DOI: 10.20378/irb-90968

DISKI 70

DISKI

Dissertationen zur Künstlichen Intelligenz

Mit Unterstützung des Fachbereichs 1 „Künstliche Intelligenz“ der Gesellschaft für Informatik e.V. herausgegeben von

G. Barth, Ulm
W. Bibel, Darmstadt
W. Brauer, München
H. Bunke, Bern
Th. Christaller, Sankt Augustin
W. Coy, Bremen
P. Deussen, Karlsruhe
R. Dillmann, Karlsruhe
L. Dreschler-Fischer, Hamburg
Chr. Freksa, Hamburg
U. Furbach, Koblenz
U. Geske, Berlin
G. Görz, Erlangen
G. Gottlob, Wien
Chr. Habel, Hamburg
W. von Hahn, Hamburg
W. Hoeppner, Duisburg
K. P. Jantke, Leipzig
M. Jarke, Aachen
A. Kobsa, Konstanz
W. Kropatsch, Wien
E. Lehmann, Stuttgart
C. Möbus, Oldenburg
K. Morik, Dortmund
H.-H. Nagel, Karlsruhe
B. Neumann, Hamburg
H. Niemann, Erlangen
Chr. Posthoff, Chemnitz
F. Puppe, Würzburg
B. Radig, München
M. M. Richter, Kaiserslautern
H. Ritter, Bielefeld
C. Rollinger, Osnabrück
G. Sagerer, Bielefeld
M. Schmidt-Schauss, Frankfurt/M
J. H. Siekmann, Saarbrücken
G. Smolka, Saarbrücken
H. S. Stiehl, Hamburg
H. Stoyan, Erlangen
G. Strube, Freiburg
R. Studer, Karlsruhe
R. Trappl, Wien
G. Veenker, Bonn
I. Wachsmuth, Bielefeld
W. Wahlster, Saarbrücken
Chr. Walther, Darmstadt
R. Wiehagen, Kaiserslautern

Erwerb rekursiver Programmiertechniken als Induktion von Konzepten und Regeln

Ein kognitionswissenschaftlicher Zugang zum Erwerb kognitiver Fertigkeiten

von
Ute Schmid

infix

Ute Schmid
Institut für Angewandte Informatik
FG Methoden der künstlichen Intelligenz
Franklinstr. 28/29
10587 Berlin

Vom Fachbereich 13 – Informatik –
der Technischen Universität Berlin
zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften
genehmigte Dissertation

Promotionsausschuß:

Vorsitzender: Prof. Dr. Fritz Wysotzki
Berichter: Prof. Dr. Bernd Mahr
Berichter: Prof. Dr. Klaus Eyferth

Tag der wissenschaftlichen Aussprache: 3. Juni 1994

Die Deutsche Bibliothek – CIP-Einheitsaufnahme

Schmid, Ute:
Erwerb rekursiver Programmiertechniken als Induktion von Konzepten und Regeln / Ute Schmid. – Sankt Augustin : Infix, 1994
(*Dissertationen zur künstlichen Intelligenz* ; 70)
Zugl.: Berlin, Techn. Univ., Diss., 1994
ISBN 3-929037-70-X
NE: GT

© 1994 Dr. Ekkehard Hundt, „*infix*", *Ringstr.* 32, *53757 Sankt Augustin*

Das Werk ist in allen seinen Teilen urheberrechtlich geschützt. Jede Verwertung ohne ausdrückliche Zustimmung des *Verlages ist unzulässig. Das gilt insbesondere für* Vervielfältigungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung in und Verarbeitung durch elektronische Systeme.

Reproduziert von einer Druckvorlage des Autors
Umschlaggestaltung: Art und Media, Bonn
Druck und Verarbeitung: Hundt Druck GmbH, Köln
Printed in Germany

ISSN 0941-5769
ISBN 3-929037-70-X

Zum Geleit

Rekursion ist ein Thema, das nicht nur in der Informatikausbildung, sondern auch in der Forschung (Berechenbarkeit, Programmiersprachensemantik, Compileroptimierung und zuletzt Typtheorie) eine herausgehobene Rolle spielt. Der Grund für diese Prominenz ist die Tatsache, daß Rekursion als Verarbeitungsstruktur sehr knappe, elegante und formal vergleichsweise leicht verifizierbare operationale Repräsentationen von Funktionen oder anderen Objekten erlaubt. So hat sich in der Semantik rekursiver Programme und Schemata umfangreiches Wissen angesammelt und in der Pragmatik ein großer Erfahrungsschatz. Wenig ist dagegen bekannt, wie dieses Wissen über Rekursion repräsentiert und organisiert werden kann, um rekursive Formalisierungen zu unterstützen. Für automatische Verifikation, Optimierung und Programmierung, sowie für intelligente Tutorsysteme ist dies jedoch eine Frage von grundlegender Bedeutung.

Ein natürlicher Zugang zu dieser Frage scheint mir das Lernen zu sein, d.h. eine Antwort auf die allgemeine Frage, wie denn die Fähigkeit zu rekursiven Formalisierungen erworben wird. Daß dieser Erwerb nicht einfach ist und deshalb die Frage nach einer möglichen Repräsentation und Organisation des Wissens um Rekursion nicht trivial, ist aus der Programmierausbildung bekannt, wo Rekursion zu den schwierigen Themen gehört. Viele Studenten erwerben niemals die Fähigkeiten zu rekursiven Formalisierungen.

Frau Schmid beschäftigt sich im vorliegenden Buch mit dieser Frage des Erwerbs. Dabei betrachtet sie ein durch Beispiele gesteuertes auf Analogie basierendes induktives Lernen als Prozeß und analysiert in Experimenten die Wirkung des "learning by example". Die Ergebnisse ihrer empirischen Untersuchungen sind interessant. Sie bestätigen, und präzisieren die Erfahrungen in der Programmierausbildung: Rekursion kann durch Beispiele ebensogut gelernt werden, wie durch direkte Vermittlung wenn auch etwas langsamer. Dabei spielt die richtig gewählte Nähe des Beispiels zu der zu lösenden Aufgabe eine wichtige Rolle. Bei wachsendem Wissen über Rekursion werden Beispiele weniger bedeutsam.

Diese Experimente sind für die oben formulierte allgemeine Frage jedoch nicht direkt nutzbar. Vielmehr ist es das Modell des Erwerbs und damit die Repräsentation des zu erlernenden Wissens. Da sich die Experimente in ihrem Aufbau auf dieses Modell stützen, wird durch die Untersuchungen auch nachgewiesen, daß damit eine taugliche Antwort auf diese allgemeine Frage gegeben ist: Es gelingt Frau Schmid in einem differenzierten Repräsentationsformat, das eine Verknüpfung aus Schemata und Regeln darstellt, induktives Lernen rekursiver Formalisierungsfähigkeit zu beschreiben.

Prof. Dr. Bernd Mahr, 14. November 1994

Danksagung

Viele Personen haben mich bei dieser Arbeit auf verschiedene Weise unterstützt und ich hoffe, daß ich an dieser Stelle niemanden vergessen habe, dem ich Dank schulde.

Wichtige Diskussionspartner und Kritiker waren mir Guido Dunker und Barbara Kaup. Technische Unterstützung erhielt ich von Martin Kindsmüller. Die empirischen Untersuchungen konnten nur durch die engagierte Mitarbeit zahlreicher Studenten erfolgreich durchgeführt werden. Weitere direkte oder indirekte Unterstützung erhielt ich von Steffen Barthel, Jens Gräbener, Jürgen Schönig, Daniela Ulber, Carla Umbach, Prof. Dr. Arnold Upmeyer, Chris Werner und Prof. Dr. Fritz Wysotzki.

Ganz besonders möchte ich mich bei Dr. Uwe Konerding bedanken, der mir seit vielen Jahren ein wichtiger Gesprächspartner und Kritiker meiner Arbeit ist.

Prof. Dr. Klaus Eyferth und Prof. Dr. Bernd Mahr gaben mir nicht nur wertvolle Anregungen zu dieser Arbeit. Ihnen sei ganz herzlich dafür gedankt, daß sie mir in vielen Bereichen stets ermutigend und unterstützend zur Seite stehen.

Inhaltsverzeichnis

1 Einleitung

Die Induktion neuen Wissens ist wohl das grundlegendste menschliche Lernprinzip: Die meisten sozialen Verhaltensweisen werden nicht über Normen vermittelt, sondern aufgrund von Beobachtung des Verhaltens von Mitmenschen sowie der Reaktion der Umgebung auf eigene Verhaltensweisen erworben (Bandura & Walters 1963). Die Einteilung der Welt in kategoriale Begriffe wird auf Grundlage der Merkmale einzelner wahrgenommener Objekte vorgenommen, selten werden Kategorien explizit über Objektmerkmale definiert, was meist auch nicht vollständig möglich ist (z.B. Rosch 1978; Smith, Shoben & Rips 1974). So sagt man einem Kind etwa bei verschiedenen Exemplaren von Hunden "Das ist ein Hund" aber nicht "Hunde sind Tiere, spezieller Säugetiere, die Fell haben, vier Beine haben und bellen". Die Fähigkeit, mathematische Probleme zu lösen, wird durch Studium bereits gelöster Aufgaben, sowie durch eigene Lösungsversuche entwickelt, aber nicht allein dadurch, daß allgemeine Regeln für den Lösungsprozeß vorgegeben werden (Anderson, Farrell & Sauers 1984).

Allen genannten Problembereichen ist gemeinsam, daß neues Wissen aus Einzelfällen, also aus **Beispielen**, inferiert wird. Beispiele sind dadurch gekennzeichnet, daß jedes Einzelbeispiel wesentliche und unwesentliche Merkmale für den zu lernenden Bereich enthält (vgl. Holland, Holyoak, Nisbett & Thagard 1986, S. 19 ff.). Die Inferenzleistung des lernenden Systems besteht darin, aus verschiedenen Beispielen diejenigen Merkmale zu isolieren, die für erfolgreiche Aufgabenlösungen relevant sind. So ist etwa die Farbe des Felles kein relevantes Unterscheidungsmerkmal um ein Tier als Hund statt als Katze zu klassifizieren, sehr wohl aber, daß das Tier bellen kann. Für das Auflösen einer Gleichung nach einer Unbekannten ist es unwesentlich, ob die Unbekannte mit x oder y bezeichnet ist, sehr wohl aber, ob die Unbekannte mit anderen Werten additiv oder multiplikativ verknüpft ist.

Die vorliegende Arbeit befaßt sich mit einem speziellen Bereich des induktiven Lernens, nämlich mit dem Erwerb von kognitiven Problemlösefertigkeiten, wie zum Beispiel dem Lösen mathematischer Gleichungen. Im Rahmen dieser Arbeit wird der Erwerb von rekursiven Techniken beim funktionalen Programmieren als Problembereich zugrundegelegt (z.B. Field & Harrison 1988).

Ziel der Arbeit ist es, einige Aspekte des induktiven Erwerbs von Problemlösefertigkeiten in einem Modell zu integrieren. Zentrale Annahmen des Modells werden empirisch überprüft. Dabei wird von der Annahme ausgegangen, daß beim Erwerb von Problemlösetechniken sowohl Regeln zur Problemklassifikation als auch Regeln zur Problemlösung inferiert werden. Zudem wird angenommen, daß die Inferenzen während der Auseinandersetzung mit konkreten Problemstellungen gebildet werden (z.B. Anderson 1982). Jede konkrete Problemstellung kann als ein Beispiel für den betrachteten Gegenstandsbereich verstanden werden.

Die erste Annahme ist ein Versuch, eine Brücke zwischen Ansätzen zum Klassifikationslernen und zum Erwerb von Problemlösefertigkeiten zu schlagen. Die meisten Arbeiten zur Induktion befassen sich mit dem Erwerb von Konzepten (Vosniadou & Ortony 1989; Cohen & Feigenbaum 1982, Kap. 14). Dabei wird jedoch häufig vernachlässigt, daß Objekte nicht nach beliebigen, sondern nach funktional distinkten - also problemrelevanten - Merkmalen klassifiziert werden: Wir unterscheiden traurige und fröhlichen Menschen oder Hunde und Katzen, weil in jedem Fall andere Verhaltensweisen sinnvoll sind (Klix 1980). Die Arbeiten zum Erwerb von Problemlösefertigkeiten gehen dagegen meist von bereits klassifizierten Problemen aus und behandeln nur noch das Problem der Induktion von Problemlöseregeln (Anderson & Thompson 1989). In dieser Arbeit wird jedoch davon ausgegangen, daß Klassifikations- und Problemlöseregeln gemeinsam, in einem Prozeß gegenseitiger Anpassung, erworben werden. Der Erwerb von Expertise in einem Problembereich umfaßt den Aufbau einer strukturierten Repräsentation von Problemklassen zusammen mit den klassenspezifischen Lösungsansätzen.

Die Arbeit ist folgendermaßen aufgebaut:

Kapitel 2 gibt einen kurzen Überblick über philosophische und logische Probleme der Induktion. Es wird argumentiert, daß eine induktive Logik zwar im Gegensatz zu deduktiven Logiken nicht zu sicheren Konklusionen führt, daß aber induktives Schließen mit alternativen Methoden, wie etwa regelbasierten Ansätzen aus dem Bereich des maschinellen Lernens, durchaus angemessen modelliert werden kann.

In **Kapitel 3** werden verschiedene Konzepte der Wissensrepräsentation im Zusammenhang mit induktivem Wissenserwerb diskutiert. Am Ende des Kapitels wird begründet, warum es aufgrund der unterschiedlichen Charakteristika der verschiedenen Ansätze sinnvoll ist, diese zur Modellierung von induktivem Wissenserwerb zu kombinieren. Genauer wird vorgeschlagen, Schemahierarchien und Produktionsregeln zu einer strukturierten Repräsentation des Wissens über einen Problembereich zu integrieren. Aktuelle Problemlösungsprozesse zusammen mit Prozessen des induktiven Wissenserwerbs werden als mentale Modelle repräsentiert. Ein mentales Modell soll dabei einem (partiell) instantiierten Schema entsprechen, das über mehrere Problemlösezeitpunkte solange verfeinert wird, bis das Schema der Problemlösung entspricht.

Kapitel 4 stellt zentrale Ansätze zum Erwerb von Problemlöseregeln und zum Klassifikationslernen vor. Dabei werden insbesondere eine Erweiterung von Anderson's ACT*-Theorie (Anderson & Thompson 1989) und ein Ansatz von Carbonell (1983; 1986) diskutiert, die beide als Modell des induktiven Erwerbs von Programmiertechniken gedacht sind. Am Ende des Kapitels wird ein Ansatz von Holyoak und Thagard (1986) vorgestellt, der primär als Modell des induktiven Konzepterwerbs gedacht ist, aber geeignet scheint, auf den Erwerb von Problemlösetechniken erweitert zu werden. Dieser Ansatz kombiniert Schemahierarchien

und Regeln in einer Repräsentation und betont die Relevanz von funktionalen Merkmalen für den Konzepterwerb.

Im fünften Kapitel wird der in dieser Arbeit verwendete Lerngegenstand genauer dargestellt. Dabei werden zunächst das Prinzip rekursiver Definitionen sowie die Charakteristika funktionaler Programmiersprachen diskutiert. Es wird eine einfache selbstentwickelte funktionale Sprache vorgestellt. Darauf aufbauend werden Expertise-Modelle zum rekursiven Programmieren diskutiert. Eines der Modelle (Vorberg & Goebel 1991) ist als Modell eines menschlichen Experten gedacht, das zweite Modell (Manna & Waldinger 1975) stammt aus dem Bereich automatisches Programmieren. Beide Modelle geben Hinweise, welche Wissensstrukturen notwendig sind, um ausgehend von nicht-rekursiv formulierten Aufgabenstellungen rekursive Programme zu entwickeln. Dabei betont der Ansatz von Vorberg & Goebel das Wissen über die Struktur rekursiver Programme, die als Hierarchie von Rekursionsschemata repräsentiert sind. Der Ansatz von Manna und Waldinger betont dagegen Wissen über Programmierkonzepte, deren Anwendungsbedingungen als Transformationsregeln repräsentiert werden.

In **Kapitel 6** wird ein Modell zum Erwerb rekursiver Programmiertechniken vorgeschlagen, das auf den bereits dargestellten Ansätzen aufbaut. Als Zielvorstellung für den Erwerb rekursiver Programmiertechniken wird eine Hierarchie von Rekursionsschemata vorgeschlagen, die neben Leerstellen für die Komponenten rekursiver
Funktionen zusätzlich Leerstellen für Problemlöseregeln enthalten. Die Repräsentation der Schemata lehnt sich einerseits an das Konzept der Programmschemata (Engelfriet 1974) und andererseits an kognitionspsychologische Schemakonzeptionen an. Die Anwendung des so repräsentierten Wissens zur Lösung einer konkreten Programmieraufgabe wird als schrittweise Transformation eines instantiierten Schemas modelliert. Schließlich wird gezeigt, wie Anfänger bei der Lösung von Programmieraufgaben mit Beispielen Wissen über rekursive Programmiertechniken erwerben, in dem die zur Schemainstantiierung notwendigen Konzepte und Regeln durch einen Vergleich von Aufgabe und Beispiel inferiert werden. Aus dem Modell lassen sich empirisch prüfbare Annahmen ableiten. Die empirische Prüfung von Annahmen über den induktiven Erwerb von Programmierfertigkeiten wird in den folgenden beiden Kapitel dargestellt.

Kapitel 7 diskutiert empirische Zugänge zur Erfassung von Lernprozessen beim Programmieren. Es wird argumentiert, daß Experimente, die auf nur einem Meßzeitpunkt basieren, nicht geeignet sind komplexere Lernprozesse zu untersuchen. Quasiexperimentelle Zugänge über Experten-Novizen-Vergleiche geben ebenfalls keinen Aufschluß über den Prozeß des Wissenserwerbs. Einzelfallstudien liefern zwar Erkenntnisse über Lernprozesse, sind aber aufgrund der geringen Stichprobenzahl sowie der fehlenden experimentellen Bedingungsvariation wenig zuverlässig. Am Ende dieses Kapitels wird die selbstentwickelte

Lernumgebung LEAR vorgestellt, die es ermöglicht, den Verlauf des Erwerbs rekursiver Programmierkenntnisse über mehrere Zeitpunkte kontrolliert zu erfassen. Diese Lernumgebung basiert auf einem Interpreter für die entwickelte einfache funktionale Sprache. Sie ermöglicht, verschiedene Lernhilfen (z.B. Beispiele) systematisch darzubieten und das Programmierverhalten der Lernenden zu erfassen.

Im **achten Kapitel** werden zwei experimentelle Untersuchungen vorgestellt, in denen die Lernumgebung LEAR eingesetzt wurde. In der ersten Untersuchung wurde die Annahme geprüft, daß bei Anfängern aufwendige Inferenzprozesse notwendig sind, um Beispielfunktionen auf aktuelle Aufgaben zu übertragen, daß diese Inferenzprozesse jedoch zum Aufbau von allgemeineren Wissensstrukturen führen, die einen Transfer des Gelernten ermöglichen. Zu diesem Zweck wurde die Vorgabe von Beispielen mit der direkten Vorgabe der Problemlöseregeln in Form von erklärenden Texten kontrastiert. In der zweiten Untersuchung wurde betrachtet, wie sich die Ähnlichkeit der Beispiele zu den Programmieraufgaben auf die inferierten Wissensstrukturen auswirkt. Dabei wurde von der Annahme ausgegangen, daß die Beispielähnlichkeit und die Lösungswahrscheinlichkeit der Aufgaben in monoton steigendem Zusammenhang stehen. Wissenserwerb und Beispielähnlichkeit sollen dagegen in einem umgekehrt u-förmigen Zusammenhang stehen. Zu ähnliche Beispiele führen zum Aufbau sehr spezifischer Wissensstrukturen, die nur auf sehr enge Problembereiche anwendbar sind, zu unähnliche Beispiele verhindern, daß strukturelle Charakteristika des Problembereichs erschlossen werden können. Also sollten Beispiele, die sich in ihrer strukturellen Ähnlichkeit zur Aufgabe zwischen diesen Extremen befinden, den Wissenserwerb am stärksten begünstigen.

Am Ende der Arbeit (**Kapitel 9**) wird das in Kapitel 6 vorgestellte Modell zusammenfassend im Hinblick auf die empirischen Ergebnisse diskutiert. In einem Ausblick wird argumentiert, daß ein empirisch prüfbares kognitionspsychologisches Modell des induktiven Erwerbs von Programmiertechniken nicht nur ein Beitrag zur psychologischen Theoriebildung ist, sondern Implikationen für die Programmierdidaktik, für die Architektur intelligenter Tutorensysteme sowie für Ansätze des maschinellen Lernens und der automatischen Programmierung besitzt.

2 Das Problem der induktiven Logik und Alternativen in der Kognitionswissenschaft

Modellvorstellungen über induktives Schließen finden sich seit der Antike und in den verschiedensten Wissenschaften. Dabei sind die Ansätze danach zu unterscheiden, ob ihre Zielsetzung ist, *induktives Schließen normativ-präskriptiv* oder psychologisch-deskriptiv zu modellieren. Die vorliegende Arbeit verfolgt die zweitgenannte Zielsetzung. In diesem Kapitel wird zunächst das *normative Konzept* der Induktion eingeführt und mit der Deduktion kontrastiert. Beide Konzepte werden dann bezüglich ihrer Angemessenheit zur psychologischen *Modellierung von Schlußfolgerungsprozessen* bewertet. Im Anschluß werden einige philosophische Zugänge zur Induktion kurz skizziert. Es wird argumentiert, daß die Paradoxien, die sich bei *normativ-formalen Modellen* induktiven Schließens ergeben, möglicherweise durch einen kognitionspsychologischen Zugang umgangen werden können. Damit würde der Entwurf eines *psychologischen Modells* gleichzeitig als alternative Konzeption eines normativen Modells induktiven Schließens aufgefaßt werden.

Auch wenn zu Ende dieses Kapitels argumentiert wird, daß nicht auf der formalen Logik basierende Zugänge angemessener zur Modellierung des induktiven Schließen sind, sind auch die Ansätze aus künstlicher Intelligenz und kognitiver Psychologie keineswegs befriedigend. Um mit Holland, Holyoak, Nisbett und Thagard (1986, S. 1) zu sprechen "*Induction, which has been called the scandal of philosophy, has become the scandal of psychology and artificial intelligence as well*".

2.1 Induktion versus Deduktion

Begriffsklärungen

Induktion wird als Generalisierung von Einzelbeobachtungen zu einer Allaussage verstanden (enumerative Induktion, vgl. Essler 1980). **Inferenz** oder Inferenzschluß bezeichnet den Prozeß oder das Ergebnis einer Schlußfolgerung und wird sowohl auf induktive als auch auf deduktive Schlüsse angewendet. Ein **Analogieschluß** stellt einen Spezialfall der Induktion dar, bei dem die Struktur oder der Lösungsweg eines bereits bekannten Problems auf ein neues Problem übertragen wird. Im Rahmen dieser Arbeit wird vor allem auf sogenannte ***within-domain* Analogien** bezug genommen. *Within-domain* Analogien bleiben auf einen Problembereich beschränkt, so wäre zum Beispiel die Übertragung einer bekannten Lösung eines Gleichungssystems mit einer Unbekannten auf ein neues derartiges Problem eine *within-domain* Analogie. Die Verwendung eines Modells des Sonnensystems zur Modellierung des Aufbaus von Atomen wäre dagegen eine *between-domain* Analogie (z.B. Gentner 1983; Johnson-Laird 1988).

Deduktive Schlüsse

In den Formalwissenschaften wurde der deduktive Schluß lange und gründlich untersucht und bildet spätestens seit dem Beginn der mathematischen Logik die grundlegende Beweismethode (Ebbinghaus, Flum & Thomas 1986).

Deduktive Schlüsse basieren auf als wahr vorausgesetzen Hypothesen (H) und Daten (D). Ist H eine Hypothese der Form "*für alle x gilt: wenn A(x) dann folgt auch B(x)*" ($\forall x(A(x) \rightarrow B(x))$ und D eine Menge von Daten $\{d_1..d_n\}$, wie $A(c)$, so hat der deduktive Schluß die Form: $H\ U\ D \vdash d_{n+1}$. Für die logische Ableitung $\vdash$ existieren eine Vielzahl syntaktischer Regeln (z.B. Hilbert & Ackermann 1959). Als Beispiel wird hier die aristotelische Schlußform des *modus ponens* angeführt:

H: $\forall x\ (A(x) \rightarrow B(x))$	Alle Menschen sind sterblich.
D: $A(c)$	Sokrates ist ein Mensch.
---	---
$\vdash B(c)$	Also ist Sokrates sterblich.

Auf dieselbe Weise könnte man mit diesem Schlußschema zu der Konklusion kommen, daß Sokrates gern in Flugzeugen fliegt, wenn man als Hypothese "Alle Menschen fliegen gern in Flugzeugen" verwendet (vgl. Russell 1939, S. 149). Dieser logisch korrekte Schluß wird aber sicher nicht als semantisch adäquat akzeptiert, da wir erstens wissen, daß nicht alle Menschen gerne fliegen und zweitens, daß es zu sokratischen Zeiten keine Flugzeuge gab. Beim alltäglichen Schließen werden Hypothesen also bezüglich ihres Wahrheitsgehaltes bewertet und die Schlüsse damit als unterschiedlich plausibel eingeschätzt.

Induktive Schlüsse

Induktive Schlüsse generalisieren von einer Menge von Einzelbeobachtungen Dn als Prämissen auf eine hypothetische Allaussage H: $D_n < H$. Da es sich hier um keine strenge Ableitung handelt, wird anstelle von $\vdash$ das Symbol $<$ notiert:

D: $d1 = (A(c1) \rightarrow B(c1))$	Im Urlaub 1984 hat es geregnet,
$d2 = (A(c2) \rightarrow B(c2))$	im Urlaub 1985 hat es geregnet,
...	...
$dn = (A(cn) \rightarrow B(cn))$	im Urlaub 1993 hat es geregnet.
---	---
$< H: \forall x(Ax) \rightarrow B(x))$	Immer, wenn ich Urlaub habe, regnet es.

Den Einzelbeobachtungen kann noch eine Theorie T als Hintergrundwissen hinzugefügt werden: $T \cup D_n < H$.

T: $C(x) \rightarrow A(x)$	Ich habe immer im August Urlaub.
D: $d1 = (C(c1) \rightarrow B(c1))$	Im August 1984 hat es geregnet,
$d2 = (C(c2) \rightarrow B(c2))$	im August 1985 hat es geregnet,
...	...
$dn = (C(cn) \rightarrow B(cn))$	im August 1993 hat es geregnet.
---	---
< H: $\forall x(A(x) \rightarrow B(x))$	Immer, wenn ich Urlaub habe, *regnet* es.

Während also deduktive Schlüsse auf als wahr vorausgesetzen Prämissen basieren, werden bei induktiven Schlüssen Allaussagen auf der Grundlage von Einzeldaten abgeleitet. Damit sind induktive Schlüsse stets nicht-monoton. Nicht-monoton hergeleitetes Wissen ist vorläufig und kann durch neu erworbenes Wissen revidiert werden.

Eine häufig benutzte Beweistechnik der Mathematik ist die vollständige Induktion. Diese Schlußform hat auf den ersten Blick viele Gemeinsamkeiten mit dem gerade skizzierten Induktionsprinzip: Die vollständige Induktion basiert auf dem fünften Axiom von Peano (z.B. Wechler 1992, S. 253; siehe Anhang A.1). Mithilfe der vollständigen Induktion wird die Gültigkeit eines Satzes A für alle natürlichen Zahlen n bewiesen (logisch abgeleitet). Dabei besitzt der Schluß von einer beliebigen aber festen natürlichen Zahl k auf die Zahl $k+1$ die Form der oben definierten Hypothese; dem Datum D entspricht die Verankerung der Induktion für $n=0$ bzw. $n=1$: $A(0)$ und $\forall k(A(k) \rightarrow A(k+1)) \vdash Vn(An)$. Die vollständige Induktion basiert auf der linearen Ordnungsstruktur der natürlichen Zahlen, wie sie in den Peanoaxiomen (1 bis 4; Wechler 1992 S.253; siehe Anhang A.1) definiert ist. Da ausgehend von der ersten natürlichen Zahl jede Zahl genau einen direkten Nachfolger besitzt, kann eine beliebige Zahl k aus dieser Sequenz herausgegriffen werden, für die ein Schluß auf den Nachfolger $k+1$ gezeigt wird.

Will man jedoch Allaussagen in Bereichen generieren, für die keine Ordnungsstruktur angebbar ist, so ist das mathematische Induktionsprinzip nicht durchführbar. So läßt sich zum Beispiel für alle Menschen, die einmal gelebt haben, jetzt leben und leben werden keine eindeutige Ordnungsstruktur angeben. Die Allaussage "Alle Menschen sind sterblich" kann also nicht mit diesem Prinzip bewiesen werden.

Induktion und Erkenntnis

Allgemeiner kann man feststellen, daß induktives Schließen im Alltag im Gegensatz zur Deduktion und zur vollständigen Induktion zwar nie zu vollständig sicheren Konklusionen führen kann, daß aber andererseits deduktiv abgeleitetes Wissen kaum neue Erkenntnis bringt. So ist etwa der Schluß "Ein Hund hat Fell" aus dem Satz "Ein Hund kann bellen und ein Hund hat Fell" trivial (vgl. z.B. Johnson-Laird, Byrne & Schaeken 1992). Zudem gründen deduktive Schlüsse auf als wahr vorausgesetzten Allaussagen, wie etwa "Alle Menschen sind sterblich". Um jedoch zu einer Allaussage zu gelangen, muß diese aus der Beobachtung möglichst vieler Einzelfälle inferiert werden. Somit wird der syllogistische Schluß "Alle Menschen sind sterblich. Sokrates ist ein Mensch. Also ist Sokrates sterblich." sinnlos, da zum Aufstellen der ersten Prämisse bereits beobachtet worden sein muß, daß alle zur Menge "Menschen" gehörenden Elemente, also auch Sokrates, sterblich sind. Fodor (1980, S. 148 f.) argumentiert in dieselbe Richtung: "*Yet I say again that learning must be inductive (non-demonstrative) inference; there is nothing else for it to be. And the only model of inductive inference that has ever been proposed anywhere by anyone is hypothesis formation and confirmation.*"

Abschließend kann festgestellt werden, daß außerhalb der normativen (mathematischen) Wissenschaften meist nur das Prinzip der "unvollständigen Induktion" einsetzbar ist, deren Ergebnisse nur mit einer bestimmten Sicherheit als wahr angenommen werden können.

Aber nicht nur im Alltag, auch beim wissenschaftlichen Arbeiten, werden neue Gesetze und Sichtweisen induktiv gewonnen (vgl. Hollan, Holyoak, Nisbett & Thagard 1986, Kap. 11). Dies gilt auch für die Mathematik. Zwar kann die Gültigkeit vieler Theoreme mittels vorhandener Beweisprinzipien deduktiv nachgewiesen werden, das Finden von Beisweisprinzipien für offene mathematische Probleme ist jedoch ebenfalls kein deduktiver, sondern ein induktiver Prozeß (Lakulos 1976, Polya 1956): "*Analogy pervades all our thinking, our everyday speech and our trivial conclusions as well as artistic ways of expression and the highest scientific achievements*" (Polya 1957).

2.2 Induktion als Problem der Philosophie

Induktives Schließen bei Aristoteles

Aristoteles[1] hat in den Analytiken neben der Begründung der deduktiven Logik mit dem Konzept des syllogistischen Schließens (vgl. z.B. Hilbert & Ackermann 1959) auch das Konzept des induktiven Schlusses als Generalisierung von Einzelbeobachtungen zu Allaussagen eingeführt. Dabei unterscheidet er zwischen Schlüssen, die auf vollständig aufzählbaren Einzelfällen basieren (Epagogen) und solchen, bei denen der Konklusion nicht alle Fälle zugrunde liegen (paradigmatische Schlüsse). Eng verbunden mit dem aristotelischen Konzept der Induktion, sind seine Definitionen von Begriff und Kategorie. Einen Begriff definiert Aristoteles als etwas Allgemeines, das das Bleibende und Notwendige erfaßt, also "das Gemeinsame innerhalb einer unter sich der Art nach verschiedenen Mehrheit" (Top. A, 5; 102 a 31) oder "die artbildende Differenz" (Met. I, 7; 1057 b 7). Begriffe werden von Aristoteles in Kategorien eingeteilt, wobei jede Kategorie durch die Wesenheit (Substanz) und Akzidentien beschrieben wird. Akzidentien umfassen dabei erstens Merkmale, die immer und not
wendig in einer Kategorie vorliegen müssen, wie etwa, daß ein Lebewesen atmet und zweitens Merkmale, die mit einer bestimmten Wahrscheinlichkeit bei dieser Kategorie auftreten, wie etwa, daß Lebewesen sich fortbewegen können. Diese Definition kommt modernen Auffassungen zur Begriffsklassifikation sehr nahe, wie etwa dem Merkmalsansatz von Smith, Shoben und Rips (1974) und dem Kategorienkonzept von Rosch (1978; siehe Kap. 4.1).

Wissen wird nach Aristoteles induktiv, durch allmähliche Abstraktion von beobachteten Einzelobjekten, erworben, wobei das Ziel der Abstraktion das Erkennen der Wesenheit von Objekten darstellt. Diese Wesenheit ist aber nach Aristoteles eine ideelle Struktur, die wie bei Platon *apriori* gegeben ist, und nicht in einem Lernprozeß entwickelt wird.

Induktion im Empirismus von Hume und J.S. Mill

Diese "metaphysische" Komponente der Abstraktion von Begriffen wird unter anderem von Hume in seinen "Enquiry concerning human understanding" (1748) aufgehoben[2]. Hume stellt die Induktion auf eine radikal empirisch-psychologische Basis. Induktion ist ein von

[1]Die Ausführungen über Aristoteles basieren sämtlich auf: Hirschberger (1963, Band I).

[2]Darstellung nach Hirschberger (1963, Bd. 2) und Maxwell (1974).

gespeicherten Erfahrungen ausgehender Assoziationsprozeß. Vorgestellte Objekte werden aufgrund von Ähnlichkeit oder raum-zeitlicher Nähe miteinander verbunden. Neues Wissen (Tatsachenwahrheiten) wird durch Induktion auf der Grundlage von Einzelerfahrungen erworben. Damit basieren alle Tatsachenwahrheiten, also auch die gesamten Ergebnisse der Naturwissenschaften, auf "Glauben" (*belief*), und sind nur mit einer bestimmten Wahrscheinlichkeit korrekt. Hume stößt bei diesen Grundannahmen jedoch auf ein noch für die heutige Philosophie schwerwiegendes Problem: Um induktive Schlüsse zu bestätigen, müssen neue induktive Schlüsse gezogen werden, die jedoch ihrerseits Bestätigungsbedarf haben.

Die Gedanken von Hume finden sich in den Arbeiten von John Stuart Mill (1843/1974) wieder. Während den Skeptiker Hume jedoch die Frage beschäftigt, ob induktive Schlüsse begründbar sind und unter welchen Umständen induktive Schlüsse gerechtfertigt werden können, geht Mill einen Schritt weiter, indem er die Grundlagen für eine induktive Logik legt. Er stellt Gesetze auf, mit denen auf Grundlage spezifischer Beobachtungen auf generelle Gesetze geschlossen werden kann. Mill wirft die Frage auf, wieviele Beobachtungen notwendig sind, um einen induktiven Schluß zu ziehen. Er stellt fest, daß für manche Fälle eine einzige Beobachtung ausreicht, während für andere Fälle eine Vielzahl von Beobachtungen vorliegen müssen.

Carnap's Konzept einer induktiven Logik

In der Folgezeit wurden die Ideen der induktive Logik mit all ihren Problemen durch Weiterentwicklungen der deduktiven Logik verdrängt. Die deduktive Logik wurde als mathematische Logik durch die Arbeiten von Boole, Frege, Russell und anderen zu einem allgemein akzeptierten konzeptuellen und methodischen Gerüst für das korrekte mathematische Schließen und Beweisen (z.B. Ebbinghaus, Flum & Thomas 1986). Erst Carnap (Carnap & Stegmüller 1959) begann von neuem den Versuch, eine induktive Logik zu entwickeln. Sein Ziel war es, ein auf dem von ihm begründeten Begriff der logischen Wahrscheinlichkeit basierendes Modell des Lernens aus Erfahrung zu erstellen (vgl. Kutschera 1972). Im folgenden sollen die Grundgedanken von Carnap's induktiver Logik dargestellt werden (vgl. Carnap & Stegmüller 1959; Hilpinen 1975; Kutschera 1972):

Die logische Wahrscheinlichkeit, daß eine Hypothese H bei gegebenem Erfahrungswissen E zutrifft, ist eine zweistellige Funktion $c(H,E)=r$, mit $0 \leq r \leq 1$. Carnap gibt für die Funktion c Axiome an, die über die Axiome der statistischen Wahrscheinlichkeit hinausgehen. Diese Axiome formalisieren rationale Entscheidungen über bedingte subjektive Wahrscheinlichkeiten, wobei unter anderem festgelegt wird, daß Ereignisse gleichwahrscheinlich sind, wenn keine Zusatzinformationen zur Verfügung stehen. Wesentlich für die Modellierung induktiven Lernens sind zwei Axiome, die festlegen, daß der Wert von c für eine

Hypothese $P(x)$ mit der Anzahl positiver Instanzen $P(i)$ in E zunimmt und daß c ausschließlich von der Zahl der positiven und negativen Instanzen $P(i)$ in E abhängt (Kutschera 1972, S. 126 ff.). Durch die Axiome wird die Wahrscheinlichkeitsfunktion c eindeutig bis auf einen positiven reellen Parameter *lambda* festgelegt. *Lambda* ist nicht von der Zahl der Beobachtungen in E abhängig und kann als personenspezifischer Parameter aufgefaßt werden, der die
Bereitschaft einer Person angibt, eine Hypothese aufgrund von Beobachtungen in E als gesichert anzunehmen (*index of caution*; vgl. Kutschera 1972, S. 132 ff.).

2.3 Paradoxien in induktiven Logiken

Die Paradoxien von Hempel und Goodman

Carnap's Ideen wurden von vielen Seiten als nicht haltbar angegriffen. Während Popper (1984) induktive Argumente strikt als irrational ableht, haben andere versucht, Probleme des Carnap'schen Ansatzes auszuräumen (Hempel 1965; Hintikka & Niinilouto 1976). Allerdings bleiben bei allen Ansätzen der logischen Induktion bisher logisch nicht auflösbare Paradoxien: Das Hempel Paradox (1965) zeigt, daß eine rein syntaktische Behandlung von induktiven Schlüssen zu unplausiblen Ergebnissen führen kann: Die Generalisierung, daß alle Raben schwarz sind ($rabe(x) \rightarrow schwarz(x)$), wird durch Beobachtung von Raben mit schwarzem Federkleid gestärkt. Jedoch kann die logisch äquivalente Aussage, daß alle nicht-schwarzen Dinge keine Raben sind ($\neg schwarz(x) \rightarrow \neg rabe(x)$) durch alle möglichen Beobachtungen andersfarbiger Objekte, wie etwa eines weißen Schuh's bestätigt werden. Solche Beobachtungen akzeptieren wir jedoch nicht als Beleg für die Hypothese, daß alle Raben schwarz sind.

Auf eine zweite Paradoxie, die in Carnap's syntaktischer Begründung der induktiven Logik begründet ist, wurde von Goodman (1965) hingewiesen: Er führt eine Eigenschaft *grue* ein, die bedeutet, daß ein Gegenstand bis zum Zeitpunkt t grün, danach aber blau ist. Wenn der Zeitpunkt t noch in der Zukunft liegt, gibt es keinen Grund, die Aussage "Alle Smaragde sind grün" der Aussage "Alle Smaragde sind *grue*" vorzuziehen. Diese Paradox steht in engem Zusammenhang mit dem bereits diskutierten Problem der Anzahl notwendiger Beobachtungen: Wir verfügen über Wissen über die Variabilität von Merkmalen, so wissen wir, daß Farben von Objekten überlicherweise konstant sind (außer z.B. bei Chamäleons) und ziehen somit die Aussage "Alle Smaragde sind grün" der Merkmalszuweisung *grue* vor.

Psychologismus als Ausweg

Alle Versuche, Mängel des Systems der induktiven Logik zu beheben, führten bisher zu neuen Problemen (z.B. Kutschera 1972, S. 137 ff.). Dies legt die Vermutung nahe, daß eine induktive Logik nicht nach dem formal-syntaktischen Prinzip der deduktiven Logik aufgebaut sein kann. Die bei Hume's, Mill's und Carnap's Ansätzen gezeigten Probleme können jedoch meiner Meinung nach durch eine in der Kognitionswissenschaft begründete Herangehensweise vermieden werden.

Hume's Dilemma, daß Inferenzschlüsse nur über weitere, ebenfalls unsichere Inferenzschlüsse bestätigbar sind, kann aufgelöst werden, indem man annimmt, daß für die meisten Vorhersagen im Alltag genügt, daß die Schlüsse eine bestimmte Wahrscheinlichkeit haben, korrekt zu sein (vgl. Holland, Holyoak, Nisbett & Thagard 1986, S. 37). Damit kann ein Schwellenwert für den Bestätigungsgrad induktiv gewonnener Aussagen angegeben werden. Hat eine Hypothese eine über diesem Wert liegende Sicherheit erlangt, kann der Zirkel induktiver Bestätigungen durchbrochen werden.

Mill's Problem der Unterschiedlichkeit bei der Anzahl notwendiger Beobachtungen als Grundlage für einen Inferenzschluß kann beantwortet werden, indem man Merkmale von Objekten bezüglich ihrer eingeschätzten Variabilität kennzeichnet. "Hat Schnupfen" oder "Name der Unbekannten ist x" wären zum Beispiel hoch variable Merkmale, "hat zwei Ohren" oder "Unbekannte ist mit einem Wert additiv verknüpft" dagegen nicht. Damit sind für Objekte, die mit einem variablen Merkmal assoziiert werden, viele Beobachtungen notwendig, um eine Inferenz mit akzeptabler Sicherheit zu ziehen; für Objekte, die mit einem als üblicherweise konstant bleibenden Merkmal verbunden werden, genügt dagegen unter Umständen eine einzige Beobachtung.

Sowohl das Hempel-Paradox, als auch das Goodman-Paradox werden also hinfällig, wenn man den strengen Rahmen der Logik verläßt und stattdessen psychologische Plausibilitätskriterien ansetzt, also auf eine rein syntaktische Form logischen Schließens verzichtet.

Damit ist der Psychologismus in der Logik kein Problem (Carnap & Stegmüller 1959, S. 30 ff.), sondern eher als ein möglicher Ausweg zu verstehen (vgl. auch Westermann & Gerjets 1991; Weimer 1975). Jedoch ist unter Psychologismus kein Rückfall in den radikalen Empirismus von Hume und Mill zu verstehen: Im Rahmen
der kognitiven Psychologie wird anstelle eines rein assoziativen, auf gemeinsam wahrgenommenem basierenden, Gedächtnismodells von einer strukturierte Form von Wissensrepräsentation mit teilweise nicht lernbaren sondern bereits vorhandenen Ordnungsstrukturen und Regelmechanismen ausgegangen (z.B. Pylyshyn 1984). Diese Annahmen bedeuten für die Modellierung von Wissenserwerbsprozessen, daß nicht von der empiristischen *tabula rasa* Konzeption ausgegangen wird, sondern Lernen immer in Abhängigkeit von bereits vorhande-

nen Wissensstrukturen beschrieben wird (vgl. Hintergrundwissen E bei Carnap). Ähnliche Voraussetzungen finden sich in Wissensrepräsentationskonzepten und Such- und Schlußalgorithmen der künstlichen Intelligenz wieder.

In den folgenden Anschnitten werden verschiedene kognitionswissenschaftliche Modellvorstellungen der Induktion vorgestellt. Dabei wird nicht zwischen Ansätzen der kognitiven Psychologie und der künstlichen Intelligenz (maschinelles Lernen) unterschieden, da die beschriebenen Ansätze stets auf kognitionspsychologisch plausiblen Vorstellungen basieren.

3 Formalismen zur Repräsentation von Problemlösewissen

In diesem Kapitel werden die in der Kognitionswissenschaft etablierten Ansätze der Wissensrepräsentation dargestellt. Zunächst wird der Informationsverarbeitungsansatz, der den konzeptuellen Rahmen für diese Repräsentationsformalismen liefert, kurz skizziert. Im folgenden werden dann semantische Netze und Schemata, Produktionssysteme und mentale Modelle eingeführt. Jeder dieser Ansätze wird zunächst allgemein dargestellt und dann bezüglich zweier Aspekte diskutiert: Der erste Aspekt ist, welcher Ausschnitt von Wissen, das zur Beschreibung von Problemlöseprozessen notwendig ist, im jeweiligen Ansatz fokussiert wird. Der zweite Aspekt ist, ob und wie im Rahmen des jeweiligen Repräsentationsformalismus Lernprozesse modellierbar sind. Am Ende dieses Kapitels wird für eine Kombination der diskutierten Repräsentationsformate zur Modellierung des induktiven Erwerbs von Problemlösefertigkeiten argumentiert.

3.1 Gedächtnisstrukturen und Lernen im Informationsverarbeitungsansatz

Konzeptuelle Grundlage für diese Arbeit ist der Informationsverarbeitungsansatz (vgl. z.B. Atkinson & Shiffrin 1968). In diesem Ansatz wird davon ausgegangen, daß der menschliche Organismus ein offenes System ist, das Informationen aufnimmt, speichert, verarbeitet und wieder ausgibt. Zur Veranschaulichung des Prozesses der Informationsverarbeitung wird dabei häufig, in Analogie zur Architektur des von-Neumann-Rechners, zwischen Arbeitsspeicher (Kurzzeitgedächtnis) und Langzeitspeicher (Langzeitgedächtnis) unterschieden. Dabei werden alle während eines Verarbeitungsprozesses aktivierten Informationen im Kurzzeitgedächtnis und alle dauerhaft gespeicherten Informationen im Langzeitgedächtnis lokalisiert. Aktivierte Informationen können sowohl über die Umwelt aufgenommen werden als auch aus dem Langzeitgedächtnis abgerufen werden. Der Begriff Information wird im folgenden synonym mit "Wissen" verwendet. Auf das schwierige Unterfangen, diese Begriffe zu definieren, wird hier verzichtet.

