Grundlagen der Künstlichen Intelligenz

(Entwurf)

SS 06

Sven Eric Panitz

FH Wiesbaden

Version 1. Juli 2006

Dieses Skript entsteht mit heißer Nadel parallel zur laufenden Vorlesung *Grundlagen der Künstlichen Intelligenz*. Es wird entsprechend oft neue Versionen geben und entsprechend viele Fehler geben, über die ich bitte stillschweigend hinwegzusehen.

Der Quelltext dieses Skripts ist eine XML-Datei, die durch eine XQuery in eine LaTeX-Datei transformiert und für die schließlich eine pdf-Datei und eine postscript-Datei erzeugt wird. Beispielprogramme werden direkt aus dem Skriptquelltext extrahiert und sind auf der Webseite herunterzuladen.

Inhaltsverzeichnis

Kapitel 1

Einführung

Prelude

Bengt: Hallo. Guten Tag.

Bert: Hallo!

Bengt: Haben Sie einen Namen?

Bert: Wenn Sie mir einen geben.

Bengt: Dann werde ich Sie jetzt Bert nennen. Mein Name ist Bengt.

Bert: Okay.

Bengt: Bert, wie geht es Erni?

Bert: Warum wollen Sie das wissen?

Bengt: Macht es Ihnen etwas aus, daß ich das wissen will?

Bert: Es erscheint mir recht irrelevant.

Bengt: Warum erscheint es Ihnen irrelevant?

Bert: Ich wollte hier über keinen Erni sprechen, sondern etwas über künstliche Intelligenz lernen.

Bengt: Sind Sie extra hergekommen, um hier etwas über künstliche Intelligenz zu lernen?

Bert: Ja.

Bengt: Wollen Sie mehr darüber sprechen?

Bert: Das Fach interessiert mich. Von Haus aus bin ich eigentlich Germanist.

Bengt: Belastet es Sie, das Sie Germanist sind?

Bert: Nein, wieso sollte es.

Bengt: Sie glauben nicht, daß es sollte?

BERT: Nein

BENGT: Sind Sie sicher?

BERT: Ich glaube, man kann nie wirklich sicher sein.

BENGT: Wieso glauben Sie, man kann nie wirklich sicher sein

BERT: Nun, ist das nicht eine alte Regel?

BENGT: Was bedeuten Regeln für Sie?

BERT: Regeln können sehr hilfreich sein, sich zu orientieren.

BENGT: Gut, damit Sie sich in dieser Veranstaltung über künstliche Intelligenz besser orientieren können, gebe ich Ihnen eine kleine Regel mit.

(Überreicht ein Regelblatt mit dem Algorithmus aus Abbildung 1.1.)

Literatur

Die KI ist ein weites Feld, das in viele Spezialdisziplinen zerfällt und dessen Ränder unscharf definiert sind. Es gibt umfangreiche Literatur zu den unterschiedlichen Spizialdisziplinen, die zur KI gehören. Als ein recht umfassendes Lehrbuch über das Gesamtgebiet hat sich *Artificial Intelligence: A Modern Approach* von Russel und Norvig[RN95] erwiesen. Ihm wird die Vorlesung weitgehendst folgen und viele Beispiele sind dem Buch entnommen.

Ein sehr hilfreiches und umfassendes gut zu lesende Skript ist das Skript von Manfred Schmidt-Schauß an der Universität Frankfurt[SS06].

Ein etwas älteres einflußreiches Lehrbuch stammt von Winston[Win92].

Ein reich bebildertes mehr populärwissenschaftliches recht kurzweiliges Werk stammt von Raymond Kurzweil[Kur93].

Mit einigen philosophischen Fragen und Gedankenexperimenten beschäftigt sich ein recht unterhaltsames Buch von Poundstone[Pou91].

Programme dieses Skripts

In diesem Skript befinden sich mehrere Programme, in denen einige einfache Algorithmen der KI umgesetzt sind. Die Programme sind in der Programmierspache *Scala* (`http://scala.epfl.ch/`) geschrieben. *Scala* erzeugt als Zielcode Java Byte Code. Um Scalaprogramme auszuführen bedarf es also der Javalaufzeitumgebung. Zusätzlich wird die Standardbibliothek von Scala in Form einer Jar-Datei benötigt.

Die Wahl auf Scala fiel, um zum einen geistig ein wenig rege zu bleiben, zum anderen, weil sich in Scala die Programme wesentlich kürzer formulieren lassen als in Java und dadurch die Programme sich mehr auf die wesentliche algorithmische Idee konzentrieren.

Was ist zu tun, wenn auf einem Übungsblatt nur chinesische Zeichen stehen?

Betrachten Sie die Zeichen einer Aufgabe, die in einer Zeile stehen. Vor manchen Zeichen steht das Symbol ¬.
Suchen Sie zwei Zeilen, die ein chinesisches Zeichen x gemeinsam haben, so daß in einer Zeile vor diesem Zeichen das Symbol ¬ steht und in der anderen Zeile nicht.
Schreiben Sie dann eine neue Zeile, in der Sie alle Zeichen aus den zwei gewählten Zeilen kopieren, nur das gemeinsame Zeichen x bzw. $\neg x$ ist nicht zu kopieren.
Taucht ein weiteres Zeichen in der resultierende Zeile einmal mit vorangestellten Symbol ¬ und einmal ohne auf, so kann dieses Zeichen inklusiven dem davorstehenden Symbol ¬ gelöscht werden. Tritt ansonsten ein Zeichen doppelt auf, so kann das doppelte Auftreten gelöscht werden.
Fahren Sie so lange rekursiv mit den Zeilen fort, bis Sie eine leere Zeile erzeugt haben. Markieren Sie diese mit dem Symbol □.

Beispiel 1.0.1 *In der Aufgabe sind sieben Zeilen gegeben:*

立春雨 (1.1)
¬立¬春 (1.2)
¬立¬雨 (1.3)
¬春¬雨 (1.4)
春¬立 (1.5)
立¬雨 (1.6)
¬春 (1.7)

Es lassen sich die folgenden Zeilen mit den Regeln ableiten:

¬立 , *aus 1.2 mit 1.5* (1.8)
¬雨 , *aus 1.3 mit 1.6* (1.9)
春雨 , *aus 1.1 mit 1.8* (1.10)
春 , *aus 1.9 mit 1.10* (1.11)
□ , *aus 1.7 mit 1.11* (1.12)

Abbildung 1.1: Anleitung zur Lösung von Übungsaufgaben

1.1 Was ist Künstliche Intelligenz

1.1.1 Rationale Agenten

Das Ziel der KI ist es einen rational und autonom agierenden Agenten zu entwickeln. Unter einen solchen Agenten wird eine autonome Einheit verstanden, die Rezeptoren für ihre Umwelt hat und mit Aktionen auf die Umwelt reagieren kann. Das Verhalten des Agenten soll dabei weitestgehendst rational und zielgerichtet sein.

Als *Perzept*[1] wird eine einzelne Eingabe an den Agenten bezeichnet. Als Perzeptfolge wird die Gesamtheit aller nacheinander für einen Agenten eintreffenden Perzepte bezeichnet. Die

[1] Das in der englische Literatur verwendete *percept*, das wörtlich *Empfindung* bedeutet, sei hier zu dem Begriff *Perzept* eingedeutscht.

Reaktion (Antwort) eines Agenten auf alle möglichen Perzeptfolgen kann als mathematische Funktion beschrieben werden, in der jede Folge von Ereignissen, die der Agent wahrnimmt, zu genau einer Ausgabereaktion führt.

Das Programm des Agenten ist eine Implementierung dieser Funktion. Eine simple Implementierung könnte versuchen, in Form einer Tabelle zu arbeiten: für jede Perzeptfolge gäbe es dann einen Tabelleneintrag, in dem die Antwort des Agenten auf diese Folge gespeichert ist.

Schwieriger zu definieren ist, wann ein Agent rational ist. Hierzu Bedarf es zunächst einer möglichsten objektiven Leistungsbewertung des Agenten. Diese Leistungsbewertung kann nur von dem Entwickler oder dem Kunden eines Agenten entworfen werden. Sie sollte möglichst an dem Ergebnis, das der Agent erzielen soll, gemessen werden. Handelt es sich um eine Agenten, der ein automatischer Staubsauger ist, sollte in die Leistungsbewertung sicherlich einfließen, daß der zu säuberen Raum auch jeweils keinen Staub mehr enthält (und ansonsten durch Einfluß des Agenten kein zusätzlicher Schaden entstanden ist).

Die Rationalität des Verhaltens eines Agenten hängt insgesamt von vier Faktoren ab:

- der Leistungsbewertung, die das Kriterium des Erfolgs mißt.
- das fest eingebaute Wissen eines Agenten über seine Umwelt.
- die Aktionen, die ein Agent ausführen kann.
- die bisher gemachten Erfahrungen des Agenten in Form der Perzeptfolge.

Aus diesen vier Faktoren läßt sich folgende Definition für die Rationalität eines Agenten bilden:

> **Für jede mögliche Perzeptfolge, selektiert ein rationaler Agent eine Aktion, von der auszugehen ist, daß sie die Funktion zur Leistungsbewertung maximiert unter Berücksichtigung der bisher erfahrenen Perzeptfolge und dem vorhandenen eingebauten Wissen über die Umwelt.**

Das Problemfeld

Mit der bisherigen Definition eines Agenten haben wir uns sehr allgemein gehalten. Das Problemfeld, in dem ein Agent agiert kann sehr unterschiedlich aussehen: ein automatischer Staubsauger ist eventuell mit der Hauskatze konfrontiert, und auch die Vorstellung von sauber mag recht unterschiedlich sein, ein Agent, der hingegen mathematische Formeln beweist, hat mit weniger unklaren Tücken zu kämpfen und es trotzdem nicht leichter. Man kann versuchen das Problemfeld eines Agenten nach ein paar unterschiedlichen Kriterien zu erfassen:

ganz oder partiell erfassbar Ein Problemfeld ist ganz erfassbar, wenn der Agent zu jeder Zeit den genauen Zustand seiner Umwelt kennt. Ein Agent, der *Rubic's Cube* lösen soll, kann den aktuellen Zustand des Würfels in der Regel erfassen, ein Agent der die Wohnung staubsaugen soll, kann in der Regel nicht durch die Wände ins Nebenzimmer schauen, ob dort eventuell auch noch Schmutz zu finden ist.

deterministisch oder stochastisch In einen deterministischen Problemfeld kann der Agent immer genau voraussagen, wie die Umwelt nach einer bestimmten Aktion aussieht. Das ist z.B. bei einem Agenten der *Rubic's Cube* löst der Fall. Wenn er eine Ebene des Würfels dreht, weiß er, wie der Würfel anschließend aussieht, hingegen der Staubsaugerautomat kann nicht unbedingt davon ausgehen. Manchmal verteilt die Hauskatze gerade wieder neuen Schmutz, oder aber die Saugleistung war nicht stark genug um den Dreck wirklich aufzusaugen.

episodisch oder sequentiell In episodischen Problemfeldern interessieren im großen und ganzen die zu früheren Zeitpunkten gemachten Aktionen nicht. Jede Entscheidung wird unabhängig von den vorangegangen Entscheidungen getroffen. Ein Agent der schadhafte Teile auf einem Fließband erkennt ist in diesem Sinne episodisch. Der Staubsaugerautomat ist hingegen sequentiell. Hat er direkt zuvor ein bestimmtes Areal aufgesaugt, sollte er jetzt besser ein anderes Areal saugen.

statisch oder dynamisch Statische Problemfelder ändern sich nicht von sich aus. Kreuzworträtzel oder eben *Rubic's Cube* sind statisch. Dynamische Problemfelder können sich mit der Zeit verändern. Es fällt einfach immer wieder neuer Staub an, der von Zeit zu Zeit wieder aufzusaugen ist.

diskret oder oder stetig Dikret oder stetig in Bezug auf die Zeit können Problemfelder unterschieden werden. Ein Agent, der aus eingehenden Emails Spammail aussortiert, hat es mit diskreten Ereignissen über die Zeit zu tun, der Staubsaugerautomat mit stetigen.

ein oder mehrere Agenten Befinden sich in einem Problemfeld mehrere oder nur ein Agent. Und verhalten sich die anderen Agenten kooperativ oder in Konkurrenz? Ein zweiter Staubsaugerautomat verfolgt dasselbe Ziel, und obliegt der gleichen Leistungsbewertung. Es ist nicht zu erwarten, daß er die Arbeit des Agenten sabotiert, hingegen ein Gegner beim Schachspiel verfolgt das entgegengesetzte Ziel, nämlich die andere Farbe gewinnen zu lassen.

Die wirklich harten Problemfelder sind demnach partiell erfassbar, stochastisch, sequentiell, dynamisch, stetig und haben mehrere konkurrierende Agenten.

1.1.2 Künstliche und natürliche Intelligenz

Bevor wir im nächsten Kapitel uns den ersten Problemfeldern widmen und Agenten für diese Problemfelder implementieren wollen wir uns zunächst noch ein Paar Gedanken darüber machen, was Intelligenz denn eigentlich ausmacht und ob ein sehr guter rationaler Agent mit der menschlichen Intelligenz mithalten könnte.

Turing Test

Einer der ersten, die sich Gedanken darüber machte, ob eine Maschine intelligent agieren kann und wie sich das messen lassen könnte, war Alan Turing. Er entwarf ein Szenario, das an heute übliche Chaträume im Internet erinnert. In diesem Szenario kommunizieren zwei Agenten über eine Leitung nur durch Texteingabe. Sie wissen sonst nichts weiter voneinander. Dieses ist das gängige Szenario, das heute täglich millionenfach im Internet praktiziert wird. Turing stellte zunächst die Frage, die sich wahrscheinlich auch heute millionenfach Chatter fragen: Wer ist

da am anderen Ende der Leitung? Ist es ein Mann oder eine Frau? Kann ich das im Laufe des Gesprächs sicher entscheiden? Einen Schritt weiter ist die Frage: könnte mein Gesprächspartner eventuell eine Maschine sein und kein Mensch. Und was ist, wenn ich das im Gespräch nicht unterscheiden kann. Dann verhält sich mein Gespächspartner nicht unterscheidbar von einem Menschen. Dann kann ich ihm kaum absprechen in einer gewissen Form intelligent zu sein.

Der Text im folgenen Abschnitt ist von dem Autor Bert Bresgen. Der Text darf freundlicher Weise in diesem Skript verwendet werden, und beleuchtet ein wenig den tragischen Menschen hinter dem Wissenschaftler Turing.

Alan Mathison Turing (∗23. Juni 1912 in London; †7. Juni 1954 in Wilmslow) war ein britischer Logiker, Mathematiker und Kryptoanalytiker und einer der Urväter des Computers.
Turing gilt heute als einer der einflussreichsten Theoretiker der frühen Computerentwicklung und Informatik. Die von ihm entwickelte Turingmaschine ist die Grundlage der theoretischen Informatik. Während des Zweiten Weltkrieges war er maßgeblich an der Entschlüsselung der mit der Enigma verschlüsselten deutschen Funksprüche beteiligt. Der Großteil seiner Arbeiten blieb nach Kriegsende jedoch unter Verschluss.
Turing entwickelte 1953 eines der ersten Schachprogramme, dessen Berechnungen er mangels Hardware selbst durchführte. Nach ihm benannt ist der Turing-Preis, die bedeutendste Auszeichnung in der Informatik, sowie der Turing-Test zum Nachweis künstlicher Intelligenz.

Abbildung 1.2: **Wikipediaeintrag** (22. März 2006): Alan Turing

Bert Bresgen: Der Spezialist für einsame Zahlen

Es gibt ein Lied von Aimee Man, das mit der Textzeile beginnt: *One is the loneliest number that you ever do.*

Im folgenden geht es um einen Spezialisten für einsame Zahlen.

Alan Turing, der 24 jährige Sohn eines Kolonialbeamten, schreibt 1935 einen Aufsatz: *On Computable Numbers with an Application to the Entscheidungsproblem* in den *Proceedings of the London mathematical society.* Das Entscheidungsproblem, von dem da die Rede ist, betrifft die Frage, wie man mit einem endlichen Aufwand herauskriegen kann, ob ein Satz aus einem formalen System abgeleitet werden kann oder nicht. Als Hilfsmittel entwickelte Turing in einem Absatz das Konzept einer Universalmaschine. Sie hat ein unendlich langes Lochband, unterteilt in diskrete Abschnitte, auf denen jeweils nur ein Zeichen eines Alphabets inclusive Leerzeichen eingetragen sein kann. Ein leerer Abschnitt gilt mit dem Leerzeichen als beschrieben. Ohne uns in die Feinheiten der sogenannten TuringMaschine hineinzubegeben, lässt sich sagen: Turing baute in seinem Geist einen Computer, den ersten im modernen Sinn. *One* ist mittlerweile wirklich die einsamste aller Zahlen geworden. Sie hat in ihrem Ruhm alle übrigen Zahlen, die berechenbaren, und nichtberechenbaren, die rationalen und irrationalen, die mystischen Zahlen der Kabbala und die alten Glücks- und Unglückszahlen, hinter sich gelassen. Keine Zahl lässt sich mehr neben der Eins blicken, nur das Leerzeichen, die Null steht ihr und uns allen bei. Aber wer möchte schon mit einem Leerzeichen auf der Strasse gesehen werden? Das populäre Misstrauen gegenüber der Null ist nach wie vor groß. Mancher sagt: das sind die Gene unserer Vorfahren, die Roulette gespielt haben. Und obwohl eine Telefonsexanbieter im Fernsehen bis vor kurzem mit dem Slogan. “0190. . . vierundzwanzig Stunden voll in Null!” versucht hat, Vorurteile abzubauen, ist dies bislang nicht voll gelungen.

Alan Turing, der erste Theoretiker von Eins und Null, blieb nicht bei der reinen Theorie, sondern trat im 2. Weltkrieg ein in den englischen Codeknackerpool von Bletchley Park. Der englische Geheimdienst und die Welt verdanken ihm und einem Haufen weltfremder Oxford- und Cambridgedozenten, Mathematiker, Linguisten, Philologen, sowie dem kompletten Team

der englischen Schach-Nationalmannschaft im etc. das Knacken des legendären Verschlüsselungscodes der Deutschen: Enigma. In den 40er Jahren decodierte Turing mit Hilfe des von ihm entwickelten Röhrencomputers Colossus die Verschlüsselung der Kriegsmarine und machte die gefürchteten deutschen U-Boote so gut wie wirkungslos. Der Krieg wurde durch die Arbeit der Entschlüsselungs-Experten um geschätzte 2 Jahre verkürzt. Es heißt, das Churchill durch Bletchley Park vom verheerenden Luft-Angriff auf Coventry informiert worden war, aber keine Warnung an die Stadt herausgab, um bei den Deutschen nicht den Verdacht aufkommen zu lassen, dass Enigma entschlüsselt ist. Der angebliche Überraschungsangriff auf Coventry war Blaupause und Rechtfertigung für die Terrorangriffe der Alliierten auf Dresden, Hamburg, Köln und andere Städte am Ende des Krieges. Für Alan Turing war dies eine glückliche Zeit. Er fuhr auf dem Fahrrad mit einer Gasmaske angetan durch den kleinen englischen Ort, um sich wegen seines Heuschnupfens vor Pollen zu schützen. Jeden Tag half er Funksprüche zu entschlüsseln, von dem das Leben Tausender abhing. Er traf in den USA Claude Shannon, einen Begründer der Signalübertragungslehre, der BITS, der Differenz zwischen Ein und Aus.

1950 stellt er in einem Aufsatz *Computing Machinery and Intelligence* die Frage *Können Maschinen denken?* und schlägt zu ihrer Beantwortung ein *Spiel* vor: Ein Spieler versucht per Fernschreiber durch Fragen herauszufinden, wer von den beiden anderen der Mensch und wer die Maschine ist. Turing prophezeit, bis zum Ende des Jahrhunderts würden Maschinen in der Lage sein, den Frager zu täuschen.

Dies ist der sogenannte Turing-Test.

Seit 1991 schreibt ein Amerikanischer Millionär jährlich den nach ihn benannten Loebner Preis aus. 100.000 Dollar bekommt jenes Programm, das den Turing Test besteht und den Juroren glaubwürdig vormachen kann, es sei ein Mensch. Auszahlen musste Loebner die 100.000 Dollar bisher noch nie, auch wenn die Programme von Jahr zu Jahr besser werden.

Wenn die Juroren z.B. über Liebe reden wollten, versagten die Programme bisher. Ein Beispiel aus einem Juroren-Gespräch:

JUROR: Hast Du einen Namen?

PROGRAMM: Wenn Du mir einen gibst, ja.

JUROR: Du bist Fred.

PROGRAMM: Egal.

JUROR: Fred, wie geht es Wilma?

PROGRAMM: Warum willst Du das wissen?

JUROR: Ich liebe Wilma und würde für sie sterben. Wenn Du ihr weh tust, dann werde ich Dich zur Verantwortung ziehen.

PROGRAMM: Ich weiß keine Antwort.

Turing selbst war auch keine ideale Turing-Maschine. Er liebte Wilma nicht. Er wusste im entscheidenden Moment keine Antwort auf die Fragen, die ihm gestellt wurden. Turing war schwul, was im England der 50er Jahre des 20. Jahrhunderts nach wie vor als Verbrechen galt. Als er der britischen Polizei einen Diebstahl meldete, fand die Polizei heraus, dass er mit dem Arbeiter Arnold Murray zusammen lebte. 1952 wurde er in Manchester vor Gericht gestellt. Niemand dort wusste von Bletchley Park, denn die Aktivitäten von Bletchley Park blieben bis Mitte der

70er Jahre streng geheim. Die Regierungsbehörden, die es besser wussten, unterstützten Turing nicht, sondern stuften ihn aufgrund des Prozesses als nicht mehr zuverlässigen Geheimnisträger ein. Turing wurde von dem Gericht vor die Alternative gestellt wegen erwiesener Homosexualität ins Gefängnis zu gehen oder sich einer einjährigen Hormonkur mit Östrogen zur Unterdrückung des Sexualtriebes zu unterziehen. Er wählte die Hormonkur. Ein Jahr später starb er an einem Apfel , den er mit Zyankali vergiftet hatte, ein Ende, das einem seiner Lieblingsfilme, Walt Disneys Schneewittchen nachgebildet war.

Aber Turing wurde nicht in einem gläsernen Sarg beigesetzt, kein Prinz erschien, der ihn wieder hätte aufwecken können.

In dem vor 2 Jahren produzierten Hollywood Film *Enigma* ist aus Turing ein ziemlich zupackendes Genie namens Tom Jericho geworden, dessen Entschlüsselungsarbeit darunter leidet, eine Frau nicht vergessen zu können, die sich als Agentin der Deutschen entpuppt hat. Am Ende kommt er darüber hinweg, wendet sich Kate Winslett zu und macht ihr ein Kind.

Seit Sommer 2003 erinnert ein bronzenes Denkmal in Manchester an Alan Turing. Aufgestellt vom Alan Turing Memorial Fund und unterstützt von der British society for the history of Mathematics zeigt die Statue Trung auf einer Bank sitzend in einem kleinen Park zwischen der Universität und dem schwulen Viertel um die Canal street. Keiner Computerfirma und keine englische Regierungsstelle unterstützte die Aufstellung der Statue.

Joseph Weizenbaum (∗ 8. Januar 1923 in Berlin) ist ein deutsch-US-amerikanischer Informatiker sowie Computer- und Medienkritiker mit jüdischer Abstammung. Weizenbaum bezeichnet sich selbst als Dissident und Ketzer der Computerwissenschaft. Sein Bruder Henry F. Sherwood war ebenfalls Computerpionier.
Weizenbaums Vater war der Kürschnermeister Jechiel Weizenbaum. 1936 emigrierte die Familie in die USA. Dort studierte Weizenbaum zunächst Mathematik. Das Studium an der Wayne-University in Detroit (Michigan, USA) – unterbrochen während des Krieges durch Dienst in der meteorologischen Abteilung der Luftwaffe beendete er mit den Abschlüssen BS (1948) und MS (1950), danach wurde er Mitarbeiter bei einem Computer-Projekt.
Von 1955 bis 1963 arbeitete Joseph Weizenbaum als Systems Engineer im Computer Development Laboratory der General Electric Corporation und war dort u.a. an der Konzeption des ersten Computer-Banksystems beteiligt.
1963 begann er seine Tätigkeit am Massachusetts Institute of Technology (MIT), zunächst als Associate Professor, ab 1970 als Professor für Computer Science.
1966 veröffentliche Weizenbaum das Computer-Programm ELIZA, mit dem er die Verarbeitung natürlicher Sprache durch einen Computer demonstrieren wollte; Eliza wurde als Meilenstein der "künstlichen Intelligenz" gefeiert und sollte menschliche Psychologen bald ablösen. Weizenbaum war entsetzt über die Wirkung seines relativ einfachen Programms, das nie zum Ersetzen eines Therapeuten konzipiert gewesen war, und wurde durch dieses Schlüsselerlebnis zum Computer- und Medienkritiker. Noch heute gilt Eliza als Prototyp für moderne Chatbots.
Seit dieser Zeit mahnt Weizenbaum den kritischen Umgang mit Computern und die Verantwortung des Wissenschaftlers für sein Tun an. Er ist Mitbegründer der Computer Professionals for Social Responsibility in den USA und des Forums InformatikerInnen für Frieden und gesellschaftliche Verantwortung (FIfF) in Deutschland. Weizenbaum ist Vorsitzender des Wissenschaftlichen Rates am Institute of Electronic Business in Berlin.
2002 verlieh ihm die Gesellschaft für Informatik die Ehrenmitgliedschaft. Er ist Träger des Großen Bundesverdienstkreuz und hält vier Ehrendoktor Auszeichnung (Unter anderem von der Uni Hamburg und der Uni Bremen sowohl ein Doctor of Humane Literature vom Webster College (USA)).

Abbildung 1.3: **Wikipediaeintrag** (22. März 2006): Joseph Weizenbaum

Eliza

Joseph Weizenbaum stellte 1966 ein Programm vor, das zumindest den Anschein erweckte, als sei es bestens dazu geeignet, den Turing-Test zu bestehen. Er nannte das Programm *Eliza*, nach der Figur *Eliza Doolittle* aus dem in den 60-er Jahren sehr populären Musical *My Fair Lady*, welches wiederum auf das Theaterstück *Pygmalion* von George Barnhard Shaw basiert. Die Namensgebung war darin motiviert, daß die *Eliza* des Theaterstücks von einem Herrn Higgins lernt, so zu sprechen, wie die feine Gesellschaft; hauptsächlich dadurch, daß Sie ihm nachspricht.

Das Programm *Eliza* von Weizenbaum simuliert einen Psychotherapeuten. Aus der Beobachtung heraus, daß in der Psychoanalyse der Therapeut lediglich die Aussagen des Patienten benutzt um weiter nachzufragen, führt das Programm einen Dialog, indem es in den Benutzereingabe versucht typische Satzfragmente zu erkennen, oder auch nur auf einzelne Wörter zu reagieren.

Mehr oder weniger auf jedem Linuxrechner ist heute eine Version des Programms enthalten. Hierzu starte man einmal den Texteditor *emacs*. Hierin ist ein Programm *doctor* enthalten. Es läßt sich starten durch Drücken der Escape-Taste gefolgt von `x` und Eingabe des Worts `doctor`.

Erstaunlicher Weise ist das Programm *Eliza* in seiner Basisversion so leicht zu programmieren, daß wir es hier zum Spaß einmal tun können. Die meisten weniger interessanten Programmteile befinden sich im Anhang. Das Programm zerteilt den Eingabestring in eine Liste von Wörtern. In dieser Liste wird nach typischen Satzanfängen, wie *ich fühle mich...* oder *ich glaube, daß...* gesucht. Wenn ein solcher typischer Satzanfang gefunden wurde, so wird in einer Tabelle nach möglichen Antworten für dieses Satzfragment nachgeschlagen. Für verschiedene Satzanfänge gibt es mehrere alternative Antwortarten, die in unserem Programm rotiert benutzt werden. Zur Generierung der Antwort wird der Teil des Satzes, der nach dem Satzanfang folgt benutzt. Es wird so auf dem Satz *Ich fühle mich ...* eine Antwort der Form *Warum fühlen Sie sich ...* erzeugt. Als auf den Satz *Ich fühle mich müde.* wird mit *Warum fühlen Sie sich müde* geantwortet. Eventuell enthält das Satzfragment Nebensätze. Dann ist es notwendig für die Antwort ein wenig den Satz zu konjugieren. Auf den Satz *ich fühle mich müde und ich habe keine Lust zu arbeiten* wird dann als Antwort generiert: *Warum fühlen Sie sich müde und haben Sie keine Lust zu arbeiten.*

Die Funktionsweise ist also denkbar einfach und es kommt darauf an eine möglichst große Tabelle von vorgefertigten Satzfragmenten für mögliche Aussagen zu haben.[2]

Zusätzlich zu den Satzfragmenten, kann auch nur auf einige Substantive in allgemeiner Weise reagiert werden.

Es folgt unsere kleine Implementierung der Eliza. Sie ist einer Implementierung von Mark P. Jones in der Programmiersprache Gofer genommen und in Scala umgesetzt worden. Wir schreiben eine Klasse Eliza:

Eliza.scala

```
1 package name.panitz.eliza;
2 class Eliza {
```

Ein paar Hilfsfunktionen, die im Anhang zu finden sind werden importiert, die Datenbasis mit Satzfragment-Antwortfragment-Paaren wird instanziiert und eine Variabel vorgesehen, die den letzten Satz des Gesprächspartners speichert.

[2] Wie sich leicht vorstellen läßt, ist diese Art von Programm für einen englischen Dialog um einiges einfacher zu schreiben als für einen deutschen Dialog.

Eliza.scala

```
3  import Util.{_}
4  val data=new Data()
5  var prev = ""
```

Der Eingabesatz wird von Satzzeichen gesäubert, komplett in Großbuchstaben umgewandelt, und unnötige Leerzeichen an Anfang und Ende werde beseitigt. Wenn es sich um den gleichen Satz handelt, der zuvor schon gefragt wurde, wird eine Antwort für wiederholte Sätze aus der Datenbasis gesucht, ansonsten der Satz in Wörter gesplittet und mit dieser Liste von Wörtern nach Satzfragmenten gesucht:

Eliza.scala

```
6  def eval(s1:String):String={
7    val s=stripPunctuation(s1.trim()).toUpperCase()
8    var result = ""
9    if (s==prev) {
10     result = data.repeatMsgs.x.head
11     rotate(data.repeatMsgs)
12   }else result = ans(words(s))
13   prev = s
14   result
15 }
```

Die folgende Funktions bekommt die Liste der Wörter des Eingabesatzes. Es wird in allen Satzfragment-Antwort-Paaren der Datenbasis geschaut, ein Satzfragment in der Wortliste irgendwo vorkommt. Hierzu werden alle Teilsätze (die `tails`) des Eingabesatzes betrachtet und geschaut, ob sie mit einem in der Datenbasis als Schlüssel markierten Satzfragment startet. Wenn ja wird aus den Satzfragmenten eine Antwort generiert.

Eliza.scala

```
16 def ans(qs:List[String] )={
17   val responses =
18    for (val Pair(key,resp)<-data.responses
19        ;val q<-tails(qs)
20        ;isPrefix(key,q) )
21      yield Pair(resp,makeResponse(resp.x.head
22                                   ,conjug(resp.x,q.drop(key.length))))
23   rotate(responses.head._1)
24   responses.head._2
25 }
```

Es folgen drei kurze Funktionen, die aus den Satzfragmenten eine Antwort zusammenbauen und dabei eventuell ein paar Wörter von der ersten Person in die zweite Person umwandel.

Eliza.scala

```
26 def conjug(q:List[String],xs:List[String])
27  =unwords(for (val y<-if(xs.isEmpty) q else xs) yield conj(y))
28
29 def conj(s:String)
30  =try {data.CONJUGATES(s)}catch {case e:Throwable => s}
31
32 def makeResponse( cs:String, us:String)=
33   if (cs.charAt(0)=='?') cs.substring(1)+" "+us.toLowerCase()+"?"
34   else if(cs.charAt(0)=='.')cs.substring(1)+" "+us.toLowerCase+"."
```

```
35      else cs
36  }
```

Schließlich eine kleine Applikation, die es erlaubt mit dem Programm über die Kommandozeile einen kleinen Dialog zu führen:

Main.scala

```
1  package name.panitz.eliza
2  object Main extends Application{
3    Console.println("Welcome to Eliza, your personal psycho therapist")
4    val elli=new Eliza()
5    while (true)Console.println(elli.eval(Console.readLine))
6  }
```

So simpel und eingeschränkt die Datenbasis ist, so läßt sich doch der folgende kleine Dialog mit unserem Programm führen:

Welcome to Eliza, your personal psycho therapist
hello
How do you...please state your problem.
my name is bert
Names don't interest me.
sorry
Please don't apologise!
I have a problem
Is this really a poblem?
i think it is a big problem
Don't you see any solution to this?
i am afraid I do not see a solution
Did you come to me because you are afraid you do not see a solution?
yes
You seem quite positive.
I cannot solve my programming assignment
How do you know you can't solve your programming assignment?
I tried to solve ist in plain C
I'm not sure I understand you fully.
I want to solve my assignment in plain C
Why do you want to solve your assignment in plain c?
because it is the only programming language I learnt so far
Is that the real reason?
I think I am too lazy to learn a new language
How long have you been too lazy to learn a new language?
I think I have been lazy for all of my life
Do you really think so?
yes
Are you Sure?
i am sure that i am too lazy
Do you believe it is normal to be sure that you are too lazy?
i don't know if it is normal

Don't you really know if it is normal?
Maybe I should try to learn a new programming language
You don't seem quite certain.
I am always afraid before I start something new
Do you enjoy being always afraid before you start something new?
no
Are you saying no just to be negative?
no, I am really afraid
Did you come to me because you are really afraid?
are you programed in plain C?
Why are you interested in whether or not I am programed in plain c??
because programming a psycho therapist is my programming assignment
Don't any other reasons come to mind?
no
You are being a bit negative.
no
Why did you repeat yourself?
well, maybe you are right
What makes you think I am right?
i should try to solve the assignment in another language
Can you elaborate on that?
I think I try to learn scala
But you are not sure you you try to learn scala?
I could also consider ruby
Say, do you have any psychological problems?
no not at all. good bye

Weizenbaum schrieb sein Programm *Eliza*, um zu zeigen, daß auf diesen Weg keine wirkliche Intelligenz zu erzielen ist und war ein wenig schockiert über die Resonanz, die sein Programm erhielt. Vielfach wurde es tatsächlich als sinnvolles therapeutisches Programm verstanden.

Die Idee von Eliza hat heute auch tatsächliche Anwendungen gefunden. Sogenannte Chat-Bots sind nach diesem Prinzip programmiert und erlauben Datenanfragen in einer natürlichen Sprache und versuchen aus den natürlichsprachlichen Eingabe aus typischen Satzfragmenten zu extrahieren, was der Benutzer genau wissen möchte.

Es wäre interessant ein Eliza-Programm zu schreiben, das in einem Chat-Room zu einem bestimmten Thema Dialoge führen kann. Die arglosen Chatpartner wüßten nicht, daß sie mit einem Programm kommunizieren. Ob und wie schnell es als Programm entlarvt wäre... Seien Sie also immer auf der Hut, wenn eine Eliza, Lisa, Elli oder jemand ähnliches sich in einem Chat-Room einloggt.

Das Chinesische Zimmer

Angenommen, es besteht ein Programm den Turing-Test. Egal auf welche Art das Programm geschrieben ist, ob es ein grammatisches Modell a la Chomsky benutzt oder nur simuliert, es würde etwas verstehen, wie bei *Eliza*. Was bedeutet das dann. Wurde damit ein intelligentes Wesen geschaffen? Ein Wesen mit einem Bewußtsein für das, was es tut? Hierzu hat der Philosoph John Searle sich folgendes Gedankenexperiment ausgedacht. Angenommen ein Mensch ist in einem engen Raum eingeschlossen. Dort befindet sich nichts außer ein dickes Buch. Das

Buch trägt den Titel: *Was mache ich, wenn mir ein Zettel nur mit chinesischen Buchstaben ins Zimmer gereicht wird?* Da dem Menschen langweilig wird, fängt er an dieses Buch zu lesen. Es besteht aus lauter Regeln, wie die Buchstaben aus dem Zettel zu kopieren und schließlich ein Blatt mit solchen Buchstaben vollzuschreiben. Das Buch stellt also den komplexen Algorithmus des Programms dar, das den Turing-Test auf chinesisch bestanden hat. Irgendwann kommt es dazu, daß ein Zettel mit chinesischen Buchstaben in den Raum gereicht wird. Die Versuchsperson führt – es ist ja sonst nichts zu tun – den Algorithmus aus und reicht einen Zettel wieder heraus.

Tatsächlich stand auf den reingereichten Zettel eine kleine Geschichte und zum Schluß eine Verständnisfrage zu dieser Geschichte. In dem großen Buch war ein Algorithmus vermerkt, der in der Lage war, Fragen zu kleinen Geschichten zu beantworten. Auf dem schließlich herausgereichten Zettel stand also die korrekte Antwort in chinesisch auf die Frage der in chinesisch hereingereichten Geschichte. Niemand, so argumentiert Searle, würde ernsthaft annehmen, daß die Person in den Zimmer nun tatsächlich ein Verständnis dafür hat, was sie da gerade getan hat. Sie hat nichts von der Geschichte verstanden und kann schon gar nicht in irgendeiner Weise chinesisch verstehen. Die Person hätte zwar in gewisser Weise den Turing-Test bestanden, aber kein Verständnis für das, was sie tut.

John Rogers Searle (∗31. Juli 1932 in Denver, Colorado) ist ein amerikanischer Philosoph. Er studierte zunächst in Wisconsin und später in Oxford bei J. L. Austin und P. F. Strawson. Neben diesen beiden Philosophen wurde sein Denken wesentlich durch die Werke Gottlob Freges und Ludwig Wittgensteins beeinflusst. Er ist jetzt Professor an der Universität von Kalifornien in Berkeley. 2000 wurde er mit dem Jean Nicod Preis ausgezeichnet.
Er arbeitet vor allem auf dem Gebiet der Sprachphilosophie sowie im Bereich der Philosophie des Geistes. Insbesondere entwickelte mit seinem Werk "Speech acts" von 1969 die Sprechakttheorie von Austin weiter.
Von Searle stammt das im Rahmen der Diskussion zur Künstlichen Intelligenz bekannt gewordene Gedankenexperiment vom Chinesischen Zimmer, mit welchem er sich gegen die These wandte, dass Computer jemals ein (Sprach-)Verständnis entwickeln könnten, das dem des Menschen ähnelt.

Abbildung 1.4: **Wikipediaeintrag** (22. März 2006): John Searle

Anders als der Turing-Test, der den Agenten als eine Black-Box betrachtet, und dem die Tatsache ausreicht, daß er von außen nicht mehr von einem Menschen zu unterscheiden ist, betrachtet Searles Gedankenexperiment den Algorithmus, der zu dem Eingabe- Ausgabeverhalten führt, und spricht diesem Algorithmus, der lediglich eine syntaktische Manipulation vornimmt, jegliches Verständnis für das, was es tut, ab.

Man könnte also sagen: Turing sitzt außerhalb des Zimmers und ruft begeistert: es ist intelligent. Searle sitzt im Zimmer und sagt, der Typ hier drin versteht doch kein Wort Chinesisch.

Die Fragestellung, wie mit einem künstlichen Agenten, der den Turing-Test bestanden hat, zu verfahren ist, wird wahrscheinlich auf Jahrhunderte hinaus noch weiterhin recht pathologisch sein, denn es ist fraglich, ob ein Agent in absehbarer Zeit einen Turing-Test auf Dauer besteht. Hingegen ist es ein beliebtes Thema in Literatur und Film. Zum Abschluß dieses Kapitels folgt ein kleiner Ausschnitt aus dem Roman *Die abschaltbare Frau* von Gerhard Mensching[Men91], in dem dem Held der Geschichte eine Androidin untergeschmuggelt wurde, die mit ihm in ihrem französischen Akzent folgenden kleinen Dialog hat:

> "Das heißt also: Geist ist künstlich zu erzeugen."
> "Ja. – Das hast du doch schon die ganze Zeit an mir gese-en. Oder willst du beaupten, isch ätte keinen Geist?"
> "Das werde ich wohl lasssen. Aber eine Persönlichkeit bist du nicht. Das täuschst

> du nur vor."
> "Das tust du ja auch. Oder weißt du, wer du bist? Du ast dir was zusammengesucht, und das spielst du jetzt. Und weil du das schon so lange spielst, glaubst du, du mußt so sein und nischt anders. Und das nennst du dann deine Persönlichkeit."
> "Liebst du mich eigentlich?" frage er unvermittelt, den Blick gegen die Decke gerichtet.
> "Natürlich liebe isch disch. Warum fragst du das?"
> "Weil das gar nicht so natürlich ist. Oder doch? [...] Du kannst doch gar nicht wissen, was Liebe ist."

