verstehen* was man tut. So eine Selbstreflexion könne es aber auf einem Computer niemals geben, da es sich hier immer "nur" um ein formales System handle, wie folgendes Beispiel des Chinesischen Zimmers zeigt:
Angenommen, ein der chinesischen Sprache nicht mächtiger Mensch sitzt in einem geschlossenen Raum, welcher ausschließlich chinesischsprachige Literatur (Lexika, Romane, Fachbücher etc.) beinhaltet. Weiter sei angenommen, dass der Insasse über Vorschriften ("Algorithmen") verfügt, die ihm mitteilen, wie er auf in chinesisch gestellte Anfragen "von außen" eine sinnvolle chinesische Antwort zusammenstellen kann, ohne dass er die Fragen und Antworten versteht. Ein "echter" Chinese außerhalb des Zimmers könnte jetzt z.B. auf einem Zettel eine Frage in chinesisch formulieren, unter der Tür durchschieben und eine sinnvolle Antwort von dem Insassen ebenfalls auf einem Zettel in chinesisch zurückbekommen. Der Turing-Test wäre offensichtlich bestanden, obwohl der Insasse selbst den Sinn weder der Frage noch der Antwort verstanden hat. Er handelte nur rein formal.

Die Anhängerschaft des **Pro-KI**-Lagers führt im wesentlichen die Argumente auf, die Eingangs schon genannt wurden, dass nämlich die Abwesenheit einer wirklich objektiv messbaren Wirklichkeit selbst bei einem anderen Menschen dazu führt, dass das Vorhandensein von Bewusstsein und Verständnis nicht

absolut festgestellt werden kann. Zum Chinesischen Zimmer argumentieren sie, dass allein die Existenz eines formalen Systems, welches den Turing-Test besteht, natürlich nicht heißt, dass jeder Computer unter allen Bedingung so ein System repräsentiert. Und selbst wenn, so wird argumentiert, wieso soll es ausgeschlossen sein, dass ein formales System ein Bewusstsein bilden kann? Es gibt z.B. einen Äquivalenzsatz, der feststellt, dass jedes Neuronale Netz durch einen prozeduralen Algorithmus ersetzt werden kann und umgekehrt[17]. Unter einem rein materialistischen Blickpunkt könnte man also auch das menschliche Gehirn als ein solches formales System betrachten (dem trotzdem Bewusstsein zuerkannt wird).
Manchmal wird auch aufgeführt, dass es heute bereits technisch möglich ist, bestimmte Neuronen im Gehirn durch einen Chip zu ersetzen, der künstliche Neuronen enthält und so bestimmte Gehirnfunktionen wiederherstellen kann. Wenn man sich jetzt vorstellt, man würde im Gehirn Neuron für Neuron nacheinander durch so einen künstlichen Neuronenchip ersetzen, wo genau wäre dann die Grenze, wo das Bewusstsein plötzlich nicht mehr „echt“ ist, sondern nur noch simuliert? Nimmt das „echte“ Bewusstsein mit jedem ersetzten Chip ab und wird sukzessive durch ein „simuliertes“ ersetzt?

Theologische Argumente
Im Rahmen der Theologie könnte man ein „Seelen-Problem“ der folgenden Art formulieren:

1. Es existiert gar keine Seele.
2. Es existiert zwar eine Seele, aber diese ist zwangsläufig an Materie gebunden (jedenfalls während unseres Lebens) und "treibt" unseren Körper (und Geist) an
3. Es existiert eine Seele, die völlig unabhängig vom Körper ist (und den Körper auch nicht braucht um zu existieren)

Im ersten Fall gibt es nichts zu diskutieren (jedenfalls was einen Unterscheid zwischen Mensch und Computer betrifft).
Im 2. und 3. Fall stellt sich bei einem Computer das Problem, ob er auch eine Seele haben kann.

Pro-KI:
Wenn man einen "Computer-Rassismus" völlig aufgibt, dann muss man auch zugestehen, dass ein Computer eine Seele besitzen kann. Die Frage, "wie sie dort hineinkommt" stellt sich genau so wenig (oder so viel), wie bei der Zeugung und Geburt eines Menschen. Theologen aus diesem Camp argumentieren, dass

[17] Kask, E.: *Wissensrepräsentation in konnektionistischen Modellen*, Diplomarbeit, Fakultät für Informatik, Technische Universität Dresden, 1991

es einem souveränen Gott überlassen bleibt, wem er eine Seele geben will. Sollte dies auch ein Computer sein, wer will Gott hier einschränken? Die Bibel beispielsweise enthält zwar Passagen, aus denen hervorgeht, wem Gott eine Seele gibt (Menschen und Tieren), doch es steht nicht darin, wem er sie *nicht* gibt.