Die Definitionen für Lernen sind zahlreich (z.B. Simon 1983). Eine im Rahmen des Informationsverarbeitungsansatzes geeignete Sicht auf Lernen ist, Lernen als das Konstruieren oder Verändern von Repräsentationen von Erfahrungen zu verstehen (Michalski 1986). Lernen bedeutet also eine Modifikation der im Langzeitgedächtnis gespeicherten Wissensstrukturen. Obwohl der Begriff "Lernen" vor allem im Zusammenhang mit behavioristischen Theorien (Skinner 1951) verwendet wird, wird er in dieser Arbeit als Synonym zu dem in der kognitiven Psychologie bevorzugten Begriff Wissenserwerb verwendet.

In den 60er Jahren wurden mathematische Lernmodelle vorgeschlagen (z.B. Estes 1959). Damit wurde ein wichtiger Schritt hin zu präziseren Formulierungungen psychologischer Theorien unternommen. Dies und die Erkenntnis, daß psychologische Prozesse fast ausnahmslos stochastischer Natur sind und deshalb mit probabilistischen Methoden modelliert

werden sollten, war ein wesentlicher Fortschritt für die Psychologie. Allerdings konnten solche mathematischen Lernmodelle nur auf sehr begrenzte Gebiete, wie den im Behaviorismus untersuchten Verhaltensaspekten, angewendet werden. Aus diesem Grund wird auf diese Tradition der Modellierung von Lernprozessen in dieser Arbeit nicht eingegangen. Neuere Ansätze, die sich um Präzision bei der Modellierung von komplexeren kognitiven Prozessen bemühen, verwenden meist eine Kombination mathematischer und informatischer Methoden (z.B. Anderson 1983).

Auch auf konnektionistische Modelle des Wissenserwerbs wird in dieser Arbeit nicht eingegangen (vgl. Rumelhart & McClelland 1986). Während sich solche Ansätze für die Modellierung des Erwerbs grundlegender, wahrnehmungsnaher Wissensstrukturen durchgesetzt haben, scheint mir ein so molekulares Repräsentationsformat für die Modellierung des Erwerbs komplexer kognitiver Fertigkeiten nicht angemessen (vgl. Fodor & Pylyshyn 1988). Die Wissensmodellierung mit klassischen, meist logikbasierten, Ansätzen stellt eine Abstraktion von der materiellen Substanz Gehirn dar, sowie etwa die Programmierung mit deklarativen Programmiersprachen von der Architektur des von-Neumann-Rechners abstrahiert.

Die im folgenden dargestellten Ansätze haben primär den Status von Modellvorstellungen über die Strukturierung von im Langzeitgedächtnis repräsentierten Wissensstrukturen.

3.2 Semantische Netze und Schemahierarchien zur Repräsentation von Problemkonzepten

Repräsentationsformalismus

Ein früher Ansatz zur strukturierten Repräsentation von Begriffen ist der der semantischen Netzwerke. Begriffe (oder Konzepte) werden dabei als mentale Repräsentation von Objektklassen verstanden. Ein Begriff erhält seine Bedeutung durch beschreibende Merkmale und durch seine Beziehung zu anderen Begriffen (Smith & Medin 1981). Quillian (1968) schlug ein Gedächtnismodell vor, bei dem Begriff-Oberbegriff-Relationen in einer Baumstruktur repräsentiert werden. Jeder Knoten im Baum steht dabei für einen Begriff. Kanten im Baum verbinden Begriffe erstens mit beschreibenden Merkmalen (*has-prop*-Relationen) und zweitens mit Oberbegriffen (*is-a*-Relationen). Eine zentrale Eigenschaft dieses Gedächtnismodells ist, daß das Prinzip der kognitiven Ökonomie über Merkmalsvererbung realisiert wird (siehe z.B. Schmid & Kindsmüller 1994, Kap. 2). Alle Eigenschaften eines Oberbegriffs werden unter anderem aufgrund der Transitivität der Inklusionsrelation an hierarchisch untergeordnete Konzepte weitergereicht (siehe Abb. 1).

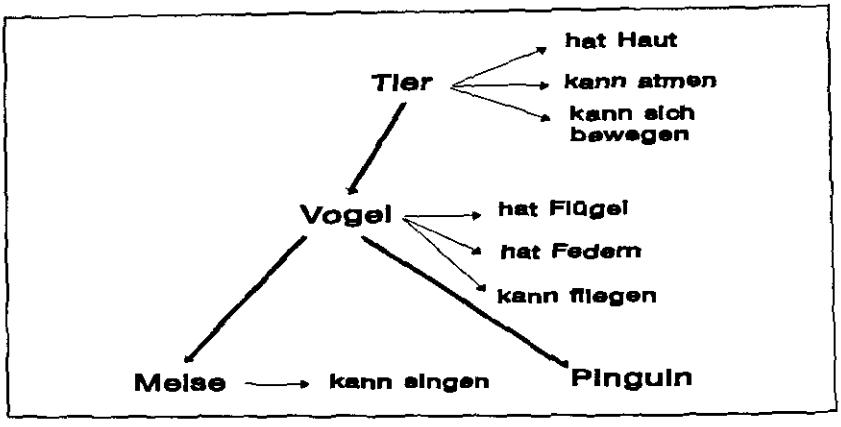


Abb. 1: Ausschnitt aus einem hierarchischen semantischen Netz nach Collins und Quillian

Das Konzept der semantischen Netze diente als Ausgangspunkt für viele Gedächtnismodelle. Es findet sich in den Aktivationsausbreitungsnetzen wieder, bei denen die Kanten zwischen Knoten mit Stärkewerten belegt werden (Anderson & Bower 1973) und fand in die Informatik unter anderem durch die Wissensrepräsentationssprache KL-ONE Eingang (Brachman & Schmolze 1985), wo es eine semantische Fundierung über die Prädikatenlogik erster Stufe erhielt (Nebel 1990). Empirische Prüfungen zur Organisation des menschlichen Konzeptgedächtnisses konnten den klassischen Ansatz semantischer Netze nur zum Teil bestätigen (Collins & Quillian 1969). Vor allem Befunde, die zeigen, daß die Prototypikalität von Begriffen einen großen Einfluß auf die Verifikation von Aussagen hat, führten zu alternativen Konzeptionen, wie dem Prototypenkonzept von Rosch (1978), das mit Hilfe unscharfer Mengen modellierbar ist (Osherson & Smith 1982).

Eine Modellierungsform, die auf dem Konzept der semantischen Netze basiert, und Grundgedanken des Prototypenansatzes integriert (Schank 1982), ist der Schema-Ansatz. Zentrale Änderung gegenüber den semantischen Netzen ist die stärkere Betonung der Strukturiertheit von Begriffen. Die Schemakonzeption wurde von Bartlett (1932) vorgeschlagen und in den 70er Jahren wieder aufgegriffen (Minsky 1975; Schank 1982). Ein Schema ist eine Wissensstruktur, die Leerstellen (*slots*) aufweist, die mit spezifischen Werten, Wertebereichen oder *defaults* belegt werden (siehe Abb. 2).

Das Vogel-Schema

Oberkategorie	: Tier
Extremitäten	: Flügel
Bedeckung	: Federn
Fortbewegung	: Fliegen

Abb. 2.: Ein Schema für das Konzept Vogel

Dabei entspricht jede Leerstelle einem Merkmal (z.B. Bedeckung) zusammen mit einer Merkmalsausprägung (z.B. Federn). *Defaults* sind dabei Standardbelegungen von Leerstellen, die solange beibehalten werden, bis spezifischere Informationen über einen Begriff vorliegen. Diese Standardbelegungen spiegeln die Beobachtung wieder, daß Erwartungen über Strukturen der Welt das menschliche Verhalten steuern. *Defaults* sind damit eine Form der Realisierung von Prototypen ("üblicherweise fliegen Vögel").

Durch Leerstellen, die auf Ober- und Unterbegriffe verweisen, wird die Idee der hierarchischen Gedächtnisorganisation beibehalten. Die strukturierte Repräsentation von Begriffen in Schemahierarchien hat einige Vorteile gegenüber der Konzeption der semantischen Netze: Erstens können die Leerstellen eines Schemas als Suchvorgaben bei Prozessen der Informationsaufnahme dienen (Schemainstantiierung). Zweitens liefert das Konzept der *default*-Werte, eine sinnvolle Beschreibung dafür, daß Menschen bei der Konfrontation mit neuen Konzepten zunächst von Vorannahmen ausgehen, die mit ihrem Vorwissen in Einklang stehen (Rumelhart & Norman 1978). Diese beiden Punkte beziehen sich auf die psychologische Plausibilität der Schema-Konzeption als Gedächtnismodell. Ein dritter Vorteil, der auch für die Computermodellierung von Konzeptrepräsentationen bedeutsam ist, ist die sinnvollere Behandlung von Ausnahmefällen im Schemakonzept. In semantische Netzen ist es problematisch, Merkmalsvererbung und Ausnahmefälle (wie nicht fliegende Vögel) zu modellieren. Im Schemaansatz kann dieses Problem über die *default*-Konzeption gelöst werden. Im Vogelschema wird "kann fliegen" als Standardannahme eingetragen. Diese Annahme wird aber durch Wertebelegung untergeordneter Schemata überschrieben.

Die psychologisch adäquatere und effizientere Behandlung von Ausnahmen durch *defaults* führt jedoch zu nicht-monotonen Schlußregeln. Das heißt, daß bereits gezogene Schlüsse aufgrund neuer Schlüsse ungültig werden können. Damit ist eine semantische Fundierung von Schema-Ansätzen mit *defaults* nicht mehr im Rahmen der klassischen Prädikatenlogik möglich (Reiter 1980).

Modelle, die auf den beschriebenen Grundannahmen basieren, sind unter anderem die *Frame*-Konzeption von Minsky (1975) und die *Scripts* von Schank und Abelson (1977), die den Schema-Ansatz auf die Repräsentation von Ereignissen übertragen. Auch die erwähnte Sprache KL-ONE hat einige Gedanken des Schemaansatzes übernommen. Im Ansatz der terminologische Logiken, der z.B. KL-ONE und BACK (Peltason, Schmiedel, Kindermann & Quantz 1989) zugrundeliegt, wird der Schema-Ansatz formalisiert.

Schemata (und semantische Netze) betonen statische Wissensaspekte. Um Informationsverarbeitungsprozesse mit solchen Wissensstrukturen zu modellieren, müssen zusätzlich Annahmen über den Prozeß der Schemaaktivierung formuliert werden. Allgemein wird hier angenommen, daß die Schemata mit den aktuellen Daten im Kurzzeitgedächtnis verglichen werden (*matching*). Aktiviert wird das Schema, das die größte Übereinstimmung mit den aktuellen Daten aufweist. Diese Daten werden dann zur Instantiierung des aktivierten Schemas verwendet.

Repräsentation von Problemkonzepten

Semantische Netze und Schemata werden üblicherweise zur Modellierung des menschlichen Konzept-Gedächtnisses verwendet. Einige Autoren verwendeten den Schemaansatz speziell zur Repräsentation von konzeptuellem Wissen beim Problemlösen: Anderson, Greeno, Kline & Neves (1981) repräsentieren das Wissen über die Eigenschaften geometrischer Objekte (z.B. Kongruenz von Dreiecken) in Schemata und nutzen diese Wissensstrukturen zur Simulation von Beweistechniken. Rumelhart und Norman (1981) modellieren das Wissen über die Bedienung eines Texteditors mit Schemata (vgl. Kap. 4.2). Bonar und Soloway (1989) und Vorberg und Goebel (1991) verwenden Schemata zur Modellierung von Programmierwissen (vgl. Kap. 5.4). Zur Modellierung von Prozessen der Handlungsplanung, die sicher einen zentralen Anteil beim Problemlösen haben, wird häufig das Script-Konzept verwendet (Anderson 1983; McDermott 1967; Wilensky 1983). Problemlösungen werden hier durch Instantiierung von, häufig hierarchisch organisierten, Handlungsplänen erzeugt.

Die Modellierung von Wissensstrukturen im Schemaansatz betont folgende, für eine Beschreibung von Problemlösungsprozessen relevante, Aspekte:

- **Strukturierte Repräsentation von Problemkonzepten:** Wissen über einen Problembereich wird strukturiert repräsentiert, in dem jedes Problemkonzept erstens durch Merkmale und Merkmalsausprägungen charakterisiert wird (horizontale Strukturierung) und zweitens mit generelleren und spezifischeren Konzepten in Beziehung gesetzt wird (vertikale Strukturierung).

- **Anbindung eines aktuellen Problemlöseprozesses an Vorwissen:** Eine aktuelle Problemstellung aktiviert ein Schema. Das aktivierte Schema liefert, durch die Vorgabe von Merkmalen, eine Struktur zur Suche und zur integrierten Repräsentation von aktueller Information. Zusätzlich liefert ein aktiviertes Schema mit bereits instantiierten Leerstellen möglicherweise bereits relevante Information für die Problemlösung.

Wird das durch aktuelle Daten instantiierte Schema als neues, spezifischeres Schema im Langzeitgedächtnis gespeichert, wurde neues Wissen erworben. Ansätze, die über diese einfache Modellierung des Schemaerwerbs hinausgehen und den Aufbau völlig neuer Schemata beschreiben, existieren jedoch kaum.

Schemainduktion

Eine Ausnahme bildet der Ansatz von Rumelhart und Norman (1978). Neben dem gerade dargestellten Wissenszuwachs durch Instantiierung von Schemata (*accretion*), beschreiben sie Aufbau und Umstrukturierung von Schemata (*restructuring*), sowie Prozesse der Feinabstimmung (*tuning*) von Schemata. Feinabstimmung umfaßt die Differenzierung von Schemata durch Einschränkung der Wertebereiche, sowie die Generalisierung von Schemata durch Erweiterung der Wertebereiche. Umstrukturierung meint sowohl die Induktion neuer Schemata, als auch das Hinzufügen oder Wegnehmen von Leerstellen vorhandener Schemata. Die Induktion eines neuen Schemas kann ebenfalls durch Generalisierung entstehen. In diesem Fall werden konstante Werte durch abstraktere Informationen ersetzt. So kann zum Beispiel auf Grundlage von vorhandenen Schemata über Quadrat und Fünfeck durch Mustervergleich ein Schema über reguläre Polygone gebildet werden (siehe Abb. 3).

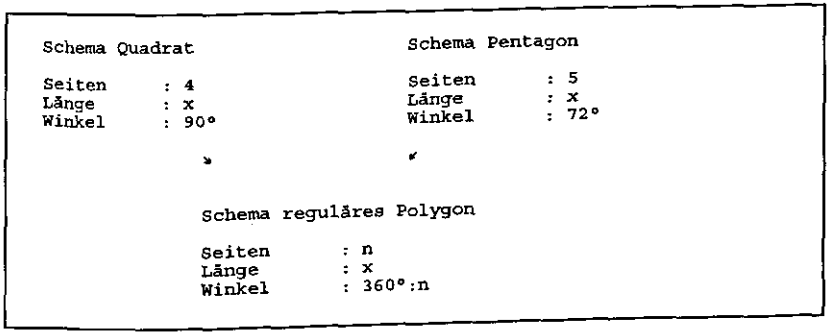


Abb. 3: Schema-Generalisierung nach Rumelhart und Norman

Eine Umstrukturierung durch Wegnehmen von Leerstellen erfolgt zum Beispiel, wenn aus Schemata für Weinglas und für Bierglas ein allgemeines Trinkglas-Schema gebildet wird, indem Leerstellen für die Form des Gefäßes oder die Art des darin gereichten Getränkes entfernt werden. Ein Beispiel für die Feinabstimmung von Schemata wäre zum Beispiel die Inferenz eines Dackel-Schemas durch Einschränkung der Wertebereiche des Hundeschemas. Der Erwerb neuer allgemeinerer Schemata aufgrund analoger Lernprozesse wird ebenfalls von Rumelhart und Norman (1981) beschrieben. Auf diese Modifikation des hier dargestellten Ansatzes wird in Kapitel 4.2 eingegangen.

Lernen wird von Rumelhart und Norman (1978) vor allem als Modifikation bestehender Wissensstrukturen beschrieben. Damit liegt diesem Modell die Annahme zugrunde, daß neues Wissen nicht *tabula rasa* erworben wird, sondern immer auf bereits vorhandene Strukturen bezogen wird. Diese Annahme scheint mir für die Modellierung des Erwerbs höherer kognitiver Strukturen sinnvoll. Prozesse der Suche in der Wissensbasis, sowie des Einfügens neuer Schemata werden im Ansatz von Rumelhart und Norman (1978) nicht exakt definiert. Allerdings gibt es in der Informatik etablierte Techniken für Suche und Integration neuer Objekte in hierarchische Strukturen (z.B. Horowitz & Sahni 1978), die man für eine Implementation des Modells von Rumelhart und Norman verwenden könnte.

3.3 Produktionssysteme zur Repräsentation von Problemlösefertigkeiten

Repräsentationsformalismus

Schemata sind gut dazu geeignet, eher statische Informationen, wie etwa konzeptuelles Wissen über verschiedenste Bereiche, zu repräsentieren (deklaratives Wissen). Ein großer Teil unseres Wissens besteht aber aus Strategien und Prozeduren zum Umgang mit verschiedenen Problemsituationen ("Wenn dich ein Hund anbellt, dann sprich mit ihm", "Wenn du eine Gleichung mit einer Unbekannten lösen willst, dann löse sie nach der Unbekannten auf"). Wissen dieser Art (prozedurales Wissen) wird meist in Form von Produktionssystemen (Newell & Simon 1972; Anderson 1983) modelliert. Während das zentrale Anliegen der Schemaansätze die Modellierung der menschlichen Gedächtnisorganisation ist, steht im Ansatz der Produktionssysteme die Modellierung von Problemlöseprozessen und Lernvorgängen im Vordergrund (Klahr, Langley & Neches 1987).

Produktionssysteme bestehen aus einem Arbeitsspeicher, in dem aktuelle Daten und/oder Ziele abgelegt sind und aus einem Regelspeicher, in dem einer Menge von Produktionsregeln abgelegt ist. Produktionsregeln sind Bedingungs-Aktionspaare der Form

IF <ausdruck-1> THEN <ausdruck-2> oder
<ausdruck-1> --> <ausdruck-2>.

Produktionsregeln und Daten werden in sogenannten *recognize-act* Zyklen miteinander in Beziehung gesetzt, der aus drei Schritten besteht:

1) *Pattern-Matching*: Suche alle Regeln, deren Bedingungsteil mit dem Zustand des Arbeitsspeichers übereinstimmen. Regel-Instantiierung.
2) *Conflict-Resolution*: Wähle eine dieser Regeln für die Anwendung aus.
3) *Act*: Wende die ausgewählte instantiierte Regel an. Modifikation des Arbeitsspeichers.

Die verwendeten Konfliktlösungsstrategien sind dabei der zentrale Kontrollmechanismus, der die Arbeitsweise des Produktionssystems maßgeblich bedingt. Eine bekannte Konfliklösungsstrategie ist die Mittel-Ziel-Analyse von Newell & Simon (1972), bei der die Regel ausgewählt wird, die die Differenz des aktuellen Zustands im Arbeitsspeicher zum gewünschten Zielzustand am stärksten verringert. Zur Bewertung der Differenz von aktuellem Zustand und Zielzustand werden üblicherweise heuristische Funktionen verwendet. Eine Alternative ist die Belegung von Regeln mit Stärkewerten, die aufgrund von erfolgreichen Regelanwendungen inkrementiert werden (Anderson 1983). Hier wird die Regel ausgewählt, die den höchsten Stärkewert besitzt. Weitere Strategien, wie zum Beispiel die Regel-Spezifität, finden sich in Anderson (1983) und Neches, Langley und Klahr (1987).

Eine zweite wichtige Entwurfsentscheidung bei der Modellierung ist das Format und der Detailliertheitsgrad der Datenrepräsentation. Dieses Problem wird z.B. bei Amarel (1983) diskutiert.

Nach Newell und Simon (1972) weisen Produktionssysteme verschiedene Merkmale auf, die sich für die Modellierung menschlichen Verhaltens als adäquat erweisen. Das wichtigste Merkmal ist die **Unabhängigkeit** der Regeln. Die Unabhängigkeit der Regeln wird dadurch erreicht, daß Produktionsregeln nur indirekt miteinander in Beziehung stehen: Die Änderung der Datenbasis durch Anwendung einer Regel schafft Bedingungen für die Anwendung einer anderen Regel. Aus diesem Grund ist Lernen in Produktionssystemen einfach zu modellieren: Eine neue Regel wird lediglich zur Menge der bereits vorhandenen Regeln hinzugefügt. Die Integration eines neuen Schemas in eine Schemahierarchie macht dagegen unter Umständen aufwendige Umstrukturierungen notwendig.

Wissensmodellierungen in Form von Produktionssystemen sind schon für relativ simple Problemlöseprozesse sehr aufwendig. So werden für Programme zur Simulation von einfachen arithmentischen Aufgabenlösungen bereits - je nach Regelformulierung - um die 10 Regeln mit komplexen (konjungierten) Bedingungs- und Aktionsteilen benötigt (Neches, Langley & Klahr 1987, S. 4ff.). Dies mag auf den ersten Blick als Nachteil dieser Form der

Wissensrepräsentation erscheinen. Faßt man Produktionssysteme aber als Modelle menschlicher Problemlöseprozesse auf, wird die Notwendigkeit, alle für eine Problemlösung notwendigen Wissensanteile im Detail zu formulieren, vorteilhaft, da die Modelle deutlich an Präzision gewinnen (Opwis 1988).

Repräsentation von Problemlösefertigkeiten

Während Schemata allgemein zur Repräsentation von konzeptuellem Wissen eingesetzt werden und lediglich einige Anwendungen der Schematheorie im Bereich Problemlösen bestehen, wurden frühe Produktionssysteme, wie der General Problem Solver (Newell 1972; Newell & Simon 1972), direkt als Modelle menschlicher Problemlöseprozesse entwickelt. Mit dem General Problem Solver (GPS) konnten verschiedenste menschliche Problemlöseprozesse, wie etwa das Lösen kryptho-arithmetischer Aufgaben, der Turm von Hanoi oder das Lösen logischer Probleme simuliert werden. Zentrales Verdienst von Newell war der Entwurf einer heuristischen Suchstrategie, der bereits erwähnten Mittel-Ziel-Analyse (MEA). Der Problemlöseprozeß wird dabei als Suche nach einem Lösungspfad in einem Problemraum beschrieben. Der Problemraum repräsentiert alle möglichen Problemzustände, die durch Transformationen mit vorgegebenen Operatoren (Produktionsregeln) erreichbar sind. In einem rekursiven Prozeß wird versucht, jeweils den Operator zu finden, der die Differenz zwischen dem aktuellen Problemzustand und dem gewünschten Zielzustand am stärksten reduziert.

Bei in Produktionsregeln gespeichertem Wissen steht der Aspekt der Performanz im Vordergrund. Modelliert wird hier nicht das konzeptuelle Wissen über einen Problembereich, sondern die Fertigkeit (*skill*), ein Problem zu lösen. Problemlösewissen ist in Form von Daten-Handlungs-Muster Verbindungen repräsentiert. Damit wird hier der Aspekt der automatisierten, effizienten Produktion von Handlungen betont.

Adaptive Produktionssysteme

Bereits zu einem sehr frühen Zeitpunkt der Produktionssystem-Entwicklung wurden lernende Systeme, sogenannte adaptive Produktionssysteme, entwickelt (vgl. Neches, Langley & Klahr 1987). Hierzu gehört zum Beispiel der *Poker-Player* von Waterman (1970). Dieses System modelliert den Erwerb von heuristischen Strategien beim Pokerspielen, allerdings ohne den Anspruch damit menschliches Verhalten abzubilden. Ein weiteres Beispiel ist das System HACKER (Sussman 1975), das das Planungssystem STRIPS (Fikes & Nilsson 1971) um eine Lernkomponente für den Aufbau von Makrooperationen ergänzt.

Die beiden wichtigsten Systeme, die als psychologische Modelle des Wissenserwerbs konzipiert sind, stammen von Newell und von Anderson. Newell konzipierte das System SOAR, das eine Weiterentwicklung des General Problem Solver ist (Rosenbloom & Newell 1987). SOAR modelliert das Potenz-Gesetz des Lernens (Snoddy 1926) durch Verknüpfung (*chunking*) von Datenmustern für Aufgabe (*stimulus*) und Handlung (*response*) in immer komplexeren Produktionsregeln. Das bekannteste Produktionssystem als umfassende Theorie menschlicher Informationsverarbeitung stammt von Anderson (1976; 1983). Er modifizierte die klassische Produktionssystem-Architektur, indem er dem Speicher für Produktionsregeln ein deklaratives Langzeitgedächtnis hinzufügte, das als Aktivationsnetzwerk realisiert ist.

Der Erwerb neuer Regeln wurde ebenfalls umfassend von Anderson (1982; vgl. Kap. 4.3) beschrieben: Neues Wissen wird zunächst in deklarativer Form, also als Faktenwissen, gespeichert. Mithilfe allgemeiner Regeln kann dieses deklarative Wissen zur Anwendung kommen. Wird das Wissen in Verbindung mit der allgemeinen Regel häufiger angewendet, findet eine "Wissenskompilierung" statt, in der die neue Information in eine Produktionsregel umgewandelt wird. Weitere Anwendungen dieser Regel führen zu einer "Wissensoptimierung". Optimierung umfaßt dabei Spezialisierungen (Hinzunahme von Anwendungsbedingungen) und Generalisierungen (Ersetzen von Konstanten durch Variablen).

Im Ansatz der Produktionssysteme sind Prozesse der Regelsuche und der Integration neuer Regeln in die Wissensbasis detailliert beschrieben: Das Auffinden einer Regel wird über Mustervergleichsprozeduren gesteuert. Das Einfügen einer neuen Regel wird einfach als Hinzufügen eines neuen Elementes in die Wissensbasis realisiert. Die Annahme einer listenartigen Wissensbasis ohne interne Strukturierung ist psychologisch allerdings nicht unbedingt plausibel. Man kann vermuten, daß nicht nur Faktenwissen, sondern auch Problemlösewissen in expliziten Strukturen organisiert ist. Holland, Holyoak, Nisbett und Thagard (1986, S. 143 f.) argumentieren, daß nur die explizite Strukturierung adäquat für induktive Prozesse des Regelerwerbs sein kann. Nur wenn Regeln mit ähnlichen Bedingungen zu Regel-Clustern zusammengefaßt sind, kann der Erwerb neuer Regeln sinnvoll durch Kombinations-, Abstraktions- und Spezialisierungsmechanismen modelliert werden.

3.4 Mentale Modelle als flexible Wissensstrukturen für aktuelle Problemlöseprozesse

Repräsentationsformalismus

Ein zur Zeit sehr aktueller Ansatz (z.B. Glenberg, Meyer & Lindem 1987) ist der der mentalen Modelle (Johnson-Laird 1983). Unter dem Begriff mentales Modell werden verschiedene Vorstellungen zusammengefaßt, denen gemeinsam ist, daß die Repräsentation

eines Realitätsbereiches strukturelle Ähnlichkeit zu diesem Realitätsbereich behält. Solche strukturellen Ähnlichkeiten können etwa relative Größenunterschiede von Objekten und ihre Lage zueinander sein. Allgemeiner sind mentale Modelle Repräsentationen von abstrahierten Objekten zusammen mit räumlichen, zeitlichen oder kausalen Relationen zwischen diesen Objekten. Diese strukturierten internen Repräsentationen von Zuständen der Welt können durch Anwendung mentaler Operationen verändert werden. Auf diese Weise können (kausale) Erklärungen über Vorgänge und Sachverhalte formuliert und Vorhersagen getroffen werden. Die Idee, menschliche Verstehensprozesse mit Hilfe von "ablauffähigen" internen Modellen zu erklären, wurde bereits von Craik (1943) formuliert.

Johnson-Laird (1983) versteht unter interner Repräsentation eine symbolische Repräsentation. Damit grenzt er den Ansatz mentaler Modelle deutlich vom Ansatz analoger Repräsentationen (Kosslyn 1980; Metzler & Shepard 1974) ab, der häufig dem Ansatz mentaler Modelle zugeordnet wird. Johnson-Laird (1983, Kap. 7) geht, wie Fodor (1975), von einer "mentalen Sprache" aus, die propositionaler Natur ist. Mentale Modelle sind für Johnson-Laird eine zusätzliche Repräsentationsebene, die auf Basis der Semantik propositionaler Repräsentationen gebildet wird.

Damit postuliert Johnson-Laird implizit, daß mentale Modelle im Kurzzeitgedächtnis aufgebaute Repräsentationen sind, die als effektive Strukturen für aktuelle Verstehens- und Problemlöseprozesse dienen. So können zum Beispiel propositional kodierte Beschreibungen von Größenverhältnissen in ein mentales Modell überführt werden, das es ermöglicht, Schlußfolgerungen zu ziehen, ohne logische Schlußregeln, wie die Transitivität, vorauszusetzen: Aus den Informationen größer_als(John,Mary) und größer_als(Mary,Julia) wird eine lineare Anordnung gebildet, aus der verschiedene Informationen direkt ablesbar sind (siehe Abb. 4).

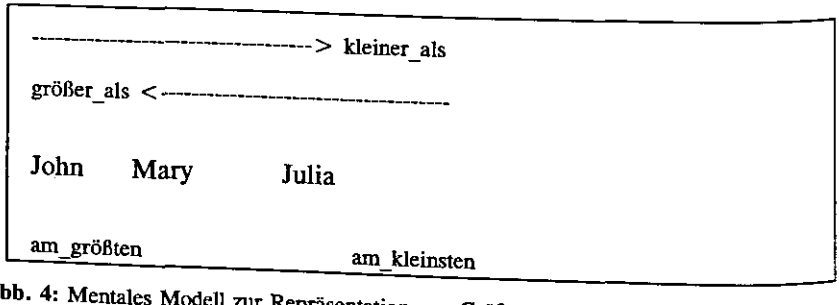


Abb. 4: Mentales Modell zur Repräsentation von Größenverhältnissen

Der Inhalt des Arbeitsgedächtnisses wird damit nicht mehr in 7 ± 2 unverbundenen Bedeutungseinheiten (Miller 1956) oder als Daten, die Muster für den Abgleich mit dem Bedingungsteil von Produktionsregeln bilden (Newell & Simon 1972), beschrieben, sondern als zusammenhängende Struktur, in der Informationen über relative Entferntheit oder Relevanz von Elementen abbildbar ist.

Johnson-Laird's Rahmenkonzept behandelt zentral Prozesse des Sprachverstehens und des Schlußfolgerns. Es enthält keine Annahmen über Prozesse des Wissenserwerbs. Eine Variante mentaler Modelle, die vor allem zur Modellierung von Repräsentationen physikalischer Zusammenhänge verwendet wird, stammt von de Kleer und Brown (1983), sowie Forbus und Gentner (1986). Diese Autoren formulieren Modelle zum Erwerb korrekter mentaler Modelle über physikalische Zusammenhänge. In Abweichung von Johnson-Laird's Modellvorstellungen nehmen diese Autoren mentale Modelle als Strukturen des Langzeitgedächtnisses an, die bei der Auseinandersetzung mit einem Gegenstandbereich stufenweise verfeinert werden.

Mentale Modelle als Kurzzeitgedächtnisstrukturen für aktuelle Problemlöseprozesse

Ein Ansatz, der mentale Modelle ebenfalls als Strukturen des Kurzzeitgedächtnisses beschreibt, stammt von Holland, Holyoak, Nisbett & Thagard (1986). Zentrales Anliegen der Autoren ist der Entwurf eines Rahmenmodells für induktive Prozesse. Dabei werden sowohl Schlußfolgerungs- und Problemlöseprozesse, als auch Prozesse des induktiven Wissenserwerbs beschrieben. Die Autoren kombinieren in ihrem Modell Gedanken des Schemaansatzes mit einer Produktionssystemarchitektur. Zentrale Wissensstruktur im Langzeitgedächtnis sind, wie beim Produktionssystemansatz, Regeln. Aber anders, als in klassischen Produktionssystemen werden diese Regeln in einer hierarchischen Struktur von Regel-Clustern repräsentiert. Für aktuelle induktive Problemlöseprozesse wird mit Hilfe dieser Regeln ein mentales Modell aufgebaut (siehe Abb. 5).

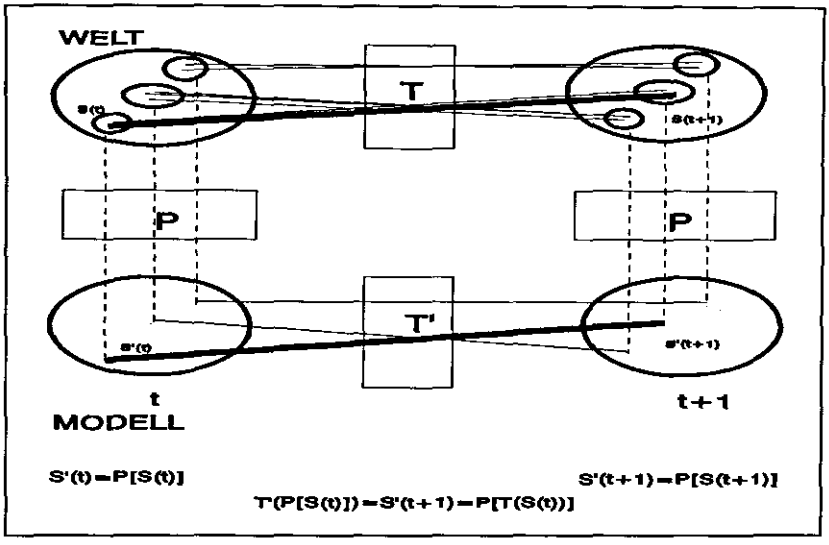


Abb. 5: Mentale Modelle als Homomorphismen nach Holland, Holyoak, Nisbett & Thagard

"Synchrone Regeln" dienen dabei der Klassifikation von Objekten der Welt. Die Regeln bilden Mengen von Objekten auf Äquivalenzklassen im mentalen Modell ab. Eine Kategorisierungsfunktion P wird dabei durch eine Menge von Detektoren definiert, die Objekte bezüglich des Vorhandenseins von Eigenschaften beschreiben. So könnte eine Abbildung P zum Beispiel alle kleinen, bewegten Objekte der Welt in eine Äquivalenzklasse im mentalen Modell abbilden. Das mentale Modell repräsentiert hier also Äquivalenzklassen von Objekten. Ein induktiver Prozeß wird mit Hilfe von Überführungsfunktionen T' beschrieben, die das mentale Modell von einem Zustand in einen anderen abbilden. Die Überführungsfunktionen T' sind dabei eine Menge von "diachronen Regeln". Ein mentales Modell, das mit Hilfe der Überführungsregeln korrekte Vorhersagen über die repräsentierten Äquivalenzklassen liefert, ist dann ein Homomorphismus (siehe Anhang A.2): Die Überführungsfunktion T' der durch P klassifizierten Objekte macht Vorhersagen über die Welt, die mit der Klassifikation der in der Welt durch eine Funktion T transformierten Zustände übereinstimmt. Das mentale Modell ist korrekt, wenn $P1 \circ T = T' \circ P1$.

Lernen bedeutet in diesem Ansatz, die Modifikation und Neubildung von Klassifikations- und Transformationsregeln, so daß homomorphe oder "quasi-homomorphe" Modelle über Zustände und Prozesse der Welt gebildet werden können. Dabei unterscheiden die Autoren drei Lernprozesse, die auftreten können, wenn ein mentales Modell fehlerhaft ist (S. 37 ff.):

In einem frühen Stadium des Lernens, in dem sich weder die Klassifikationsfunktionen noch die Transformationsregeln bewährt haben, werden alternative Kategorisierungen der Umgebung gebildet und davon ausgehend neue Übergangsfunktionen aufgebaut. Haben sich Klassifikations- und Transformationsregeln über mehrere Problemlösesituationen bewährt, werden Modellfehler darauf zurückgeführt, daß die Zustände, die ein Modell zu falschen Vorhersagen bringen, Ausnahmen von der Regel sind. In diesem Fall wird die Mapping-Funktion *P* verfeinert, das heißt, es werden Bedingungen hinzugenommen, so daß die Äquivalenzklassen feiner aufgelöst werden. Entsprechend werden die Transformationsregeln *spezifiziert*. Wenn es kaum mehr gelingt, spezifischere Kategorien zu bilden, oder der Gewinn an Korrektheit den Aufwand einer feineren Spezifizierung nicht mehr rechtfertigt, werden die Transformationsregeln mit Sicherheitswerten belegt.

3.5 Kombination von Repräsentationsformaten zur Modellierung von Problemlösefertigkeiten

Die dargestellten unterschiedlichen Konzeptionen für die Wissensrepräsentation betonen unterschiedliche Aspekte der Leistung des menschlichen Gedächtnisses: Der Schemaansatz stellt die strukturierte Organisation von Wissen in den Vordergrund. Produktionssysteme beschreiben dagegen vor allem Problemlösewissen, das flexibel anwendbar ist und zu effizienten und hochautomatisierten Performanzleistungen führt. Mentale Modelle kombinieren Strukturierung und Flexibilität, wobei die Betonung bei neueren Ansätzen (Holland, Holyoak, Nisbett & Thagard 1986) auf der Integration von Vorwissen und aktuellen Daten *in einer* Problemlösesituation liegt.

Die Debatte über das geeignete Repräsentationsformat zur Modellierung des menschlichen Gedächtnisses ist nicht empirisch entscheidbar. Annahmen über Repräsentationsformate können nur in Zusammenhang mit Prozeßannahmen empirisch geprüft werden, indem menschlicher Performanzleistungen mit vom Modell vorhergesagten *Leistungen verglichen* werden.

Im Anschluß an eine Argumentation von Anderson (1978), die von Simon (Larkin & Simon 1987) wieder aufgegriffen wurde, gehe ich davon aus, daß letztendlich alle im Informationsverarbeitungsansatz verwendeten (symbolischen) Repräsentationsformate ineinander überführbar sind. Die Adäquatheit einer Repräsentation kann immer nur im Zusammenhang mit der Effizienz von Algorithmen (Beschreibung kognitiver Operationen), die auf dieser Repräsentation arbeiten, gemessen werden (zum Aufwand von Algorithmen siehe z.B. Harel 1987). So ist zum Beispiel das römische Zahlensystem in das arabische überführbar und umgekehrt, arithmetische Operationen lassen sich aber wesentlich effizienter im

arabischen Zahlensystem durchführen. Analog kann das Vogel-Schema als Regel dargestellt werden: *Wenn x Federn hat und fliegen kann und in einem Nest wohnt, dann ist es ein* Vogel. Ebenso kann eine Regel in Schemaform angegeben werden, indem der Bedingungsteil in eine Bedingungsleerstelle und der Aktionsteil in eine Aktionsleerstelle notiert wird.

Geht man also davon aus, daß sich Adäquatheit der vorgestellten Repräsentationsformate nur in Bezug auf die Prozesse, die modelliert werden sollen, entscheiden läßt, muß für die vorliegende Arbeit beurteilt werden, wie geeignet diese Ansätze für die Modellierung des induktiven Erwerbs von Problemlösefertigkeiten sind. Dabei geht es letztendlich um den Vergleich von deklarativen und prozeduralen Repräsentationssystemen, also Schemata versus Produktionssysteme. Winograd (1975; vgl. auch Rumelhart & Norman 1981) diskutiert vier Grundmerkmale, in denen sich deklarative und prozedurale Repräsentationsformate unterscheiden:

1) **Flexibilität**: In deklarativen Systemen wird Wissen kontextunabhängig, in Produktionssystemen dagegen kontextabhängig gespeichert. Die Kontextabhängigkeit basiert auf der Abhängigkeit der Aktivation einer Produktionsregel von dem aktuellen Vorhandensein der im Bedingungsteil spezifizierten Informationen im Kurzzeitspeicher.

2) **Lernbarkeit**: Die Modularität von Produktionssystemen ermöglicht es, neue Regeln einfach in die vorhandene Regelmenge zu integrieren, während bei hochstrukturierten deklarativen Strukturen das Einfügen neuer Informationen häufig aufwendige Umstrukturierungsprozesse notwendig macht. Andererseits ist der Transfer von Produktionsregeln in verschiedene Gegenstandsbereiche problematisch, da aus den Bedingungsteilen der Regeln nicht ersichtlich ist, welche Wissensaspekte bereichsspezifisch und welche allgemein sind.

3) **Zugänglichkeit**: Die Strukturiertheit und Kontextunabhängigkeit deklarativer Repräsentationen bedingt, daß Informationen in deklarativen Systemen leicht zugänglich sind, während prozedurales Wissen nur bei Vorhandensein bestimmter kontextueller Information zugänglich ist (Eine Produktionsregel ist aktiviert, wenn ihr Bedingungsteil mit Daten des Kurzzeitgedächtnisses verträglich ist). Die Zugänglichkeit steht in engem Zusammenhang mit der Möglichkeit, Wissen (verbal) zu explizieren. Wissen, das leicht explizierbar ist, sollte also deklarativ, schwer explizierbares Wissen (*tacid knowledge*) prozedural repräsentiert werden.

4) **Effizienz**: Deklarative Systeme haben nur allgemeine Inferenzmechanismen zur Verfügung. Die Kontextabhängigkeit der Produktionsregeln ermöglicht dagegen eine effiziente Anwendung von spezifischen Problemlöseoperatoren.

Ergänzend zu diesen Punkten ist anzumerken, daß der Bedingungsteil von Produktionsregeln mit der Festlegung von Anwendungskontexten auch Problembeschreibungen definiert. Diese Problembeschreibungen können als Repräsentation von Problemklassen aufgefaßt werden. Jedoch geht bei dieser Repräsentation die explizite (hierarchische) Strukturierung der

Problemklassen, so wie sie im Schemaansatz gegeben ist, verloren.

Der induktive Erwerb von Problemlösetechniken umfaßt meiner Meinung nach folgende Aspekte:

- das Auffinden von bereits im Gedächtnis repräsentierten Wissensstrukturen, die für den neuen Problembereich relevant sind;
- die Möglichkeit, Teilaspekte dieses Wissens schnell abzurufen;
- eine strukturierte Repräsentation neuer Informationen;
- die Modifikation bestehender Wissensstrukturen und die Integration neuer Wissensbestandteile ins Gedächtnis.

Problemklassen und -merkmale lassen sich in strukturierter Form in Schemata abbilden, solche Strukturen eignen sich auch für die angemessene Repräsentation neuer problemlöserelevanter Informationen im Kurzzeitgedächtnis. Modifikation und Integration sind dagegen mithilfe von Produktionsregeln effizienter zu modellieren. Daraus folgt, daß Schemata und Produktionsregeln für die Modellierung induktiver Lernprozesse eng miteinander verbunden sein müssen. Diese Anforderungen an Such-, Abruf- und Integrationsprozesse legen eine Kombination der beschriebenen Repräsentationsformate nahe: *"Analysis of a process as complex as analogical problem solving requires a cognitive architecture with diverse knowledge representations that are nevertheless integrated into a coherent whole"* (Holyoak & Thagard 1989, S.243 f.).

Ansätze, die Wissensrepräsentationen als Kombination von Schemata und Produktionsregeln modellieren, sind bisher kaum vorhanden. Anderson (1983, 1986) verwendet beide Arten von Repräsentationssystemen, er nimmt jedoch getrennte Speicher mit unterschiedlichen Prozessen für deklarative und prozedurale Information an. Ein Vorschlag zur Integration beider Repräsentationsformate stammt von Rumelhart und Norman (1981). Die Autoren postulieren Schemata als zentrale Struktur. Diese Schemata können jedoch flexibel als deklarative oder prozedurale Repräsentation genutzt werden. So repräsentiert die Operation

```
define square(:x)
    loop(4,&(forward(:x),right(90)))
```

das Wissen, um ein Quadrat zu zeichnen, und gleichzeitig das Konzept "Quadrat". Induktive Lernprozesse können hier einfach als Generalisierung über analoge Probleme modelliert werden. Jedoch wird bei diesem Repräsentationssystem auf eine explizite Wissensstrukturierung mit Ober- und Unterkonzepten verzichtet.

Holyoak und Thagard (1989) modellieren induktiven Erwerb von Konzepten und Problemlösetechniken auf der Basis einer Integration von Schemata und Produktionsregeln. Grundlegende Wissensstruktur ist dabei die Repräsentation von Konzepten in einer hierarchischen Schemastruktur. Solche Konzeptschemata enthalten unter anderem Leerstellen für Instanzen und zusätzlich Leerstellen für Regeln. Die Regeln werden als Schemata mit Leerstellen für Bedingungs- und Aktionsteil repräsentiert. Ein formaler Ansatz zur Integration von Regeln in eine hierarchische terminologische Repräsentationssprache wurde von Schild (1989) vorgeschlagen.

In Kapitel 6 wird, aufbauend auf dem Vorschlag von Holyoak und Thagard (1989), eine Produktionssystemarchitektur vorgeschlagen, in der eine integrierte Repräsentation von Schemahierarchien und Produktionsregeln als Wissensbasis dient. Aufbauend auf dieser Repräsentation werden Prozesse der Anwendung dieses Wissens auf das Problemlösen zusammen mit induktiven Lernprozessen definiert.

4 Modellvorstellungen über induktive Lernprozesse

Bei der Darstellung der verschiedenen Repräsentationskonzepte im letzten Kapitel wurde bereits auf die Modellierung von Wissenserwerbsprozessen eingegangen. In den vorgestellten Ansätzen wurden aber keine Annahmen über Lernstrategien formuliert. Im folgenden werden zunächst Varianten induktiver Lernstrategien vorgestellt. Danach wird der Ansatz des analogen Problemlösens[3] genauer beschrieben, wobei zwei Ansätze zum analogen Wissenserwerb ausführlich diskutiert werden. Der Ansatz von Anderson (Anderson & Thompson 1989) beschreibt den Erwerb neuer Produktionsregeln als Ergebnis der Anwendung von Beispiellösungen auf Aufgaben. Die Vergleichsprozesse selbst werden dabei jedoch nur oberflächlich behandelt. Carbonell (1983; 1987) stellt dagegen das Problem des Vergleichs von vorhandenen Aufgabenlösungen mit der aktuellen Aufgabenstellung in den Vordergrund. Beide Modelle konzentrieren sich auf den Erwerb von Problemlösetechniken durch Rückgriff auf bereits gelöste Probleme. Am Ende dieses Kapitels wird das System PI von Holyoak und Thagard (1989) vorgestellt, das Konzept- und Fertigkeitserwerb kombiniert.