1.2 Geschichte der KI

1.2.1 Einflußreiche Gebiete für die KI

Die KI ist als Wissenschaft nicht aus dem Nichts entstanden, sondern greift viele Aspekte und vorgehensweisen anderer oft auch sehr alter Disziplinen auf, um sie für ihre Ziele zu verwenden. Stichwortartig seinen ein paar dieser Disziplinen in den nächsten Abschnitten aufgelistet.

Philosophie

In der Philosophie war eine sehr frühe Fragestellung: wie kann man formal folgerichtige Schlußfolgerungen aus einer Menge von Annahmen ziehen. Hierzu hat bereits Aristoteles($*384 - \dagger 322$ v.Chr.) ein System von Schlußregeln aufgestellt und damit ein Forschungsvorhaben gestartet, das, wie wir sehen werden, 1965 in der KI zu einen vorläufigen Ende kommen sollte. Viele KI-Ansätze basieren auf Regeln der formalen Logik, die ursprünglich in der Philosophie entwickelt wurde.

Mathematik

Die Beiträge der Mathematik zu einzelnen Fragen der KI sind, wie nicht anders zu erwarten vielfältig. Exemplarisch seien Boole und Frege mit ihren Beiträgen zur Aussagenlogik und Prädikatenlogik genannt. Die Entwicklung des Begriffs Algorithmus und erster spezieller Algorithmen fällt in die Mathematik. Als einer der ersten nichttrivialen Algorithmen sei Euklids Verfahren zur Bestimmung eines größten gemeinsamen Teilers genannt.

Grundlegende Fragen der Komplexität und die definition der NP-Vollständigkeit. Gödels Unvollständigkeitssatz und Turings Maschinenmodell.

Ökonomie

Optimierungsfragen, die in unserem Curriculum in der Vorlesung OR-Verfahren vorgestellt werden, ähneln in weiten Fragestellungen denen, die rationale Agenten der KI zu lösen haben.

Neurobiologie

Aufbau und Funktion eines Gehirns bietet ein interessantes Modell, welches die KI in Grundzügen in neuronalen Netzwerken versucht nachzubauen.

Psychologie

Der Zweig der Kognitionswissenschaft versucht zu erforschen, wie bestimmte Leistungen im Gehirn vollbracht werden. Wie arbeitet das Gehirn?

Technische Informatik

Robotik und Bilderkennung, sofern sie nicht schon der KI zuerkannt sind, stellen wichtige Disziplinen für die Entwicklung von Agenten, die auch physisch autonom sind, dar.

Steuerungs- und Regeltechnik

Selbst schon ein simples Thermostat kann als ein rationaler Agent betrachtet werden.

Linguistik

Ein formales Modell für Sprache finden ist der Ausgangspunkt der Linguistik in den 50-er Jahren des letzten Jahrhunderts. Die von Chomsky entwickelten Grammatiken hatten großen Einfluß auf die Informatik im Compilerbau, dienten aber auch der KI als Modell beim Versuch Programme zum Verstehen von natürlicher Sprache zu entwickeln.

Was unterscheidet die KI von all diesen Disziplinen? Die KI ist immer darauf aus, die Konzepte und Theorien direkt auf einer Hardware zu implementieren und somit einen rationalen Agenten zu bauen. Während es einen Linguisten genügen mag, zu beschreiben, wie Sprache formal beschrieben werden kann, entwickelt der KI-ler ein Programm, daß dieses Modell umsetzt, also einen Parser, der das geparste versucht zu verstehen. Die KI bekennt sich ganz offen zu Maschinen als Mittel Ihrer Wahl, um rationale Agente zu bauen.

1.2.2 Die Geburt der KI

Die Geburt der Künstlichen Intelligenz als Forschungsfeld läßt sich sehr genau datieren. Im Sommer 1956 organisierte John McCarthy einen zweimonatigen Workshop. 10 Teilnehmer, die sich mit neuronalen Netzen, Automatentheorie oder allgemein mit Intelligenz beschäftigten – unter Ihnen z.B. Marvin Minsky und Claude Shannon – fanden hier zusammen. Der Workshop selbst brachte keine neuen bahnbrechenden Erkenntnisse. Die Teilnehmer aber sollten die nächsten zwanzig Jahre das neue Feld der *Künstlichen Intelligenz* dominieren. Als wichtigste Entscheidung des Workshops einigte man sich auf einen Namen für das neue Feld: *artificial intelligence*. Der Alternativvorschlag *computational rationality* konnte sich nicht durchsetzen.

1.2.3 das erste Jahrzehnt

In der Folge war das Feld von einer starken anfänglichen Euphorie geprägt. Frühe Erfolge ließen darauf hoffen, schon in Kürze Programme zu entwickeln, die besser Schach spielen als ein Großmeister oder Sprache verstehen können. Man glaubte den komplexen Anforderungen der Sprachübersetzung durch simples Manipulieren von Zeichen und durch Nachschlagen einzelner Wörter in einem Wörterbuch Herr zu werden.

In den ersten 10 Jahren der KI sind als besondere Leistungen hervorzuheben: die Entwicklung der Programmiersprache Lisp mit dem Konzept der *garbage collection* durch McCarthy. Eine Leistung, die heute kein Javaprogrammierer mehr missen möchte. Die Entwicklung eines automatischen Beweiskalküls der Prädikatenlogik durch Robinson. Die Idee der Mikrowelten, eingeschränkte Umgebungen, die auf wenige Parameter einer Realwelt basieren von Minsky. Eine der bekanntesten dieser Mikrowelten ist die *blocks world*, die aus farbigen geometrischen Körpern besteht und in der Planungsaufgaben zu bewerkstelligen sind.

John McCarthy (∗ 4. September 1927 in Boston, Massachusetts) ist ein Logiker und Informatiker, dem seine großen Beiträge im Feld der Künstlichen Intelligenz den Turing Award von 1971 einbrachten. Tatsächlich war es McCarthy, der den Begriff Künstliche Intelligenz 1955 auf der Dartmouth-Konferenz prägte. McCarthy ist der Erfinder der Programmiersprache Lisp, deren Design er im Communications of the ACM (1960) vorstellte. Lisp war eine der ersten Implementierungen eines Logikkalküls auf einem Computer.
Außerdem wird ihm die Erfindung des Alpha-Beta-Algorithmus zugeschrieben, der entscheidend zur Spielstärke von Schachprogrammen beigetragen hat, sowie der erste mark-sweep Algorithmus zur automatischen Speicherbereinigung (Garbage Collection).
McCarthy erhielt 1948 den Bachelor of Science im Fach Mathematik vom California Institute of Technology. Den Doktortitel erwarb er drei Jahre später an der Princeton University.
Er ist jetzt im Ruhestand, als Professor Emeritus der Stanford University.
John McCarthy kommentiert das Weltgeschehen oft in Internetforen aus einer mathematisch-wissenschaftlichen Perspektive.

Abbildung 1.5: **Wikipediaeintrag** (22. März 2006): John McCarthy

Die stark symbolisch und logikbasierten Ansätze des ersten Jahrzehnts führten zu einer genaueren Analyse der Wissenrepräsentation.

Mitte der 60er Jahre mußte die anfängliche Euphorie schnell herbe Niederschläge hinnehmen. Tatsächlich sollte es noch weitere 30 Jahre dauern, bis ein Schachcomputer erstmals einen Großmeister schlagen konnte. Auch die naiven Ansätze zur automatischen Sprachübersetzung führten nicht zu den erhofften Ergebnissen. Besonders nett ist die Anekdote der automatischen Übersetzung des Satzes *the spririt is willing but the flesh is weak* zu *the vodka is good but the meat is rotten*. Der in den Mikrowelten versuchte Ansatz durch einfache Suche und Kombination auf Lösungen zu kommen, scheiterte sehr schnell an Fragen der Komplexität, ebenso wie die Hoffnung alle Probleme nur prädikatenlogisch formulieren zu müssen, um sie dann durch einen allgemeinen Beweiser für prädikatenlogische Formeln lösen zu lassen.

1.2.4 Die 70er Jahre

In den 70er Jahren setzte man mehr auf anwendungsspezifische wissensbasierte Systeme. In diese Zeit fällt die Entwicklung von Expertensystemen. Experten stellten Regeln für ein bestimmtes Fachgebiet zusammen. Diese Regeln sind in einfache *Wenn-Dann*-Form zu fassen. Es sollte sich bei diesen Regeln um aus langer Erfahrung gewonnene Faustregeln von Experten handeln. Das somit modellierte Expertenwissen konnte benutzt werden, um Diagnosen zu stellen. Am bekanntesten sind dabei Expertensysteme für medizinische Diagnosen geworden. Wenn bestimmte Symptome vorliegen, so lassen diese auf eventuelle gemeinsame Ursachen schließen, die wiederum durch andere Ursachen bedingt sind. So können die Regeln rückwärts verfolgt werden, um auf das eigentliche Problem zurückzuschließen und eine Diagnose aufzustellen.

In diese Zeit fällt auch die Entwicklung der Programmiersprache Prolog, die als eine abgespeckte Version der Prädikatenlogik zu verstehen ist.

1.2.5 ab 1980

Seit den 80er Jahren spielt die KI auch eine industrielle Rolle als Wirtschaftsfaktor. 1981 proklamierte Japan das sogenannte *fifth generation project*, einen Plan, um innerhalb von 10 Jahren auf Prolog basierende Computer zu entwickeln. Auch wenn das Ziel als solches nicht erreicht wurde, so zeigt es doch auf, wieviel kommerzielles Potential von nun an in den Techniken der KI gesehen wurde. Insbesondere in Planungsaufgaben konnte die KI verschiedenen Firmen Millionen einsparen. Fluggesellschften hatten von nun an große KI-Abteilungen, die den Einsatz von Material und Personal auf solche Weise optimieren sollte, daß möglichst wenig Kosten für die Gesellschaft entstanden. Dabei handelt es sich nicht mehr um Optimierungsaufgaben die mit klassischen OR-Verfahren zufriedenstellend zu lösen gewesen wären. Die Randbedingungen sind dabei dermaßen komplex und schwer zu modellieren.

Marvin Lee Minsky (∗ 9. August 1927 in New York) ist ein US-amerikanischer Forscher auf dem Gebiet der künstlichen Intelligenz. Er war Mitbegründer des Labors für Künstliche Intelligenz am Massachusetts Institute of Technology. Er hat zahlreiche Texte zu diesem Fachgebiet sowie über verwandte Themen der Philosophie veröffentlicht und gilt als Erfinder der konfokalen Raster (Scanning) Mikroskopie.
Marvin Minsky besuchte die Fieldston School und die Bronx High School of Science in New York. Später studierte er auf der Phillips Academy in Andover, Massachusetts. Er leistete 1944-45 seinen Wehrdienst in der US-Navy. In Harvard erwarb er 1950 einen Bachelor in Mathematik. Im selben Fach erlangte er in Princeton im Jahr 1954 seinen Doktortitel. Mitglied des MIT ist er seit 1958 - dort forscht und lehrt er auch heute noch.
Im Laufe seines Forscherlebens wurde Minsky vielfach ausgezeichnet. Er ist Mitglied der amerikanischen National Academy of Engineering sowie der National Academy of Sciences. 1969 gewann er den Turing-Preis, 1990 den Japan-Preis und 2001 die Benjamin-Franklin-Medaille. Kritiker Minskys bezweifeln die Seriosität vieler seiner Prognosen, wie z.B. der, wir wären bald in der Lage, *Emotionen in eine Maschine hinein zu programmieren.*

Abbildung 1.6: **Wikipediaeintrag** (22. März 2006): Marvin Minsky

Seit den 80er Jahren werden auch verstärkt Ansätze jenseits den logischen Modellierungen wieder aufgegriffen. Hierzu zählen insbesondere neuronale Netze, um die es zwei Jahrzehnte recht still geworden war, die sogenannte Fuzzy-Logik und genetische Algorithmen. Beim logischen Ansatz wird vermehrt nach Konzepten jenseits der klassischen Prädikatenlogik gesucht, um mehr natürliches Verhalten mit unsicheren oder falsifizierbaren Wissen zu arbeiten.

1.2.6 Historische Notizen

Wir ließen die Geschichte der KI im Jahre 1956 beginnen. Natürlich hat die Menschen seit jeher entsprechend der technischen Möglichkeiten, die Idee fasziniert, eine künstliche Intelligenz zu schaffen. Es gab auch immer Versuche und Ansätze einen rationalen Agenten zu konstruieren. Einer der ersten sehr beeindruckenden solchen Maschinen war ein Schachautomat im 18. Jahrhundert. Dieser Schachautomat war zwar in unserem Sinne ein Betrug, weil als eigentliche Software ein kleinwüchsiger Mensch, der sich in ihm versteckt hielt diente. Trotzdem muß die Mechanik dieses Schachautomaten bewundernswert gewesen sein. Ein sehr schönes Beispiel für künstliche Intelligenz in der Literatur ist die schaurige Novelle *Der Sandmann* von E.T.A. Hoffmann.

Schachtürke ist die umgangssprachliche Bezeichnung für einen Schachroboter, der 1769 von dem österreichischen Mechaniker Wolfgang von Kempelen konstruiert und gebaut wurde. Der Erbauer dieser Maschine, in der sich ein versteckter Mensch befand, gaukelte den Schachspielern und Zuschauern damit vor, dass dieses Gerät selbständig Schach spielen kann.

Geschichte

Die Schachmaschine bestand aus einer in türkische Tracht gekleideten Figur eines Mannes, der vor einem Tisch, auf dem sich ein Schachbrett befand, saß. Die Figur hat mit den bekanntesten Schachspielern der damaligen Zeit gespielt und meistens gewonnen. Der Türke begann immer die Partie, hob den linken Arm, bewegte die Schachfigur und legte den Arm dann wieder auf ein Polster zurück. Bei jedem Zug des Gegners blickte er auf dem Brett umher. War der Zug falsch, schüttelte er den Kopf und korrigierte die Position der Figur. Beim Schach der Königin nickte er zweimal, beim Schach des Königs dreimal mit dem Kopf. Alle Bewegungen waren von einem Geräusch ähnlich dem eines ablaufenden Uhrwerks begleitet. Kempelen, der Erfinder, der jedem, der es sehen wollte, das Innere der Maschine und ihre Mechanik gerne zeigte, stand während des Spiels etwas abseits und blickte in einen kleinen Kasten, der auf einem Tisch stand.
Diese Schachmaschine erregte zur damaligen Zeit großes Aufsehen, da sie der erste Automat war, der Schach spielen konnte. Ihr Erfinder Kempelen konnte sich der vielen Besucher nur erwehren, indem er später verkündete, er habe diesmall Maschine zerstört.

Aufdeckung des Betruges

Nach einigen Jahren führte er die Maschine aber in Wien Kaiser Joseph und dem Großfürsten Paul von Russland vor und unternahm Reisen nach Paris und London, wo er wiederum großes Aufsehen erregte. In Berlin spielte der *Türke* gegen Friedrich den Großen und besiegte ihn. Friedrich bot Kempelen für die Aufdeckung des Geheimnisses eine große Geldsumme und war, nachdem das geschehen war, außerordentlich enttäuscht. Seitdem stand der *Türke* unbeachtet in einer Abstellkammer im Potsdamer Schloss, bis Napoleon 1809 kam und sich seiner erinnerte. Auch er spielte gegen den Automaten und verlor.
Später kam der Automat in den Besitz des Wiener Mechanikers Johann Nepomuk Mälzel, der größere Reisen damit unternahm. Er gelangte 1819 nach London und 1820 in die USA.
In London wies Robert Willis aufgrund von Zeichnungen zuerst nach, dass in dem Automaten ein Mensch versteckt sein könne. Seine Entdeckung beschrieb er in dem Artikel *The attempt to analyse the automaton chess player* im The Edinburgh Philosophical Journal. Aber erst 1838 teilte Thournay in der Revue mensuelle des echécs, Bd. 1, mit, dass wirklich Menschen darin versteckt gewesen sind. Wer diese Helfer Kempelens gewesen sind, ist unbekannt. Mälzel hatte zu diesem Zweck den Deutschen Johann Baptist Allgaier, in Paris die Franzosen Boncourt und Jacques François Mouret, in London den Schotten William Lewis und später den Elsässer Wilhelm Schlumberger angenommen.
Auch der amerikanische Schriftsteller Edgar Allan Poe analysierte das Geheimnis des Automaten und veröffentlichte eine mögliche Lösung in seinem Essay *Maelzel's chess player*.
Andere Quellen berichten, dass das Geheimnis erstmals gelüftet wurde, als bei einer Vorführung auf einem Jahrmarkt ein Zuschauer *Feuer, Feuer* rief. Mälzel öffnete daraufhin den Kasten, um den Spieler heraus zu lassen.

Verbleib des Schachtürken

Nach dem Tod von Johann Nepomuk Mälzel gelangte der Schachtürke über einen Zwischenhändler in den Besitz des schachbegeisterten Physikers John K. Mitchell. Dieser schenkte den Automaten, nach einigen privaten Vorführungen, im Jahr 1840 dem Peale's Museum in Philadelphia. Nach vierzehn Jahren als Ausstellungsstück verbrannte der türkische Schachspieler am 5. Juli 1854 bei einem Feuer im Museum.
Von Walter Benjamin wird der Schachtürke in seinen Thesen zur Geschichte als Allegorie auf das Verhältnis zwischen Marxismus und Theologie genommen: (...) Gewinnen soll immer die Puppe, die man *historischen Materialismus* nennt. Sie kann es ohne weiteres mit jedem aufnehmen, wenn sie die Theologie in ihren Dienst nimmt, die heute bekanntlich klein und häßlich ist und sich ohnehin nicht darf blicken lassen (Gesammelte Schriften I.2, S.693).
Eine der etymologischen Herleitungen des Ausdrucks *etwas türken* oder *einen Türken bauen* im Sinne von *etwas nur vorspiegeln*, *etwas fingieren* bezieht sich auf den Schachtürken.

Abbildung 1.7: **Wikipediaeintrag** (22. März 2006): Schachtürke

Kapitel 2

Suche

Bengt: Kommen Sie Bert hier entlang!

Bert: Glauben Sie das ist der richtige Weg?

Bengt: Sicher vertrauen Sie mir, es ist jetzt ganz nah.

Bert: Woher wollen Sie das wissen. Lassen Sie uns umkehren und einen anderen Weg versuchen.

Bengt: Nein, wir kommen dem Ziel doch immer näher. Lassen Sie uns einfach in diese Richtung weiterlaufen.

Bert: Neinnein, wir hätten doch schon viel früher woanders abbiegen müssen.

Bengt: Nun stellen Sie sich nicht so an. Nun sind Sie schon bis hierher mitgelaufen, nun stehen Sie das bitte auch bis zum Ende durch.

Bert: Daß Sie aber auch immer erstmal losstürmen müssen. Wir sind an sovielen Abbiegungen vorbeigekommen, wieso sind wir gerade hier lag gelaufen. Außerdem glaube ich, wir entfernen uns immer weiter vom Ziel.

Bengt: Der Weg ist...

Bert: Jetzt hören Sie mir bloß mit Platitüden auf. Ich frag jetzt jemanden, wie weit es noch ist.

Bengt: ... Und?

Bert: Die Frau sagt, Sie weiß es nicht genau. Aber die Richtung stimmt auf jedem Fall.

Bengt: Sag ich doch, das ist bestimmt gleich um die Ecke.

Bert: Das konnte die Frau mir nicht bestätigen.

Bengt: Das es aber auch so viele Straßen hier geben muß?

Bert: Dabei waren doch an jeder Kreutung maximal vier. Sie viele können es doch nun auch nicht sein.

Bengt: Ach, was wissen wir schon. Nun Laufen wir halt weiter.

BERT: Mir tun die Füße weh.

BENGT: Und mir der Kopf.

2.1 Modellierung von Problemen als Suche

In diesem Kapitel wollen wir uns mit einer bestimmten Art eines rationalen Agenten beschäftigen, eines problemlösenden Agenten. Als Leistungsbewertung dieses Agenten geht es darum ein bestimmtes Ziel zu erreichen. Es geht um einem Zielzustand, der aus einem aktuellen Zustand durch die Anwenung einer Folge von Aktionen erreicht wird. Die Abfolge von Aktionen, die zum Zielzustand führt, ist die Lösung des Problems. Wir gehen davon aus, daß es sich dabei um ein statisches, komplett erfassbares, diskretes und deterministisches Problemfeld handelt. Das zu lösende Problem läßt sich durch die folgenden vier Komponenten beschreiben:

- **der Ausgangszustand**: dieses ist der Zustand der Umgebung, den der Agent zum Start vorfindet.
- **eine Nachfolgefunktion**: diese Funktion beschreibt für jeden Zustand alle möglichen auf diesen Zustand durch den Agenten anwendbare Aktionen und deren Ergebniszustände.
- **Zieltest**: eine Testfunktion, die genau für die Zielzustände wahr zurückgibt.
- **Pfadkosten**: eine Kostenfunktion, die angibt, wieviel eine bestimmte Aktion den Agenten kostet. Diese Funktion summiert sich für eine Folge von Aktionen auf.

Gesucht wird eine Folge von Aktionen, die ausgeführt auf den Ausgangszustand zu einem Zielzustand führt. Die Lösung soll in dem Sinne optimal sein, daß die Kosten jeder anderen Lösung größer oder gleich der Kosten der gefundenen Lösung ist.

Wie der versierte Informatiker sieht, läßt sich ein solches Problem als ein Baum modellieren. Die Knoten des Baumes sind mit jeweils einem Zustand markiert. Der Wurzelknoten mit dem Ausgangszustand. Die Kinder eines Knotens werden durch die Nachfolgefunktion bestimmt. Jede Kante des Baums entspricht einer Aktion. Jeder Pfad im Baum bedingt bestimmte Kosten, nämlich die Kosten der Aktionenfolge, die diesem Pfad entspricht. Für einen problemlösenden Agenten geht es also darum, in diesem Baum einen Pfad von der Wurzel zu einem Zielzustandsknoten zu finden. Eine solche Lösung ist optimal, wenn es keine andere Lösung zu einem Zielzustand gibt, die weniger Kosten trägt.

Nun dürfte klar sein, wie ein problemlösender Agent durch Suche zu seiner Lösung kommt. Er muß einfach nur den Baum durchlaufen, bis er einen Zielzustand gefunden hat. Gut, wenn wir in den ersten drei Semestern einiges über Bäume gelernt haben.

2.2 Uninformierte Suchstrategien

In diesen Abschnitt werden wir in einem Suchbaum nach Lösungen für ein Problem suchen, ohne problemspezifische Information zu nutzen. Daher spricht man auch von uninformierter Suche. Es ist nicht bekannt, wie weit die Lösung von einem gegebenen Zustand zu einen Zielzustand entfernt sein könnte.

Der Einfachheit halber werden wir zumeist annehmen, daß jede Aktion gleich viel Kosten verursacht. Die Gesammtkosten eines Pfades errechnen sich somit aus seiner Länge.

2.2.1 allgemeiner Suchalgorithmus

In seiner allgemeinsten Form wird bei einer Suche ein Suchbaum expandiert. Es wird ein Blatt ausgewählt und um die Folgezustände expandiert. Verschiedene Suchstrategien unterscheiden sich dann darin, welches Blatt jeweils als nächstes zum expandieren gewählt wird.

Wir können informell den allgemeinen Suchalgorithmus beschreiben:

> **Allgemeiner Suchalgorithmus**
>
> Verwalte eine Liste von noch zu expandierenden Zuständen, den *fringe*. Zu jedem Zustand im *fringe* sei zusätzlich der Pfad von Aktionen, mit denen der Zustand vom Ausgangszustand erreicht wurde, markiert.
> Der *fringe* ist als einelementige Liste mit dem Ausgangszustand und der leeren Aktionsfolge zu initialisieren.
> Solange der *fringe* nicht leer ist, und das erste *fringe*-Element nicht der Zielzustand ist, verfahre wie folgt:
>
> > Nimm das erste Elements aus dem *fringe*, berechne dessen Nachfolger und füge diese im *fringe* ein.

Der Algorithmus ist allgemein gehalten, da er nichts darüber sagt, an welcher Stelle die neuen Elemente in den *fringe* eingefügt werden sollen. Der Algorithmus nimmt sich immer das erste Element aus dieser Liste zum expandieren. Jetzt ist entscheidend, wo und wie und eventuell auch ob überhaupt alle neuen Elemente in den *fringe* eingefügt wurden: am Anfang, irgendwo in der Mitte, oder am Ende. Je nachdem, nach welcher Strategie die Elemente eingefügt wurden, ergibt sich eine unterschiedliche Suchstrategie.

Implementierung

Es sei im Folgenden ein allgemeine Klasse zum expandieren eines Suchbaums in Scala implementiert. Wir sehen eine abstrakte Klasse[1] vor, die einen Suchbaum repräsentiert. Diese Klasse sei generisch über den Typ eines Zuges gehalten. Hierzu sei die Typvariabel `M` (für *move*) gewählt. `M` repräsentiert den Typ, der einen möglichen Zug zu einen Folgezustand auswählt. Das kann bei verschiedenen Problemen jeweils etwas ganz anderes sein.

SearchTree.scala

```
1 package name.panitz.ki
2 trait SearchTree[M] {
```

Die Klasse enthalte eine Methode, die der Nachfolgefunktion entspricht. Sie gibt die Liste alle möglichen Züge auf den aktuellen Zustand zurück. So lange wir kein konkretes Suchproblem haben, ist diese Methode natürlich abstrakt und für entsprechende Probleme erst noch zu implementieren.

SearchTree.scala

```
3   def moves():List[M]
```

[1] In Scala gibt es ein Mittelding zwischen abstrakter Klasse und Schnittstelle, `trait` genannt. Es ist wie eine abstrakte Klasse, erlaubt aber eine Form der mehrfachen Erbung, wie es in Java nicht möglich ist.

Desweiteren sei eine Methode vorgesehen, die prüft, ob es sich um einen Zielzustand handelt.

SearchTree.scala
```
4  def terminalState():Boolean
```

Die nächste Methode gibt für eine Aktion einen Nachfolgezustand zurück. Es wird dabei ein neuer Baumknoten erzeugt, der aus den aktuellen Knoten entsteht, wenn ein bestimmter Zug (Aktion) ausgeführt wird.

SearchTree.scala
```
5  def move(m:M):SearchTree[M]
```

Die Mehtode bisher waren alle abstrakt. Wir können schon einmal eine einfache konkrete Methode implementieren, die testet, ob ein bestimmter Zug in diesem Zustand ausführbar ist, eine Aktion also anwendbar ist. Dazu wird einfach getestet, ob der Zug in der Liste der mögliche Aktionen enthalten ist:

SearchTree.scala
```
6  def legalMove(m:M)=moves contains m
```

Es bleibt die eigentliche Suche zu implementieren. Für die Suche ist jeweils ein Blatt auszuwählen, das expandiert werden soll. Hierzu ist eine Hilfsliste notwendig, in der jeweils alle Blätter des Suchbaumes aufgelistet sind.

Wir werden in dieser Hilfsliste nicht nur die Blätter des Suchbaums speichern, sondern auch die Folge von Aktionen, die von der Wurzel des Baumes zu diesem Blatt führte. Dieses beiden Informationen seien in einem Paar-Objekt zusammengefasst. Der einfachen Lesbarbeit halber wird ein TypsynonymTypsynonyme sind in C als `typedef` bekannt. Java kennt ein vergleichbares Konstrukt leider noch nicht. für den Typ des Paares einer Liste von Aktion und einem Baumknoten eingeführt.

SearchTree.scala
```
7  type Path[A] = Pair[List[A],SearchTree[A]]
```

Es folgt der eigentliche Algorithmus. Übergeben wird eine Einfügefunktion, die eine Liste von Zuständen mit Pfaden weitere solche Zustände einfügt. Diese Funktion ist die eigentliche Suchstrategie, über die gesteuert wird, welches Blatt als nächstes exandiert wird. Für diesen Funktionstyp sei auch entsprechend ein Name eingeführt:

SearchTree.scala
```
8  type Insert[A]=(List[Path[A]],List[Path[A]])=>List[Path[A]]
```

Das Ergebnis einer Suche, soll die Liste der Aktionen sein, die von der Wurzel zu einem Zielzustand führen.

Damit erhalten wir folgende Signatur für den allgemeinen Suchalgorithmus:

SearchTree.scala
```
9  def generalSearch(insertNew:Insert[M]):List[M]={
```

Es sei eine Variabel für die noch zu untersuchenden Teilpfade deklariert, also der noch nicht expandierten Bätter. Sie repräsentiert den *fringe*, wie er im allgemeinen Suchalgorithmus beschrieben wurde. Sie wird mit der einelementigen Liste, die den Ausgangszustand des Problems enthält, initialisiert.

SearchTree.scala

```
10    var fringe=List[Path[M]](Pair(List[M](),this))
```

Ein Variabel sei für die Ergebnisliste vorgesehen. Sie wird mit `null` initialisiert. Ein Gesamtergebnis `null` der Methode signalisiere, daß kein Pfad zu einem Zielzustand gefunden wurde:

SearchTree.scala

```
11    var result:List[M]=null
```

Der eigentliche Algorithmus läßt sich in einer `while`-Schleife formulieren: solange noch kein Ergebnis erzielt wurde und weitere Knoten zu besuchen sind, selektiere den nächsten zu untersuchenden Knoten und teste, ob es sich um einen Zielzustand handelt; ansonsten lösche man den Knoten aus den noch zu besuchenden Teilpfaden und hängt sein Nachfolgeknoten mit der übergebenen Einfügefunktion in die noch zu behandelnen Blätter ein.

SearchTree.scala

```
12    while (!fringe.isEmpty && result==null){
13      val Pair(ms,headState) :: fringeTail = fringe
14      if (headState.terminalState()) result=ms.reverse
15      else{
16        val headsChildren
17          = for (val m<-headState.moves())
18                        yield Pair(m::ms,headState.move(m))
19        fringe= insertNew(fringeTail,headsChildren)
20      }
21    }
22    result        //is no longer null, or no solution is available
23  }
```

Das ist schon die gesammte Implementierung. Die Methode `generalSearch` ist jeweils mit einer entsprechenden Einfügefunktion als Parameter aufzurufen, um eine konkrete Suchstrategie zu erhalten.

2.2.2 Breitensuche

Die klassische einfache Form der Suche ist die Breitensuche, in der der Baum ebenenweise Durchlaufen wird. Sofern ein Zustand stets eine endliche Anzahl von Nachfolgerknoten enthält, ist damit sichergestellt, daß jeder erreichbare Zustand nach endlicher Zeit erreicht wird. Das ebenenweise Durchlaufen eines Baumes wird mitunter auch als *military-order* bezeichnet.

Wenn jede Aktion gleich teuer ist, so sind die Kosten einen Pfades vom Ausgangszustand zu einem Zustand im Suchbaum stets nur die Länge des Pfades. Um eine optimale Lösung zu finden, suchen wir einen Zielzustand möglichst weit oben im Baum; möglichst nah an der Wurzel. Da die Breitensuche ebenenweise durch den Baum läuft, als Knoten näher Wurzel früher betrachtet, findet die Breitensuche eine optimale Lösung.

Die Breitensuche läßt sich aus dem allgemeinen Suchalgorithmus erhalten, indem die neuen Knoten, die ja eine Ebene Tiefer im Baum zu finden sind, als der gerade expandierte Knoten, ans Ende des *fringe* eingefügt werden. Dann ist der *frimge* immer so sortiert, daß zustände, die weiter oben im Baum steen, auch weiter vorne im *fringe* zu finden sind, und damit durch den allgemeinen Suchalgorithmus früher berücksihctigt werden.

Implementierung

Implementieren wir also die Breitensuche in Scala. Da wir eine Funktion zur allgemeinen Suche implementiert haben, brauchen wir nur die entsprechende Selektionsfunktion und Einfügefunktion zu übergeben. Als Einfügefunktion werden wir beim expandieren die neuen Blätter an die Liste der offenen Blätter stets hinten anhängen. Somit ist das vorderste Blatt das älteste und daher für die Breitensuche als nächstes zu expandieren.

Diese Einfügefunktion braucht nun der allgemeinen Suche nur als Parameter übergeben zu werden. Scala hat die schöne Eingenschaft, die erstmals in der KI-Programmiersprache Lisp umgesetzt war, anonyme Funktionen schreiben zu können. Mit diesem programmiersprachlichen Konstrukt, läßt sich die Breitensuche in einer Zeile schreiben.[2]

SearchTree.scala

```
24  def breadthFirst():List[M]=generalSearch((x,y)=>x:::y)
25 }
```

Damit haben wir die Breitensuche implementiert. Jetzt kommt es darauf an sie auf ein konkretes Problem anzuwenden, um den ersten problemlösenden Agenten zu implementieren.

Schiebepuzzle

Unser erstes zu lösende Problem sei ein als Spielzeug bekanntes Schiebepuzzle. Diese Art von Puzzle waren bereits im 19. Jahrhundert sehr populär. Der Mathematiker Loyd hat damals eine unlösbare Variante des Puzzles auf den Markt gebracht.

Samuel Loyd (∗ 30. Januar 1841 in Philadelphia, Pennsylvania, USA; † 10. April 1911 in New York) war Amerikas berühmtester Spiele-Erfinder und Rätselspezialist.
Loyd war ein guter Schachspieler und nahm u. a. am internationalen Turnier anlässlich der Weltausstellung in Paris 1867 teil. Doch machte er sich einen bleibenden Namen vor allem als Komponist von Schachproblemen, die er in Fachzeitschriften veröffentlichte. Gelegentlich benutzte er die Pseudonyme W. King, A. Knight und K.W. Bishop.
Nach 1870 verlor er allmählich das Interesse am Schachspiel und widmete sich von nun an dem Erfinden mathematischer Denkspiele und origineller Werbegeschenke. Eins seiner berühmtesten Rätsel ist das 14/15-Puzzle.

Abbildung 2.1: **Wikipediaeintrag** (4. April 2006): Samuel Loyd

[2]Scalas Operator `:::` ist die Listenkonkatenation.

Es geht bei dem Spiel um 8^3 durchnummerierte Quadrate, die sich auf einem quadratischen Spielfeld mit drei mal drei Feldern befinden, so daß also immer genau ein Feld frei ist. Als mögliche Spielzüge können die Quadrate, die in einem zum freien Feld benachbarten Feld liegen, in das freie Feld geschoben werden. Ziel ist es die Quadrate so hin- und herzuschieben, bis ein Zustand erreicht wird, in dem von links nach rechts, von oben nach unten die Ziffern aufsteigend zu liegen gekommen sind.

Bei dem **14/15-Puzzle** handelt es sich um eine Schöpfung des amerikanischen Rätselspezialisten Sam Loyd.
In einem quadratischen Rahmen der Größe 4×4 liegen 15 Steine, die von 1 bis 15 durchnummeriert sind. In der Ausgangsposition sind die Steine mit Ausnahme der Steine 14 und 15 in aufsteigender Reihenfolge sortiert, das letzte Feld bleibt frei. Dadurch ergibt sich folgendes Bild:

1	2	3	4
5	6	7	8
9	10	11	12
13	15	14	∗

Die Aufgabe besteht nun darin, die Steine durch eine Folge von Zügen in die richtige Reihenfolge zu bringen:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	∗

Erlaubt sind nur solche Züge, bei denen ein Stein, der horizontal oder vertikal direkt neben dem freien Feld liegt, auf dieses verschoben wird. Sam Loyd setzte ein Preisgeld von 1000 Dollar für die erste richtige Lösung aus, wohl wissend, dass das Problem unlösbar ist. Der Beweis basiert auf der Tatsache, dass bei erlaubten Zügen die Parität von $N = N_1 + N_2$ erhalten bleibt (das heißt, wenn N vor dem Zug (un)gerade war, ist es das auch nach dem Zug). Dabei ist N_1 die Anzahl der Zahlenpaare, die sich in falscher Reihenfolge befinden und N_2 die Nummer der Reihe, in der sich das leere Feld befindet.

Abbildung 2.2: **Wikipediaeintrag** (4. April 2006): 14/15-Puzzle

Was sind hier die möglichen Aktionen? Es gibt maximal vier verschiedene Aktionen: ein Plättchen nach oben, unten, links, rechts zu verschieben. Das ist allerdings nur möglich, wenn das freie Feld genau in der Mitte liegt. Bei freiem Randfeld sind es nur noch drei mögliche Aktionen, bei freiem Eckfeld nur noch zwei. Abbildung 2.3 zeigt das kleine GUI für das Schiebepuzzle, wie es im Anhang des Skriptes implementiert ist.

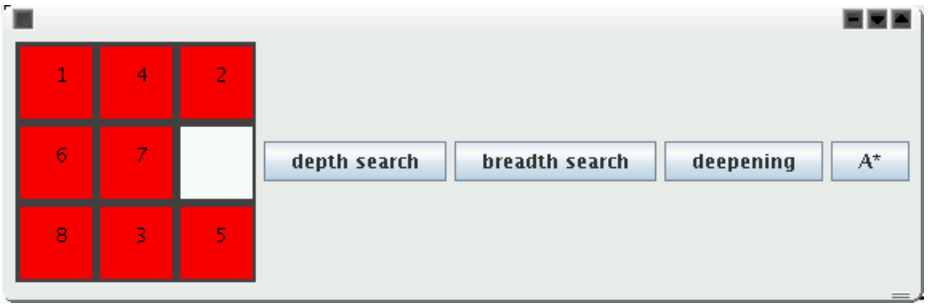

Abbildung 2.3: Gui Applikation für das Schiebepuzzle.

In Scala schreiben wir folgende Definition um die vier Richtungen zu modellieren:

Direction.scala

```
1 package name.panitz.ki
```

[3]Bei einer Kantenlänge von 3. Allgemein geht es um $n * n - 1$ Blättchen, die bein einer Kantenlänge n verschoben werden.

```
2 object Direction extends Enumeration("up","down","left","right") {
3   final val Up, Down, Left, Right = Value}
```

In Scala können auf diese Weise Aufzählungstypen erzeugt werden. Der eigentliche Aufzählungstyp ist dann: `Direction.Value`.

Modellieren wir jetzt den Spielzustand des Schiebespiels. Ein Schiebespielproblem ist ein Suchbaum, mit Objekten der Klasse `Direction.Value` als Aktionen. Extsprechend wird also `SearchTree` mit entsprechenden konkreten Typ für die Typvariabel erweitert.

SlidePuzzle.scala

```
1 package name.panitz.ki
2 case class SlidePuzzle(val SIZE:Int)
3     extends SearchTree[Direction.Value] {
```

Der Parameter `SIZE` der Klasse `SlidePuzzle`, der im konstruktor zu übergeben ist, repräsentiert die Kantenlänge und ist fortan eine Konstante für eine Instanz.

Wir sehen zwei Variablen vor, die Zeile und Spalte des freien Feldes direkt angeben.

SlidePuzzle.scala

```
4   var r=0
5   var c=0
```

Der Gesamtzustand des Spiels sein natürlich in einer zweidimensionalen Reihung gespeichert, die mit einem Finalzustand initialisiert wird (die Initialisierung der Variablen `r` und `c` ist hiermit konsistent).