Contra-KI:
Erwartungsgemäß argumentieren KI-Gegner, dass nur der Mensch eine Seele besitzt. Der Computer als rein formales, nicht verstehendes System hat keine Seele und braucht sie auch gar nicht, da sein "menschliches" Verhalten ja nur simuliert ist. Manche Theologen aus diesem Lager verweisen sogar auf die Bibel, z.B. Offenbarung 13:15, wo von einem „belebten Standbild“ die Rede ist, und verweisen darauf, dass es sich hier um eine Verführung und Irreführung der Menschheit handelt.

Diese wenigen Argumente können im Rahmen des vorliegenden Buches nicht weiter diskutiert werden, doch liefern vielleicht dem einen oder anderen die Anregung, sich weitergehend in diese interessante Problematik einzuarbeiten.

Lösungen der Übungen zum Selbsttest

Kapitel 2:

1. a) Wahrheitstafel:

F	G	F→G	F ∧ (F→G)
W	W	W	W
W	F	F	F
F	W	W	F
F	F	W	F

F→G ist nur gültig in Zeile 1,3 und 4, und F ist gültig in Zeile 1 und 2. GLEICHZEITIG gültig sind sie nur in Zeile 1. Da ist dann aber auch G gültig.

b) Aus a) ist F→G gültig in Zeilen 1,3 und 4. Da F erfüllbar ist (in allen 4 Zeilen) ist auch G erfüllbar.

c) Wahrheitstafel:

F	G	F∧G	F→G	(F ∧G)→ (F→G)
W	W	W	W	W
W	F	F	F	W
F	W	F	W	W
F	F	F	W	W

Die Bedingung |= F∧G ist nur in Zeile 1 erfüllt, |= F→G ist in Zeile 1,3 und 4 erfüllt. |= (F ∧G)→ (F→G) ist in allen 4 Zeilen erfüllt. Es ist schon ausreichend, dass alles in der 1. Zeile erfüllt ist, denn da sind die Voraussetzungen erfüllt.

2. a) Wenn "Alles" relativ ist, dann heißt das, dass es nichts Absolutes gibt. Dann ist die Aussage "Alles ist relativ" selbst auch relativ bzw. die äquivalente Aussage "Es gibt nichts Absolutes" ist selbst auch nicht absolut. Wenn sie aber nicht absolut (wahr) ist, kann sie auch falsch sein. Wenn es aber falsch sein kann, dass es nichts Absolutes gibt, kann es also doch etwas Absolutes geben. Damit hat sich die Behauptung selbst zum Widerspruch geführt: Deswegen muss das Gegenteil wahr sein: Es gibt etwas Absolutes und damit kann nicht alles relativ sein.

b) Die Behauptung "Die Welt ist schlecht" setzt gerade einen Maßstab voraus, an dem "Gut und Schlecht" gemessen werden kann. In dem "Weil-Teil" wird dies aber verneint.

c) Der Satz "Wenn die Logik nicht gilt, dann..." ist Unsinn, denn ohne Logik gibt es auch keine Regel für das Schlussfolgern. Damit kann kein "Dann" aus dem "Wenn" geschlussfolgert werden, falls keine Logik da ist.

3. Die Aussage: „Weil es keine absoluten moralischen Gesetze gibt, soll man die (relativen) moralischen Gesetze anderer Kulturen tolerieren“ macht gleiche mehrfach logische Probleme. Im vorderen Teil des Satzes wird die Existenz moralisch absoluter Werte voraussetzungsmäßig verneint. Nun wird aber auch gefordert, etwas „tolerieren (zu) sollen“. Diese Toleranz ist selbst ein moralischer Wert. Dafür gibt es nun 2 Möglichkeiten: (i) das „tolerieren sollen“ ist selbst ein absoluter moralischer Wert oder (ii) es ist selbst (auch nur) ein relativer moralsicher Wert. Wir zeigen, dass beide Fälle einen Widerspruch erzeugen.
   Für (i) haben wir den Widerspruch sofort, denn wenn „tolerieren sollen“ ein absoluter moralischer Wert darstellen würde, dann würde er direkt der Voraussetzung, nämlich dass es keine absoluten moralischen Werte gibt, widersprechen. Das kann also nicht sein. Bleibt nur noch Fall (ii) übrig.
   Für (ii) können wir jetzt aber eine Kultur betrachten, welche der Meinung ist, die moralischen Werte anderer Kulturen gerade nicht zu tolerieren (solch eine Kultur lässt sich leicht finden). Nach dem (jetzt relativen) „tolerieren sollen“ fordert die Aussage, dass auch diese Kultur toleriert werden soll. Damit ist aber ein Widerspruch für den hinteren Teil der Aussage erzeugt worden, denn es sollen ja gerade die moralischen Werte anderer Kulturen toleriert werden. Man müsste diese Intoleranz tolerieren, was sich selbst widerspricht. Außerdem stellt sich für (ii) grundsätzlich die Frage, wieso man überhaupt etwas „sollen“ soll, wenn es nur relativ ist.