4.1 Varianten induktiven Lernens

Konzepterwerb

Der größte Teil von Arbeiten zum induktiven Lernen befaßt sich mit dem Erwerb von Konzepten. Aus psychologischer Sicht muß dabei zwischen dem Erwerb natürlicher Konzepte und künstlicher Kategorien unterschieden werden. Künstliche Kategorien zeichnen sich dadurch aus, daß die zu klassifizierenden Objekte mithilfe eindeutig identifizierbarer Merkmale beziehungsweise Merkmalskombinationen (Neisser & Weene 1962) in klar abgrenzbare Kategorien einteilbar sind (Bourne 1974; Bruner, Goodnow & Austin 1956). Bei natürlichen Kategorien ist die Zugehörigkeit von Objekten zu Kategorien dagegen eher gradueller Natur und häufig kontextabhängig (Rosch 1973, Labov 1973). Ansätze der Computersimulation von induktivem Konzepterwerb beziehen sich zwar meist auf die Prototypenkategorie von Rosch (1973), formalisieren aber eher die Prozesse des Erwerbs künstlicher Kategorien.

Die Computermodelle zum Konzepterwerb lassen sich bezüglich der verwendeten Lernstrategien in zwei Gruppen teilen: Eine - die weit größere - Familie von Ansätzen modelliert Konzepterwerb als **Lernen aus Beispielen** (vgl. Unger & Wysotzki *1981*). Dabei werden positive und negative Exemplare einer Kategorie vorgegeben und das System identifiziert die Merkmalskombinationen, die zu einer konsistenten Charakterisierung der Beispiele sowie späterer Testobjekte führt. Korrektheit der Charakterisierung bedeutet dabei, daß die Klassi-

[3]Der hier verwendete Analogiebegriff bezieht sich auf eine Definition von Gentner (1983). Es besteht keine Beziehung zu den Annahmen der analogen Wissensrepräsentation (vgl. Kap. 3.4).

fikation erstens vollständig gelingt (alle positiven Beispiele werden abgedeckt) und zweitens konsistent ist (kein negatives Beispiel wird der Kategorie zugeordnet). Bekannte Ansätze des Konzepterwerbs aus Beispielen sind die Versionenraummethoden von Mitchell (1982) und von Winston (1975), sowie das Konzept der Regel-Verfeinerung von Quinlan (1983). Die drei genannten Ansätze modellieren den Konzepterwerb als *bottom-up*-Prozeß der Merkmalsselektion. Einige Ansätze modellieren den Konzepterwerb aus Beispielen als *top-down*-Prozeß, bei dem vorhandene Wissensstrukturen (z.B. Schemata) den Prozeß der Merkmalsselektion steuern (z.B. Dietterich & Michalski 1981).

Eine zweite Gruppe von Computermodellen zum Konzepterwerb verwendete das **Beobachtungslernen** (*discovery learning*) als Lernstrategie. Während beim Konzepterwerb aus Beispielen lediglich die Merkmale zur Charakterisierung von Objekten erworben werden müssen, wird beim Beobachtungslernen auch die Aggregation von Objekten zu einer Kategorie modelliert. Eine Gruppe von Ansätzen verwendetet Modellierungsmethoden, die auf dem Konzept der Clusteranalyse (Eckes & Roßbach 1980) basieren. Bekannte Realisationen dieses Ansatzes stammen von Michalski & Stepp (1983) sowie von Lebowitz (1987).

Alternativ hierzu modelliert deJong (1986) den Erwerb abstrakter Schemata aufgrund der Vorgabe eines einzigen Beispiels. Anders als die Ansätze des *conceptual clustering* beruht diese Methode (*Explanatory Schema Acquisition*) auf einer umfangreichen Wissensbasis, die unter anderem Regeln enthält, mit denen versucht wird, ein Beispiel durch Erklärung der in diesem enthaltenen allgemeineren Prinzipien zu einem generellen Schema zu abstrahieren. Damit benötigt das Modell von deJong eine umfangreiche Wissensbasis als Voraussetzung für die Anwendbarkeit von Lernmechanismen. Ein sehr früher Ansatz, der ebenfalls eine umfangreiche Wissensbasis zur Inferenz allgemeiner Konzepte benutzt, ist das AM-System von Lenat (1980). AM erwirbt Konzepte im Bereich der Elementar-Mathematik und der Mengentheorie, jedoch ohne die Konzepte für die Lösung von Aufgaben zu verwenden. Die ursprüngliche Wissensbasis von AM besteht aus einer Menge von 115 Konzepten der Mengentheorie. Diese Konzepte sind als Schemata repräsentiert. Mithilfe von Produktionsregeln und Bewertungsfunktionen entwickelt AM ausgehend von dieser Wissensbasis neue Konzepte. So "entdeckte" AM unter anderem das Konzept der Primzahlen, indem es auf das Konzept der ganzen Zahlen die Regel zur Zerlegung von Zahlen anwendete.

Die meisten maschinellen Ansätze zum Konzepterwerb liefern zwar Methoden zur Generierung korrekter Klassifikationen, indem sie diejenigen Merkmale identifizieren, die positive und negative Instanzen einer Kategorie trennen, ob dieses Kriterium der Merkmalsauswahl jedoch genügt, um ein psychologisch plausibles Modell des Konzepterwerbs zu formulieren, ist sehr fraglich. Kriterien für die Auswahl von Merkmalen beim menschlichen Konzepterwerb sind wesentlich vielschichtiger: Entwicklungspsychologische Studien (Carey 1985) belegen, daß Objekte zunächst nach leicht erkennbaren Merkmalen klassifiziert werden

und erst mit zunehmender Expertise nach Merkmalen, die eine sichere Unterscheidung von Objekten ermöglichen. Die Unterscheidung von Objekten wird zudem vor allem nach tätigkeitsbezogenen, funktionalen Kriterien vorgenommen (Klix 1980, S. 80 f., S. 143 ff.). Von den genannten Systemen genügen lediglich die Ansätze von deJong und von Lenat diesen Kriterien.

Relevante Merkmale für die Objektklassifikation sind also solche, die Objekte nach gleichen Verwendungszwecken klassifizieren. Diese Forderung nach einem Bezug des Konzepterwerbs auf Problemlöseaufgaben wurde von mehreren Autoren geäußert (Anderson & Thompson 1989; Carbonell 1983; Rumelhart & Norman 1982). Anderson (Anderson & Thompson 1989, S. 291 f.) kritisiert die Beschränktheit induktiver Systeme zum Konzepterwerb wegen ihrer fehlenden Anbindung an Problemlöseprozesse. Rumelhart und Norman (1981, S. 338 ff.) postulieren, daß konzeptuelles Wissen (*knowledge that*) immer in Problemlösewissen (*knowledge how*) eingebettet ist. Am explizitesten wird diese Kritik von Carbonell (1983, S. 138 f.) formuliert: "*Problem-solving and learning are inalienable aspects of a unified cognitive mechanism*".

Problemlösen wird allgemein definiert als das Finden einer Sequenz von Operationen, die eine Problemstellung in eine Problemlösung überführen (z.B. Hussy 1984, S. 114). Der Erwerb von Problemlösefertigkeiten kann das als das Generieren von Problemlöseoperatoren (Produktionsregeln) oder - in abgeschwächter Form - als das Finden neuer Sequenzen von Operatoren (Regel-Compilierung; Anderson 1986) aufgefaßt werden. Die meisten Ansätze zum Erwerb von Problemlöseregeln ordnen sich in den Produktionssystem-Ansatz ein (Rosenbloom & Newell 1986; Anderson 1986). Als zentrale Lernstrategie wird das Lernen beim Problemlösen (*learning by doing*; Anzai & Simon 1979) verwendet. Die Integration von Ansätzen des Konzepterwerbs in Ansätze zum Erwerb von Problemlösetechniken kann auf zwei Ebenen erfolgen:

Erstens kann man Konzepterwerb als Erwerb von Problemlösekompetenz verstehen. Die Problemlöseaufgabe ist dabei die Einordnung eines neuen Objektes in ein Kategoriensystem. Damit wäre die Definition von Problemlösen als Finden einer Sequenz von Operationen, die eine Problemstellung in eine Problemlösung überführen auf den Fall der Anwendung eines einzigen Operators verallgemeinert. Dies entspricht einem Postulat von Carbonell (1983, S. 138): "*The same learning mechanisms that account for concept formation in declarative domains, operate in acquiring problemsolving skills*".

Zweitens kann man der Auffassung sein, daß beim Erwerb von Problemlösefertigkeiten nicht nur neue Operatoren und Operatorsequenzen induziert werden, sondern gleichzeitig immer eine Kategorisierung der Problemstellungen erfolgt. Diese Annahme ist implizit in den Produktionssystem-Ansätzen zum Erwerb von Problemlösefertigkeiten enthalten: Die Problemklassifikation entspricht hier der Generalisierung oder Spezialisierung der Regel-

bedingungen.

Erwerb von Problemlösefertigkeiten

Während die Ansätze zum Konzepterwerb sich vor allem mit den Lernstrategien Lernen aus Beispielen und Beobachtungslernen befassen, werden bei Ansätzen zum Erwerb von Problemlösefertigkeiten die Lernstrategien *learning by doing* und *learning by analogy* verwendet. Dabei kann ***learning by doing*** (Anzai & Simon 1979) als Gegenstück zum Beobachtungslernens aufgefaßt werden: Beim Beobachtungslernen wird bei der Konfrontation mit verschiedenen Objekten versucht, die Objekte aufgrund gemeinsamer Merkmale zu Kategorien zu aggregieren, beim *learning by doing* wird bei der Konfrontation mit einer Aufgabe versucht, Problemlöseoperatoren zur Lösung dieser Aufgabe zu identifizieren. Die Ansätze des *learning by doing* stellen eine Explikation der bekannten Tatsache "*practice makes perfect*" dar. Umfassende psychologische Modellierungen dieses Lernmechanismus sind die in Kapitel 3.3 dargestellten Ansätze von Anderson (1982) und von Rosenbloom & Newell (1987).

Einige Ansätze zum maschinellen Lernen modellieren den Erwerb von Problemlösekonzepten: Zwei bereits erwähnte Systeme, die sich auf den Erwerb von problemlöserelevanten Konzepten beschränken, sind AM (Lenat 1980) und der Ansatz des *Explanatory Schema Acquistion* (deJong 1986).

Zu den maschinellen Ansätzen zum Erwerb von Problemlösefertigkeiten gehört das gemeinhin als erstes lernendes System bezeichnete Checker's Programm von Samuel (1959). Dieses System erwirbt unter anderem Gewichtungen für Regeln aufgrund einer Bewertungsfunktion für in einem konkreten Spiel ausgeführten Spielzüge.

LEX (Mitchell, Utgoff & Banerji 1983) ist ein System, das das Lösen einfacher symbolischer Integrale durch Lösen von Aufgaben erlernt (*learning by doing*). Dabei werden die Integrier-Aufgaben bei jedem Lerndurchgang generalisiert, so daß eine Menge von abstrahierten Funktionen entsteht, die als Anwendungsbedingungen für Operationen verwendet werden. LEX erwirbt also verallgemeinerte Konzepte von Funktionen - wie Konstante, Sinus-Funktion, Exponentialfunktion etc. - und ordnet diesen Regeln zur Integrierung zu. Konzepterwerb wird hier also als Generalisierung der Anwendungsbedingungen von Produktionen modelliert. Die Operatoren selbst (Aktionsteil der Produktionen) sind dabei vorgegeben.

HACKER (Sussman 1975) ist ein System, das Programmier-Pläne für die Manipulation einer Klötzchen-Welt (STRIPS) erwirbt. Dabei werden, ähnlich wie bei Carbonell (1983), Lösungssequenzen einer Planungsaufgabe gespeichert und verallgemeinert.

Learning by analogy entspricht dem Ansatz des Lernens aus Beispielen. Beim Lernen aus Beispielen werden positive und negative Expemplare eines Konzeptes angeboten, für die eine Klassifikationsregel gefunden werden muß. Beim *learning by analogy* werden bereits gelöste Probleme zum Finden der Problemlösungsregeln für ein neues Problem verwendet. Im folgenden wird auf diese Variante des induktiven Lernens genauer eingegangen.

4.2 Lernen im Ansatz des analogen Problemlösens

In dieser Arbeit ist die Lernstrategie *learning by analogy* als Variante des induktiven Erwerbs von Problemlösefertigkeiten zentral. Zudem wird entsprechend der oben formulierten Annahmen davon ausgegangen, daß beim Erwerb von Problemlösefertigkeiten sowohl Problemlöseregeln, als auch eine Klassifikation von Problemen erworben wird.

Analoges Problemlösen kann nach Novick und Holyoak (1991) als vierstufiger Prozeß beschrieben werden:

- Erstens wird, ausgelöst von einer aktuellen Problemstellung, im Gedächtnis nach einem ähnlichen, bereits gelösten Problem, gesucht (*retrieval*).
- In einem zweiten Schritt wird das gelöste Problem mit der aktuellen Problemstellung verglichen (*mapping*). Ergebnis dieses Vergleichsprozesses ist die Identifikation der Teile, die zwischen altem und neuem Problem identisch sind und entsprechend der Teile, für die die alte Problemlösung an das neue Problem angepaßt werden muß.
- Die Anpassung des alten Lösungsweges an das neue Problem (*adaptation*) bildet die dritte Stufe des Prozesses.
- Die vierte und letzte Stufe dieses Prozesses ist der durch die Problemlösung resultierende Wissenserwerb (*learning*).

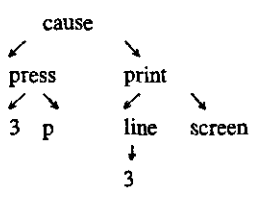
Novick und Holyoak (1991) postulieren hier, daß die Vergleichs- und Anpassungsprozesse zur Generalisierung der vorhandenen Wissensstrukturen zu abstrakten Problemschemata führen. Sie machen jedoch keine detaillierteren Angaben über die postulierten Teilprozesse. Im folgenden werden zwei Ansätze beschrieben, die als mögliche Konkretisierungen der Annahmen von Novick und Holyoak verstanden werden können.

Der erste Teil des Prozesses, der Abruf eines ähnlichen Problems, wird im Rahmen dieser Arbeit nicht problematisiert. Ansätze, die sich zentral mit diesem Aspekt befassen, sind zum Beispiel im Bereich das fallbasierten Schließens (Schank 1983; Kolodner 1984) zu finden. Die im folgenden Abschnitt dargestellte Arbeit von Anderson und Thompson (1989) befaßt sich zentral mit dem Einfluß der Art des gegebenen analogen Problems auf den resultierenden Wissenserwerb. Carbonell (1983; 1986) konzentriert sich dagegen auf die Beschreibung der Vergleichs- und Anpassungsprozesse. Beide Ansätze konzentrieren sich auf den Erwerb

von Problemlösetechniken.

Zwei weitere Ansätze zum analogen Wissenserwerb stammen von Rumelhart und Norman (1981) und von Gentner (1983). Gentner's *Structure-Mapping*-Ansatz beschreibt analogen Transfer als Vergleich von Objekt-Relationen zweier Gegenstandsbereiche. Grundlegende Annahmen dieses Ansatz sind in allen Modellen zum analogen Transfer zu finden. Da Gentner keine Annahmen über Lernprozesse formuliert und sich zudem auf *between-domain* Analogien konzentriert, wird auf diesen Ansatz jedoch nicht weiter eingegangen.

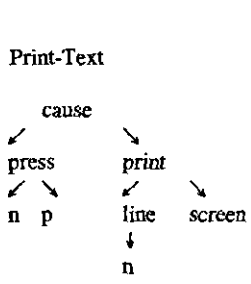
Rumelhart und Norman (1981) schlagen einen Ansatz vor, bei dem der Erwerb problemlöserelevante Konzepte als analoger Schemaerwerb beschrieben wird. Dieser Ansatz stellt eine Modifikation des in Kapitel 3.2 dargestellten Modells der Schemainduktion dar. Die Autoren modellieren Schemata als Operationen, die je nach Aufgabenanforderung als Konzeptrepräsentation oder als Regel zur Generierung von Problemlösungen aufgefaßt werden können. Diese Schemata lassen sich flexibel als semantische Netze oder als LISP-ähnliche Funktionen notieren. Der Ansatz wurde beispielhaft auf die Modellierung von Wissenserwerb beim Umgang mit einem Texteditor angewendet. Haben Personen beim Umgang mit einem Texteditor zum Beispiel gelernt, daß das Drucken der dritten Zeile eines Textes mit dem Befehl "3p" erfolgt, so kann dieses Wissen in folgender Form repräsentiert werden:



Dies entspricht der function-form Schemastruktur bei Anderson (Anderson & Thompson 1989; vgl. Kap. 4.3):

structure print-text-3
is-a : *editor-command*
function : (print line 3 on screen)
form : (press 3 p)

Will eine Person nun die fünfte Zeile des Textes drucken, so gelingt dies durch folgenden Analogieschluß: *print-text-5 is-like print-text-3 with 5 for 3*. Ein erfolgreicher Analogieschluß führt dann zur Inferenz eines neuen generalisierten print-text-Schemas:



Der Mechanismus des analogen *Transfers* wird damit von Rumelhart und Norman wie bei Anderson modelliert. Die Ansätze unterscheiden sich jedoch in der Modellierung des induzierten Wissens: Bei Anderson führt die Induktion zum Aufbau einer Produktionsregel, bei Rumelhart und Norman zur Schemainduktion (hier Prozeduren ohne spezifische Anwendungsbedingungen). Damit entfällt bei beiden Ansätzen die explizite Strukturierung der Wissensbasis (vgl. Diskussion in Kap. 3.5).

4.3 Analoges Problemlösen in einer Erweiterung von Anderson's ACT*-Theorie

In Kapitel 3.3 wurden die Annahmen der ACT*-Theorie (*A*daptive *C*ontrol of *T*hought) zum Wissenserwerb bereits kurz dargestellt. Im folgenden werden sie etwas mehr ausgeführt, um dann auf eine Modifikation der ACT*-Theorie, die analogen Erwerb von Problemlösetechniken modelliert, genauer vorzustellen.

Lernen wird in ACT* als *Kompilation* deklarativen Wissens und als Optimierung von Produktionsregeln modelliert. Anderson (1982) schlug zwei ausführliche Modelle für den Erwerb von Beweistechniken in der Geometrie und von Programmiertechniken in LISP (vgl. Winston & Horn 1989) vor. Letzteres Modell mündete in der Entwicklung eines Intelligenten Tutoriellen Systems (Anderson, Conrad & Corbett 1989).

Anderson nimmt an, daß Wissen über neue Problembereiche zunächst deklarativ, aufgrund von vorgegebener Information, kodiert wird. Eine solche Information kann zum Beispiel sein: "CAR liefert in LISP das erste Element einer Liste". Die Anwendung deklarativen Wissens auf Beispielprobleme erfolgt mit Hilfe allgemeiner Problemlösemethoden, die in Form von Produktionsregeln vorliegen ("Wenn das Ziel ist eine Funktion in LISP zu schreiben, dann setze als Unterziel einen LISP-Ausdruck zu finden, der diese Funktion erfüllt"). Werden solche allgemeinen Regeln in einem Problembereich häufig in einer festen Sequenz erfolgreich angewendet, so werden diese Regeln zu einer komplexen Regel mit konjunktiven Bedingungen und Aktionen verknüpft (Kombination). Durch häufige Paarung allgemeiner Regeln mit deklarativem Wissen kann das deklarative Wissen mit der Regel dauerhaft

verknüpft werden, so daß eine neue Produktionsregel entsteht (Prozeduralisierung). Kombination und Prozeduralisierung sind zwei Methoden der Wissenskompilierung. Sind auf diese Art neue Produktionsregeln entstanden so werden sie bei der Anwendung auf weitere Probleme optimiert. Für die Regeloptimierung nimmt Anderson drei Prozesse an: Generalisierung, Diskrimination und Verstärkung. Generalisierung meint die Ersetzung von Konstanten durch Variablen, Diskrimination schränkt die Anwendungsbedingungen einer Regel ein, indem neue Bedingungen hinzugenommen werden. Verstärkung meint die Erhöhung des einer Produktionsregel zugeordneten Stärkewertes nach erfolgreicher Anwendung dieser Regel. Damit werden Regeln, für die es keine Anwendungsbedingungen gibt oder solche, die sich bei Anwendungen nicht bewähren von erfolgreichen Regeln verdrängt. Die Annahmen von Anderson stimmen gut mit empirischen Befunden überein (Anderson 1982). Sie erklären die Zunahme der Problemlösungsgeschwindigkeit und Korrektheit im Laufe des Lernprozesses. Anwendungen deklarativen Wissens benötigen aufwendige Suchprozesse und führen zu hohen Belastungen des Arbeitsgedächtnisses, nach der Kompilierung wird eine Suche im deklarativen Speicher unnötig, die Optimierung führt zu Regeln, die auf spezifische Probleme direkt anwendbar sind (Diskrimination) und ermöglicht die Übertragung von Gelerntem auf neue Probleme (Generalisierung).

Ein Mangel des Modells von Anderson ist, daß es nur mit großem Aufwand an Produktionsregeln möglich ist, die intensive Nutzung von Beispielen bei Anfängern in einem Problembereich zu simulieren. Damit verbundene Probleme sind, daß die starre Abarbeitung von Zielen die häufig beobachtbare Flexibilität von Lösungsstrategien nicht abbildet (van Lehn 1983) und daß Generalisierung und Diskriminierung als automatische Prozesse modelliert wurden, die damit weder fehleranfällig sind, noch einer bewußten Kontrolle des Lernenden unterliegen (Anderson 1986).

Integration von Analogieschlüssen bei Anderson

Aus diesem Grund modifizierte Anderson seine ACT*-Theorie. Zentrale Neuerung ist die Wissensrepräsentation in Form von Schemahierarchien. Durch Anwendung von Regeln auf diese Strukturen kann das Lernen durch Analogien plausibel modelliert werden (Anderson & Thompson 1989). Zentrale Leerstellen dieser Schemata sind *is-a*, *function* und *form*. Dabei gibt *function* an, welche Funktion das Schema erfüllt, *form* gibt eine Realisation in einem konkreten Kontext (z.B. LISP-Systax) an. *Function* und *form* beschreiben also Semantik und Syntax einer Programmkomponente. So kann etwa das Auslesen des ersten Elements einer Liste (Parameter *lis*) in LISP durch folgendes Schema repräsentiert werden (Anderson & Thompson 1989, S. 270f.):

structure_1
is-a : function-call
function : (extract first lis)
form : (list car lis).

Diese Struktur wird nun bei der Auseinandersetzung eines Lernenden mit Beispielaufgaben mehrfach instantiiert:
Es sei einem Lernenden das Beispiel *CAR'(p q r s)* liefert *p* gegeben:

structure_y
is-a : function-call
function : (extract-first(p q r s))
form : (list car '(p q r s)).

Der Lernende erhält nun die Aufgabe das erste Element der Liste *(a b c)* zurückzugeben:

structure_x
is-a : function-call
function : (extract first (a b c))
form : ???.

Die Aufgabe kann nun mittels eines Analogieschlusses gelöst werden: Das Verhältnis (:) der *function* von *structure y* zur Form von *structure y* ist analog (::) zum Verhältnis der *function* von *structure x* zu der zu erschließenden *form* von *structure x*:

function(structure y)	: form(structure y)	::	function(structure x)	: ???
(extract-first(p q r s))	: (list car '(p q r s))	::	(extract-first(a b c))	: ???.

Die Form-Leerstelle von *structure_x* kann nun durch folgenden *Mapping*-Prozeß erschlossen werden:

list	-> list
car	-> car
'	-> '
(p q r s)	-> (a b c).

Aufgrund eines erfolgreichen Analogieschlusses kann die Regel *(list car '(p q r s)) -> (extract-first (p q r s))* generalisiert werden zu: *(list car 'x) -> (extract-first x)*.

Das revidierte Modell erklärt Lernprozesse beim Problemlösen mit Beispielen sehr überzeugend. Allerdings wird im Rahmen des Modells nicht geklärt, wie die Schemastrukturen gebildet werden und die Prozesse des Vergleichs und der Anpassung von Beispiel und

Aufgabe werden wenig ausführlich behandelt.

4.4 Analoges Problemlösen als Anpassung von Problemlöseplänen bei Carbonell

Carbonell (1983) sieht analoges Problemlösen als den Abruf von Lösungsplänen für ähnliche Probleme und deren Anpassung an die neue Problemstellung. Eine reaktive Umgebung gibt Aufschluß über Erfolg oder Mißerfolg des angewendeten modifizierten Lösungsplans. Erfolgreiche Anwendungen führen dann zur Generalisierung des urprünglichen und des modifizierten Lösungsplans. Partiell erfolgreiche Anwendung führt zur Diskrimination von Lösungsplänen durch Hinzunahme von Anwendungsbedingungen.

Carbonell modelliert diese Annahmen mit einer Erweiterung der Mittel-Ziel-Analyse (vgl. Kap. 3.3). In der Mittel-Ziel-Analyse wird ein aktuell zu lösendes Problem als Problemraum repräsentiert. Ein Problemraum besteht aus:

- einer Menge P möglicher Problemzustände
- ein als Anfangszustand ausgezeichneter Zustand $I \in P$
- ein als Endzustand ausgezeichneter Zustand $G \in P$
- einer Menge von Operatoren O: P -> P mit bekannten Anwendungsbedingungen, die einen Problemzustand in einen anderen transformieren
- eine Differenzfunktion D: P x P -> R, die den Unterschied zwischen aktuellem Problemzustand und einem anderen Zustand oder speziell dem Zielzustand angibt
- eine Methode, die Operatoren als Funktion der Differenz, die sie reduzieren, zu kennzeichnen (z.B. eine Unterschiedstabelle)
- eine Menge von globalen *constraints* C, die im folgenden als Prädikate auf Teilabschnitten eines Lösungspfads, also auf Operator-Sequenzen definiert sind: $C: O^m \rightarrow \{0,1\}$.

Der Problemlöseprozeß kann nun als Suche einer Operatorfolge, die unter Einhaltung aller *Constraints*, den Anfangszustand in einen Zielzustand transformiert beschrieben werden. Dabei wird als Suchstrategie die Mittel-Ziel-Analyse eingesetzt:

- Vergleiche aktuellen Zustand mit dem Zielzustand
- Wähle einen Operator, der die Differenz zum Zielzustand reduziert
- Wende den Operator an, wenn es möglich ist; wenn es nicht möglich ist, speichere den aktuellen Zustand und wende die Mittel-Ziel-Analyse auf das Teilproblem an, den Zustand zu erreichen, der den Anwendungsbedingungen des Operators genügt
- wenn ein Teilproblem gelöst ist, rufe den gespeicherten Zustand ab und arbeite an der Problemlösung weiter.

Eine ausführliche Darstellung der Mittel-Ziel-Analyse findet sich in Newell und Simon (1972, S. 414ff.).

Ein psychologisch unplausibler Aspekt der Mittel-Ziel-Analyse als Problemlösestrategie ist, daß ein informationsverarbeitendes System für jedes neu dargebotene Problem den gesamten Lösungsweg neu berechnet, Erfahrung mit bereits gelösten Problemen wird nicht für das Finden neuer Lösungen genutzt. Diese Beschränkung wird durch eine Erweiterung der Mittel-Ziel-Analyse durch Carbonell (1983) aufgehoben. Carbonell führt einen *Reminding*-Prozeß ein, bei dem für ein aktuelles Problem geprüft wird, ob Lösungswege für ähnliche Probleme im Speicher vorhanden sind. Die gespeicherten Lösungswege werden bezüglich folgender Aspekte mit dem neuen Problem verglichen:

- Der Anfangszustand des gelösten Problems mit dem aktuellen Anfangszustand
- der Zielzustand des gelösten Problems mit dem aktuellen Zielzustand
- die Constraints bei der gespeicherten Problemlösung und mit den aktuellen Problemlöse-Constraints
- Der Anteil von Operator-Anwendungsbedingungen der gespeicherten Lösung, die in der neuen Problemsituation erfüllt sind.

Diese Aspekte werden in einem Ähnlichkeitsmaß verrechnet und geben damit den Grad der Anwendbarkeit einer Problemlösung auf ein neues Problem an. Damit ist die Phase des Abrufs (*retrieval*) zusammen mit *Mapping* im analogen Problemlöseprozeß beschrieben.

In einem nächsten Schritt muß nun der Lösungsweg des gelösten Problems so transformiert werden, daß ein Lösungsweg für das aktuelle Problem entsteht, wobei beachtet werden muß, welche Operatoren, Anwendungsbedingungen und *Constraints* beim neuen Problem gelten. Dies entspricht dem Adaptations-Prozeß zusammen mit einem weiteren Aspekt des Vergleichs (*mapping*) von aktuellem Problem und analogem Problem. Carbonell definiert die Transformation des Lösungswegs als eigenen Problemlöseprozeß. Dieses Transformationsproblem wird in einem eigenen Problemraum, dem T-Problemraum repräsentiert:

- Zustände im T-Problemraum sind mögliche Problemlösungen, also eine Menge von Operatorsequenzen $P_T = \{O: P \rightarrow P\}$, wobei O solche Operatoren sind, die beim aktuellen Problem zur Verfügung stehen und P solche Zustände, die beim aktuellen Problem mit den Operatoren, unter Berücksichtigung der Constraints erreicht werden können.
- Anfangszustand ist eine Problemlösung, die dem aktuellen Problem ähnlich ist, $I_T \in P_T$
- Zielzustand ist eine Lösungssequenz für das aktuelle Problem, die den aktuellen Constraints genügt: $G_T \in P_T$
- eine Menge von T-Operatoren, die eine Lösungssequenz in eine andere transformieren: O_T: $P_T \rightarrow P_T$; solche T-Operatoren sind zum Beispiel: Einfügen eines Operators in die Sequenz, Löschen eines Operators aus der Sequenz, Ausschneiden einer Teilsequenz aus einem Lösungspfad, Verbinden von zwei (Teil-)Lösungssequenzen, Ersetzung von Operatoren etc. Eine Ausführliche Beschreibung der T-Operatoren gibt Carbonell (1983, S. 144 f.)

- eine Differenzmetrik, die den Unterschied zwischen der aktuellen Problemlösungssequenz und der zu erreichenden Lösungssequenz angibt. Das Differenzmaß besteht aus 4 Differenzfunktionen, eine Lösung für das aktuelle Problem ist gefunden, wenn alle 4 Funktionen den Wert 0 liefern:

$$D_T = <D_P(I_{ret},I_{act}),\ D_P(G_{ret},G_{act}),\ D_C(C_{ret},C_{act}),D_{SOL}(SOL_{ret},SOL_{act})>$$

mit

ret	=	Index für gelöstes Problem bzw. Transformationen des gelösten Problems
act	=	Index für das aktuelle Problem, für das eine Lösung gesucht wird
I	=	Anfangszustand
G	=	Zielzustand
C	=	Constraints
SOL_{ret}	=	Lösungssequenz
SOL_{act}	=	aktuelle Problemspezifikation
D_P	=	Unterschiedsfunktion zwischen Problemzuständen (die Differenzfunktion der originalen Mittel-Ziel-Analyse)
D_C	=	Unterschiedsfunktion zwischen Constraints
D_{SOL}	=	Maß der Anwendbarkeit der alten Problemlösung auf das neue Problem als Anteil der Operatoren in der alten Lösungssequenz, deren Anwendungsbedingungen im neuen Problem nicht erfüllbar sind (also ist DSOL komplentär zu dem für das Problem-Retieval verwendeten Ähnlichkeitsmaß)

- eine Unterschiedstabelle zur Indexierung der T-Operatoren, analog zur Unterschiedstabelle in der originalen Mittel-Ziel-Analyse
- für die Mittel-Ziel-Analyse auf dem T-Problemraum existieren keine *Constraints*.

Zur Veranschaulichung der Mittel-Ziel-Analyse für analoges Problemlösen wird im folgenden ein kurzes Beispiel gegeben (vgl. Carbonell 1983, S.149 ff.): Ein Informatikstudent hat folgendes Problem bereits gelöst: Zeige, daß das Produkt zweier gerader Zahlen gerade ist. *Dabei hat er als Beweisidee verwendet*, daß eine gerade Zahl ganzzahlig durch zwei teilbar und als *2*x* repräsentierbar ist. Er erhält nun die Aufgabe zu zeigen, daß das Produkt zweier ungerader Zahlen ungerade ist.

Anfangszustand des gelösten Problems: Zeige: (x gerade und y gerade) --> (x*y gerade)

Zielzustand des gelösten Problems: *x*y gerade, mit (x gerade und y gerade)*

Lösungssequenz (Abfolge von Operatoren):

1) Abruf der Definition "gerade Zahl": Eine Zahl ist gerade, wenn sie ganzzahlig durch zwei teilbar ist
2) Schreibe einen Ausdruck, der eine gerade Zahl repräsentiert, also z.B. 2 * beliebige ganze Zahl
3) Multipliziere zwei Ausdrücke für gerade Zahlen: 2N * 2M = 4NM
4) Wende die Repräsentation gerader Zahlen auf 4NM an, also 4NM = 2 * 2NM.

Anfangszustand des neuen Problems: Zeige: (x ungerade und y ungerade) --> (x*y ungerade)

Zielzustand des neuen Problems: x*y ungerade, mit (x ungerade und y ungerade)

Das alte und das aktuelle Problem unterscheiden sich nur darin, daß es sich einmal um gerade und einmal um ungerade Zahlenn handelt. Der Lösungsweg des gelösten Problems muß also nur geringfügig durch T-Operatoren verändert werden:

Transformierte Lösungssequenz:
1) Abruf der Definition "ungerade Zahl": Eine Zahl ist ungerade, wenn sie nicht ganzzahlig durch zwei teilbar ist
2) Schreibe einen Ausdruck, der eine ungerade Zahl repräsentiert, also z.B. 2 * beliebige ganze Zahl + 1
3) Multipliziere zwei Ausdrücke für gerade Zahlen: (2N+1) * (2M+1) = 4NM + 2N + 2M + 1
4) Wende die Repräsentation gerader Zahlen auf 4NM + 2N + 2M + 1 an, also 4NM + 2N + 2M + 1 = 2*(2NM+N+M) + 1.

Carbonell's Ansatz der Erweiterung der Mittel-Ziel-Analyse auf Transformationsprobleme läßt sich rekursiv anwenden: auch das Finden einer Transformation von einer gegebenen Problemlösung zur Lösung des neuen Problems kann als *T-Problemlösung gespeichert* werden, ein zu einem aktuellen Transformationsproblem ähnliches Transformationsproblem kann dann durch Adaptation der vorhandenen Transformation gelöst werden. Auf diese Art kann modelliert werden, daß eine Person mit zunehmender Problemlöseerfahrung neue Probleme schneller korrekt lösen kann, indem sie über eine Vielzahl von (verallgemeinerten) Transformationen von Problemen eines Bereichs verfügt. Carbonell (1986) erweitert seinen Ansatz auf die Verwendung mehrerer bereits gespeicherter Problemlösungen. Zentral an Carbonell's Ansatz ist der Gedanke der Rekonstruktion von Lösungsplänen, um sie durch analogen Transfer auf neue Probleme anzuwenden. Damit kann der Ansatz als eine Anwendung schematheoretischer Annahmen (Schank 1982; Minsky 1975) auf das analoge Problemlösen verstanden werden.

Carbonell (1986, S. 377 f.) betont auch die Anwendbarkeit dieses Ansatzes auf das Lösen von Programmierproblemen sowohl zur Modellierung menschlicher Programmierprozesse als auch im Bereich automatisches Programmieren. Man kann davon ausgehen, daß menschliche Programmierer bei der Implementierung einer Programmes auf bereits gelöste ähnliche Probleme zurückgreifen. So wird ein Programmierer, der Quicksort in LISP implementieren soll, und dieses Problem bereits in PASCAL gelöst hat, sicher auf die PASCAL-Lösung zurückgreifen und sie an die Erfordernisse von LISP anpassen. Auch für das automatische Programmieren könnte es sinnvoller sein, das übliche Vorgehen der schrittweisen Verfeinerung abstakter Spezifikationen aufzugeben (Barstow 1977) und stattdessen Verfahren des analogen Problemlösens einzusetzen (z.B. Dershowitz 1986; Manna & Waldinger 1975).

4.5 Kombination von Konzept- und Fertigkeitserwerb

Ein System, das Konzepterwerb und Regelerwerb zur Modellierung analoger Problemlöseprozesse integriert, ist PI von Holyoak und Thagard (*1989*; vgl. Kap. 3.5). Die Autoren kombinieren Schemata und Regeln in einer Wissensstruktur. Grundlegende Wissensstruktur ist dabei die Repräsentation von Konzepten in einer hierarchischen Schemastruktur. Solche Konzeptschemata enthalten unter anderem Leerstellen für Instanzen und zusätzlich Leerstellen für Regeln. Die Regeln werden als Schemata mit Leerstellen für Bedingungs- und Aktionsteil repräsentiert. Eine aktuelle Problemstellung wird in einem Problemschema mit den Leerstellen Startbedingungen und Ziele repräsentiert. Die im Problem vorkommenden Konzepte werden aktiviert, wobei die Instanzenleerstellen mit konkreten Objekten belegt werden. Diese Konzeptinstanzen werden als aktive *Messages* betrachtet. *Messages* und Regeln aktiver Konzepte stehen für den Problemlösungsprozeß zur Verfügung. Regeln, deren Bedingungsteil mit den aktiven Messages übereinstimmen (*matching*), werden ausgeführt und führen damit zur Aktivation neuer Konzepte. Zusätzlich werden aktive Konzepte miteinander verglichen (*mapping*), diese Vergleichsprozesse führen zur Induktion allgemeinerer Konzepte. Die Architektur von PI entspricht einem Produktionssystem: Regeln sind mit Stärkewerten versehen, die bei erfolgreicher Regelanwendung inkrementiert werden. Der von Anderson eingeführte Aktivationsausbreitungsmechanismus im deklarativen Speicher wird hier über Zuweisung von Aktivationswerten an die Konzepte realisiert.

PI modelliert sowohl Konzepterwerb als Generalisierung über Beispiele, als auch Wissenszuwachs durch analoges Problemlösen. So konnte PI zum Beispiel erfolgreich zur Simulation der Lösungsprozesse bei Duncker's (1945) Bestrahlungsproblem eingesetzt. Die Autoren formalisieren den Analogieschluß als Morphismen zwischen zwei mentalen Modellen (Holland, Holyoak, Nisbett & Thagard 1986; Kap. 10). Dabei werden sowohl Beispiel, als auch Aufgabe als Homomorphismen dargestellt (vgl. Kap. 3.5, Abb. 5), wobei das Modell der Aufgabe mittels analogem Mapping P^* von der Aufgabe zum Beispiel konstruiert wird. Diese Idee wird in Kapitel 6 aufgegriffen und für die Modellierung des Erwerbs rekursiver Programmiertechniken konkretisiert.

Sowohl Anderson (Anderson & Thompson 1989) als auch Carbonell (*1983*) verwenden Schemastrukturen zur Repräsentation von Problemlösewissen. Anderson repräsentiert Wissen über Problemlösungen (LISP-Funktionen) in Funktion und Formschemata, Carbonell repräsentiert vollständige Problemlösepläne bereits gelöster Probleme. Damit repräsentieren beide Autoren konkretes Problemwissen in Schema/Plan-Format.

Andere Autoren gehen dagegen davon aus, daß abstrahierte Problemlösungen (Vorberg & Goebel 1991) oder Problemlösepläne (Spohrer, Soloway & Pope 1989) zur Lösung neuer Probleme herangezogen werden Diese Autoren machen aber keine Annahmen über den

Erwerb solcher abstrakter Schemata.

Konkrete Beispiele und abstrakte Pläne lassen sich in einer Wissensstruktur kombinieren (vgl. Carbonell 1983; Abb. 14-1). Anderson (1982) modelliert Lernen als Automationsprozess, bei dem ausgehend von spezifischem Problemlösewissen zunehmend komplexere Produktionsregeln gebildet werden, die zu immer höherer Effizienz der Wissensanwendung führen. Diese Annahmen können auf Schemahierarchien übertragen werden, in dem ausgehend von der Repräsentation konkreter Beispiele abstraktere Schemata aufgebaut werden, die als Problemlösungspläne dienen. Diese Idee wird ebenfalls in Kapitel 6 konkretisiert.

5 Modellvorstellungen über den Erwerb von rekursiven Programmiertechniken

In diesem Kapitel wird der Gegenstandsbereich induktiven Lernens, der dieser Arbeit zugrundegelegt ist, dargestellt. Als zu erwerbende Problemlösetechnik wird das Programmieren rekursiver Funktionen in einer funktionalen Sprache betrachtet. Im folgenden wird zunächst begründet, warum der Erwerb von Programmiertechniken als Lerngegenstand fokussiert wird. Danach wird das Prinzip der Rekursion erläutert und eine selbstentwickelte funktionale Programmiersprache vorgestellt. Die letzten beiden Abschnitte diskutieren mögliche Wissensstrukturen von Programmierexperten. Solche Expertenmodelle können als Zielvorstellungen für Modelle des Wissenserwerbs verwendet werden.

5.1 Programmieren als komplexe kognitive Fertigkeit

Als Gegenstandsbereich für die Betrachtung des induktiven Erwerbs von Problemlösefertigkeiten wird in dieser Arbeit das Erstellen von Computerprogrammen in einer funktionalen Sprache zugrundegelegt. Die Wahl dieses Lerngegenstands ist aus mehreren Gründen zweckmäßig: (1) Programmieren ist eine Fertigkeit, die zunehmend größere Bedeutung in schulischer, universitärer und beruflicher Ausbildung gewinnt (Eyferth 1988). Eine Analyse der Lernprozesse kann somit zur Erstellung von Curricula und Lehrstrategien für die effiziente Vermittlung von Programmierkenntnissen beitragen. (2) Programmierfertigkeiten sind relativ unabhängig von Vorkenntnissen aus anderen Bereichen (vgl. Anderson, Farrell & Sauers 1984), man kann allenfalls vermuten, daß eine allgemeine Begabung zum formalen und analytischen Denken den Lernprozeß erleichtert (vgl. Corno & Haertel 1982). Aus diesem Grund kann das Erlernen von Programmiertechniken gut ohne Bezug zu Wissensstrukturen aus anderen Bereichen modelliert werden und individuelle Voraussetzungsunterschiede sind bei empirischen Untersuchungen einfach zu kontrollieren. (3) Zudem ist das Erstellen von Programmen eine hinreichend komplexe Problemstellung. Programmier-Anfänger müssen grundlegend neue Problemlösetechniken erwerben, die es ihnen ermöglichen, aus einer Aufgabenstellung ein syntaktisch und semantisch korrektes Programm zu generieren. Gleichzeitig gibt es jedoch eindeutige Kriterien für die Korrektheit von Programmen. (4) Gerade beim Programmieren ist die Verwendung von Beispielen zur Lösung einer neuen Aufgabe die natürlichste Methode: Programmieranfänger erstellen Programme, indem sie versuchen, vorhandene Programme mit ähnlicher Problemstruktur an die neuen Erfordernisse anzupassen (Pirolli & Anderson 1985). Erfahrene Programmierer entwickeln in den seltensten Fällen grundlegend neue Programme, stattdessen modifizieren sie (im Gedächtnis und/oder auf dem Computer) gespeicherte Programme (Dershowitz 1986). Das Prinzip des Programmierens mit Beispielen scheint auch bei Ansätzen des automatischen Programmie-

rens erfolgversprechend (Manna & Waldinger 1975; Dershowitz & Manna 1977; Carbonell 1983; Amarel 1986).

Der Weg vom Programmieranfänger zum Experten ist durch den Aufbau von Wissensstrukturen beschreibbar, in denen insbesondere folgende Inhalte repräsentiert werden: eine immer größere Menge von selbständig erstellten Programmen, zusammen mit abstrakten Schemata verschiedener Programmierprinzipien; Techniken zum Vergleich der Struktur verschiedener Programm-Anforderungen; und Techniken zur Modifikation (Anpassung) von repräsentierten Programmen an neue Anforderungen. Dieser Weg ist allerdings wesentlich langwieriger und stellt offensichtlich wesentlich höhere kognitive Anforderungen als der Erwerb anderer Problemlösetechniken. Ein wesentlicher Grund dafür ist sicherlich, daß ein Programmieranfänger kaum auf bereits im Gedächtnis gespeichertes Wissen zurückgreifen kann, um die Anforderungen der Programmerstellung zu bewältigen.

Die kognitiven Anforderungen an Programmieranfänger lassen sich in zwei Klassen teilen: Erstens muß eine neue Sprache mit all ihren Syntaxregeln erlernt werden und zweitens, die weitaus größere Hürde, müssen neue Konzepte zusammen mit Implikationen für ihren Verwendungszusammenhang erworben werden. Solche Programmierkonzepte sind zum Beispiel Variablen und Schleifen (Soloway, Bonar & Ehrlich 1989) im Rahmen imperativer Programmiersprachen (wie MODULA oder PASCAL: Wirth 1983) oder Rekursion im Rahmen funktionaler Sprachen (wie HOPE: Burstall, MacQueen & Sanella 1980; LISP: McCarthy, Abrahams, Edwards, Hart & Levin 1962 oder ML: Milner, Tofte & Harper 1990).

Gerade das Konzept der Rekursion wird durch die zunehmende Bedeutung deklarativer Sprachen (logische und funktionale Sprachen; vgl. Beckstein 1993) relevant. Programmieranfänger haben jedoch erfahrungsgemäß große Schwierigkeiten mit dem Erwerb dieses Konzeptes (Anderson, Farrell & Sauers 1984; Pirolli 1986; Schömann 1991; Vorberg & Goebel 1991). Eine Ursache ist wohl, daß rekursive Strukturen außerhalb der Mathematik und Informatik kaum vorkommen.

Die oben beschriebene Sichtweise von induktivem Lernen als Aufbau abstrakter Schemata (z.B. Novick & Holyoak 1991) läßt sich meiner Meinung nach im Programmierbereich besonders gut auf das Konzept der Rekursion anwenden. Während Expertenlösungen bei imperativen Sprachen hoch variabel sind, sind rekursive Programme in funktionalen Sprachen sehr uniform (z.B. Haak, Hahn & Wagner 1989). Die höhere Abstraktheit funktionaler Sprachen ermöglicht es, Programme als knapp formulierte Problemstellungen zu formulieren. Schon mit einer sehr eingeschränkten Grundsyntax liefern solche Sprachen mächtige Programmierwerkzeuge.

Auf das Konzept der Rekursion und auf funktionale Sprachen wird in den beiden folgenden Abschnitten eingegangen.