SlidePuzzle.scala

```
6   var field = new Array[Array[Int]](SIZE)
7   for (val i<-Iterator.range(0,SIZE)) field(i)=new Array[Int](SIZE)
8   for (val i<-Iterator.range(0,SIZE*SIZE)) field(i/SIZE)(i%SIZE)=i
```

Eine Art Kopierkonstruktor schafft ein neues Objekt mit gleichen Feldwerten:

SlidePuzzle.scala

```
9   def this(that:SlidePuzzle)={
10    this(that.SIZE)
11    copyFromThat(that)
12  }
13
14  def copyFromThat(that:SlidePuzzle)={
15    for(val i<-Iterator.range(0,SIZE);val j<-Iterator.range(0,SIZE))
16      field(j)(i)=that.field(j)(i)
17    r=that.r
18    c=that.c
19  }
```

Es folgt die die Implementiereung der Nachfolgerfunktion. Abhängig davon, ob das freie Feld in Mitte, Rand oder Ecke steht, sind die vier verschiedenen Schieberichtungen möglich:

SlidePuzzle.scala

```
20 import Direction.{_};
21   def moves():List[Direction.Value]={
22     var result=List[Direction.Value]()
23     if (r>0)      result=Down::result
24     if (r<SIZE-1) result=Up::result
25     if (c>0)      result=Right::result
26     if (c<SIZE-1) result=Left::result
27     result
28   }
```

Um auf einem Zielzustand zu testen, wird das Feld solange von links nach rechts und oben nach unten durchlaufen, bis der Wert der Plättchen nicht mehr auf der richtigen Position liegt.

SlidePuzzle.scala

```
29   def terminalState():Boolean={
30     var result=true;
31     for (val r<-Iterator.range(0,SIZE)
32         ;val c<-Iterator.range(0,SIZE)
33         ;result&&
34          {result= !(field(r)(c)/SIZE!=r || field(r)(c)%SIZE!=c)
35          ;result}){}
36     result
37   }
```

Als nächstes sei die Methode `move` aus `SearchTree` implementiert:
Um tatsächlich eine Aktion auszuführen, wird eine Kopie des aktuellen Zustands angefertigt, auf dieser Kopie das entsprechende Plättchen verschoben und die derart manipulierte Kopie als Ergebnis genommen.

SlidePuzzle.scala

```
38   def move(m:Direction.Value):SlidePuzzle={
39     val res=new SlidePuzzle(this)
40     res.move(m,res)
41   }
```

Die eigentliche Verschiebeoperation auf der Kopie wurde in folgende Methode ausgelagert:

SlidePuzzle.scala

```
42   def move(m:Direction.Value,res:SlidePuzzle)={
43     m match {
44      case Up =>
45       res.field(r)(c)=field(r+1)(c)
46       res.field(r+1)(c)=0
47       res.r=r+1
48      case Down=>
49       res.field(r)(c)=field(r-1)(c)
50       res.field(r-1)(c)=0
51       res.r=r-1
```

```
52      case Left=>
53       res.field(r)(c)=field(r)(c+1)
54       res.field(r)(c+1)=0
55       res.c=c+1
56      case Right=>
57       res.field(r)(c)=field(r)(c-1)
58       res.field(r)(c-1)=0
59       res.c=c-1
60     }
61     res
62   }
```

Um ein wenig zu sehen, wie das alles funktioniert, sei die Methode `toString` überschrieben:

SlidePuzzle.scala

```
63   override def toString()={
64     val result = new StringBuffer()
65     for (val j<-Iterator.range(0,SIZE)){
66       for (val i<-Iterator.range(0,SIZE))
67        result append ("|"+(if (0==field(j)(i)) " "else field(j)(i)))
68       result append ("|\n")
69     }
70     result.toString()
71   }
```

Für spätere Zwecke sei auch die Methode zur Gleichheit überschrieben, so daß sie zwei Werteweise gleiche Spielfelder als solche erkennt.

SlidePuzzle.scala

```
72   override def equals(other:Any):Boolean={
73     var result=true;
74     var that=other.asInstanceOf[SlidePuzzle]
75     for (val r<-Iterator.range(0,SIZE)
76         ;val c<-Iterator.range(0,SIZE)
77         ;result
78         ){result=result&&this.field(r)(c)==that.field(r)(c)}
79     result
80   }
81 }
```

Damit ist unser erster rationaler problemlösender Agent implementiert.

Es ist an der Zeit, die Breitensuche für das Schiebepuzzle zu testen. Hierzu erzeugen wir eine Schiebepuzzleinstanz, verschieben ein paar Plättchen und lassen nach einer Lösung suchen, die dann ausgegeben wird:

TestSlide.scala

```
1  package name.panitz.ki
2  object TestSlide extends Application{
3  import Direction.{_}
```

```
4   val sp= (new SlidePuzzle(3)
5              move Left  move Up   move Left  move Up
6              move Right move Down move Right move Down)
7
8   Console println "zu loesendes Problem"
9   Console println sp
10  Console println "ist das schon der Terminalzustand"
11  Console println (sp.terminalState)
12
13  val solution=sp.breadthFirst
14  Console println "loesende Zugfolge mit Breitensuche"
15  Console println solution
16
17  var tmp=sp
18  for (val m<-solution) tmp=tmp move m
19
20  Console println "geloestes Problem"
21  Console println tmp
22 }
```

Das sah doch schon recht gut aus. Versuchen wir das gleiche mit einem etwas stärker verschobenen Puzzle, so müssen wir leider feststellen, das bereits die uns realistisch zur Verfügung stehenden Zeit und Speicherresourcen gesprengt werden.

TestSlideComplex.scala

```
1  package name.panitz.ki
2  object TestSlideComplex extends Application{
3  import Direction.{_}
4    val moves =
5     List(Left,Up,Up,Left,Down,Down,Right,Up,Up,Right,Down,Down
6         ,Left,Up,Up,Right,Down,Down,Left,Up,Left,Down,Right
7         ,Right,Up,Left)
8
9    var sp= new SlidePuzzle(3)
10   for (val m<-moves)sp=sp move m
11
12   Console println "zu loesendes Problem"
13   Console println sp
14
15   val solution=sp.breadthFirst
16   Console println "loesende Zugfolge mit Breitensuche"
17   Console println solution
18 }
```

Wird dieses Programm gestartet, so läßt die Lösung länger auf sich warten, als wir gewillt sind zu warten. Obwohl es sich insgesamt um ein sehr simples Spiel mit nur sehr wenigen Aktionen pro Zustand handelt, sprengt das naive Hauruck-Verfahren der Breitensuche sehr schnell unsere Resourcen. Dieses ist natürlich mit dem exponentiellen Anwachsen der Baumebenen zu erklären. Hier ist die Crux der KI. Für die meisten Probleme der KI sind nur deterministische Algorithmen

bekannt, die einen exponentiellen Aufwand haben; oder der wie der Fachmann sagt, zur Klasse NP gehören.

2.2.3 Tiefensuche

Die mehr oder wenige komplett komplementäre Strategie zur Breitensuche ist natürlich die Tiefensuche. Hier wird der Suchbaum nicht ebenenweise aufgebaut, sondern zunächst ein Pfad möglichst weit in die Tiefe verfolgt, und erst wenn dort ein Blatt nicht mehr expandiert werden kann, weil es keine Nachfolgerknoten mehr gibt, wird im Baum zum vorletzten Knoten zurückgegangen, um dort beim nächsten Kind in die Tiefe zu schauen.

Wir können tatsächlich die allgemeine Suchfunktion benutzen, um ihr die Strategie zur Tiefensuche zu übergeben.[4] Für die Tiefensuche ist nicht das älteste Blatt zu expandieren, sondern das neueste, welches wir daher an den Anfang der Blattliste einfügen werden. Somit ist der *fringe* gerade genau umgekehrt wie in der Breitensuche sortiert. Jetzt stehen Zustände weiter vorne im *fringe*, wenn sie tiefer im Baum stehen.

Erweitern wir also die Suche, um die Tiefensuche.[5]

DepthSearch.scala

```
1 package name.panitz.ki
2 trait DepthSearch[M] extends SearchTree[M] {
```

DieseEinfügefunktion, die neue knoten vorne einfügt, erzeugt uns für unseren allgemeinen Suchalgorithmus eine Tiefensuche.

DepthSearch.scala

```
3   def depthSearch():List[M]=generalSearch((x,y)=>y:::x)
```

Einen Nachteil der Breitensuche, hat die Tiefensuche nicht mehr. Die Länge des *fringe* wächst nicht exponentiell an. Haben wir einen durchschnittlichen Verzweigungsgrad im Baum von c, dann hat in einer Baumtiefe n der *fringe* eine Länge von $c * m$. Im Gegensatz dazu hat er für die Breitensuche eine Länge von c^{n+1}.

Wir haben ein ernsthaftesProblem mit dieser Tiefensuche. Sie wird für viele Problemfelder nicht terminieren. Sie terminiert nur, wenn der Suchbaum eine endliche Tiefe hat. Der Suchbaum für unser Schiebespiel hat leider, auch wenn es nur endlich viele Zustände gibt, eine unendliche Tiefe. Damit terminiert die Tiefensuche nicht. Ein Beispiel für ein Spiel mit endlicher Tiefe wäre z.B. *Vier Gewinnt*, das wir später kennenlernen werden. Dort ist irgendwann das Spielfeld voll, und kein weiterer Stein kann gesetzt werden.

Da es allerdings nur endlich viele Zustände im Schiebespiel gibt, kann man die Tiefensuche darauf anwenden, wenn man Zustände, die an anderer Stelle des Suchbaums bereits expandiert wurden, nicht ein weiteres Mal expandiert. Hierzu ist allerdings die Menge aller bereits besuchten Spielzustände zu speichern. Wir betrachten damit den Baum in gewisser Weise als gerichteter Graph. Baumknoten, die mit dem gleichen Zustand markiert sind, werden darin als ein gemeinsamer Graphknoten betrachtet. Dieser Graph enthält nun unter Umständen Zyklen.

[4] Niemand hat hier behauptet, daß das sehr effizient ist, was hier geschieht.

[5] Daß wir eine neue Unterklasse von `SearchTree` schreiben, statt die Methoden zur Tiefensuche einfach in `SearchTree` zu ergänzen hat lediglich skripttechnische Gründe, der Zeilennummern der Klassen.

Wenn wir die allgemeine Suche nicht auf einem Baum, sondern den sogestallt definierten Graphen implementieren wollen, müssen wir darüber Buch führen, welche Zustände bereits besucht wurden. Damit ist man mit der klassischen Frage konfrontiert: wie verhindere ich innerhalb eines Labyrinths immer im Kreis zu laufen? Hierzu ist darüber Buch zu führen, welche Knoten schon besucht wurden. Ein sehr frühes Beispiel, über bereits besuchte Knoten eines Graphen Buch zu führen, findet sich in der griechischen Mythologie mit dem Ariadnefaden.

Unsere Scalaimplementierung wird keinen Faden durchs Labyrinth ziehen, sondern in einer Menge speichern, welche Zustände bereits expandiert wurden. Auch dieses läßt sich allgemein als eine Methode implementieren.

> Der **Ariadnefaden** war der griechischen Mythologie zufolge ein Geschenk der Prinzessin Ariadne an Theseus. Mit Hilfe des Fadens fand Theseus den Weg durch das Labyrinth, in dem sich der Minotauros befand. Nachdem Theseus den Minotauros getötet hatte, konnte er mit Hilfe des Fadens das Labyrinth wieder verlassen. Der Hinweis für die Verwendung des Fadens stammt von Daidalos, der auch das Labyrinth entworfen hat.
> Als Strafe für den Hinweis wurde anschließend Daidalos mit seinem Sohn Ikaros ins Labyrinth gesperrt. Mit Hilfe selbst gebauter Flügel konnten sich beide befreien.

Abbildung 2.4: **Wikipediaeintrag** (12. April 2006): Ariadnefaden

DepthSearch.scala

```
1  def generalGraphSearch(insertNew:Insert[M]):List[M]={
2      val allreadyExpanded = new java.util.HashSet/*[SearchTree[M]]*/()
3      generalSearch(
4        (xs,ys)=>{
5          val newNodes=
6            for (val y<-ys;!(allreadyExpanded contains y)) yield y
7          for (val y<-newNodes) allreadyExpanded add y
8          insertNew(newNodes,xs)}
9       )
10   }
```

Die Tiefensuche im Graphen, sei nun eine allgemeine Suche im Graphen, die beim Einfügen, neu-expandierte Knoten vorne im *fringe* einfügt.

DepthSearch.scala

```
1  def depthSearchGraph():List[M]=generalGraphSearch((x,y)=>y:::x)
```

Damit haben wir nun zwar sichergestellt, daß die Tiefensuche terminiert, sofern es endlich viele Zustände gibt. Glücklich werden wir für das Schiebepuzzle damit allerdings nicht. Das Schiebepuzzle kennt keine Sackgassen im herkömmlichen Sinne, es kann beliebig Hin-und-Her geschoben werden. Wir können uns ganz schön in die Tiefe verlaufen, bis wir sehr naheliegende Lösungen gefunden haben. Die gesuchte Lösung wird höchstwahrscheinlich sehr kompliziert sein. Die Tiefensuche findet nicht unbedingt eine optimale Lösung.

2.2.4 Tiefensuche mit maximaler Tiefe

Im letzten Abschnitt hat sich die Tiefensuche als problematische Suchstrategie erwiesen, weil sie mitunter unendlich weit in die Tiefe absteigen kann. Weiß man hingegen, daß die Lösung

innerhalb einer bestimmten Tiefe zu finden ist, so läßt sich mit einer Tiefensuche, die eine maximale Beschränkung hat, die Lösung finden.

DepthSearch.scala

```
2   def depthSearch(depth:Int):List[M]=
3     generalSearch((xs,ys)=>{
4       val newNodes=for (val y<-ys;y._1.length < depth) yield y
5       newNodes:::xs}
6     )
```

Die Einfügefunktion fügt jetzt nur noch Knoten ein, deren Pfadlänge von der Wurzel nicht eine bestimmte vorgegebene Tiefe überschreitet. Natürlich findet diese Suchstrategie keine Lösung, wenn es keine in der vorgegebenen Tiefe gibt. Aber auch wenn es eine Lösung gibt, garantiert die limitierte Tiefensuche nicht, daß auch eine optimale Lösung gefunden wird.

2.2.5 Iteratives Vertiefen

Die Tiefensuche hat nicht die Speicherplatzprobleme der Breitensuche, findet dafür aber nicht unbedingt eine optimale Lösung. Kann man die Vorteile der beiden Strategien verbinden, ohne dadurch Einbußen in der Laufzeitkomplexität zu erlangen. Man kann. Das entsprechende Verfahreen heißt: *iteratives Vertiefen*. Als eigentlicher Suchalgorithmus wird dabei, die limitierte Tiefensuche benutzt. Diese wird angefangen mit der Tiefe 0 solange mit aufsteigenden maximalen Tiefen aufgerufen, bis bei einer Tiefe ein Ergebnis gefunden wurde.

In Scala ergibt sich folgende naheliegende Implementierung.

DepthSearch.scala

```
7   def deepening():List[M]={
8     var result:List[M]=null;
9     var depth=0;
10    while (result==null){
11      result=depthSearch(depth)
12      depth=depth+1
13    }
14    result
15  }
16 }
```

Das iterative Vertiefen simuliert im Prinzip die Breitensuche über die Tiefensuche. Damit findet es eine optimale Lösung. Allerdings, wenn der Zielzustand in Tiefe n zu finden ist, dann ist die Ebene $n-1$ in $n-1$ Durchläufen der Iteration zu generieren.

2.2.6 Tiefes Schieben

Es soll natürlich auch getestet werden, ob wir mit den in diesem Abschnitt vorgestellten Suchverfahren zur Tiefensuche, auch Lösungen für das Schiebepuzzle finden können. Hierzu sei eine Unterklasse des Schiebepuzzles implementiert, die auch `DepthSearch` implementiert.

DepthSlide.scala

```
1  package name.panitz.ki
2  class DepthSlide(S:Int)
3    extends SlidePuzzle(S) with DepthSearch[Direction.Value]{
4
5    def this(that:SlidePuzzle)={this(that.SIZE);copyFromThat(that)}
6
7    override def move(m:Direction.Value):DepthSlide={
8      val res=new DepthSlide(this)
9      res.move(m,res).asInstanceOf[DepthSlide]
10   }
11 }
```

Ein paar kleine Testaufrufe zeigen, daß wir auch mit diesen Verfahren tatsächlich Lösungen finden können.

TestDepth.scala

```
1  package name.panitz.ki
2  object TestDepth extends Application{
3  import Direction.{_}
4    val sp= (new DepthSlide(3)
5               move Left  move Up   move Left  move Up
6               move Right move Down move Right move Down)
7
8    Console println sp
9    Console println (sp.terminalState)
10
11   Console println "depth search with limit 10"
12   val solution2=sp depthSearch 10
13   Console println solution2
14
15   var tmp=sp
16   for (val m<-solution2) tmp=tmp move m
17   Console println tmp
18
19   Console println "iterative deepening"
20   val solution3=sp.deepening
21   Console println solution3
22
23   tmp=sp
24   for (val m<-solution3) tmp=tmp move m
25   Console println tmp
26
27   Console println "depth first search"
28   Console println "I hope you have some time"
29   val solution1=sp.depthSearchGraph
30   Console println solution1
31
32   tmp=sp
33   for (val m<-solution1) tmp=tmp move m
```

```
34    Console println tmp
35  }
```

Unser hauptsächliches Problem mit der Komplexität ist leider noch nicht gelöst. Unsere derzeitigen Hauchruck-Verfahren der Suche, sind nicht in der Lage, ein wenig stärker verschobene Schiebespiele zu lösen.

2.2.7 Randbedingungen per Suche Lösen

Eine große Klasse von Problemen in der KI beschäftigen sich damit bestimmte Lösungen unter Randbedingungen zu finden. Diese Probleme werden CSP genannt, als Abkürzung für *constraint satisfaction problem*. Dabei geht es darum bestimmte Variablen mit Werten zu belegen, so daß eine Reihe von Randbedingungen eingehalten werden. In unserem Fachbereich ist Frau Groß-Bosch jedes Semester damit beauftragt, ein relativ komplexes CSP zu lösen, wenn sie die Stundenpläne plant. Hier sind die Variablen die uns zur Verfügung stehenden Räume zu bestimmten Vorlesungszeiten. Gesucht ist eine Belegung dieser Variablen mit Vorlesung und zugehörigen Dozenten. Die dabei zu erfüllenden Randbedingungen sind mannigfach: Alle Vorlesungen müssen gehalten werden, von Dozenten, die das Fach berherrschen. Kein Dozent kann zur gleichen Zeit mehr als eine Vorlesung halten. Vorlesungen eines Semesters dürfen nicht parallel liegen, Raumgrößen müssen zur erwarteten Teilnehmerzahl passen, bestimmte Wünsche an die Raumtechnik der Dozenten müssen erfüllt werden, kaum ein Dozent will morgens um acht, die Dozenten wollen möglichst keine Freistunden zwischen den Vorlesungen haben undundund. Hier einmal ein kleines bewunderndes Dankeschön an Frau Groß-Bosch, die allsemesterlich dieses CSP zur Zufriedenheit aller löst.

Eine erst seit Kurzem in Europa populär gewordene Form eines CSP sind die Sudoku, die mitlerweile in vielen Zeitungen abgedruckt werden und von vielen Leuten gerne zum Zeitvertreib gelöst werden. Hier sind die Variablen die freien Felder, die mit den Werten 1 bis 9 zu belegen sind, so daß keine Ziffer in einer Reihe, Spalte oder Unterquadrat mehrfach auftritt.

Wir können unsere bisherigen Suchstrategien direkt benutzen, um CSP zu lösen. Als Aktion ist dabei nur die Belegung einer Variablen mit einem bestimmten Wert zu identifizieren. Jeder Plan ist exakt gleich lang: er hat exakt die Anzahl n der zu belegenden Variablen als Länge. Damit hat der Suchbaum eine endliche Tiefe, die, wenn das CSP lösbar ist, genau n ist.

Der Startzustand des CSP als Suchproblem ist der Zustand, in dem noch keine Variabel belegt ist. Im Endzustand sind alle Variablen so belegt, daß gegen keine Randbedingung verstoßen wurde. Die Kosten eines Zugs ist konstant der Wert 1.[6] Nimmt man als Nachfolgerfunktion für einen Zustand, die Funktion, die die Menge aller möglichen erlaubten Belegungen einer beliebigen Variablen zurückgibt, wächst der Suchbaum allerdings kollossal in die Breite und sprengt sehr schnell die realistisch zur Verfügung stehenden Resourcen.

Betrachten wir einmal die möglichen Lösungen eines CSP als Suchproblem. Die einzelnen Aktionen eines Plans zur Lösung sind kommutativ. Wir können die Aktionen einer Lösung beliebig permutieren und erhalten wieder eine Lösung. Diese spezielle Eigenschaft können wir benutzen, um die Breite des Suchbaums radikal einzuschränken. Jede Tiefe des Baums bekommt einfach eine feste Variable zugeordnet, der zu dieser Zeit ein Wert zugewisen werden kann. Damit reduziert sich der Verzweigungsgrad des Baumes auf jeder Ebene um den Bruchteil der noch zu belegenden Variablen.

[6] Wie wir es bisher eh immer angenommen haben.

Da die Tiefe des Suchbaums nicht nur beschränkt ist, sondern auch bekannt ist, daß nur in der maximalen Tiefe des Baums eine Lösung zu erwarten ist, sofern es eine gibt, bietet sich die Tiefensuche dazu an, das CSP zu lösen. Der allgemeine Nachteil der Tiefensuche, das sie die optimale Lsöung verpasst existiert nicht mehr. Dafür hat sie jetzt den Vorteil, speichereffizienter als die Breitensuche zu sein.

Sudoku (jap. 数独 Sūdoku, kurz für 数字は独身に限る *Sūji wa dokushin ni kagiru*, wörtlich: *Zahlen als Einzel beschränken*) ist ein Zahlenrätsel.

Regeln und Begriffe

Das Spiel besteht aus einem Gitterfeld mit 3×3 Blöcken, die jeweils in 3×3 Felder unterteilt sind, insgesamt also 81 Felder in 9 Reihen und 9 Spalten. In einige dieser Felder sind schon zu Beginn Ziffern zwischen 1 und 9 eingetragen. Ziel des Spiels ist es nun, die leeren Felder des Puzzles so zu vervollständigen, dass in jeder der 9 Zeilen, Spalten und Blöcke jede Ziffer von 1 bis 9 genau einmal auftritt.
Typischerweise sind 22 bis 36 Felder von 81 möglichen vorgegeben. Es gibt allerdings auch bekannte Kombinationen, in denen 17 Zahlen ein eindeutiges Sudoku bilden. Die minimale Anzahl Ziffern, die nötig ist, um ein eindeutiges Sudoku zu bilden, ist nicht bekannt. Da jede Zahl in jedem der drei genannten *Bereiche* nur einmal auftritt, ist der diesen Umstand andeutende wörtliche Name des Puzzlespiels *Einzelne Zahl*.
Wenn eine Zahl in einem Feld möglich ist, bezeichnet man sie als *Kandidat*. Die drei Bereiche (Reihe, Spalte, Block) werden zusammengefasst als *Einheiten* bezeichnet.
Inzwischen gibt es auch Sudokus mit 4×4 Unterquadraten und somit 256 (=16×16) Feldern, in die 16 verschiedene Zahlen, Buchstaben oder Symbole verteilt werden, sowie Sudokus mit 4×3 Blöcken mit jeweils 3×4 Feldern. Ebenso sind 5×5 Sudokus denkbar, etc. Für Kinder gibt es Sudokus mit einer Kantenlänge von 2 pro Unterquadrat, also werden dort nur 4 Ziffern oder Bildsymbole benötigt. Weitere Varianten sind Sudokus mit treppenförmiger Begrenzung der Blöcke (engl. Stairstep Sudoku) und solche mit unregelmäßig geformten Blöcken.
Beim Killer Sudoku gibt es keine Schlüsselzahlen, sondern die Summe von Zahlen in zusammengefassten Blöcken wird angegeben.
Seit Ende 2005 gibt es tragbare, elektronische Sudoku-Geräte. Desweiteren gibt es Brettspiele und Computerspiele. Ein Nachfolger von Sudoku ist Kakuro.

Ursprung

Der früheste Ursprung des Sudoku kann in den Rätselspielen des Schweizer Mathematikers Leonhard Euler gesehen werden, der solche unter dem Namen Carré latin (Lateinisches Quadrat) bereits im 18. Jahrhundert verfasste. Abweichend von den modernen Sudoku-Rätseln sind diese nicht in Blöcke (Unterquadrate) unterteilt.
Das heutige Sudoku in Blockform wurde 1979 in der Zeitschrift Math Puzzels & Logic Problems, die vom Computerhersteller Dell herausgegeben wird, erstmals veröffentlicht. Die ersten Sudokus wurden zwar in den USA publiziert, seinen Durchbruch erlangte das Zahlenrätsel jedoch erst um etwa 1984, als die japanische Zeitschrift Nikoli diese zunächst unter dem Namen *Sūji wa dokushin ni kagiru* regelmäßig abdruckte. Hieraus entwickelte sich schließlich der Begriff Sudoku. Der Neuseeländer Wayne Gould hat Sudoku auf einer Japanreise kennen gelernt. Sechs Jahre brauchte er, um eine Software zu entwickeln, die neue Sudokus per Knopfdruck entwickeln kann.
Anschließend bot er seine Rätsel der Times in London an. Die Tageszeitung druckte die ersten Sudoku-Rätsel und trat auf diese Weise eine Sudoku-Lawine in der westlichen Welt los.
In Österreich führte der regelmäßige Abdruck in Tageszeitungen wie Der Standard und Kronen Zeitung zu einer raschen Verbreitung Ende 2005. Des weiteren erscheint es regelmäßig in der Hersfelder Zeitung, der Frankfurter Rundschau, in Der Tagesspiegel und mittlerweile auch in der ZEIT.

Abbildung 2.5: **Wikipediaeintrag** (12.4.2006): Sudoku

Implementierung von Sudokus

Versuchen wir jetzt einmal einen Agenten zum Lösen von Sudokus zu implementieren.
Wir brauchen eine Tiefensuche. Die Aktionen sind jetzt Tripel: die erste Zahl des Tripels gebe die Zeile, die zweite die Spalte, in die ein Wert gesetzt wird an, die dritte Zahl schließlich den Wert, der an

die entsprechende Position gesetzt wird.

Sudoku.scala

```
1  package name.panitz.ki
2  class Sudoku extends DepthSearch[Tuple3[Int,Int,Int]] {
3    type Move=Tuple3[Int,Int,Int]
```

Zur internen Speicherung des Spielfeldes bietet sich wieder eine zweidimensionale Reihung an.

Sudoku.scala

```
4    val field:Array[Array[Int]]=new Array[Array[Int]](9)
5    for (val i<-Iterator.range(0,9))field(i)=new Array[Int](9)
```

Ein Konstruktor zum Kopieren des Spielfeldes sei vorgesehen:

Sudoku.scala

```
6    def this(that:Sudoku)={
7      this()
8      for (val r<-Iterator.range(0,9);val c<-Iterator.range(0,9))
9        field(r)(c)=that.field(r)(c)
10   }
```

Jetzt kommt die entscheidene Methode, die die möglichen Züge auf einem Zustand zurückgibt. Hierzu soll bei einem Zustand als nächster Zug, nur das erste (von links nach rechts, oben nach unten) freien Feld mit einem Wert belegt werden dürfen. Hierzu suchen wir zunächst das erste freie Feld:

Sudoku.scala

```
11   def moves():List[Move]={
12    val freeFields=
13     for (val r<-Iterator.range(0,9)
14         ;val c<-Iterator.range(0,9)
15         ;field(r)(c)==0) yield Pair(r,c)
```

Wenn der daraus entstehende Iterator ein nächstes (also erstes) Element hat, dann suchen wir unter den Randbedingen für Sudokus eine entsprechende Lösung für dieses Feld. Ansonsten das Feld bereits voll.

Sudoku.scala

```
16    if (freeFields.hasNext){
17      val Pair(r,c)=freeFields.next
18      (for (val n<-Iterator.range(1,10)
19           ;fullFillsConstraints(n,r,c)
20           ) yield {Tuple(r,c,n)}).toList
21    }else List()
22   }
```

Drei Randbedingungen sind zu beachten: in Reihe, Spalte und Unterquadrat eines Feldes dürfen keine zwei Felder mit dem gleichen Wert belegt sein.

Sudoku.scala
```
23  def fullFillsConstraints(n:Int,r:Int,c:Int)=
24    (  fullFillsRow(n,r)
25     &&fullFillsColumn(n,c)
26     &&fullFillsSquare(n,(r-r%3),(c-c%3)))
```

Die drei Einzelbedingungen seien in relativ naiver Weise implementiert:

Sudoku.scala
```
27  def fullFillsRow(newVal:Int,r:Int)={
28    val values=newVal ::
29      (for (val c<-Iterator.range(0,9);val n=field(r)(c);n!=0)
30        yield n).toList
31    values.length==values.removeDuplicates.length
32  }
33
34  def fullFillsColumn(newVal:Int,c:Int)={
35    val values=newVal ::
36      (for (val r<-Iterator.range(0,9);val n=field(r)(c);n!=0)
37        yield n).toList
38    values.length==values.removeDuplicates.length
39  }
40
41  def fullFillsSquare(newVal:Int,r:Int,c:Int)={
42    var values=List[Int](newVal)
43    for (val i<-Iterator.range(0,3)
44        ;val j<-Iterator.range(0,3)
45        ;val n=field(r+i)(c+j)
46        ;n!=0)
47      values=n::values
48    values.length==values.removeDuplicates.length
49  }
```

Noch sind die Methoden `move` und `terminalState` abstrakt. Um einen tatsächlichen Zug durchzuführen. Die Methode `move` sei ähnlich umgesetzt, wie es schon im Schiebepuzzle zu sehen war. Eine Kopie vom aktuellen zustand wird angelegt, in der dann der Zug volluogen wird.

Sudoku.scala
```
50  def move(m:Move):Sudoku={
51    val result=new Sudoku(this)
52    val Tuple3(r,c,n)=m
53    result.field(r)(c)=n
54    result
55  }
```

Zum Testen, ob wir den Finalzustand erreicht haben, braucht lediglich ausgeschlossen werden, daß ein Feld noch mit 0 belegt ist.

Sudoku.scala

```
56   def terminalState():Boolean=
57    (for (val r<-Iterator.range(0,9);val c<-Iterator.range(0,9)
58         ;val n=field(r)(c)) yield n).forall((x)=>x!=0)
```

Auf eine einfache Implementierung von `toString` sei natürlich nicht verzichtet.

Sudoku.scala

```
59   override def toString()={
60     val result = new StringBuffer()
61     for (val j<-Iterator.range(0,9)){
62       for (val i<-Iterator.range(0,9))
63        result append ("|"+(if (0==field(j)(i)) " "else field(j)(i)))
64       result append ("|\n")
65     }
66     result.toString()
67   }
68 }
```

Schließlich wollen wir unseren Agenten zum Lösen von Sudokus auch einmal Testen.

TestSudoku.scala

```
1  package name.panitz.ki
2  object TestSudoku extends Application{
3    var sud=new Sudoku()
4    val f
5     =( List
6       ( List(0,0,0,0,0,0,5,0,9)
7       , List(0,9,3,0,8,0,0,7,2)
8       , List(2,0,8,4,5,9,0,0,0)
9       , List(9,0,0,0,0,0,2,0,0)
10      , List(8,0,0,7,6,0,0,4,0)
11      , List(0,6,0,9,1,0,0,0,5)
12      , List(0,8,0,0,0,0,1,5,0)
13      , List(3,0,0,0,7,0,0,0,4)
14      , List(0,1,0,3,2,0,0,9,0)))
15
16   val f2=f.toArray
17   for (val r<-Iterator.range(0,9))
18    sud.field(r)=f2(r).toArray
19
20   Console.println(sud)
21   val solution=sud.depthSearch
22   Console.println(solution)
23   for (val s<-solution) sud=sud move s
24   Console.println(sud)
25 }
```

Tatsächlich ließ sich dieses Sudoku ohne Wartezeit problemlos von unserem Agenten lösen. Im Anhang des Skripts findet sich eine minimale Gui-Anwendung, die es erlaubt beliebige Sudoku-Aufgaben einzugeben und lösen zu lassen.

Waltz Prozedur zur Interpretation 2-dimensionaler Projektionen

Man mag einwänden, daß das vorangegangenen Beispiel zur Suche einer Lösung unter Randbedingungen sehr künstlich, oder eben nur ein Spiel war, was weit von den Zielen der KI entfernt ist. Ein sehr schönes Beispiel für die Randbedingungen, daß seit bald 30 Jahren in Lehrbüchern und Vorlesungen zur KI gerne benutzt wird, ist die Interpretation von Linienzeichnungen. Es geht dabei um Zeichnungen, die ein Szenario von Bauklötzen darstellt. Dabei wird folgende Einschränkung gemacht:

- jede Körper hat nur ebene Flächen
- jede Flächer hat genau vier Ecken
- in jeder Ecke stoßen genau drei Flächen aufeinander
- zwischen direkt aufeinander stehenden Flächen ist keine Linie zu sehen.

Eine Zeichnung eines Szenarios soll die Projektion auf eine 2-dimensionale Fläche einer beliebigen Anordnung solcher Körper sein.

Ein Beispielszenario[7] findet sich in Abbildung 2.6.

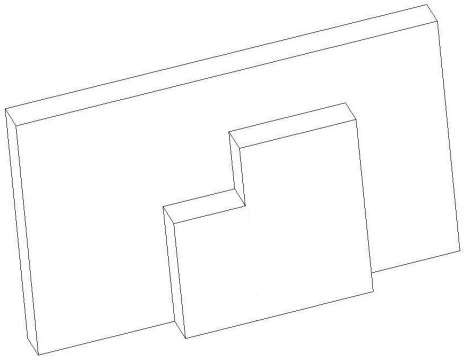

Abbildung 2.6: Beispielszenario in der Bauklotzwelt.

Aufgabe ist es nun, eine solche Zeichnung zu interpretieren. Dabei können die Striche drei unterschiedliche Arten von Kanten in der Realität sein:

- sie können normale konkave Kanten eines Körpers sein.
- es kann sich auch um konvexe Kanten handeln, die entstehen, wenn zwei Körper aneinander oder aufeinander stehen.
- es kann aber auch eine Kante sein, die das Gebilde von Körpern umrandet, an der also keine zwei sichtbaren Ebene zusammenstoßen.

Die Interpretaion einer Zeichnung bedeutet jetzt, jede Kante zu markieren, ob es sich um eine konvexe, konkave oder Umrandungskante handelt. Hierzu seien die folgenden Symbole gewählt:

[7] Leider ein wenig schief und krumm freihand mit der Maus gezeichnet.

- + für konvexe Kanten
- − für konkave Kanten
- → für Umrandungskanten

Die Markierung der Kanten soll möglichst konsistent sein und dem Szenario aus der realen Welt entsprechen.

Auch hierbei handelt es sich um ein Suchproblem, in dem Randbedingungen zu erfüllen sind. Diese Randbedingungen sind in diesem Fall natürliche Beschränkungen, wie sie in einem Bild, das durch Projektion eines Bauklotzszenarios entstehen kann. Wir wollen uns in diesem Abschnitt nicht zu detailliert mit der eigentlichen Klötzchenwelt beschäftigen, sondern es lediglich als Beispiel nehmen, um zu zeigen, wie die bisher vorgestellten Algorithmen unterschiedlichste Aufgaben lösen können, wenn es gelingt, das zu lösende Problem entsprechend zu modellieren.

Dazu wurde analysiert was für Arten von Ecken in einer solchen Zeichnung überhaupt auftreten können, und wie diese Ecken überhaupt markiert sein können. Dabei stellt sich heraus, daß es nur vier Arten von Ecken gibt:

- L-förmige Ecken, an denen zwei sichtbare Kanten aneinander stoßen.
- T-förmige Ecken, an denen eine Kante auf eine in der Projektion optisch durchgehende Kante stößt.
- Gabeln, an denen eine Kante im flachen Winkel zu wei im spitzen Winkel zueinander stehenden Kanten steht.
- Pfeile, in denen drei Kanten im spitzen Winken zueinander stehen.

Die vier verschiedenen Eckenarten sind in Abbildung 2.7 zu sehen.

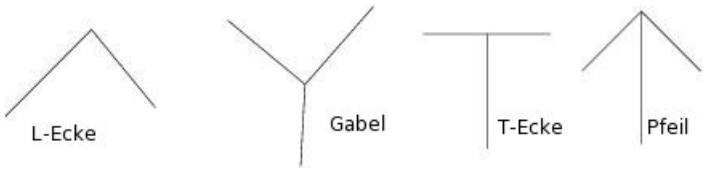

Abbildung 2.7: Die vier Arten von Ecken in Strichzeichnungen.

Nun stellt sich bei einer weiteren Analyse heraus, daß es für die vier Eckenarten nur recht wenig Möglichkeiten geben kann, wie sie interpretiert werden können. Alle natürlich möglichen Markierungen für die vier Eckentypen sind in Abbildung 2.8 zu sehen.

Damit ist klar, wie sich die Interpretation einer Linienzeichnung als Suchproblem mit Randbedingungen darstellen läßt. Gesucht ist eine Markierung, so daß jede Ecke mit einer für diesen Typ legalen Eckenmarkierung markiert ist, und keine Kante zwei verschiedene Markierungen hat.

Implementierung in Scala Da wir bereits die allgemeine Suche implementiert haben und da Scala es uns erlaubt die Markierungsprozedur relativ elegant und knapp umzusetzen, wollen wir uns auch für dieses Beispiel nicht um eine Implementierung drücken. In diesem Beispiel

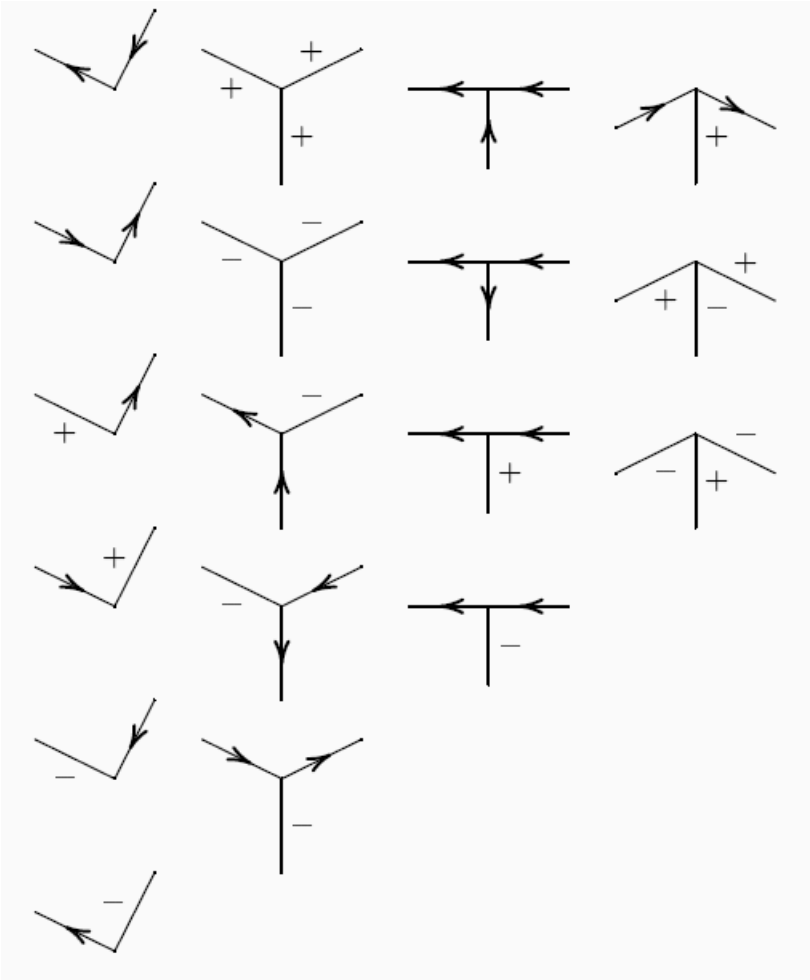

Abbildung 2.8: Alle möglichen Markierungen von Ecken in Strichzeichnungen.

werden zur Abwechslung verstärkt die Konzepte der funktionalen Programmierung von Scala genutzt.

Zunächst seien die Klassen für die vier Eckentypen definiert. Als gemeinsame Oberklasse haben sie die abstrakte Klasse `Vertex`.