4. Die Harmonisierung der beiden Aussagen:
   a) Es gibt einen allmächtigen und allgütigen Gott
   b) Es gibt (moralisch) Böses in der Welt
   kann auf verschiedene Weisen erfolgen. Eine sehr einfache Harmonisierung, die sogar für Böses gilt, das nicht moralischen Ursprungs ist (z.B. Naturkatastrophen etc.) wäre die Hinzunahme folgender Aussage:
   c) Gott hat gute moralische Argumente, Böses in der Welt zuzulassen
   Alle 3 Aussagen sind damit logisch konsistent, es gibt keine Widersprüche. Damit ist das Theodizee-Problem vollständig logisch sauber gelöst. Theolo-

gen philosophieren natürlich dann darüber, welches die „guten Argumente“ für Gott sein könnten, Böses in der Welt zuzulassen. Doch das hat nichts mit Logik, sondern mit Theologie und Philosophie zu tun!
Deshalb sei hier eine weitere, wenn auch etwas aufwändigere Harmonisierung gegeben, die vielleicht philosophisch etwas befriedigender sein könnte. Wir nehmen dafür zusätzlich zu den beiden Aussagen a) und b) noch folgendes an:

c) Logik ist ein Attribut Gottes, gegen das er nicht verstoßen kann
d) Gott gab den Menschen freien Willen und Gott hält das für etwas Gutes
e) Der Mensch lernt nur aus Erfahrung (eigener oder der von anderen)

Diese Zusatzannahmen sind plausibel und entsprechen auch dem Gottesbild der meisten großen Weltreligionen; z.B. Annahme c) in der Bibel: Tit. 1:2 oder Joh. 1:1, Annahme d) aus Gen. 1 und so weiter.
Es ist klar, dass Gott nicht gegen seine Attribute wie Güte und Logik und Gerechtigkeit etc. verstoßen kann; im „Konfliktfall“, d.h. wenn ein Attribut nicht gleichzeitig mit einem anderen komplett ausgeführt werden kann, sind Prioritäten erforderlich, und zwar so, dass dabei das insgesamt notwendig auftretende Übel minimiert wird. So kommt es wohl, dass Gott eine moralisch falsche Entscheidung bei Menschen zulässt, obwohl das etwas Schlechtes ist, doch er muss gleichzeitig den freien Willen dieser Person respektieren; da einzugreifen würde Gottes Güte widersprechen. Er macht das nur, wenn es für ein größeres Gut unbedingt erforderlich ist. Gott würde zudem vermutlich „unlogisch“ werden müssen, würde er alle moralisch falschen Entscheidungen verhindern (da diese sich widersprechen könnten). Wegen der Zusatzannahmen c) und d) sind daher keine größeren Eingriffe zu erwarten. Bleibt noch die Frage, ob Gott die Menschen nicht hätte von vorn herein so machen können, dass sie „freiwillig“ immer nur moralisch das Richtige entscheiden. Abgesehen davon, ob so was logisch überhaupt geht (vermutlich nicht, sonst hätte Gott es so gemacht), ist aufgrund der Annahme e) es nicht verhinderbar, dass moralisch Böses in die Welt kommt, denn um aus etwas zu lernen, muss es ja mind. ein Mensch mal falsch gemacht haben.

5. Wir bezeichnen die Hebewerke mit A,B,C und die Zusatzhebewerke mit E und F.

   Unter Sollwert für A,B und C: 0, darüber: 1

   Zugeschaltet E bzw. F: 1, falls nicht: 0

   Aus den Bedingungen der Aufgabe ergibt sich folgende Wahrheitstafel:

A	B	C	E	F
1	1	1	0	0
1	1	0	1	1
1	0	1	1	1
1	0	0	1	0
0	1	1	1	1
0	1	0	1	0
0	0	1	1	0
0	0	0	0	0

KNF: E = $(\overline{A} + \overline{B} + \overline{C})(A + B + C)$

DNF: F = $AB\overline{C} + A\overline{B}C + \overline{A}BC$

Beide Formen sind bereits minimal!

6. a) $C\overline{D} \to T$ äq. $\overline{C} + D + T$

b) $(S + T) \to (C + D)$ äq. $\overline{S}\,\overline{T} + C + D$

c) $\overline{S} \to \overline{D}$ äq. $S + \overline{D}$

d) $\overline{C}D \to \overline{T}$ äq. $C + \overline{D} + \overline{T}$

Alle 4 Fälle müssen gleichzeitig gelten, daher:

$F = (\overline{C} + D + T)(\overline{S}\,\overline{T} + C + D)(S + \overline{D})(C + \overline{D} + \overline{T})$

muss wahr sein.