5.2 Das Konzept der Rekursion

Zum besseren Verständnis des Lerngegenstands wird das Konzept der Rekursion kurz eingeführt (vgl. z.B. Abelson & Sussman 1985; Field & Harrison 1988; Loeckx & Sieber 1984). Am anschaulichsten wird das Konzept am Beispiel rekursiver Definitionen mathematischer Funktionen, wie etwa der Fakultätsfunktion. Die Fakultät einer natürlichen Zahl (*n!*) ist das Produkt der Zahl mit allen von null verschiedenen Vorgängern; die Fakultät von null ist als als eins definiert. So ist zum Beispiel *5!* gleich *5*4*3*2*1*, also gleich *120*. Die allgemeine nicht-rekursive Definition des Problems ist:

$$n! = \begin{cases} 1 & ; \textit{für } n=0 \\ \prod_{i=1}^{n} i; & \textit{für } n>0 \end{cases}$$

Die entsprechende rekursive Definition lautet:

$$n! = \begin{cases} 1 & ; \textit{für } n=0 \\ n*(n-1)!; & \textit{für } n>0 \end{cases}$$

Wie bei der nicht-rekursiven Definition, wird der Funktionswert für das kleinstmögliche Argument der Fakultätsfunktion direkt angegeben. Für alle anderen Argumente wird die Funktion *n!* definiert als das Produkt der Zahl *n* mit dem Funktionsergebnis der Vorgängerzahl *n-1*, also als *(n-1)!*. Die Produktformel wird also so umgeformt, daß die zu definierende Funktion *!* auf beiden Seiten der Gleichung verwendet wird. Dieses Prinzip, die Definition einer Funktion durch sich selber, wird als rekursive Definition bezeichnet.

Allgemein sind alle berechenbaren Funktionen rekursiv darstellbar (Hopcroft & Ullman 1990, Kap. 8). Zentrale Voraussetzung für eine rekursive Definition ist, daß sich ein Problem in strukturgleiche Teilprobleme zerlegen läßt. Soll die Anwendung einer rekursiven Definition für beliebige Eingaben terminieren, so müssen die Teilprobleme dabei "kleiner" als das entsprechende Ausgangsproblem sein, das heißt, daß eine Ordnung auf dem Argumentbereich der Funktion angebbar sein muß. Zudem muß es ein ausgezeichnetes kleinstes Teilproblem geben, das heißt, daß die Ordnungsrelation wohlfundiert sein muß (vgl. Wechler 1992, S. 35; Manna & Waldinger 1975).

Dies ist bei der Fakultätsfunktion der Fall: Die Fakultät einer natürlichen Zahl *n* kann auf das Problem der Berechnung der Fakultät der Zahl *n-1* mit *(n-1)* < *n* reduziert werden. Durch die Reduzierung des Ausgangsproblems *n!* auf immer kleinere Teilprobleme wird das

Funktionsargument solange minimiert, bis das kleinstmögliche Argument - die kleinste natürliche Zahl null - erreicht wird, für das ein Ergebnis direkt angebbar ist. Entsprechend wird der Fall der Definition, bei dem ein direktes Ergebnis angebbar ist, direkter Fall, der Fall, bei dem die Funktion selbst mit minimiertem Argument aufgerufen wird, rekursiver Fall genannt.

Zur Veranschaulichung wird im folgenden die obige Definition auf die Berechnung von *5!* angewendet (siehe Abb. 6).

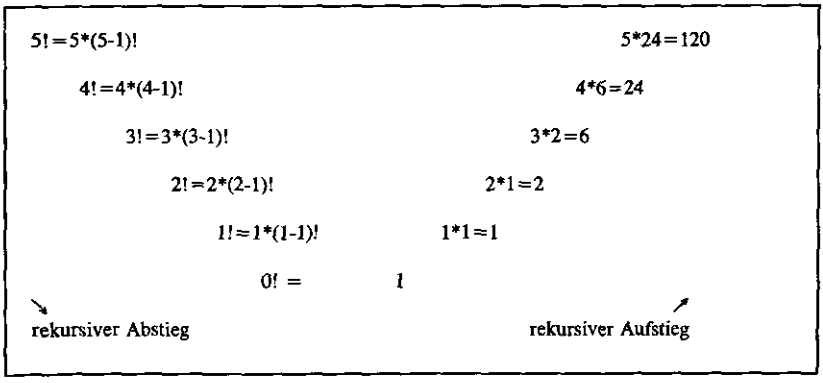


Abb. 6: *Aufrufdiagramm für die rekursive Berechnung der Fakultät*

Die rekursive Definition eines Problems liefert zugleich eine Berechnungsvorschrift für die Problemlösung. Entsprechend muß bei der Erstellung einer rekursiven Funktion in einer Programmiersprache eine Problemdefinition gefunden werden, die das Problem und seine Lösung für das kleinste Argument (direkter Fall) sowie die Zerlegung des Problems in ein kleineres Problem zusammen mit einer Verknüpfungoperation (rekursiver Fall) beschreibt. Zur Laufzeit des Programms wird dann die Berechnungsvorschrift (durch einen Interpreter) auf konkrete Argumente angewendet.

Die kognitive Leistung des Programmierers besteht also in der Lösung folgender Punkte:

- Identifikation des zu minimierenden Arguments (bei n! das einzige Funktionsargument n)
- Definition einer wohldefinierten Ordnungsrelation auf dem Datentyp des zu minimierenden Arguments (bei n! die Ordnung auf den natürlichen Zahlen $0 < 1 < 2 < 3 \ldots < n-1 < n$)
- Festlegung des kleinsten Teilproblems (bei n! 0)
- Festlegung des Funktionsresultats für das kleinste Teilproblem (bei n!: 0! = 1)
- Identifikation der Operation, die das zu minimierende Argument so verändert, daß das

kleinste Teilproblem erreicht werden kann (bei n! die Subtraktion von 1: n-1)
- Festlegung der Operation, die das gelöste Teilproblem mit dem Ausgangsproblem verknüpft (bei n! die Multiplikation n * (n-1)!).

Bei der Entwicklung einer rekursiven Problemdefinition können zwei Klassen von Fehlern unterschieden werden, die vor allem mit Rekursion unerfahrenen Personen unterlaufen: Zum einen kann es vorkommen, daß eine rekursive Definition angegeben wird, die keine Lösung des Problems darstellt, eine Ausführung der Berechnungsvorschrift mit konkreten Werten führt dann zu nicht beabsichtigten Ergebnissen. So kann es sein, daß ein falsches kleinstes Teilproblem identifiziert wird und damit entweder nicht alle Teilberechnungen durchgeführt werden oder es zu einer Bereichsüberschreitung kommt, oder daß eine falsche Verknüpfungsoperation gewählt wird und damit andere Werte berechnet werden, als es die Problemstellung verlangt.

Zum anderen kann es vorkommen, daß die Berechnungsvorschrift so definiert wird, daß der als kleinstes Teilproblem definierte Wert nicht erreichbar ist. Damit wird bei Ausführung der Berechnungsvorschrift stets der rekursive Fall ausgeführt. Es werden also immer mehr Teilprobleme erzeugt, die Definition wird (theoretisch) unendlich oft expandiert, ohne daß ein Ergebnis berechnet werden kann. Im Programmierbereich wird das Abschließen einer Berechnung als Termination bezeichnet. Also entspricht die zweite Fehlerklasse den nichtterminierenden Definitionen. Solche Fehler entstehen, wenn die Minimierungsoperation so gewählt wurde, daß das zu minimierende Argument nicht in Richtung des kleisten Teilproblems verändert wird.

In der Algorithmik (Berechenbarkeits- und Komplexitätstheorie) gibt es verschiedene Klassifikationsmöglichkeiten für rekursive Funktionen. Bei diesen Klassifikationen stehen Kriterien der Berechenbarkeit und des Berechnungsaufwands im Mittelpunkt (z.B. Hopcroft & Ullman 1990 Kap. 8.8, Kap. 9). Generell ist für jede gegebene Funktion unabhängig von ihrer gegebenen Definitionsform zu prüfen, ob sie linear expandierbar (primitiv-rekursiv; Davis 1958) ist oder nicht (z.B. Field & Harrison 1988, Kap. 18). Linear expandierbare Funktionen können mit Techniken der Programmtransformation in sogenannte endrekursive (*tail-recursion*) Formen umgewandelt und damit iterativ (mittels einer Zählschleife) berechnet werden (Backus 1978).

Fragen der (effizienten) Berechenbarkeit und der Programmtransformation werden im folgenden nicht berücksichtigt, stattdessen werden am Ende des folgenden Abschnitts einige strukturelle Aspekte rekursiver Funktionen angegeben.

5.3 Funktionale Sprachen und Klassifikationskriterien für rekursive Programme

Das Grundkonzept funktionaler Sprachen basiert auf dem Konzept mathematischer Funktionen: Eine Funktion ist definiert als eine Zuordnungsvorschrift von Eingabewerten (Argumentbereich; *domain*) in einen Ausgabewert (Wertbereich; *codomain*, *range*). Im Rahmen der Programmierung spricht man bei den Eingabewerten häufig von (Eingangs-)Parametern, beim Ausgabewert vom Resultat einer Funktion.

Ein Programm wird als Definition einer Transformationsvorschrift von Eingabeparametern in das Ergebnis der Funktion aufgefaßt, entspricht also zum Beispiel der oben angegebenen Fakultätsfunktion.

Eine Funktion besteht also aus einem Namen (z.B. *f*) zusammen mit Platzhaltern für die Argumente (z.B. *n*) und einer Transformationsvorschrift (z.B. *1 wenn n=0; n*(n-1)! sonst*).

Reine funktionale Sprachen (z.B. Kern-LISP, McCarthy et al. 1962; FP Backus 1978) benötigen lediglich eine Menge vordefinierter Datentypen und primitiver Funktionen. Neue Funktionen können dann mithilfe dieser Primitive definiert werden. Alle Berechnungen, die eine wiederholte Anwendung von Operationen benötigen, können als rekursive Funktionen definiert werden. Damit selbstdefinierte Funktionen referenzierbar sind, werden sie bei der Definition mit einem Namen angegeben.

In den in Kapitel 8 dargestellten Untersuchungen wird eine selbstdefinierte funktionale Sprache verwendet. Anders, als die bekannteste funktionale Sprache LISP, ist diese Sprache streng typisiert (vgl. z.B. ML; Milner, Tofte & Harper 1990). Das heißt, daß die Datentypen der Funktionsparamter und -resultate angegeben werden müssen, was den Vorteil hat, daß nur solche Programme ausführbar sind, die korrekt typisiert sind. Als Datentypen stehen natürliche Zahlen, einfache Listen aus natürlichen Zahlen, sowie Wahrheitswerte zur Verfügung.

Im folgenden wird die Sprache in BNF-Notation (Backus 1959; Naur 1960) angegeben und danach zur Veranschaulichung eine Beispielfunktion definiert. Der Interpreter für dies Sprache ist in Anhang B.1 erläutert.

TABELLE 1
BNF einer einfachen typisierten Funktionalen Sprache

```
<program>                 ::   <funktionskopf>
                               <funktionsrumpf>

<funktionskopf>           ::   FUN <fname> ( {<paramter>} |
                               <parameter> {,<parameter>} )
                               : <datentyp>

<funktionsrumpf>          ::   <anweisung> FIN
<parameter>               ::   <pname> : <datentyp>
<datentyp>                ::   NAT | NATLIST | BOOL
<anweisung>               ::   <cons> | <bedingte Anweisung> | <ausdruck>
<cons>                    ::   NIL | TRUE | FALSE | <zahl> | <liste>

<bedingte Anweisung>      ::   IF <boolscher Ausdruck>
                               THEN <anweisung>
                               ELSE <anweisung>

<boolscher Ausdruck>      ::   EQUAL(<pname>|<zahl>,<pname>|<zahl>)
                               GREATER(<pname>|<zahl>,<pname>|<zahl>)
                               EMPTY(<pname>|NIL|<liste>)
                               NOT(<boolscher Ausdruck>)
                               AND(<boolscher Ausdruck>,<boolscher
                                                  Ausdruck>)
                               OR(<boolscher Ausdruck>,<boolscher
                                                  Ausdruck>)

<ausdruck>                ::   PLUS(<pname>|<zahl>|<ausdruck>,
                                     <pname>|<zahl>|<ausdruck> )   |
                               MINUS(<pname>|<zahl>|<ausdruck>,
                                     <pname>|<zahl>|<ausdruck> )   |
                               MULT(<pname>|<zahl>|<ausdruck>,
                                     <pname>|<zahl>|<ausdruck> )   |
                               HEAD( <pname> | <liste> )           |
                               TAIL( <pname> | <liste> )           |
                               CONS (<pname>|<zahl>, <pname>|
                                     <liste> | NIL )               |
                               <boolscher Ausdruck>                |
                               <fname> ({<ausdruck>} | <ausdruck>
                                     {, <ausdruck>})

<zahl>                    ::   0 | 1 | 2 | 3 ...
<liste>                   ::   list({<zahl>})
<fname>, <pname>          ::   <name>
<name>                    ::   <letter>{<letter>}
<letter>                  ::   a | b | c | d ... | z |
                               A | B | C | D ... | Z
```

Die beschriebene einfache funktionale Sprache hat einen sehr geringen Syntaxumfang. Trotzdem kann eine große Menge von Funktionen damit ausgedrückt werden. Dies liegt daran, daß funktionale Sprachen auf dem mathematischen Funktionenkonzept basieren und alle berechenbaren Funktionen als rekursive Funktionen ausgedrückt werden können. Ein Programmierer hat damit kaum kognitiven Aufwand beim Erwerb der Syntax einer einfachen funktionalen Sprache. Die kognitive Leistung besteht darin, für ein gegebenes Problem eine abstrakte funktionale Definition zu finden (vgl. Kap. 5.2).

Die in Kapitel 5.2 sprachunabhängig definierte Fakultätsfunktion kann in dieser einfachen funktionalen Sprache definiert werden als:

```
FUN faku (n:NAT):NAT
IF EQUAL(n,0)
THEN 1
ELSE MULT (n, faku(MINUS(n,1)))
FIN
```

An der Fakultätsfunktion wird deutlich, daß vorgegebene rekursive Definitionen problemlos in die Syntax einer funktionalen Sprache überführbar sind.

Funktionale Programme sind also abstrakte Problemlösepläne, die bei Aufruf mit konkreten Werten (z.B. *faku(5)*) durch einen Interpreter ausgewertet werden. Die größere Abstraktheit stellt zwar einerseits höhere Anforderungen an die kognitiven Leistungen der Programmierer, führt aber andererseits zu prägnanteren und kürzeren Programmen, die auch besser mit formalen Methoden auf Korrektheit prüfbar sind. Allgemein können Eingeschaften rekursiver Funktionen mit Induktionsbeweisen (siehe Anhang A.1) geprüft werden.

Um den Unterschied funktionaler Sprachen zu den bekannteren imperativen Sprachen zu verdeutlichen, wird im folgenden die Fakultätsfunktion noch einmal als nicht-rekursives PASCAL-Programm angegeben:

```
PROGRAM fakultät;
VAR
  n,i,fac: INTEGER;
BEGIN
  fac := 0;
  Write('Zahl eingeben: ');
  ReadLn (n);
  FOR i := 1 TO n DO
    fac := fac*i;
  Write ('Ergebnis: ', fac);
END.
```

Die imperative Lösung der Fakultätsfunktion stellt keine abstrakte Problemspezifikation dar, sondern beschreibt die Lösungsmethode in Anlehnung an die Berechnungslogik der von-Neumann-Rechner. Dabei bilden die Variablen (*n, i, fac*) Abstraktionen der Speicherregister, die durch Zuweisung mit konkreten Werten belegt werden. Die Logik funktionaler Sprachen befreit den Programmierer also von der engen Anlehnung an die Funktionsweise der Rechenmaschine (Backus 1978). Anstelle des "*I tell you how*" tritt ein natürlicheres Programmierprinzip des "*I tell you what, you work out how*" (Field & Harrison 1988).

Im Rahmen der funktionalen Programmierung lassen sich verschiedene Grundmuster rekursiver Programme beschreiben. Im wesentlichen werden rekursive Funktionen in die Klassen *map*, *reduce* und *filter* eingeteilt. Diese rekursiven Grundstrukturen können als nicht-rekursive Anwendungen von *higher-order*-Funktionen dargestellt werden (Field & Harrison 1988, Kap. 3). In dieser Arbeit genügt es, die drei genannten rekursiven Grundmuster informell zu beschreiben. Dabei gehen wir davon aus, daß die rekursiven Funktionen auf linearen Listen (wie z.B. *[0,1,2,3,4,5,6]* oder *[a,b,c,d]*) definiert sind. Prinzipiell beschreiben die drei Grundstrukturen Funktionen auf beliebigen Datentypen.

Die *Map*-Struktur umfaßt alle Funktionen, die auf jedes Element einer Liste angewendet werden. So weist eine Funktion, die jedes Element einer Liste natürlicher Zahlen verdoppelt eine Map-Struktur auf. Die *Reduce*-Struktur umfaßt alle Funktionen, die eine komplexere Struktur (z.B. eine Liste von natürlichen Zahlen) in eine reduzierte einfachere Struktur (z.B. eine natürliche Zahl) transformieren. Eine Funktion, die die Anzahl von Elementen einer Liste errechnet, wäre von diesem Typ. Die *Filter*-Struktur umfaßt alle Funktionen, die bestimmte Elemente einer Liste aussortieren. So wäre zum Beispiel eine Funktion, die alle geraden Zahlen aus einer Liste von natürlichen Zahlen zurückliefert, von dieser Struktur.

Es gibt eine Reihe weiterer Klassifikationsvorschläge, die sich an strukturellen Merkmalen rekursiver Funktionen orientieren. Dabei finden sich in Lehrbüchern über rekursive Programmierung folgende Klassifikationsvorschläge:

- Direkte versus indirekte Rekursion (z.B. Perl 1979):
  Bei direkter Rekursion ruft eine Funktion sich selber auf (vgl. die Fakultätsfunktion). Wird eine Funktion F von einer Funktion G aufgerufen, die ihrerseits F aufruft, spricht man von indirekter Rekursion.

- Endrekursive versus echtrekursive Programme (z.B. Beckstein 1993; Field & Harrison 1988, Kap. 2):
  Endrekursive Funktionen sind solche, bei denen der rekursive Aufruf am Ende steht und das Funktionsergebnis in einem Parameter aggregiert wird. Damit entsprechen endrekursive Funktionen der Logik von Zählschleifen: bei Termination der Funktion wird das Ergebnis direkt ausgegeben. Echtrekursive Funktionen sind dagegen durch "wartende" Funktionsaufrufe charakterisiert. Das heißt, daß eine Ergebnisberechnung erst erfolgen kann, wenn das Ergebnis des nachfolgenden rekursiven Aufrufs feststeht. Damit benötigen echtrekursive Funktionen nicht nur einen rekursiven Abstieg (wobei die Funktionsaufrufe auf einem Stack gespeichert werden) sondern auch einen rekursiven Aufstieg. Das Prinzip der wartenden Funktion ist am Beispiel der Fakultätsfunktion in Kapitel 5.1 veranschaulicht. Manche Autoren bezeichnen die echtrekursiven Funktionen auch als Teil-Rest-Rekursion (z.B. Vorberg & Goebel 1991; vgl. Kap. 5.4).

- Lineare versus baumrekursive Programme (Beckstein 1993; Anderson, Corbett & Reiser 1987):
  Linear rekursive Programme sind solche, bei denen im rekursiven Fall (Reduktionsschritt) jeweil nur ein rekursiver Aufruf vorkommt. Baumrekursive Programme haben dagegen in

einem Reduktionsschritt mehrere rekursive Aufrufe. Solche Strukturen werden auch als Simultanrekursion bezeichnet. Das klassische Beispiel einer baumrekursiven Funktion ist Fibonacci:

```
FUN fib(n:NAT):NAT
IF EQUAL (n,0)
THEN 0
ELSE IF EQUAL(n,1)
    THEN 1
    ELSE PLUS(fib(MINUS(n,1)),fib(MINUS(n,2)))
FIN
```

Betrachtet man die Berechnung von fib(4) im Vergleich zu fac(4), so wird deutlich, warum diese Art von rekursiver Funktion als Baumrekursion bezeichnet wird (siehe Abb. 7).

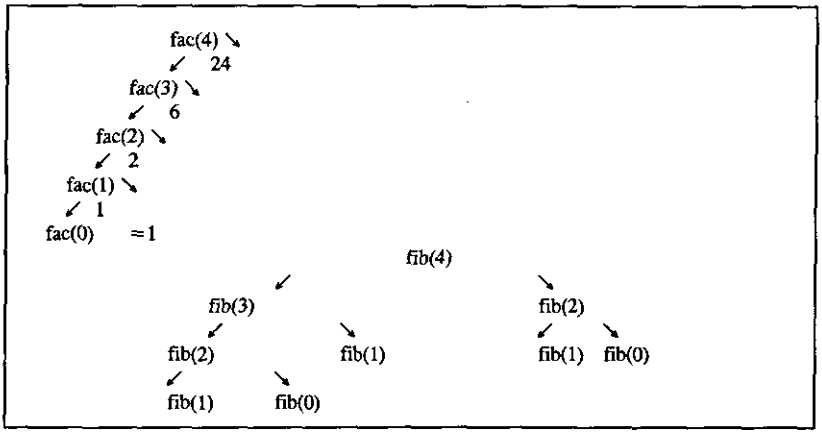


Abb. 7: Aufrufdiagramme für eine lineare und eine baumrekursive Funktion

Bei den angegebenen Kategorien rekursiver Funktionen ist zu beachten, daß nicht die mathematischen Eigenschaften der Funktionen (primitv-rekursive versus μ-rekursive Funktionen), sondern die Strukturen "implementierter" Funktionen zur Klassifikation herangezogen werden. So können zum Beispiel einige baumrekursive Funktionen auch als lineare Funktionen kodiert werden. Dies ist zum Beispiel für die Fibonacci-Funktion möglich, in dem der Eingangsparamter n als Zähler (z) verwendet wird und x und y mit den direkten Fällen *0* und *1* belegt werden:

```
FUN fib1 (x:NAT, y:NAT, z: NAT):NAT
IF EQUAL(z,0)
THEN y
ELSE fib(PLUS(x,y), x, MINUS(z,1))
FIN

FUN fib(n:NAT):NAT
  fib1(0,1,n)
FIN
```

Ich gehe davon aus, daß sich Programmierer von solchen strukturellen Eigenschaften rekursiver Funktionen leiten lassen, in dem sie neue Problemstellungen mit bereits gelösten vergleichen und versuchen, die dort verwendeten Strukturen auf das neue Problem zu übertragen. Eine weitere Klassifikation, die explizit als Modell für Expertenwissen über Rekursion vorgeschlagen wurde, wird im folgenden Abschnitt diskutiert.

5.4 Expertenwissen über Rekursion: Schemahierarchien

In Kapitel 5.1 wurden kognitive Leistungen aufgeführt, die zu erbringen sind, um eine rekursive Lösung für eine Problemstellung zu entwickeln. Experten - in Mathematik und Informatik - scheinen diese Anforderungen (in den meisten Fällen) problemlos zu erfüllen. Die Expertiseforschung ist, vor allem durch die Entwicklungen im Bereich Expertensysteme, zu einem breiten Forschungsfeld geworden (z.B. Wachsmuth 1993), auf das hier nur am Rande eingegangen wird. In der Kognitionspsychologie wurde Expertiseforschung vor allem in den Bereichen Schach (DeGroot 1965; Chase & Simon 1973), Physik (Chi, Feltovich & Glaser 1981) und Programmierung (McKeithen, Reitman, Rueter & Hirtle 1981) durchgeführt. Sowohl im Bereich der Expertensysteme als auch im Bereich der psychologischen Expertiseforschung sind Fragen nach dem Format des repräsentierten Wissens zentral. Allgemein wird davon ausgegangen, daß Experten umfassendes Wissen über den Bereich ihrer Expertise besitzen, das in hochstrukturierter Form repräsentiert sein muß. Eine integrative hochstrukturierte Repräsentation von Wissen ist die Voraussetzung dafür, daß Experten in der Lage sind, neue Probleme direkt auf ihren Erfahrungskontext zu beziehen und so sehr schnell in die passende Problemklasse einzuordnen. Dies ermöglicht ihnen, das notwendige Faktenwissen sowie die angemessenen Lösungsstrategien unmittelbar auf ein neues Problem zu übertragen (Strube 1993; Vorberg & Goebel 1991).

Zur Modellierung von Expertenwissen über rekursive Programmierung wurden Produktionsregeln (Anderson, Conrad & Corbett 1989) und Pläne bzw. Schemata (Soloway 1985; Vorberg & Goebel 1991) verwendet. In Kapitel 3 wurde argumentiert, daß in Produk-

tionsregeln repräsentiertes Wissen zwar den Vorteil hat, modular organisiert und effizient anwendbar zu sein, aber andererseits keine strukturierte Repräsentation der Zusammenhänge einzelner Wissenskomponenten möglich ist. Da Experten sowohl effizient auf Problemanforderungen reagieren können als auch in der Lage sind, Zusammenhänge zwischen verschiedenen Wissensaspekten in ihrem Expertisebereich zu nutzen, sollte ein Expertisemodell beide Repräsentationsformen integrieren (vgl. Kapitel 4 und 6).

Ein Modell des Expertenwissens über rekursive Funktionen, das die Struktur rekursiver Funktionen betont, stammt von Vorberg und Goebel (1991; Goebel & Vorberg 1991). Die Autoren gingen davon aus, daß die Autoren von Lehrbüchern über funktionale Programmierung ausgewiesene Experten für rekursive Programmiertechniken sind. Aus diesem Grund analysierten sie *383* Beispielprogramme aus *7* Lehrbüchern über LISP und LOGO. Das Ergebnis der Analyse war eine Hierarchie von Rekursionsschemata. In diese Hierarchie wurden jedoch nur noch die *214* Funktionen, die auf Listen arbeiten, aufgenommen. Diese Funktionen wurden in die Kategorien Endrekursion (*83*) und echte Rekursion (*131*) unterteilt. Eine weitere Differenzierung wurde dann nur für die echtrekursiven Funktionen vorgenommen, die die Autoren als Teil-Rest-Rekursion bezeichnen: Ausgehend von einem Grundschema für Rekursion wird ein Teil-Rest-Schema angegeben, das sich in die Untergruppen *Map*, *Reduce* und *Filter* unterteilt (vgl. Kap. 5.2). Mit dieser Hierarchie konnten *120* der *214* Listen-Funktionen erfaßt werden.

Das Grundschema besteht aus den Leerstellen direkter Fall, Rückgabe, Hilfsfunktion, Restselektor und Verknüpfer, mit der Struktur:

direkter Fall?
ja -> Rückgabe
nein -> Verknüpfer (*Hilfsfunktion*; [Restselektor; Rekursion]).

Eine Funktion, die die Summe je zwei benachbarter Listenelemente berechnet, kann in dieser Struktur beschrieben als:

direkter Fall:	
OR(EMPTY(TAIL(l)), EMPTY(l))	Hat die Liste nur ein Element?
Rückgabe:	
NIL	leere Liste
Hilfsfunktion:	
PLUS(HEAD(l),HEAD(TAIL(l)))	Summe des ersten und zweiten Listenelements

Restselektor:	
TAIL(l)	Liste ohne erstes Element
Verknüpfer:	
CONS	Füge Objekt an die erste Stelle einer Liste.

Das Teil-Rest-Schema entsteht, indem das Grundschema um eine Leerstelle für den Teilselektor erweitert wird. Die Schemata *Map*, *Reduce* und *Filter* entstehen, in dem die Wertebereich der Leerstellen des Teil-Rest-Schemas eingeschränkt werden.

Die Autoren konnten mit Computersimulation belegen, daß die vorgeschlagene Schemahierarchie zusammen mit einer Strategie zur Zerlegung rekursiver Probleme in Teilprobleme genügen, um rekursive LOGO-Prozeduren zu kodieren (Goebel & Vorberg 1991). Dabei wird, ausgehend von einer Problemrepräsentation, in der Schemahierarchie nach der spezifischsten Struktur gesucht, die mit der Problemstruktur übereinstimmt. Mithilfe einer Regelbasis wird das aktivierte Schema dann in allen Leerstellen schrittweise verfeinert, bis ein lauffähiges Programm vorliegt.

Die Modellierung von Expertenwissen über Rekursion in Form eine Schemahierarchie zusammen mit Regeln zur Verfeinerung scheint überzeugend. Bei rekursiven Programmen sind die notwendigen Bestandteile und deren struktureller Zusammenhang eindeutig festgelegt. Damit ist es plausibel, die Struktur rekursiver Programme als Schemata zu beschreiben. In Kapitel 6 wird eine eigene Schemahierarchie zur Repräsentation rekursiver Funktionen vorgeschlagen, bei der etwas andere strukturelle Unterscheidungen getroffen werden, als bei Vorberg und Goebel.

5.5 Modellierung von Programmier-Expertise beim automatischen Programmieren

Die Computersimulation von Goebel und Vorberg (1991) zum Erstellen rekursiver LOGO-Funktionen ist sowohl ein Modell kognitiver Prozesse als auch ein Beitrag zum automatischen Programmieren. Im Prinzip leisten alle Ansätze zum automatischen Programmieren wertvolle Beiträge, um Aufschluß über Inhalt und Struktur der Wissensrepräsentation von Programmierexperten zu erhalten. Ein System, das formale Spezifikationen in eine *LISP*-ähnliche Zielsprache transformiert, ist DEDALUS von Manna und Waldinger (1975). Dieses System soll kurz vorgestellt werden:

Die Spezifikationssprache für DEDALUS enthält Konstrukte, die dem nahe kommen, wie ein Programmierer über ein Problem nachdenkt, und ermöglicht gleichzeitig eindeutige und logisch vollständige Spezifikationen des gewünschten Programms. Als Beispiel wird im folgenden die Synthese eines Programms skizziert, das prüft, ob eine Zahl x kleiner als alle

Elemente einer Liste von Zahlen ist. Die Spezifikation dieses Problems lautet:

LESSALL(x l) <= **compute** x < ALL(l)
where x is a number and l is a list of numbers.

Allgemein haben Spezifikationen die Form f(x) <= *compute* P(x) where Q(x). Dabei werden bei der Spezifikation für P und Q Konstrukte verwendet, die durch eine wiederholte Anwendung von Transformationsregeln in die Syntax der Zielsprache überführt werden. Im obigen Beispiel wurde das Konstrukt *ALL* verwendet, das so transformiert werden muß, daß das Programm für alle Elemente aus *l* prüft, ob die Zahl *x* kleiner als dieses Element ist.

Die Anwendung der Transformationsregeln auf die Spezifikation entspricht einer schrittweisen Verfeinerung: Jeder Ausdruck der Spezifikation wird solange transformiert, bis er vollständig in die Zielsprache überführt ist. Zu den Transformationsregeln gehört unter anderem eine Menge allgemeiner Programmierprinzipien, wie die Bildung bedingter Anweisungen, rekursiver Funktionen und die Konstruktion wohlfundierter Ordungsrelationen zur Gewährleistung, daß die rekursiven Funktionen terminieren. Diese Regeln enthalten also einen wichtigen Teil des Problemlösewissens von Programmieren (vgl. 5.1).

Eine bedingte Anweisung wird erzeugt, wenn DEDALUS versucht, eine Regel zur Transformation eines Ausdrucks *P* (z.B. betrachte das erste Element einer Liste) anzuwenden wobei eine Bedingung *Q* (z.B. die Liste ist nicht leer) gelten muß, deren Gültigkeit nicht bewiesen aber auch nicht widerlegt werden kann. In diesem Fall wird eine "*condition-formation*"-Regel aktiviert, die eine Analyse durchführt, in welchen Fällen *Q* gilt und in welchen nicht, und gleichzeitig *P* durch zwei Anweisungen *S1* und *S2* ersetzt, die beide das in *P* formulierte Ziel erfüllen.

Die obige Problemspezifikation *LESSALL* wird in DEDALUS in eine rekursive Funktion transformiert. Zunächst wird aus dem *ALL*-Konstrukt abgeleitet, daß um das Ziel *x < ALL(l)* zu erreichen, das Teilziel *x < ALL(tail(l))* erreicht werden muß. Damit wird die "*recursion-formation*"-Regel aktiviert, die als Unterziel für *compute LESSALL(x l)* den Aufruf *LESSALL(x tail(l))* generiert, wobei weiterhin *Q* gelten muß, also *x is a number and tail(l) is a list of numbers*. Eine Analyse der Anwendungsbedingungen für tail ergibt, daß eine Fallunterscheidung notwendig ist: die Eingabe *l* darf nur durch *tail(l)* ersetzt werden, wenn *l* nicht leer ist. Zudem wird bei der Synthese einer rekursiven Funktion geprüft, ob die generierten Unterziele einer wohlfundierten Ordnungsrelation entsprechen, das heißt in diesem Fall, es muß gezeigt werden, daß *tail(l)* kleiner ist als *l* und daß es einen kleinsten Fall gibt, so daß die synthetisierte Funktion terminiert. In der oben eingeführten Fallunterscheidung zwischen leerer und nicht leerer Liste ist der kleinste Fall für Listen, die leere Liste bereits enthalten. Eine Zahl *x* ist kleiner als jedes Element einer leeren Liste. Das

System synthetisiert mithilfe dieser Transformationsregeln, die eine große Menge an Wissen über Programmierkonzepte enthalten schließlich folgende Funktion:

LESSALL(x l) <= **if** EMPTY(l) **then** TRUE
else *x* < *HEAD(l) and LESSALL(x TAIL(l))*.

In der oben eingeführten einfachen funktionalen Sprache entspricht dies der Funktion

```
FUN lessall(x:NAT, l:NATLIST): BOOL
IF EMPTY(l)
THEN TRUE
ELSE AND(NOT(GREATER(x,HEAD(l)) OR EQUAL(x,HEAD(l))),lessall(x,TAIL(l)))
FIN
```

Dabei kommt der etwas komplexere Ausdruck beim rekursiven Aufruf dadurch zustande, daß keine kleiner-Relation im Wortschatz der Sprache vorhanden ist und die Prüfung $x <$ *head(l)* durch "*x ist weder größer noch gleich head(l)*" ausgedrückt werden muß.

Das System DEDALUS synthetisiert Programme, indem es mit Hilfe von Tranformationsregeln einen Ausdruck *compute P(x) where Q(x)* solange verfeinert, bis er in der Syntax einer einfachen funktionalen Sprache vorliegt. Auch wenn die implementierten Programmierkonzepte sicher eine geeignete Formalisierung des Wissens über Programmieren darstellen, scheint die Idee der schrittweisen Verfeinerung ohne Verwendung einer strukturierten Repräsentation rekursiver Konzepte wenig plausibel als Modell eines menschlichen Programmierers (dafür war DEDALUS auch nicht konzipiert!). Hier ist die Idee der Programmschemata, wie sie in Ansätzen verwendet wird, die mit Beispiel-Spezifikationen arbeiten, wesentlich überzeugender. Die Idee der Programmschemata entspricht prinzipiell der Modellierung von Expertenwissen über Rekursion mit einer Hierarchie von Rekursionsschemata (Vorberg & Goebel 1991; vgl. Kap. 5.4).

Im folgenden Kapitel werden die Idee einer Hierarchie von Rekursionsschemata und die Idee der regelhaften Anwendung von Wissen über Programmierkonzepte zu einem Expertenmodell über rekursive Programmiertechniken kombiniert. Ein solches Expertenmodell kann als Zielmodell für die Modellierung des Erwerbs rekursiver Techniken dienen.

6 Ein integratives Modell zum induktiven Erwerb von rekursiven Programmiertechniken

Im folgenden wird ein Modell zum induktiven Erwerb von rekursiven Problemlösefertigkeiten mit Beispielen vorgestellt. In diesem Modell werden verschiedene Ansätze, die in den vorangegangenen Kapiteln diskutiert wurden, integriert:

Zur Repräsentation des erworbenen Wissens über rekursive Programmierung im Langzeitgedächtnis wird eine Schemahierarchie verwendet, die neben Leerstellen für die Komponenten rekursiver Funktionen auch Leerstellen für Regeln über Programmierkonzepte enthält. Damit wird die in Kapitel 3 diskutierte Idee der Kombination von Schemata zur Konzeptrepräsentation mit Produktionsregeln (vgl. Kap. 3.5) aufgenommen und für die Repräsentation von Wissen über Rekursion konkretisiert. Die im folgenden als notwendig postulierten Wissensaspekte für die Entwicklung rekursiver Funktionen aus nicht rekursiv formulierten Spezifikationen wurden zum Teil bereits von anderen Autoren verwendet: Eine Repräsentation der Syntax und Semantik rekursiver Funktionen findet sich ähnlich bei Anderson und Thompson (1989). Die Idee, rekursive Funktionen nach strukturellen Merkmalen zu klassifizieren und Leerstellen für die Komponenten rekursiver Funktionen zu verwenden, wurde ähnlich bereits von Vorberg und Goebel (1991; vgl. Kap. 5.4) formuliert. Regeln zur Bildung rekursiver Funktionen wurden von Manna und Waldinger (1975) und von Goebel und Vorberg (1991) angegeben (vgl. Kap. 5.5).

Die Idee, mentale Modelle zur Repräsentation aktueller Problemlösungen zu verwenden, und den Lösungsprozeß als Transformation eines mentalen Modells zu beschreiben, wurde von Holland, Holyoak, Nisbett und Thagard (1986) formuliert (vgl. Kap. 3.5 und 4.5). Während diese Autoren jedoch sehr unspezifisch bleiben, wird im folgenden konkret angegeben, auf welche Weise aktuelle Problemstellungen in Problemlösungen transformiert werden.

Das Konzept des induktiven Wissenserwerbs durch die Lösung von Aufgaben mit strukturell analogen Beispielen als Unterstützung ist gut in die Literatur eingeführt und wird sowohl in der kognitiven Psychologie (Anderson & Thompson 1989; Novick & Holyoak 1991) als auch beim maschinellen Lernen (Carbonell 1983; Carbonell 1986) als zentraler Mechanismus für induktives Lernen angesehen (vgl. Kap. 4). Dieses Konzept wird im folgenden für den Erwerb rekursiver Programmiertechniken präzisiert: Eine Programmieraufgabe wird als Instantiierung eines Semantik-Schemas (Spezifikation) repräsentiert. Es wird versucht, mithilfe der bereits im Gedächtnis vorhandenen Schemahierarchie und der Repräsentation der Spezifikation und der rekursiven Definition einer Beispielfunktion, das Rekursionsschema der Programmieraufgabe zu instantiieren. Die Instantiierung eines Rekursionsschemas wird als Prozeß einer schrittweisen Transformation beschrieben. Die für diese Transformation notwendigen Inferenzen über einen Vergleich von Beispiel und Aufgabe führen zu einer Erweiterung der im Langzeitgedächtnis repräsentierten Schemahierarchie. Dabei werden die

Prozesse des Vergleichs von Beispiel und Aufgabe, sowie der Anpassung des Beispiels an die Aufgabe in Anlehnung an die Ansätze von Anderson und Thompson (1989) und Carbonell (1983) beschrieben. Zunehmende Expertise wird als Erwerb charakteristischer Merkmale zur Unterscheidung verschiedener Typen rekursiver Funktionen und als Erwerb von Regeln zur Kodierung solcher Funktionen beschrieben. Nur wenn sowohl Konzepte für rekursive Funktionen als auch Techniken zur rekursiven Programmierung erworben und aufeinander abgestimmt sind, sind die kognitiven Leistungen eines Experten in diesem Bereich modellierbar. Expertise umfaßt hier sowohl die Fähigkeit, rekursive Funktionen zu klassifizieren und ihre Arbeitsweise zu beschreiben, als auch die Fähigkeit, neue rekursive Funktionen zu erstellen.

Es sei hier nocheinmal darauf hingewiesen (vgl. Kap. 3.5), daß die im folgenden formulierten Annahmen über Repräsentationsformate keine empirisch prüfbaren Hypothesen sind, sondern lediglich einen Modellierungsvorschlag darstellen, den ich als sparsame und psychologisch plausible Repräsentation der notwendigen Wissensinhalte und -strukturen zum Programmieren rekursiver Funktionen ansehe. Am Ende dieses Kapitels werden jedoch aus den formulierten Annahmen zur Wissensinduktion beim Lernen mit Beispielen Hypothesen abgeleitet, die in den in Kapitel 8 dargestellten Untersuchungen überprüft wurden.

6.1 Grundgedanken des Modells: Eine integrative Struktur zur Repräsentation von Konzept- und Regelwissen

Expertise in einem Problembereich bedeutet vor allem, daß bereichsspezifische Probleme effizient und korrekt gelöst werden können. Diese Fähigkeit basiert im Wesentlichen auf dem Vorhandensein geeigneter Problemlöseoperatoren. Eine notwendige Voraussetzung, um diese Operatoren erfolgreich einsetzen zu können, ist die Fähigkeit zur Klassifikation von Problemtypen. In Produktionssystemen steckt die Problemklassifikation implizit in den Bedingungsseiten der Produktionsregeln. Auf diese Weise kann effizientes und hochautomatisiertes Problemlösen modelliert werden. Allerdings kann durch diese Art der Modellierung nicht abgebildet werden, daß Experten in vielen komplexen kognitiven (nicht unbedingt in motorischen) Bereichen auch in der Lage sind, ihren Problembereich explizit zu strukturieren. Kontextunabhängiges und verbalisierbares Wissen über einen Problembereich kann nicht sinnvoll mit Produktionsregeln repräsentiert werden, hier ist vor allem der Schemaansatz ein geeignetes Repräsentationsformat.

Ich gehe davon aus, daß für die Beschreibung von Expertenwissen über Rekursion sowohl die Repräsentation von konzeptuellem Wissen als auch die Repräsentation von Regelwissen notwendig ist. Nur durch die Berücksichtigung beider Wissensaspekte können die unter-

schiedlichen Fähigkeiten eines Experten modelliert werden. Die Fähigkeit eines Experten, schnell und korrekt rekursive Lösungen für vorgegebene Probleme zu entwickeln, kann gut mit einem Produktionssystem modelliert werden: Regelbedingungen werden über aktuelle Daten und Ziele im Arbeitsgedächtnis aktiviert und die Ausführung der Aktionsteile der Regeln erzeugt schrittweise den geforderten Programmcode. Dies entspricht einer hochspezialisierten, kontextabhängigen, effizienten und nicht verbalisierten Wissensanwendung (Anderson 1981; Rumelhart & Norman 1981).

Die Fähigkeit eines Experten, Konzepte und Strukturen, die der Entwicklung rekursiver Funktionen zugrunde liegen, verbalisieren zu können, kann gut über Schemata modelliert werden. Entsprechend schlage ich ein Repräsentationsformat vor, bei dem die zentrale Struktur eine Schemahierarchie ist, bei der an jedes Schema die für das repräsentierte Rekursionskonzept spezifischen Produktionsregeln gebunden sind. Jedes Schema hat also Leerstellen für die strukturellen Merkmale einer rekursiven Funktion (Funktionsstruktur) sowie für Produktionsregeln, die die Definition einer rekursiven Funktion, die der entsprechenden Struktur entspricht, ermöglichen (Funktionsdefinition).

Die Leerstellen der Funktionsstruktur vererben sich in der Schemahierarchie, wobei die Schemata in tieferen Hierarchieebenen Spezialisierungen der entsprechenden Oberschemata darstellen. Spezialisierungen ergeben sich erstens durch zunehmende Vorinstantiierung und zweitens durch Einschränkungen der Komponenten der entsprechenden rekursiven Struktur (vgl. Rumelhart & Norman 1981). Die Produktionsregeln sind zielorientiert, das heißt im Bedingungsteil wird jeweils das Vorliegen eines Teilziels bei der Definition einer rekursiven Funktion geprüft (vgl. Anderson 1983).

Damit kann innerhalb einer einzigen Repräsentationsstruktur sowohl das Beherrschen von Problemlösestechniken, hier also das Kodieren rekursiver Funktionen, als auch das Verfügen über Problemkonzepte, also eine strukturierte Klassifikation von Typen rekursiver Funktionen, modelliert werden. Dabei ist ein seperates Agieren mit der Funktionsstruktur und der Funktionsdefinition, je nach Aufgabenanforderung, möglich. Liegt die Aufgabe vor, eine rekursive Funktion zu programmieren, so kann der Lösungsprozeß allein mit den Produktionsregeln für eine spezifische rekursive Aufgabe modelliert werden. Liegt die Aufgabe vor, rekursive Funktionen zu beurteilen, so kann der Klassifikationsprozeß allein mit der Funktionsstruktur der Schemahierarchie beschrieben werden.

Mein Anliegen ist es jedoch, vor allem für den Prozeß der Erstellung rekursiver Funktionen beide Aspekte der Schemastruktur zu verwenden. Das Lösen einer rekursiven Aufgabe, ausgehend von einer Spezifikation, kann mit diesem Repräsentationsformat folgendermaßen beschrieben werden: Die Spezifikation führt zur Aktivation des spezifischsten passenden Schemas in der Schemahierarchie. Die an das Schema gebundenen Produktionsregeln können unter Verwendung der in der Spezifikation gegebenen Information die Funktionsleerstellen

instantiieren. Damit ergänzt sich Konzept- und Regelwissen zur erfolgreichen Bewältigung der gestellten Aufgabe.

Der hier weniger zentrale Aspekt der Klassifikation gegebener rekursiver Funktionen kann meiner Meinung nach ebenfalls angemessener unter Verwendung beider in den Schemata integrierter Wissensaspekte beschrieben werden: Um eine rekursive Funktion klassifizieren zu können, müssen strukturelle Merkmale zur Charakterisierung der Funktion verfügbar sein. Zusätzlich ist es hilfreich für eine sinnvolle Funktionsklassifikation, wenn auch die Semantik der Funktion erschlossen werden kann. Dies ist prinzipiell nur möglich, wenn er selbst in der Lage ist, aus einer Aufgabenspezifikation (Semantik) die entsprechende rekursive Definition zu generieren.

Bisher wurde über die Wissensstrukturen diskutiert, die notwendig zur Beherrschung rekursiver Programmierung sind. Zur sinnvollen Modellierung eines Wissenserwerbsprozesses ist es jedoch notwendig, zunächst zu identifizieren, welche Wissensstrukturen am Ende eines erfolgreichen Lernprozesses vorhanden sein müssen. Aus diesem Grund wird in Kapitel 6.2 ein Ausschnitt des Expertenwissens über rekursive Programmierung dargestellt und demonstriert, daß mit dem vorgeschlagenen Repräsentationsformat sowohl die Modellierung des Prozesses der Programmierung rekursiver Funktionen als auch der Klassifikation rekursiver Funktionen möglich ist.