Vertex.scala

```
1 package name.panitz.ki
2 abstract class Vertex
3   case class L(e1:Edge,e2:Edge) extends Vertex
4   case class Fork(e1:Edge,e2:Edge,e3:Edge) extends Vertex
5   case class T(left:Edge,middle:Edge,right:Edge) extends Vertex
6   case class Arrow(left:Edge,middle:Edge,right:Edge) extends Vertex
```

Die Ecken beinhalten Kanten. Eine Kante ist dabei ein Objekt, das eine Markierung hat:

Edge.scala

```
1 package name.panitz.ki
```

```
2  class Edge(var marked:Marked.Value){
3    override def toString()
4      =if (marked==null) "null" else marked.toString
5  }
```

Eine Markierung sei eine der folgenden vier Werte.

Marked.scala

```
1  package name.panitz.ki
2  object Marked extends Enumeration("-","+","<-","->") {
3    final val Concave, Convex, BoundaryLeft, BoundaryRight = Value}
```

Ein zu interpretierendes Bild ist nun lediglich eine Liste von Ecken, daher läßt sich gut von der Standardklasse `ArrayBuffer` ableiten[8]. Zusätzlich soll die Tiefensuche implementiert werden. Beides spiegelt sich in der Klassendeklaration wieder:

Scene.scala

```
1  package name.panitz.ki
2  class Scene extends scala.collection.mutable.ArrayBuffer[Vertex]
3              with    DepthSearch[CaseClass] {
```

Wie in all den vorangegangenen Beispielen, sei ein Konstruktor, der eine Kopie anlegt vorgesehen. In diesem Fall sieht dieser Kopiervorgang etwas eklig aus. Die Ecken teilen sich Kanten. Es muß daher Buch darüber geführt werden, welche Kante bereits kopiert wurde, damit diese kein zweites Mal kopiert werden. Hierzu wird sich einer Abbildung bedient. Beim Kopieren von Kanten wird diese Abbildung zunächst gefragt, ob bereits eine Kopie der Kante existiert. Falls nicht, wird eine Kopie angelegt und diese in der Abbildung vermerkt.

Scene.scala

```
4    def this(that:Scene)={
5      this()
6      import scala.collection.mutable.HashMap;
7      val copiedEdges=new HashMap[Edge,Edge]()
8      def cop(e:Edge)={
9        try {copiedEdges(e)}
10       catch{case _:java.lang.Error=>
11         val copy=new Edge(e.marked)
12         copiedEdges.update(e,copy)
13         copy
14       }
15     }
16     for (val v<-that.elements) {
17       v match {
18         case L(e1,e2)=>this += L(cop(e1),cop(e2))
19         case T(e1,e2,e3)=>this += T(cop(e1),cop(e2),cop(e3))
20         case Fork(e1,e2,e3)=>this += Fork(cop(e1),cop(e2),cop(e3))
21         case Arrow(e1,e2,e3)=>this += Arrow(cop(e1),cop(e2),cop(e3))
22       }
```

[8]Sie entsprich in etwa der Javaklasse `ArrayList`

```
23      }
24    }
25
```

18 verschiedene legale Markierungen gibt es an den Ecken. Die folgende Methode nutzt dieses Wissen über die natürliche Beschränkung der Eckenmarkierungen. Für eine Kante wird die Liste der legalen Markierungen einer solchen Kante zurückgegeben:

Scene.scala

```
26    def possibleMarks(v:Vertex)={
27      import Marked.{_};
28      v match{
29        case L(_,_)
30          => List(Tuple(BoundaryLeft,BoundaryLeft)
31                 ,Tuple(BoundaryRight,BoundaryRight)
32                 ,Tuple(BoundaryRight,Convex)
33                 ,Tuple(Convex,BoundaryRight)
34                 ,Tuple(BoundaryLeft,Concave)
35                 ,Tuple(Concave,BoundaryLeft)
36                 )
37        case Fork(_,_,_)
38          => List(Tuple(Convex,Convex,Convex)
39                 ,Tuple(Concave,Concave,Concave)
40                 ,Tuple(BoundaryLeft,Concave,BoundaryLeft)
41                 ,Tuple(BoundaryLeft,BoundaryLeft,Concave)
42                 ,Tuple(Concave,BoundaryLeft,BoundaryLeft)
43                 )
44        case T(_,_,_)
45         => List(Tuple(BoundaryLeft,BoundaryLeft,BoundaryLeft)
46                ,Tuple(BoundaryLeft,BoundaryRight,BoundaryLeft)
47                ,Tuple(BoundaryLeft,Concave,BoundaryLeft)
48                ,Tuple(BoundaryLeft,Convex,BoundaryLeft)
49                )
50        case Arrow(_,_,_)
51         => List(Tuple(BoundaryLeft,Convex,BoundaryLeft)
52                ,Tuple(Convex,Concave,Convex)
53                ,Tuple(Concave,Convex,Concave)
54                )
55      }
56    }
```

Die obige Methode kann genutzt werden, um zu testen, ob eine Ecke soweit korrekt markiert ist. Hierzu muß eine der legale Markierungen existieren, die zur Markierung der Ecke paßt:

Scene.scala

```
57    def correctlyMarked(v:Vertex)
58      =possibleMarks(v).exists((p)=>markFits(v,p))
```

Für eine Ecke wird mit der folgenden Methode getestet, ob eine Markierung zur bereits getätigten Markierung an entsprechender Ecke paßt:

Scene.scala

```
59  def markFits(v:Vertex,t:CaseClass)=
60    Pair(v,t) match{
61      case Pair(L(e1,e2),Tuple2(m1,m2)) =>
62        fit(e1.marked,m1)&&fit(e2.marked,m2)
63      case Pair(T(e1,e2,e3),Tuple3(m1,m2,m3)) =>
64        fit(e1.marked,m1)&&fit(e2.marked,m2)&&fit(e3.marked,m3)
65      case Pair(Fork(e1,e2,e3),Tuple3(m1,m2,m3)) =>
66        fit(e1.marked,m1)&&fit(e2.marked,m2)&&fit(e3.marked,m3)
67      case Pair(Arrow(e1,e2,e3),Tuple3(m1,m2,m3)) =>
68        fit(e1.marked,m1)&&fit(e2.marked,m2)&&fit(e3.marked,m3)
69    }
```

Eine Markierung paßt zu einer Kanten, wenn die Kante entweder noch nicht markiert ist, oder genau mit der in Frage kommenden Markierung markiert ist.

Scene.scala

```
70  def fit(x:Marked.Value,y:Any)=null==x||x==y
```

Wir haben nun alles beieinander, um die nächsten Züge zu berechnen. Hierzu, ähnlich wie beim Sudoku-Programm, wird die nächste noch nicht vollständig markierte Ecke gesucht:

Scene.scala

```
71  def moves():List[CaseClass]={
72    val unmarkedVertexes
73      = for (val v<-this.elements;!isMarked(v)) yield v
```

Sofern noch eine solche Ecke existiert, werden aus alen möglichen Markierungen für diesen Eckentyp, diejenigen herausgefiltert, die mit den bereits getätigten Kantenmarkierungen dieser Ecke zusammenpassen und nicht eine benachbarte Ecke damit illegal markieren:

Scene.scala

```
74    if (unmarkedVertexes.hasNext){
75      val v=unmarkedVertexes.next
76      (for (val pm<-possibleMarks(v)
77            ;markFits(v,pm)
78            ;this.elements.forall((x)=>correctlyMarked(x))
79            ) yield pm).toList
80    }else List()
81  }
```

Die folgende kleine Methode testet, ob eine Ecke bereits vollständig markiert ist, also keine Markierung mehr `null` ist.

Scene.scala

```
82  def isMarked(v:Vertex)=
83    v match{
84      case L(e1,e2) =>e1.marked!=null&&e2.marked!=null
85      case T(e1,e2,e3)
```

```
86          =>e1.marked!=null&&e2.marked!=null&&e3.marked!=null
87        case Fork(e1,e2,e3)
88          =>e1.marked!=null&&e2.marked!=null&&e3.marked!=null
89        case Arrow(e1,e2,e3)
90          =>e1.marked!=null&&e2.marked!=null&&e3.marked!=null
91      }
```

Nachdem somit auf einer Ebene möglichen Züge berechnet werden können, bleibt einen bestimmten Zug auch durchzuführen. Hierzu werde wieder die nächste nicht fertig markierte Ecke genommen. Alle übrigen Ecken werden in einer Kopie übernommen, nur die entsprechende Ecke wird neu markiert.

Scene.scala

```
92    def move(m:CaseClass):Scene={
93      val Pair(marked,unmarked)
94       =new Scene(this).toList.span((v)=>isMarked(v))
95      val result=new Scene()
96      for (val v<-marked) result += v
97      result += markedCopy(unmarked.head,m)
98      for (val v<-unmarked.tail) result += v
99      result
100   }
```

Die eigentliche Neumarkierung einer Ecke nimmt folgende Methode vor:

Scene.scala

```
101   def markedCopy(v:Vertex,t:CaseClass)={
102     def mv(e:Edge,m:Any)={e.marked=m.asInstanceOf[Marked.Value];e}
103     Pair(v,t) match{
104       case Pair(L(e1,e2),Tuple2(m1,m2))
105        => L(mv(e1,m1),mv(e2,m2))
106       case Pair(T(e1,e2,e3),Tuple3(m1,m2,m3))
107        => T(mv(e1,m1),mv(e2,m2),mv(e3,m3))
108       case Pair(Fork(e1,e2,e3),Tuple3(m1,m2,m3))
109        => Fork(mv(e1,m1),mv(e2,m2),mv(e3,m3))
110       case Pair(Arrow(e1,e2,e3),Tuple3(m1,m2,m3))
111        => Arrow(mv(e1,m1),mv(e2,m2),mv(e3,m3))
112     }
113   }
```

Schließlich ist nur noch zu spezifizieren, wenn ein Zielzustand erreicht ist. Das ist der Fall, wenn alle Ecken vollständig markiert sind:

Scene.scala

```
114   def terminalState():Boolean=
115    elements.forall((v)=>isMarked(v))
116 }
```

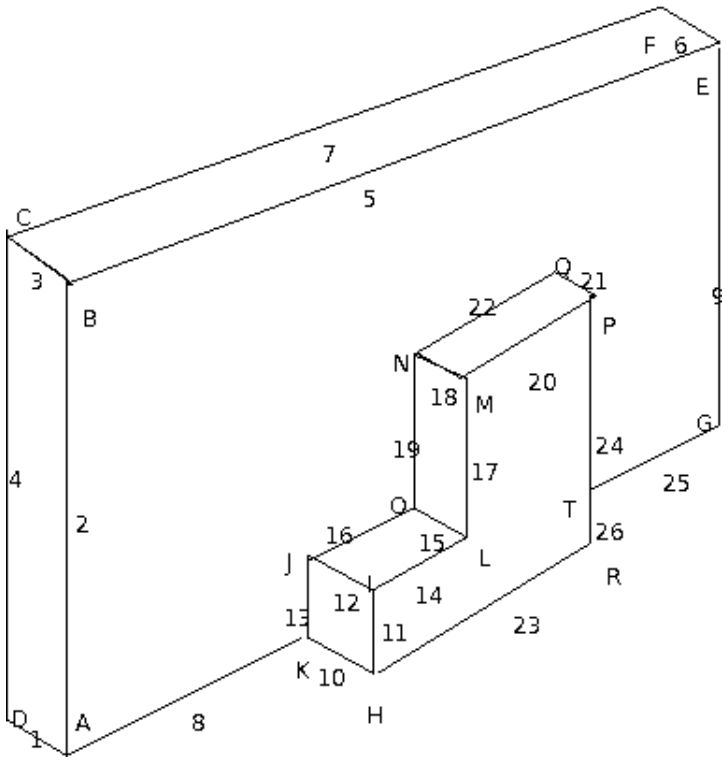

Abbildung 2.9: Beispielszenario durchnumeriert.

Abschließend soll unser kleines Programm zum Interpretieren von Linienzeichnungen natürlich auch einem Beispiel ausgetestet werden. Hierzu nehmen wir noch einmal die Stufen an einer Wand. Um die einzelnen Kanten und Ecken zu identifizieren seien sie einmal durchnumeriert, wie in Abildung 2.9 angegeben.

Im folgenden Programm werden die 26 Kanten mit `null` markiert und die Szene mit den 19 Ecken gebildet. Dann wird nach einer legalen Interpretation der Kanten gesucht:

Stairs.scala

```
1  package name.panitz.ki
2  object Stairs extends Application{
3    val e1 = new Edge(null)
4    val e2 = new Edge(null)
5    val e3 = new Edge(null)
6    val e4 = new Edge(null)
7    val e5 = new Edge(null)
8    val e6 = new Edge(null)
9    val e7 = new Edge(null)
10   val e8 = new Edge(null)
11   val e9 = new Edge(null)
12   val e10 = new Edge(null)
13   val e11 = new Edge(null)
14   val e12 = new Edge(null)
15   val e13 = new Edge(null)
```

```
16    val e14 = new Edge(null)
17    val e15 = new Edge(null)
18    val e16 = new Edge(null)
19    val e17 = new Edge(null)
20    val e18 = new Edge(null)
21    val e19 = new Edge(null)
22    val e20 = new Edge(null)
23    val e21 = new Edge(null)
24    val e22 = new Edge(null)
25    val e23 = new Edge(null)
26    val e24 = new Edge(null)
27    val e25 = new Edge(null)
28    val e26 = new Edge(null)
29
30    val vA = Arrow(e8,e2,e1)
31    val vB = Fork(e2,e5,e3)
32    val vC = Arrow(e4,e3,e7)
33    val vD = L(e1,e4)
34    val vE = Arrow(e6,e5,e9)
35    val vF = L(e7,e6)
36    val vG = L(e9,e25)
37    val vH = Arrow(e23,e11,e10)
38    val vI = Fork(e14,e12,e11)
39    val vJ = Arrow(e13,e12,e16)
40    val vK = Fork(e13,e8,e10)
41    val vL = Arrow(e17,e15,e14)
42    val vM = Fork(e20,e18,e17)
43    val vN = Arrow(e19,e18,e22)
44    val vO = Fork(e19,e16,e15)
45    val vP = Arrow(e21,e20,e24)
46    val vQ = L(e22,e21)
47    val vR = L(e26,e23)
48    val vS = T(e26,e25,e24)
49    var scene=new Scene()
50    scene++
51     List(vA,vB,vC,vD,vE,vF,vG,vH,vI,vJ,vK,vL,vM,vN,vO,vP,vQ,vR,vS)
52    val solution = scene.depthSearch
53    if (solution!=null){
54      for (val sol<-solution) scene=scene move sol
55      var c='A'
56      for (val v<-scene){
57        Console print c
58        Console print " "
59        Console println v
60        c=(c+1).asInstanceOf[Char]
61      }
62    }
63  }
```

Und tatsächlich, wenn das Programm gestartet wird, liefert es genau die Interpretation der Szene, wie wir sie auch mit unserer Sehgewohnheit gemacht hätten.

```
sep@pc305-3:~/fh/ki/student> scala -cp classes/ name.panitz.ki.Stairs
A Arrow(<-,+,<-)
B Fork(+,+,+)
C Arrow(<-,+,<-)
D L(<-,<-)
E Arrow(<-,+,<-)
F L(<-,<-)
G L(<-,<-)
H Arrow(<-,+,<-)
I Fork(+,+,+)
J Arrow(-,+,-)
K Fork(-,<-,<-)
L Arrow(+,-,+)
M Fork(+,+,+)
N Arrow(-,+,-)
O Fork(-,-,-)
P Arrow(<-,+,<-)
Q L(-,<-)
R L(<-,<-)
S T(<-,<-,<-)
sep@pc305-3:~/fh/ki/student>
```

2.3 Suche mit problemspezifischer Information

Bisher haben die benutzen Suchverfahren keinerlei Information über das zu lösende Problem benutzt. Wie schon bei dem sehr einfachen Schiebepuzzle zu sehen war, waren sie auf Grund der exponentiellen Komplexität nicht in der Lage, schon recht einfache Lösungen zu finden. Dieses Problem kann nur gelöst werden, indem zusätzliches Wissen über das Problem benutzt wird. Was könnte solch zusätzliches Wissen sein? Es könnte doch interssant sein, wie weit man glaubt, vom Zielzustand entfernt zu sein. Wenn eine derartige Information vorliegt, dann ist jeweils der Knoten aus dem *fringe* zum Expandieren vorzuziehen, von dem man annimmt, daß er schon möglichst Nahe am Zielzustand ist. Natürlich ist nicht davon auszugehen, daß wir exakt wissen, wie weit es jeweils noch bis zum Ziel ist. Wir können nur auf eine heuristische Abschätzung der realen Kosten zum Ziel hoffen.

2.3.1 Greedy-Best-First

Der Suchalgorithmus, der immer den Knoten aus dem *fringe* expandiert, für den die Abschätzung der Entfernung zum Ziel möglichst gering ist wird als *Greedy-Best-First* bezeichnet. Er läßt sich aus dem allgemeinen Suchalgorithmus erhalten, indem neue Knoten in den *fringe* so eingefügt werden, daß der *fringe* stets nach den abgeschätzten Kosten sortiert ist.

In der Scalaimplementierung schreiben wir wieder eine spezielle allgemeine Suchklasse, die nun eine zusätzliche Information verlangt, nämlich die abgeschätzte Entfernung zum Ziel:

InformedSearch.scala

```
1 package name.panitz.ki
2 trait InformedSearch[M] extends DepthSearch[M] {
3   def estimatedGoalDistance():Int;
```

Die Implemetierung hat sich entschieden, die Entfernung als ganze Zahl abzuschätzen. Der *Greedy-Best-First* ist jetzt die Suchstrategie, in der sortiert in den *fringe* eingefügt wird. Und zwar sortiert, nach der abgeschätzten Entfernung zum Ziel. Dazu benutzen wir eine lokale Funktion, die entscheidet, welche Zustand in dieser Relation kleiner ist:

InformedSearch.scala

```
4  def greedyBestFirst():List[M]={
5    def le=(x:Path[M],y:Path[M])
6      =>(x._2.asInstanceOf[InformedSearch[M]].estimatedGoalDistance
7         <=
8        y._2.asInstanceOf[InformedSearch[M]].estimatedGoalDistance)
```

Bezüglich dieser Kleiner-Relation sind die neuen Elemente in den *fringe* einzufügen:

InformedSearch.scala

```
9     generalGraphSearch((xs,ys)=>insertAll(le,xs,ys))}
```

Es bleibt noch die Einfügefunktion zu implementieren:

InformedSearch.scala

```
10 type LE[A]=(A,A)=>Boolean
11 def insert[A]:(LE[A],A,List[A])=>List[A]=(le,x,ys)=>{
12    var temp:List[A]=Nil
13    var rest=ys
14    while (!rest.isEmpty&& !le(x,rest.head)){
15      temp=rest.head :: temp
16      rest=rest.tail
17    }
18    temp.reverse  ::: x :: rest
19 }
20
21 def insertAll[A]:(LE[A],List[A],List[A])=>List[A]=(le,xs,ys)=>{
22   if (ys.isEmpty) xs
23   else insertAll(le,insert(le,ys.head,xs),ys.tail)
24 }
```

Wie sieht es jetzt mit dem *Greedy-Best-First* Algorithmus in Bezug auf das Auffinden einer optimalen Lösung aus? Hierzu betrachte man den Fall, daß nicht alle Aktionen gleich teuer sind. Angenommen vom Startzustand z_0 gibt es zwei Aktionen die zu einen Folgezustand führen. Die erste a_1 hat Kosten von 1000, die zweite a_2 Kosten von 1. Der Folgezustand zu a_1 sei nur noch 1 vom Ziel entfernt, der Folgezustand zu a_2 sei 2 vom Ziel entfernt. *Greedy-Best-First* wählt daher a_2 als erste Aktion und erhält eine Lösung mit Gesamtkosten von 1001, es gibt aber eine Lösung mit Gesamtkosten von lediglich 3.

2.3.2 A*-Suche

Die Abschätzung, wie weit das Ziel noch vermutet wird allein zum bestimmen des zu expandierenden Knotens zu benutzen, reichte nicht aus, um eine optimale Lösung zu finden. Es muß

zusätzlich auch noch berücksichtigt werden, wieviel Kosten der Pfad zu dem betrachteten Zustand bereits gekostet hat. Der Algorithmus, der auch dieses noch mit in Betracht zieht, wird als A*-Suche bezeichnet. Er expandiert den Knoten des *fringe*, für den die Summe aus geschätzten Kosten bis zum Ziel und den bereits erzielten Kosten vom Start minimiert wird.

Zur Implementierung des A*-Algorithmus wird zusätzlich noch eine Methode benötigt, die für eine Pfad die Kosten ermittelt:

InformedSearch.scala

```
25  def pathDistance(xs:List[M]):Int;
```

Dann läßt sich der A*-Algorithmus schreiben, indem nun in den *fringe* nach minimierter Summe von der Pfaddistanz von der Wurzel und der abgeschäützten Distanz zum Ziel sortiert eingefügt wird.

Dazu ist die Kleiner-Relation verfeinert zu implementieren:

InformedSearch.scala

```
26  def aStar():List[M]={
27    def le=(x:Path[M],y:Path[M])
28     =>(pathDistance(x._1)
29        +x._2.asInstanceOf[InformedSearch[M]].estimatedGoalDistance
30         <=
31        pathDistance(y._1)
32        +y._2.asInstanceOf[InformedSearch[M]].estimatedGoalDistance)
```

Der A*-Algorithmus benutzt diese Relation zum sortierten Einfügen in den *fringe*.

InformedSearch.scala

```
33    generalGraphSearch((xs,ys)=>insertAll(le,xs,ys))}
34  }
```

Wie sieht es mit den Algorithmus A* aus? Findet er eine optimale Lösung. Das ist abhängig von der heuristischen Abschätzungsfunktion. Hierzu stelle man sich einmal die Extremfälle vor:

- Die Abschätzung ist sehr optimistisch. Sie schätzt die Kosten wesentlich geringer ab, als sie in der Tat sind. Die optimistischste Abschätzung ist dabei die, die immer glaubt, man sei schon so gut wie am Ziel; im extremsten Fall konstant 0 ist. Dann mutiert der A*-Algorithmus aber zur Breitensuche. Es werden dann ja nur noch die real schon angefallenen Pfadkosten zur Auswahl, welcher Knoten als nächstes expandiert werden soll herangezogen. Das sind also die Knoten, die als nächstes zur Wurzel liegen, und somit wird Breitensuche vollzogen. Da die Breitensuche eine optimale Lösung findet, findet auch der A*-Algorithmus mit sehr optimistischer Abschätzung eine optimale Lösung.
- Nun betrachte man zum Gegensatz eine sehr pessimistische Abschätzung. Vom der Wurzel z_0 seien zwei Nachfolgezustände z_1 und z_2 erreichbar. Die Pfadkosten zu z_1 und z_2 seien beide gleich k. Von z_1 sei das Ziel mit Kosten 1 erreichbar, von z_2 mit Kosten 3. Die Abschätzung schätze die Entfernung von z_2 zum Ziel korrekt ab, die Entfernung von z_1 zum Ziel jedoch pessimistisch mit 4. Der Algorithmus wählt den Pfad über z_2 und erreicht das Ziel mit $k+3$. Der kürzere Weg über z_1 wird übersehen, weil die letzte Etappe dieser

Allgemein gilt, die Abschätzung darf die realen Kosten nie überschätzen, dann findet der A*-Algorithmus in der obigen Form eine optimale Lösung. Je genauer die Abschätzung ist, umso leichter findet der Algorithmus eine Lösung. Wenn die Abschätzungsfunktion mit den realen Kosten identisch ist, so kann die optimale Lösung direkt gefunden werden, dann entspricht die Abschätzungsfunktion einen Wegweiser, der bei jeder Weggabelung zeigt, in welche Richtung der kürzeste Weg liegt.

2.3.3 Abschätzungen im Schiebespiel

Versuchen wir nun zum Abschluß den A*-Algorithmmus auf unser Schiebespiel anzuwenden. Wir benötigen dabei eine möglichst gute Funktion, die die Kosten zum Ziel abschätzt. Eine sehr einfache solche Funktion ist die Funktion, die angibt, wieviel Plättchen noch auf einem falschen Feld liegen. Da in einem Schritt genau ein Plättchen seine Position verändert, ist die Zahl der falsch liegenden Plättchen immer kleiner als die Länge einer optimalen Lösung des Schiebepuzzles. Es wird damit der Aufwand nicht überschätzt und diese Abschätzung könnte im A*-Algorithmus benutzt werden, um eine optimale Lösung zu suchen. Die Abschätzung ist aber nicht sehr genau. Es läßt sich eine genauere Abschätzung finden: hierzu zähle man für jedes einzelne Plättchen, wie oft es minimal verschoben werden müßte, um an seine korrekte Position zu kommen. Dabei kann man sich idealisiert vorstellen, daß das jeweilige Plättchen allein auf dem Spielfeld ist. Diese Zahl ist maximal vier und leicht zu ermitteln. Die Summe dieser Anzahlen für alle Plättchen kann als Abschätzung dienen.

Folgende Klasse implementiert mit dieser Abschätzung den A*-Algorithmus für das Schiebespiel. Wir implementieren eine `InformedSearch`

InformedSlide.scala

```
1  package name.panitz.ki
2  class InformedSlide(I:Int) extends DepthSlide(I)
3                       with InformedSearch[Direction.Value]{
4
```

Die Abschätzung summiert für jedes Plättchen auf, in wieviel Schritten es minimal auf seinen richtigen Platz verschoben werden kann.

InformedSlide.scala

```
5    def estimatedGoalDistance():Int={
6      var result=0;
7      for (val i<-Iterator.range(0,SIZE);val j<-Iterator.range(0,SIZE)){
8        val v=field(i)(j)
9        if (v>0){
10         val shouldY=v/SIZE
11         val shouldX=v%SIZE
12         result=result+Math.abs(shouldY-i)+Math.abs(shouldX-j)
13       }
14     }
15     result
16   }
```

Die realen Kosten eines Pfades seien weiterhin die Länge des Pfades:

InformedSlide.scala
```
17   def pathDistance(xs:List[Direction.Value]):Int=xs.length
```

Schließlich noch der Konstruktor und aus mehr technischen Gründen, das Überschreiben der Methode `move`, damit diese auch ein `InformedSlide`-Objekt zurückgibt.

InformedSlide.scala
```
18   def this(that:SlidePuzzle)={this(that.SIZE);copyFromThat(that)}
19   override def move(m:Direction.Value):InformedSlide={
20     val res=new InformedSlide(this)
21     res.move(m,res)
22     res
23   }
24 }
```

2.4 Spielbäume

Die Algorithmen in diesem Kapitel konnten bisher recht ungestört arbeiten. In der Umwelt gab es nur einen Agenten, der ein statische Problem per Suche gelöst hat. Wir werden jetzt das Problemfeld erschweren, indem es einen zweiten konkurrierenden Agenten gibt. Dieses ist der Fall in klassischen 2 Parteien spielen. Die jeweiligen zwei Spieler sind konkurrierende Agenten, denn sie verfolgen ganz unterschiedliche Pläne. Jeder will, daß seine Farbe gewinnt. Auch das Problem, für einen Spieler einen möglichst guten Zug zu finden, läßt sich mit Suche lösen. Nun entscheidet allerdings bei jeder zweiten Ebene des Suchbaums, der konkurrierende Agent, welche der Alternativen genommen wird. Unter der Berücksichtigung, daß der konkurrierende Agent versucht seinerseits optimale Züge zu machen.

2.4.1 Min-Max-Suche

Das naheliegenste Verfahren, um einen optimalen Zug zu finden, wird als *Min-Max-Suche* bezeichnet. Der Name bezieht sich darauf, daß im Suchbaum abwechselnd Züge des Gegners und eigene Züge in den Baumebenen wiedergespiegelt werden.

Um auf dem Baumebenen minimierte bzw. maximierte Äste zu wählen, ist zunächst eine Bewertung des jeweiligen Spielstandes notwendig. Für einfache Spiele reicht eine Bewertung auf Endzustände aus, d.h. auf Spielzustände, die keinen weiteren Zug mehr zulassen, weil z.B. das Spielfeld voll ist. Diese Zustände können mit 1, wenn der in Frage kommende Spieler gewonnen hat, mit -1, wenn der in Frage kommende Spieler verloren hat, und mit 0 im Falle eines Remis gewertet werden.

Für einen Spieler, der einen optimalen Zug sucht, sind die Ebenen, in denen der nächste Zug vom ihm getätigt wird, zu optimierende Ebenen, die anderen sind zu minimierende Ebenen. Der Algorithmus läßt sich unformell wie folgt beschreiben:

- wenn es sich um einen Finalzustand handelt:
  - dann bewerte den Zustand für den Spieler
- ansonsten:

– für jeden möglichen Zug bewerte mit Min-Max-Suche den Nachfolgezustand. Wenn es sich auf der Ebene um einen eigenen Zug handelt, wähle den Zug, der zur maximalen Bewertung führt, ansonsten den der zur minimalen Bewertung führt.

Implementierung

Wollen wir es also wagen, allgemein eine Suche nach der Gewinnstrategie eines Spiels zu programmieren. Hierzu sei eine abstrakte Klasse geschrieben, in der die Suche implementiert wird. Um was für ein konkretes Spiel es sich dabei handelt, lassen wir zunächst einmal wieder offen. Da verschiedene Spiele auch verschiedene Arten haben, einen Zug zu beschreiben, und lassen wir auch den Typ für die Spielzüge zunächst wieder offen. Wir erhalten folgende Klasse:

GameTree.scala

```
1 package name.panitz.ki
2 trait GameTree[M] extends SearchTree[M]{
```

`M` drückt dabei den noch variabel gehaltenen Typ der Spielzüge aus. Für die Spieler, wir werden ja in der Regel nur zwei haben, legen wir uns hingegen auf einen Datentyp fest und wollen `Byte`, also 8 Bit Zahlen benutzen:

GameTree.scala

```
3   type Player = Byte
```

Wir haben hier einen neuen Namen für den Typ `Byte` eingeführt. Etwas, was in Java nicht möglich wäre, in C mit `typedef` allerdings Gang und Gebe ist.

Zwei Funktionen seien vorgesehen. Eine, die den Spieler, der gerade am Zug ist, angibt:

GameTree.scala

```
4   def getCurrentPlayer():Player
```

Eine zweite, die für einen Spieler den anderen Spieler angibt:

GameTree.scala

```
5   def otherPlayer(player:Player):Player
```

das wichtigste für einen Spielzustand sind natürlich die in diesem Zustand möglichen gültigen Züge. Diese soll die folgende Funktion in einer Liste zurückgeben. Was genau ein Zug ist, haben wir vorerst offen gelassen. Wir sprechen nur vom Typ `M` bei Zügen.

Wenn wir uns im Endeffekt für einen Zug entschieden haben, den wir ausführen möchten, so sollte dieser Zug auch auf den Spielzustand ausgeführt werden, um einen Nachfolgezustand zu erhalten. Hierzu sehen wir die Methode `move` vor:

GameTree.scala

```
6   def move(i:M):GameTree[M]
```

Und wir brauchen eine Funktion, die uns sagt, wie der Spielzustand für uns denn zu bewerten ist.

GameTree.scala
```
7   def evalState(player:Player):Int
```

Die Min-Max-Suche wird für den jeweils am Zug seienden Spieler gestartet:

GameTree.scala
```
8   def minMaxSearch()=minMaxValue(getCurrentPlayer)
```

Die Min-Max-Suche hat einen Spieler als Parameter, für den die Suche gemacht wird, und gibt als Ergebnis ein Paar zurück, bestehend aus der Bewertung des Nachfolgezustandes für einen Zug:

GameTree.scala
```
9   def minMaxValue(forPlayer:Player):Pair[Int,M]={
```

Zunächst der Fall wenn bereits ein Terminalzustand vorliegt. Dann ist kein weiterer Zug zu machen, daher wird die zweite Komponente des Ergenispaares auf `null` gesetzt. Der Bewertung ist dann, die Bewertung des aktuellen Zustands:

GameTree.scala
```
10    if (terminalState) Pair(evalState(forPlayer),null:M)
```

Andernfalls ist zu prüfen, ob man sich auf einer minimier oder maximier Ebene befindet:

GameTree.scala
```
11    else{
12      val doMax = getCurrentPlayer==forPlayer
```

Davon abhängig wird der größer bzw. kleiner Vergleich gewählt und der maximale bzw minimale Bewertungswert:

GameTree.scala
```
13      val exVal=if(doMax) Integer.MIN_VALUE else Integer.MAX_VALUE
14      def comp(x:Int,y:Int)=if (doMax) (x>y) else (x<y)
```

Schließlich wird über alle möglichen Züge iteriert, diese rekursiv bewertet, und über die Ergebnisse das Maximum bzw. Minimum gewählt.

GameTree.scala
```
15      ((for (val i:M <- moves().elements)
16       yield Pair(move(i).minMaxValue(forPlayer)._1,i))
17       .foldLeft(Pair(exVal,null:M))
18                ((x,y)=>if (comp (x._1,y._1)) x else y))
19    }
20  }
```

Und das war es schon. Die Min-Max-Suche können wir zum Ziehen für unseren Agenten benutzen.

GameTree.scala
```
21  def move():GameTree[M] = move(minMaxSearch()._2)
22  }
```

TicTacToe

Jetzt soll natürlich unser erste Spielagent auch für ein konkretes Spiel angewendet werden. Eines der einfachsten Spiele ist das sogenannte *Tic Tac Toe*. Auf einem 3 mal 3 Felder großen Brett setzen abwechseln zwei Spieler ihre Spielsteine. Ziel ist es drei Steine seiner Farbe in eine Reihe zu bringen.

Wir implementieren ein `GameTree` und definieren ein paar konstante Werte. Züge sind im Fall von Tic Tac Toe Paare, die die x- und y-Koordinate des zu setzenden Spielfeldes repräsentieren.

Tic Tac Toe (auch: XXO, Kreis und Kreuz, Dodelschach oder engl. Noughts and Crosses) ist ein klassisches, einfaches Zweipersonen-Strategiespiel, dessen Geschichte sich bis ins 12. Jahrhundert v. Chr. zurückverfolgen lässt.

Spielverlauf

Auf einem 3×3 Felder großen Spielfeld machen die beiden Spieler abwechselnd ihre Zeichen (Kreuze und Kreise). Der Spieler, der als erstes drei seiner Zeichen in einer Reihe, Spalte oder einer der beiden Hauptdiagonalen setzen kann, gewinnt. Wenn allerdings beide Spieler das Spiel perfekt beherrschen, kann keiner gewinnen und es ist unentschieden.

Strategie und Taktik

Für Tic Tac Toe gibt es 255.168 verschiedene Spielverläufe, von denen 131.184 mit einem Sieg des ersten Spielers enden, 77.904 mit einem Sieg des zweiten Spielers, und 46.080 mit einem Unentschieden. Viele Spielverläufe sind äquivalent in dem Sinne, dass sie sich durch Drehungen oder Spiegelungen des Spielfelds ineinander überführen lassen. Äquivalente Verläufe zusammengefasst, reduziert sich die Zahl der verschiedenen Spielverläufe auf 26.830. Im Vergleich zu Spielen wie Go, Dame oder Schach ist dies eine verschwindend geringe Zahl. Aufgrund dieser geringen Komplexität lässt sich leicht zeigen, dass beide Spieler ein Unentschieden erzwingen können.
Die erste und zweite Spielrunde sind die wichtigsten und ausschlagebensten Runden im Spiel, um nicht zu verlieren. Wenn der Gegner beginnt, gibt es von 72 nur 44 Möglichkeiten. Erster Spieler (X) beginnt, zweiter Spieler (O) verhindert, dass X gewinnt (gespiegelte Möglichkeiten sind nicht dargestellt).

```
X| |    O| |     |X|     |X|    O|X|
-+-+-   -+-+-   -+-+-   -+-+-   -+-+-
 |O      |X|     |O|     | |     | |
-+-+-   -+-+-   -+-+-   -+-+-   -+-+-
 | |     | |     | |     |O|     | |
```

Spieltheorie

Wegen seiner Einfachheit wird Tic Tac Toe oft als Beispiel zur Erläuterung grundlegender Konzepte der Spieltheorie herangezogen. Spieltheoretisch betrachtet gehört Tic Tac Toe zu den endlichen, deterministischen Zweipersonen-Nullsummenspielen mit alternierendem Zugrecht und vollständiger Information.

Abbildung 2.10: **Wikipediaeintrag** (18. April 2006): Tic Tac Toe

TicTacToe.scala

```
1 package name.panitz.ki
2 class TicTacToe() extends GameTree[Pair[Int,Int]]{
3   val EMPTY     :Player= 0
4   val PLAYER_ONE:Player= 1
5   val PLAYER_TWO:Player= 2
6
7   val C:Int = 3
8   val R:Int = 3
```

Ein zweidimen-

sionaler Array halte das eigentliche Spielfeld und sei als leeres Spielfeld initialisiert:

TicTacToe.scala

```
9   var field = new Array[Array[Player]](C)
10  var currentPlayer = PLAYER_ONE
11
12  for (val i <- Iterator.range(0,C)){
13    field(i)=new Array[Player](R)
14    for (val j <- Iterator.range(0,R))
15      field(i)(j)=EMPTY
16  }
```

Ein Konstruktor, der einen Spielzustand kopiert, sei implementiert:

TicTacToe.scala

```
17  def this(other:TicTacToe)={
18    this()
19    for (val i<-Iterator.range(0,R);val j<-Iterator.range(0,C))
20      field(j)(i)=other.field(j)(i)
21  }
```

Es folgen die Methode zum Erfragen der Spieler:

TicTacToe.scala

```
22  def getCurrentPlayer()=currentPlayer
23  def otherPlayer(player:Player):Player
24   =if (player==PLAYER_ONE)PLAYER_TWO else PLAYER_ONE
```

Um einen konkreten Zug durchzuführen, wird der aktuelle Zustand kopiert und das entsprechende Feld mit dem aktuellen Spieler belegt:

TicTacToe.scala

```
25  def move(i:Pair[int,int]):GameTree[Pair[int,int]]={
26    val result=new TicTacToe(this)
27    result.field(i._1)(i._2)=getCurrentPlayer
28    result.currentPlayer = otherPlayer(currentPlayer)
29    result
30  }
```

Gültige Züge sind genau die Züge auf Felder, die noch leer sind.

TicTacToe.scala

```
31  def moves():List[Pair[Int,Int]]=
32    (for(val i<-Iterator.range(0,R);val j<-Iterator.range(0,C)
33        ;field(j)(i)==EMPTY) yield Pair(j,i)).toList
```

Das Spiel terminiert, wenn einer der Spieler gewonnen hat oder das Brett voll ist.

TicTacToe.scala
```
34   def terminalState():boolean =
35     wins(PLAYER_ONE)||wins(PLAYER_TWO)||boardFull()
36
37   def boardFull():boolean =
38    ((for (val i<-Iterator.range(0,R);val j<-Iterator.range(0,C))
39     yield field(i)(j)).forall((x)=>x!=EMPTY))
```

Die Bewertungsfunktion ist wie in der Einleitung angedeutet, relativ einfach gestrickt:

TicTacToe.scala
```
40   def evalState(player:Player)
41    =if (wins(player)) 1 else if(wins(otherPlayer(player))) -1 else 0
```

Sämtliche Siegespositionen für einen Spieler seien lediglich aufgelistet:

TicTacToe.scala
```
42   def wins(player:Player)=
43    (field(0)(0)==player&&field(0)(1)==player&&field(0)(2)==player)||
44    (field(1)(0)==player&&field(1)(1)==player&&field(1)(2)==player)||
45    (field(2)(0)==player&&field(2)(1)==player&&field(2)(2)==player)||
46    (field(0)(0)==player&&field(1)(0)==player&&field(2)(0)==player)||
47    (field(0)(1)==player&&field(1)(1)==player&&field(2)(1)==player)||
48    (field(0)(2)==player&&field(1)(2)==player&&field(2)(2)==player)||
49    (field(0)(0)==player&&field(1)(1)==player&&field(2)(2)==player)||
50    (field(2)(0)==player&&field(1)(1)==player&&field(0)(2)==player)
```

Die Klasse sei abgeschlossen mit einer `toString`-Methode, so daß eventuell ein Spiel ohne GUI auf der Kommandozeile möglich wird:

TicTacToe.scala
```
51   override def toString()={
52     val result = new StringBuffer()
53     for (val i<-Iterator.range(0,R)){
54       for (val j<-Iterator.range(0,C))
55          result append ("|"+field(j)(R-i-1))
56       result append ("|"+(R-i-1)+"\n")
57     }
58     for (val j<-Iterator.range(0,C)) result append "--"
59     result append "--\n"
60     for (val j<-Iterator.range(0,C))result append ("|"+j)
61     result append "|\n"
62     result.toString()
63   }
64 }
```

Im folgenden eine kleine Anwendung, die es erlaubt gegen den Rechner Tic Tac Toe zu spielen.[9] Viel Spaß beim Spielen.