Idee: Daraus eine DNF machen, dann sieht man immer, welche Krankheit mit welchen Symptomen einhergeht. Um dies zu erreichen, kann man entweder den Term algebraisch solange umformen, bis eine vollständige DNF herauskommt, oder man macht eine Wahrheitstafel und konstruiert daraus wie gehabt die DNF. In beiden Fällen kommt heraus:

$F = (\overline{C}\,\overline{D}\,\overline{S}\,\overline{T} + CDST + CDS\overline{T} + \overline{C}DS\overline{T} + C\overline{D}ST + C\overline{D}\,\overline{S}T)$

Daraus kann man ablesen: Der Kranke leidet für

(i) unter der Krankheit C (zweiter und fünfter Term)
(ii) unter der Krankheit D (dritter und vierter Term)
(iii) unter der Krankheit C (sechster Term)
(iv) weder unter der Krankheit C noch D (erster Term)

Kapitel 3:

1. a) Bereinigen:
Es müssen unbenannt werden:
s, weil es an zwei ∀ gebunden ist
r, weil es im vorderen Teil an ∃ gebunden und im hinteren Teil ungebunden vorkommt
t muss nicht umbenannt werden, da es an keinen Quantor gebunden ist

Dies ergibt:
$F = \neg\exists r(A(r,t) \vee \forall s B(r,f(s))) \vee \forall u A(g(w,u),t))$

Alle Negationen nach innen ziehen:
$F = \forall r(\neg A(r,t) \wedge \exists s \neg B(r,f(s))) \vee \forall u A(g(w,u),t))$

Pränex-Form:
$F = \forall r \exists s \forall u\, [(\neg A(r,t) \wedge \neg B(r,f(s))) \vee A(g(w,u),t)]$

Skolem-Form: (s wird k(…))
$F = \forall r \forall u\, [(\neg A(r,t) \wedge \neg B(r,f(k(r)))) \vee A(g(w,u),t)]$

Universelles Quantifizieren :
$F = (\neg A(r,t) \wedge \neg B(r,f(k(r)))) \vee A(g(w,u),t)$

Unvollständige Konjunktive Normalform :
$(\neg A(r,t) \vee A(g(w,u),t)) \wedge (\neg B(r,f(k(r))) \vee A(g(w,u),t))$

b) Bereinigen:
Es muss unbenannt werden:
k, weil es an zwei ∃ gebunden ist
Eigentlich muss „hinten“ noch x und y umbenannt werden, doch da hinter diesen „hinteren“ Quantoren gar kein x und y mehr vorkommen, können diese Quantoren ganz weggelassen werden (anstatt diese umzubenennen und am Schluss dann doch wegzulassen)

Dies ergibt:
$F = \forall x \forall y(\neg(P(x) \vee \forall z Q(y,z)) \wedge \exists k \neg P(k)) \wedge \neg(\exists w \neg P(w))$

Alle Negationen nach innen ziehen:
$F = \forall x \forall y(\, (\neg P(x) \wedge \exists z \neg Q(y,z)) \wedge \exists k \neg P(k)) \wedge (\forall w P(w))$

Pränex-Form:
$F = \forall x \forall y \exists z \exists k \forall w((\neg P(x) \wedge \neg Q(y,z) \wedge \neg P(k) \wedge P(w))$

Skolem-Form:
$F = \forall x \forall y \ \forall w((\neg P(x) \wedge \neg Q(y,a(x,y)) \wedge \neg P(b(x,y)) \wedge P(w))$

Universelles Quantifizieren :
$F = \neg P(x) \wedge \neg Q(y,a(x,y)) \wedge \neg P(b(x,y)) \wedge P(w)$

Unvollständige Konjunktive Normalform :
$F = \neg P(x) \wedge \neg Q(y,a(x,y)) \wedge \neg P(b(x,y)) \wedge P(w)$

c) Bereinigen:
Es muss nichts unbenannt werden

Alle Negationen nach innen ziehen:
$F = \exists x \exists y \forall z(P(x,y,z) \vee (\forall a \exists b \neg P(a,b,x)))$

Pränex-Form:
$F = \exists x \exists y \forall z \forall a \exists b \ (P(x,y,z) \vee \neg P(a,b,x))$

Skolem-Form:
$F = \forall z \forall a \ (P(x_0,y_0,z) \vee \neg P(a,c(a,z),x_0))$

Universelles Quantifizieren :
$F = P(x_0,y_0,z) \vee \neg P(a,c(a,z),x_0)$

Unvollständige Konjunktive Normalform :
$F = P(x_0,y_0,z) \vee \neg P(a,c(a,z),x_0)$

2. Hier sind zunächst die Axiome in unvollständige, konjunktive Normalform zu bringen, und zwar nach der Methode wie in Aufgabe 1. Zuvor müssen die verbal formulierten Aussagen natürlich in prädikatenlogische umgesetzt werden. Das Universum der Struktur sei die Menge aller Einträge (Programmnamen) in die File Allocation Table (FAT). Die Klauselformen sind *kursiv* gekennzeichnet.