Entsprechend der vorgeschlagenen integrierten Repräsentationsstruktur nehme ich an, daß der Wissenserwerb in einem Problembereich den Erwerb struktureller Merkmale zur Unterscheidung von Problemtypen in diesem Bereiche und den Erwerb von Regeln zur Problemlösung umfaßt. Zudem gehe ich davon aus, daß der natürliche Erwerb von Problemlösekompetenz über das Lösen von Aufgaben aus diesem Bereich funktioniert (*learning by doing*; vgl. Kapitel 4). *Learning by doing* ist jedoch nur möglich, wenn - zumindest zu Beginn des Lernprozesses - zusätzlich Informationen zur Verfügung stehen. Solche Zusatzinformationen können Beispiellösungen für Probleme sein, die aus dem Bereich, in dem Expertise erworben werden soll, stammen. Diese Art des Wissenserwerbs wurde in Kapitel 4 als analoges Lernen bezeichnet, welches als ein Spezialfall des induktiven Lernens angesehen wird. Der Prozeß des analogen Lernens wird von vielen Autoren übereinstimmend in mehrere zentrale Teilprozesse gegliedert (vgl. Anderson & Thompson 1989; Carbonell 1983). Zur Lösung einer Aufgabe bei Vorgabe eines Beispiels müssen Aufgabe und Beispiel miteinander verglichen werden (*mapping*). Dieser Vergleich basiert auf der Identifikation von strukturellen Ähnlichkeiten. Auf Grundlage der Ergebnisse des Vergleichsprozesses wird im nächsten Schritt versucht, den Lösungsweg des Beispiels auf die Aufgabe zu übertragen (*adaptation*). Das Erkennen von strukturell identischen aber je nach Aufgabe anders instantiierten Komponenten führt zum Aufbau einer abstrakteren Repräsentation des Problembereichs und damit zu einem Lernzuwachs (*learning*).

Die Unterscheidung dieser drei Prozesse, die sich auch empirisch als sinnvoll erwies (Novick & Holyoak 1991; Reed & Bolstad 1991; Ross & Kennedy 1990), behalte ich bei. Bezüglich der Modellierung der Vergleichs- und Anpassungsprozesse besteht ebenfalls große Einigkeit (Anderson & Thompson 1989; Carbonell 1983). Die Beschreibung des resultierenden Lernzuwachses fällt jedoch bei den verschiedenen Autoren sehr unterschiedlich aus. Im kognitionspsychologischen Modell von Holland, Holyoak, Nisbett & Thagard (1989) werden lediglich sehr vage Aussagen über den Aufbau abstrakter Schemata über den Problembereich formuliert. Die empirische Prüfung des Schemaerwerbs beschränkt sich entsprechend auf den Nachweis einer erhöhten Fähigkeit zum Lerntransfer als Ergebnis des Lösens von Aufgaben mit Beispielunterstützung (z.B. Novick & Holyoak 1991).

Carbonell (1983) modelliert den Wissenserwerb dagegen konkreter, in dem er die Mittel-Zielanalyse auf die Anwendung von Problemlöseprozessen bei analogen Problemen erweitert. Am spezifischsten ist das Modell von Anderson und Thompson (1989), bei dem die Autoren Lernen als Aufbau und Verfeinerung von Produktionsregeln beschreiben. Dieses Modell wird auch explizit auf die Modellierung des Erwerbs von Programmierkenntnissen angewendet. Obwohl in diesem Modell Schemata zur Repräsentation von Programmieraufgaben und rekursiven Funktionen verwendet werden, wird das im Langzeitgedächtnis gespeicherte Wissen primär mit Produktionsregeln modelliert. Ich beschreibe Wissenserwerb dagegen als Aufbau von Schema- und Regelwissen in einem integrativen Prozeß.

Das im folgenden konkreter dargestellte Modell erhebt keinen Anspruch an Vollständigkeit, weder im Bezug auf die betrachtete Menge rekursiver Funktionen, noch im Bezug auf die Beschreibung des Lernprozesses. Ziel der folgenden Abschnitte ist es lediglich, zu demonstrieren, daß mit der von mir vorgeschlagenen Wissensstruktur eine sinnvolle Modellierung von Expertenwissen und Wissenserwerb im Bereich rekursives Problemlösen möglich ist. Die vorgeschlagene Schemahierarchie für die Repräsentation von Expertenwissen umfaßt nur einen Ausschnitt rekursiver Funktionen. Dieser Ausschnitt, sowie die verwendeten Merkmale zur Klassifikation verschiedener Typen rekursiver Funktionen, sind stark auf die in den empirischen Untersuchungen (Kap. 8) verwendeten Funktionen orientiert. Bei der Darstellung des analogen Wissenserwerbs sind sowohl die Annahmen über die Prozesse des Lösens von Aufgaben mit Beispielen als auch die Annahmen über den Aufbau neuer Wissensstrukturen lediglich beispielhaft beschrieben.

6.2 Eine Wissensstruktur zur Definition rekursiver Funktionen

In diesem Abschnitt wird ausgeführt, welche Wissensinhalte und -strukturen notwendig sind, um erfolgreich rekursive Probleme lösen zu können. Damit befaßt sich dieser Abschnitt mit der Darstellung des Langzeitgedächtnismodells von Experten rekursiver Programmierung. Das Modell wird für die folgenden Abschnitte als Zielmodell für den Wissenserwerb zugrundegelegt. Aspekte des Expertenwissens über rekursive Programmierung wurden im letzten Kapitel diskutiert. Dabei wurde gezeigt, daß sich Wissen über rekursive Funktionen in einer Schemahierarchie repräsentieren läßt (Vorberg & Goebel 1991) und daß rekursive Aufgaben durch schrittweise Verfeinerung einer Spezifikation in einen lauffähigen Programm-Code transformiert werden können (Manna & Waldinger 1975; Goebel & Vorberg 1991).

Im folgenden wird das von mir vorgeschlagene Repräsentationsformat eingeführt, das Schemata und Regeln zur Modellierung von konzeptuellem Wissen und Lösungstechniken für den Bereich rekursives Programmieren in eine Struktur integriert. Zentrale Struktur ist eine Hierarchie von Rekursionsschemata. Dabei sind jedoch zusätzlich an jedes Schema die Produktionsregeln gebunden, die zur Instantiierung der Leerstellen verwendet werden (vgl. Holyoak & Thagard 1989; Kap. 4.5). Das Repräsentationsformat wird in folgenden Schritten eingeführt. Zunächst wird dargestellt, wie natürlichsprachige Aufgabenstellungen als - diesen Texten relativ nahe - Spezifikationen repräsentiert werden. Danach wird die Struktur der postulierten Schemata am allgemeinsten Schema, der Wurzel des Hierarchiebaums erläutert. Die dabei verwendeten Produktionsregeln werden schrittweise eingeführt, indem exemplarisch dargestellt wird, wie die Fakultätsfunktion mithilfe der angegebenen Wissensstruktur in Programmcode umgesetzt wird. Dies soll zudem Veranschaulichen, daß problemlöserelevantes Expertenwissen mit der vorgeschlagenen Wissensstruktur hinreichend beschreibbar ist. Im nächsten Schritt wird die vollständige Schemahierarchie eingeführt.

Die Datentypen und Operationen in der Spezifikation, sowie die Lösung einer rekursiven Aufgabe, werden im folgenden direkt in der in Kapitel 5.3 beschriebenen einfachen funktionalen Sprache angegeben. Im Prinzip sollten Aufgabenstellung und Aufgabenlösung mit abstrakten und programmiersprachenunabhängigen Konstrukten repräsentiert werden. Darauf wird jedoch aus Gründen der Einfachheit verzichtet, da somit (recht einfach anzugebende) Regeln zur Übersetzung des abstrakten Codes in die Syntax einer konkreten funktionalen Programmiersprache entfallen.

Die im folgenden detailliert beschriebenen Aspekte des Expertenwissens können folgendermaßen als Produktionssystem aufgefaßt werden:

(1) **Langzeitgedächtnis:** Das Langzeitgedächtnis enthält eine **Schemahierarchie** rekursiver Funktionen. Dabei besteht jedes Schema erstens aus **Leerstellen für die Struktur rekursiver Funktionen** und zweitens aus **Leerstellen für die Regeln zur Definition rekursiver Funktionen**. Diese Produktionsregeln sind, wie in Produktionssystemen üblich, als Liste in einem Produktionsspeicher abgelegt. Durch ihre Anbindung an die Schemahierarchie sind sie jedoch als strukturierte Gruppen organisiert.

(2) **Kurzzeitgedächtnis:** Das Kurzzeitgedächtnis enthält drei Arten von Daten: Die Aufgabenstellung, das aktuelle Teilziel und ein aktives Schema. Die Aufgabenstellung wird dabei als Schema repräsentiert, das die **Spezifikation der zu entwickelnden rekursiven Funktion** enthält. Auf die Spezifikation wird mit dem Schlüsselwort Aufgabe.Semantik zugegriffen. Gleichzeitig mit der Aufgabenstellung wird das globale Ziel der Problemlösung im Kurzzeitgedächtnis repräsentiert. Im folgenden wird als **globales Ziel** stets "Schreibe eine rekursive Funktion" angenommen. Die Spezifikation löst eine Suche in der Schemahierarchie aus, wobei das **Schema** aktiviert wird, das mit den meisten Aspekten der Spezifikation übereinstimmt. Das aktive Schema wird als Aufgabe.Rekursion bezeichnet. Das globale Ziel initiiert den Prozeß der Aufgabenlösung (siehe 4).

(3) **Schemaaktivierung:** Es wird ein möglichst spezifisches Schema aktiviert, also ein Schema, das sich auf einer möglichst tiefen Ebene in der Hierarchie befindet. Zur Schemaauswahl werden Merkmale der Spezifikation und Merkmale der Rekursionstypen aus der Schemahierarchie verwendet. Solche Merkmale können zum Beispiel das Vorhandensein einer Zählvariable (-> Schema für Endrekursion) oder die Datentypen in der Spezifikation sein.

(4) **Regelaktivierung:** Die Bedingungsseiten der Produktionsregeln enthalten entweder nur (Teil-)Ziele oder zusätzlich Merkmale der Spezifikation und/oder des aktiven Schemas. Ist ein Schema aktiviert stehen nur noch die Regeln aus dem Produktionsspeicher zur Auswahl, die an dieses Schema gebundenen sind. Aus dieser Teilmenge von Produktionsregeln wird dann die Regel aktiviert, die die größte Übereinstimmung mit den Daten im Kurzzeitgedächtnis aufweist. Zunächst wird eine Regel aktiviert, die lediglich das globale Ziel "Schreibe eine rekursive Funktion" im Bedingungsteil enthält. Diese Regel gliedert dann den Prozeß der Aufgabenlösung, indem das globale Ziel zunächst in Teilziele zerlegt wird.

(5) **Regelanwendung:** Die ausgewählte Regel greift auf Information aus der Spezifikation und dem (partiell instantiierten) aktiven Schema zu. Der Aktionsteil einer Regel verändert das aktive Schema, indem eine (weitere) Instantiierung von Schemakomponenten vorgenommen wird.

Spezifikation von Aufgaben

Damit die Produktionsregeln sinnvoll formuliert werden können, wird davon ausgegangen, daß die Aufgabenstellung nicht in natürlicher Sprache, sondern in einer standardisierten Spezifikationssprache vorliegt. Eine Aufgabe wird in einem Schema der Form

Schema: Aufgabe.Semantik
fname : Name der Funktion
finput : Eingangsparameter (Namen und Datentypen)
foutput: Transformationsvorschrift auf den Eingangsparametern und Ergebnistyp der Funktion

repräsentiert. So würde die Aufgabenstellung: "Schreibe eine rekursive Funktion faku, die das Produkt aller natürlicher Zahlen von n bis 1 berechnet, wobei faku von 0 das Ergebnis 1 liefert." folgendermaßen als Spezifikation repräsentiert:

Schema: Aufgabe.Semantik
fname : faku
finput : n:NAT
foutput: (MULT (*all*(n), *range*[n..1]); faku(0)=1): NAT

Die Konstrukte *all* und *range* sind Schlüsselwörter der Spezifikationssprache. *All(x)* bedeutet, daß eine Operation für alle *x*, die in den Grenzen der in *range* angegebenen Werte liegen, ausgeführt werden soll. Weitere Schlüsselwörter der Spezifikationssprache sind *count*, *cond* und *length*. Dabei gibt *count(x)* an, daß eine Operation x-mal ausgeführt werden soll. Dieses Schlüsselwort wird bei einigen endrekursiven Problemen benötigt. In der natürlichsprachigen Aufgabenstellung befinden sich hier Formulierungen wie "Schreibe eine Funktion *abc*, die mit einem Parameter *z* *x*-mal eine Operation *op* durchführt." Das Schlüsselwort *cond* gibt eine Bedingung an, der die Eingangsparameter genügen müssen. Solche zur Terminationsbedingung zusätzlichen Bedingungen, sind typisch für *Filter*-Funktionen (vgl. Kap. 5.2), bei denen geprüft wird, ob ein Eingangsparamter bestimmte Eigenschaften aufweist. Das Schlüsselwort *length* liefert die Länge einer Liste. Die Schlüsselwörter für die Spezifikation sowie die Spezifikation der in den empirischen Untersuchungen verwendeten Aufgaben sind Anhang C.1 zu entnehmen.

Das allgemeine Rekursionsschema

Allgemein können Programme mithilfe von Programm-Schemata beschrieben werden (Elgot 1971; Engelfriet 1974; Greibach 1975). Programm-Schemata sind eine abstrakte Repräsentation von Programmen in einer formalen Sprache. Die Idee der Programm-Schemata wird im folgenden benutzt, um die Struktur rekursiver Funktionen allgemein zu beschreiben. Dabei wird jedoch auf eine formale Einführung der Syntax und Semantik von Programm-Schemata verzichtet.

Gemeinsames Merkmal aller rekursiven Funktionen ist, daß der Funktionsname in der Definition der Transformationsvorschrift selbst verwendet wird. Für terminierende rekursive Funktionen ist zudem eine notwendige (aber nicht hinreichende) Voraussetzung, daß ein

direkter und ein rekursiver Fall unterschieden werden.

Allgemein hat eine rekursive Funktion in Anlehnung an die oben definierte funktionale Sprache also die Form:

```
FUN name (r: tr, p1:t1,...pn:tn): t
IF bottom (r)
THEN wert:t
ELSE [name(r' | r' < r); anweisung]
FIN
```

mit

r	:	Parameter, auf dem die wohlfundierte Ordnungsrelation beschrieben wird, nach der im rekursiven Aufruf minimiert wird
t_i	:	*vordefinierter Datentyp der verwendeten Sprache*
p_i	:	mögliche weitere Funktionsparameter mit $0 \leq i \leq n$
bottom	:	abstrakte boolsche Anweisung zur Prüfung, ob r dem kleinsten Teilproblem entspricht
wert	:	Ausgabe im direkten Fall
[]	:	geschachtelte Anweisung, die in jedem Fall den rekursiven Aufruf und möglicherweise eine weitere Anweisung enthält.
<	:	wohlfundierte Ordnungsrelation

Die oben definierte Fakultätsfunktion läßt sich in ein spezifischeres Schema einordnen, bei dem der rekursive Fall die Struktur *verknüpfer(operation,name(r' | r' < r))* besitzt.

Die im folgenden verwendete Notation der Schemata ist informeller als die oben beschriebene. Zudem werden die einschränkenden Bedingungen für die Belegung der Leerstellen (wie Restriktion von *wert* auf Datentyp *t*, oder Ordnungsrelation bei der Minimierung) nicht an den Leerstellen für die Funktionsstruktur, sondern in den ebenfalls an das Schema gebundenen Regeln angegeben. Nun wird das allgemeinste Schema der postulierten Hierarchie rekursiver Funktionen dargestellt. Anhand dieses Schemas werden die Leerstellen und Regeln eingeführt, die für alle untergeordneten Schemata zunehmend spezifiziert werden. Alle Schemata weisen folgende Struktur auf:

Schema: Rekursion.allgemein
Kopf: FUN *name* (*parlist* = $\{p_i:t_i\}_{i=1..n}$):t_{res}
Rumpf:
Abbruchbedingung: IF *(r = bottom)*
direkter Fall: THEN *wert*
rekursiver Fall: ELSE *verknüpfer*(*hilfsop*($\{p_i\}_{i\le n}$), *name*([$\{hilfsop_i(p_i)\}_{\forall i \text{ mit } p_i \neq r}$, *minop*(*r*)])) FIN

schreibe Rekursion: R1
verfeinere Kopf: R2
verfeinere Rumpf: R3
verfeinere rekursiven Fall: R4,R5,R6,R7,R8
verfeinere Abbruchbedingung und direkten Fall: R9,R10,R11,R12

Das allgemeine Schema einer rekursiven Funktion enthält Leerstellen für den Kopf und den Rumpf einer Funktion. Dabei gliedert sich der Rumpf in Abbruchbedingung, direkten Fall und rekursiven Fall. Die folgenden Leerstellen beinhalten Produktionsregeln zur vollständigen Instantiierung der Konzeptleerstellen (*Kopf, Abbruchbedingung, direkter Fall, rekursiver Fall*). Diese Regeln werden weiter unten noch genauer ausgeführt. Aus Gründen der Vereinfachung werden, wie schon bei der Aufgabenspezifikation, direkt Schlüsselwörter aus der in Kapitel 5.3 vorgestellten einfachen funktionalen Sprache in den Rekursionsschemata verwendet.

Alle *kursiv* gedruckten Symbole sind Platzhalter, die für konkrete Aufgabenstellungen zu instantiieren sind. Dabei wird angenommen, daß für diese schrittweise Instantiierung das spezifischste Schema der Hierarchie verwendet wird. Dieses Schema wird im folgenden als Aufgabe.Rekursion bezeichnet.

Die Leerstellen für die Struktur der rekursiven Funktion repräsentieren Informationen über die verschiedenen Aspekte rekursiver Funktionen: In der Abbruchbedingung wird geprüft, ob der rekursive Parameter bereits mit dem kleinsten Teilproblem identisch ist. Ist dies der Fall, wird das Ergebnis für das kleinste Teilproblem ausgegeben. Der rekursive Fall ist im obigen Schema so allgemein gehalten, daß möglichst viele Varianten rekursiver Funktionen dadurch abgebildet werden: Auf einen oder mehrere der Eingangsparameter kann eine Operation angewendet werden, deren Ergebnis mit dem Ergebnis des rekursiven Aufrufs verknüpft wird. Der rekursive Aufruf selbst enthält alle Eingangsparameter, wobei der als rekursive Parameter identifizierte Parameter minimiert wird. Der rekursive Aufruf repräsentiert das Format einer Teil-Rest-Rekursion. Würde das erste Argument der Verknüpfungsoperation ebenfalls mit einem rekursiven Aufruf belegt, entsteht eine Simultanrekursion. Fallen Verknüpfer und erste Hilfsoperation weg, entsteht eine endrekursive Struktur.

Der Prozeß der schrittweisen Instantiierung wird durch die Produktionsregel R1 eingeleitet:

```
R1:
IF (Ziel = Schreibe rekursive Funktion)
THEN WITH Aufgabe.Rekursion DO
    verfeinere_Kopf        (* Teilziel -> R2 *)
    verfeinere_Rumpf       (* Teilziel -> R3 *)
```

Die Produktionsregeln werden hier halbformal angegeben. Die *WITH x.y*-Notation gibt an, auf welche Komponente der Spezifikation beziehungsweise des aktiven Schemas eine Regel zugreift. Auf eine einführende Spezifikation der in der Regelsprache verwendeten Schlüsselwörter wird verzichtet (siehe dazu Anhang C.2), stattdessen werden die Regeln im Text erläutert. Es wird angenommen, daß sich alle bereits von den Regeln gesetzten Teilziele, das Schema für die Aufgabenspezifikation und das zu instantiierende beziehungsweise partiell instantiierte Rekursionsschema als aktive Elemente im Arbeitsspeicher befinden. In den Regelbedingungen wird jeweils das aktuelle Teilziel abgefragt und die Regeln haben Zugriff auf die Spezifikation und das Rekursionsschema. Aus dem Rekursionsschema können bereits instantiierte Werte abgefragt werden und die Regeln können weitere Instantiierungen im Schema vornehmen. Die jeweils ersten Regeln einer Leerstelle strukturieren den Aufruf der nachfolgenden Regeln, in dem neue Teilziele gesetzt werden. Regel 1 repräsentiert das Wissen, daß zum Erstellen einer rekursiven Funktion der Funktionskopf und der Funktionsrumpf kodiert werden müssen. Diese Aspekte werden als neue Teilziele im Kurzzeitgedächtnis repräsentiert.

Zunächst wird nun der Kopf instantiiert. Hier müssen der Funktionsname (*name*), sowie Namen für die Parameter (p_i), die Datentypen der Parameter (t_i) und den Ergebnistyp der Funktion (t_{res}) eingetragen werden. Dies ist einfach durch Auslesen der entsprechenden Informationen aus der Spezifikation realisierbar:

```
R2:
IF (Ziel = verfeinere Kopf)
THEN WITH Aufgabe.Rekursion DO      (* aktives Schema *)
    name    := Aufgabe.Semantik.fname  (* Spezifikation *)
    WITH Aufgabe.Rekursion.Kopf DO
    FOR i := 1 TO n DO
      (p_i  := Aufgabe.Semantik.finput.parametername_i
       t_i  := Aufgabe.Semantik.finput.parametertyp_i )
      t_res := Aufgabe.Semantik.foutput.ergebnistyp
```

Der Rumpf der Funktion wird schrittweise verfeinert, indem zunächst der rekursive Fall und danach Abbruchbedingung und direkter Fall instantiiert werden. Die hier verwendeten Regeln sind relativ komplex, da sie zur Generierung möglichst vieler Arten rekursiver Funktionen eingesetzt werden sollen. Für die Fakultätsfunktion wäre es angemessener die an das Teil-Rest-Schema gebundenen Regeln zu verwenden.

Regel 3 steuert den Ablauf der Instantiierung der Leerstellen für den Funktionsrumpf.

```
R3:
IF (Ziel = verfeinere Rumpf)
THEN WITH Aufgabe.Rekursion.Rumpf DO
    verfeinere_rekursiven_Fall      (* R4 *)
    verfeinere_direkten_Fall        (* R9 *)
```

Es wird zuerst der rekursive Fall verfeinert, da sich aus den dort ermittelten Strukturen die Wertebelegung für Abbruchbedingung und direkten Fall leicht ermitteln lassen. Die Produktionsregel für die Verfeinerung des rekursiven Falls ist sehr spezifisch für die verschiedenen rekursiven Konzepte der Schemahierarchie. Im allgemeinen Fall lautet die Regel:

```
R4:
IF (Ziel = verfeinere rekursiven Fall)
THEN WITH Aufgabe.Rekursion.rekursiver_Fall DO
    finde_rekursiven Parameter      (* R5 *)
    finde_Minimierung(r)            (* R6 *)
    finde_verknüpfer(t_rs)          (* R7 *)
    finde_Operation                 (* R8 *)
```

Zunächst wird also ermittelt, welcher Parameter in der Rekursion zu minimieren ist. Dieser Parameter wird im folgenden mit *r* bezeichnet. Dieser Schritt entspricht der Zerlegung des Ausgangsproblems in ein strukturgleiches Teilproblem. Regel 5 ermittelt den Parameter wie folgt:

```
R5:
IF (Ziel = finde rekursiven Parameter(r))
THEN WITH Aufgabe.Semantik DO
  IF ParameterZahl(finput) = 1
  THEN r := Parametername
  IF ParameterZahl(finput) > 1
  THEN r := Parametername ϵ ALL in foutput
       OR Parametername ϵ Count in foutput
```

Für die Erstellung der Fakultätsfunktion wird also der einzige Parameter *n* als zu minimierender Parameter identifiziert. Für Aufgaben, die mehr als einen Eingangsparameter verlangen, wird derjenige Parameter ausgewählt, der als Argument der Spezifikationsschlüsselwörter *all* oder *count* erscheint. Dies spiegelt das Expertenwissen wieder, daß ein rekursiver Parameter üblicherweise entweder ein Schleifenparameter oder ein Paramter ist, der mit verschiedenen konkreten Werten in die Berechnung eingeht. Regel 5 zeigt bereits eine Einschränkung der vorgeschlagenen Repräsentationsstruktur: Rekursionen, die mit mehr als einem rekursiven Parameter arbeiten (wie z.B. die Ackermann-Funktion) werden nicht behandelt.

In Regel 4 wird als nächster Verfeinerungsschritt angegeben, daß die Minimierungsoperation für den rekursiven Parameter zu instantiieren ist. Dabei wird das Wissen zugrundegelegt, daß die Teilproblemzerlegung einer wohlfundierten Ordnungsrelation gehorchen muß. Für die Datentypen natürliche Zahlen und lineare Listen gilt, daß *MINUS(r,x)* immer kleiner als *r* ist ($0 < x \leq r$) und daß die Länge der Liste $TAIL^x(r)$ immer geringer ist als die Länge der Liste *r* ($0 < x < Length(r)$). Die Minimpierungsoperation wird durch Regel 6 instantiiert, die den rekursiven Parameter als Argument erhält:

R6:
IF (Ziel = finde Minimierung(r))
THEN
  IF DatenTyp(r) = NAT
  THEN minop := MINUS(r,x) mit $0 < x \leq r$; *default*: x=1
  IF DatenTyp(r) = NATLIST
  THEN minop := $TAIL^x$(r) mit $0 < r \leq Length(r)$; default: x=1

Für die Fakultätsfunktion führt Regel 6 also zur Instantiierung von *minop* mit *MINUS(n,1)*. Im nächsten Schritt wird geprüft, ob es sich um eine teil-rest-rekursive Funktion handelt, also, ob ein Verknüpfer benötigt wird. Dies erledigt Regel 7:

R7:
IF (Ziel = finde Verknüpfer)
THEN WITH Aufgabe.Semantik.foutput DO
  IF IsArgument(all,Operation)
  THEN verknüpfer := Operation
  ELSE verknüpfer := []

Regel 7 ist für den allgemeinen Fall relativ vage, da sowohl end- als auch teil-rest-rekursive Fälle berücksichtigt werden müssen. Die Regel beschreibt, daß in der Spezifikation geprüft wird, ob der rekursive Parameter (*all(r)*) als Argument einer Operation verwendet wird. Ist dies der Fall, so wird die Operation aus der Spezifikation übernommen, ansonsten wird

angenommen, daß es sich um eine Endrekursion (*verknüpfer* = *[]*) handelt. Eine Überprüfung der Konsistenz der Operation mit dem verlangten Ergebnistyp der Funktion erfolgt hier nicht. Für die Fakultätsfunktion wird entsprechend die Operation *MULT* als Verknüpfer identifiziert.

Nun ist der rekursive Aufruf fast vollständig instantiiert. Im Falle einer Teil-Rest-Rekursion muß nun die erstgenannte Hilfsoperation ermittelt werden, im Falle einer Endrekursion mit mehr als einem Eingangsparameter muß die Hilfsoperation auf diesem Funktionsargument bestimmt werden. Dies geschieht in Regel 8:

```
R8:
IF (Ziel = Finde Operation)
THEN WITH Aufgabe.Semantik.foutput DO
  IF verknüpfer ≠ []
  THEN
    IF isArgument(Operation,verknüpfer)
    THEN hilfsop := Operation
    ELSE hilfsop := id(r)
  IF verknüpfer = []
  THEN
    IF ParameterZahl(Aufgabe.semantik.finput) > 1
    THEN hilfsop := GetOperation
    ELSE hilfsop := []
```

Bei der Darstellung dieser Regel wurde aus Übersichtsgründen auf die Prüfung der Datentyp-Konsistenz verzichtet. Der Ergebnistyp der ermittelten Operation muß jeweils dem entsprechenden Argumenttyp entsprechen, den die umschließende Funktion (Verknüpfer oder rekursiver Aufruf) verlangt. Für den Fall einer Teil-Rest-Rekursion wird geprüft, ob eine weitere Operation neben der als Verknüpfer identifizierten in der Spezifikation verlangt wird. Ist dies der Fall, so wird diese Operation zur Instantiierung von *hilfsop* verwendet, ansonsten wird der rekursive Parameter (*id(r)*) als erstes Argument der Verknüpfungsoperation gesetzt. Als Operation werden hier auch Konstanten zugelassen, so daß zum Beispiel die Konstruktion einer Liste, die *n*-mal die Zahl *7* enthält, mit dieser Regel darstellbar wird. Für die Fakultätsfunktion wird keine weitere Operation in der Spezifikation ermittelt und damit der Parameter n als erstes Argument der Multiplikation gewählt.

Die Instantiierung des rekursiven Falls gestaltet sich, wie gesehen, mit den an das allgemeine Rekursionsschema gebundenen Regeln sehr aufwendig. Natürlich haben Experten spezifischere Regeln für die verschiedenen rekursiven Konzepte zur Verfügung. Nun wenden wir uns der Instantiierung von Abbruchbedingung und direktem Fall zu. Diese Instantiierung wird durch Regel 9 initiiert:

```
R9:
IF (Ziel = verfeinere direkten Fall)
THEN WITH Aufgabe.Rekursion.Rumpf DO
  finde_kleinstes_Teilproblem(r)    (* R10 *)
  Finde_Prüfbedingung(bottom)       (* R11 *)
  Finde_Ergebnis(bottom)            (* R12 *)
```

Zunächst wird das kleinste Teilproblem ermittelt, also der kleinste Wert, den der rekursive Parameter annehmen kann.

```
R10:
IF (Ziel = Finde kleinstes Teilproblem)
THEN WITH Abbruchbedingung DO
  IF DatenTyp(r) = NAT
  THEN bottom :=
       Minimum(Aufgabe.Semantik.foutput.range) (* default = 0 *)
  IF DatenTyp(r) = NATLIST
  THEN WITH Aufgabe.Semantik.foutput DO
    IF ALL(r) AND hilfsop(r) ≠ TAIL
    THEN bottom := 0
    IF ALL(r) AND hilfsop(r) = TAIL
    THEN bottom := 1
    IF verknüpfer=[] AND minop = TAIL^x
    THEN bottom := Length(wert)
```

Ist der rekursive Parameter eine natürliche Zahl, so wird der kleinere Wert des in der Spezifikation angegebenen Bereichs (*range*) als kleinstes Teilproblem identifiziert. Für die Fakultätsfunktion wird also die *null* als kleinstes Teilproblem identifiziert. Ist der rekursive Parameter eine Liste, so müssen verschiedene Fälle unterschieden werden. Für die Fälle, in denen die Spezifikation verlangt, daß die gesamte Liste abgearbeitet wird (*all(r)*), wird geprüft, ob zusätzlich zur Minimierung von *r* auch in der Hilfsoperation eine kleinere Teilliste (*TAIL(r)*) verwendet wird. Ist dies nicht der Fall, so wird angenommen, daß das kleinste Teilproblem die leere Liste ist, ansonsten wird die einelementige Liste als kleinstes Teilproblem identifiziert. Hier fließt Wissen über Listenstrukturen in die Regel ein: In der Hilfsoperation wird üblicherweise eine Elementselektion (*HEAD*) oder eine Rechenoperation ausgeführt. Beides ist nicht auf leeren Listen möglich, der Tail einer einelementigen Liste ist jedoch die leere Liste. Wird in der Spezifikation nicht verlangt, daß eine Operation auf allen Listenelementen durchgeführt wird, wie das bei *map*-Strukturen und bei *filter*-Strukturen notwendig ist (vgl. Kap. 5.3), sondern, daß ein Element der Liste ausgewählt wird (*reduce*-Struktur), und handelt es sich zusätzlich um eine endrekursive Struktur, so muß zunächst ermittelt werden, welche Ausgabe im direkten Fall erwartet wird. Das kleinste

Teilproblem entspricht damit dem verlangten Rückgabewert. Das klassische Beispiel für diesen Fall ist eine Funktion, die das letzte Element einer Liste liefert. Hier wird die Liste solange verkürzt, bis sie nur noch ein Element enthält, das die Funktion zurückliefert. Auch für Listen wird bottom mit einer natürlichen Zahl instantiiert. Diese Zahl gibt die Länge der Liste für den direkten Fall an.

Die Ermittlung der Abbruchbedingung ist einfach:

```
R11:
IF (Ziel = Finde Prüfbedingung(bottom))
THEN
  IF DatenTyp(r) = NAT
  THEN Abbruchbedingung := EQUAL(r,bottom)
  IF Datentyp(r) = NATLIST
  THEN Abbruchbedingung := EMPTY(TAIL*(r)) mit * = bottom
```

Ist der rekursive Parameter eine natürliche Zahl, so wird in der Abbruchbedingung geprüft, ob der rekursive Parameter gleich dem als kleinstes Teilproblem (*bottom*) identifizierten Wert ist. Für die Fakultätsfunktion wird also geprüft, ob der rekursive Parameter *n* den Wert *null* hat. Ist der rekursive Parameter eine Liste, so wird der in Regel 10 in *bottom* gespeicherte Wert für die minimale Länge der Liste verwendet, um die Prüfbedingung zu formulieren. Ist die Länge *null*, so entfällt die *TAIL*-Operation.

Nun ist das Schema bis auf die Rückgabe im direkten Fall instantiiert. Die Rückgabe kann direkt aus der Spezifikation entnommen werden, wenn die Rückgabe für den direkten Fall explizit in der Aufgabenstellung gegeben ist. Für die Fakultätsfunktion ergibt sich also der Wert *1*. Ansonsten ist die Rückgabe aus der Spezifikation errechenbar, indem in der Zeile foutput das kleinste Teilproblem eingesetzt wird. Scheitert die Berechnung wird als default das kleinste Element des Datentyps des Funktionsergebnisses verwendet.

```
R12:
IF (Ziel = Finde Ergebnis(bottom))
THEN WITH direkter_Fall DO
  IF isGiven(dF in Aufgaben.Semantik.foutput)
  THEN wert := df
  ELSE wert := Calculate(Aufgabe.Semantik.foutput(r=bottom),ok)
  IF NOT ok
  THEN
    IF tres = NAT THEN wert := 0
    IF tres = NATLIST THEN wert := NIL.
```

Wird die Teil-Rest-Rekursion verwendet, so fliest in die Regel 12 das Wissen ein, daß das neutrale Element bezüglich der Verknüpfungsoperation als Wert zurückgegeben wird, also null bei der Addition, eins bei der Multipliktation und die leere Liste bei der Listen-Konstruktion (*CONS*).

Zusammenfassend stellt sich der Prozeß der schrittweisen Instantiierung des Schemas durch die an das Schema gebundenen Regeln folgendermaßen dar:

Kopf:	FUN *name* (*parlist* = $\{p_i{:}t_i\}_{i=1..k}$):t_{res}
Rumpf:	
Abbruchbedingung:	IF *(r = bottom)*
direkter Fall:	THEN *wert*
rekursiver Fall:	ELSE *verknüpfer(hilfsop*($\{p_i\}_{i\le n}$), *name*([$\{hilfsop_i(p_j)\}_{\forall i \text{ mit } p_i \text{ in } r}$, *minop*(r)])) FIN

↓ R2

Kopf:	FUN faku (n:NAT):NAT
Rumpf:	
Abbruchbedingung:	IF *(r = bottom)*
direkter Fall:	THEN *wert*
rekursiver Fall:	ELSE *verknüpfer(hilfsop*($\{p_i\}_{i\le n}$), faku([$\{hilfsop_i(p_j)\}_{\forall i \text{ mit } p_i \text{ in } r}$, *minop*(r)])) FIN

↓ R5

Kopf:	FUN faku (n:NAT):NAT
Rumpf:	
Abbruchbedingung:	IF *(n = bottom)*
direkter Fall:	THEN *wert*
rekursiver Fall:	ELSE *verknüpfer(hilfsop*($\{p_i\}_{i\le n}$), faku([$\{hilfsop_i(p_j)\}_{\forall i \text{ mit } p_i \text{ in } r}$, *minop*(n)])) FIN

↓ R6

Kopf:	FUN faku (n:NAT):NAT
Rumpf:	
Abbruchbedingung:	IF *(n = bottom)*
direkter Fall:	THEN *wert*
rekursiver Fall:	ELSE *verknüpfer(hilfsop*($\{p_i\}_{i\le n}$), faku(MINUS(n,1))) FIN

↓ R7

Kopf:	FUN faku (n:NAT):NAT
Rumpf:	
Abbruchbedingung:	IF *(n = bottom)*
direkter Fall:	THEN *wert*
rekursiver Fall:	ELSE MULT(*hilfsop*($\{p_i\}_{i\le n}$), faku(MINUS(n,1))) FIN

↓ R8

Kopf:	FUN faku (n:NAT):NAT
Rumpf:	
Abbruchbedingung:	IF *(n = bottom)*
direkter Fall:	THEN *wert*
rekursiver Fall:	ELSE MULT(n, faku(MINUS(n,1))) FIN

↓ R10

```
Kopf:                   FUN faku (n:NAT):NAT
Rumpf:
  Abbruchbedingung:     IF (n = 0)
  direkter Fall:        THEN wert
  rekursiver Fall:      ELSE MULT(n, faku(MINUS(n,1))) FIN

↓ R11

Kopf:                   FUN faku (n:NAT):NAT
Rumpf:
  Abbruchbedingung:     IF EQUAL(n,0)
  direkter Fall:        THEN wert
  rekursiver Fall:      ELSE MULT(n, faku(MINUS(n,1))) FIN

↓ R12

Kopf:                   FUN faku (n:NAT):NAT
Rumpf:
  Abbruchbedingung:     IF EQUAL(n,0)
  direkter Fall:        THEN 1
  rekursiver Fall:      ELSE MULT(n, faku(MINUS(n,1))) FIN
```

Bereits mit der gerade dargestellten Schemastruktur, die die allgemeinen Charakteristika rekursiver Funktionen repräsentiert, können verschiedene Fähigkeiten eines Experten modelliert werden. Der Prozeß des Erstellens einer rekursiven Funktion für eine Problemspezifikation (Aufgabenstellung) kann als Auswahl der jeweils spezifischsten passenden Regeln und deren Anwendung zur Instantiierung der zum selben Schema gehörigen Konzeptleerstellen beschrieben werden. Die Erläuterung einer Aufgabenlösung kann mithilfe des verwendeten Schemas und seiner Stellung in der Schemahierarchie beschrieben werden. Zudem kann auf Fragen zu rekursiven Konzepten mit den in der Schemahierarchie verwendeten Merkmalen und Konzepten argumentiert werden. Auch die Fähigkeit, vorliegende rekursive Funktionen zu beschreiben, kann mithilfe der hierarchischen Schemastruktur modelliert werden. Konkrete Funktionen aktivieren über Mustervergleich das spezifischste instantiierbare Schema, wodurch diese Funktionen klassifizierbar werden.

In der dargestellten Entwicklung der Fakultätsfunktion wurde allerdings das allgemeinste Schema (Wurzel der Hierarchie) verwendet. Auch mit diesem allgemeinen Schema ist also die Kodierung einer rekursiven Aufgabe prinzipiell möglich, sie gestaltet sich aber aufwendiger, als bei Verwendung eines spezifischeren Schemas, da dann die Wertebereiche der Funktionskomponenten erstens eingeschränkter sind und zweitens bereits Teile des Schemas vorinstantiiert sind.

Die Schemahierarchie

Ausgehend von dem oben dargestellen allgemeinen Rekursionsschema als Wurzel, werden durch zunehmende Vorinstantiierung der Konzeptleerstellen und zunehmende Spezialisierung der Regeln Schemata für spezifischere Rekursionskonzepte generiert. Die von mir vorgeschlagene Schemahierarchie weicht von anderen Vorschlägen (vgl. Kap. 5.3; Vorberg & Goebel 1991) an einigen Stellen ab. Diese Abweichungen kommen dadurch zustande, daß ich mich um eine Klassifikation rekursiver Funktionen bemüht habe, bei der rein strukturelle Merkmale, die direkt aus der Oberflächenstruktur ableitbar sind, im Mittelpunkt stehen. Die Wurzel der Hierarchie kann nach Rosch (1978) als Schema der Oberklasse "Rekursive Funktionen" aufgefaßt werden. Die ersten Nachfolger - Endrekursion und Teil-Rest-Rekursion - als Basiskonzepte und alle darunter liegenden Strukturen als entsprechende Unterkonzepte. Die Basiskonzepte werden analog zu Rosch als wesentliche Strukturen zur Klassifikation rekursiver Funktionen angesehen. Bei der Darstellung der Schemahierarchie (siehe Abb. 8) werden hier nur die Bezeichnungen der Schemata angegeben, eine Darstellung verschiedener Strukturen sowie der schemaspezifischen Regeln ist Anhang C.2 zu entnehmen.

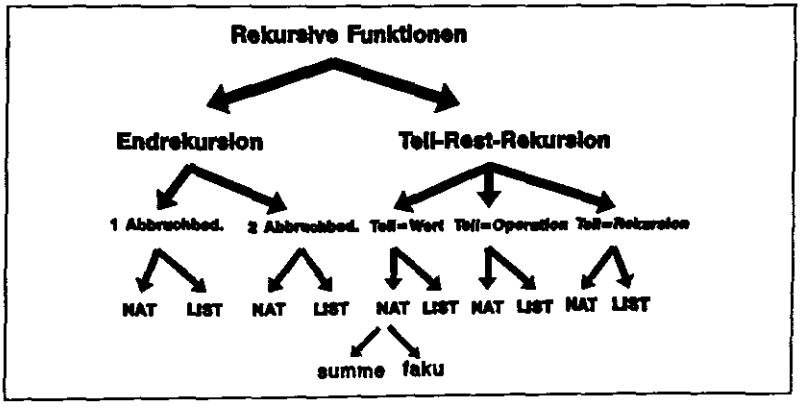


Abb. 8: Hierarchie von Rekursionsschemata

Die Endrekursion enthält als Unterkonzepte spezifische Schemata für Funktionen mit einer oder zwei Abbruchbedingungen. Die Teil-Rest-Rekursiven Funktionen unterscheiden sich nach der Struktur des ersten Argumentes der Verknüpfungsoperation. Das erste Argument kann ein Wert sein (wie bei der Fakultät) oder aber eine Operation. Diese Operation kann auf dem rekursiven oder auch anderen Parametern definiert sein. Wird als Operation wieder die rekursive Funktion selbst eingesetzt, ergibt sich das Schema für die Simultanrekursion.

Damit wird Simultanrekursion mit unter das Schema für lineare Rekursion eingeordnet, obwohl damit nach Gesichtspunkten der Berechenbarkeit (vgl. Kap. 5.2 und 5.3) die Klasse der primitiv-rekursiven Funktionen verlassen wird. Auf der nächsten Ebene werden die Funktionen danach unterschieden, ob sie nur auf natürlichen Zahlen oder auch auf Listen arbeiten. Daran anschließend können konkrete rekursive Funktionen als Schemata repräsentiert sein. So haben Experten sicher Funktionen, wie die Fakultätsfunktion und gängige Such- und Sortieralgorithmen, direkt in abrufbarer Form repräsentiert. Damit werden konkrete Beispiele und abstrakte Pläne in einer einzigen Repräsentationsstruktur zusammen mit Problemlösemethoden (Produktionsregeln) integriert (vgl. Carbonell 1986, S. 373).

So wie die konzeptuellen Leerstellen der Schemata von Ebene zu Ebene immer stärker durch Vorinstantiierungen eingeschränkt werden, werden auch die an das Schema gebundenen Regeln immer spezifischer. In Anhang C.2 werden die Regeln für die spezifischeren Schemata aufgeführt.

Die hier vorgeschlagenen Merkmale zur Klassifikation rekursiver Funktionen erheben keinen Anspruch auf Vollständigkeit, sondern sollen lediglich das Prinzip verdeutlichen, nach dem rekursive Funktionen nach verschiedenen Kriterien klassifiziert werden können. Die Merkmale, die die Schemahierarchie konstituieren, hängen von der Lerngeschichte des Experten ab, also von der Menge der rekursiven Probleme, mit denen er konfrontiert wurde.

In der hier vorgeschlagenen Hierarchie werden folgende Merkmale verwendet:
- Rekursionstyp: (1) Endrekursion oder (2) Teil-Rest-Rekursion
- Art der Teil-Rest-Rekursion: (1) Das Argument, das mit dem rekursiven Aufruf verknüpft wird, ist eine Konstante oder ein Parameter. (2) Das Argument, das mit dem rekursiven Aufruf verküpft wird, ist eine Operation auf Parametern und/oder Konstanten. (3) Das Argument, das mit dem rekursiven Aufruf verknüpft wird, ist selbst ein rekursiver Aufruf.
- Anzahl der Abbruchbedingungen: (1) Eine Abbruchbedingung oder (2) zwei Abbruchbedingungen.
- Datentyp: Die Funktion ist (1) auf natürlichen Zahlen oder (2) auf Listen oder Listen und natürlichen Zahlen definiert.

Für die Teil-Rest-Rekursion werden verschiedene Spezialisierungen angegeben, die nur bei diesem Typ von Rekursion möglich sind. Die Merkmale "Anzahl der Abbruchbedingungen" und "Datentyp" sind dagegen zur Klassifikation aller rekursiven Funktionen anwendbar, wobei sowohl Funktionen mit mehr als zwei Abbruchbedingungen, als auch mit anderen Datentypen denkbar sind. Daß in der obigen Hierarchie nur im Zweig der Endrekursiven Funktionen die Anzahl der Abbruchbedingungen als Merkmal verwendet wurde, liegt daran, daß diese Hierarchie sich speziell auf die in den Untersuchungen (vgl. Kap. 8) verwendeten Funktionen bezieht und dort nur teil-rest-rekursive Funktionen mit einer Abbruchbedingung

verwendet wurden.

Für den Prozeß der Aktivierung eines Schemas für eine konkrete Aufgabenstellung (Spezifikation) werden folgende Regeln verwendet:

(1) Wenn in der Spezifikation eine Zählvariable vorkommt (Schlüsselwort *count*) dann gehe in den Zweig der endrekursiven Funktionen, sonst wähle als *default* den Zweig der teil-rest-rekursiven Funktionen.
(2) Wenn bereits der Zweig der teil-rest-rekursiven Funktionen gewählt wurde und in der Spezifikation eine Operation als Argument des Ausdrucks der die Funktion spezifiziert vorkommt, dann wähle das Schema Teil=Operation, wenn der Name der Funktion zweimal als Argument der Spezifikation erscheint, dann wähle das Schema Teil=Rekursion (Simultan-Rekursion), sonst wähle das Schema Teil= Wert.
(3) Wenn in der Spezifikation eine *Filter*-Bedingung angegeben wird (Schlüsselwort *cond*), dann wähle ein Schema mit zwei Abbruchbedingungen, sonst wähle als *default* eine Abbruchbedingung.
(4) Wenn in der Spezifikation für Eingangsparameter sowie Ergebnistyp der Funktion nur natürliche Zahlen vorkommen, dann wähle ein Schema, das nur für natürliche Zahlen definiert ist, sonst wähle ein Schema, das für Listen definiert ist.