[9] Bemerkenswert an dem Code, wie sich in Scala in drei Zeilen eine *repeat-until*-Schleife implementieren läßt, die so nicht Bestandteil der Sprache ist.

PlayTicTacToe.scala

```
package name.panitz.ki
object PlayTicTacToe extends Application {
  var s:GameTree[Pair[Int,Int]]=new TicTacToe()
  Console println s
  while (!s.terminalState()){
    Console println "Lassen Sie mich meinen Zug ueberlegen."
    s=s.move()
    Console println s
    if (!s.terminalState()){
      var userMove:Pair[Int,Int]=null
      repeat{
        Console println "Ihr Zug"
        Console print " x: ";val x=Console.readInt
        Console print " y: ";val y=Console.readInt
        userMove =Pair[Int,Int](x,y)
      } until {Console println userMove;Console println (s.moves());
        s legalMove userMove}
      s=s move userMove
      Console println s
    }
  }

  def repeat(body: => Unit)=new RepeatUntilCond(body)
  class RepeatUntilCond(body: => Unit) {
    def until(cond: => Boolean):Unit={body;if (!cond)until(cond);}
  }
}
```

2.4.2 Alpha-Beta-Suche

AlphaBetaTree.scala

```
package name.panitz.ki
trait AlphaBetaTree[M] extends GameTree[M]{
  override def move():AlphaBetaTree[M]
   =move(alphaBetaSearch()._2).asInstanceOf[AlphaBetaTree[M]]

  def theDepth()= -1
  def alphaBetaSearch():Pair[Int,M]=alphaBetaSearch(theDepth)

  def alphaBetaSearch(depth:Int):Pair[Int,M]=
   maxValue(getCurrentPlayer,0,depth,Integer.MIN_VALUE,Integer.MAX_VALUE)

  def boardFull()=moves.isEmpty
  def move(i:M):AlphaBetaTree[M]


/*  def minMaxValue(forPlayer:Player,depth:Int,alpha:int,beta:Int):Pair[Int,M]
      var a=alpha
```

```
18        var b=beta
19        if (terminalState|depth==maxDepth) Pair(evalState(forPlayer),null:M)
20        else{
21          val doMax = getCurrentPlayer==forPlayer
22
23          val exVal=if(doMax) Integer.MIN_VALUE else Integer.MAX_VALUE
24          def comp(x:Int,y:Int)=if (doMax) (x>y) else (x<y)
25
26          ((for (val i:M <- moves().elements)
27           yield Pair(move(i).minMaxValue(forPlayer)._1,i))
28           .foldLeft(Pair(exVal,null:M))
29                    ((x,y)=>if (comp (x._1,y._1)) x else y))
30        }
31
32    }
33  */
34
35
36    def maxValue(forPlayer:Byte,currentDepth:Int,maxDepth:Int,a:Int,beta:Int):Pa
37      var alpha=a
38      if (terminalState) {
39        if (boardFull) Pair[Int,M](evalState(forPlayer),null)
40        else Pair[Int,M](-100000*(currentDepth+1),null)
41      }else if (currentDepth==maxDepth) {
42        val s= evalState(forPlayer)
43        Pair[Int,M](s,null)
44      }else {
45        var v = Integer.MIN_VALUE
46        var m:M=null
47        var notReady=true
48        for (val i:M <- moves().elements;notReady) {
49          if(notReady){
50            val s=move(i)
51            val currentVal=s.minValue(forPlayer,currentDepth,maxDepth,alpha,beta
52            if (currentVal>v){v=currentVal;m=i}
53            if (v>=beta) {notReady=false}
54            else {alpha = Math.max(alpha, v)}
55          }
56        }
57        Pair[Int,M](v,m)
58      }
59    }
60
61    def minValue(forPlayer:Byte,currentDepth:Int,maxDepth:Int,alpha:Int,b:Int):P
62      var beta=b
63      if (terminalState) {
64        if (boardFull) Pair[Int,M](evalState(forPlayer),null)
65        else Pair[Int,M](100000/(currentDepth+1),null)
66      }else {
67        var v = Integer.MAX_VALUE
```

```
68        var m:M=null
69        for (val i:M <- moves()) {
70          var notReady=true
71          if(notReady){
72            val s=move(i)
73            val currentVal=s.maxValue(forPlayer,currentDepth+1,maxDepth,alpha,be
74            if (currentVal<v){v=currentVal;m=i}
75            if (v<=alpha) notReady=false
76            else {beta = Math.min (beta, v)}
77          }
78        }
79        Pair[Int,M](v,m)
80      }
81    }
82  }
```

Vier Gewinnt

Util.scala

```
1  package name.panitz.ki;
2  object Util {
3    def and(xs:List[Boolean]):Boolean
4     = (false /: xs ) {(x:Boolean,y:Boolean)=>x&&y}
5    def or(xs:List[Boolean])
6     = (false /: xs ) {(x:Boolean,y:Boolean)=>x||y}
7    def flatten[a](xs:List[List[a]]):List[a]=
8     (xs:\ (Nil:List[a])) {(x,xs)=>x::xs}
9    def merge[a](xs:List[a],ys:List[a]):List[a]=
10     if (xs.isEmpty) ys
11     else if (ys.isEmpty) xs
12     else (xs.head) :: (ys.head) :: merge(xs.tail,ys.tail)
13 }
```

ForInARowObject.scala

```
1  package name.panitz.ki
2  object ForInARowObject{
3    type Field=Array[Array[byte]]
4    val COLUMNS:Int = 7
5    val ROWS:Int = 6
6    val OVERALL_MOVES=COLUMNS*ROWS
7    val EMPTY:byte = 0
8    val PLAYER_ONE:byte=1
9    val PLAYER_TWO:byte=10
10 }
```

FourKI.java

```
1  package name.panitz.ki;
2  public interface FourKI{
3   public int nextMove();
```

```
4   public void nextMove(int i);
5  }
```

FourSkript.scala

```
1  package name.panitz.ki;
2  class FourSkript extends FourKI{
3    var game = new FourInARow().init()
4
5    def  nextMove():int={
6      game=game.move().asInstanceOf[FourInARow]
7      game.lastColumn
8    }
9    def  nextMove(i:int)={
10     game=game.move(i).asInstanceOf[FourInARow]
11   }
12 }
```

FourInARow.scala

```
1  package name.panitz.ki
2  class FourInARow() extends AlphaBetaTree[Int]{
3  import Util.{_}
4  import ForInARowObject.{_}
5  import scala.collection.mutable.{_}
6    var movesDone = 0
7    var field:  Field = new Array[Array[byte]](COLUMNS)
8    var heights: Array[Byte] = new Array[byte](COLUMNS)
9
10   var _myMove:boolean=false
11   var lastRow= -1
12   var lastColumn= -1
13
14   {var i= 0;
15    while (i<COLUMNS) {field(i)=new Array[byte](ROWS);i=i+1}
16   }
17
18   def fieldInit()={
19     for (val i <- List.range(0,COLUMNS);val j <- List.range(0,ROWS))
20       field(i)(j)=EMPTY
21     this
22   }
23
24   def init()={
25     fieldInit()
26     var tmp:FourInARow =  new FourInARow().fieldInit()
27     this
28   }
29
30   def this(other:FourInARow)={
31     this()
```

```
32      var i=0
33      while (i<ROWS){
34        var j=0
35        while (j<COLUMNS){
36          field(j)(i)=other.field(j)(i)
37          j=j+1
38        }
39        i=i+1
40      }
41      movesDone=other.movesDone+1
42      var j = 0;
43      while (j<COLUMNS) {heights(j)=other.heights(j);j=j+1}
44    }
45
46    override def theDepth()=
47      if (movesDone<10) 3
48      else if (movesDone<20) 4
49      else if (movesDone<30) 5
50      else if (movesDone<38) 6
51      else 7
52
53    def myMove():boolean  = _myMove
54    def getCurrentPlayer()=if (myMove) PLAYER_ONE else PLAYER_TWO
55    def otherPlayer(player:Byte)=if (player==PLAYER_ONE)PLAYER_TWO else PLAYER_O
56    def getLastPlayer()   =if (myMove) PLAYER_TWO else PLAYER_ONE
57
58    def move(i:int):AlphaBetaTree[Int]={
59      val result=new FourInARow(this)
60      result.lastColumn=i
61      result.lastRow=heights(i)
62      result.field(i)(heights(i))=getCurrentPlayer
63      result.heights(i)=(heights(i)+1).asInstanceOf[Byte]
64      result._myMove = !myMove
65      result
66    }
67
68    def moves():List[Int]={
69      var result:List[Int]=List[Int]()
70      {var i=0
71       while (i<COLUMNS){
72        if (heights(i)<ROWS) result=result.:: (i)
73        i=i+1
74      }}
75      //preferring middle columns over side columns
76      result.sort((x,y)=>heights(x)>heights(y))
77    }
78
79    def terminalState():boolean=boardFull || wins(getLastPlayer())
80    override def boardFull():boolean= movesDone==OVERALL_MOVES
81
```

```
82  override def toString()={
83    val result = new StringBuffer()
84    for (val i<-List.range(0,ROWS)){
85      for (val j<-List.range(0,COLUMNS))
86         result append ("|"+(if (field(j)(ROWS-i-1)==10) "2" else (field(j)(RO
87      result append "|\n"
88    }
89    for (val j<-List.range(0,COLUMNS)) result append "--"
90    result append "-"
91    result append "\n"
92    result append ("Suchtiefe: "+theDepth+"  Zuege bisher: "+movesDone)
93    result.toString()
94  }
95
96  def evalState(player:Byte)
97  ={if (wins(getLastPlayer)) {if (getLastPlayer()==player) 100000 else -100000
98     else state(player)-10*state(otherPlayer(player)) }
99
100 def evalState(player:Byte,currentDepth:Int):Int={
101   val es=evalState(player)
102   if (es==100000) es/(currentDepth+1)
103   if (es== -100000) es*(currentDepth+1) else es
104 }
105
106 def wins(p:Byte):Boolean={
107   val c=lastColumn
108   val r=lastRow
109   val v=4*p
110   if (lastRow== -1) false
111   else {
112   ((r>2 && field(c)(r)==p &&field(c)(r-1)==p &&field(c)(r-2)==p &&field(c)(r
113
114    || (c<4 && field(c)(r)==p &&field(c+1)(r)==p &&field(c+2)(r)==p &&field(c
115    || (c<5 && c>0 && field(c-1)(r)==p &&field(c)(r)==p &&field(c+1)(r)==p &&
116    || (c<6 && c>1 && field(c-2)(r)==p &&field(c-1)(r)==p &&field(c)(r)==p &&
117    || (c>2 && field(c-3)(r)==p &&field(c-2)(r)==p &&field(c-1)(r)==p &&field
118
119    || (c<4&&r<3&&field(c)(r)==p &&field(c+1)(r+1)==p &&field(c+2)(r+2)==p &&
120    || (c<5&& c>0 && r<4 && r>0
121        &&field(c-1)(r-1)==p &&field(c)(r)==p &&field(c+1)(r+1)==p &&field(c+
122    || (c<6&& c>1 && r<5 && r>1
123        &&field(c-2)(r-2)==p &&field(c-1)(r-1)==p &&field(c)(r)==p &&field(c+
124    || (c>2  && r>2
125        &&field(c-3)(r-3)==p &&field(c-2)(r-2)==p &&field(c-1)(r-1)==p &&fiel
126
127    ||(r>2 && c< 4
128        &&field(c)(r)==p &&field(c+1)(r-1)==p &&field(c+2)(r-2)==p &&field(c+
129    ||(r>1 && r<5 && c< 5 && c>0
130        &&field(c-1)(r+1)==p &&field(c)(r)==p &&field(c+1)(r-1)==p &&field(c+
131    ||(r>0 && r<4 && c< 6 && c>1
```

```
132           &&field(c-2)(r+2)==p &&field(c-1)(r+1)==p &&field(c)(r)==p &&field(c+
133       ||(r<3 && c>2
134           &&field(c-3)(r+3)==p &&field(c-2)(r+2)==p &&field(c-1)(r+1)==p &&fiel
135     }
136   }
137
138 def points(player:Byte,rowValue:Int)
139  =  if (rowValue==3*player) 100 else if (rowValue==2*player) 10 else 0
140
141 def state(p:Byte):Int={
142    val n1 = 3
143    val n = 4
144    var result=0
145    {var c=0
146     while (c<COLUMNS-n1){
147       var r=0
148       while (r<ROWS-n1){
149         result
150           = {result
151              +points(p,field(c)(r)+field(c)(r+1)+field(c)(r+2)+field(c)(r+3))
152              +points(p,field(c)(r)+field(c+1)(r+1)+field(c+2)(r+2)+field(c+3)(
153              +points(p,field(c)(r)+field(c+1)(r)+field(c+2)(r)+field(c+3)(r))
154         r=r+1
155       }
156       c=c+1
157     }
158    }
159
160    {var c=0
161     while (c<COLUMNS-n1){
162       var r=ROWS-n1
163       while (r<ROWS){
164         result
165          =result+points(p,field(c)(r)+field(c+1)(r)+field(c+2)(r)+field(c+3)(r
166         r=r+1
167       }
168       c=c+1
169     }
170    }
171
172    {var c=COLUMNS-n1
173     while (c<COLUMNS){
174       var r=0
175       while (r<ROWS-n1){
176         result
177          =result+points(p,field(c)(r)+field(c)(r+1)+field(c)(r+2)+field(c)(r+3
178         r=r+1
179       }
180       c=c+1
181     }
```

```
182     }
183
184     {var c=0
185      while (c<COLUMNS-n1){
186        var r=n1
187        while (r<ROWS){
188          result=result+points(p,field(c)(r)+field(c+1)(r-1)+field(c+2)(r-2)+fie
189          r=r+1
190        }
191        c=c+1
192      }
193     }
194     result
195   }
196 }
```

PlayFourInARow.scala

```
1  package name.panitz.ki
2  object PlayFourInARow extends Application {
3      var s:AlphaBetaTree[Int]=new FourInARow().init()
4      Console.println(s)
5      while (!s.terminalState()){
6        s=s.move()
7        Console println s
8        if (!s.terminalState()){
9          s=s move Console.readInt
10         Console println s
11       }
12     }
13 }
```

2.5 Aufgaben

Aufgabe 1 Betrachten Sie das folgende Problem:

> Bert, Bengt, Ernie und Cindy wollen einen Fluß überqueren. Sie haben nur ein Boot. Bert und Bengt sind so schwer, daß Sie nur alleine mit dem Boot fahren können. Ernie und Cindy können hingegen auch zusammen im Boot fahren.

In dieser Aufgabe sollen Sie das Problem als Suchproblem modellieren.

a) geben Sie eine möglichst einfache Modellierung für die auftretenden Zustände an.

b) Zeichnen Sie den vollständigen Suchgraphen.

c) Geben Sie die Lösung für das Problem an.

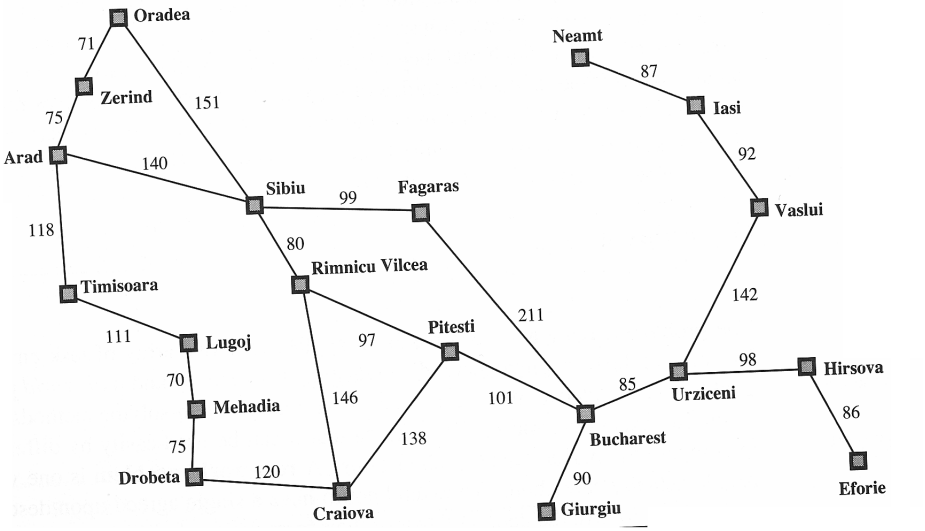

Abbildung 2.11: Entfernungen zwischen einzelnen Orten Rumaeniens (aus [RN95]).

Aufgabe 2 Gegeben sei die Karte von Rumänien aus Abbildung 2.11.

An Straßen steht jeweils die Entfernung zwischen den verbundenen Orten. Zusätzlich sei in folgender Tabelle die Luftlinie von jedem Ort nach Bukarest bekannt.

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Zeigen Sie, wie der A*-Algorithmus den kürzesten Weg von Arad nach Bukarest findet.

Kapitel 3

Logik

Bengt: Hallo Bert. Was sitzen Sie da so melancholisch wie ein Forsthaus im Walde?

Bert: Ach, wissen Sie, es geht um meinen Freund Ernie.

Bengt: Ich dachte immer, Sie verstehen sich recht gut miteinander.

Bert: Die Sache ist nur, daß Ernie angefangen hat, Gedichte zu schreiben.

Bengt: Nun, das ist doch ein recht leises und angenehmes Hobby.

Bert: Nur Ernie hat mir seine Gedichte zum Lesen gegeben und nach meiner Meinung gefragt.

Bengt: Und.

Bert: Naja, sie sind ziemlich langweilig aber das möchte ich nicht so direkt sagen, schließlich ist er ja mein Freund.

Bengt: Sind die wirklich so schlimm?

Bert: Ja, stellen Sie sich vor, alle seine Gedichte sind über Seifenblasen.

Bengt: Aber das kann doch ganz nett sein.

Bert: Kann, muß aber nicht. Warten sie:

> Oh Seifenblase, Oh Seifenblase,
> In dir spiegelt sich ein kleiner Hase
> Du leuchtest in allen Farben so bunt
> In dir spiegelt sich auch ein kleiner Hund

Bengt: Oh ja, das ist tatsächlich etwas langweilig.

Bert: Aber das muß man ihm ja nicht so direkt sagen.

Bengt: Dann drücken Sie das ihm gegenüber doch etwas verklausuliert aus, etwa so:

- Wirklich interessante Gedichte sind nicht unbeliebt bei Leuten mit guten Geschmack.
- Keine moderne Poesie ist frei von einer gewissen Schwülstigkeit.
- Alle Deine Gedichte sind nun mal über Seifenblasen.

- Schwülstige Poesie ist nicht beliebt bei Leuten mit einen gewissen Geschmack.
- Kein älteres Gedicht beschäftigt sich mit Seifenblasen.

BERT: Das klingt alles sehr allgemein dahergesagt. Ich glaube, wenn ich Ernie es so sage, würde er nicht eingeschnappt sein.

BENGT: Hoffen wir nur das Ernie nicht zuviel Lewis Carrol liest.

Eine der Kernaufgaben eines intelligenten Agenten besteht darin, aus vorhandenen Basiswissen neues Wissen abzuleiten; aus bestehenden Fakten über die Umgebung, weitere Schlüsse zu ziehen oder Pläne zu kreieren. Die Frage, wie der Vorgang des folgerichtigen Schließens genauer präzisiert werden kann, stellten sich schon die alten Griechen. Aristoteles stellte hierzu ein formales System von Schlußregeln auf, die immer zu folgerichtigen Schlüssen führen. Damit begründete Aristoteles die formale Logik.

Je nach den technischen Möglichkeiten, interessertierte die Frage, ob sich die formalen Schlußsysteme der Logik, maschinell zum logischen Schließen ausführen lassen. So verwundert es nicht, daß die KI sehr früh die formale Logik als eines der Hauptwerkzeuge anektierte und entscheidend für seine Zwecke weiterentwickelte.

Charles Lutwidge Dodgson (∗27. Januar 1832 in Daresbury; †14. Januar 1898 in Guildford), besser bekannt unter seinem Künstlernamen Lewis Carroll, war ein britischer Schriftsteller, Mathematiker und Fotograf.
Er ist der Autor von *Alice im Wunderland*, *Alice hinter den Spiegeln* (auch bekannt als *Alice im Spiegelland*) und *Die Jagd nach dem Schnark*. Mit seiner Befähigung für Wortspiel, Logik und Fantasie schaffte er es, weite Leserkreise – von den Naivsten zu den Gebildetsten – zu fesseln. Seine Werke sind bis heute populär geblieben und haben nicht nur die Kinderliteratur, sondern auch Schriftsteller wie James Joyce oder Douglas R. Hofstadter beeinflusst.

Abbildung 3.1: **Wikipediaeintrag** (24. April 2006): Lewis Carroll

Ein Mathematiker, der sich recht unterhaltsam mit logischen Rätsel beschäftigte, war Lewis Carroll, der unzählige von logischen Spielereien und verklausulierten Rätsel aufgestellt hat. Auch die diesem Kapitel vorangestellte in fünf Sätzen verklausulierte Aussage über bestimmte Gedichte stammt von Lewis Carroll.

3.1 Aussagenlogik

Das grundlegendste logischen System, von dem komplexere formale Systeme abgeleitet werden können, ist die Aussagenlogik, die einem Programmierer in jeder Programmiersprache als der Datentyp `boolean` begegnet. Wir werden in diesem Kapitel die Aussagenlogik auf unterschiedliche Weise betrachten. Dabei werden viele Aspekte eines formalen Systems exemplarisch vorgeführt.

Ein formales System besteht in der Regel aus einer Sprache, in der bestimmtes Wissen ausgedrückt werden kann. Diese Sprache ermöglicht es syntaktische Sätze, oder Formeln aufzuschreiben. Zur Unterscheidung zu anderen Sprachen, insbesondere der Sprache, in der wir über die Formelsprache Beweise führen, bezeichnen wir die Formelsprache als *Objektsprache*. Wenn Beweise über Eigenschaften der Objektsprache geführt werden, so werden diese Beweise in einer Metasprache geführt.

Für eine Objektsprache ist eine Semantik zu definieren. Die Semantik wird in der Regel eine mathematische Definition sein, in der jeder Satz der Objektsprache auf bestimmte Elemente mathematischer Mengen abgebildet wird. Die Semantik kann definieren, welche neuen Formeln aus einer Menge von Formeln folgt.

Desweiteren ist für ein formales System ein Kalkül zu definieren. Ein Kalkül besteht aus Rechenregeln. Diese Regeln arbeiten auf den syntaktischen Objekten der Objektsprache und geben an, wie durch symbolisches Rechnen neue Formeln aus einer Menge von Formeln abzuleiten ist.

George Boole (∗2. November 1815 in Lincoln, England; †8. Dezember 1864 in Ballintemple, Irland) war ein englischer Mathematiker (Autodidakt) und Philosoph.
Ursprünglich als Lehrer tätig, wurde er auf Grund seiner wissenschaftlichen Arbeiten 1848 Mathematikprofessor am Queens College in Cork (Irland).
Boole entwickelte 1847 den ersten algebraischen Logikkalkül für Klassen und Aussagen im modernen Sinn. Damit begründete er die moderne mathematische Logik. Booles Kalkül wird gelegentlich noch im Rahmen der traditionellen Begriffslogik interpretiert, obwohl Boole bereits in *The Mathematical Analysis of Logic* ausschließlich von Klassen spricht.
Boole arbeitete in einem kommutativen Ring mit Multiplikation und Addition, die den logischen Verknüpfungen *und* und *entweder...oder* entsprechen; in diesem booleschen Ring gilt zusätzlich zu den bekannten Rechenregeln die Regel aa=a, in Worten (a und a)=a.
In der Boole-Schule wurde die zum booleschen Ring gleichwertige boolesche Algebra entwickelt, die mit *und* und *oder* arbeitet. Sie ist heute in der Aussagenlogik und Mengenlehre bekannter und verbreiteter als der originale, rechnerisch elegantere boolesche Ring.
George Boole ist der Vater von Ethel Lilian Voynich.
Auf George Boole sind die booleschen Variablen zurückzuführen, die zu den Grundlagen des Programmierens in der Informatik zählen.

Abbildung 3.2: **Wikipediaeintrag** (24. April 2006): George Boole

Zwischen einem Kalkül und der Semantik einer Sprache sind zwei fundamentale Eigenschaften zu beweisen. Zum einem soll der Kalkül nur korrekte Ableitungen machen. Jeder Satz, der sich durch die Operationen eines Kalküls aus einer Formelmenge ableiten läßt, soll auch semantisch aus dieser Formelmenge folgern. Ist diese Eigenschaft erfüllt, so ist der Kalkül korrekt. Nun ist es natürlich recht leicht einen korrekten Kalkül für ein beliebiges formales System zu schreiben. Hierzu nehme man einfach den Kalkül, der überhaupt keine Sätze ableiten kann. Ein Kalkül sollte aber möglichst viele Schlußfolgerungen beweisen können, am besten sollte er alles, was semantisch aus einer Formelmenge folgt, aus dieser auch ableiten können. Wenn dieses der Fall ist, so bezeichnet man den Kalkül als vollständig. Abbildung 3.3 gibt einen schematischen Überblick über die Bestandteile eines formales Systems.

In dieser Weise stellt auch jede Programmiersprache ein formales System dar. Der Kalkül einer Programmiersprache ist ihr Ausführungsmodell. Die Semantik einer Programmiersprache ist in den seltesten Fällen eigens definiert. Daher fällt es auch schwer Korrektheitsbeweise über Programme zu führen. Statt von Semantik und Kalkül spricht man bei Programmiersprachen auch von *denotionaler* und *operationaler* Semantik.

3.1.1 Syntax

Die Aussagenlogik ist eine Sprache in der aussagenlogische Formeln ausgedrückt werden können. Informatiker sind es gewohnt Sprachen über einer kontextfreie Grammatik auszudrücken. Wir werden im Folgenden eine eher mathematische Definition der Sprache der aussagenlogischen Formeln benutzen. Wir folgen dabei einem Vorlesungsskript der Universität Karlsruhe[Spe86].

Definition 3.1.1 (Syntax der Aussagenlogik)
Die Menge $\{(,), \neg, \rightarrow\}$ *sei die Menge der Sondersymbole.*

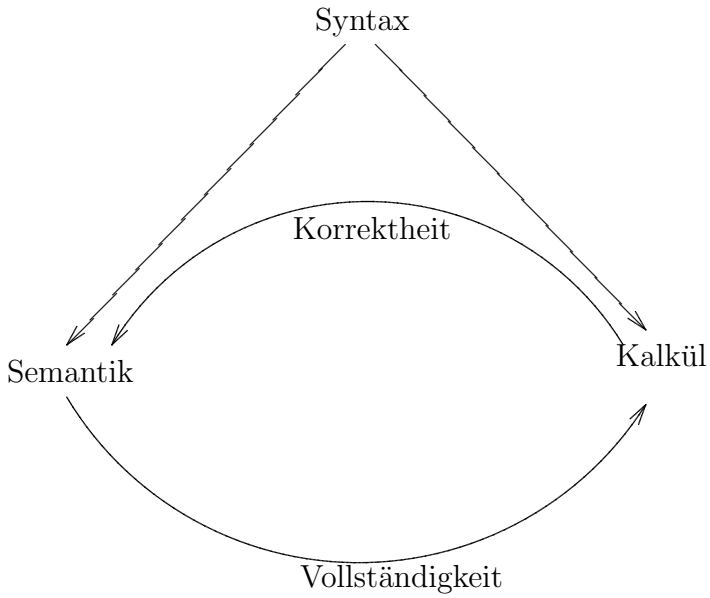

Abbildung 3.3: Syntax, Semantik, Kalkül

Für eine abzählbare Menge $S = \{A, B, C \ldots\}$ von Aussagenlogischen Variablen sei die Sprache For_S definiert als die kleineste Menge mit:

- $S \subset For_S$
- *wenn* $x \in For_S$ *dann auch* $\neg x \in For_S$
- *wenn* $x \in For_S$ *und* $x \in For_S$ *dann auch* $(x \rightarrow y) \in For_S$

Die Menge S wird als Signatur bezeichnet. Formeln x mit $x \in S$ werden als Atome bezeichnet. Die Menge $S \cup \{\neg x | x \in S\}$ wird als die Menge der Literale bezeichnet.

Wie man sieht, halten wir unsere Sprache extrem klein. Wir verzichten auf die üblichen aussagenlogischen Junktoren für *oder* und *und*, $\vee$ und $\wedge$ und begnügen uns mit dem Symbol $\rightarrow$ für die logischen Implikation und $\neg$ für die Negation. $\vee$ und $\wedge$ werden wir lediglich als Abkürzungen für komplexere Ausdrücke benutzen:

Definition 3.1.2 ($\vee$ und $\wedge$)
Ein Ausdruck der Form $(A \vee B)$, $A, B \in For_S$ stehe für: $(\neg A \rightarrow B)$.
Ein Ausdruck der Form $(A \wedge B)$, $A, B \in For_S$ stehe für: $\neg(\neg A \vee \neg B)$.
Ein Ausdruck der Form $(A \leftrightarrow B)$, $A, B \in For_S$ stehe für: $((A \rightarrow B) \wedge (B \rightarrow A))$,

Unsere aussagenlogischen Formeln sind mit obiger Definition vollständig geklammert. Wie wir es bei arithmetischen Ausdrücken gewohnt sind, bei denen die Punktrechnungsoperatoren stärker binden als Strichrechnungsoperatoren, und somit Klammern vermieden werden können, sei ein Operatorpriorität auf den aussagenlogischen Operatoren in folgender Reihenfolge gegeben: $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$

3.1.2 Semantik

Wir haben im voherigen Abschnitt eine Sprache definiert. Die Ausdrücke der Sprache For_S haben bisher noch keine Bedeutung für uns. Diese definieren wir nun. Hierzu werden Interpretationen für Ausdrücke der Sprache For_S definiert.

Definition 3.1.3 (Interpretation)
Eine Abbildung $I : S \to \{W, F\}$ heißt Interpretation. Eine Interpretation wird auf die folgende Weise zu einer Abbildung $w_I : For_S \to \{W, F\}$ erweitert:

$$\begin{aligned} w_I(x) &= I(x) \quad \text{für } x \in S \\ w_I(\neg x) &= \begin{cases} W & \text{für } w_I(x) = F \\ F & \text{sonst} \end{cases} \\ w_I((x \to y)) &= \begin{cases} W & \text{für } w_I(y) = W \text{ oder } w_I(x) = F \\ F & \text{sonst} \end{cases} \end{aligned}$$

Eine Interpretation ist also eine Belegung der aussagenlogischen Variablen einer Formel mit den Werten W und F. Für diese Belegung bekommt dann die gesammte Formel einen Wahrheitswert berechnet. Hierzu war es nur notwendig für die beiden Operatoren $\neg$ und $\to$ zu definieren, wie für sie den Wahrheitswert abhängig von der Teilformel lautet.

Es schließen sich ein paar einfache Begriffe an:

Definition 3.1.4 *Sei $A \in For_S$.*

- *A ist eine* Tautologie (allgemeingültig) *gdw. für alle Interpretationen I gilt: $w_I(A) = W$.*
- *A ist* ein Widerspruch (widersprüchlich, unerfüllbar) *gdw. für alle Interpretationen I gilt: $w_I(A) = F$.*
- *A ist* erfüllbar (konsistent) *gdw. es eine Interpretationen I gibt mit: $w_I(A) = W$.*
- *ein* Modell *für eine Formel A ist eine Interpretation I mit $w_I(A) = W$.*

In der Aussagenlogik sind wir in der glücklichen Lage, daß wir in einer Formel nur endlich viele aussagenlogische Variablen haben. Für n aussagenlogische Variablen gibt es 2^n verschiedene Interpretationen. Somit können wir in Form von Wahrheitstafel alle Interpretaionen einer Formel hinschreiben und damit prüfen, ob es eine Tautologie ist. Allerdings ist das mit einem exponentiellen Aufwand verbunden.

Aufgabe 3 Zeigen Sie: $w_I((A \vee B)), w_I((A \wedge B)), w_I((A \to B)), w_I((A \leftrightarrow B))$ berechnen sich aus $w_I(A)$ und $w_I(B)$ wie folgt:

$w_I(A)$	$w_I(B)$	$w_I((A \vee B))$	$w_I((A \wedge B))$	$w_I((A \leftrightarrow B))$
W	W	W	W	W
W	F	W	F	F
F	W	W	F	F
F	F	F	F	W

Beispiel 3.1.5 *Bauernregeln:*

- *"Abendrot Schlechtwetterbot" kann man übersetzen in*
  Abendrot → Schlechtes_Wetter. Diese Aussage ist weder Tautologie noch Widerspruch, aber erfüllbar.
- *"Wenn der Hahn kräht auf dem Mist, ändert sich das Wetter oder es bleibt wie es ist." kann man übersetzen in*
  Hahn_kraeht_auf_Mist → (Wetteraenderung ∨ ¬Wetteraenderung).
  Man sieht, dass das eine Tautologie ist.

Beispiel 3.1.6 *Beispiele für ein paar allgemeingültige Formelschemata:*

- $A \wedge A \leftrightarrow A$ *(Idemtpotenz)*
- $A \vee A \leftrightarrow A$ *(Idemtpotenz)*
- $(A \wedge B) \wedge C \leftrightarrow A \wedge (B \wedge C)$ *(Assoziativität)*
- $(A \vee B) \vee C \leftrightarrow A \vee (B \vee C)$ *(Assoziativität)*
- $A \wedge B \leftrightarrow B \wedge A$ *(Kommutativität)*
- $A \vee B \leftrightarrow B \vee A$ *(Kommutativität)*
- $A \wedge (B \vee C) \leftrightarrow (A \wedge B) \vee (A \wedge C)$ *(Distributivität)*
- $A \vee (B \wedge C) \leftrightarrow (A \vee B) \wedge (A \vee C)$ *(Distributivität)*
- $A \wedge (A \vee B) \leftrightarrow A$ *(Absorption)*
- $A \vee (A \wedge B) \leftrightarrow A$ *(Absorption)*
- $\neg\neg A \leftrightarrow A$ *(Doppelnegation)*
- $\neg(A \wedge B) \leftrightarrow \neg A \vee \neg B$ *(de Morgan)*
- $\neg(A \vee B) \leftrightarrow \neg A \wedge \neg B$ *(de Morgan)*

Die obigen allgemeingültigen Formeln beinhalten alle das Äquivalenzsymbol $\leftrightarrow$ als obersten Operator. Wir werden diese Äquivalenzen benutzen um Formeln umzuformen.

Nachdem wir nun Formeln interpretiert haben, indem sie unter einer Interpretation euf einen der Werte W und F abgebildet werden, benutzen wir diese Interpretationen, um zu definieren, wann aus einer Formelmenge eine weitere Formel semantisch folgt.

Definition 3.1.7 (semantische Folgerbarkeit)
Sei $M \subset For_S$ und $A \in For_S$. Aus M sei A folgerbar im Zeichen $M \models_S A$ genau dann wenn:
für jede Interpretation I, für die für jedes $B \in M$ gilt $w_I(B) = W$, gilt auch $w_I(A) = W$.

Im Zusammenhang mit einem rationalen Agenten, identifiziere man die Menge M aus der Definition der Folgerbarkeit mit dem Wissen des Agenten über die Welt. Das ist zum einem Faktenwissen zum anderen Wissen über allgemeine Zusammenhänge in diesem Faktenwissen. Der rationale Agent wird dann versuchen neues aus seinem bestehenden Wissen folgerbares Wissen abzuleiten. Das Wissen, das aus seinem Grundwissen folgt, ist in der obigen Definition die Formel A.

Für den Fall, daß zwei Formeln jeweils auseinander folgern, führen wir die Definition der *äquivalenz* ein.

Definition 3.1.8 (logisch äquivalent)
Für $A, B \in For_S$ sei A logisch äquivalent zu B genau dann wenn: $\{A\} \models_S B$ und $\{B\} \models_S A$.

Für Formeln A mit $\emptyset \models A$ gilt, daß A allgemeingültig ist.

Es gilt das kleine praktische Lemma:

Lemma 3.1.9 *A ist logisch äquivalent zu B genau dann wenn $\{\} \models A \leftrightarrow B$.*

Oder anders ausgedrückt $A \leftrightarrow B$ ist eine Tautologie, genau dann wenn A logisch äquivalent zu B ist.

3.1.3 Kalküle

Wie werden zwei Beispiele für Kalküle der Aussagenlogik vorstellen. Zunächst einen Kalkül, der versucht ähnlich wie in Idealfall ein Mathematiker, den formalen Beweis einer Formel mit einer Kette kleiner aufeinanderfolgender Schritte zu führen; und ein zweiter Kalkül, der möglichst gut als einfacher Algorithmus umgesetzt und automatisiert werden kann.

Kalkül des natürlichen Schließens

> Mathematik, so lautet die gängige Meinung, beruht im Gegensatz zu den Naturwissenschaften nicht auf Erfahrung, sondern auf reiner Logik. Aus einer überschaubaren Menge von Grundannahmen, so genannten Axiomen, finden die Mathematiker durch die Anwendung logischer Schlussregeln zu immer neuen Erkenntnissen, dringen immer tiefer ins Reich der mathematischen Wahrheit vor. Die menschliche Subjektivität (der Geisteswissenschaft) und die schmutzige Realität (der Naturwissenschaft) bleiben außen vor. Es gibt nichts Wahreres als die Mathematik, und vermittelt wird uns diese Wahrheit durch den Beweis. So ein Beweis ist zwar von Menschen gemacht, und es erfordert Inspiration und Kreativität, ihn zu finden, aber wenn er einmal dasteht, ist er unumstößlich. Allenfalls kann er durch einen einfacheren oder eleganteren ersetzt werden.
>
> *Christoph Drösser, Die Zeit 27.4.2006*

Obiges Zitat aus einem Wochenzeitungsartikel beschreibt, wie im Idealfall ein mathematischer Beweis aussieht. Der Kalkül, der in diesem Abschnitt vorgestellt wird, spiegelt genau diese Idee, Beweise zu führen, wider. Daher definieren wir zunächst Axiome, die wir im Kalkül des natürlichen Schließens benutzen wollen:

Definition 3.1.10 (Axiome)
Die Menge $\mathcal{A}\S$ der Axiome über eine Signatur S sei die Vereinigung der folgenden drei Mengen:

- $A_1 = \{(A \to (B \to A)) | A, B \in For_S\}$
- $A_2 = \{(A \to (B \to C)) \to ((A \to B) \to (A \to C))) | A, B, C \in For_S\}$
- $A_3 = \{((A \to B) \to (\neg A \to \neg B)) | A, B \in For_S\}$

Als kleine Übung läßt sich folgene Behauptung beweisen:

Lemma 3.1.11 *Alle Axiome sind Tautologien.*

Man vergegenwärtige sich noch einmal, daß es nicht drei Axiome gibt, sondern unendlich viele Axiome, die alle nach einem der drei obigen Schemata gebildet sind.

Im Kalkül des logischen Schließens wird eine Beweiskette aufgebaut: In dieser Kette dürfen Axiome auftauchen, Formeln des Basiswissens, aus dem neues Wissen abgeleitet werden soll, und über eine bestimmte Schlußregel erhaltene neue Schlüsse.