1. "LAN.EXE" ist ein Programm
   prg(lan)

2. "LAN.EXE" ist eine Software
   $sw(lan)$

3. Jede Software ist virusinfiziert
   $\forall X\ (\ sw(X) \rightarrow vi(X)\ )$
   Universell quantifiziert:
   $\neg sw(X) \vee vi(X)$

4. "WORD.EXE" ist ein Anwendungsprogramm
   $awprg(word)$

5. Alle virusinfizierten Programme ließen "WORD.EXE" in Ruhe oder greifen es an
   $\forall X\ \{\ [prg\ (X) \wedge vi(X)\ ] \rightarrow [\ irl(X, word) \vee angr(X, word)]\ \}$
   Universell quantifiziert:
   $\neg prg(X) \vee \neg\ vi(X) \vee irl(X, word) \vee angr(X, word)$

6. Es gibt Programmnamen, die von allen anderen in Ruhe gelassen werden
   $\exists Y\ \forall X\ irl\ (X,Y)$
   Universell quantifiziert:
   $irl(X,Y)$

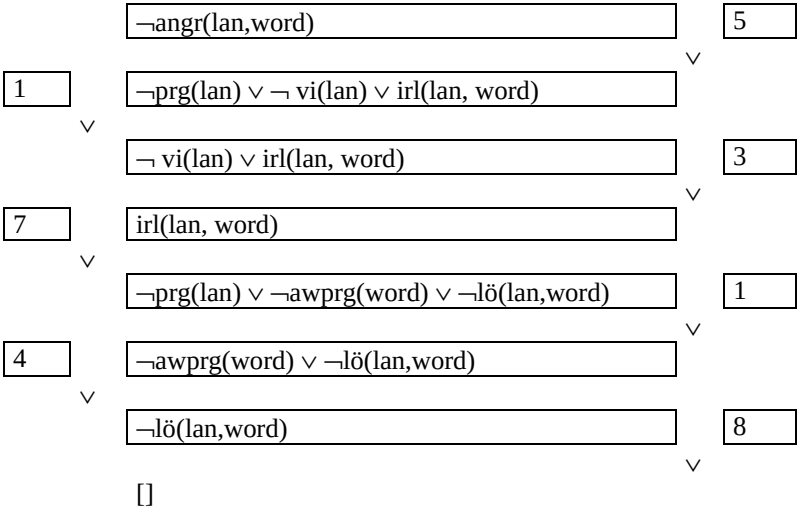
7. Wenn Programme versuchen, Anwenderprogramme zu löschen, dann heißt das, das diese Programme die Anwenderprogramme nicht in Ruhe lassen
   $\forall X\ \forall Y\{\ [\ prg(X) \wedge awprg(Y) \wedge lö(X,Y)\ ] \rightarrow \neg irl\ (X,Y)\ \}$
   Universell quantifiziert:
   $\neg prg(X) \vee \neg awprg(Y) \vee \neg lö(X,Y) \vee \neg irl\ (X,Y)$

8. "LAN.EXE" versucht, "WORD.EXE" zu löschen
   $lö(lan, word)$

Der Beweis der Aussage:

9. "LAN.EXE" greift "WORD.EXE" an
   $angr(lan, word)$

erfolgt durch sukzessive Resolventenbildung des Gegenteils dieser Aussage (also $\neg angr(lan,word)$) mit den Axiomen 1-8, wobei die jeweiligen Variablen der Axiom ggf. an die vorkommenden Konstanten gebunden werden:



Da die leere Klausel erzeugt werden konnte (was einem Widerspruch gleichkommt), muss nach dem Resolutionstheorem das Gegenteil der mit zur Resloventbildung gestarteten Aussagen wahr sein, also die Aussage 9).

3. Das Vorgehen ist analog der Aufgabe 2. Das Universum sei die Menge aller lebenden Entitäten auf der Erde. Es folgt:

1. Hugo ist ein bekannter Schauspieler.
   sp(hugo)

2. Karl ist ein Mensch und er ist auch schön.
   eigentlich: mensch(karl) ∧ schön(karl). Besser aber:

2a. *mensch(karl)*

2b. *schön(karl)*

3. Leider sind alle schönen Menschen auch eitel.
   $\forall X \{ [schön(X) \wedge mensch(X)] \rightarrow eitel(X) \}$
   Universell quantifiziert:
   $\neg schön(X) \vee \neg mensch(X) \vee eitel(X)$

4. Wenn ein Mensch eitel ist, so will er nicht so sein wie Hugo oder er will ihm alles nachmachen.

$\forall X \{ [mensch(X) \wedge eitel(X)] \rightarrow [\neg ssw(X, hugo) \vee anm(X, hugo)] \}$
Universell quantifiziert:
$\neg mensch(X) \vee \neg eitel(X) \vee \neg ssw(X, hugo) \vee anm(X, hugo)$