Repräsentation von aktuellen Problemlöseprozessen

Bisher wurde am Beispiel der Fakultätsfunktion dargestellt, wie bei Vorhandensein einer Experten-Wissensstruktur über Rekursion aktuelle Aufgaben gelöst werden können. Dieser Prozeß soll nun abstrakter angegeben werden. Dabei wird zunächst dargestellt, wie ein Experte ausgehend von einer Aufgabenspezifikation einen lauffähigen Programmcode erstellt. Danach wird gezeigt, wie Experten andere kognitive Leistungen im Bereich Rekursion mit der vorgeschlagenen Wissensstruktur erbringen können.

Allgemein nehme ich an, daß ausgehend von einer Aufgabenstellung die dafür spezifischsten Wissensstrukturen im Langzeitgedächtnis aktiviert werden. Im Falle der von mir vorgeschlagenen Repräsentation wird also ein Rekursionsschema zusammen mit den daran gebundenen Produktionsregeln aktiviert. Ein aktives Rekursionsschema wird von mir als mentales Modell im Sinne von Johnson-Laird (1983) und Holland, Holyoak, Nisbett und Thagard (1987, Kap. 2) bezeichnet. Das aktive Schema repräsentiert erstens die **Struktur** einer rekursiven Funktion und wird zweitens, mit den daran gebundenen Produktionsregeln, **dynamisch verändert**. Wie im vorangegangenen Abschnitt dargestellt, wird das partiell instantiierte Schema solange verfeinert, bis eine vollständig definierte rekursive Funktion vorliegt.

Eine abstrakte Darstellung des Lösens rekursiver Aufgaben durch Aktivation eines Schemas mit daran gebundenen Regeln gibt Abbildung 9.

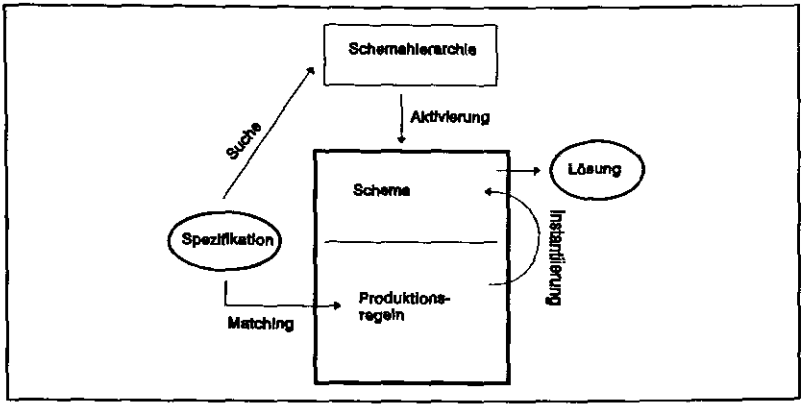


Abb. 9: Prozeß der rekursiven Programmierung bei Verwendung der Schema-Regel-Struktur

Eigentlich beginnt der dargestellte Prozeß mit der Präsentation einer natürlichsprachigen Aufgabenstellung. Diese Aufgabe kann von einer zweiten Person gegeben werden, oder der Programmierer kann sie sich selbst stellen. Die Aufgabe wird im nächsten Schritt als Spezifikation (geforderte Semantik des zu schreibenden Programms) repräsentiert. Diese Spezifikation ist *Input* in das hier vorgeschlagene Modell. Die Repräsentation der Spezifikation instantiiert einen Suchprozeß in der im Langzeitgedächtnis gespeicherten Schemahierarchie. Hier gehe ich davon aus, daß bei Experten eine effiziente Suchstrategie vorhanden ist: So führt das Vorkommen einer Zählvariable (Schlüsselwort *count* in der Spezifikation) zum Beispiel direkt zu einer Suche im Ast der endrekursiven Strukturen. In Fällen, in denen aus der Aufgabenspezifikation kein Hinweis auf die Struktur der Rekursion ableitbar ist, wird als default-Wert das Teil-Rest-Schema aktiviert, da die meisten rekursiven Funktionen auf dieser Struktur basieren.

Jedes Schema besteht aus zwei Komponenten: Den Leerstellen für die Funktionsstruktur und den Leerstellen für die Regelstruktur. Da beim Experten die an ein Schema gebundenen Regeln vollständig vorhanden sind und hinreichen, eine rekursive Funktion zu kodieren, werden hier in Übereinstimmung mit Anderson's ACT*-Theorie (1983) diese Regeln als dominante Wissensquelle der Problemlösung betrachtet. Die Regeln beziehen zunächst Information aus der Aufgabenspezifikation, um das aktivierte Schema zu instantiieren. Sobald das Schema partiell instantiiert ist, greifen die Regeln zusätzlich auf Werte in der Funktionsstruktur zu, um diese weiter zu belegen. Der Prozeß der Regelanwendung wird solange fortgesetzt, bis die Funktionsstruktur vollständig definiert ist.

Das Ziel des Modells ist es, abzubilden, wie rekursive Aufgaben von Anfängern gelöst werden und wie dies zu einer Erweiterung der Wissensstruktur führt. Prinzipiell könnte das Modell einfach dahingehend erweitert werden, zu erklären, wie andere kognitive Aufgaben in Bezug auf rekursive Funktionen bewältigt werden. Die von mir vorgeschlagene Wissensstruktur bildet hierfür eine ausreichende Basis. So könnte zum Beispiel modelliert werden, wie ein Experte nach Lösung einer Aufgabe erklären kann, wie er zu der Lösung kam: Die Funktionsstruktur des Schemas ermöglicht es, die einzelnen Instantiierungsprozesse zu verbalisieren. Eine weitere modellierbare Leistung ist das Beurteilen vorliegender, also nicht selbst programmierter rekursiver Funktionen. Mit Beurteilen ist zunächst das Klassifizieren von Funktionen nach problemrelevanten Merkmalen gemeint. Dies ist einfach durch Einordnen der Funktion in die Schemahierarchie möglich. Darauf aufbauend könnte auch das Erschließen der Semantik einer Funktion modelliert werden. Zu diesem Zweck müßte die vorgeschlagene Repräsentationsstruktur jedoch erweitert werden. Prinzipiell müßten die Regeln, die ausgehend von der Spezifikation einer Aufgabe zur Erzeugung eines Funktionscodes führen, "rückwärts" angewendet werden, also von der im Aktionsteil angegebenen Kodiervorschrift auf die Anwendungsbedingungen und somit auf die Spezifikation verweisen.

Das bisher dargestellte Modell bildet ab, wie Personen, die bereits eine Menge von Wissen über Rekursion erworben haben, dieses Wissen im Langzeitgedächtnis repräsentieren und zur Lösung neuer Aufgaben anwenden können. Thema der Arbeit ist jedoch die Induktion neuen Wissens. Das bisher dargestellte Modell dient als Zielvorstellung für den Wissenserwerb und liefert die strukturelle Basis für die Modellierung des Lernprozesses, auf die in den folgenden Abschnitten eingegangen wird.

6.3 Aufgabenlösung und Wissensinduktion beim Programmieren mit Beispielen

Dieser Abschnitt dient der Darstellung von Wissensstrukturen bei Novizen im Bereich rekursive Programmierung. Dabei wird davon ausgegangen, daß ein Novize zunächst sein Wissen über rekursive Funktionen in einem einzigen, nur rudimentär strukturierten Schema repräsentiert hat. Dieses Schema ist unterspezifiziert, so daß eine Aufgabenstellung nicht ohne Zugriff auf weitere Information zu bewältigen ist. Als zusätzliche Information wird hier eine Beispielfunktion vorgegeben. Der Anfänger muß also - im Gegensatz zum Experten - zusätzlich Informationen über eine Beispielfunktion im Kurzzeitgedächtnis halten. Eine Aufgabe wird nicht mehr allein durch Anwendung von Regeln zur Instantiierung eines Schemas gelöst, sondern die fehlenden Wissensstrukturen müssen über Prozesse des anlogen Transfers inferiert werden. Als Ergebnis des analogen Transfers wird ein neues Schema inferiert, das die strukturellen Gemeinsamkeiten von Aufgabe und Beispiel repräsentiert. Zudem werden neue Produktionsregeln generiert, die eine Abstraktion des Lösungsprozesses von Aufgabe und Beispiel darstellen.

Man kann beim Erwerb von derart bereichsspezifischem Wissen, welches zudem die Beherrschung eines neuen Formalismus - nämlich einer Programmiersprache - verlangt, nicht davon ausgehen, daß der Lernprozeß ausschließlich durch analogen Transfer von Beispielen gelingt. Einige Informationen zur Struktur und Funktionsweise rekursiver Funktionen sollten einem Neuling direkt verbal (Vorlesung, Buch) vermittelt werden. Im folgenden wird davon ausgegangen, daß die Lernenden bereits die Syntax der in Kapitel 5.3 definierten einfachen funktionalen Sprache beherrschen und in der Lage sind, einfache nicht rekursive Funktionen in dieser Sprache zu definieren. Damit verfügen sie bereits über das Wissen, daß Funktionen aus einem Kopf und einem Rumpf bestehen, wobei im Kopf der Name der Funktion, die Namen und Typen der Eingangsparameter sowie der Typ des Funktionsergebnisses definiert werden und im Rumpf eine, möglicherweise verschachtelte Anweisung kodiert werden muß, die in Abhängigkeit von den Eingangsparametern das Funktionsergebnis definiert.

Zudem wird vorausgesetzt, daß die Lernenden einige grundlegende mathematische Kenntnisse besitzen, wie zum Beispiel die Beherrschung der arithmetischen Grundoperationen. Es wird weiter angenommen, daß Anfänger folgende Grundinformationen über rekursive Funktionen erhalten, bevor sie versuchen, die erste rekursive Funktion zu kodieren: Erstens sollte ihnen mitgeteilt werden, daß rekursive Funktionen stets einen Aufruf ihrer selbst enthalten. Zweitens sollte ihnen verdeutlicht werden, daß rekursive Funktionen immer mit Hilfe bedingter Anweisungen definiert werden, wobei für einen Fall direkt ein Ergebnis angegeben wird und für alle anderen Fälle das Ergebnis rekursiv berechnet wird. Schließlich sollte ihnen bekannt sein, daß die Abbruchbedingung der bedingten Anweisung nur erreicht

werden kann, wenn der Parameter, der in der bedingten Anweisung erscheint, im rekursiven Aufruf in die entsprechende Richtung verändert wird. Anfänger sollten also über folgendes Wissen über Rekursion verfügen:

Schema: Rekursion.allgemein
Kopf: FUN *name* (*parlist* = $\{p_i{:}t_i\}_{i=1..n}$):t_{res}
Rumpf:
  Abbruchbedingung: IF <*boolsche Anweisung auf einem* $p_i \in$ *parlist*>
  direkter Fall: THEN *wert*
  rekursiver Fall: ELSE <*Anweisung mit name (Minimierung eines* $p_i \in$ *parlist)*>
FIN
schreibe Rekursion: R1
verfeinere Kopf: R2
verfeinere Rumpf: R3
  verfeinere rekursiven Fall: R4'
  verfeinere Abbruchbedingung und direkten Fall: R9'

Die allgemeine Information über die Struktur rekursiver Funktionen soll bekannt sein. Anstelle der konkreten Vorgabe der Konzeptstrukturen (vgl. Kap. 6.2) sind die Leerstellen hier mit sehr vagen Informationen belegt. Ein korrekt erworbenes Konzept ist erst dann vorhanden, wenn die entsprechende Regel zur Belegung dieses Konzeptes erworben ist. Zu Beginn verfügt ein Lernender lediglich über die Regeln, daß eine rekursive Funktion zu schreiben ist (R1), über die notwendigen Schritte zur Kodierung eines Funktionskopfes (R2) und über das Wissen, daß bei der Programmierung des Rumpfes (R3) der rekursive (R4') und der direkte Fall zusammen mit der Abbruchbedingung (R9') kodiert werden muß. Regeln 4 und 9 sind damit lediglich rudimentär vorhanden. Das Ziel (*IF bedingung*) ist klar, der Weg dorthin (*THEN aktion*) nicht.
Alle weiteren Regeln zur Kodierung des Rumpfes der Funktion sind noch nicht erworben.

Soll nun eine rekursive Funktion programmiert werden, ist der Anfänger auf zusätzliche Information, wie etwa eine strukturgleiche Beispielfunktion, angewiesen. Nehmen wir an, es soll eine Funktion *summe* programmiert werden, die die Summe aller Zahlen von *null* bis *n* ausgibt:

Schema: Aufgabe.Semantik
fname : summe
finput : *n*:NAT
foutput: (PLUS (*all*(n), *range*[n..0]); sum(0)=0): NAT

Das einzig vorhandene Rekursionsschema wird aktiviert. Regeln 1, 2 und 3 werden ausgeführt, so daß die Funktion nun folgendermaßen vorliegt:

Schema: Rekursion.allgemein
Kopf: FUN summe (n:NAT):NAT
Rumpf:
Abbruchbedingung: IF *<boolsche Anweisung auf n>*
direkter Fall: THEN *wert*
rekursiver Fall: ELSE *<Anweisung mit* summe *(Minimierung von n)>* FIN

Würde der Lernende keine weitere Information erhalten, so könnte er versuchen, durch Ausprobieren ans Ziel zu kommen. Die Erfolgswahrscheinlichkeit bei diesem Vorgehen wäre jedoch sehr gering. In einer natürlichen Lernsituation würde für die Lösung der ersten Aufgaben sicher entweder ein Tutor oder Dozent Beispiele vorgeben, in für das Selbststudium geeigneten Lehrbüchern würden ebenfalls Beispielprogramme angegeben. Natürlich hängt der Lösungserfolg und auch der Lerngewinn zu diesem frühen Stadium stark von der Art der zur Verfügung stehenden Beispiele ab. Nehmen wir an, der Lernende würde als Beispiel die Faktultätsfunktion erhalten. Für das Beispiel liegt sowohl die Semantik, als auch die Lösung vor. Die Semantik des Beispiels würde in einer natürlichen Lernsituation verbal beschrieben (vgl. Kap. 6.2), hier wird wieder direkt die Spezifikation angegeben:

Schema: Beispiel.Semantik
fname : faku
finput : n:NAT
foutput: (MULT (*all*(n), *range*[n..1]); faku(0)=1): NAT

Die Lösung der Fakultätsfunktion wird in einer zweiten Instantiierung des Rekursionsschemas repräsentiert:

Schema: faku.Rekursion.allgemein
Kopf: FUN faku (n:NAT):NAT
Rumpf:
Abbruchbedingung: IF EQUAL(n,0)
direkter Fall: THEN 1
rekursiver Fall: ELSE MULT(n, faku(MINUS(n,1))) FIN

Nun liegen dem Anfänger also Aufgabenspezifikation, Beispielspezifikation und Beispiellösung vor, um das Rekursionsschema für die Aufgabenlösung zu instantiieren. Der weitere Prozeß wird von bereits vorhandenen allgemeinen Strategien des Vergleichs von Aufgabe und Beispiel (*Mapping*), sowie der Anpassung des Beispiels an die Aufgabe (*Adaptation*) bestimmt (vgl. Kapitel 4; Novick & Holyoak 1992). Dabei wird prinzipiell folgendes Vorgehen angenommen:

1) Finden des zu einem Teil der Syntax des Beispiels passenden Teils in der Spezifikation des Beispiels:
   bsp.rekursion -> bsp.spezifikation

2) Finden eines zur Beispielspezifikation identischen oder strukturgleichen Teils in der Aufgabenspezifikation:
   bsp.spezifikation -> aufgabe.spezifikation

3) Mapping:
   bsp.spezifikation : bsp.rekursion :: aufgabe.spezifikation : ?

Nehmen wir an, daß ein Anfänger, im Gegensatz zu einem Experten, zunächst versucht, die Funktion "von oben nach unten", also ausgehend von der Abbruchbedingung zu kodieren (vgl. Anderson, Conrad & Corbett 1989). Dieses Vorgehen kann auch bei Experten häufig beobachtet werden, man kann jedoch vermuten, daß dabei intern bereits die Struktur der Rekursion erschlossen wurde. Nun versucht der Anfänger, die Abbruchbedingung und den direkten Fall bei Beispiel und Aufgabe zu vergleichen. Dabei geht er so vor, daß er versucht, die in der Syntax des Beispiels vorgegebenen Symbole in der Spezifikation der Aufgabe wiederzufinden:

Rekursion.faku		*Spezifikation.faku*
EQUAL(n,0)	:	range[n..0]

In der Spezifikation der Aufgabe sind analoge Teile zu den bei *faku* ermittelten identifizierbar: *range[n..0]* wird dort ebenfalls gefunden. Damit führt der Vergleich von Aufgabe und Beispiel zu folgendem Ergebnis:

faku		summe
range(n..0) : EQUAL(n,0)	::	range[n,0] : ?

Da die Spezifikation hier für Fakultät und Summe denselben Range angibt, wird ein vollständiges Mapping auf Identität durchgeführt:

EQUAL	-> EQUAL
n	-> n
0	-> 0

Für den direkten Fall werden Unterschiede in den Spezifikationen identifiziert:

faku		summe
faku(0)=1 : 1	::	*summe*(0)=0 : ?

Das Mapping führt also zu

1 -> 0.

Dadurch entstehen folgende vorläufige Regeln:

R10/11':
IF (Ziel = Finde Prüfbedingung)
THEN Abbruchbedingung := EQUAL (p,Minimum(Aufgabe.semantik.foutput.range)

R12':
IF (Ziel = Finde Ergebnis)
THEN direkter_Fall := x IN Aufgabe.semantik.name(bottom)=x.

Die Aufrufstruktur dieser Regeln wird in R9' festgelegt:

R9'
IF (Ziel = verfeinere direkten Fall)
THEN WITH Aufgabe.Rekursion.rumpf DO
    Finde_Prüfbedingung; (* R10/11'*)
    Finde_Ergebnis; (* R12' *)

Damit sind sehr spezifische Regeln für die Kodierung von Abbruchbedingung und direktem Fall erworben: Anstelle von zwei Regeln zur Ermittlung des kleinsten Teilproblems (R10) und zur Formulierung der entsprechenden Abbruchbedingungen (R11) existiert nur eine Regel (R10/11').Die Regel ist noch nicht parametrisiert und es wird noch nicht auf den rekursiven Parameter Bezug genommen, um das kleinste Teilproblem zu ermitteln. Zudem behandelt Regel 10/11' nur Funktionen auf natürlichen Zahlen. Regel 12' verlangt, daß der Wert für den direkten Fall direkt aus der Aufgabenspezifikation abzulesen ist.

Das Rekursionsschema für die Aufgabe ist nun partiell instantiiert: Im Funktionsrumpf muß nun noch der rekursive Fall ermittelt werden. Wieder wird zunächst versucht, die Syntax der Fakultätsfunktion und ihre Spezifikation aufeinander zu beziehen:

Rekursion.faku		Spezifikation.faku
MULT(n,faku(MINUS(n,1)))	:	MULT(all(n), range[n..0])

In der Aufgabenspezifikation wird entsprechend *PLUS(all(n), range[n..0])* identifiziert und es erfolgt folgendes Mapping:

MULT(all(n),range[n..0]) : MULT(n,faku(MINUS(n,1))) :: PLUS(all(n),range[n..0] : ?

MULT	-> PLUS
n	-> n
faku	-> summe
MINUS(n,1)	-> MINUS(n,1)

Nach diesem Mapping entstehen folgende weitere vorläufige Regeln:

R6'
IF (Ziel = finde Minimierung)
THEN Minimierung := *MINUS(p,1)*

R7'
IF (Ziel = Finde "Verknüpfer")
THEN WITH Aufgabe.Semantik.foutput DO
  verknüpfer := Operation in Operation(all(x))

R8'
IF (Ziel = Finde Operation)
THEN "hilfsop" := id(p)

R4'
IF (Ziel = verfeinere rekursiven Fall)
THEN WITH Aufgabe.Rekursion.rekursiver *Fall DO*
  finde_Minimierung; (* R6' *)
  finde_verknüpfer; (* R7' *)
  finde_operation; (* R8' *)

Hier ist zu beachten, daß die in den Regeln verwendeten Konzepte und Ziele bereits mit den beim Expertenmodell belegten Namen verwendet werden. Dies macht keine Aussage darüber, daß *die Anfänger nun tatsächlich zum Beispiel* einen Verknüpfer oder eine Operation finden wollen, relevant für die Annahmen des Modells sind lediglich die Regelaspekte, die zur Suche in der *Spezifikation und zur Instantiierung von* Schemaleerstellen führen. Die Regeln zur Kodierung des rekursiven Falls bleiben bisher sehr eingschränkt: Eine Regel zur Identifikation des rekursiven *Parameters entfällt*, da Aufgabe und Beispielfunktion nur je einen Parameter *p* enthalten. Als Minimierungsregel wird direkt *MINUS(p,1)* gespeichert. Der Verknüpfer wird gefunden, *in dem in der Spezifikation* geprüft wird, welche Operation die Anweisung *all* umschließt. Zusätzlich wird angenommen, daß das erste Argument der Verknüpfungsfunktion immer *der Eingangsparameter* der Funktion ist.

Nach Lösung der Aufgabe *summe* mit Hilfe des Beispiels *faku* verfügen die Lernenden jetzt also zusätzlich zum *allgemeinen* Rekursionsschema über ein spezifischeres Schema, nämlich das der Teil-Rest-Rekursion mit Verwendung des Eingangsparameters als "Teil" auf natürlichen Zahlen. An *dieses neuerworbene* Schema sind auch die während der Aufgabenlö-

sung inferierten Regeln gebunden:

Schema: Rekursion.allgemein
Kopf: FUN *name* (*parlist* = $\{p_i:t_i\}_{i=1..n}$):t_{res}
Rumpf:
Abbruchbedingung: IF <*boolsche Anweisung auf einem* $p_i \in parlist$>
direkter Fall: THEN *wert*
rekursiver Fall: ELSE <*Anweisung mit name (Minimierung eines* $_{pi} \in parlist$)> FIN
schreibe Rekursion: R1
verfeinere Kopf: R2
verfeinere Rumpf: R3
verfeinere rekursiven Fall: R4'
verfeinere Abbruchbedingung und direkten Fall: R9'

↓

Schema: Rekursion.Teil-Rest.wert.nat
Kopf: FUN *name* (parlist = {p:NAT}):NAT
Rumpf:
Abbruchbedingung: IF EQUAL(p,*Minimum*)
direkter Fall: THEN *wert*
rekursiver Fall: ELSE *verknüpfer*(p, *name*(p,MINUS(p,1))) FIN
schreibe Rekursion: R1
verfeinere Kopf: R2
verfeinere Rumpf: R3
verfeinere rekursiven Fall: R4',R6',R7',R8'
verfeinere Abbruchbedingung und direkten Fall: R9',R10',R11',R12'

Die Aufgabenlösung und die Beispiellösung sind als vollständig instantiierte Schemata als Kinder des inferierten Schemas ebenfalls im Langzeitgedächtnis gespeichert.

Wie aus der Darstellung offensichtlich wurde, ist die Lösung von Aufgaben bei geringem Vorwissen ein sehr langwieriger Prozess. Dieser Prozeß ist auch fehleranfällig, da nie auszuschließen ist, daß vorliegende Gemeinsamkeiten von Aufgabe und Beispiel nicht erkannt werden oder aber Unterschiede zwischen Aufgabe und Beispiel fälschlicherweise als strukturelle Gemeinsamkeiten identifiziert werden. Dies kann vorallem durch in der Oberflächenstruktur fehlende Gemeinsamkeiten beziehungsweise Irreleitung durch ebensolche Gemeinsamkeiten bedingt sein (Novick 1988). Dies ist der Fall, solange die Schemahierarchie noch sehr unvollständig und instabil aufgebaut ist und solange noch wenige Regeln inferiert sind. Eine unvollständige Repräsentation von Funktionsstrukturen führt zur Fehlklassifikation von Aufgaben, eine unvollständige Regelmenge führt zu einer fehlerhaften Instantiierung des Schemas.

Die bisher erworbene Schemastruktur ist sehr speziell, da sie aus zwei, sehr ähnlich strukturierten Aufgaben abstrahiert wurde. Entsprechend sind für die Lösung einer anders gearteten Aufgabe wieder aufwendige Inferenzprozesse notwendig. Es ist aber ebenso möglich, daß das neu inferierte Schema zur Lösung der neuen Aufgabe verwendet wird und sich so fehlerhafte Lösungen ergeben. In Fällen, in denen es einer Person aufgrund

der vorliegenden Wissensstruktur nicht möglich ist, ein erfolgreiches *Mapping* durchzuführen, kann eine Lösungsstrategie sein, die nicht klassifizierbaren Teile der Beispielfunktion einfach in die Aufgabenlösung zu übernehmen.

Je mehr rekursive Funktionen erfolgreich mithilfe von Beispielfunktionen gelöst wurden, desto einfacher und schneller sollten neue Programmieraufgaben bewältigt werden. Dies ist aber offensichtlich nur dann möglich, wenn Beispiele und Aufgaben so gewählt wurden, daß sinnvolle Verallgemeinerungen möglich sind.

6.4 Zusammenfassung des Modells und Ableitung empirisch prüfbarer Hypothesen

Im folgenden werden die in Kapitel 6.2 und 6.3 formulierten Annahmen zunächst abstrakt dargestellt. Im Anschluß daran werden dann aus dem formulierten Modell des Wissenserwerbs empirisch prüfbare Annahmen abgeleitet.

Ganz allgemein ist der Prozeß der Lösung einer Aufgabe folgendermaßen darstellbar (Abb. 10):

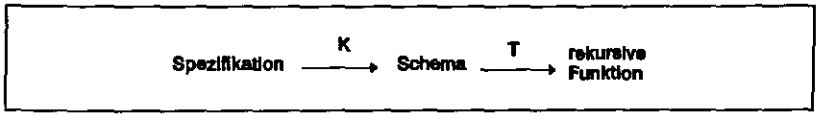
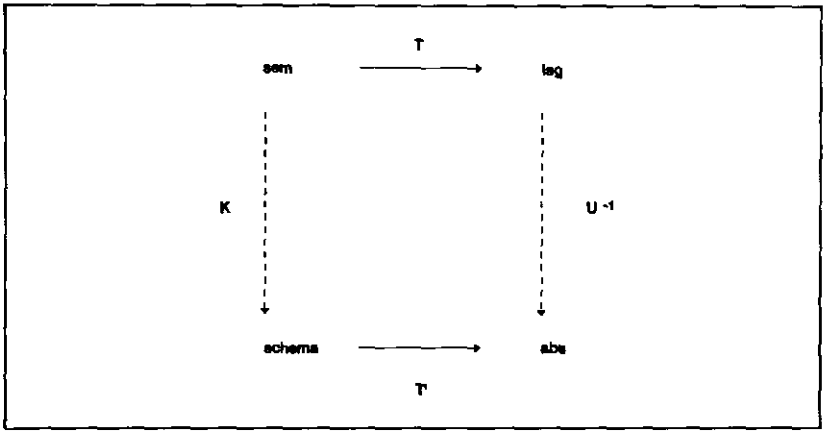


Abb. 10: Schematische Darstellung des Prozesses der Aufgabenlösung

Ausgehend von einer Aufgabenstellung wird mithilfe einer Klassifikationsfunktion (K) ein entsprechends Schema ausgewählt. Dieses Schema wird mit den daran gebundenen Transformationsregeln (*T*) schrittweise solange verfeinert, bis eine Lösung der Aufgabe vorliegt. Würde in dem Modell die Aufgabenlösung programmiersprachenunabhängig kodiert, so würde eine Menge von Übersetzungsfunktionen (*U*) benötigt, die die Implementierung der rekursiven Funktion in einer konkreten Programmiersprache beschreiben. Damit ergibt sich in Anlehnung an die Vorstellung von induktiven Prozessen als Homomorphismen (Holland, Holyoak, Nisbett & Thagard 1986; vgl. Kap. 3.5, Kap. 4.5; siehe Anhang A.2) also folgendes Bild (Abb. 11):



mit
sem = Spezifikation
lsg = Funktionscode
schema = aktiviertes Schema
abs = abstrakte Lösung

K : Klassifikator mit vier Merkmalsdetektoren
sem -> schema mit K = (k1, k2, k3, k4)
k1: count -> Endrekursion
sonst -> default Teil-Rest-Rekursion
k2: Endrek. + cond -> 2 Abbruchbed.
Endrek. sonst -> default 1 Abbruchbed.
k3: Teil-Rest + Operation -> Teil = Operation
Teil-Rest + sonst -> Teil = Wert
k4: NAT -> NAT
(NAT + LIST) v LIST -> LIST

T : Transformationsfunktion
sem -> lsg

T' : mentale Transformationsfunktion
schema -> abs

U : "Übersetzungsfunktion"
ordnet Strukturen der abstrakten Lösung und der Syntax einer Programmiersprache einander zu
abs -> lsg

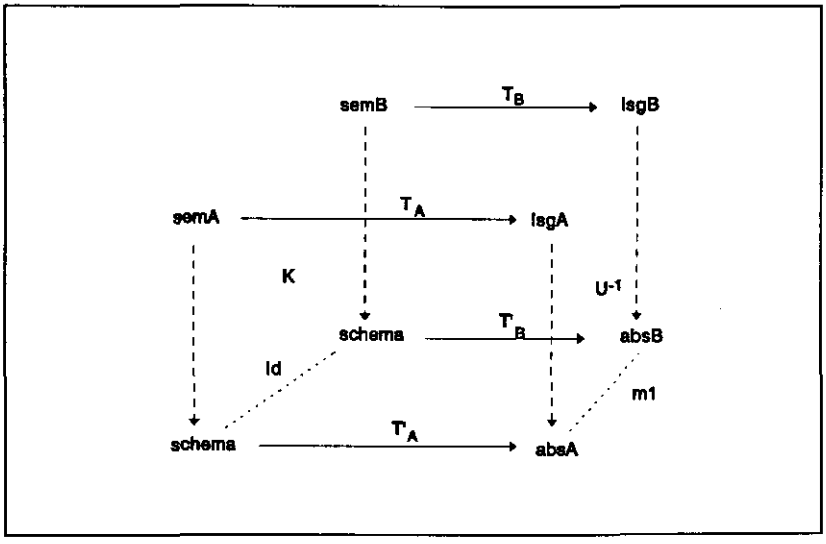
Abb. 11: Darstellung der Lösung einer rekursiven Aufgabe als kommutatives Diagramm

In der oberen Zeile des Diagramms (Welt) sei die korrekte Transformation von einer Aufgabenstellung in eine rekursive Funktion abgebildet. In der unteren Zeile entsprechend die notwendigen internen Repräsentationen zusammen mit den Regeln zur Aufgabenlösung.

In einem ersten Schritt muß die vorliegende Programmieraufgabe mit Hilfe der durch die Schemahierarchie gegebenen Merkmale klassifiziert werden. Dies geschieht mithilfe einer Klassifikationsfunktion *K: sem -> schema*. Die Klassifikationsfunktion ist eine Familie von Merkmalsdetektoren. Jeder Detektor prüft die Ausprägung eines durch die Schemahierarchie gegebenen Merkmals. Dadurch wird das Schema, das der Aufgabenspezifikation am stärksten entspricht, aktiviert. Mit Hilfe der an das Schema gebundenen Regeln (T') wird das Schema zu einer abstrakten Definition einer rekursiven Funktion verfeinert. Diese abstrakte Lösung wird mit Übersetzungsregeln U in eine Programmiersprache transformiert. Ist in der oberen Zeile des Diagramms die korrekte Transformation angegeben, so kommutiert das Diagramm, wenn K und U korrekt sind. Das heißt, daß die Klassifikation der Aufgabe ein Schema aktiviert, an das Regeln gebunden sind, mit denen die korrekte Funktion zur Spezifikation gebunden werden kann und daß die Überführung in die Syntax einer Programmiersprache ebenfalls beherrscht wird: $T'(K(sem)) = U\text{-}1(T(sem))$. Da die Übersetzungsfunktion U "in beide Richtungen" funktionieren muß, also ein Isomorphismus ist, ergibt sich auch: $U(T'(K(sem))) = T(sem)$.

Bisher wurde dargestellt, wie eine Aufgabe mithilfe gegeigneter Wissensstrukturen, die aus dem Langzeitgedächtnis abgerufen werden, gelöst werden kann. Sind nur rudimentäre Wissensstrukturen im Langzeitgedächtnis vorhanden, ist also der Lernprozeß noch nicht abgeschlossen, kann als zusätzliche Information eine Beispielfunktion herangezogen werden. Während also ein Experte zur erfolgreichen Aufgabenlösung lediglich die Spezifikation der Aufgabe und das entsprechende Schema zusammen mit den daran gebundenen Regeln benötigt, müssen Lernende zusätzlich die Spezifikation und Lösung eines Beispiels zur Verfügung haben. Wie oben bereits angedeutet, wird zur Vereinfachung des Modells auf die explizite Repräsentation von Syntaxwissen und grundlegendem mathematischen Wissen verzichtet.

Entsprechend erweitert sich das Modell für den Lernprozeß (Abb. 12):



mit

sem	= Spezifikation
lsg	= Funktionscode
schema	= aktiviertes Schema
abs	= abstrakte Lösung

K	: sem -> schema mit K = (k1, k2, k3, k4)
T_A, T_B	: sem -> lsg
T'_A, T'_B	: schema -> abs
U	: abs -> lsg

K' : lsgB -> schema ; zur Ergänzung der unvollständigen Klassifikationsfuktion K

Inferenzfunktionen:

Inferenz des Lösungswegs:

i1(semB,absB) -> z(semB,absB)	; Zuordnung von Konstrukten der Spezifikation und der Lösung des Beispiels
i2(z(semB,absB)) -> T'B	; Inferenz des Lösungsweges des Beispiels
i3(T'B,semA) -> T'A	; Inferenz des Lösungsweges der Aufgabe

Inferenz eines generalisierten Schemas:
m1: (absA,absB) -> schema'.fkt
m2: (T'A,T'B) -> schema'.rgl
L (m1(absA,absB),m2(T'A,T'B) -> schema'

Abb. 12: Darstellung des analogen Transfers

Das zweidimensionale Diagramm zur Modellierung des Expertenverhaltens bei der Lösung einer Programmieraufgabe wird hier zu einem dreidimensionalen Diagramm erweitert. Die obere Ebene stellt wieder die korrekte Transformation einer Aufgabenstellung in eine rekursive Lösung dar, in diesem Fall für das Beispiel und die Aufgabe. Dabei liegen für das Beispiel Spezifikation sowie Lösung vor, für die Aufgabe nur die Spezifikation. Es exisitiert eine vorläufige Klassifikationsfunktion *K*, die entweder unvollständig oder sogar fehlerhaft ist. In einem frühen Stadium des Lernprozesses kann anstelle von *K* auch die Beispiellösung zur Auswahl eines Schemas verwendet werden: K': lsg_B -> *schema*. Das Rekursionsschema, mit dem sich die Beispielfunktion repräsentieren läßt, wird aktiviert. Im beschriebenen Fall war nur ein Schema vorhanden, das aktiviert werden konnte. Es wird versucht, die Aufgabe ebenfalls mithilfe dieses Schemas zu lösen. Das heißt, in dem Modell wird angenommen, daß Anfänger die Beispielstruktur zunächst unhinterfragt übernehmen.

Ist ein Schema aktiviert, wird versucht, die in diesem Schema noch nicht vorhandenen, aber notwendigen Regeln zur Schemainstantiierung mit Hilfe einer Zuordnung von Beispielspezifikation und Beispiellösung zu inferieren. Dies geschieht, in dem der Funktionsrumpf der Beispielfunktion komponentenweise auf Entsprechungen in der Beispielspezifikation überprüft wird (*i2(i1(semB,absB))*). Auf diese Weise werden die Regeln T_B' generiert und durch einen Vergleichsprozeß von Spezifikation und Lösung des Beispiels mit der Spezifikation der Aufgabe in Regeln T_A' überführt (*i3(T'B, semA)*). Die Anwendung dieser Regeln führt zu einer Lösung der gestellten Aufgabe. Die Zuordnung von T_A' und T_B' (*m1*) sowie der Vergleich der instantiierten Funktionsleerstellen von Aufgabe und Beispiel (*m2*) führt zu einer Inferenz der notwendigen Struktur zur Lösung der Aufgabe. Die identifizierten Regeln zur Kodierung der Beispielfunktion werden durch Übertragung auf die Aufgabe generalisiert. So entsteht ein neues Schema, das die Charakteristika von Beispiel und Aufgabe repräsentiert: Dieses Schema enthält sowohl spezifische Werte für die Funktionsstruktur als auch die entsprechende Regelstruktur zur Instantiierung der Funktionsstruktur.

Die Aufgabenlösung mit Beispielen ist also ein Prozeß $T'_{AB}(T'_A(K(sem_A)), T'_B(K(sem_B))) = U^1_{AB}(U^1(T_A(sem_A)), U^1(T_B(sem_B)))$. T'_{AB} und U^1_{AB} sind dabei Abbildungen von Tupeln, deren erste Komponente sich auf die Aufgabe und deren zweite Komponente sich auf das Beispiel bezieht.

Wissenserwerb kann als Funktion $L(m1(abs_A, abs_B), m2(T'_A, T'_B))$ beschrieben werden, die als Ergebnis ein neues Schema liefert. Das inferierte Schema wird an das zur Lösung aktivierte Schema als Oberschema gebunden und hat als Unterschemata die gelöste Funktion und die Beispielfunktion.

Ob die Aufgabenlösung gelingt, hängt davon ab, ob die obige Gleichung gilt. Die obige Gleichung kann unerfüllt sein, wenn entweder die Klassifikationsfunktion und/oder die inferierten Transformationsregeln fehlerhaft sind.

Aus dem skizzierten Modell lassen sich nun verschiedene Annahmen ableiten, die empirisch prüfbar sind: Eine erste, sehr allgemeine Annahme ist, daß Anfänger, beim Lösen von Aufgaben mit Beispielen sehr viel Zeit benötigen. Der hohe Zeitaufwand kommt zustande, da aufwendige Vergleichs- und Anpassungsprozesse stattfinden müssen, um die Aufgabe durch analogen Transfer zu lösen. In direktem Zusammenhang damit ist ebenfalls anzunehmen, daß zu Beginn des Lernprozesses viele Fehler bei der Aufgabenlösung gemacht werden, die durch ein Fehlen geeigneter Klassifikations- und Transformationsregeln verursacht werden. Die wesentlichen Fehler sind dabei entweder, daß irrtümlich strukturelle Übereinstimmungen zwischen Aufgabe und Beispiel angenommen werden, oder daß tatsächlich vorliegende Übereinstimmungen nicht erkannt werden. Neben diesen Annahmen zur Aufgabenlösung kann ebenfalls aus dem Modell abgeleitet werden, daß bei jedem Versuch, eine Aufgabe mit Hilfe eines strukturähnlichen Beispiels zu lösen, ein Schema inferiert wird, das die Gemeinsamkeiten von Aufgabe und Beispiel abstrakt repräsentiert. Diese Annahmen wurden in Untersuchung I (Kapitel 8.2) überprüft, indem die Vorgabe von Beispielen mit der direkten Vorgabe der zur Aufgabenlösung notwendigen Regeln kontrastiert wird.

Zusätzlich zu diesen allgemeinen, aus dem Modell ableitbaren Annahmen, können spezifischere Aussagen über den Einfluß von Beispielen auf den Aufbau neuer Schemata gemacht werden. Aus Einzellfallstudien von Pirolli und Anderson (1985; vgl. Kap. 7) gibt es bereits Hinweise darauf, daß die Ähnlichkeit eines Beispiels zur Aufgabe maßgeblichen Einfluß auf das induzierte Wissen hat (vgl. auch Anderson & Thompson 1989). Beispiele, die sehr hohe strukturelle Ähnlichkeit zur Aufgabe aufweisen (wie *summe* zu *faku*, vgl. Kap. 6.3), ermöglichen zwar, daß die Aufgabenlösung schneller und einfacher gelingt, führen aber zu sehr spezifischen Schemainferenzen. Damit ist ein Lerntransfer nur für einen sehr engen Bereich möglich. Sind Aufgabe und Beispiel strukturell sehr verschieden, gelingt möglicherweise keine Verallgemeinerung, so daß kein neues Schema inferiert werden kann. Aufgabenlösungen ergeben sich in diesem Fall durch Verwendung des allgemeinen Schemas zusammen mit einer Versuchs- und Irrtumsstrategie. Erfolgreiches Lernen sollte also mit Beispielen, die nicht zu strukturähnlich sind aber dennoch eine gemeinsame Struktur mit der Aufgabe aufweisen, möglich sein. Es ist bei dem derzeitigen Stand der Forschung zum Lernen mit Beispielen sicher noch nicht möglich, anzugeben, welcher Grad an struktureller Ähnlichkeit zwischen Aufgabe und Beispiel die beste Voraussetzung für einen effizienten Wissenserwerb liefert. Zudem kann man vermuten, daß solche Angaben auch stark von dem Problembereich, der erlernt werden soll, abhängig sind. Deshalb werden für eine empirische Untersuchung dieser Annahmen hier spezifische Beispiele konstruiert, die systematisch in ihrer strukturellen Ähnlichkeit zur Aufgabe variieren. Für diese Beispiele können dann mit Hilfe der im Modell postulierten Prozesse konkrete Annahmen über die inferierten Schemata formuliert werden. Dieses Vorgehen wird in Untersuchung II (Kap. 8.3) realisiert.

7 Empirische Zugänge zum Erwerb von Problemlösefertigkeiten

Kognitionspsychologische Modelle sollten zwei Klassen von Gütekriterien genügen: Erstens sollte ein kognitionspsychologisches Modell so präzise formuliert sein, daß es im Prinzip implementierbar ist, um zu gewährleisten, daß die postulierten Prozesse kognitiver Verarbeitung konsistent und vollständig definiert wurden. Zweitens sollte ein kognitionspsychologisches Modell seinen Gegenstand adäquat beschreiben. Das heißt, die im Modell formulierten Annahmen über kognitive Prozesse sollten nicht nur intuitiv psychologisch plausibel sein, sondern einer empirischen Prüfung standhalten. Ob empirische Befunde allerdings hinreichen, um Modellannahmen als psychologisch adäquat zu beurteilen, hängt in hohem Maße von der Art und Weise ab, in der empirische Untersuchungen konzipiert sind. Dabei ist einmal darauf zu achten, daß die relevanten Aspekte des Untersuchungsgegenstands tatsächlich in der Untersuchung erfaßt werden (externe Validität). Gleichzeitig sollten in einer hypothesenprüfenden Untersuchung die erhobenen Kennwerte möglichst eindeutig auf die in den Hypothesen genannten Ursachen rückführbar sein (interne Validität).

Im folgenden werden verschiedene empirische Zugänge zum induktiven Erwerb von Problemlösefertigkeiten dargestellt und bezüglich ihrer Tauglichkeit zur empirischen Prüfung dieses Gegenstands bewertet. Am Ende stelle ich ein Untersuchungskonzept dar, das meiner Meinung nach besser als die dargestellten empirischen Zugänge zur Erfassung von induktiven Lernprozessen geeignet ist.

7.1 Erfassung von Schemainduktion

Modellvorstellungen zum analogen Lernen von Problemlösetechniken (vgl. Kap. 4.2, Kap. 6) beschreiben Wissenserwerb als Induktion abstrakter Schemata. Ein induziertes Schema repräsentiert dabei die Problemklasse, die durch eine gelöste Aufgabe (Zielproblem) und das dafür verwendete Beispiel definiert ist. Die meisten empirischen Untersuchungen der Schemainduktion beim analogen Lernen haben folgende Merkmale gemeinsam: Erstens werden meistens einfache mathematische Problemlösetechniken (z.B. "Finde kleinsten gemeinsamen Teiler") als Lerngegenstand verwendet. Zweitens wird experimentell gearbeitet, wobei meistens der Einfluß verschiedener Bedingungen der Beispielpräsentation auf den Lernerfolg geprüft wird. Im folgenden werden drei zentrale Untersuchungen zum analogen Lernen exemplarisch für diesen Forschungsbereich dargestellt.

Ross und Kennedy (1990) arbeiteten mit Aufgaben aus dem Bereich der Wahrscheinlichkeitsrechnung. Sie untersuchten, welche Arten von Wissen beim analogen Lösen von Aufgaben generalisiert werden. Probanden erhielten zunächst Instruktionen über vier grundlegende Prinzipien der Wahrscheinlichkeitsrechnung (Permutation; Kombination; Eintretenswahrscheinlichkeit eines Ereignisses beim k-ten unabhängigen Versuch; Wahrscheinlichkeit,

bei *k* Ziehungen mit Zurücklegen ein bestimmtes Item mindestens einmal zu erhalten) indem ihnen je eine abstrakte Beschreibung des Prinzips und eine Beispielaufgabe zusammen mit dem Lösungsweg präsentiert wurde. Danach erhielten sie Zielaufgaben, wobei einem Teil der Probanden jeweils ein Hinweis auf die Beispielaufgabe gegeben wurde, die den für die Aufgabenlösung korrekten Ansatz realisiert. Nachdem Zielaufgaben für jedes Prinzip mit und ohne Hinweise gelöst wurden, wurde eine zweite Menge von Zielaufgaben vorgegeben, die jetzt in jedem Fall ohne Hinweise zu bearbeiten waren. Es zeigte sich, daß diejenigen Aufgaben der zweiten Menge erfolgreicher gelöst werden konnten, für deren Berechnungstyp im ersten Durchgang Hinweise auf die zu verwendenden Beispiele gegeben wurden. Die explizite Vorgabe eines zu einer Aufgabe strukturäquivalenten Beispiels erleichtert also die Generalisierung eines Problemtyps, so daß neue Probleme dieses Bereichs leichter korrekt klassifiziert und die entsprechenden Problemlöseprinzipien angewendet werden können.

In Studie von Ross und Kennedy (1990) wurde die beim Lernen mit Beispielen postulierte Schemainferenz also mittels des Lösungserfolgs bei einer Transferaufgabe empirisch erfaßt. Diese Art der Datenerhebung erbringt zwar den Nachweis, daß die Vorgabe eines Beispiels den Wissenserwerb positiv beeinflußt, bleibt aber sehr unspezifisch, da die Schemainferenz nur indirekt, über die Erhöhung der Problemlösungsperformanz, abgebildet wird.