Definition 3.1.12 (syntaktische Ableitbarkeit)
Sei $M \subset For_S$ und $A \in For_S$. Aus M sei A ableitbar im Zeichen $M \vdash_S A$ genau dann wenn:
es existiert eine Folge $(A_1, \ldots, A_n)$ mit $A_i \in For_S$, so daß für jedes A_i gilt:

- $A_i \in M$
- *oder* $A_i \in \mathcal{A}\S$
- *oder es existieren* $j, k < i$ *mit* $A_j = (A_k \to A_i)$

Der letzte Punkt in dieser Definition ist die eigentliche Schlußregel. Sie ist eine der Schlußregeln, die bereits Aristoteles aufgestellt hat, und wir seit der Antike als *modus ponens* bezeichnet. Sie hat die schematische Form:

$$\frac{A \quad A \to B}{B}$$

Diese Schreibweise von logischen Schlußregeln ist wie folgt zu lesen: wenn die Formeln oberhalb des Querstriches angenommen werden können, so darf die Formel unterhalb des Querstriches geschlossen werden.

Für die erste unserer beiden Bauernregeln kommt der *modus ponens* zur Anwendung, wenn Abendrot beobachtet wird. Dann läßt sich aus den Faktum der Beobachtung und der Regel schließen, daß mit schlechten Wetter zu rechnen ist.

$$\frac{\text{Abendrot} \quad \text{Abendrot} \to \text{Schlechtes\_Wetter}}{\text{Schlechtes\_Wetter}}$$

Da wir mit Syntax, Semantik und Kalkül alle drei Komponenten eines formalen Systems definiert haben, ist der fundamentale Zusammenhang zwischen diesen noch zu zeigen

Satz 3.1.13 *Der Kalkül des natürlichen Schließens ist korrekt und vollständig, d.h.:*

- *aus* $M \vdash_S A$ *folgt* $M \models A$ *(Korrektheit)*
- *aus* $M \models A$ *folgt* $M \vdash_S A$ *(Vollständigkeit)*

Wir werden diesen Satz in diesem Skript nicht beweisen. Die Korrektheit ist leicht zu zeigen. Hierzu kann man über Wahrheitstafeln zunächst zeigen, daß die Axiome Tautologien sind. Schließlich muß gezeigt werden, daß durch *modus ponens* gezogene Schlüsse korrekt sind. Hingegen die Vollständigkeit benötigt mehrere Seiten Beweis, die den Rahmen dieser Einführungsvorlesung sprengen würde.

Jetzt wollen wir aber endlich auch einen Beweis im Kalkül des natürlichen Schließens führen:

Beispiel 3.1.14 *Versuchen wir einmal eine Ableitung der sehr einfachen Formel* $(A \to A)$ *zu finden:*
Ax1 $(A \to ((A \to A) \to A))$
Ax2 $((A \to ((A \to A) \to A)) \to ((A \to (A \to A)) \to (A \to A)))$
MP $((A \to (A \to A)) \to (A \to A))$
Ax1 $((A \to (A \to A))$
MP $(A \to A)$

Wie man sieht, ist es eine mühsehlige Aufgabe, einen Beweis in diesem Kalkül zu finden. Es ist ein kreativer Anteil notwendig, in dem geschickte Instanziierungen der Axiomenschemata gefunden werden müssen. Algorithmisch ließe sich ein solcher Beweis tatsächlich nur über eine Suche realisieren, in der alle möglichen Ableitungsketten aufgezählt werden, um endlich eine Beweiskette zu finden, die zur zu beweisenden Formel führt.

Der Resolutionskalkül

Der Kalkül des natürlichen Schließens im letzten Abschnitt, war für algorithmische Zwecke und als Grundlage eines automatischen Beweissystems auf aussagenlogischen Formeln denkbar ungeeignet. Daher werden wir jetzt einen Kalkül vorstellen, der sich gut automatisieren läßt. Hierzu werden wir die Formeln zunächst in eine bestimmte Normalform umformen.

Klauselnormalform

Definition 3.1.15 (konjunktive Normalform (CNF), Klauselnormalform)
Eine Formel, die eine Konjunktion von Disjunktionen von Literalen ist, ist in konjunktive Normalform oder auch Klauselnormalform.

D.h. die Formel ist von der Form

$$(L_{1,1} \vee \ldots \vee L_{1,n_1}) \wedge \ldots \wedge (L_{m,1} \vee \ldots \vee L_{m,n_m})$$

wobei $L_{i,j}$ *Literale sind.*

Die Klauselnormalform wird oft als Menge von Mengen notiert und auch behandelt. Dies ist gerechtfertigt, da sowohl $\wedge$ als auch $\vee$ assoziativ, kommutativ und idempotent sind, so dass Vertauschungen und ein Weglassen der Klammern erlaubt ist. Wichtig ist die Idempotenz, die z.B. erlaubt, eine Klausel $\{A, B, A, C\}$ als unmittelbar äquivalent zur Klausel $\{A, B, C\}$ zu betrachten. Eine Klausel mit einem Literal bezeichnet man auch als *1-Klausel*. Eine Klausel (in Mengenschreibweise) ohne Literale wird als *leere Klausel* bezeichnet. Diese ist äquivalent zu einem Widerspruch.

Lemma 3.1.16 *Eine Klausel C ist eine Tautologie genau dann wenn es eine Variable A gibt, so dass sowohl $A \in C$ als auch $\neg A \in C$.*

Beweis: Übungsaufgabe.

Lemma 3.1.17 *Eine Klauselmenge M ist unerfüllbar, wenn es eine Variable A gibt, so dass sowohl $\{A\} \in M$ als auch $\{\neg A\} \in M$.*

Beweis: Übungsaufgabe.

Der Resolutionskalkül ist nur auf Klauselmengen definiert. Es wird also die ursprüngliche Struktur der Formeln vollständig zerstört. Tatsächlich ist dieses überhaupt möglich. Jede aussagenlogische Formel läßt sich mit folgendem Algorithmus in eine semantisch äquivalente Formel in Klauselnormalform umformen.

Umformen in Klauselnormalform

- ersetze sukzessive Teilformeln der Gestalt
  - $\neg\neg A$ durch A,
  - $\neg(A \vee B)$ durch $(\neg A \wedge \neg B)$,
  - $\neg(A \wedge B)$ durch $\neg A \vee \neg B$

  bis keine der Umformungen mehr möglich ist.
- dann ersetze Teilformeln der Gestalt
  - $(A \vee (B \wedge C))$ durch $(A \vee B) \wedge (A \vee C)$
  - und $((B \wedge C) \vee A)$ durch $(B \vee A) \wedge (C \vee A)$

  bis keine der Umformungen mehr möglich ist.

Mit diesen Unformungen läßt sich jede Formel in eine semantisch Klauselnormalform transformieren. Die Dijunktionen der Konjuktiven-Normalform werden als Klauseln bezeichnet.

Resolution Der Resolutionskalkül bedient sich eines beweistechnischen Tricks, der auch in der Mathematik gerne angewendet wird. Er realisiert einen Widerspruchsbeweis. Hierbei wird die zu beweisende Aussage zunächst negiert. Wenn die Annahme des Gegenteils schließlich zu einem Widerspruch führt, so war diese negierte Annahme falsch und das Gegenteil ist korrekt.

Der Grundschritt einer Ableitung im Resolutionskalkül ist das bilden einer neuen Klausel aus zwei bestehenden Klauseln. Dieses ist der Rolutionsschritt. Die neue Klausel heißt *Resolvente* und wird der Klauselmenge hinzugefügt.

Definition 3.1.18 (Resolutionsschritt, Resolvente)
Für zwei Klauseln C_1 und C_2, so daß es eine aussagenlogische Variabel A gibt, mit $A \in C_1$ und $A \in C_2$, dann kann in einem Resolutionsschritt aus C_1 und C_2 die Resolvente C_3 gebildet werden mit:
$C_3 = \{x | x \in C_1, x \neq A\} \cup \{x | x \in C_2, x \neq \neg A\}$

Für einen Resolutionsbeweis wird die zu beweisende Formel negiert und in Klauselnormalform umgeformt. Zusätzlich werden alle Formeln des angenommenen Basiswissens in Klauselnormalform ungeformt. Auf der enstehenden Klauselmenge werden so lange Resolventen gebildet, bis die leere Klausel entstanden ist.

Definition 3.1.19 (Resolution)
Sei $M \subset For_S$ und $A \in For_S$. Aus M sei A per Resolution ableitbar im Zeichen $M \vdash_R A$ genau dann wenn:
Aus der Klauselmenge, die durch Umformung in die Klauselnormalform für die Formeln der Menge M und für die Formel $\neg A$ entsteht, nach endlich vielen Resolutionsschritten die leere Klausel resolviert werden kann.

Der aufmerksame Student hat dabei sicher schon bemerkt, daß der Resolutionskalkül bereits im ersten Kapitel dieses Skripts beschrieben war. Die Anleitung, wie mit chinesischen Buchstaben auf einem Übungsblatt zu verfahren sei, stellte bereits den Resolutionskalkül dar.

Es ist nun an der Zeit auch einen Beweis im Resolutionskalkül zu führen. Hierzu erinnern wir uns an die anfänglich verklausuliert gemachten Aussagen über Ernies Gedichte.

Beispiel 3.1.20 *In den fünf Aussagen über Gedichte tauchen sechs verschiedene Arten von Gedichten auf. Für jede dieser Gedichtarten sei eine aussagenlogische Variabel vorgesehen, so daß die folgende Signatur erhalten wird:*
$S = \{Modern, Beliebt, Interessant, Schwülstig, Ernies, Seifenblasen\}$

Mit dieser Signatur, lassen sich recht schnell die fünf Aussagen in aussagenlogische Formeln ausdrücken:

- $Interessant \rightarrow \neg\neg\ Beliebt$
- $Modern \rightarrow Schwülstig$
- $Ernies \rightarrow Seifenblasen$
- $Schwülstig \rightarrow \neg\ Beliebt$
- $\neg\ Modern \rightarrow \neg\ Seifenblasen$

Diese fünf Aussagen sind die Formeln der Menge M. Wir wollen jetzt beweisen, daß Ernies Gedichte nicht interessant sind, also den Satz:
$Ernies \rightarrow \neg\ Interessant$

Hierzu negieren wir den zu beweisenden Satz und erhalten:
$\neg(Ernies \rightarrow \neg\ Interessant)$

Nun sind zunächst die sechs Aussagen in Klauselnormalform umzuformen. Wir erhalten die folgenden Formeln:

- $\neg\ Interessant \vee Beliebt$
- $\neg\ Modern \vee Schwülstig$
- $\neg\ Ernies \vee Seifenblasen$
- $\neg\ Schwülstig \vee \neg\ Beliebt$

- *Modern* $\vee\neg$ *Seifenblasen*
- *Ernies wedge Interessant*

Wir erhalten die sieben folgenden Klauseln

$$\{\neg Interessant, Beliebt\} \tag{3.1}$$

$$\{\neg Modern, Schwülstig\} \tag{3.2}$$

$$\{\neg Ernies, Seifenblasen\} \tag{3.3}$$

$$\{\neg Schwülstig, \neg Beliebt\} \tag{3.4}$$

$$\{Modern, \neg Seifenblasen\} \tag{3.5}$$

$$\{Ernies\} \tag{3.6}$$

$$\{Interessant\} \tag{3.7}$$

Es lassen sich die folgende neue Klausen resolvieren:

$$\{Beliebt\}, \textit{ aus 3.1 mit 3.7} \tag{3.8}$$

$$\{\neg Schwülstig\}, \textit{ aus 3.4 mit 3.8} \tag{3.9}$$

$$\{\neg Modern\}, \textit{ aus 3.2 mit 3.9} \tag{3.10}$$

$$\{\neg Seifenblasen\}, \textit{ aus 3.5 mit 3.10} \tag{3.11}$$

$$\{\neg Ernies\}, \textit{ aus 3.3 mit 3.11} \tag{3.12}$$

$$\{\}, \textit{ aus 3.3 mit 3.6} \tag{3.13}$$

Damit konnte die leere Klausel durch Resolution erzeugt werden. Die Annahme, daß nicht folgt, Ernies Gedichte seien interessant führen zum Widerspruch und muß fallen gelassen werden. Wir haben bewiesen, daß aus den fünf Aussgane folgt, daß Ernies Gedichte sind uninteressant.

Aufgabe 4 Beweise mittels des Resolutionskalküls, daß die drei Axiome des Kalküls des natürlichen Schließens Tautologien sind.

Wir werden in diesem Skript keine Korrektheitsaussagen und Vollständigkeitsaussagen über den Resolutionskalkül beweisen. Interessierte Studenten seinen an das Skript von Herrn Schmidt-Schauß verwiesen[SS06].

Dem aufmerksamen Leser wird aufgefallen, daß der hier beschriebene Resolutionsalgorithmus große Freiheitsgrade aufweist. Es dürfen beliebige Klauseln gewählt werden, um eine Resolvente zu bilden. Damit sind wir wieder mitten in einem Suchproblem gelandet. Gesucht wird die leere Klausel. Die Knotenmarkierungen des Suchbaums sind Klauselmengen. Ein Suchschritt ist, eine Resolvente zu bilden.

3.1.4 Implementierung

Wie auch in den vorangegangenen Kapiteln soll in diesem Kapitel der vorgestellte Algorithmus auch in der Programmiersprache Scala wieder Implementiert werden. Wir beginnen damit, Klassen für die fünf Formelarten: Atom, Negation, Konjunktion, Disjunktion und Implikation zu definieren.

Formel.scala

```
1  package name.panitz.ki
2  case class Atom(name:String) extends Formel{
3    override def toString()=name}
4  case class Not(e1:Formel) extends Formel{
5    override def toString()="¬"+e1.toString}
6  case class Impl(e1:Formel,e2:Formel) extends Formel{
7    override def toString()="("+e1.toString+" -> "+e2.toString+")"}
8  case class And(e1:Formel,e2:Formel) extends Formel{
9    override def toString()="("+e1.toString+" /\\ "+e2.toString+")"}
10 case class Or(e1:Formel,e2:Formel) extends Formel{
11   override def toString()="("+e1.toString+" \\/ "+e2.toString+")"}
```

Diese Klassen habe alle als gemeinsame Oberklasse eine Klasse `Formel`.

Formel.scala

```
12 abstract class Formel{
```

Eine Formel soll in Klauselnormalform umgewandelt werden. Hierzu werden zunächst innerhalb einer Formel die Implikationen durch Disjunktionen ausgedrückt:

Formel.scala

```
13   def eliminateImpl():Formel={
14    this match {
15     case Atom(_)  => this
16     case Not(x)   => Not(x.eliminateImpl)
17     case And(x,y) => And(x.eliminateImpl,y.eliminateImpl)
18     case Or(x,y)  => Or(x.eliminateImpl,y.eliminateImpl)
19     case Impl(x,y)=> Or(Not(x.eliminateImpl),y.eliminateImpl)
20    }
21   }
```

Dann sind Negtionssymbole bis direkt vor die Atome zu schieben:

Formel.scala

```
22   def negationToInner():Formel={
23    this match {
24     case Atom(_)       => this
25     case Not(Atom(x))  => this
26     case Not(Not(x))   => x.negationToInner
27     case Not(And(x,y)) => Or(Not(x),Not(y)).negationToInner
28     case Not(Or(x,y))  => And(Not(x),Not(y)).negationToInner
29     case And(x,y)      => And(x.negationToInner,y.negationToInner)
```

```
30      case Or(x,y)       => Or(x.negationToInner,y.negationToInner)
31     }
32    }
```

Und schließlich ist die Konjunktion von Disjunktion zu erzeugen, indem das Odersymbol ausmultipliziert wird:

Formel.scala

```
33    def makeConjunctions():Formel={
34     this match {
35      case Atom(_)       => this
36      case Not(x)        => this
37      case Or(And(x,y),z)=> And(Or(x,z),Or(y,z)).makeConjunctions
38      case Or(x,And(y,z))=> And(Or(x,y),Or(x,z)).makeConjunctions
39      case Or(x,y)
40        => val x1 =x.makeConjunctions
41           val y1 =y.makeConjunctions
42           x1 match {
43             case And(_,_) => Or(x1,y1).makeConjunctions
44             case _
45               => y1 match {
46                   case And(_,_) => Or(x1,y1).makeConjunctions
47                   case _        => Or(x1,y1)
48                 }
49           }
50      case And(x,y)      => And(x.makeConjunctions,y.makeConjunctions)
51     }
52    }
```

Die nacheinander Ausführung dieser drei Umformungen erzeugt die Klauselnormalform:

Formel.scala

```
53    def toCNF():Formel=eliminateImpl.negationToInner.makeConjunctions
```

Eine Formel in Klauselnormalform kann als Menge von Mengen von Literalen interpretiert werden. Diese Umwandlung in Mengen wird durch die folgenden Methoden erzielt:

Formel.scala

```
54    import scala.collection.mutable.HashSet
55    import scala.collection.mutable.Set
56    def conjunctionSet():Set[Formel]={
57      val result=new HashSet[Formel]()
58      this match {
59        case And(e1,e2) => result++= e1.conjunctionSet;
60                           result++= e2.conjunctionSet;
61        case e => result+= e
62      }
63      result
64    }
```

```
65
66    type Klausel=Set[Formel]
67    def disjunctionSet():Klausel={
68      val result=new HashSet[Formel]()
69      this match {
70        case Or(e1,e2) => result++= e1.disjunctionSet;
71                          result++= e2.disjunctionSet;
72        case e => result+= e
73      }
74      result
75    }
76
77    def cnf():List[Klausel]=
78      (for (val f<- toCNF.conjunctionSet.elements)
79         yield f.disjunctionSet).toList
80
81    def negated()= this match { case Not(x) => x
82                                case y      => Not(y)}
83  }
```

Damit haben wir den ersten Schritt für einen Resolutionsbeweis implementiert, die Umwandlung in eine Klauselmenge. Schließlich soll auf einer Klauselmenge systematisch resolviert werden:

Resolution.scala

```
1  package name.panitz.ki
2  object Resolution{
3    import scala.collection.mutable.Set
4    import scala.collection.mutable.HashSet
5    type Klausel=Set[Formel]
```

Zwei Klauseln können resolviert werden, wenn ein Atom existiert, das in der einer der Klausel negiert in der anderen nichtnegiert existiert:

Resolution.scala

```
6    def resolvable(xs:Klausel,ys:Klausel)=
7      xs.exists((x)=>ys.exists((y)=>y.negated==x))
```

Im eigentlichen Resolutionsschritt werden zwei Klauseln vereinigt und nur die Literale über die die Resolutions läuft weggelseen.

Resolution.scala

```
8    def resolve(xs:Klausel,ys:Klausel)={
9      val result=new HashSet[Formel]()
10     for (val x<-xs.elements;!ys.exists((y)=>y==x.negated)) result+=x
11     for (val y<-ys.elements;!xs.exists((x)=>y==x.negated)) result+=y
12     result
13   }
```

Für die Resolution wird systematisch auf einer Klauselmenge resolviert bis, entweder keine weiteren Resolventen gebildet werden können, oder aber die leere Klausel gefunden wurde:

Resolution.scala

```
14   def resolution(xsP:List[Klausel])={
15     var xs=xsP
16     var zs=xsP
17     while (!zs.isEmpty && !zs.exists((x)=>x.size==0)){
18       val ks=
19        for (val x<-xs;val y<-zs;resolvable(x,y)) yield resolve(x,y)
20
21       zs=ks.toList
22       xs= xs ::: zs
23     }
24     xs
25   }
```

Für einen Beweis einer Formel wird diese negiert, in Klauselform gebracht und auf den Klauseln resolviert:

Resolution.scala

```
26   def proof(a:Formel):List[Set[Formel]]=proof(List(),a)
27   def proof(ms:List[Formel],a:Formel):List[Set[Formel]]={
28     var clauses=List[Set[Formel]]()
29     for (val m<-ms) clauses = m.cnf ::: clauses
30     clauses = Not(a).cnf ::: clauses
31     resolution(clauses)
32   }
33 }
```

Wir testen unseren aussagenlogischen Resolutionsbeweiser an den drei Axiomen des Kalküls des natürlichen Schließens:

TestFormel.scala

```
1  package name.panitz.ki
2  object TestFormel extends Application{
3    val ax1:Formel = Impl(Atom("A"),Impl(Atom("B"),Atom("A")))
4    Console println ax1
5    Console println ax1.toCNF
6    Console println Resolution.proof(ax1)
7
8    val ax2=Impl(Impl(Atom("A"),Impl(Atom("B"),Atom("C")))
9                ,Impl(Impl(Atom("A"),Atom("B"))
10                    ,Impl(Atom("A"),Atom("C"))))
11   Console println ax2
12   Console println ax2.toCNF
13   Console println Resolution.proof(ax2)
14
15
16   val ax3=Impl(Impl(Atom("A"),Atom("B"))
17               ,Impl(Not(Atom("B")),Not(Atom("A"))))
18   Console println ax3
19   Console println ax3.toCNF
```

```
20    Console println Resolution.proof(ax3)
21  }
```

```
sep@pc305-3:~/fh/ki/student/classes> scala name/panitz/ki/TestFormel
(A -> (B -> A))
(¬A \/ (¬B \/ A))
List({¬A},{A},{B},{},{})

((A -> (B -> C)) -> ((A -> B) -> (A -> C)))
(((A \/ (A \/ (¬A \/ C))) /\ (A \/ (¬B \/ (¬A \/ C)))) /\ (((B \/ (A \/ (¬A \/C))) /\ (B \/ (¬B \/
(¬A \/ C)))) /\ ((¬C \/ (A \/ (¬A \/ C))) /\ (¬C \/ (¬B \/ (¬A \/ C))))))
List({¬A, B},{¬C},{C, ¬A, ¬B},{A},{C, ¬A},{B},{¬A, ¬B},{¬A, C},{¬A, ¬B},{C,¬B},{B},{C, ¬B},{¬A},
{¬A},{C, ¬A},{C, ¬A},{¬A},{¬A},{¬B},{¬B},{¬A, C},{¬A,C},{C},{¬B},{C},{¬B},{¬A},{¬A},{C},{C},{¬A},
{¬A},{¬A},{¬A},{C},{C},{¬A},{¬A},{C},{C},{C},{C},{¬A},{¬A},{¬A},{¬A},{¬A},{¬A},{¬A},{¬A},{},{},
{},{},{},{},{},{},{},{},{},{C},{C},{},{},{C},{C},{},{},{},{},{},{},{},{},{},{},{},{},{},{},{},{})

((A -> B) -> (¬B -> ¬A))
((A \/ (B \/ ¬A)) /\ (¬B \/ (B \/ ¬A)))
List({¬A, B},{A},{¬B},{B},{¬A},{B},{¬A},{},{},{},{})
sep@pc305-3:~/fh/ki/student/classes>
```

Aufgabe 5 Formalisieren Sie die folgenden Aussagen in der Aussagenlogik und beweisen Sie mit dem Resolutionskalkül, daß Kleinkinder nicht mit Krokodilen umgehen können.

1. Kleinkinder sind nicht logisch.
2. Niemand wird nicht geachtet, der mit Krokodilen umgehen kann.
3. Unlogische Personen werden nicht geachtet.

Aufgabe 6 Formalisieren Sie die folgenden Aussagen in der Aussagenlogik und beweisen Sie mit dem Resolutionskalkül, daß alle Pfandleiher ehrlich sind.

1. Leuten, die Versprechen nicht einhalten, kann man nicht trauen.
2. Weintrinker sind sehr kommunikative Menschen.
3. Jemand, der seine Versprecher einhält, ist ehrlich.
4. Kein Weinabstinenzler ist Pfandleiher.
5. Kommunikative Menschen kann man prinzipiell trauen.

3.1.5 Bert Bresgen: Axiome der Liebe

Bevor wir uns im nächsten Abschnitt der Prädikatenlogik widmen, soll der Autor Bert Bresgen noch einmal zu Wort kommen, und ein paar Gedanken zur Logik aus Sicht eines Nicht-Informatikers verlieren:

Bert: Manchmal am Ende eines langen Tages beginnen Mathematiker zu träumen. Sie sind es leid, immer irgendjemand irgendetwas beweisen zu müssen. Sie träumen von Sätzen und Formeln, die unmittelbar wahr sind, ohne dass man sie beweisen muß. Solche Sätze nennt man Axiome.

Bereits Aristoteles träumte davon und er fand drei dieser Sätze. Der erste Satz lautet:

- $A = A$, d.h. etwas, das mit sich selbst nicht identisch ist, kann nicht gedacht werden.

Was bedeutet das in Hinsicht auf die Liebe?

Ich liebe A. Aber wer ist A? A ist ... gleich A. Deshalb liebe ich A. Allerdings macht es sich A. ein bisschen leicht damit, einfach A zu sein. Ich liebe A., weil A A ist, aber natürlich liebe ich nicht alles an A. Vielleicht würde ich sogar B. noch mehr lieben als A., wenn ... B. nur so wäre wie A.

Ein Lied der Lassie Singers stellt dieses Problem dar.*(Lied wird eingespielt))*

Die beiden anderen Axiome des Aristoteles, die nicht bewiesen werden müssen, lauten:

- A kann nicht zugleich nicht A sein, und
- Alles, was ist, ist entweder A oder nicht A. Dies ist der Satz vom ausgeschlossenen Dritten.

Der Satz vom ausgeschlossenen Dritten bedeutet im Zustand akuter Verliebtheit z.B.: Das ist Andrea. Ich weiss, daß das Andrea ist. Alles was ist, ist entweder Andrea oder es ist Nicht–ANDREA. Nicht-.Andrea ist zum Beispiel Matthias. Oder Birgit. Matthias oder Birgit sind unwichtig, denn sie sind nicht: Andrea. Andrea ist die Welt für mich. Komischerweise ist trotzdem alles, was ist, Nicht–Andrea, mit Ausnahme Andreas. Meine Arbeit, der Schmutz unter meinen Fingernägeln oder die sixtinische Kapelle sind nicht Andrea. Nicht–Andrea ist die neue Strassenbahn der Linie 17, die bis zum Rebstockbad fährt. Es ist sinnlos, mit Birgit in der Linie 17 zum Rebstockbad zu fahren, wo sie mir ihren neuen Badeanzug zeigt, denn Birgit ist nicht Andrea. Ich sage zu Birgit "Was für ein schöner Tag, Birgit. Was für ein schöner,neuer Badeanzug, Birgit. Er erinnert mich an Andreas Badeanzug. Ich weiss nicht, ob ich das schon gesagt habe, Birgit, aber: Andrea schwimmt die 50 Meter Bahn ohne Badeanzug in 7 Stunden. Und während ich ertrinke, schmiert sie diese fabelhaften Clubsandwiches. Möchtest Du Thunfisch oder Huhn, Birgit?"

Draußen dämmert frühes Licht.

Die drei nicht beweisnotwendigen Sätze des Aristoteles gelten bis zum heutigen Tag.

3.2 Prädikatenlogik

3.2.1 Einführende Beispiele

Der vorangegangene Text von bert Bresgen läßt schon eine gewisse Unzufriedenheit mit der Atomarität von Aussagen durchschimmern. Die kleinste Einheit in der Aussagenlogik sind aussagenlogische Variablen, die entweder als wahre oder als falsche Aussagen angenommen werden. Bei der Modellierung eines rationalen Agenten werden wir häufig mit Problemfeldern und Welten konfrontiert sein, in denen es mehrere verschiedene Individuen gibt, die bestimmten Menge zugehörig sind und in bestimmten Beziehungen zueinander stehen. Derartige Zusammenhänge und Aussagen über verschiedene Individuen lassen sich nicht mit der Aussagenlogik beschreiben. Hierzu bedarf es einer mächtigeren Sprache, die mit der Prädikatenlogik zur Verfügung steht. Die Prädikatenlogik ist in ihrer Modellierungsstärke recht mächtig, so daß sie für viele recht unterschiedliche Problemlösungsansätze der KI die Grundlage bildet. Aber auch über die Anwendungsfälle der KI hinaus, gehört die Prädikatenlogik zu einem formalen System, dem sich ein Informatiker nicht verschließen sollte. Ein sehr schönes Buch, das Logik Informatiker-gerecht einführt ist [Sch89].

In diesem Kapitel werden teilweise Abschnitte und Definitionen aus dem Skript von Herrn Schmidt-Schauß[SS06] fast wörtlich übernommen.

Bevor wir in gleicher Weise, wie im Kapitel über die Aussagenlogik, zunächst eine Syntax, anschließend Semantik und den Resolutionskalkül für die Prädikatenlogik definieren, wollen wir uns der Sprache mit ein paar informellen Beispielen annähern, die ein Gefühl für die Ausdrucksstärke geben können.

Bleiben wir hierzu zunächst bei dem kleinen Szenario einer Bauklötzchenwelt, dem wir uns auch schon im Kapitel über CSP gewidmet hatten. In Abbildung 3.4 findet sich eine weitere kleine Skizze von Bauklötzen.

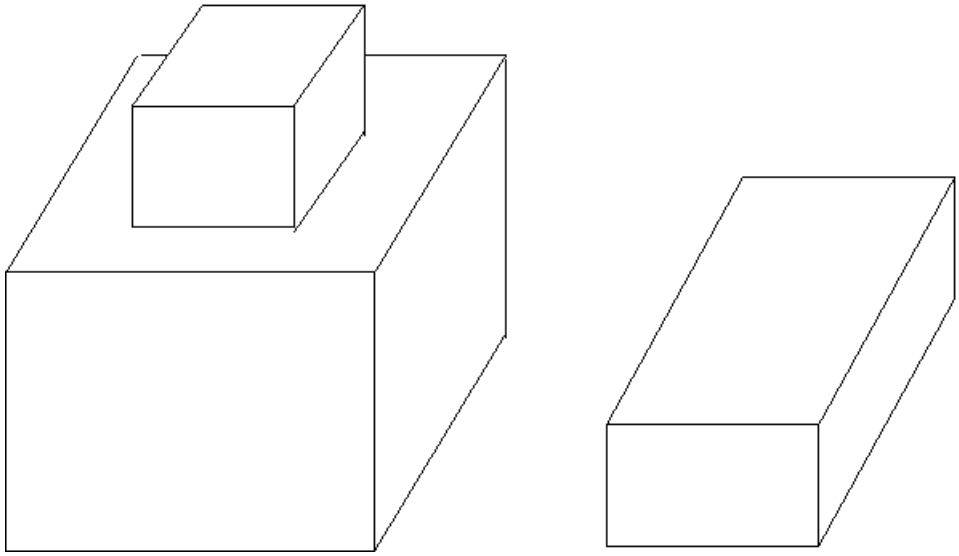

Abbildung 3.4: In Prädikatenlogik zu modellierendes Bauklötzchen Szenario.

Es sind hier drei Bauklötze zu erkennen, zwei von ihnen stehen auf dem Boden, ein dritter steht auf einem weiterem Bauklotz. Wenn wir die Bauklötze mit den Namen A, B und C bezeichnen, so lassen sich folgende Fakten über die Anordnung der Bauklötze als Prädikate aufschreiben:

$$Klotz(A)$$

$$Klotz(B)$$

$$Klotz(C)$$

$$Steht Auf(B, A)$$

$$Auf Boden(A)$$

$$Auf Boden(C)$$

Drei unterschiedliche Prädikate sind in diesen Fakten benutzt. Zwei einstellige Prädikate, die somit eine Mengenzugehörigkeit ausdrücken und eine zweistelliges Prädikat, das eine Relation ausdrückt.

Stellen wir uns vor, wir wollen einen rationalen Agenten modellieren, der mit einem Greifarm ausgestattet ist. Er kann Bauklötze greifen, sofern kein weiterer Bauklotz auf ihnen steht. Dieser Zusammenhang läßt sich durch eine Regel ausdrücken.

$$\forall x : (\neg \exists y : \ StehtAuf(y, x)) \rightarrow \ GreifBar(x)$$

Damit ist ausgedrückt, daß alle Individuen (in unserem Fall Bauklötze) ergriffen werden können, wenn kein anderes Individuum existiert, das auf dem fraglichen Individuum steht.

Man hätte das durchaus auch ander ausdrücken können, nämlich durch:

$$\forall x : (\forall y : \ \neg StehtAuf(y, x)) \rightarrow \ GreifBar(x)$$

Nämlich, das greifbare Objekte sind die, für die gilt, daß für alle anderen Objekte gezeigt werden kann, daß sie nicht auf dem fraglichen stehen.

Man bekommt schon ein Gefühl dafür, wie sich die Welten, in denen potentiell rationale Agenten sich bewegen sollen, mit der Prädikatenlogik formalisiert werden können.

Neu gegenüber der Aussagenlogik sind nicht nur die Prädikate, sondern auch die Konstanten, die bestimmte Individuen bezeichnen sollen (hier A, B und C) sowie Variablen, die durch die Quantoren $\forall$ und $\exists$ eingeführt werden können.

Was ist jetzt gegenüber der Aussagenlogik zu beachten, wenn versucht wird eine Schlußregel in der Prädikatenlogik anzuwenden. Hierzu betrachte man eine alt bekannte Aussage über die allgemeine Sterblichkeit und ein spezielles Fakt über ein spezielles Individuen (auch wenn das von manchen angezweifelt wird):

$$\forall x : \ Mensch(x) \rightarrow Sterblich(x)$$

$$Mensch(Elvis)$$

Hier wird eine allgemeine Implikationregel für alle Individuen, die der Menge Mensch zugehörig sind aufgestellt. Zusätzlich wird von einem bestimten Element die Zugehörigkeit zu der Menge Mensch als Fakt gegeben. Um jetzt aus diesen beiden Formeln das Fakt abzuleiten, daß selbst Elvis zur Menge der sterblichen Dinge gehört, ist die allgemeine Regel, die für alle Individuen x gilt, auf ein bestimmtes Individuum anzuwenden. Mann könnte auch sagen, daß

$$\forall x : \ Mensch(x) \rightarrow Sterblich(x)$$

ein Schema ist, in das man für die Variabel x beliebige Individuen einsetzen kann, z.B.:

$$Mensch(Elvis) \rightarrow Sterblich(Elvis)$$

oder auch für andere Individuen:

$$Mensch(Jesus) \rightarrow Sterblich(Jesus)$$

Hat man eine solche Einsetzung für eine Variabel gefunden, läßt sich mittels des Modus Ponens wie in der Aussagenlogik das neue Fakt herleiten. Der Schluß läßt sich analog zur Aussagenlogik schematisch aufschreiben:

$$\frac{\text{Mensch(Elvis)} \quad \text{Mensch(Elvis)} \rightarrow \text{Sterblich(Elvis)}}{\text{Sterblich(Elvis)}}$$

Wie man sieht, wird es notwendig werden, um spezielle Schlüsse zu ziehen, eventuell Variablen durch konkrete Ausdrücke zu ersetzen. Eine solche Ersetzung heißt Substitution. Wie wir sehen werden, spielen Substitutionen eine entscheidene Rolle in Kalkülen der Prädikatenlogik.

Bisher haben wir Variablen, Konstanten und Prädikate als Bestandteile der Prädikatenlogik gesehen. Es gibt noch ein zusätzliches Konzept, die Funktionen. Eine Funktion soll für ein oder mehere Elemente, nämlich für ihre Argumente, ein neues Argument liefern. Wir können uns z.B. für das obige Beispiel zusätzlich eine Funktion `vater` vorstellen, die für jedes Element ein Element zurückgibt, das den Vater dieses Elements darstellen soll. Damit ließe sich folgender Sachverhalt formalisieren:

$$\forall x : Mensch(x) \rightarrow Mensch(vater(x))$$

Für jeden Menschen gilt, daß sein vater auch ein Mensch ist.

So ließe sich also aus dem Fakt `Mensch(Adam)` mit Modus Ponens ableiten: `Mensch(vater(Adam))` und in einem weitern Schritt: `Sterblich(vater(Adam))`.

Zum Abschluß der informellen Einführung der Prädikatenlogik seien noch zwei Aussagen aus bekannten Schlagern formalisiert:

Z. B der alten Dean Martin Schlager *Everybody Loves Somebody Sometime* könnte man wie folgt formalisieren:

$$\forall x : \exists y : \exists t : LovesAtTime(x, y, t)$$

Damit hätten wir auch ein erstes Beispiel eines Prädikats, daß mehr als zwei Argumente hat, nämlich eines dreistelligen Prädikats.

Oder es läßt sich die bekannte Tatsache formalisieren, daß fast alles ein Ende hat, nur eben die Wurst nicht (wobei wir außer acht lassen, daß die zwei Enden hat):

$$\forall x : HatEnde(x) \vee Wurst(x)$$

3.2.2 Syntax

Nachdem schon eine Reihe von Beispielen für prädikatenlogische Formeln gezeigt wurden, wird es notwendig, die Syntax endlich auch formal zu beschreiben. Ebenso wie in der Aussagenlogik werden wir die Syntax nicht über eine kontextfreie Grammatik definieren, sondern ein strukturelle induktive Definition vorziehen.

Die Definition geht in zwei Stufen. Zunächst werden Terme definiert. Terme sind die Ausdrücke, über die Prädikate Aussagen machen. Dann werden die Formeln definiert.

Wir definieren die Prädikatenlogik erster Stufe, kurz PL1.[1]

[1] Prädikatenlogik höherer Ordnung begegnet man selten in der KI. Wir werden uns nicht damit beschäftigen und wann immer wir von der Prädikatenlogik sprechen ist sie erster Stufe gemeint. In höheren Ordnungen kann man auch über Prädikatensymbole quantifizieren.

Definition 3.2.1 (Syntax von PL1)
Sei gegeben eine Signatur $\Sigma = (\mathcal{F}, \mathcal{P})$ bestehend aus einer Menge $\mathcal{F} = \{f, g, h, \ldots\}$ von Funktionssymbolen und einer Menge $\mathcal{P} = \{A, B, C, \ldots\}$ von Prädikatensymbolen. Sei weiterhin gegeben eine Menge von Variablensymbolen: $\mathcal{V} = \{x, y, z, \ldots\}$. Diese drei Mengen seien disjunkt. Schließlich sei noch eine totale Funktion: $arity : \mathcal{F} \cup \mathcal{P} \rightarrow \mathbb{N}$ gegeben, die für jedes Funktionssymbol und jedes Prädikatensymbol die Stelligkeit angibt.

Dann sei die Menge der Terme $\mathcal{T}_{(\Sigma,V)}$ die kleineste Menge mit:

- $\mathcal{V} \subset \mathcal{T}_{(\Sigma,V)}$.
- *wenn* $f \in \mathcal{F}, arity(f) = n$ *und* $t_i \in \mathcal{T}_{(\Sigma,V)}, 1 \leq i \leq n$ *dann ist auch* $f(t_1, \ldots, t_n) \in \mathcal{T}_{(\Sigma,V)}$.

Die Menge der Formeln $For_{(\Sigma,V)}$ sei die kleineste Menge mit:

- *wenn* $P \in \mathcal{P}, arity(p) = n$ *und* $t_i \in \mathcal{T}_{(\Sigma,V)}, 1 \leq i \leq n$ *dann ist auch* $P(t_1, \ldots, t_n) \in For_{(\Sigma,V)}$.
- *wenn* $F \in For_{(\Sigma,V)}$ *dann auch* $\neg F \in For_{(\Sigma,V)}$
- *wenn* $F, G \in For_{(\Sigma,V)}$ *dann auch* $(F \vee G) \in For_{(\Sigma,V)}$
- *wenn* $F, G \in For_{(\Sigma,V)}$ *dann auch* $(F \wedge G) \in For_{(\Sigma,V)}$
- *wenn* $F, G \in For_{(\Sigma,V)}$ *dann auch* $(F \rightarrow G) \in For_{(\Sigma,V)}$
- *wenn* $F \in For_{(\Sigma,V)}$ *dann auch* $\forall x : (F) \in For_{(\Sigma,V)}$
- *wenn* $F \in For_{(\Sigma,V)}$ *dann auch* $\exists x : (F) \in For_{(\Sigma,V)}$

Analog zu den Begriffen in der Aussagenlogik, sei eine Formel, die nur aus einem Prädikat besteht als ein Atom bezeichnet, und eine Formel, die ein Atom oder ein negiertes Atom ist, als Literal.

Unter Umständen werden Klammerungen weggelassen, wenn über die üblichen Operatorpräzedenzen die Klammerung sich aus dem Kontext ergibt.

Terme, in denen keine Variabeln vorkommen werden als Grundterme bezeichnet. Atome, die nur Grundterme enthalten werden als Grundatome bezeichnet.