5. Menschen wollen so sein wie Schauspieler, falls sie diese mögen.
   $\forall X \ \forall Y \{ [mensch(X) \wedge sp(Y) \wedge mögen(X,Y)] \rightarrow ssw(X,Y) \}$
   Universell quantifiziert:
   $\neg mensch(X) \vee \neg sp(Y) \vee \neg mögen(X,Y) \vee ssw(X,Y)$

6. Karl mag Hugo.
   mögen(karl, hugo)

Das Beweisziel ist:
7. Karl macht Hugo alles nach.
   anm(karl, hugo)

Resloutionsmoethode:

	¬anm(karl, hugo)	4
	∨	
2a	¬mensch(karl) ∨ ¬eitel(karl) ∨ ¬ssw(karl, hugo)	
	∨	
	¬eitel(karl) ∨ ¬ssw(karl, hugo)	3
	∨	
2a	¬ssw(karl, hugo) ∨ ¬schön(karl) ∨ ¬mensch(karl)	
	∨	
	¬ssw(karl, hugo) ∨ ¬mensch(karl)	2b
	∨	
5	¬ssw(karl, hugo)	
	∨	
	¬mensch(karl) ∨ ¬sp (hugo) ∨ ¬mögen(karl,hugo)	2b
	∨	
1	¬sp (hugo) ∨ ¬mögen(karl,hugo)	
	∨	
	¬mögen(karl,hugo)	6
	∨	
	[]	

Damit ist auch hier die ursprüngliche Behauptung 7. wahr.

Kapitel 4:

1. Das Prologprogramm zur Bestimmung von `n!` lautet:

```
faku(0,1):-!.
faku(N,Erg):-N1 is N-1,
             faku(N1,Zwierg),
             Erg is N*Zwierg.
```

2. Das Prolog-Programm zur rekursiven Berechnung von *cos(x)* und *sin(x)* lautet:

```
complex(X,Cos,Sin):- abs(X)=<0.000000001,
                     Cos is 1, Sin is X,!.

complex(X,Cos,Sin):- abs(X)>0.000000001, Xneu is X/2,
                     complex(Xneu,T,W),
                     Cos is T*T-W*W,
                     Sin is 2*T*W.
```

Hinweis:
Für die Berechung wurde in Regel 1 ausgenutzt, dass die Tangentengleichung für "kleine" *x* für *cos(x)=1* und für *sin(x)=x* ist. In Regel 2 wurde die komplexe Darstellung von *cos(x)* und *sin(x)* benutzt:

$$e^{x} = \cos(x) + i\sin(x)$$

Quadriert man diesen Term, so erhält man:

$$e^{2x} = (\cos(x) + i\sin(x))^2 = \cos^2 x - \sin^2 x + 2i\cos(x)\sin(x)$$

Ersetzt man hier *x* mit *x/2*, so folgt daraus:

$$e^{x} = \cos^2\frac{x}{2} - \sin^2\frac{x}{2} + 2i\cos(\frac{x}{2})\sin(\frac{x}{2})$$

Vergleich von Real- und Imaginärteil aus der ersten und der letzten Gleichung ergibt („Additionstheoreme"):

$$\cos(x) = \cos^2\frac{x}{2} - \sin^2\frac{x}{2}$$

$$\sin(x) = 2\cos(\frac{x}{2})\sin(\frac{x}{2})$$

Dies sind genau die benutzten Rekursionsformeln.

3. Das Prolog-Programm lautet:

```
mensch(markus).
pompeianer(markus).
roemer(X):-pompeianer(X).
herrscher(caesar).
loaylzu(X,caesar):-roemer(X).
hassen(X,caesar):-roemer(X).
loyalzu(X,Y):-
not(mensch(X),herrscher(Y),ermorden(X,Y)).
ermorden(markus,caesar).
```