Novick & Holyoak (1991) führten ein Experiment zum Transfer von Lösungstechniken für mathematische Textaufgaben durch. Die Textaufgaben verlangten sämtlich die Angabe der kleinsten Zahl, für die mehrere genannte Teiler denselben konstanten Restbetrag ergeben und ein weiterer genannter Teiler einen Rest von *null* ergibt. Die Lösung aller Aufgaben basiert auf einem Vielfachen des kleinsten gemeinsamen Teilers für die genannten Divisoren (plus Rest). Probanden erhielten mehrere Aufgaben dieser Art zusammen mit einer detaillierten Beschreibung des Lösungsprozesses. Eine dieser Aufgaben war strukturell analog zu einer im Anschluß präsentierten Zielaufgabe. Die Probanden sollten diese Zielaufgabe lösen, wobei folgende Bedingungsvariation durchgeführt wurde: Eine Gruppe von Probanden erhielt einen Hinweis, welche der gelösten Aufgaben als Beispiel für die Zielaufgabe verwendet werden kann (*retrieval* Gruppe). Eine weitere Gruppe erhielt zusätzlich den Hinweis, welche Zahlen aus Beispiel und Aufgabe korrespondieren (*number-mapping* Gruppe). Eine andere Gruppe erhielt den Hinweis, welche Konzepte sich in Aufgabe und Beispiel entsprechen (*concept-mapping* Gruppe). Eine Kontrollgruppe erhielt keinerlei Hinweise. Nachdem die Probanden die Zielaufgabe gelöst oder maximal 15 Minuten an der Lösung gearbeitet hatten, wurde die Schemainduktion erfaßt. Die Probanden sollten angeben, welche Ähnlichkeiten sie in der Problemstruktur und im Lösungsweg zwischen Beispiel- und Zielaufgabe feststellen. Am Ende der Untersuchung sollte eine weitere Textaufgabe gelöst werden, die zusätzlich einen Teiler mit einem weiteren Restbetrag enthielt (Generalisierungs-Problem).

In diesem Experiment wurde erstens versucht, den Einfluß der Teilprozesse Mapping und

Adaptation durch die verschiedenen Hinweisbedingungen zu isolieren. Dabei wurde angenommen, daß die Zuordnung der entsprechenden Zahlen aus Beispiel und Zielaufgabe zentral für die Adaptation ist. Das Ergebnis bestätigt diese Hypothese: Der Lösungserfolg des Zielproblems ist am größten in der *number-mapping* Gruppe und am zweithöchsten in der *concept-mapping* Gruppe, dicht gefolgt von der *retrieval* Gruppe. Die Kontrollgruppe zeigt mit Abstand den geringsten Lösungserfolg bei der Zielaufgabe. Zweitens wurde der Zusammenhang zwischen dem Lösungserfolg bei der Transferaufgabe und der Schemainduktion erfaßt. Dabei wurden die beantworteten Fragen zur Ähnlichkeit zwischen Aufgabe und Beispiel von Ratern den Kategorien gute, mittlere und schwache Schemainferenz zugeordnet. Es zeigte sich, daß eine gute Schemainferenz einen starken Zusammenhang mit der erfolgreichen Lösung der Zielaufgabe aufweist. Schließlich wurde der Einfluß des inferierten Schemas auf die Lösung der Generalisierungsaufgabe erfaßt. Dabei zeigte sich, daß die Qualität des inferierten Schemas in höherem Zusammenhang mit dem Lösungserfolg der Generalisierungsaufgabe steht, als die erfolgreiche Lösung des Zielproblems.

Novick und Holyoak (1991) prüfen den Prozeß des analogen Lernens differenzierter als Ross und Kennedy (1990). Sie führen erstens eine Trennung der beim analogen Problemlösen postulierten Teilprozesse (*retrieval*, *mapping*, *adaptation*) durch und erfassen zweitens die Schemainferenz auf zwei unterschiedliche Arten, indem sie zum einen direkt nach Ähnlichkeiten zwischen Aufgabe und Beispiel fragen und zum anderen den Lösungserfolg bei einer Generalisierungsaufgabe erheben. Die Generalisierungsaufgabe ist, anders als die Transferaufgabe im Experiment von Ross und Kennedy (1990), nicht nur eine weitere Aufgabe des selben Problemtyps sondern stellt eine Verallgemeinerung der bisher bearbeiteten Probleme dar. Die Erhebung des beim analogen Problemlösen inferierten Schemas durch explizite Nachfrage ist, trotz der methodischen Probleme, die bei der Klassifikation offener Antworten auftreten (vgl. Tergan 1986), ein Fortschritt gegenüber der alleinigen Erhebung des Lösungserfolgs bei Transferaufgaben.

Sowohl in der Studie von Ross und Kennedy (1990) als auch in der Studie von Novick und Holyoak (1991) wird mit Beispielen gearbeitet, die so konstruiert wurden, daß sie eine möglichst hohe strukturelle Ähnlichkeit zu den Zielaufgaben aufweisen. Unter Umständen gelingt es nicht immer einfach, Beispiele auf diese Art zu konstruieren. In jedem Fall sind immer Annahmen darüber zu treffen, welche Abweichungen zwischen Beispiel und Aufgabe die Struktur nicht betreffen. So könnte ein strukturähnliches Beispiel für eine Aufgabe nur eine Konstante a durch eine Konstante b ersetzen, ein anderes Beispiel aber eine Konstante a durch einen Ausdruck $b+x$. Zudem erheben die beiden bisher dargestellen Untersuchungen nicht, welchen Einfluß verschiedene Grade struktureller Ähnlichkeit auf den Schemaerwerb haben. Zu diesem Zweck müßten die Beispiele sich systematisch in ihrer strukturellen Ähnlichkeit zu den Aufgaben unterscheiden. Eine Studie, in der Beispiele auf diese Art

konstruiert werden, wurde von Reed und Bolstad (1991) durchgeführt.

Reed & Bolstad (1991) arbeiten in ihrer Untersuchung mit sogenannten *work problems*. Als einfachster Fall muß für ein Paar von Arbeitern mit unterschiedlichen Arbeitsgeschwindigkeiten (*rate*) berechnet werden, wieviel Zeit (*time*) sie gemeinsam zur Erledigung einer Aufgabe (*task*) benötigen (Typ 0). Komplexere Probleme geben die Arbeitsgeschwindigkeit eines Arbeiters relativ zum anderen an, oder ein Arbeiter arbeitet länger an der Aufgabe als der andere, oder ein Teil der Aufgabe ist bereits erledigt. Dabei können in einer Aufgabe entweder eine dieser Komponenten (Typ 1) oder zwei Komponenten (Typ 2) oder alle drei Komponenten (Typ 3) verändert werden. Die Autoren untersuchten den Einfluß der Vorgabe von Beispielen und Regeln auf den Lösungserfolg von mehreren unterschiedlich komplexen Zielaufgaben (Typ 0 bis Typ 3). Dabei wurden folgende Instruktionsbedingungen verwendet: Probanden erhielten entweder ein einfaches Beispiel (Typ 0) und dessen Lösung oder ein komplexes Beispiel (Typ 3) und dessen Lösung oder beide Beispiele oder eine Beschreibung des Lösungsprozesses oder schließlich eines der Beispiele zusammen mit den Lösungsregeln. Um eine Aufgabe mit einem Beispiel zu lösen, müssen dabei nur Zahlen ausgetauscht werden, wenn Aufgabe und Beispiel vom selben Typ sind, oder aber eine bis drei Transformationen gemacht werden, je nach dem wie weit der Typ der Aufgabe vom Beispieltyp entfernt ist. Während für die Probanden, die nur mit den Regeln arbeiteten, der Lösungserfolg der verschieden komplexen Aufgaben generell sehr niedrig lag, zeigte sich für die anderen Lernbedingungen ein annähernd linearer Zusammenhang zwischen der Anzahl notwendiger Transformationen und dem Lösungserfolg. Dabei brachte die Kombination eines Beispieltyps mit der Regelvorgabe für beide Beispieltypen kaum einen Vorteil. Die Vorgabe des komplexen Beispiels hatte einen geringen Vorteil gegenüber der Vorgabe des einfachen Beispiels und der Lösungserfolg bei Vorgabe beider Beispieltypen war deutlich höher als bei allen anderen Lernbedingungen. Reed und Bolstad (1991) konnten einen Einfluß der Beispielähnlichkeit auf den Lösungserfolg der Zielaufgaben nachweisen. Es wäre jedoch sinnvoll, zusätzlich zu prüfen, wie sich unterschiedliche Beispielähnlichkeiten auf den Schemaerwerb auswirken.

Die drei dargestellten Arbeiten sind typische Vertreter für die im Bereich analoges Lernen durchgeführten Untersuchungen. Der experimentelle Zugang - durch kontrolliere Variation unabhängiger Variablen und zufällige Zuweisung von Probanden zu den Bedingungen - liefert Ergebnisse, die es erlauben, die Befunde von der untersuchten Stichprobe zu generalisieren. Nachteilig an den beschriebenen Experimenten ist jedoch, daß sie auf einen Meßzeitpunkt beschränkt bleiben. Der Erwerb von Problemlösefertigkeiten in einem komplexen kognitiven Bereich ist ein längerfristiger Prozeß, dem diese Art von Untersuchungsstrategie nicht gerecht wird. In natürlichen Lernsituationen lösen Anfänger sicher eine Menge von Aufgaben mit Hilfe strukturähnlicher Beispiele und inferieren dabei allmählich immer

adäquatere Schemata über den Problembereich, so daß sich Effizienz und Korrektheit der Aufgabenlösung immer mehr erhöhen und allmählich kein Bedarf mehr für die Nutzung von Beispielen besteht. Um also tatsächlich den **Prozeß** des analogen Lernens zu erfassen, sollten die induzierten Schemata sowie die Lösungsperformanz bei Transferaufgaben zu mehreren Zeitpunkten erhoben werden.

7.2 Experten-Novizen-Vergleiche

Eine indirekte Methode, den Prozeß des Wissenserwerbs zu erfassen, sind Experten-Novizen-Vergleiche. Dieses Untersuchungsparadigma wurde von Chase und Simon (1973) in Studien zur Wissensorganisation beim Schach bekannt gemacht und dann in einer Vielzahl von Studien in komplexen kognitiven Bereichen, wie Physik (Chi, Feltovich & Glaser 1981) und Programmierung (Shneiderman 1976) übernommen.

Dabei werden Personen ohne oder mit geringer Erfahrung in einem Bereich (Novizen) mit Personen, die langjährige Erfahrung in diesem Bereich haben (Experten) bezüglich verschiedener kognitiver Leistungen verglichen. Im Bereich Programmierwissen werden meistens Reproduktionsaufgaben, Fehlersuchaufgaben oder Fragen zum Programmverständnis, aber nur sehr selten Programmieraufgaben verwendet (vgl. Soloway & Spohrer 1989). Häufig werden zusätzlich zu den Extremgruppen Zwischengruppen mit mittlerer Lernerfahrung eingeführt (Soloway, Bonar & Ehrlich 1989).

Die Analysen der Unterschiede, die Gruppen mit verschiedener Erfahrung bezüglich der erhobenen Variablen aufweisen, gibt indirekten Aufschluß über den Lernprozeß. Lesestrategien bei der Memorierung von Programmcode, sowie Güte und Reihenfolge der Programmreproduktion lassen Schlußfolgerungen auf eine Entwicklung der Wissensorganisation vom Novizen zum Experten zu (Widowski & Eyferth 1986). Fehlersuchstrategien sowie tatsächlich identifizierte Fehler in Programmtexten geben zusätzlich Hinweise über die Repräsentation von Programmierkonzepten (Gugerty & Olson 1987; Putnam, Sleeman, Baxter & Kuspa 1986).

Kahney (1989) versuchte zum Beispiel zu identifizieren, welches Konzept Novizen über Rekursion haben. Zu diesem Zweck präsentierte er Experten und Novizen drei kurze Programme. Das geforderte Programmverhalten war anschaulich beschrieben. Im Prinzip sollte jedes Programm an jedes Element einer Liste einen Eintrag anhängen. Eines der Programme war nicht rekursiv und veränderte lediglich das erste Listenelement. Ein weiteres Programm war als Endrekursion realisiert, jeder Listeneintrag wurde direkt beim Einlesen verändert. Das dritte Programm war teil-rest-rekursiv programmiert, so daß zunächst alle Listenelemente auf einen Stack gepackt wurden und die Listenelemente beim rekursiven

Aufstieg in umgekehrter Reihenfolge verändert wurden. Kahney (1989) bat Novizen und Experten anzugeben, welche der drei Programme das gewünschte Verhalten zeigen und die getroffene Wahl zu begründen. Er konnte zeigen, daß Experten ein korrektes Modell rekursiver Programme haben, das er als *Copy*-Modell bezeichnet: Jeder rekursive Aufruf erzeugt eine Kopie des rekursiven Programms. Novizen verfügen dagegen über ein Schleifenmodell, bei dem das rekursive Programm Parameter erhält, sie verändert und dann das rekursive Programm mit neuen Parametern aufruft (*Loop*-Modell). Bei diesem Modell von Rekursion sollten die Novizen zwar das endrekursive Programm aber nicht das teil-rest-rekursive Programm als korrekt akzeptieren. Dies war bei einem Großteil der Novizen tatsächlich der Fall.

Experten-Novizen-Vergleiche geben zwar Aufschluß über unterschiedliche Wissensstrukturen zu verschiedenen Lernzeitpunkten und ermöglichen es, daraus Hinweise auf den Lernprozeß abzuleiten. Diese indirekt erschlossenen Annahmen über Wissenserwerbsprozesse müssen aber sehr global bleiben, da bei diesem Vorgehen keine Kenntnisse über Bedingungen des Lernens vorliegen, die zur Veränderung der beobachteten Novizen-Repräsentationen in Richtung der Experten-Repräsentationen führen. Dies liegt daran, daß bei Experten-Novizen-Vergleichen die Lernerfahrung nicht kontrolliert ist: Es bleibt offen, wie häufig und auf welche Art und Weise sich die Mitglieder der verschiedenen Erfahrungsgruppen mit dem untersuchten Problembereich auseinandersetzen und damit, welche Lernbedingungen auf den Lernprozeß einwirkten.

Im Bereich Programmierwissen sind solche eher explorativen Zugänge üblich. Häufig werden anstelle eines Experten-Novizen-Vergleiches auch Personen mit geringer aber vergleichbarer Programmiererfahrung aufgrund eines Vortests in verschiedene Leistungsgruppen eingeteilt und dann versucht, Programmlese- und Fehlersuchstrategien zu identifiziert, die diese Leistungsunterschiede erklären (z.B. Samurcay 1989). Oder Lernende werden nach Merkmalen, wie allgemeine Intelligenz oder motivationalen Orientierungen, in Gruppen geteilt und es wird geprüft, ob diese Merkmale einen Einfluß auf Erwerben von Programmiertechniken haben (z.B. Kurland, Pea, Clement & Mawby 1989).

Systematische Untersuchungen zum Einfluß von Lehrstrategien, so wie sie in den experimentellen Untersuchungen zum analogen Lernen (Kap. 7.1) durchgeführt werden, fehlen fast vollständig. Stattdessen werden häufig eher aus dem Bereich der Pädagogik stammende Curricula, die kaum theoretisch fundiert sind, in quasi-experimentellen Studien miteinander verglichen (Heller 1987). Eine Ausnahme bildet eine Studie von Kessler & Anderson (1986). Hier wurde der Einfluß der Reihenfolge der Vermittlung iterativer und rekursiver Konzepte auf das Verständnis dieser Kontrollstrukturen experimentell geprüft und es zeigte sich, daß der Transfer zwischen diesen Konzepten asymmetrisch ist. Während eine Kenntnis iterativer Kontrollstrukturen den Erwerb rekursiver Techniken positiv unterstützt, erleichtert eine

Beherrschung rekursiver Kontrollstrukturen den Erwerb von Schleifentechniken nicht.

7.3 Einzelfallstudien

Experten-Novizen-Vergleiche sind dann ein sinnvolles Forschungsinstrument, wenn kaum theoretische Modellvorstellungen über den Forschungsgegenstand vorhanden sind, wie dies in den 80er Jahren im Bereich Programmierwissen der Fall war. Ein explorativer Zugang ermöglicht, eine erste Annäherung an den Untersuchungsgegenstand und kann die Grundlage für die Generierung von Modellvorstellungen liefern, die dann in hypothesenprüfende Untersuchungen münden. Ein explorativer Zugang, der detaillierte Aufschlüsse über längerfristige Lernprozesse liefert, sind Einzelfallstudien. Die Forschergruppe um Anderson (Anderson, Farrell & Sauers 1984; Pirolli & Anderson 1985) sammelte Protokolle von Personen beim Lernen der Programmiersprache LISP. Diese Protokolle bildeten die Grundlage für die Entwicklung des Schüler-Moduls in einem intelligenten tutoriellen System zur Vermittlung von LISP (Anderson, Conrad & Corbett 1989). Zudem führten die Beobachtungen von LISP-Lernenden zur Entwicklung eines Produktionssystemmodells zum analogen Lernen (Anderson & Thompson 1989; vgl. Kap. 4.3).

Pirolli und Anderson (1985) untersuchten mehrere Personen, die ihre ersten Erfahrungen mit rekursiver Programmierung machten. Ein Teil der Personen arbeitete dabei mit LISP, andere Personen mit LOGO (Papert 1980) bzw. SIMPLE (Shrager & Pirolli 1983). Die Personen hatten dabei Lehrbücher, wie zum Beispiel *"Lets Talk LISP"* (Siklossy 1976) zur Verfügung. Das allgemeine Vorgehen bei Einzelfallstudien ist, daß eine Person in Gegenwart eines Versuchsleiters versucht, eine Aufgabe zu lösen und dabei angewiesen ist laut zu denken. Die laute Denken Protokolle zusammen mit den Lösungsversuchen liefern dann die Datenbasis zur Analyse des Lernprozesses. Pirolli und Anderson stellten zunächst fest, daß alle Probanden Beispiele aus den Lehrbüchern zurate zogen, um ihre ersten rekursiven Programme zu erstellen. Zudem wurde im Schnitt mehr als ein Drittel der Zeit auf das Betrachten der Beispiele verwendet. Detaillierte Protokollanalysen ergaben zum Beispiel folgendes Bild: Eine Person sollte als erste rekursive LISP-Funktion die Funktion SETDIFF kodieren. SETDIFF erhält zwei Listen als Argumente und liefert als Ergebnis eine Liste der Elemente, die nur in der ersten, nicht aber in der zweiten Liste enthalten sind. Die Person benötigte etwa eine Stunde, um diese Funktion zu kodieren. Sie zog dabei eine rekursive Funktion aus dem Lehrbuch von Siklossy (1976) zu Hilfe, die eine Liste der gemeinsamen Elemente zweier Listen liefert:

```
(INTERSECTION (LAMBDA (SET1 SET2)
  (COND ((NULL SET1 ())
        ((NULL SET2 ())
        ((MEMSET (CAR SET1) SET2)
         (CONS (CAR SET1) (INTERSECTION (CDR SET1) SET2))
        )
        ( T (INTERSECTION (CDR SET1) SET2))
) )          )
```

Die Person bildete zunächst eine Analogie zwischen INTERSECTION und der zu lösenden Aufgabe SETDIFF, in dem sie erkannte, daß SETDIFF ein Element aus SET1 nicht übernimmt (CDR SET1), wenn INTERSECTION dieses Element übernimmt (CAR SET1) und umgekehrt. Im Anschluß versuchte die Person, den Code von INTERSECTION so zu verändern, daß er den Anforderungen von SETDIFF genügte. Die erste Bedingung wurde demzufolge direkt übernommen. Die Person erkannte dann, daß die zweite Bedingung zu verändern ist: Enthält die zweite Liste kein Element, so muß SETDIFF die erste Liste zurückliefern. Die folgenden zwei rekursiven Fälle bereiten der Person Schwierigkeiten, obwohl die korrekte Analogie bereits erkannt wurde. Am Ende erkannte die Person, daß die korrekte Lösung durch eine Vertauschung des dritten und vierten Ausdrucks erzeugt werden kann:

```
(SETDIFF (LAMBDA (SET1 SET2)
  (COND ((NULL SET1 ())
        ((NULL SET2 SET1)
        ((MEMSET (CAR SET1) SET2) (SETDIFF (CDR SET1) SET2))
        ( T
         (CONS (CAR SET1) (SETDIFF (CDR SET1) SET2))
        )
) )      )
```

Das Vorgehen dieses und weiterer Probanden lieferte erste Hinweise, daß beim Erlernen rekursiver Programmierung die Nutzung von Analogien von entscheidender Bedeutung sind. Zugleich wurde offensichtlich, daß der Vergleichsprozeß zwischen Beispiel und Aufgabe für Anfänger keineswegs trivial ist. Die Beobachtung der Person bei der Lösung nachfolgender Aufgaben lieferte wichtige Hinweise, wie stark beim Problemlösen verwendete Analogien die Lösung nachfolgender Aufgaben beeinflussen. Als zweite Aufgabe sollte die Person eine Funktion kodieren, die prüft, ob eine Menge Teilmenge einer zweiten ist. Zur Lösung dieser Aufgabe benötigte die Person nur eine halbe Stunde. Wissensstrukturen die durch Lösung von SETDIFF mit hilfe von INTERSECTION inferiert wurden, konnten direkt zur Lösung der neuen Aufgabe verwendet werden. Für eine weitere Aufgabe, bei der die Potenzmenge

einer Liste zu bilden war, benötigte die Person dagegen *3,5* Stunden, wobei zusätzlich viele Hinweise vom Versuchsleiter gegeben wurden. Um die Potenzmenge einer Menge zu erzeugen, ist eine andere rekursive Struktur als für die zuvor gelösten Aufgaben notwendig (Baumrekursion versus lineare Rekursion).

Einzelfallstudien dieser Art liefern wertvolle Hinweise über längerfristige Lernprozesse. Die "Lautes-Denken"-Protokolle können Auskunft über Prozesse der Informationssuche und -verarbeitung bei der Lösung einer Aufgabe geben. Die skizzierte Studie von Pirolli und Anderson (1985) liefert zudem wertvolle Hinweise über die Abhängigkeit von Wissenserwerbsprozessen von den verwendeten Beispielen. Allerdings ist, anders als bei Experimenten, eine Generalisierung der Befunde über die analysierten Einzelpersonen hinaus nicht zulässig, da keine Kontrolle möglicher konfundierender Variablen (Vorwissen, Motivation, Versuchsleitereffekte etc.) vorgenommen werden kann.

7.4 Experimentell kontrollierte Erfassung des Erwerbs von rekursiven Problemlösefertigkeiten mit LEAR

In den vorangegangenen Abschnitten wurde argumentiert, daß der adäquate empirische Zugang zur Prüfung von Hypothesen das Experiment ist. Nur experimentelle Designs ermöglichen die Prüfung von in einem Modell (einer Theorie) formulierten Kausalannahmen. Allerdings umfaßten die oben dargestellten Experimente zum analogen Lernen nur einen Meßzeitpunkt, so daß lediglich rudimentäre Lernprozesse in wenig komplexen Wissensbereichen dadurch empirisch prüfbar sind. Im Bereich Programmieren Lernen wurden dagegen bisher hauptsächlich quasiexperimentelle Untersuchungen oder Einzelfallstudien durchgeführt. In Experten-Novizen-Vergleichen kann Aufschluß über die Wissensrepräsentation zu mehreren Erfahrungszeitpunkten gewonnen werden, die Lernerfahrung selbst bleibt dabei jedoch außerhalb der empirischen Kontrolle. Einzelfallstudien geben detaillierten Aufschluß über Lernprozesse in einer relativ natürlichen Lernumgebung, lassen jedoch keine Verallgemeinerung zu.

Um Annahmen über Lernprozesse in einem kognitiv komplexen Wissensbereich zu prüfen, ist ein experimenteller Zugang, der mehrere Meßzeitpunkte umfaßt, notwendig. Die einzige mir bekannte Untersuchung, die dieser Strategie folgt, stammt von Kessler und Anderson (1986; vgl. Kap. 7.3). Hier wird jedoch nicht analoges Lernen, sondern der Einfluß der Reihenfolge des Erwerbs iterativer versus rekursiver Kontrollstrukturen auf den Lernzuwachs geprüft. Zur empirischen Prüfung von Annahmen zum analogen Lernen rekursiver Programmiertechniken, so wie sie in Kapitel 6 formuliert wurden, wurde eine Versuchsumgebung geschaffen, die folgenden Kriterien genügen sollte:

- Der Erwerb rekursiver Programmiertechniken sollte in Interaktion mit einer Programmierumgebung am Computer stattfinden.
- Die Programmierumgebung sollte einfach zu bedienen sein, so daß auch Personen ohne Computererfahrung damit zurecht kommen.
- Die Interaktion der Lernenden mit der Programmierumgebung sollte vollständig erfaßbar sein.
- Der Lernprozeß sollte mehrere Sitzungen umfassen.
- Lehrmaterial (Beispiele) sollte kontrolliert dargeboten werden.
- Die funktionale Programmiersprache sollte einen möglichst geringen Umfang an Syntax haben, so daß der Erwerb rekursiver Programmiertechniken möglichst wenig von Problemen mit der korrekten syntaktischen Kodierung der Funktionen belastet ist.

Entsprechend habe ich die Lernumgebung LEAR entwickelt, die diesen Anforderungen genügt (siehe auch Anhang B.2). Als funktionale Sprache, wurde dabei die einfache Sprache, die in Kapitel 5.3 vorgestellt wurde, verwendet. Die Lernumgebung wurde auf einen Interpreter für diese Sprache aufgesetzt. Für den Benutzer stellt sich die Oberfläche von LEAR folgendermaßen dar (siehe Abb. 13):

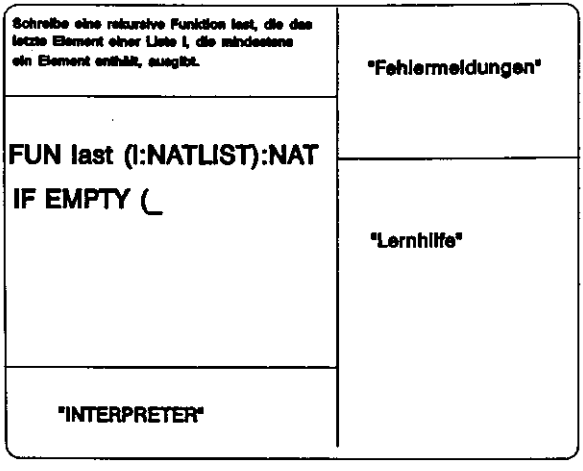


Abb. 13: Die Benutzeroberfläche von LEAR

Der Bildschirm ist in 5 Fenster eingeteilt. In der linken Bildschirmhälfte befinden sich Fenster für folgende drei Funktionsbereiche:

- Im Fenster "Aufgabe" wird eine Aufgabenstellung präsentiert beziehungsweise einige Eingaben zum Testen der Funktion vorgeschlagen.
- Im Editorfenster kann eine Funktion kodiert werden. Der Editor ist ein einfach zu bedienender Zeileneditor.
- Darunter ist ein Fenster für die Interaktion mit dem Interpreter. Hier können die kodierten Funktionen mit konkreten Eingabewerten ausgewertet werden.

In der rechten Bildschirmhälft befinden sich zwei weitere Fenster.

- Im oberen Fenster werden Fehlermeldungen gegeben. Syntaxfehler werden zusätzlich durch Curserpositionierung im Editorfenster signalisiert. Die Fehlermeldungen sind dabei sehr spezifisch, so daß die Korrektur von syntaktischen Fehlern mit möglichst wenig Aufwand verbunden ist. Für die in den Experimenten verwendeten Aufgaben auch semantische Fehler zurückgemeldet. Semantische Fehler werden durch interne Simulation von Interpreteraufrufen mit kritischen Testwerten ermittelt. Dadurch kann festgestellt werden, ob die Funktion die in der Aufgabe vorgegebene Semantik erfüllt und ob sie auch für "Extremfälle" (leere Listen, null) funktioniert. Nichtterminierende Funktionen werden über eine Kombination von Speicherplatzabnahme und Zeitbedarf bei der Interpretation ermittelt.
- Schließlich wird in einem Lernhilfefester das Lehrmaterial (Beispiele) vorgegeben.

In den beiden in Kapitel 8 dargestellten Untersuchungen wurde ein *Curriculum*, das vier Lektionen von jeweils ca. 3 Stunden umfaßt, verwendet (siehe Tab. 2). Die ersten beiden Lektionen dienten der Einübung der Syntax der Programmiersprache und dem Vertrautwerden mit der Lernumgebung. In den folgenden beiden Lektionen sollten rekursive Funktionen erstellt werden.

TABELLE 2
Curriculum zur Einführung von Grundkonzepten funktionaler Programmierung

```
                                LEKTION I
   Struktur von Funktionen, Parameter, Datentypen,
   arithmentische Anweisungen, Verschachtelung von Anweisungen

* Interaktive Einführung in die Bedienung von LEAR

                                Aufgabe 1.1
Schreibe eine Funktion malvier, die FUN malvier(x:NAT):NAT
eine Zahl x mit 4 multipliziert     MULT(x,4)
ausgibt.                            FIN

                                Aufgabe 1.2
Schreibe eine Funktion addxy, die   FUN addxy(x:NAT,Y:NAT):NAT
eine um 1 erniedrigte Zahl x zu     PLUS(MINUS(x,1),y)
einer Zahl y addiert.               FIN
Also (x-1) + y.
erster Parameter: x
zweiter Parameter: y

                                LEKTION II
   Boolsche Ausdrücke, Bedingte Anweisungen, Listen
   Listenoperationen
                                Aufgabe 2.1
Schreibe eine Funktion suby, die    FUN suby(x:NAT,y:NAT):NAT
0 ausgibt, wenn eine Zahl y         IF GREATER (y,x)
größer als eine Zahl x ist, und     THEN 0
sonst y von x subtrahiert.          ELSE MINUS (x,y)
Also wenn y > x dann 0 sonst x-y.   FIN
erster Parameter: x
zweiter Parameter: y

                                Aufgabe 2.2
Schreibe eine Funktion sub1, die    FUN sub1 (x:NAT):NAT
0 ausgibt, wenn die Zahl x gleich   IF EQUAL(x,0)
0 ist, und sonst x um 1 erniedrigt. THEN 0
                                    ELSE MINUS(x,1)
                                    FIN

                                Aufgabe 2.3
Schreibe eine Funktion sec, die     FUN sec (l:NATLIST):NAT
das zweite Element einer Liste l    HEAD(TAIL(l))
ausgibt.                            FIN


                                Aufgabe 2.4
Schreibe eine Funktion rest, die    FUN rest (l:NATLIST):NATLIST
NIL ausgibt, wenn die Liste l       IF EMPTY(l)
leer ist und sonst l ohne ihr       THEN NIL
erstes Element ausgibt.             ELSE TAIL(l)
                                    FIN

* Interaktives Frage-Antwort-Progamm zur Syntax
```

TABELLE 2 (Fortsetzung)

LEKTION III

Einführung in das Konzept der Rekursion

(3 endrekursive Aufgaben)

Aufgabe 3.1

Schreibe eine rekursive Funktion dubx, die eine Zahl z x-mal verdoppelt.
erster Parameter: zu verdoppelnde Zahl z
zweiter Parameter: Anzahl d. Verdopp. x

Beispiel: z=3, x=2
dubx(3,2)->dubx(6,1)->dubx(12,0) -> 12

```
FUN dubx(z:NAT,x:NAT):NAT
IF EQUAL(x,y)
THEN z
ELSE dubx(PLUS(z,z),
     MINUS(x,1))
FIN
```

Aufgabe 3.2

Schreibe eine rekursive Funktion last, die das letzte Element einer Liste l, die mindestens ein Element enthält, ausgibt.

Beispiel: l=list(1,2,3)
last(list(1,2,3)) -> 3

```
FUN last(l:NATLIST):NAT
IF EMPTY(TAIL(l))
THEN HEAD(l)
ELSE last(TAIL(l))
FIN
```

Aufgabe 3.3

Schreibe eine rekursive Funktion member, die 1 liefert, wenn Element n in Liste l enthalten ist, und 0, wenn n nicht in l ist.
erster Parameter: Zahl n
zweiter Parameter: Liste l

Beispiel: n=2, l=list(1,2,3)
member(2,list(1,2,3)) -> 1

```
FUN member(n:NAT,l:NATLIST):NAT
IF EMPTY(l)
THEN 0
ELSE IF EQUAL(n,HEAD(l))
     THEN 1
     ELSE member(n,TAIL(l))
FIN
```

LEKTION IV

(3 teil-rest-rekursive Aufgaben)

Aufgabe 4.1

Schreibe eine rekursive Funktion summe, die den Summenwert einer Zahl x mit all ihren Vorgängern bis 0 berechnet.

Beispiel: x=5 ergibt 5+4+3+2+1+0 = 15

```
FUN summe(x:NAT):NAT
IF EQUAL(x,0)
THEN 0
ELSE PLUS(x,summe(
       MINUS(x,1)))
FIN
```

Aufgabe 4.2

Schreibe eine rekursive Funktion genlist, die eine Liste absteigender Zahlen von n bis 1 ausgibt.

Beispiel: n=5 ergibt (5,4,3,2,1)

```
FUN genlist(n:NAT):NATLIST
IF EQUAL(n,0)
THEN NIL
ELSE CONS(n,genlist(
       MINUS(x,1)))
FIN
```

Aufgabe 4.3

Schreibe eine rekursive Funktion suml, die eine Liste ausgibt, die die Summen je zweier benachbarter Elemente der Liste l enthält.
l enthält mindestens ein Element.

Beispiel: l=list(6,4,1) ergibt (10,5)

```
FUN suml(l:NATLIST):NATLIST
IF EMPTY(TAIL(l))
THEN NIL
ELSE CONS(PLUS(HEAD(l),
            HEAD(TAIL(l))),
       suml(TAIL(l)))
FIN
```

Der Ablauf des Programms ist folgendem Flußdiagramm zu entnehmen (vgl. Abb. 14):

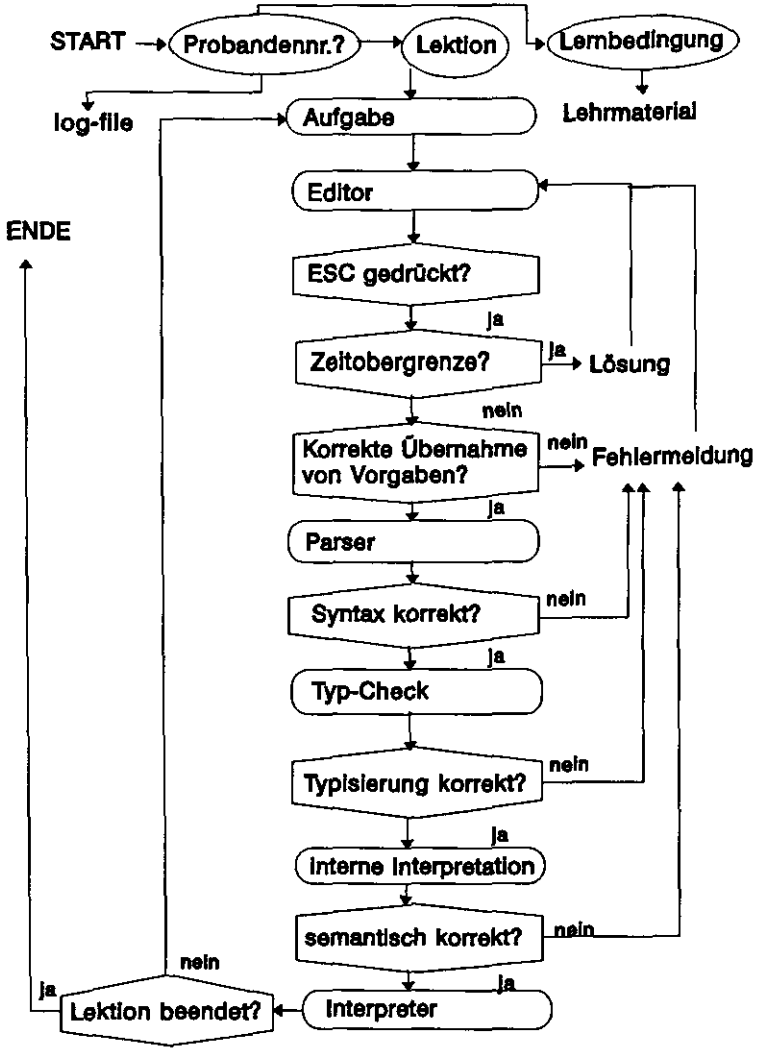


Abb. 14: Flußdiagramm für den Ablauf einer Sitzung mit LEAR

Zunächst wird eine Kennummer für die Person erfragt, die mit dem System arbeitet. Aus einer Informationsdatei wird die aktuelle Lektion sowie die Versuchsbedingung erfragt. Die Versuchsbedingung gibt vor, welche Art von Lehrmaterial der Proband erhält. Im folgenden werden alle Eingaben in einem *log-file* protokolliert. Im *log-file* werden Lektionsnummer und Aufgabennummern vermerkt und alle Eingaben zusammen mit Zeiten protokolliert.

Die erste Aufgabenstellung der aktuellen Lektion wird im Aufgabenfenster ausgegeben. Bei der Kodierung der Funktionen müssen dabei der vorgegebene Funktionsname sowie die Paramternamen und die Parameterreihenfolge aus der Aufgabenstellung übernommen werden. Diese Restriktion war notwendig, um die semantischen Fehler durch eine einfach gehaltene interne Simulation ermitteln zu können.

Im folgenden ist der Curser im Editorfenster positioniert. Der Editor kann durch Drücken der *ESC*-Taste verlassen werden. Hier wird dann zunächst geprüft, ob ein Zeitmaximum überschritten ist. Um die Untersuchungen in Gruppen durchführen zu können, wurde als Zeitobergrenze *30* bzw. *20* Minuten pro Aufgabe vorgegeben. Ist die Zeitobergrenze überschritten, so wird im Lehrhilfefenster die korrekte Funktion vorgegeben.

Ansonsten wird die Funktion parsiert. Hier wird zunächst geprüft, ob die in der Aufgabenstellung vorgegebenen Namenskonventionen eingehalten wurden. Danach wird die Funktion auf syntaktische Korrektheit überprüft. Wird ein Syntaxfehler gefunden, so wird dieser im Fehlerfenster gemeldet und der Curser wird im Editor an die Fehlerposition gesetzt. Analog werden in einem nächsten Schritt Typfehler ermittelt und gemeldet. Ist auch die Typisierung fehlerfrei, so wird die Funktion mit verschiedenen kritischen Testwerten dem Interpreter übergeben und die Auswertung wird - nach außen nicht sichtbar - durchgeführt. Werden semantische Fehler ermittelt, so werden diese ebenfalls im Fehlerfenster angegeben. Der Curser wird auf die erste Zeile der Funktion positioniert.

Nur wenn die Funktion syntaktisch und semantisch korrekt ist, kann der Editor verlassen werden und der Curser wird ins Interpreterfenster gesetzt. Die Funktion kann nun mit konkreten Testwerten aufgerufen werden, das Tracing der Funktionsauswertung wird im Hilfefenster ausgegeben. Sind alle Aufgaben der Lektion bearbeitet worden, wird das Programm beendet, ansonsten wird die nächste Aufgabe vorgegeben.

Die im folgenden Kapitel dargestellten Untersuchungen wurden mit dieser Lernumgebung durchgeführt. Die dort konkret verwendeten Lernhilfen sind den entsprechenden Abschnitten zu entnehmen.

8 Experimentelle Untersuchungen zum induktiven Erwerb rekursiver Programmiertechniken

Im folgenden werden zwei empirische Untersuchungen zum induktiven Erwerb von rekursiven Programmiertechniken vorgestellt. Dabei wird zunächst der allgemeine Rahmen für beide Untersuchungen skizziert. Die untersuchungsspezifischen Hypothesen und Ergebnisse werden dann in den entsprechenden Abschnitten berichtet. Untersuchung I diente unter anderem dazu, die Tauglichkeit des grundlegenden Vorgehens zu erproben und zu prüfen, ob das Lernen von rekursiven Programmiertechniken mit Beispielen im Rahmen der gewählten Untersuchungsstrategie zu einem mit den entwickelten Meßinstrumenten erfaßbaren Lernerfolg führt. Insbesondere mußte zunächst gewährleistet werden, daß die Lernumgebung LEAR, sowie das viertägige Curriculum geeignet sind (vgl. Kap. 7.4), um Personen ohne Computervorerfahrung und mit wenig formaler Vorbildung in wenigen Tagen die Grundlagen rekursiver Programmierung zu vermitteln. Eine sinnvolle Prüfung von Hypothesen über Lernprozesse ist nur dann möglich, wenn die Personen generell ein Mindestmaß an Lernleistung erbringen können, da sonst aufgrund von *bottom*-Effekten keine Unterschiede bezüglich der verwendeten Lernbedingungen erfaßbar sind. Zudem wurde geprüft, ob das Lernen mit Beispielen sich mit den gewählten Meßinstrumenten von einem anderen Lehrmaterial unterscheiden läßt.

In Untersuchung II konnte dann davon ausgegangen werden, daß die entwickelte Untersuchungsprozedur geeignet ist, um Lernprozesse zu erfassen. Hier wurde das Lernen mit Beispielen dann spezifischer untersucht, indem der Einfluß der Ähnlichkeit, die Beispiele und zu lösende Aufgaben aufweisen, auf den Lernprozeß und den Wissenserwerb geprüft wurde.

8.1 Überblick über die Untersuchungen

Beiden empirischen Untersuchungen liegt ein gemeinsamer Ablaufplan zugrunde, der zum besseren Überblick den Untersuchungen vorangestellt wird. Die in beiden Untersuchungen verwendeten Lehr- und Testmaterialien werden ebenfalls bereits an dieser Stelle vorgestellt.

TABELLE 3
Untersuchungsablauf für Untersuchung I und II
(spezifische Erhebungen in Klammern)

1. Tag Begrüßung > Erhebung von demographischen Daten, Computervorerfahrung & Mathematikkenntnissen (30 min) (> I: IST-70 ZR,RA 30 min) *** Einführungsphase *** Unterricht zu Lektion I (60 min) Def. von Funktionen, Datentypen, arithmetische Anweisungen, Verschachtelte Anweisungen Pause (10 min) Hinweise zur PC-Bedienung LEKTION I: (60 min) Programmieren von 2 nichtrekursiven Funktionen	3. Tag > Syntaxtest (30 min) *Einf. in die Rekursion* (30 min) *** Experimentelle Phase *** (> II: **1. Durchgang Sortierversuche (30 min)** Pause (10 min) **LEKTION III: (75 min)** **Programmieren von drei endrekursiven Funktionen**
2. Tag Wiederholung der Syntax aus Lektion I (30 min) Unterricht zu Lektion II(60 min) Boolsche Ausdrücke, bedingte Anweisungen, Listen Pause (10 min) LEKTION II: (90 min) Programmieren von 4 nichtrekursiven Funktionen	4. Tag (> II: **2. Durchgang Sortierversuche (30 min)** **LEKTION IV: (75 min)** **Programmieren von drei teilrest-rekursiven Funktionen** Pause (10 min) (> II: **3. Durchgang Sortierversuche (30 min)** > **Abschlußtest (30 min)** Entlohnung

Das Curriculum ist so konzipiert, daß in den ersten beiden Lektionen der Umgang mit der Programmierumgebung sowie die Syntax der funktionalen Sprache vermittelt wird und in zwei darauf folgenden Lektionen jeweils drei rekursive Funktionen zu programmieren sind.

In der ersten Lektion wird dabei der Aufbau von Funktionen, das Parameterkonzept, Datentypen und arithmetische Anweisungen und die Verschachtelung von Anweisungen vermittelt. In einem interaktiven Programm wird die Bedienung der Programmierumgebung eingeübt. In der zweiten Lektion werden bedingte Anweisungen, boolsche Operatoren, Listen und Listenanweisungen eingeführt. Am Ende der Lektion wird in einem interaktiven Frage-Antwortprogramm die gesamte Syntax nocheinmal wiederholt. In den ersten beiden Lektionen wird der Stoff jeweils zunächst im Frontalunterricht eingeführt und danach sind (zwei bzw. vier) Aufgaben am Rechner zu bearbeiten. Der vermittelte Stoff steht jeder Person in einem Handbuch zur Verfügung, zusätzlich wurde ein Merkblatt zur Bedienung der Lernumgebung, insbesondere die Editorbefehle, und ein Merkblatt, auf dem die Syntax der funktionalen Sprache zusammengefaßt ist, ausgegeben. Die Handbücher standen nur zur jeweiligen Lektion zur Verfügung, die Merkblätter durften während des gesamten Versuches verwendet werden. Materialien durften grundsätzlich nicht mit nachhause genommen werden. Während der ersten beiden Lektionen waren Nachfragen jederzeit erlaubt.

Zu Beginn der dritten Lektion wurde eine kurze mündliche Einführung in das Prinzip der Rekursion gegeben. Es wurde die Fakultätsfunktion und eine endrekursive Funktion vorgestellt und die Arbeitsweise der Funktionen wurde an Handsimulationen verdeutlicht. Es wurde darauf hingewiesen, daß rekursive Funktionen stets bedingte Anweisungen benötigen, daß es mindestens einen direkten und einen rekursiven Fall geben muß und daß ein Parameter im rekursiven Aufruf so verändert werden muß, daß die Abbruchbedingung wahr werden kann und somit der direkte Fall erreicht wird. Im folgenden waren keine Fragen mehr erlaubt. Zur Unterstützung standen nur die beiden Merkblätter, sowie die von der Lernumgebung versuchsspezifisch präsentierte Lernhilfe zur Verfügung. In Untersuchung I handelte es sich hier um Beispiele und Erklärungstexte, in Untersuchung II wurden prinzipiell Beispiele zur Verfügung gestellt (vgl. Kap. 8.2 und 8.3). In der dritten Lektion waren drei endrekursive und in der vierten Lektion drei teil-rest-rekursive Funktionen zu programmieren, dies wurde den Probanden jedoch nicht mitgeteilt. Für jede Aufgabe waren *30* (in Untersuchung I) beziehungsweise *20* Minuten (in Untersuchung II) als Zeitlimit gesetzt.

Über die vier Lektionen hinweg wurden verschiedene Kennwerte zur Beschreibung der Stichprobe, Kontrollvariablen sowie die abhängigen Variablen zur Erfassung des lernbedingungsspezifischen Wissenserwerbs erhoben.

In der ersten Lektion wurden in einem Fragebogen demographische Daten (Alter, Geschlecht, Studienfach, Semesterzahl) sowie Computervorerfahrung und Mathematikkenntnisse erhoben. Computererfahrung und Mathematikkenntnisse dienten als Kontrollvariablen. Es sollte gewährleistet sein, daß die Probanden über die Lernbedingungen hinweg vergleichbare Computer- und Mathematikkenntnisse aufweisen, da anzunehmen ist, daß diese den Lernerfolg mitbeeinflussen.