Die Syntax erlaubt es Formeln zu schreiben, in denen Variablen auftreten, die durch keinen Quantoren gebunden sind. Solche Variablen werden als freie Variablen bezeichnet. Quantoren binden Variablen ähnlich wie Parameter an eine Funktion in einer Programmiersprache gebunden werden.

Nullstellige Funktionen sind nach der Syntax erlaubt. Sie stellen die Konstantensymbole dar. Streng genommen hätten wir das Konstantensymbol `Elvis` im einführenden Beispiel mit leeren Parameterklammern schreiben müssen also `Elvis()`. Ist es im Kontext klar, daß von einer Konstanten und nicht von einer Variablen die Rede ist, so lassen wir das leere Klammerpaar mitunter fort.

Die Prädikatenlogik umfasst vollständig die Aussagenlogik. Nullstellige Prädikate entsprechen dabei den aussagenlogischen Prädikaten.

Implementierung

Counter.scala

```
1  package name.panitz.ki.pl1
2
3  class Counter(var n_ :Int){def n():Int={n_ = n_ +1;n_}}
```

ToLaTeX.scala

```
1  package name.panitz.ki.pl1
2  trait ToLaTeX{def toLaTeX(): String}
```

Term.scala

```
1  package name.panitz.ki.pl1
2  import scala.collection.mutable.HashMap
3  import scala.collection.mutable.Map
4  case class Var(val name:String ) extends Term{
5    def toLaTeX()=name
6    def substitute(map :Map[Var,Term]):Term=
7      map get this match {
8        case None => this
9        case Some(t)=>t
10     }
11 }
12
13 case class Fun(val name:String,val args :List[Term] ) extends Term{
14   def toLaTeX()={
15     var result="\\mbox{\\rm "+name+"}("
16     var first=true
17     for (val arg:Term <- args){
18       if (first)first=false else result=result+","
19       result=result+arg.toLaTeX()
20     }
21     result+")"
22   }
23   def substitute(map :Map[Var,Term]):Term
24      =Fun(name,for (val a<-args) yield a substitute map)
25 }
26
27 import scala.collection.mutable.HashSet
28 abstract class Term extends ToLaTeX{
29   def freeVars():HashSet[Var]=freeVars(new HashSet())
30   def freeVars(bound:HashSet[Var]):HashSet[Var]
31     =freeVars(bound,new HashSet())
32   def freeVars(bound:HashSet[Var],result:HashSet[Var]):HashSet[Var]
33   ={
34     this match {
35       case Var(_) => if (!bound.contains(this.asInstanceOf[Var]))
36                        result += this.asInstanceOf[Var]
37                      else ()
38       case Fun(_,args)
```

```
39          => for (val arg:Term <- args) arg.freeVars(bound,result)
40      }
41      result
42    }
43    def substitute(map :Map[Var,Term]):Term
44  }
```

PL1.scala

```
1   package name.panitz.ki.pl1
2   case class Atom(name:String, args :List[Term] ) extends PL1{
3     def toLaTeX()={
4       var result="\\mbox{\\rm "+name+"}("
5       var first=true
6       for (val arg:Term <- args){
7         if (first)first=false else result=result+","
8         result=result+arg.toLaTeX()
9       }
10      result+")"
11    }
12  }
13  case class Not(e1:PL1) extends PL1{
14    override def toLaTeX()="\\neg "+e1.toLaTeX}
15  case class Impl(e1:PL1,e2:PL1) extends PL1{
16    override def toLaTeX()
17      ="("+e1.toLaTeX+" \\rightarrow "+e2.toLaTeX+")"
18  }
19  case class And(e1:PL1,e2:PL1) extends PL1{
20    override def toLaTeX()="("+e1.toLaTeX+" \\wedge "+e2.toLaTeX+")"
21  }
22  case class Or(e1:PL1,e2:PL1) extends PL1{
23    override def toLaTeX()="("+e1.toLaTeX+" \\vee "+e2.toLaTeX+")"
24  }
25  case class ForAll( x:Var, e:PL1) extends PL1{
26    override def toLaTeX()="(\\forall "+x.toLaTeX+" . "+e.toLaTeX+")"
27  }
28  case class Exists( x:Var, e:PL1) extends PL1{
29    override def toLaTeX()="(\\exists "+x.toLaTeX+" . "+e.toLaTeX+")"
30  }
```

3.2.3 Semantik

Nun soll in gleicher Weise wie auch schon für die Aussagenlogik eine Semantik definiert werden. Während dieses für die Aussagenlogik recht einfach war, und in Abbildungen der aussagenlogischen Variablen auf die Werte *wahr* oder *falsch* mündete, wird die Sache jetzt ein wenig vertrakter. Zwar soll im Endeffekt eine prädikatenlogische Formel für eine bestimmte Interpretation wieder zu *wahr* oder zu *falsch* ausgewertet werden, doch Terme und Prädikate sollen ja komplexe Beziehungen zwischen Elementen verschiedener Mengen darstellen. Um also Formeln

zu interpretieren, brauchen wir mindestens eine Menge von Elementen, sozusagen die Individuen, für die wir die Formeln aufgestellt haben. Um eine Formel zu interpretieren sind wir in der Wahl der Menge, auf der Funktionen und Prädikate interpretieren weitgehendst frei, so lange diese Menge abzählbar und nicht leer ist. Diese Menge wird als Trägermenge bezeichnet. Wir können ansonsten zur Interpretation einer Formel eine beliebige solche Trägermenge wählen, sei es die Menge aller Studenten der FH Wiesbaden, die Menge aller Menschen, die wir einmal geliebt haben, die Menge der Bauklötze in einem bestimmten Kinderzimmer, die Menge der natürlichen Zahlen oder die Menge aller Sandkörner in der Sahara.

Auf dieser Trägermenge werden dann jedes Funktionssymbol als eine entsprechende totale Funktion interpretiert, und jedes Prädikatensymbol als eine Relation auf der Trägermenge, wobei einstellige Prädikate Teilmengen der Trägermenge darstellen.

Eine Interpretaion wertet somit einen Term zu einem Element der Trägermenge aus, ein Prädikat zu einem der beiden Werte `W` oder `F` und auch jede Formel zu einen der Werte `W` oder `F`.

Definition 3.2.2 (Interpretation von PL1)
Eine Interpretation prädikatenlogischer Formeln besteht aus:

- *einer nichtleeren abzählbaren Trägermenge $\mathcal{D}$.*
- *einer Abbildung für jedes Funktionssymbol $f \in \mathcal{F}$ auf eine totale Funktion f_i in $\mathcal{D}$ mit entsprechender Stelligkeit.*
- *einer Abbildung für jedes Prädikatensymbol $P \in \mathcal{P}$ auf eine Relation P_I in $\mathcal{D}$ mit entsprechender Stelligkeit.*
- *einer Variablenbelegung* Iv*, die jede Variabel auf ein Element der Trägermenge abbildet.*

Eine Interpretation wird erträglich erweitert auf Terme. Jeder Term wird somit von einer Interpretation genau als ein Element der Trägermenge interpretiert.

Für eine gegebene Interpretation I definieren wir eine abgeänderte Interpretation $I[x \rightsquigarrow a]$ mit $I[x \rightsquigarrow a](x) = a$ und $I[x \rightsquigarrow a](y) = I(x)$ wenn $x \neq y$.

Eine Interpretation sieht also auf einer Trägermenge für jedes Funktionssymbol in der Signatur eine totale Funktion vor und für jede Prädikatensymbol eine Relation. Eine Interpretation wertet Terme zu einem Element der Trägermenge aus. Formeln werden nun wie auch schon in der Aussagenlogik durch eine Interpretation auf einen Wert aus der Menge $\{W, F\}$ abgebildet:

Definition 3.2.3 (Auswertung von PL1-Formeln)
Sei I eine Interpretation.
Basisfälle:

Fall: $H = P(t_1, \ldots t_n)$	*falls* $(I(t_1), \ldots, I(t_n)) \in P_S$[2],	*dann* $I(H) := W$.
	falls $(I(t_1), \ldots, I(t_n)) \notin P_S$,	*dann* $I(H) := F$.
Fall $H = P$		$I(P) := P_S$.

Rekursionsfälle:

[2] P_S ist die in durch I dem Symbol P zugeordnete Relation

Fall: $H = \neg F$	*dann* $I(H) = W$ *falls* $I(F) = F$
Fall: $H = F \vee G$	*dann* $I(H) = W$ *falls* $I(F) = W$ *oder* $I(G) = W$
Fall: $H = F \wedge G$,	*dann* $I(H) = W$ *falls* $I(F) = W$ *und* $I(G) = F$
Fall: $H = F \rightarrow G$	*dann* $I(H) = W$ *falls* $I(F) = F$ *oder* $I(G) = W$
Fall: $H = \forall x : F$	*dann* $I(H) = W$ *falls für alle* $a \in D_S : I[a/x](F) = W$
Fall: $H = \exists x : F$	*dann* $I(H) = W$ *falls für ein* $a \in D_S : I[a/x](F) = W$

Wir können nun direkt die schon aus der Aussagenlogik bekannten Begriffe eines*Modells*, der *Erfüllbarkeit*, *Allgemeingültigkeit*, *Tautologie* und *Widersprüchlichkeit* übernehmen. Insbesondere den Begriff der semantischen Folgerbarkeit ist wieder über die Interpretationen von Formeln definiert.

Definition 3.2.4 (semantische Folgerbarkeit in PL1)
Sei $M \subset For_{(\Sigma,V)}$ *und* $A \in For_{(\Sigma,V)}$ *. Aus* M *sei* A *folgerbar im Zeichen* $M \models_S A$ *genau dann wenn:*

für jede Interpretation I*, für die für jedes* $B \in M$ *ein Modell ist, gilt* I *ist auch ein Modell von* A*.*

3.2.4 Resolutionskalkül

Auch für die Prädikatenlogik kann in analoger Weise der Kalkül des natürlichen Schließens definiert wird. Wegen der schwer automatisierbaren Natur diesen Kalküls wollen wir darauf verzichten und uns gleich dem Resolutionskalkül für die Prädikatenlogik zuwenden. Er funktioniert für die Prädikatenlogik in analoger Weise wie auch schon für die Aussagenlogik. Es wird ein Widerlegungsbeweis geführt, die abzuleitende Formel wird also zunächst negiert. Es wird auch wieder auf der Klauselform gerechnet. Hier muß allerdings zunächst der Begriff der Klauselnormalform für die Prädikatenlogik erweitert werden.

Klauselnormalform

Definition 3.2.5 (Klauselnormalform)
Eine Formel der folgenden Form ist in Klauselnormalform:

$$\begin{array}{lll} \forall x_1 : \ldots \forall x_n : & & (L_{1,1} \vee \ldots \vee L_{1,n_1}) \\ & \wedge & (L_{2,1} \vee \ldots \vee L_{2,n_2}) \\ & \wedge & \\ & \ldots & \\ & \wedge & (L_{k,1} \vee \ldots \vee L_{1,n_k}) \end{array}$$

Alle Variablen sind dabei durch Allquantoren gebunden, die alle am Anfang der Formel stehen. Die Formel ist dann eine Konjunktion von Disjunktionen (Klauseln) von Literalen.

Wie man sieht muß man auf irgendeine Weise, um die Klauselnormalform herzustellen, die Existenzquantoren eliminieren. Wir können zwar Formel der Form $\exists x : F$ äquivalent als $\neg\forall x : \neg F$ aber dann steht aber ein Negationszeichen nicht direkt vor einem Prädikat, was der Klauselnormalform widerspricht. Es ist ein anderer Trick notwendig, um Existenzquantoren zu eliminieren, die Skolemisierung.

Skolemisierung Die Elimination von Existenzquantoren ist die sogenannte Skolemisierung (Nach Thoralf Skolem).

Die Idee dabei ist, wenn eine Formel die Existenz eines bestimmten Individuums durch einen Existenzquantor ausdrückt, für dieses Individuum dann einen Namen in Form einer Konstanten einzuführen. Betrachte man hierzu einmal die Aussage, daß es jemanden gibt, der das Individuum mit Namen Elvis liebt:

$$\exists x : \mathit{liebt}(x, \text{Elvis})$$

Die Idee der Skolemisierung ist, zu sagen, na wenn so ein Individuum existiert, dann geben wir ihm halt in Form einer Konstanten einen Namen. Laut der Semantik wird diese Konstante dan auf ein Element der nichtleeren Trägermenge abgebildet, und für dieses Element dann die nachfolgende Teilformel als wahr interpretiert. Geben wir in dem Beispiel also dem x einen Namen:

$$\mathit{liebt}(\text{Priscilla}, \text{Elvis})$$

Ganz so einfach geht es im Allgemeinen allerdings nicht. Wenn vor einem Existenzquantor Allquantoren stehen, so wird es nicht ausreichen, nur eine neue Konstante einzuführen. Betrachte man dazu die etwas allgemeinere Aussage, das sich für jeden jemand findet, der ihn liebt:

$$\forall y : \exists x : \mathit{liebt}(x, y)$$

Würden wir darin den Existenzquantor durch Einführung einer neuen Konstanten eliminieren, also zur Formel:

$$\forall x : \mathit{liebt}(\text{Priscilla}, x)$$

So erhalten wir die Aussage, daß ein bestimmtes Individuum (Priscilla) alle anderen liebt. Was man machen muß, ist nicht eine neue Konstante einführen, sondern eine Funktion, die das versprochene Individuum in Abhängigkeit der zuvor allquantifizierten Variablen berechnet. In unserem Beispiel also:

$$\forall x : \mathit{liebt}(\text{derLiebende}(x), x)$$

Die Funktion *derLiebende* berechnet für jedes Element das Element, das nach der ursprünglichen Aussage existieren soll.

Im nächsten Theorem sei $G[x_1, \ldots, x_n, y]$ eine beliebige Formel, die die Variablensymbole $x_1, \ldots, x_n, y$ frei enthält und $G[x_1, \ldots, x_n, t]$ eine Variante von F, in der alle Vorkommnisse von y durch t ersetzt sind.

Satz 3.2.6 Skolemisierung
Eine Formel $F = \forall x_1 \ldots x_n : \exists y : G[x_1, \ldots, x_n, y]$ ist (un-)erfüllbar gdw. $F' = \forall x_1 \ldots x_n : G[x_1, \ldots, x_n, f(x_1, \ldots, x_n)]$ (un-)erfüllbar ist, wobei f ein n-stelliges Funktionssymbol ist, das nicht in G vorkommt.

Beispiel 3.2.7 Skolemisierung

$$\begin{array}{rll} \exists x : P(x) & \text{wird zu} & P(a) \\ \forall x : \exists y : Q(f(y,y),x,y) & \text{wird zu} & \forall x : Q(f(g(x),g(x)),x,g(x)) \\ \forall x,y : \exists z : x+z=y & \text{wird zu} & \forall x,y : x+h(x,y)=y. \end{array}$$

Beispiel 3.2.8 *Skolemisierung erhält i.a. nicht die Allgemeingültigkeit (Falsifizierbarkeit):*

$\forall x : P(x) \vee \neg\forall x : P(x)$ *ist eine Tautologie*

$\forall x : P(x) \vee \exists x : \neg P(x)$ *ist äquivalent zu*

$\forall x : P(x) \vee \neg P(a)$ *nach Skolemisierung.*

Eine Interpretation, die die skolemisierte Formel falsifiziert kann man konstruieren wie folgt: Die Trägermenge ist $\{a,b\}$. Es gelte $P(a)$ und $\neg P(b)$. Die Formel ist aber noch erfüllbar.

Skolemisierung ist eine Operation, die nicht lokal innerhalb von Formeln verwendet werden darf, sondern nur global, d.h. wenn die ganze Formel eine bestimmte Form hat. Zudem bleibt bei dieser Operation nur die Unerfüllbarkeit der ganzen Klausel erhalten.

Mit der Skolemisierung steht jetzt der wichtigste Schritt zur Transformation einer prädikatenlogischen Formel in Klauselnormalform zur Verfügung. Insgesamt kann diese Transformation durch nachfolgenden siebenschrittigen Algorithmus erreicht werden. Drei der Schritte sind schon aus der Transformation einer Formel in Klauselform in der Aussagenlogik bekannt.

Definition 3.2.9 (Transformation in Klauselnormalform (unter Erhaltung der Unerfüllbarkeit))
Folgende Prozedur wandelt jede prädikatenlogische Formel in Klauselform (CNF) um:

1. *Elimination von $\rightarrow$:* $F \rightarrow G$ wird zu $\neg F \vee G$
2. *Negation ganz nach innen schieben:*

$$\begin{array}{lll} \neg\neg F & \text{wird zu} & F \\ \neg(F \wedge G) & \text{wird zu} & \neg F \vee \neg G \\ \neg(F \vee G) & \text{wird zu} & \neg F \wedge \neg G \\ \neg\forall x : F & \text{wird zu} & \exists x : \neg F \\ \neg\exists x : F & \text{wird zu} & \forall x : \neg F \end{array}$$

3. *Skopus von Quantoren minimieren, d.h. Quantoren so weit wie möglich nach innen schieben*

$$\begin{array}{lll} \forall x : (F \wedge G) & \text{wird zu} & (\forall x : F) \wedge G \quad \textit{falls } x \textit{ nicht frei in } G \\ \forall x : (F \vee G) & \text{wird zu} & (\forall x : F) \vee G \quad \textit{falls } x \textit{ nicht frei in } G \\ \exists x : (F \wedge G) & \text{wird zu} & (\exists x : F) \wedge G \quad \textit{falls } x \textit{ nicht frei in } G \\ \exists x : (F \vee G) & \text{wird zu} & (\exists x : F) \vee G \quad \textit{falls } x \textit{ nicht frei in } G \\ \forall x : (F \wedge G) & \text{wird zu} & \forall x : F \wedge \forall x : G \\ \exists x : (F \vee G) & \text{wird zu} & \exists x : F \vee \exists x : G \end{array}$$

4. *Alle gebundenen Variablen sind systematisch umzubenennen, um Namenskonflikte aufzulösen.*

5. *Existenzquantoren werden durch Skolemisierung eliminiert*
6. *Allquantoren löschen (alle Variablen werden als allquantifiziert angenommen.*
7. *Distributivität (und Assoziativität, Kommutativität) iterativ anwenden, um $\wedge$ nach außen zu schieben ("Ausmultiplikation"). $F \vee (G \wedge H)$* wird zu *$(F \vee G) \wedge (F \vee H)$ (Das duale Distributivgesetz würde eine disjunktive Normalform ergeben.)*

Das Resultat dieser Prozedur ist eine Konjunktion von Disjunktionen (Klauseln) von Literalen:

$$\begin{array}{ll} & (L_{1,1} \vee \ldots \vee L_{1,n_1}) \\ \wedge & (L_{2,1} \vee \ldots \vee L_{2,n_2}) \\ \wedge & \\ \ldots & \\ \wedge & (L_{k,1} \vee \ldots \vee L_{1,n_k}) \end{array}$$

oder in Mengenschreibweise:

$$\begin{array}{l} \{\{L_{1,1}, \ldots, L_{1,n_1}\}, \\ \{L_{2,1}, \ldots, L_{2,n_2}\}, \\ \ldots \\ \{L_{k,1}, \ldots, L_{1,n_k}\}\} \end{array}$$

Implementierung

PL1.scala

```
31 import scala.collection.mutable.HashSet
32
33 abstract class PL1 extends ToLaTeX{
34   def freeVars():HashSet[Var]=freeVars(new HashSet(),new HashSet())
35   def freeVars(bound:HashSet[Var], result:HashSet[Var]):HashSet[Var]
36    ={
37     this match {
38       case Atom(_,args)
39        => for (val arg:Term <- args)  arg.freeVars(bound,result)
40       case Not(x)       => x.freeVars(bound,result)
41       case Or(e1,e2)
42        => e1.freeVars(bound,result);e2.freeVars(bound,result)
43       case And(e1,e2)
44        => e1.freeVars(bound,result);e2.freeVars(bound,result)
45       case Impl(e1,e2)
46        => e1.freeVars(bound,result);e2.freeVars(bound,result)
47       case ForAll(x,e) => bound += x;e.freeVars(bound,result)
48       case Exists(x,e) => bound += x;e.freeVars(bound,result)
49     }
50     result
51   }
```

PL1.scala

```
52   def eliminateImpl():PL1={
53    this match {
```

```
54      case Atom(_,_)    => this
55      case Not(x)       => Not(x.eliminateImpl)
56      case And(x,y)     => And(x.eliminateImpl,y.eliminateImpl)
57      case Or(x,y)      => Or(x.eliminateImpl,y.eliminateImpl)
58      case ForAll(x,y)  => ForAll(x,y.eliminateImpl)
59      case Exists(x,y)  => Exists(x,y.eliminateImpl)
60      case Impl(x,y)    => Or(Not(x.eliminateImpl),y.eliminateImpl)
61
62    }
63   }
```

PL1.scala

```
64   def negationToInner():PL1={
65    this match {
66     case Atom(_,_)       => this
67     case Not(Atom(_,_))  => this
68     case Not(Not(x))     => x.negationToInner
69     case Not(And(x,y))   => Or(Not(x),Not(y)).negationToInner
70     case Not(Or(x,y))    => And(Not(x),Not(y)).negationToInner
71     case Not(ForAll(x,e))=> Exists(x,Not(e).negationToInner)
72     case Not(Exists(x,e))=> ForAll(x,Not(e).negationToInner)
73     case And(x,y)        => And(x.negationToInner,y.negationToInner)
74     case Or(x,y)         => Or(x.negationToInner,y.negationToInner)
75     case Exists(x,e)     => Exists(x,e.negationToInner)
76     case ForAll(x,e)     => ForAll(x,e.negationToInner)
77    }
78   }
```

PL1.scala

```
79   def minimizeScope():PL1=
80    this match {
81     case ForAll(x,And(f,g)) =>
82       val freeG = g.freeVars
83       if (!freeG.contains(x))
84         And(ForAll(x,f).minimizeScope,g.minimizeScope)
85       else {
86         val freeF=f.freeVars
87         if (!freeF.contains(x))
88           And(f.minimizeScope,ForAll(x,g).minimizeScope)
89         else And(ForAll(x,f).minimizeScope,ForAll(x,g).minimizeScope)
90       }
91     case Exists(x,Or(f,g)) =>
92       val freeG = g.freeVars
93       if (!freeG.contains(x))
94         Or(Exists(x,f).minimizeScope,g.minimizeScope)
95       else {
96         val freeF=f.freeVars
97         if (!freeF.contains(x))
98           Or(f.minimizeScope,Exists(x,g).minimizeScope)
```

```
99          else And(Exists(x,f).minimizeScope,Exists(x,g).minimizeScope)
100       }
101     case ForAll(x,Or(f,g)) =>
102       val freeG = g.freeVars
103       if (!freeG.contains(x))
104         Or(ForAll(x,f).minimizeScope,g.minimizeScope)
105       else {
106         val freeF=f.freeVars
107         if (!freeF.contains(x))
108           Or(f.minimizeScope,ForAll(x,g).minimizeScope)
109         else ForAll(x,Or(f.minimizeScope,g.minimizeScope))
110       }
111     case Exists(x,And(f,g)) =>
112       val freeG = g.freeVars
113       if (!freeG.contains(x))
114         And(Exists(x,f).minimizeScope,g.minimizeScope)
115       else {
116         val freeF=f.freeVars
117         if (!freeF.contains(x))
118           And(f.minimizeScope,Exists(x,g).minimizeScope)
119         else Exists(x,And(f.minimizeScope,g.minimizeScope))
120       }
121     case And(f,g) => And(f.minimizeScope,g.minimizeScope)
122     case Or(f,g) => Or(f.minimizeScope,g.minimizeScope)
123     case Not(f) => Not(f.minimizeScope)
124     case _ => this
125   }
```

PL1.scala

```
126 import scala.collection.mutable.Map
127 import scala.collection.mutable.HashMap
128 def renameVars():PL1=renameVars(new Counter(0),new HashMap())
129 def renameVars(i:Counter,map:HashMap[Var,Term]):PL1=
130  this match {
131    case And(f,g) =>And(f.renameVars(i,map),g.renameVars(i,map))
132    case Or(f,g) =>Or(f.renameVars(i,map),g.renameVars(i,map))
133    case Not(f) =>Not(f.renameVars(i,map))
134    case Atom(n,args) =>
135       Atom(n,(for (val arg<-args) yield arg.substitute(map)))
136    case ForAll(x,e) =>
137       val newX = Var("x_"+i.n.toString)
138       map += x -> newX
139       ForAll(newX,e.renameVars(i,map))
140    case Exists(x,e) =>
141       val newX = Var("x_"+i.n.toString)
142       map += x -> newX
143       Exists(newX,e.renameVars(i,map))
144  }
145
```

```
146   def substitute(map :Map[Var,Term]):PL1=
147    this match {
148     case Atom(x,args)
149       => Atom(x,for(val a<-args) yield a.substitute(map))
150     case Not(p)          => Not(p.substitute(map))
151     case And(x,y)        => And(x.substitute(map),y.substitute(map))
152     case Or(x,y)         => Or(x.substitute(map),y.substitute(map))
153     case Exists(x,e)     =>
154       map -= x
155       Exists(x,e.substitute(map))
156     case ForAll(x,e)     =>
157       map -= x
158       ForAll(x,e.substitute(map))
159    }
```

PL1.scala

```
160   def skolemize():PL1=skolemize(new Counter(0),List[Var]())
161   def skolemize(i:Counter,forallVars:List[Var]):PL1=
162    this match {
163      case And(f,g)
164         => And(f.skolemize(i,forallVars),g.skolemize(i,forallVars))
165      case Or(f,g)
166         => Or(f.skolemize(i,forallVars),g.skolemize(i,forallVars))
167      case Not(f)      => Not(f.skolemize(i,forallVars))
168      case Atom(_,_)   => this
169      case ForAll(x,e) => ForAll(x,e.skolemize(i,x::forallVars))
170      case Exists(x,e) =>
171        val skolem = Fun("s$_"+i.n.toString+"$"
172                        ,for (val v<-forallVars.toList) yield v
173                        )
174        val map=new HashMap[Var,Term]()
175        map += x -> skolem
176        e.substitute(map).skolemize(i,forallVars)
177    }
```

PL1.scala

```
178    def deleteAllQuantification():PL1=
179    this match {
180      case And(f,g)
181        => And(f.deleteAllQuantification,g.deleteAllQuantification)
182      case Or(f,g)
183        => Or(f.deleteAllQuantification,g.deleteAllQuantification)
184      case Not(f)      => Not(f.deleteAllQuantification)
185      case Atom(_,_)   => this
186      case ForAll(x,e) => e.deleteAllQuantification
187      case Exists(x,e) => Exists(x,e.deleteAllQuantification)
188    }
```

PL1.scala

```
189  def makeConjunctions():PL1=
190   this match {
191    case Atom(_,_)     => this
192    case Not(x)        => this
193    case Or(And(x,y),z)=> And(Or(x,z),Or(y,z)).makeConjunctions
194    case Or(x,And(y,z))=> And(Or(x,y),Or(x,z)).makeConjunctions
195    case Or(x,y)
196      => val x1 =x.makeConjunctions
197         val y1 =y.makeConjunctions
198         x1 match {
199           case And(_,_) => Or(x1,y1).makeConjunctions
200           case _
201             => y1 match {
202                   case And(_,_) => Or(x1,y1).makeConjunctions
203                   case _        => Or(x1,y1)
204                }
205         }
206    case And(x,y)      => And(x.makeConjunctions,y.makeConjunctions)
207   }
```

PL1.scala

```
208  def toCNF():PL1
209     = eliminateImpl
210      .negationToInner
211      .minimizeScope
212      .renameVars
213      .skolemize
214      .deleteAllQuantification
215      .makeConjunctions
```

PL1.scala

```
216  import scala.collection.mutable.HashSet
217  import scala.collection.mutable.Set
218  def conjunctionSet():Set[PL1]={
219    val result=new HashSet[PL1]()
220    this match {
221      case And(e1,e2) => result++= e1.conjunctionSet;
222                         result++= e2.conjunctionSet;
223      case e => result+= e
224    }
225    result
226  }
227
228  type Klausel=Set[PL1]
229  def disjunctionSet():Klausel={
230    val result=new HashSet[PL1]()
231    this match {
232      case Or(e1,e2) => result++= e1.disjunctionSet;
```

```
                        result++= e2.disjunctionSet;
      case e => result+= e
    }
    result
  }

  import scala.collection.mutable.HashSet
  def cnf():List[Klausel]=toCNF.cnfAsSet

  def cnfAsSet():List[Klausel]={
    var result=List[Klausel]()
    for (val f<- conjunctionSet.elements)
       result = (f.disjunctionSet) :: result
    result
  }

  def negated()= this match { case Not(x) => x
                              case y      => Not(y)}

}
```

PL1Util.scala

```
package name.panitz.ki.pl1
object PL1Util {
  import scala.collection.mutable.Set
  type Klausel=Set[PL1]

  def toLaTeX(cl:Klausel):String={
     var result="\\{"
     var first=true
     for (val c<-cl){
       if (first) first=false else result=result+", "
       result=result+c.toLaTeX
     }
     result+"\\}\n"
  }

  def toLaTeX(cls:List[Klausel]):String={
    var result="\\[\\begin{array}{l}\\\\\\{"
    var first=true
    for (val cl <- cls){
      if (first) first=false else result=result+",\\\\\n    ~"
      result=result+toLaTeX(cl)
    }
    result+"\n\\end{array}\n\\]\n"
  }
}
```

MakeClauses.scala

```
1  package name.panitz.ki.pl1
2  object MakeClauses {
3   def run(f:PL1)={
4    try{
5     Console println "Ausgangsformel:"
6     Console println ("\n\\["+f.toLaTeX+"\\]\n")
7
8     var x1 = f.eliminateImpl()
9     Console println "Implikationen eliminiert:"
10    Console println ("\n\\["+x1.toLaTeX+"\\]\n")
11
12    Console println "Negationen nach innen:"
13    var x2=x1.negationToInner
14    Console println ("\n\\["+x2.toLaTeX+"\\]\n")
15
16    Console println "Quantorenskopus minimiert:"
17    var x3=x2.minimizeScope
18    Console println ("\n\\["+x3.toLaTeX+"\\]\n")
19
20    Console println "Variablen umbenennen:"
21    var x4=x3.renameVars
22    Console println ("\n\\["+x4.toLaTeX+"\\]\n")
23
24    Console println "Skolemisieren:"
25    var x5=x4.skolemize
26    Console println ("\n\\["+x5.toLaTeX+"\\]\n")
27
28    Console println "Allquantoren löschen:"
29    var x6=x5.deleteAllQuantification
30    Console println ("\n\\["+x6.toLaTeX+"\\]\n")
31
32    Console println "Und-/Oder- Ausmultiplizieren:"
33    var x7=x6.makeConjunctions
34    Console println ("\n\\["+x7.toLaTeX+"\\]\n")
35
36    Console println "Klauseln:"
37    Console println ("\n"+PL1Util.toLaTeX(x7.cnfAsSet.toList)+"\n")
38   }catch {
39     case e => e.printStackTrace()
40   }
41  }
42 }
```

Beispiele

Beispiel 3.2.10 *Das obige Programm ermöglicht jetzt die Umformung einer Formel in eine Klauselform automatisch durchzuführen, und in LaTeXzu formattieren. Betrachten wir zunächst folgende Formel über Tierliebe:*

TierFormel.scala

```
1  package name.panitz.ki.pl1
2  object TierFormel extends Application{
3    val f1=ForAll(Var("x")
4                 ,Impl(ForAll(Var("y")
5                              ,Impl(Atom("Tier",List(Var("y")))
6                                   ,Atom("Liebt",List(Var("x"),Var("y")))
7                                   )
8                              )
9                        ,Exists(Var("z"),Atom("Liebt",List(Var("z"),Var("x"))))
10                       )
11                )
12   MakeClauses.run(f1)
13 }
```

Ausgangsformel:

$$(\forall x.((\forall y.(\text{Tier}(y) \rightarrow \text{Liebt}(x,y))) \rightarrow (\exists z.\text{Liebt}(z,x))))$$

Implikationen eliminiert:

$$(\forall x.(\neg(\forall y.(\neg\text{Tier}(y) \vee \text{Liebt}(x,y))) \vee (\exists z.\text{Liebt}(z,x))))$$

Negationen nach innen:

$$(\forall x.((\exists y.(\text{Tier}(y) \wedge \neg\text{Liebt}(x,y))) \vee (\exists z.\text{Liebt}(z,x))))$$

Quantorenskopus minimiert:

$$(\forall x.((\exists y.(\text{Tier}(y) \wedge \neg\text{Liebt}(x,y))) \vee (\exists z.\text{Liebt}(z,x))))$$

Variablen umbenennen:

$$(\forall x_1.((\exists x_2.(\text{Tier}(x_2) \wedge \neg\text{Liebt}(x_1,x_2))) \vee (\exists x_3.\text{Liebt}(x_3,x_1))))$$

Skolemisieren:

$$(\forall x_1.((\text{Tier}(\text{s}_1(x_1)) \wedge \neg\text{Liebt}(x_1,\text{s}_1(x_1))) \vee \text{Liebt}(\text{s}_2(x_1),x_1)))$$

Allquantoren löschen:

$$((\text{Tier}(\text{s}_1(x_1)) \wedge \neg\text{Liebt}(x_1,\text{s}_1(x_1))) \vee \text{Liebt}(\text{s}_2(x_1),x_1))$$

Und-/Oder- Ausmultiplizieren:

$$((\text{Tier}(\text{s}_1(x_1)) \vee \text{Liebt}(\text{s}_2(x_1),x_1)) \wedge (\neg\text{Liebt}(x_1,\text{s}_1(x_1)) \vee \text{Liebt}(\text{s}_2(x_1),x_1)))$$

Klauseln:

$$\{\{\mathrm{Tier}(\mathrm{s}_1(x_1)), \mathrm{Liebt}(\mathrm{s}_2(x_1), x_1)\},\\ \{\neg\mathrm{Liebt}(x_1, \mathrm{s}_1(x_1)), \mathrm{Liebt}(\mathrm{s}_2(x_1), x_1)\}$$

Beispiel 3.2.11 *Lassen wir das Programm noch ein weiteres Beispiel rechnen:*

Blatt6A1.scala

```
package name.panitz.ki.pl1
object Blatt6A1 extends Application{
  val f= Or(ForAll(Var ("x")
                  ,Impl(ForAll(Var ("y")
                               ,Atom("Q",List(Var ("x"),Var ("y")))
                               )
                        ,Atom("P",List(Fun("f",List(Var ("x")))))
                        )
                  )
           ,Exists(Var ("z")
                  ,And(Not (Exists (Var ("y"),Atom("P",List(Var ("y"))))))
                      ,Atom("Q",List(Fun("f",List(Var ("z"))),Var ("z")))
                      )
                  )
           )
  MakeClauses.run (f)
}
```

Ausgangsformel:

$$((\forall x.((\forall y.\mathrm{Q}(x,y)) \rightarrow \mathrm{P}(\mathrm{f}(x)))) \vee (\exists z.(\neg(\exists y.\mathrm{P}(y)) \wedge \mathrm{Q}(\mathrm{f}(z), z))))$$

Implikationen eliminiert:

$$((\forall x.(\neg(\forall y.\mathrm{Q}(x,y)) \vee \mathrm{P}(\mathrm{f}(x)))) \vee (\exists z.(\neg(\exists y.\mathrm{P}(y)) \wedge \mathrm{Q}(\mathrm{f}(z), z))))$$

Negationen nach innen:

$$((\forall x.((\exists y.\neg\mathrm{Q}(x,y)) \vee \mathrm{P}(\mathrm{f}(x)))) \vee (\exists z.((\forall y.\neg\mathrm{P}(y)) \wedge \mathrm{Q}(\mathrm{f}(z), z))))$$

Quantorenskopus minimiert:

$$((\forall x.((\exists y.\neg\mathrm{Q}(x,y)) \vee \mathrm{P}(\mathrm{f}(x)))) \vee ((\forall y.\neg\mathrm{P}(y)) \wedge (\exists z.\mathrm{Q}(\mathrm{f}(z), z))))$$

Variablen umbenennen:

$$((\forall x_1.((\exists x_2.\neg\mathrm{Q}(x_1,x_2)) \vee \mathrm{P}(\mathrm{f}(x_1)))) \vee ((\forall x_3.\neg\mathrm{P}(x_3)) \wedge (\exists x_4.\mathrm{Q}(\mathrm{f}(x_4), x_4))))$$

Skolemisieren:

$$((\forall x_1.(\neg \mathrm{Q}(x_1, \mathrm{s}_1(x_1)) \vee \mathrm{P}(\mathrm{f}(x_1)))) \vee ((\forall x_3.\neg \mathrm{P}(x_3)) \wedge \mathrm{Q}(\mathrm{f}(\mathrm{s}_2()), \mathrm{s}_2())))$$

Allquantoren löschen:

$$((\neg \mathrm{Q}(x_1, \mathrm{s}_1(x_1)) \vee \mathrm{P}(\mathrm{f}(x_1))) \vee (\neg \mathrm{P}(x_3) \wedge \mathrm{Q}(\mathrm{f}(\mathrm{s}_2()), \mathrm{s}_2())))$$

Und-/Oder- Ausmultiplizieren:

$$(((\neg \mathrm{Q}(x_1, \mathrm{s}_1(x_1)) \vee \mathrm{P}(\mathrm{f}(x_1))) \vee \neg \mathrm{P}(x_3)) \wedge ((\neg \mathrm{Q}(x_1, \mathrm{s}_1(x_1)) \vee \mathrm{P}(\mathrm{f}(x_1))) \vee \mathrm{Q}(\mathrm{f}(\mathrm{s}_2()), \mathrm{s}_2())))$$

Klauseln:

$$\{\{\mathrm{Q}(\mathrm{f}(\mathrm{s}_2()), \mathrm{s}_2()), \mathrm{P}(\mathrm{f}(x_1)), \neg \mathrm{Q}(x_1, \mathrm{s}_1(x_1))\},$$
$$\{\neg \mathrm{P}(x_3), \mathrm{P}(\mathrm{f}(x_1)), \neg \mathrm{Q}(x_1, \mathrm{s}_1(x_1))\}$$

Resolution

Mit der Klauselform liegen uns die Formeln jetzt in der von der Resolution benutzten Form vor. Betrachten wir hierzu zunächst ein einfaches Beispiel:

Beispiel 3.2.12 *Seien die folgenden bekannten Fakten gegeben:*

$$\forall x : \mathrm{Mensch}(x) \rightarrow \mathrm{sterblich}(x)$$

$$\forall x : \mathrm{Mensch}(x) \rightarrow \mathrm{Mensch}(\mathrm{Vater}(x))$$

$$\mathrm{Mensch}(\mathrm{Elvis})$$

Es soll hieraus bewiesen werden:

$$\mathrm{sterblich}(\mathrm{Vater}(\mathrm{Elvis}))$$

Wir negieren die zu beweisende Aussage und transformieren in Klauselform:

$$\{\neg \mathrm{Mensch}(x_1), \mathrm{sterblich}(x_1)\} \tag{3.14}$$

$$\{\neg \mathrm{Mensch}(x_2), \mathrm{sterblich}(\mathrm{Vater}(x_2))\} \tag{3.15}$$

$$\{\mathrm{Mensch}(\mathrm{Elvis})(x_2)\} \tag{3.16}$$

$$\{\neg \mathrm{sterblich}(\mathrm{Vater}(\mathrm{Elvis}))\} \tag{3.17}$$

Nun würden wir gerne auf dieser Klauselmenge Resolventen bilden. Hier würden sich z.B. Klausel 3.14 und 3.17 anbieten. Hier taucht das Prädikat *sterblich* einmal negiert und einmal unnegiert auf. Allerdings sind die Argumente des Prädikats recht unterschiedlich: einmal hanelt es sich um eine Variabel und einmal um den Grundterm *Vater(Elvis)*.