Die Eingabe: `?-hassen(markus,caesar).`

liefert: `Yes`

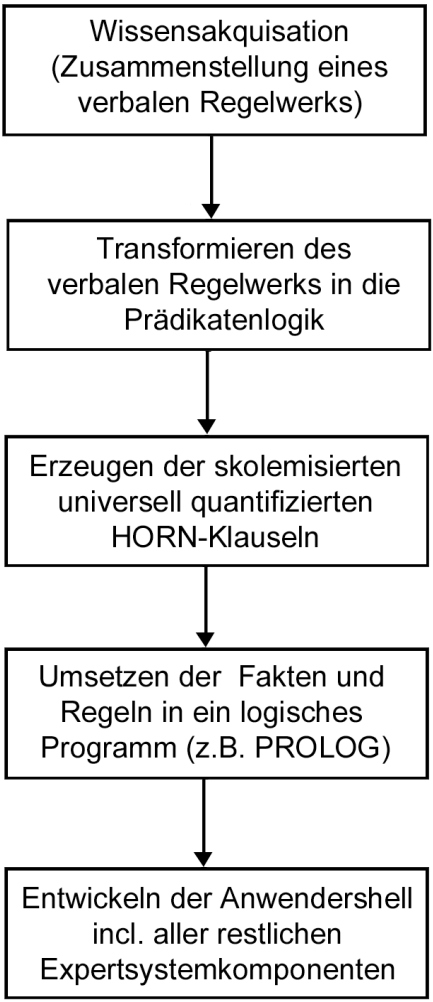
4. Die Lösung des Einstein-Rätsels lautet:

```
erstes(E,[E|_]).
mittleres(M,[_,_,M,_,_]).
links(A,B,[A|[B|_]]).
links(A,B,[_|R]):-links(A,B,R).
neben(A,B,L):-links(A,B,L);links(B,A,L).

run:-
  X = [_,_,_,_,_],
  member([rot,brite,_,_,_],X),
  member([_,schwede,_,_,hund],X),
  member([_,daene,tee,_,_],X),
  links([gruen,_,_,_,_],[weiss,_,_,_,_],X),
  member([gruen,_,kaffee,_,_],X),
  member([_,_,_,pallmall,vogel],X),
  mittleres([_,_,milch,_,_],X),
  member([gelb,_,_,dunhill,_],X),
  erstes([_,norweger,_,_,_],X),
  neben([_,_,_,marlboro,_],[_,_,_,_,katze],X),
  neben([_,_,_,_,pferd],[_,_,_,dunhill,_],X),
  member([_,_,bier,winfield,_],X),
  neben([_,norweger,_,_,_],[blau,_,_,_,_],X),
  member([_,deutsche,_,rothmans,_],X),
  neben([_,_,_,marlboro,_],[_,_,wasser,_,_],X),
  member([_,N,_,_,fisch],X),
  write(X),nl,
  write('Der '),write(N),write(' hat einen Fisch als
  Haustier.'),nl.
```

Kapitel 5:

1. Ablauf beim Entwickeln eines Expertensystems:



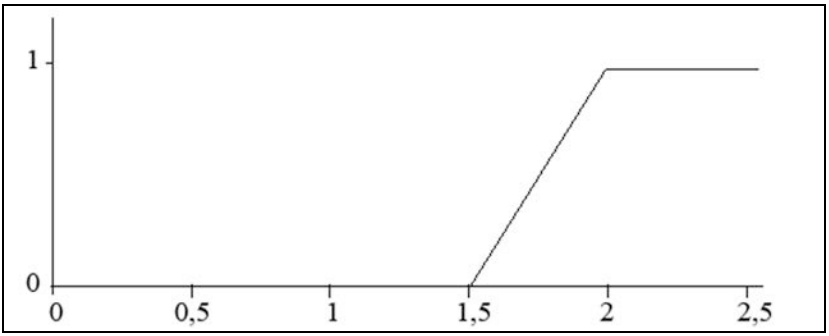
Kapitel 6:

1. Siehe Definitionen dieser Begriffe in Kapitel 6.

2. Sei A = Menge aller Personen und $\Omega(x) = groß(x)$ die Zugehörigkeitsfunktion, welche ein Maß dafür sein soll, ob eine Person groß ist oder nicht. Die Körpergröße in Meter (m) sei nachfolgend mit *höhe(x)* bezeichnet. Damit sei X die Fuzzy-Menge aller großen Personen. Man könnte $\Omega(x)$ beispielsweise definieren durch:

$$gro(x) = \begin{cases} 0 & \text{falls } höhe(x) \prec 1,5m \\ \frac{höhe(x)-1,5m}{0,5m} & \text{falls } 1,5m \le höhe(x) \le 2m \\ 1 & \text{falls } höhe(x) \succ 2m \end{cases}$$

Grafische Darstellung:

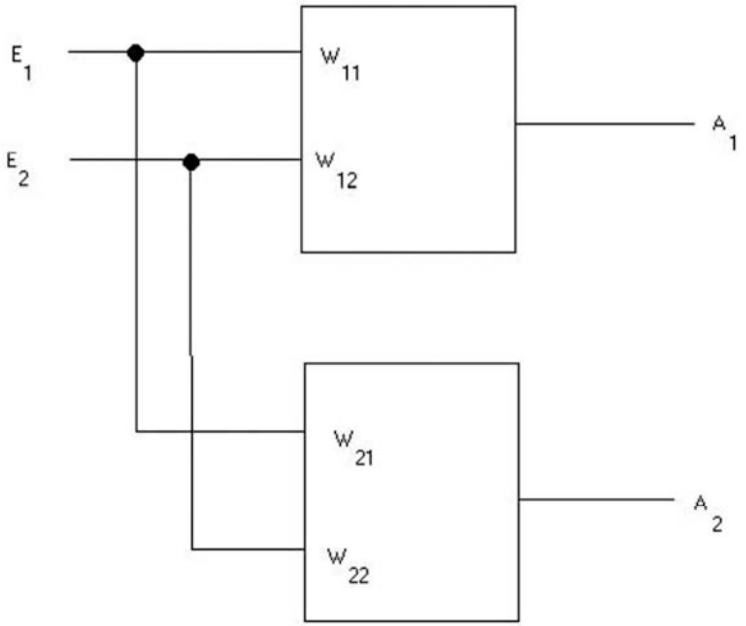


Damit kann man jetzt die Werte der Zugehörigkeit zu großen Personen für jeden beliebigen Fall feststellen, z.B.

PERSON	HÖHE	WERT DER ZUGEHÖRIGKEITSFUNKTION
Karl	1,48m	0
Ernst	1,60m	0,2
Schorsch	1,80m	0,6
August	1,90m	0,8
Ludwig	2,10m	1

Kapitel 7:

1. Layout:



Perzeptron-Ansatz:

$$W\vec{E} = \vec{A}$$
$$\Leftrightarrow$$
$$\begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{pmatrix} \begin{pmatrix} E_1 \\ E_2 \end{pmatrix} = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix}$$

Einsetzen der Werte aus der Tabelle der Aufgabe liefert ein einfaches Gleichungssystem mit den Lösungen:

$w_{11} = -1$
$w_{12} = 1$
$w_{21} = 0$
$w_{22} = 0$

Kapitel 8:

1. Begriffserklärungen:

Gen Feste Bitfolgen werden Gene genannt. Dabei kommt dann jeder Bit-Position eines solchen Bit-Strings eine bestimmte Bedeutung zu, und der Wert (0 oder 1) legt gewisse Eigenschaften fest.

Population Eine Menge von Bit-Strings. Hat man ein Optimierungsproblem codiert, so wird das Optimum durch die Anwendung folgender, aus der Evolutionstheorie entlehnter Methoden gesucht:

Selektion Darunter versteht man die Auswahl einer festgelegten Anzahl Bit-Strings aus einer Menge von Bit-Strings ("Survival of the Fittest"). Die Auswahl erfolgt nach bestimmten Kriterien (Fitness- bzw. Zielfunktion) und Methoden (z.B. Rouletteverfahren).

Crossover Hierunter versteht man das Hervorbringen neuer Bit-Strings aus "Eltern-Bit-Strings" durch Aufschneiden der Eltern-Gene an zufälligen Stellen und Aneinanderheften nach Vertauschung der abgeschnittenen Gen-Teile.

Mutation Zufälliges Abändern eines Bits im Gen.

Weiterführende Literatur

Zur Aussagen- und Prädikatenlogik:
Dassow, J.: **Logik für Informatiker**, Teubner 2005
Kutschera, F.v. und Breitkopf, A.: **Einführung in die moderne Logik,** Alber Karl 1992
Barwise, J. und Etchemendy, J.: **Sprache, Beweis und Logik. Band I. Aussagen- und Prädikatenlogik,** Mentis 2005

Zu PROLOG und Expertensystemen:
Bratko, I.: **Prolog Programming for Artifical Intelligence,** Addison-Wesley 2000
Clocksin, W.F. und Christopher, S.M.: **Programming in Prolog,** Springer 2003
Kurbel, K.: **Entwicklung und Einsatz von Expertensystemen,** Springer 1992

Zu Fuzzy-Systemen:
Michels, K. et all: **Fuzzy-Regelung. Grundlagen, Entwurf, Analyse,** Springer 2002
Kruse, R. et. All: **Fuzzy-Systeme,** Teubner 1993

Zu Neuronalen Netzen:
Macho, S.: **Kognitive Modellierung mit Neuronalen Netzen. Eine anwendungsorientierte Einführung,** Huber Hans 2003
Walde, J.F.: **Design Künstlicher Neuronaler Netze,** Deutscher Universitätsverlag 2005

Zu Genetischen Algorithmen:
Gerdes, I. et all: **Evolutionäre Algorithmen (Computational Intelligence),** Vieweg 2004
Rothlauf, F.: **Representations for Genetic and Evolutionary Algorithms (Studies in Fuzziness and Soft Computing),** Springer 2006

Zu Philosophischen Problemen mit KI:
Zimmerli, W.C. und Wolf, F.: **Künstliche Intelligenz. Philosophische Probleme,** Reclam 1994
Becker, K.: **Philosophische Diskussion der künstlichen Intelligenz und des künstlichen Bewusstseins. Denkmaschinen,** Fouque Literaturverlag 2000

Index