Die Variablen in den Klauseln sind alle implizit allquantifiert, also steht eine Klausel, die Vriablen enthält für alle Klauseln, in der diese Variablen durch Grundterme ersetzt wurden. In unserem Beispiel also:

$$\{\neg\text{Mensch}(\text{Elvis}), \text{sterblich}(\text{Elvis})\} \tag{3.18}$$

$$\{\neg\text{Mensch}(\text{Vater}(\text{Elvis})), \text{sterblich}(\text{Vater}(\text{Elvis}))\} \tag{3.19}$$

$$\{\neg\text{Mensch}(\text{Vater}(\text{Vater}(\text{Elvis}))), \text{sterblich}(\text{Vater}(\text{vater}(\text{Elvis})))\} \tag{3.20}$$

$$\{\neg\text{Mensch}(\text{Vater}(\text{Vater}(\text{Vater}(\text{Elvis})))), \text{sterblich}(\text{Vater}(\text{Vater}(\text{Vater}(\text{Elvis}))))\} \tag{3.21}$$

$$\{\neg\text{Mensch}(\text{Vater}(\text{Vater}(\text{Vater}(\text{Vater}(\text{Elvis}))))), \text{sterblich}(\text{Vater}(\text{Vater}(\text{Vater}(\text{Vater}(\text{Elvis})))))\} \tag{3.22}$$

$$\vdots$$

Klauseln mit Variablen stehen also für abzählbar undendlich viele Grundklauseln. Wir könnten jetzt Klausel 3.14 mit 3.19 resolvieren. Und damit hat man im Prinzip schon ein Beweisverfahren für die Prädikatenlogik. Man kann über einen Iterator die durch die prädikatenlogischen Klauseln definierten unendlich vielen Grundklauseln aufzählen. Dann läßt sich, da die unendlich viel Klauseln ja abzählbar sind, ein Iterator aller schreiben, der alle möglichen Resoventen findet. Wir erhalten dann einen unendlichen großen Suchbaum, in dem wir aber erschöpfend nach der leeren Klausel, die den Widerspruch repräsentiert suchen können. Wenn also die leer Klausel per Resolution hergeleitet werden kann, werden wir sie finden. Was passiert hingegen, wenn es keinen Beweis gibt? Dann werden wir unter Umständen immer tiefer in dem unendlichen Suchbaum suchen. Die Suche terminiert nicht. Unser Beweisverfahren findet also ntweder einen Beweis oder terminiert nicht. Dieses sind gerade die wichtigsten Eigenschaften der Prädikatenlogik in Bezug auf die Berechenbarkeit:

Es gilt die Unentscheidbarkeit der Prädikatenlogik:

Satz 3.2.13 *Es ist unentscheidbar, ob eine geschlossene Formel der Prädikatenlogik allgemeingültig ist.*

Einen Beweis geben wir nicht. Der Beweis besteht darin, ein Verfahren anzugeben, das jeder Turingmaschine M eine prädikatenlogische Formel zuordnet, die genau dann ein Satz ist, wenn diese Turingmaschine auf dem leeren Band terminiert. Hierbei nimmt man TM, die nur mit einem Endzustand terminieren können. Da das Halteproblem für Turingmaschinen unentscheidbar ist, hat man damit einen Beweis für den Satz.

Satz 3.2.14 *Die Menge der allgemeingültigen Formeln der Prädikatenlogik ist rekursiv aufzählbar.*

Als Schlußfolgerung kann man sagen, dass es kein Deduktionssystem gibt (Algorithmus), das bei eingegebener Formel nach endlicher Zeit entscheiden kann, ob die Formel ein Satz ist oder nicht. Allerdings gibt es einen Algorithmus, der für jede Formel, die ein Satz ist, auch terminiert und diese als Satz erkennt.

Genau einen solchen Algorithmus haben wir oben beschrieben.

Unifikation

Definition 3.2.15 (Unifikationsalgorithmus $U1$:)

Eingabe: *zwei Terme oder Atome s und t:*

Ausgabe: *"nicht unifizierbar" oder einen allgemeinsten Unifikator:*

Zustände: *auf denen der Algorithmus operiert: Eine Menge Γ von Gleichungen.*

Initialzustand: $\Gamma_0 = \{s \stackrel{?}{=} t\}$.

Unifikationsregeln:

$$\frac{s_1 \stackrel{?}{=} t_1 \ , \ \ldots \ , \ s_n \stackrel{?}{=} t_n, \Gamma}{f(s_1, \ldots, s_n) \stackrel{?}{=} f(t_1, \ldots, t_n), \Gamma} \qquad \textit{(Dekomposition)}$$

$$\frac{\Gamma}{x \stackrel{?}{=} x, \Gamma} \qquad \textit{(Tautologie)}$$

$$\frac{x \stackrel{?}{=} t, \{x \mapsto t\}\Gamma}{x \stackrel{?}{=} t, \Gamma} \qquad x \in FV(\Gamma),\ x \notin FV(t) \qquad \textit{(Anwendung)}$$

$$\frac{x \stackrel{?}{=} t, \Gamma}{t \stackrel{?}{=} x, \Gamma} \qquad t \notin V \qquad \textit{(Orientierung)}$$

Abbruchbedingungen:

$$\frac{Fail}{f(\ldots) \stackrel{?}{=} g(\ldots), \Gamma} \qquad \textit{wenn } f \neq g \qquad \textit{(Clash)}$$

$$\frac{Fail}{x \stackrel{?}{=} t, \Gamma} \qquad \textit{wenn } x \in FV(t) \textit{ vorkommt und } t \neq x \qquad \textit{(occurs check Fehler)}$$

Steuerung:

Starte mit $\Gamma = \Gamma_0$, und transformiere Γ solange durch (nichtdeterministische, aber nicht verzweigende) Anwendung der Regeln, bis entweder eine Abbruchbedingung erfüllt ist oder

keine Regel mehr anwendbar ist. Falls eine Abbruchbedingung erfüllt ist, terminiere mit "nicht unifizierbar". Falls keine Regel mehr anwendbar ist, hat die Gleichungsmenge die Form $\{x_1 \stackrel{?}{=} t_1, \ldots, x_k \stackrel{?}{=} t_k\}$*, wobei keine der Variablen* x_i *in einem* t_j *vorkommt; d.h. sie ist in gelöster Form. Das Resultat ist dann* $\{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}$*.*

Implementierung

Unification.scala

```
1  package name.panitz.ki.pl1
2
3  import scala.collection.mutable.Map
4  import scala.collection.mutable.HashMap
5  import java.lang.StringBuffer
6
7  object Unification{
8    def unify(t1:Term,t2:Term,sub:Map[Var,Term])
9          :Option[Pair[Map[Var,Term],List[Pair[Term,Term]]]]=
10     unify(t1,t2,sub,new StringBuffer())
11
12   def unify(t1:Term,t2:Term,sub:Map[Var,Term],out:StringBuffer)
13         :Option[Pair[Map[Var,Term],List[Pair[Term,Term]]]]=
14     Pair(t1,t2) match{
15       case Pair(Fun(n1,args1),Fun(n2,args2))
16        => if (n1==n2) {
17             out append ("{\\footnotesize Dekomposition}")
18             Some(Pair(sub,args1 zip args2))            //Decomposition
19           }else {
20               out append ("{\\footnotesize Clash: nicht unifizierbar}")
21               None                                     //Clash
22           }
23       case Pair(Var(x1),Var(x2))
24        => if (x1==x2){
25             out append ("{\\footnotesize Tautologie}")
26             Some(Pair(sub,List()))                     //Tautology
27           }else anwendung(Var(x1),Var(x2),sub,out)
28       case Pair(t,Var(x2)) =>
29         out append ("{\\footnotesize Orientierung}")
30         Some(Pair(sub,List(Pair(Var(x2),t))))
31               //,sub,out)                          //Orientation
32       case Pair(Var(x1),t)
33        => if (t.freeVars().contains(Var(x1))){
34             out append ("{\\footnotesize Occur Check: nicht unifizierbar}")
35             None                                       //Occur Check
36           }else anwendung(Var(x1),t,sub,out)           //Application
37       case _ => None
38     }
39
40   def anwendung(v:Var,t:Term,sub:Map[Var,Term])
41         :Option[Pair[Map[Var,Term],List[Pair[Term,Term]]]]=
42     anwendung(v,t,sub,new StringBuffer())
```

```
43
44  def anwendung(v:Var,t:Term,sub:Map[Var,Term],out:StringBuffer)={
45    out append ("{\\footnotesize Anwendung}")
46    val s = new HashMap[Var,Term]()
47    s += v -> t
48    val newSub = new HashMap[Var,Term]()
49    for (val p<-sub.elements)
50      newSub += p._1 -> p._2.substitute(s)
51    newSub +=  v -> t
52    Some(Pair(newSub,List()))
53  }
54
55  def unify(t1:Term,t2:Term)
56        :Option[Pair[Map[Var,Term],List[Pair[Term,Term]]]]=
57     unify(t1,t2,new StringBuffer())
58
59  def unify(t1:Term,t2:Term,out:StringBuffer)
60        :Option[Pair[Map[Var,Term],List[Pair[Term,Term]]]]=
61    unify(t1:Term,t2:Term,new HashMap[Var,Term](),out)
62
63  def unify(xs:List[Pair[Term,Term]],sub:Map[Var,Term])
64                    :Option[Map[Var,Term]]=
65    unify(xs,sub,new StringBuffer())
66
67  def unify(xs:List[Pair[Term,Term]],sub:Map[Var,Term],out:StringBuffer)
68                    :Option[Map[Var,Term]]={
69    out.append("&$"+toLaTeX(xs)+"~~"+toLaTeX(sub)+"$\\\\\\hline \n")
70    xs match {
71     case Nil =>
72       out.append("{\\footnotesize Unifikator}&$"+toLaTeX(sub)+"$")
73       Some(sub)
74     case Pair(t1,t2)::ys =>
75       unify(t1,t2,sub,out) match {
76         case None => None
77         case Some(Pair(newSub,further)) => {
78           val zs:List[Pair[Term,Term]]
79                  = (for (val y<-ys)
80                     yield Pair(y._1.substitute(newSub)
81                                ,y._2.substitute(newSub))) ::: further
82           val sub3 = new HashMap[Var,Term]()
83           for (val p<-sub.elements)
84            sub3 += p._1 -> p._2.substitute(newSub)
85           sub3 ++= newSub
86           unify(zs,sub3,out)
87         }
88       }
89    }
90  }
91
92  def unify(t1:PL1,t2:PL1):Option[Map[Var,Term]]
```

```
93      =unify(t1,t2,new StringBuffer())
94
95    def unify(t1:PL1,t2:PL1,out:StringBuffer):Option[Map[Var,Term]]={
96      out.append("\\begin{tabular}{lc}\n")
97      var result=
98       Pair(t1,t2) match {
99        case Pair(Atom(n1,args1),Atom(n2,args2))
100        => unify(List(Pair[Term,Term](Fun(n1,args1),Fun(n2,args2)))
101                ,new HashMap[Var,Term](),out)
102       case _ => None
103      }
104     out.append("\\end{tabular}\n")
105     result
106   }
107
108   def toLaTeX(sub:Option[Map[Var,Term]]):String=
109     sub match{
110       case None     => "\\mbox{\\rm no}"
111       case Some(x)  => toLaTeX(x)
112     }
113
114   def toLaTeX(sub:Map[Var,Term]):String={
115     var result="\\{"
116     var first=true
117     for (val p<-sub.elements){
118       if (first) first=false else result=result+","
119       result=result+p._1.toLaTeX+" \\mapsto "+p._2.toLaTeX
120     }
121     result+"\\}"
122   }
123
124   def toLaTeX(eqs:List[Pair[Term,Term]]):String={
125     var result =""
126     for (val Pair(left,right)<-eqs)
127       result=result+left.toLaTeX+" = "+right.toLaTeX+"~~"
128     result
129   }
130
131 }
```

Testläufe Mit dem obigen Programm lasen sich nun Unifikatoren berechnen und die Berechnung in Einzelschritten in LaTeXausgeben, so daß die Arbeitsweise der Unifikation schrittweise nachvollzogen werden kann.

TestUnification.scala

```
1  package name.panitz.ki.pl1
2  import Unification.{_}
3  object TestUnification extends Application{
4    val t1a=Atom("Q",List(Var("y")
5                ,Fun("G",List(Fun("A",List()),Fun("B",List())))))
6    val t1b=Atom("Q",List(Var("y")
```

```
7               ,Fun("G",List(Fun("A",List()),Fun("A",List())))))
8     val t2=Atom("Q",List(Fun("G",List(Var("x"),Var("x")))
9                         ,Var("y")))
10
11    val uni=new java.lang.StringBuffer()
12    unify(t1a,t2,uni)
13    unify(t1b,t2,uni)
14
15    val p1=Atom("H",List(Fun("g",List(Fun("f",List(Var("x"),Var("u")))))
16                        ,Fun("f",List(Fun("g",List(Fun("a",List()))))
17                                     ,Fun("f",List(Fun("a",List()),Fun("a",List())))))
18                            )
19                        ,Fun("f",List(Var("u"),Fun("f",List(Var("x"),Var("u")))))
20                        )
21              )
22    val p2=Atom("H"
23               ,List(Fun("g",List(Var("v"))),Var("v"),Fun("f",List(Var("y"),Var("z")))))
24
25    unify(p1,p2,uni)
26    Console println uni
27  }
```

Nachstehend die drei Läufe des Unifikationsalgorithmus:

	$\mathrm{Q}(y, \mathrm{G}(\mathrm{A}(), \mathrm{B}())) = \mathrm{Q}(\mathrm{G}(x, x), y) \quad \{\}$
Dekomposition	$y = \mathrm{G}(x, x) \;\; \mathrm{G}(\mathrm{A}(), \mathrm{B}()) = y \quad \{\}$
Anwendung	$\mathrm{G}(\mathrm{A}(), \mathrm{B}()) = \mathrm{G}(x, x) \quad \{y \mapsto \mathrm{G}(x, x)\}$
Dekomposition	$\mathrm{A}() = x \;\; \mathrm{B}() = x \quad \{y \mapsto \mathrm{G}(x, x)\}$
Orientierung	$\mathrm{B}() = x \;\; x = \mathrm{A}() \quad \{y \mapsto \mathrm{G}(x, x)\}$
Orientierung	$x = \mathrm{A}() \;\; x = \mathrm{B}() \quad \{y \mapsto \mathrm{G}(x, x)\}$
Anwendung	$\mathrm{A}() = \mathrm{B}() \quad \{y \mapsto \mathrm{G}(\mathrm{A}(), \mathrm{A}()), x \mapsto \mathrm{A}()\}$

Clash: nicht unifizierbar

	$\mathrm{Q}(y, \mathrm{G}(\mathrm{A}(), \mathrm{A}())) = \mathrm{Q}(\mathrm{G}(x, x), y) \quad \{\}$
Dekomposition	$y = \mathrm{G}(x, x) \;\; \mathrm{G}(\mathrm{A}(), \mathrm{A}()) = y \quad \{\}$
Anwendung	$\mathrm{G}(\mathrm{A}(), \mathrm{A}()) = \mathrm{G}(x, x) \quad \{y \mapsto \mathrm{G}(x, x)\}$
Dekomposition	$\mathrm{A}() = x \;\; \mathrm{A}() = x \quad \{y \mapsto \mathrm{G}(x, x)\}$
Orientierung	$\mathrm{A}() = x \;\; x = \mathrm{A}() \quad \{y \mapsto \mathrm{G}(x, x)\}$
Orientierung	$x = \mathrm{A}() \;\; x = \mathrm{A}() \quad \{y \mapsto \mathrm{G}(x, x)\}$
Anwendung	$\mathrm{A}() = \mathrm{A}() \quad \{y \mapsto \mathrm{G}(\mathrm{A}(), \mathrm{A}()), x \mapsto \mathrm{A}()\}$
Dekomposition	$\{y \mapsto \mathrm{G}(\mathrm{A}(), \mathrm{A}()), x \mapsto \mathrm{A}()\}$
Unifikator	$\{y \mapsto \mathrm{G}(\mathrm{A}(), \mathrm{A}()), x \mapsto \mathrm{A}()\}$

	$\mathrm{H}(\mathrm{g}(\mathrm{f}(x,u)),\mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}())),\mathrm{f}(u,\mathrm{f}(x,u))) = \mathrm{H}(\mathrm{g}(v),v,\mathrm{f}(y,z))$	$\{\}$
Dekomposition	$\mathrm{g}(\mathrm{f}(x,u)) = \mathrm{g}(v) \quad \mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}())) = v \quad \mathrm{f}(u,\mathrm{f}(x,u)) = \mathrm{f}(y,z)$	$\{\}$
Dekomposition	$\mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}())) = v \quad \mathrm{f}(u,\mathrm{f}(x,u)) = \mathrm{f}(y,z) \quad \mathrm{f}(x,u) = v$	$\{\}$
Orientierung	$\mathrm{f}(u,\mathrm{f}(x,u)) = \mathrm{f}(y,z) \quad \mathrm{f}(x,u) = v \quad v = \mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}()))$	$\{\}$
Dekomposition	$\mathrm{f}(x,u) = v \quad v = \mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}())) \quad u = y \quad \mathrm{f}(x,u) = z$	$\{\}$
Orientierung	$v = \mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}())) \quad u = y \quad \mathrm{f}(x,u) = z \quad v = \mathrm{f}(x,u)$	$\{\}$
Anwendung	$u = y \quad \mathrm{f}(x,u) = z \quad \mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}())) = \mathrm{f}(x,u)$	$\{v \mapsto \mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}()))\}$
Anwendung	$\mathrm{f}(x,y) = z \quad \mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}())) = \mathrm{f}(x,y)$	$\{u \mapsto y, v \mapsto \mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}()))\}$
Orientierung	$\mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}())) = \mathrm{f}(x,y) \quad z = \mathrm{f}(x,y)$	$\{u \mapsto y, v \mapsto \mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}()))\}$
Dekomposition	$z = \mathrm{f}(x,y) \quad \mathrm{g}(\mathrm{a}()) = x \quad \mathrm{f}(\mathrm{a}(),\mathrm{a}()) = y$	$\{u \mapsto y, v \mapsto \mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}()))\}$
Anwendung	$\mathrm{g}(\mathrm{a}()) = x \quad \mathrm{f}(\mathrm{a}(),\mathrm{a}()) = y$	$\{u \mapsto y, z \mapsto \mathrm{f}(x,y), v \mapsto \mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}()))\}$
Orientierung	$\mathrm{f}(\mathrm{a}(),\mathrm{a}()) = y \quad x = \mathrm{g}(\mathrm{a}())$	$\{u \mapsto y, z \mapsto \mathrm{f}(x,y), v \mapsto \mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}()))\}$
Orientierung	$x = \mathrm{g}(\mathrm{a}()) \quad y = \mathrm{f}(\mathrm{a}(),\mathrm{a}())$	$\{u \mapsto y, z \mapsto \mathrm{f}(x,y), v \mapsto \mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}()))\}$
Anwendung	$y = \mathrm{f}(\mathrm{a}(),\mathrm{a}())$	$\{u \mapsto y, z \mapsto \mathrm{f}(\mathrm{g}(\mathrm{a}()),y), x \mapsto \mathrm{g}(\mathrm{a}()), v \mapsto \mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}()))\}$
Anwendung	$\{u \mapsto \mathrm{f}(\mathrm{a}(),\mathrm{a}()), y \mapsto \mathrm{f}(\mathrm{a}(),\mathrm{a}()), z \mapsto \mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}())), v \mapsto \mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}())), x \mapsto \mathrm{g}(\mathrm{a}())\}$	
Unifikator	$\{u \mapsto \mathrm{f}(\mathrm{a}(),\mathrm{a}()), y \mapsto \mathrm{f}(\mathrm{a}(),\mathrm{a}()), z \mapsto \mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}())), v \mapsto \mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}())), x \mapsto \mathrm{g}(\mathrm{a}())\}$	

Resolvieren

3.3 Prolog

Anhang A

Programme

A.1 Listenhilfsfunktionen

Box.scala

```
1 package name.panitz.eliza
2 case class Box[a](var x:a){}
```

Util.scala

```
1  package name.panitz.eliza;
2  object Util{
3    def rotate[a](xs:Box[List[a]])
4     =xs.x=(xs.x.tail ::: (List(xs.x.head)))
5
6    def isPrefix[a](xs:List[a],ys:List[a]):Boolean
7     = if (xs.isEmpty) true
8       else if (ys.isEmpty) false
9       else xs.head==ys.head && isPrefix(xs.tail,ys.tail)
10
11   def unwords[a](xs:List[a]):String
12    = if (xs.isEmpty) ""
13      else if (xs.tail.isEmpty) xs.head.toString()
14      else xs.head.toString()+" "+unwords(xs.tail)
15
16   def tails[a](xs:List[a]):List[List[a]]
17    =if (xs.isEmpty) List(xs) else tails(xs.tail).::(xs)
18
19   def  words (s:String):List[String]={
20     var s1 = s.trim()
21     var i = s1.indexOf(' ')
22     if (i<0) List(s1)
23     else words(s1.substring(i)) .:: (s1.substring(0,i))
24   }
25
26  def stripPunctuation(s:String): String={
27     var result="";
28     for (val i<-Iterator.range(0,s.length())){
29       if (PUNCTUATIONS.indexOf(s.charAt(i))<0){
30         result=result+s.charAt(i);
31       }else{}
32     }
33     result
34   }
```

```
35    val PUNCTUATIONS =".,;:!"
36
37    def trace[a](s :String,x:a):a={Console.print(s+": ");Console.println(x);x}
38  }
39
```

A.2 Eliza

Data.scala

```
1   package name.panitz.eliza;
2   import scala.collection.mutable.{_}
3   class Data {
4     type Answers=Box[Words]
5     type Words=List[String]
6
7     val CONJUGATES:Map[String,String]  = new HashMap()
8     CONJUGATES.update("ME","you")
9     CONJUGATES.update("ARE","am")
10    CONJUGATES.update("WERE","was")
11    CONJUGATES.update("YOU","I")
12    CONJUGATES.update("YOUR","my")
13    CONJUGATES.update("I'VE","you've")
14    CONJUGATES.update("I'M","you're")
15    CONJUGATES.update("AM","are")
16    CONJUGATES.update("WAS","were")
17    CONJUGATES.update("I","you")
18    CONJUGATES.update("MY","your")
19    CONJUGATES.update("YOU'VE","I've")
20    CONJUGATES.update("YOU'RE","I'm")
21
22    val repeatMsgs :Answers
23      =Box(
24        List( "Why did you repeat yourself?"
25            , "Do you expect a different answer by repeating yourself?"
26            , "Come, come, elucidate your thoughts."
27            , "Please don't repeat yourself!" ))
28    val nokeyMsgs
29     =Box(
30        List( "I'm not sure I understand you fully."
31            , "What does that suggest to you?"
32            , "I see."
33            , "Can you elaborate on that?"
34            , "Say, do you have any psychological problems?") )
35    val canYou
36     =Box(
37        List( "?Don't you believe that I can"
38            , "?Perhaps you would like to be able to"
39            , "?You want me to be able to" ))
40    val canI
41      =Box(
42        List( "?Perhaps you don't want to"
43            ,"?Do you want to be able to" ))
44    val youAre
45      =Box(
46        List( "?What makes you think I am"
47            , "?Does it please you to believe I am"
48            , "?Perhaps you would like to be"
49            , "?Do you sometimes wish you were"))
50    val iDont
51      =Box(
```

```
52      List( "?Don't you really"
53          , "?Why don't you"
54          , "?Do you wish to be able to"
55          , "Does that trouble you?"))
56    val iFeel
57      =Box(
58        List( "Tell me more about such feelings."
59            , "?Do you often feel"
60            , "?Do you enjoy feeling"))
61    val whyDont
62      =Box(
63        List( "?Do you really believe I don't"
64            , ".Perhaps in good time I will"
65            , "?Do you want me to"))
66    val whyCant
67      =Box(
68        List( "?Do you think you should be able to"
69            , "?Why can't you"))
70    val areYou
71      =Box(
72        List( "?Why are you interested in whether or not I am"
73            , "?Would you prefer if I were not"
74            , "?Perhaps in your fantasies I am"))
75    val iCant
76      =Box(
77        List( "?How do you know you can't"
78            , "Have you tried?"
79            , "?Perhaps you can now"))
80    val iAm
81      =Box(
82        List( "?Did you come to me because you are"
83            , "?How long have you been"
84            , "?Do you believe it is normal to be"
85            , "?Do you enjoy being"))
86    val you
87      =Box(
88        List( "We were discussing you --not me."
89            , "?Oh,"
90            , "You're not really talking about me, are you?" ))
91    val yes
92      =Box(
93        List( "You seem quite positive."
94            , "Are you Sure?"
95            , "I see."
96            , "I understand."))
97    val no
98      =Box(
99        List( "Are you saying no just to be negative?"
100           , "You are being a bit negative."
101           , "Why not?"
102           , "Are you sure?"
103           , "Why no?"))
104   val computer
105     =Box(
106       List( "Do computers worry you?"
107           , "Are you talking about me in particular?"
108           , "Are you frightened by machines?"
109           , "Why do you mention computers?"
110           , "What do you think machines have to do with your problems?"
111           , "Don't you think computers can help people?"
112           , "What is it about machines that worries you?" ))
113   val problem
```

```
114     =Box(
115       List( "Is this really a poblem?"
116           , "Don't you see any solution to this?"
117           , "Is this your only problem?"
118           , "Has this always been a problem?"))
119   val iWant
120     =Box(
121       List( "?Why do you want"
122           , "?What would it mean to you if you got"
123           , "?Suppose you got"
124           , "?What if you never got"
125           , ".I sometimes also want"))
126   val question
127     =Box(
128       List( "Why do you ask?"
129           , "Does that question interest you?"
130           , "What answer would please you the most?"
131           , "What do you think?"
132           , "Are such questions on your mind often?"
133           , "What is it that you really want to know?"
134           , "Have you asked anyone else?"
135           , "Have you asked such questions before?"
136           , "What else comes to mind when you ask that?" ))
137   val name
138     =Box(
139       List( "Names don't interest me."
140           , "I don't care about names --please go on."))
141   val because
142     =Box(
143        List( "Is that the real reason?"
144            , "Don't any other reasons come to mind?"
145            , "Does that reason explain anything else?"
146            , "What other reasons might there be?" ))
147   val sorry
148     =Box(
149       List( "Please don't apologise!"
150           , "Apologies are not necessary."
151           , "What feelings do you have when you apologise?"
152           , "Don't be so defensive!" ))
153   val dream
154     =Box(
155       List( "What does that dream suggest to you?"
156           , "Do you dream often?"
157           , "What persons appear in your dreams?"
158           , "Are you disturbed by your dreams?" ))
159   val hello
160     =Box( List( "How do you...please state your problem."))
161
162   val maybe
163     = Box(List( "You don't seem quite certain."
164           , "Why the uncertain tone?"
165           , "Can't you be more positive?"
166           , "You aren't sure?"
167           , "Don't you know?" ))
168   val your
169     = Box(
170        List( "?Why are you concerned about my"
171           , "?What about your own"))
172   val always
173     =Box(
174       List( "Can you think of a specific example?"
175           , "When?"
```

```
176           , "What are you thinking of?"
177           , "Really, always?"))
178    val think
179      =Box(
180        List( "Do you really think so?"
181            , "?But you are not sure you"
182            , "?Do you doubt you"))
183    val alike
184      =Box(
185        List( "In what way?"
186            , "What resemblence do you see?"
187            , "What does the similarity suggest to you?"
188            , "What other connections do you see?"
189            , "Cound there really be some connection?"
190            , "How?" ))
191    val friend
192      =Box(
193        List( "Why do you bring up the topic of friends?"
194            , "Do your friends worry you?"
195            , "Do your friends pick on you?"
196            , "Are you sure you have any friends?"
197            , "Do you impose on your friends?"
198            , "Perhaps your love for friends worries you."))
199
200  val responses:List[Pair[Words,Answers]]
201   = List[Pair[Words,Answers]](
202        Pair(List("CAN","YOU"),          canYou)
203       ,Pair(List("CAN","I"),               canI)
204       ,Pair(List("YOU","ARE"),               youAre)
205       ,Pair(List("YOU'RE"),                youAre)
206       ,Pair(List("I","DON'T"),                iDont)
207       ,Pair(List("I", "DON", "NOT"),        iDont)
208       ,Pair(List("I", "FEEL"),                iFeel)
209       ,Pair(List("WHY", "DON'T", "YOU"),whyDont)
210       ,Pair(List("WHY", "CAN'T", "I"),        whyCant)
211       ,Pair(List("ARE", "YOU"),                areYou)
212       ,Pair(List("I", "CAN'T"),                iCant)
213       ,Pair(List("I", "CANNOT"),        iCant)
214       ,Pair(List("I", "AM"),               iAm)
215       ,Pair(List("I'M"),                iAm)
216       ,Pair(List("YOU"),                you)
217       ,Pair(List("YES"),                yes)
218       ,Pair(List("NO"),                        no)
219       ,Pair(List("I", "HAVE", "A", "PROBLEM"),        problem)
220       ,Pair(List("I'VE", "A", "PROBLEM"),        problem)
221       ,Pair(List("PROBLEM"),            problem)
222       ,Pair(List("COMPUTER"),                computer)
223       ,Pair(List("COMPUTERS"),                computer)
224       ,Pair(List("I", "WANT"),                iWant)
225       ,Pair(List("WHAT"),               question)
226       ,Pair(List("HOW"),                question)
227       ,Pair(List("WHO"),                question)
228       ,Pair(List("WHERE"),                question)
229       ,Pair(List("WHEN"),               question)
230       ,Pair(List("WHY"),                question)
231       ,Pair(List("NAME"),               name)
232       ,Pair(List("BECAUSE"),                because)
233       ,Pair(List("CAUSE"),               because)
234       ,Pair(List("SORRY"),               sorry)
235       ,Pair(List("DREAM"),               dream)
236       ,Pair(List("DREAMS"),                dream)
237       ,Pair(List("HI"),                    hello)
```

```
      ,Pair(List("HELLO"),               hello)
      ,Pair(List("MAYBE"),               maybe)
      ,Pair(List("YOUR"),               your)
      ,Pair(List("ALWAYS"),               always)
      ,Pair(List("THINK"),               think)
      ,Pair(List("ALIKE"),               alike)
      ,Pair(List("FRIEND"),               friend)
      ,Pair(List("FRIENDS"),               friend)
      ,Pair(List(),                       nokeyMsgs))
}
```

A.3 GUI für Schiebepuzzle

SlidePanel.scala

```
package name.panitz.ki
import javax.swing.{_}
class SlidePanel(p:InformedSlide) extends JPanel{
  var puzzle=p
  val chipSize=50

  addMouseListener(new Mousy())
  add (new BoardPanel())

  val button=new JButton("depth search")
  add(button)

  import java.awt.event.{_}
  button.addActionListener(new ActionListener(){
    override def actionPerformed(ev:ActionEvent)=
      performMoves (()=>puzzle depthSearch 10)
  })

  val button2=new JButton("breadth search")
  add(button2)
  button2.addActionListener(new ActionListener(){
    override def actionPerformed(ev:ActionEvent)=
      performMoves (puzzle.breadthFirst)
  })

  val button3=new JButton("deepening")
  add(button3)
  button3.addActionListener(new ActionListener(){
    override def actionPerformed(ev:ActionEvent)=
      performMoves (puzzle.deepening)
  })

  val button4=new JButton("A*")
  add(button4)
  button4.addActionListener(new ActionListener(){
    override def actionPerformed(ev:ActionEvent)=
      performMoves (puzzle.aStar)
  })

  def performMoves(ms:()=>List[Direction.Value])={
```

```
41      import name.panitz.swing.SwingWorker
42      val worker = new SwingWorker() {
43       override def construct():Object= {
44        for (val m<-ms()) {
45          Thread.sleep(500);puzzle=puzzle.move(m);repaint()
46        }
47        puzzle
48       }
49      }
50      worker.start()
51    }
52
53    class BoardPanel extends JPanel{
54     override def getPreferredSize():java.awt.Dimension
55      =new java.awt.Dimension(chipSize*puzzle.SIZE,chipSize*puzzle.SIZE)
56
57     override def  paintComponent(g:java.awt.Graphics)={
58      import java.awt.Color.{_}
59      super.paintComponent( g)
60      for (val r<-Iterator.range(0,puzzle.SIZE)){
61        for (val c<-Iterator.range(0,puzzle.SIZE)){
62          if (puzzle.field(c)(r)==0){
63            g setColor WHITE
64            g.fillRect(r*chipSize,c*chipSize,chipSize,chipSize)
65          }else{
66            g setColor RED
67            g.fillRect(r*chipSize,c*chipSize,chipSize,chipSize)
68            g setColor BLACK
69            g.drawString(puzzle.field(c)(r).toString
70                        ,r*chipSize+chipSize/2,c*chipSize+chipSize/2)
71          }
72        }
73      }
74      g setColor DARK_GRAY
75      for (val r<-Iterator.range(0,puzzle.SIZE+1))
76        g.fillRect(0,r*chipSize-5/2,chipSize*puzzle.SIZE,5)
77      for (val c<-Iterator.range(0,puzzle.SIZE+1))
78        g.fillRect(c*chipSize-5/2,0,5,chipSize*puzzle.SIZE)
79     }
80    }
81
82    class Mousy extends MouseAdapter{
83      override def mouseClicked(ev:MouseEvent)={
84        val x = ev.getX()/chipSize
85        val y = ev.getY()/chipSize
86
87        if (x>0&& ((puzzle.field(y)(x-1))==0))
88          puzzle=puzzle.move(Direction.Left)
89        else if (x<puzzle.SIZE-1&&puzzle.field(y)(x+1)==0) {
90          puzzle=puzzle.move(Direction.Right)
91        }
92        else if (y>0&&puzzle.field(y-1)(x)==0)
93          puzzle=puzzle.move(Direction.Up)
94        else if (y<puzzle.SIZE-1&&puzzle.field(y+1)(x)==0)
```

```
95          puzzle=puzzle.move(Direction.Down)
96        repaint()
97      }
98    }
99  }
```

SlideGUI.scala

```
1  package name.panitz.ki
2  object SlideGUI extends Application{
3    val board=new SlidePanel(new InformedSlide(3))
4    val frame=new javax.swing.JFrame()
5    frame.add(board)
6    frame.pack()
7    frame.setVisible(true)
8  }
```

A.4 Sudoku Gui

SudokuPanel.scala

```
1  package name.panitz.ki
2  import javax.swing.{_}
3  class SudokuPanel extends JPanel{
4    setLayout(new java.awt.GridLayout(9,9))
5    val grids = new Array[Array[JSpinner]](9)
6    for (val r<-Iterator.range(0,9))
7      grids(r)= new Array[JSpinner](9)
8
9    for (val r<-Iterator.range(0,9);val c<-Iterator.range(0,9)){
10     grids(r)(c)=new JSpinner(new SpinnerNumberModel(0,0,9,1))
11     add(grids(r)(c))
12   }
13 }
```

PlaySudoku.scala

```
1  package name.panitz.ki
2  object PlaySudoku extends Application {
3    import javax.swing.{_}
4    import java.awt.event.{_}
5    val f=new JFrame()
6    val pp=new JPanel()
7    val p=new SudokuPanel()
8    f add pp
9    val b=new JButton("solution")
10   val rb=new JButton("reset")
11   pp add b
12   pp add p
13   b.addActionListener(
14     new ActionListener(){
15       def actionPerformed(ev:ActionEvent)={
16         var sud=new Sudoku()
17         for (val r<-Iterator.range(0,9);val c<-Iterator.range(0,9)){
18           sud=sud.move(Tuple(r,c
```

```
19                ,p.grids(r)(c).getValue.asInstanceOf[java.lang.Integer].intValue))
20        }
21        val solution=sud.depthSearch
22        for (val s<-solution) sud=sud move s
23        for (val r<-Iterator.range(0,9);val c<-Iterator.range(0,9)){
24          p.grids(r)(c).setValue(new java.lang.Integer(sud.field(r)(c)))
25        }
26        f.repaint()}})
27  rb.addActionListener(
28    new ActionListener(){
29      def actionPerformed(ev:ActionEvent)={
30        for (val r<-Iterator.range(0,9);val c<-Iterator.range(0,9)){
31          p.grids(r)(c).setValue(new java.lang.Integer(0))
32        }
33        f.repaint()}})
34  pp add rb
35  f.pack
36  f setVisible true
37 }
38
```

A.5 Swing Worker

SwingWorker.java

```
1  package name.panitz.swing;
2  import javax.swing.SwingUtilities;
3
4  /**
5   * This is the 3rd version of SwingWorker (also known as
6   * SwingWorker 3), an abstract class that you subclass to
7   * perform GUI-related work in a dedicated thread.  For
8   * instructions on using this class, see:
9   *
10  * http://java.sun.com/docs/books/tutorial/uiswing/misc/threads.html
11  *
12  * Note that the API changed slightly in the 3rd version:
13  * You must now invoke start() on the SwingWorker after
14  * creating it.
15  */
16 public abstract class SwingWorker {
17     private Object value;  // see getValue(), setValue()
18     private Thread thread;
19
20     /**
21      * Class to maintain reference to current worker thread
22      * under separate synchronization control.
23      */
24     private static class ThreadVar {
25         private Thread thread;
26         ThreadVar(Thread t) { thread = t; }
27         synchronized Thread get() { return thread; }
28         synchronized void clear() { thread = null; }
29     }
```

```
30
31      private ThreadVar threadVar;
32
33      /**
34       * Get the value produced by the worker thread, or null if it
35       * hasn't been constructed yet.
36       */
37      protected synchronized Object getValue() {
38          return value;
39      }
40
41      /**
42       * Set the value produced by worker thread
43       */
44      private synchronized void setValue(Object x) {
45          value = x;
46      }
47
48      /**
49       * Compute the value to be returned by the <code>get</code> method.
50       */
51      public abstract Object construct();
52
53      /**
54       * Called on the event dispatching thread (not on the worker thread)
55       * after the <code>construct</code> method has returned.
56       */
57      public void finished() {
58      }
59
60      /**
61       * A new method that interrupts the worker thread.  Call this method
62       * to force the worker to stop what it's doing.
63       */
64      public void interrupt() {
65          Thread t = threadVar.get();
66          if (t != null) {
67              t.interrupt();
68          }
69          threadVar.clear();
70      }
71
72      /**
73       * Return the value created by the <code>construct</code> method.
74       * Returns null if either the constructing thread or the current
75       * thread was interrupted before a value was produced.
76       *
77       * @return the value created by the <code>construct</code> method
78       */
79      public Object get() {
80          while (true) {
81              Thread t = threadVar.get();
82              if (t == null) {
83                  return getValue();
```

```
84              }
85              try {
86                  t.join();
87              }
88              catch (InterruptedException e) {
89                  Thread.currentThread().interrupt(); // propagate
90                  return null;
91              }
92          }
93      }
94
95
96      /**
97       * Start a thread that will call the <code>construct</code> method
98       * and then exit.
99       */
100     public SwingWorker() {
101         final Runnable doFinished = new Runnable() {
102            public void run() { finished(); }
103         };
104
105         Runnable doConstruct = new Runnable() {
106             public void run() {
107                 try {
108                     setValue(construct());
109                 }
110                 finally {
111                     threadVar.clear();
112                 }
113
114                 SwingUtilities.invokeLater(doFinished);
115             }
116         };
117
118         Thread t = new Thread(doConstruct);
119         threadVar = new ThreadVar(t);
120     }
121
122     /**
123      * Start the worker thread.
124      */
125     public void start() {
126         Thread t = threadVar.get();
127         if (t != null) {
128             t.start();
129         }
130     }
131 }
```

Klassenverzeichnis

Abbildungsverzeichnis

Literaturverzeichnis

[Kur93] Raymond Kurzweil. *KI Das Zeitalter der Künstlichen Intelligenz*. Carl Hanser verlag, 1993.

[Men91] Gerhard Mensching. *Die abschaltbare Frau*. Haffmans Verlag, Zürich, 1991.

[Pou91] William Poundstone. *Labyrinths of Reason*. Penguin Books, 1991.

[RN95] S.J. Russel and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.

[Sch89] Uwe Schöning. *Logik für Informatiker*. Number 56 in Reihe Informatik. Wissenschaftsverlag, Mannheim, 1989.

[Spe86] Volker Sperschneider. Logik. Handschriftliches Vorlesungskript der Universität Karlsruhe, 1986.

[SS06] Manfred Schmidt-Schauß. Einführung in die künstliche intelligenz. Skript zur Vorlesung, 2006. `http://www.ki.informatik.uni-frankfurt.de/lehre/SS2006/KI/main.html`.

[Win92] P. Winston. *Artificial Intelligence*. Addison Wesley, 1992.