In Untersuchung I wurde zusätzlich die Begabung zum formal-mathematischen Denken mit zwei Untertests des IST-70 (Amthauer 1970) erfaßt. Der Untertest ZR (Zahlenreihen) mißt die Fähigkeit zum induktiv formalen denken, der Untertest RA (Rechenaufgaben) mißt die Fähigkeit zum deduktiven Denken mit Zahlen. Zu Beginn der dritten Lektion wurde ein Syntaxtest durchgeführt. Die Rohsummenwerte des Tests dienten als Kontrollvariable, um zu gewährleisten, daß die in den ersten beiden Sitzungen erworbenen Syntaxkenntnisse zwischen den Lernbedingungen vergleichbar waren.

Zur Erfassung des Wissenserwerbs wurden aus den *log-files* bei der Bearbeitung der Aufgaben der dritten und vierten Lektion verschiedene Kennwerte ermittelt. Hierzu gehören insbesondere, ob eine Aufgabe gelöst wurde oder nicht, sowie die Lösungszeit. In Untersuchung I wurde zusätzlich erhoben, wie häufig und wie lange eine Lernhilfe genutzt wurde. Zudem wurden die beim Kodieren der Funktionen begangenen Fehler aus den *log-files* ermittelt (vgl. Kap. 8.2).

Am Ende der vierten Lektion wurde das erworbene Wissen über Rekursion in einem Abschlußtest erhoben, der in *30 Minuten* zu bearbeiten war (siehe Anhang D.3). Dieser Test diente zur Erfassung des Lerntransfers, so wie dies in den in Kapitel 7.1 dargestellten Untersuchungen zur Schemainduktion berichtet wurde. Allerdings wurden, anders als in diesen Untersuchungen, nicht nur echte Transferaufgaben gestellt, sondern zusätzlich andere Aspekte, die das Verständnis rekursiver Programmiertechniken umfaßt, abgefragt. Der Test besteht aus *21* (Untersuchung I) bzw. *22* (Untersuchung II) Einzelitems, die vier Aspekte von Wissen über Rekursion erfassen:

(1) **Abstraktes Wissen über Rekursion** wurde in Mehrfachwahlaufgaben erfaßt (7 Items).
(2) **Verständnis rekursiver Funktionen** wurde über eine Auswahl der rekursiven Funktionen erfaßt, die eine vorgegebene Aufgabenstellung korrekt lösen (4 Items; vgl. Kahney 1989; Kap. 7.2).
(3) **Verständnis des Ablaufs rekursiver Funktionen** wurde durch Handsimulationen zur Berechnung von Funktionswerten für vorgegebene Eingabeparameter und zur Angabe von Eingabeparametern bei vorgegebenen Funktionsergebnissen erfaßt (8 Items).
(4) Zur Erfassung des **Lerntransfers** war eine weitere endrekursive sowie eine weitere teil-rest-rekursive Funktion zu kodieren. In Untersuchung II war zusätzlich eine Simulatanrekursion zu kodieren (2 bzw. 3 Items).

Damit wurde sowohl konzeptuelles (deklaratives) Wissen über rekursive Funktionen als auch die Anwendung der erworbenen Problemlösetechniken für verschiedene Anforderungen erhoben. Um eine zuverlässige Skala zur Erfassung des erworbenen Wissens zu erhalten, wurden die Testrohwerte in beiden Untersuchungen raschskaliert (vgl. Sixtl 1967; Kap. 7).

In Untersuchung II wurde neben dieser indirekten Erfassung des Schemaerwerbs zusätzlich mittels Sortieraufgaben versucht, die Struktur der induzierten Schemata auf direktere Art zu erheben. Zu diesem Zweck hatten die Probanden eine Menge von *16* rekursiven und *8* nicht-rekursiven Funktionen zu drei Meßzeitpunkten nach Ähnlichkeit zu sortieren (vgl. Kap. 8.3).

Zum Abschluß der Untersuchung wurden die Probanden in einem kurzen Fragebogen gebeten, die Untersuchung für sich zu bewerten. Hierzu wurden ihnen unter anderem Fragen nach dem selbsteingeschätzten Lernerfolg und nach dem eingeschätzten Unterstützungswert der Lehrmaterialien (Beispiele) gestellt.

8.2 Untersuchung I: Der Einfluß von Beispielfunktionen und erklärenden Texten auf den Erwerb rekursiver Programmiertechniken[4]

In Untersuchung I wird der Einfluß von Beispielfunktionen mit dem Einfluß von erklärenden Texten auf den Erwerb rekursiver Programmiertechniken verglichen. Die Beispiele hatten dabei maximale strukturelle Ähnlichkeit zu den Aufgaben. Sie unterschieden sich nur in Oberflächenmerkmalen wie Namen der Funktion, Namen der Parameter und verwendeter Verknüpfungs- oder Hilfsfunktion (vgl. Kap. 6.2). So sind sich zum Beispiel die Funktionen *summe* und *faku* maximal strukturähnlich (vgl. Kap. 6.3).

In Kapitel 6.3 wurde dargestellt, wie durch Vergleich von Beispiel und Aufgabe und Anpassung des Lösungswegs des Beispiel an die Anforderungen der Aufgabe eine Aufgabenlösung inferiert wird. Die entdeckten Gemeinsamkeiten von Beispiel und Aufgabe werden dann in einer generalisierten Wissensstruktur repräsentiert. Lernen mit Beispielen ermöglicht also den Aufbau allgemeinerer Wissensstrukturen. Bei wenig Lernerfahrung ist jedoch die Inferenz der zur Aufgabenlösung notwendigen Regeln aufwendig und fehleranfällig. Aus diesem Grund wurden als alternatives Lehrmaterial erklärende Texte erstellt, die die Regeln, die zur Kodierung der Aufgabe notwendig sind, direkt vorgeben. Tabelle 4 zeigt ein Beispiel und eine Erklärung für die Aufgabe *genlist*. Alle in der Untersuchung verwendeten Beispiele und Erklärungen sind Anhang D.1 zu entnehmen.

---

[4]Dieser Untersuchungsbericht wurde bereits zur Veröffentlichung eingereicht und erscheint als:
Schmid, U. (1994). Programmieren lernen: Der Einfluß von Beispielfunktionen und erklärenden Texten auf den Erwerb rekursiver Programmiertechniken. *Kognitionswissenschaft*, *4*, (1), 47-54.

TABELLE 4
Beispiel und Erklärung für die Aufgabe *genlist*

Aufgabenstellung: Schreibe eine Funktion genlist, die eine Liste absteigender Zahlen von n bis 1 ausgibt.

Beispiel: Die Funktion slist erzeugt eine Liste, die n-mal die Zahl 7 enthält:

```
FUN slist (n:NAT):NATLIST
IF EQUAL(n,0)
THEN NIL
ELSE CONS(7,slist(MINUS(n,1)))
FIN
```

Erklärung: *Um eine Liste absteigender Zahlen von n bis 1 zu erzeugen, wird im Fall, daß n = 0 ist, die leere Liste ausgegeben. Anderenfalls wird mit dem Listenkonstruktor der aktuelle Wert von n mit dem rekursiven Aufruf der Funktion verknüpft. Dabei wird n im rekursiven Aufruf jeweils um eins minimiert.*

Lösung:
```
FUN genlist(n:NAT):NATLIST
IF EQUAL(n,0)
THEN NIL
ELSE CONS(n,genlist(minus(n,1)))
FIN.
```

Um Hypothesen über den Einfluß von Beispielen und Erklärungen auf den Wissenserwerb zu formulieren, wird das Konzept der Informations- bzw. Berechnungsäquivalenz von Larkin und Simon (1987) verwendet. Die Autoren schlagen dieses Konzept vor, um die Äquivalenz von Repräsentationen (Text versus Diagramm) zu beurteilen. Zwei Materialien sind informationsäquivalent, wenn alle Informationen, die in einem Lehrmaterial explizit gegeben werden, aus dem jeweils anderen Material inferiert werden können. Zwei Materialien sind berechnungsäquivalent, wenn sie informationsäquivalent sind und zudem die in einem Material explizit gegebene Information "schnell und einfach" aus dem jeweils anderen Material inferiert werden kann.

Diese etwas vagen Definitionen können folgendermaßen auf den Erwerb von rekursiven Programmiertechniken mit Erklärungen und Beispielen übertragen werden: Berechnungsäquivalenz bezieht sich auf den Lösungsaufwand von Aufgaben bei Vorgabe eines Materials, Informationsäquivalenz auf den resultierenden Wissenserwerb. Der Lösungsaufwand sollte bei Unterstützung durch Beispiele größer sein, als bei Unterstützung durch Erklärungen, da, wie in Kapitel 6.3 beschrieben, bei der Beispielübertragung aufwendige Inferenzprozesse notwendig sind, während die Regeln zur Lösung der Aufgabe in der Erklärung direkt vorgegeben sind. Ich nehme also an, daß Erklärungen und Beispiele nicht berechnungsäquivalent sind. Dagegen sollte der resultierende Wissenserwerb beim Arbeiten mit Beispielen mindestens genauso hoch sein, wie beim Arbeiten mit Erklärungen. Die beiden Materialien sind dann informationsäquivalent, wenn die beim Lernen erworbenen Wissens-

strukturen gleichermaßen auf Transferaufgaben anwendbar sind und wenn eine Vorgabe beider Materialien keinen zusätzlichen Lerngewinn im Vergleich zum Lernen mit nur einem Material bringt.

Diese Annahmen wurden bereits in einigen Experimenten im Bereich mathematisches Problemlösen überprüft (vgl. Kap. 7.1). Beim Lösen mathematischer Textaufgaben zeigte sich, daß Arbeiten mit Beispielen zu höherem Lösungserfolg der Aufgaben führt als Arbeiten mit der Vorgabe von Lösungsregeln (Reed & Bolstad 1991). Beim Lernen deduktiver bzw. statistischer Schlüsse führten Beispiele und verbal formulierte Erklärungen von Problemlöseregeln zu vergleichbarem Lerntransfer (Cheng, Holyoak, Nisbett & Oliver 1986; Fong, Krantz & Nisbett 1986). Untersuchungen, die nicht nur den resultierenden Lerngewinn, sondern auch den Lösungserfolg während des Lernprozesses betrachten, sind mir nicht bekannt.

Variablen und Hypothesen

Um die Annahmen über das Lernen mit Beispielen im Vergleich zum Lernen bei Vorgabe der Problemlöseregeln in erklärenden Texten empirisch zu prüfen, wurde die unabhängige Variable "Lehrmaterial" dreifach gestuft: Eine Probandengruppe erhält Beispiele, eine zweite Erklärungen und eine dritte beide Materialien.

Die Hypothese, daß Beispiele und Erklärungen informationsäquivalent sind, wurde über einen Vergleich der Transferleistungen nach Abschluß der Lernphase geprüft. Als abhängige Variable wurden die Fähigkeitsparameter der raschskalierten Abschlußtestwerte verwendet (vgl. Kap. 8.1). Die Materialien sind dann informationsäquivalent, wenn sich keine Unterschiede zwischen der Beispiel- und Erklärungsgruppe ergeben und wenn ein Lernen mit beiden Materialien ebenfalls keinen Vorteil erbringt.

Die Hypothese, daß Beispiele und Erklärungen nicht berechnungsäquivalent sind, wurde über den Aufgabenlösungserfolg während der Lernphase geprüft. Hier wurden drei verschiedene abhängige Variablen betrachtet: Erstens wurde die Anzahl der in der vorgegebenen Zeit (jeweils *30* Minuten) korrekt gelösten Aufgaben betrachtet. Probanden, die mit Beispielen lernen, sollten weniger Aufgaben innerhalb des Zeitlimits lösen, als Probanden der beiden anderen Lerngruppen.

Zudem sollten sich die Gruppen bezüglich der Art der Fehler, die ihnen beim Kodieren der Aufgabe unterlaufen unterscheiden. Probanden, die mit Erklärungen arbeiten, müßten mehr syntaktische Fehler unterlaufen als Probanden, die mit Beispielen arbeiten. Während Probanden der Beispielgruppe eine korrekte syntaktische Funktion als Muster verwenden können, müssen Probanden der Erklärungsgruppe die Umsetzung der Funktion in die Syntax der Programmiersprache selbst leisten. Dagegen sollten Probanden der Beispielgruppe mehr

semantische Fehler bei der Kodierung der Funktion begehen. Zu Beginn des Wissenserwerbs sind wenig Wissensstrukturen vorhanden, die beim Vergleich von Aufgabe und Beispiel sowie bei der Anpassung der Aufgabe an das Beispiel genutzt werden können. Aus diesem Grund sollten vor allem solche semantischen Fehler begangen werden, die auf fehlerhafte Übernahme von Strukturen aus den Beispielen zurückzuführen sind. Probanden, die beide Lehrmaterialien zur Verfügung haben, sollten sowohl weniger syntaktische als auch semantische Fehler begehen als die beiden anderen Lerngruppen.

Schließlich sollte sich die unterschiedliche Berechnungsäquivalenz auch in der Intensität der Nutzung der Lehrmaterialien widerspiegeln. Da das Lösen von Aufgaben mit Beispielen umfangreiche Inferenzen beim Vergleich und der Anpassung der Beispiele an die Aufgabe benötigen, sollten Beispiele häufiger und länger betrachtet werden, als Erklärungen.

Methode

Stichprobe. An der Untersuchung nahmen *49* Psychologiestudenten der TU Berlin teil. Davon waren *36* Personen weiblichen und *13* Personen männlichen Geschlechts. Die Personen waren zwischen *20* und *42* Jahren alt, mit einem mittleren Alter von *26* Jahren. Sie befanden sich im ersten bis achten Semester, im Mittel im zweiten Semester. *31* Personen hatten Computererfahrung (Textverarbeitung, Statistikpakete), *4* dieser Personen hatten rudimentäre Programmiererfahrungen mit imperativen Sprachen (BASIC bzw. PASCAL). Die Probanden wurden zufällig einer der drei Lernbedingungen zugeteilt: *17* Probanden erhielten Erklärungen und jeweils *16* Probanden Beispiele oder beide Materialien.

Material und Durchführung. Die verwendeten Instruktions- und Testmaterialien, sowie der Ablauf der Untersuchung wurden bereits in Kapitel 8.1 dargestellt. Die Lehrmaterialien (Beispiele, Erklärungen, beides) wurden in der dritten und vierten Lektion im Hilfefenster von LEAR angeboten. Um die Häufigkeit und Dauer der Nutzung der Materialien erfassen zu können, mußten diese per Tastendruck angefordert werden und wieder abgeschaltet werden, um weiter an der Aufgabenlösung zu arbeiten. Für jede der 6 rekursiven Programmieraufgaben aus Lektion *3* und *4* konnten zwei Beispiele und zwei Erklärungen angefordert werden. Diese Materialien sind Anhang D.1 zu entnehmen.

Die bearbeiteten Untertests des IST-70 (Amthauer 1970) wurden nicht nur zur Erfassung der Kontrollvariable "Begabung zum formal-mathematischen Denken" erhoben, sondern auch, um die Probanden nach ihren kognitiven Stilen zu unterscheiden. Dieser Aspekt der Untersuchung wird hier nicht berichtet, die entsprechenden Hypothesen und Ergebnisse sind in Schmid und Ulber (1993) nachzulesen.

Ergebnisse und Interpretation

Zunächst wurden die erfaßten Voraussetzungen für das Lernen von Rekursion für die Gesamtstichprobe kontrolliert und geprüft, ob die Voraussetzungen bei den drei Lernbedingungen vergleichbar waren (Tabelle 5).

TABELLE 5
Leistungsniveau der Probanden

		Syntaxtest	Mathematikbegabung Note[a]	Selbsteinsch.[b]	IST-70 RA	ZR
Gesamt	(N=49)	15.4 (5.3)	2.8 (1.2)	2.5 (0.9)	103.4 (11.6)	106.4 (11.3)
Erklärungen	(n=17)	15.9 (6.0)	2.3 (1.1)	2.8 (0.8)	106.9 (11.0)	108.1 (12.0)
Beispiele	(n=16)	14.7 (4.6)	3.2 (1.3)	1.9 (0.9)	100.6 (11.6)	102.4 (11.6)
beides	(n=16)	15.5 (5.4)	3.0 (1.0)	2.7 (0.9)	102.5 (11.8)	108.7 (9.6)

Anmerkungen.
Angegeben sind arithemtisches Mittel und Standardabweichung in Klammern.
[a]: Die Note ist als Schulnote (1 = sehr gut bis 6 = ungenügend) angegeben.
[b]: Die Selbsteinschätzung wurde auf einer Skala von 1 (wenig begabt) bis 5 (sehr begabt) gegeben.

Das erworbene Syntaxwissen der Probanden war ausreichend zur Kodierung rekursiver Funktionen. Zu Beginn der experimentellen Bedingungsvariation gab es keine Unterschiede zwischen den drei Versuchsgruppen (ANOVA über die Anzahl korrekt gelöster Items mit dem Faktor Lernbedingung; $F < 1$). Die Fähigkeit zum formalen Denken war bei den Probanden durchschnittlich. Die letze Mathematiknote war im Schnitt eine *3*, die Selbsteinschätzung auf einer fünfstufigen Ratingskala war im Schnitt als mittelmäßig begabt und die Intelligenztestskalen der IST-70 Untertests lagen etwa bei dem für Psychologen vorausgesetzten Niveau (Skalenwert von *105* für beide Untertests; siehe Amthauer, 1970). In allen genannten Maßen schnitten die Probanden, die der Beispielbedingung zugeordnet waren, etwas schlechter ab, als die der anderen beiden Bedingungen.

Transferleistung. Der Abschlußtest wurde raschskaliert, um einen Kennwert der Transferleistung für jeden Probanden zu erhalten. Dabei konnte das Itemmodell sowie das Personenmodell nach Entfernung eines der Wissensitems angenommen werden (Chi^2-Anpassungstest mit $z=-.09$ für Itemmodell und $z=.83$ für Personenmodell). Die Probanden lösten im

Schnitt 9 der verbliebenen 20 Items. Die Lernbedingungen unterschieden sich nicht bezüglich ihrer Transferleistungen (ANOVA über die Personenwerte des Raschmodells mit Faktor Lernbedingung; $F < 1$). Die durch die Items erfaßten vier verschiedenen Wissensaspekte wurden unterschiedlich gut beherrscht, wobei das selbständige Programmieren weiterer rekursiven Funktionen am wenigsten gelang (siehe Tab. 6).

TABELLE 6
Transferleistungen der Gesamtstichprobe (N=49) in Abhängigkeit von der geprüften Wissensart

Wissensart	Itemzahl	Itemschwierigkeit*
deklaratives Wissen	6	85,7
Korrektheit von Funktionen	4	32,7
Ein-Ausgabe-Verhalten	8	26,5
Programmierung	2	14,3

Anmerkung.
*: Prozentwert mit
Itemschwierigkeit in % = (Anzahl gelöster Items / N) * 100

Auch bezüglich der Beherrschung der verschiedenen Wissensaspekte ergaben sich keine bedeutsamen Unterschiede zwischen den Lernbedingungen (ANOVA über die Anzahl korrekt gelöster Items mit den Faktoren Lernbedingung und Wissensaspekt; signifikanter Haupteffekt des Faktors Wissensaspekt: $F = 54.28$, $p < .001$; Interaktion: $F < 1$).

Performanz beim Lernprozeß: Aufgabenlösung. Die Performanz wurde durch die Anzahl der korrekt programmierten rekursiven Funktionen der Lektionen 3 und 4 erfaßt. Bezüglich des Lösungserfolgs der rekursiven Programmieraufgaben ergaben sich signifikante Unterschiede zwischen den Lernbedingungen ($Chi^2 = 9.21$, $p < .001$), obwohl das am Ende des Curriculums erworbene Verständnis rekursiver Programmierung, so wie es im Transfertest erfaßt wurde, gleichwertig ist. Die Probanden, die mit Beispielen als Lehrmaterial arbeiteten, lösten dabei deutlich weniger Aufgaben in der vorgegebenen Zeit, als die Probanden, die Erklärungen oder beide Materialien zur Verfügung hatten (siehe Abb. 15).

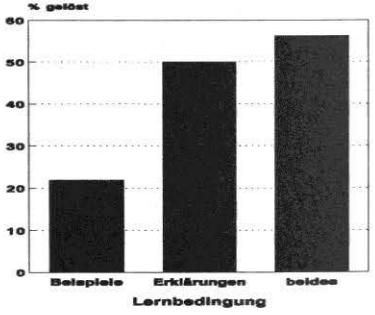


Abb. 15: Anzahl gelöster Aufgaben in Abhängigkeit von der Lernbedingung

Performanz beim Lernprozeß: Fehlerklassen. Die Lerngruppen unterschieden sich in der Art der Fehler, die sie beim Versuch, die Aufgaben zu lösen, produzieren: Probanden, die nur Erklärungen zur Verfügung hatten, erzielten lediglich bei der Klammersetzung höhere Fehlerwerte (Erklärungsgruppe: *62*, Beispielgruppe: *36*, Gruppe mit beiden Lernhilfen: *31* Fehler; *Kruskal-Wallis H-Test 9.561, $p < .05$*). Zudem erzeugten sie mehr Lösungen, die syntaktische Fehler enthalten, aber semantisch korrekt sind (Erklärungsgruppe: *42*, Beispielgruppe: *27*, Gruppe mit beiden Lernhilfen: *18* Fehler; *Kruskal-Wallis H-Test 6.979, $p < .05$*).

Probanden, die nur mit Beispielen arbeiteten, hatten dagegen einen sehr hohen Anteil an semantischen Fehlern. Diese Fehler sind vor allem darauf zurückzuführen, daß einerseits relevante Strukturgleichheiten von Beispiel und Aufgabe nicht erkannt wurden und andererseits für die rekursive Struktur irrelevante Anteile der Beispiele übernommen wurden (vgl. Tab. 7).

TABELLE 7
Vorkommen semantischer Fehler bei der Aufgabenlösung in Abhängigkeit von der Lernbedingung

Fehlerart	Lernbedingung[a] B	E	EB	H-Test[b]
Fehlerhafte Abbruchbedingung (hätte von Beispielgruppe aus dem Beispiel übernommen werden können)	60	30	31	11.11, $p < .05$
keine oder falsche Minimierung (hätte von Beispielgruppe aus dem Beispiel übernommen werden können)	38	12	17	10.11, $p < .05$
falsche Verknüpfungsfunktion (von Beispielgruppe aus dem Beispiel übernommen)	62	42	39	04.22, n.s.
falsche Rückgabe im direkten Fall (von Beispielgruppe aus dem Beispiel übernommen)	55	48	38	03.96, n.s.

Anmerkungen.
In der Tabelle sind absolute Fehlersummen aufgeführt, die aus der Analyse der Aufgabenlösungen von jeweils 13 Personen pro Lernbedingung hervorgingen.
[a]: B= Beispielgruppe, E = Erklärungsgruppe, EB = Gruppe mit beiden Lehrmaterialien
[b]: Kruskal-Wallis H-Test

So wird etwa bei der Aufgabe zur Berechnung des Summenwertes (*summe*, siehe Kap. 7.4) von zahlreichen Probanden dieser Gruppe nicht erkannt, daß die Abbruchbedingung für Fakultäts- und Summenfunktion identisch ist (Parameter hat den Wert *0*). Dagegen wird die Verknüpfungsoperation übernommen und für den direkten Fall nicht erkannt, daß das neutrale Element für die Summenbildung null ist. Ebenso begehen vor allem die Probanden der Beispielgruppe den Fehler, daß sie auf die Minimierung des für den Rekursionsabbruch relevanten Parameters verzichten, obwohl die Minimierung im Beispiel immer analog zur Aufgabenstellung funktioniert. Eine ausführliche Darstellung der Fehleranalysen ist Schmid und Gräbener (1993) zu entnehmen.

Performanz beim Lernprozeß: Hilfeaufrufe. Um genauere Aufschlüsse über den Einfluß der Lernhilfen auf das Lernverhalten zu bekommen, wurde geprüft, wie häufig die Probanden jeder Lernbedingung die Hilfen pro Aufgabe nutzten und wie lange eine Hilfe im Schnitt gelesen wurde. Die Probanden forderten im Schnitt etwa *8* Hilfen an ($\bar{x} = 7.6$; $sd = 5.7$) und lasen eine Hilfe etwa *2 Minuten* lang ($\bar{x} = 114.36$ *Sekunden*; $sd = 133.42$). Die

Probanden zeigten eine hohe Variabilität in ihrem Hilfeaufrufverhalten. Betrachtet man die Probanden der drei Lernbedingungen getrennt, lesen die Probanden der Beispielgruppe ihre Hilfen häufiger und länger als die Probanden der beiden anderen Gruppen, die Unterschiede sind aber nicht signifikant (siehe Abb. 16).

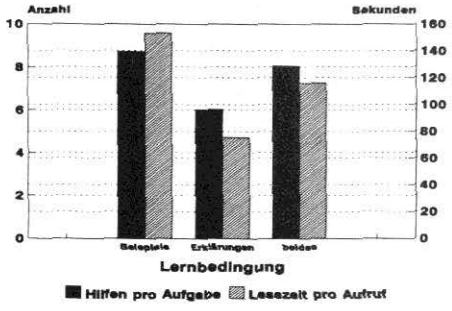


Abb. 16: Aufrufhäufigkeiten und Lesezeiten der Lernhilfen

Probanden, der Lernbedingung, die beide Hilfen zur Verfügung hatte, nutzten für alle 6 rekursiven Aufgaben beide Hilfen. Allerdings werden Erklärungen etwas häufiger gelesen als Beispiele (Erklärungen: $\bar{x} = 4.4$; $sd = 2.9$; Beispiele: $\bar{x} = 3.7$; $sd = 2.9$). Die Lesezeiten für Beispiele sind dagegen länger als für Erklärungen (Beispiele: $\bar{x} = 65.9$ *Sekunden;* $sd = 85.9$; Erklärungen: $\bar{x} = 50.4$ *Sekunden;* $sd = 25.9$). Die Unterschiede sind jedoch nicht signifikant.

Untersuchung I zeigte also, daß die entwickelte Untersuchungsanordnung dazu geeignet ist, Lernprozesse im Rahmen eines experimentellen Vorgehens zu erfassen. Zudem konnten die ersten allgemeinen Hypothesen zum Erwerb von rekursiven Programmiertechniken mit Beispielen (vgl. Kap. 6.1; siehe oben) bestätigt werden. Damit sind die Voraussetzungen für eine detailliertere Betrachtung des Wissenserwerbs mit Beispielen in Untersuchung II gegeben.

8.3 Untersuchung II: Der Einfluß von Beispielähnlichkeit auf induktive Lernprozesse beim rekursiven Programmieren

In Untersuchung II wird der Einfluß des Grades der strukturellen Ähnlichkeit von Beispiel und Aufgabe auf den Aufgabenlösungserfolg und den Wissenserwerb betrachet. Dabei wird die Beispielähnlichkeit systematisch stufenweise variiert. Das einzige mir bekannte Experiment, das ansatzweise Aussagen über den Einfluß der Beispielähnlichkeit auf den Wissenserwerb zuläßt, stammt von Reed und Bolstad (1991; vgl. Kap. 7.1). Allerdings prüfen die Autoren lediglich den Einfluß der Beispielähnlichkeit auf den Aufgabenlösungserfolg, nicht aber auf den resultierenden Wissenserwerb. Aus den Einzelfallstudien von Pirolli und Anderson (1985) ergeben sich Hinweise darauf, daß Beispielähnlichkeit die aufgebauten Wissensstrukturen beeinflußt: Hohe strukturelle Ähnlichkeit zwischen Beispiel und Aufgabe führt zur Induktion sehr spezifischer Wissensstrukturen so daß der Transfer auf anders strukturierte Aufgaben erschwert wird.

Zusätzlich wird die Wirkung der Teilprozesse *mapping* und *adaptation* getrennt erfaßt, indem einer Gruppe von Probanden die Ergebnisse des Mapping-Prozesses vorgegeben werden (vgl. Novick & Holyoak 1991; Kap. 7.1). Aufgrund der Befunde von Reed und Bolstad (1991) sowie von Novick und Holyoak (1991) gehe ich von der Annahme aus, daß eine hohe Beispielähnlichkeit sowie die Vorgabe der Ergebnisse des *Mapping*-Prozesses sich positiv auf den Aufgabenlösungserfolg auswirken (vgl. Kap. 7.1). Der resultierende Wissenserwerb wird in dieser Untersuchung detaillierter als in Untersuchung I und den in Kapitel 7.1 vorgestellten Experimenten zur Schemainferenz erfaßt. Das in Kapitel 6 vorgestellte Modell beschreibt den Wissenserwerb als Inferenz von Schemata mit daran gebundenen Produktionsregeln. Eine indirekte Erfassung der inferierten Schemata ist es, zu prüfen, wie gut Transferaufgaben gelöst werden können (vgl. Untersuchung I und Kap. 7.1). Andererseits wird von Schematheoretikern angenommen, daß Schemastrukturen die Klassifikation von Objekten nach den durch die Schemahierarchie vorgegebenen Merkmalen ermöglichen (vgl. Kap. 3.2). Werden Schemata entsprechend der in Kapitel 6.2 vorgestellten Hierarchie inferiert, sollten Probanden also in der Lage sein, Funktionen nach den durch die in der Hierarchie vorgegebenen Merkmalen zu klassifizieren. Um die beiden Herangehensweisen zur Erfassung der Schemainduktion zu unterscheiden, wird im folgenden von Transfer und Schemaerwerb gesprochen.

In der Untersuchung von Novick und Holyoak (1991) wurde die Schemainduktion durch Transferaufgaben und Abfragen der Ähnlichkeit zwischen Beispiel und Aufgabe erfaßt (vgl. Kap. 7.1). Die Autoren zeigten, daß ein hoher Zusammenhang zwischen dem Aufgabenlösungserfolg und der aus den Ähnlichkeitsangaben abgeleiteten Ratings der Qualität des inferierten Schemas besteht. Zudem ist der Zusammenhang zwischen der Schemaqualität und

der Lösung der Transferaufgabe höher als zwischen Aufgabenlösungserfolg und Transferleistung. Die Autoren berichten nicht über den Einfluß der *Mapping*-Vorgabe auf Schemaerwerb und Transfer. Man kann jedoch annehmen, daß die *Mapping*-Vorgabe sich positiv auf den Wissenserwerb auswirkt, da die Vorgabe zu einem höheren Aufgabenlösungserfolg führt und dieser in positivem Zusammenhang zu Schemaerwerb und Transfer steht.

Untersuchungsergebnisse zum Einfluß der Beispielähnlichkeit auf den Wissenserwerb liegen nicht vor. Aus den Analysen von Pirolli und Anderson (1985) sowie aus dem in Kapitel 6.3 vorgestellten Modell läßt sich die Hypothese ableiten, daß eine mittlere Ähnlichkeit von Beispiel und Aufgabe sich am positivsten auf den Wissenserwerb (Transfer und Schemaerwerb) auswirken sollte. Zu hohe strukturelle Ähnlichkeit führt zur Induktion zu spezifischer Schemata, so daß ein Transfer auf Aufgaben mit anderer rekursiver Struktur kaum erleichtert wird und die inferierten Merkmale zur Klassifikation rekursiver Funktionen mit unterschiedlichen Strukturen kaum geeignet sind. Große strukturelle Abweichungen zwischen Aufgabe und Beispiel verringern den Aufgabenlösungserfolg, so daß möglicherweise kaum Inferenzen vorliegen, die zum Aufbau einer Schemastruktur und zur Bildung von Produktionsregeln verwendet werden können.

Variablen und Hypothesen

Um den Einfluß der Beispielähnlichkeit auf den analogen Wissenserwerb zu prüfen, wurde die strukturelle Ähnlichkeit der Beispiele zur Aufgabe systematisch variiert. In das experimentelle Design gingen dabei *5* Ähnlichkeitsstufen ein (Faktor *Beispielähnlichkeit*: von *I*: sehr ähnlich bis *V*: sehr unähnlich). Zusätzlich wurde an einer kleinen Gruppe von *5* Personen geprüft, wie sich die Vorgabe von Beispielen mit vollständig abweichender rekursiver Struktur auf den Lernprozeß auswirkt. Der *Mapping*-Prozeß wurde den Probanden entweder abgenommen, indem zur Aufgabe strukturgleiche Teile im Beispiel fett markiert waren, oder die Probanden mußten das *Mapping* selber durchführen (Faktor *Mapping*: *0*: keine Vorgabe; *1*: Vorgabe der Ergebnisse des *Mapping*-Prozesses). Auf die Konstruktion der Beispiele wird noch genauer eingegangen.

Wie in Untersuchung I wird die Wirkung der Beispielbedingungen (Ähnlichkeit und Mapping) sowohl auf den Aufgabenlösungserfolg, als auch auf den Wissenserwerb geprüft. Der Aufgabenlösungserfolg wird dabei über die Anzahl gelöster Aufgaben erfaßt, wobei als Zeitlimit *20* Minuten pro Aufgabe gesetzt wurden. Die Anzahl gelöster Aufgaben sollte mit zunehmender Beispielähnlichkeit monoton ansteigen und eine Vorgabe der Ergebnisse des *Mapping*-Prozesses sollte sich ebenfalls positiv auf den Aufgabenlösungserfolg auswirken.

Der Wissenserwerb wird über die in einem Abschlußtest erhobene Transferleistung und zusätzlich über Sortieraufgaben erfaßt. Die Sortieraufgaben sind dabei als zuverlässigere Alternative zu der von Novick und Holyoak (1991) verwendeten Befragungsmethode zur direkteren Erfassung des Schemaerwerbs gedacht. Sortieraufgaben sind aus zwei Gründen als eine empirisch zuverlässigere Herangehensweise zur Erhebung von repräsentierten Merkmalsstrukturen anzusehen als die direkte Erfragung der verwendeten Merkmale. Zum einen sind Personen häufig nicht in der Lage, die Merkmale zu benennen, nach denen sie Objekte klassifizieren, zum anderen wird das Problem der Beurteilung freier verbaler Äußerungen durch Rater umgangen. Aus Gruppierungen von Objekten nach ihrer Ähnlichkeit lassen sich mithilfe mathematischer Methoden quantitative Skalen erzeugen, die die Distanzen der Objekte abbilden (vgl. Sixtl 1967). Zudem ermöglichen verschiedene deskriptive Verfahren, wie Clusteranalysen (Eckes & Roßbach 1980) und multidimensionale Skalierung (Ahrens 1974), eine Ermittlung der Anzahl von zur Klassifikation verwendeten Merkmalen. Durch zusätzliche Interpretation der Ausprägung der Objekte auf diesen Merkmalsdimensionen können dann die inhaltliche Beschreibungen der Merkmale ermittelt werden.

Die Probanden erhalten zu drei Meßzeitpunkten eine Menge von Funktionen vorgegeben, die sie nach ihrer Ähnlichkeit sortieren sollen. Dabei wurden insgesamt 24 Funktionen, die in der Syntax der verwendeten funktionalen Sprache kodiert waren, vorgegeben. *16* dieser Funktionen waren rekursiv, *8* nicht-rekursiv. Dabei waren die rekursiven Funktionen so konstruiert, daß jeweils zwei an ein Blatt der in Kapitel 6.2 vorgestellten Schemahierarchie gebunden werden können und damit nach den Merkmalen Rekursionstyp, Anzahl der Abbruchbedingungen und verwendete Datentypen beschrieben werden konnten. Die nicht rekursiven Funktionen waren in Paaren bezüglich der Merkmale Anzahl der Abbruchbedingungen und Datentyp variiert. Aus den Sortierungen wurden zwei abhängige Variablen errechnet. Zum einen wurde die Distanz der Sortierungen zur in Kapitel 6.2 vorgeschlagenen Expertenhierarchie errechnet und zum anderen wurden die bei der Sortierung verwendeten Merkmale erschlossen. Unabhängig von den Beispielbedingungen sollte die Distanz zur Expertenhierarchie über die drei Sortierdurchgänge abnehmen und zunehmend häufiger das Merkmal "Rekursionstyp" (Endrekursion, Teil-Rest-Rekursion) zur Klassifikation der Funktionen verwendet werden. Die Konstruktion dieser beiden Kennwerte wird weiter unten ausführlich dargestellt.

Zum Einfluß der Beispielbedingungen Ähnlichkeit und Mapping auf den Wissenserwerb werden folgende Hypothesen aufgestellt: Die durch den Abschlußtest ermittelte Transferleistung sollte bei mittlerer Beispielähnlichkeit (Stufe *III*) am höchsten sein und für ähnlichere bzw. unähnlichere Beispiele abnehmen. Die induzierte Schemahierarchie sollte sich damit am Ende der Lernphase bei Vorgabe von Beispielen mittlerer Ähnlichkeit am stärksten in Richtung der postulierten Expertenhierarchie entwickelt haben. Damit sollte die

Distanz zur Expertenhierarchie für Probanden, die Beispiele mittlerer Ähnlichkeit erhalten haben, am geringsten sein und für ähnlichere bzw. unähnlichere Beispiele zunehmen. Desgleichen sollten Probanden der Beispielgruppe *III* häufiger das Merkmal Rekursionstyp zur Klassifikation von Funktionen verwenden, als Probanden der anderen Beispielbedingungen.

Um die Befunde von Novick und Holyoak (1991) zum Zusammenhang von *Mapping*-Vorgabe und Wissenserwerb zu replizieren, müßte sich zeigen, daß Probanden mit Vorgabe der Ergebnisse des *Mapping*-Prozesses in allen Kennwerten zur Erfassung des Wissenserwerbs günstigere Ergebnisse erzielen, als Probanden, die den *Mapping*-Prozeß selbst leisten mußten.

Schließlich werden, wie bei Novick und Holyoak (1991), die Zusammenhänge zwischen Aufgabenlösungserfolg, Transfer und Schemainduktion erfaßt.

Methode

Stichprobe. An der Untersuchung nahmen *55* Psychologiestudenten der TU, FU und HU Berlin teil. Davon waren *28* Personen weiblichen und *27* Personen männlichen Geschlechts. Die Personen waren zwischen *19* und *43* Jahren alt, mit einem mittleren Alter von *26* Jahren. Sie befanden sich im ersten bis dreizehnten Semester, im Mittel im zweiten Semester. *40* Personen hatten Computererfahrung (Textverarbeitung, Statistikpakete), *2* dieser Personen hatten rudimentäre Programmiererfahrungen mit imperativen Sprachen (BASIC bzw. PASCAL). Die Probanden wurden zufällig einer der *5 x 2* Lernbedingungen zugeteilt (siehe Tabelle 8).

TABELLE 8
Verteilung der Stichprobe auf die Zellen

		Beispielähnlichkeit					
		I	II	III	IV	V	Σ
Map-	0	5	5	5	6	6	27
ping	1	5	7	6	5	5	28
Σ		10	12	11	11	11	55

Zusätzlich wurden *5* Physikstudenten der HU Berlin untersucht. Sie erhielten Beispiele der Ähnlichkeitsstufe *V* (*3* Personen) beziehungsweise Beispiele mit vollständig abweichender rekursiver Struktur (Ähnlichkeit *VI*, *2* Personen). Die Beispiele wurden grundsätzlich ohne

Vorgabe der Ergebnisse des *Mapping*-Prozesses präsentiert. Auf die Analyse dieser Probandengruppe wird im folgenden nicht eingegangen (siehe Schmid & Kaup 1993).

Material und Durchführung. Die verwendeten Instruktions- und Testmaterialien, sowie der Ablauf der Untersuchung wurden bereits in Kapitel 8.1 dargestellt. Als **Lehrmaterial** wurde jedem Proband pro Aufgabe ein Beispiel angeboten, das während der Bearbeitung der Aufgabe durchweg zur Verfügung stand. Dabei waren für einen Probanden die gleichbleibenden Strukturen von Beispiel und Aufgabe entweder fett markiert (*Mapping=1*) oder nicht (*Mapping=0*) und die Beispielstruktur war der Aufgabenstruktur mehr oder weniger ähnlich (Beispielähnlichkeit *I..V*). Auf die Konstruktion der Beispiele mit unterschiedlicher Ähnlichkeit wird im folgenden genauer eingegangen, alle Beispiele sind Anhang D.2 zu entnehmen.

Für die durch **Sortierversuche** ermittelte Schemainferenz wurden zwei abhängige Variablen konstruiert. Auf die Sortierversuche wird weiter unten ebenfalls noch genauer eingegangen.

Der **Abschlußtest** aus Untersuchung I wurde um eine Transferaufgabe erweitert: Die Probanden hatten zusätzlich eine simultanrekursive Aufgabe zu programmieren. Da die Probanden während der Lernphase nicht mit Simultanrekursion konfrontiert waren, wird mit dieser Aufgabe geprüft, inwieweit die erworbene Schemastruktur Generalisierungen auf neue rekursive Strukturen ermöglicht.

Konstruktion der Beispiele. Die unterschiedliche Beispielähnlichkeit wird durch Anzahl und Schwierigkeit der notwendigen Transformationen im Bezug auf die zu lösende Aufgabe realisiert. In der ersten Ähnlichkeitsstufe wurde die Struktur der Aufgabe vollständig beibehalten und lediglich Namen, Konstanten oder vordefinierte Operationen durch Ausdrücke gleichen Typs ersetzt (vgl. Untersuchung I). In der zweiten Ähnlichkeitsstufe wurde entweder die Struktur der Abbruchbedingung oder des direkten Falls um eine Komplexitätsstufe erweitert, in dem dem entsprechenden Ausdruck eine Operation hinzugefügt wurde. In der dritten Ähnlichkeitsstufe wurde der rekursive Aufruf um eine Operation erweitert. Damit wird der für die Struktur rekursiver Funktionen zentrale Ausdruck verändert. Die vierte Ähnlichkeitsstufe ist eine Kombination der Stufen *II* und *III*. In der fünften Ähnlichkeitsstufe wurde das Beispiel der Stufe *IV* um eine zusätzliche Operation erweitert. Diese Operation wurde bei dem in der zweiten Ähnlichkeitsstufe nicht manipulierten Ausdruck eingefügt.

In einer sechsten Ähnlichkeitsstufe, die nicht in das experimentelle Design eingeht, sondern einer kleinen Personengruppe zu explorativen Zwecken präsentiert wurde, wurde die rekursive Struktur verändert: Für endrekursive Aufgaben wurde ein teil-rest-rekursive Beispiele gegeben und umgekehrt. Tabelle 9 zeigt exemplarisch alle Beispielähnlichkeiten für die Aufgabe *summe*. Eine Übersicht über alle konstruierten Beispiele gibt Anhang D.2.

Sortierversuche. Durch die mehrmalige Durchführung der Sortierversuche soll erfaßt werden, ob die Probanden, so wie in dem in Kapitel 6 vorgestellten Modell postuliert, Schemata über rekursive Funktionen aufbauen. Die in Kapitel 6.2 vorgeschlagene Expertenhierarchie wurde um einen Zweig für nicht-rekursive Funktionen ergänzt. Mittels der durch die Hierarchie vorgegebenen Merkmalsstruktur wurden dann 28 Funktionen konstruiert, wobei jeweils zwei Funktionen demselben Blatt der Hierarchie zuordenbar waren. Eine Übersicht über die Funktionen gibt Anhang D.4.

Die Probanden wurden instruiert, die Funktionen nach ihrer Ähnlichkeit zu sortieren, wobei ihnen freigestellt war, linear oder hierarchisch zu strukturieren. Sie wurden ebenfalls darauf hingewiesen, daß es keine "richtige" oder "falsche" Sortierung gäbe. In jedem Durchgang hatten die Probanden *30* Minuten Zeit zur Lösung der Sortieraufgabe. Ein erster Sortierdurchgang vor der experimentellen Phase diente als Ausgangslage für die beiden folgenden Sortierungen, die nach Bearbeitung der dritten bzw. vierten Lektion erhoben wurden.

Die Sortierungen der *28* Funktionen wurden erstens bezüglich der Distanz zur Expertenhierarchie und zweitens hinsichtlich der verwendeten Sortiermerkmale analysiert. Die in Kapitel 6.2 vorgestellte Expertenhierarchie rekursiver Funktionen definiert eine Merkmalsstruktur, bei der die Relevanz der verschiedenen Merkmale durch die Hierarchieebene festgelegt ist: Je höher ein Merkmal in der Hierarchie erscheint, desto wichtiger ist es. Die beiden entwickelten Maße zur Analyse der Sortierungen erfassen die Schemainduktion unterschiedlich streng. Die Distanz zur Expertenhierarchie gibt an, ob und inwieweit beim analogen Lernen bereits eine hierarchische Struktur von Schemata inferiert wurde. Erstens wird hier erfaßt, ob und in welchem Ausmaß die für den Experten postulierten Merkmale zur Klassifikation von Funktionen verwendet werden. Zweitens wird erfaßt, ob die durch die Expertenhierarchie definierte unterschiedliche Relevanz der Merkmale sich bereits in der Repräsentation entwickelt hat.

TABELLE 9
Konstruktion der Beispiele (hier mit Mapping-Vorgabe)

```
Aufgabe:
Schreibe eine rekursive Funktion summe,     FUN summe(x:NAT):NAT
die den Summenwert einer Zahl x mit all     IF EQUAL(x,0)
ihren Vorgängern bis 0 berechnet.           THEN 0
                                            ELSE PLUS(x,summe(MINUS(x,1)))
Beispiel: x=5 ergibt 5+4+3+2+1+0=15         FIN
5+4+3+2+1+0=15

Beispiel I: Änderung von Konstanten oder vordefinierten Operationen
Die Funktion bsp berechnet die Summe  FUN bsp(x:NAT):NAT
einer Zahl mit all ihren Vorgängern   IF EQUAL(x,2)
ohne 2 und 1.                         THEN 0
                                      ELSE PLUS(x,bsp(MINUS(x,1)))
Beispiel: x=5 ergibt 5+4+3=12         FIN

Beispiel II: Erweiterung von Abbruchbedingung oder direktem Fall
Die Funktion bsp berechnet die Summe  FUN bsp(x:NAT):NAT
einer Zahl mit all ihren Vorgängern   IF EQUAL(MINUS(x,1),0)
ohne 1.                               THEN 0
                                      ELSE PLUS(x,bsp(MINUS(x,1)))
Beispiel: x=5 ergibt 5+4+3+2=14       FIN

Beispiel III: Erweiterung des rekursiven Aufrufs
Die Funktion bsp berechnet die Summe  FUN bsp(x:NAT):NAT
einer quadrierten Zahl mit all ihren  IF EQUAL(x,0)
quadrierten Vorgängern.               THEN 0
                                      ELSE PLUS(MULT(x,x),
Beispiel: x=5 ergibt                     bsp(MINUS(x,1)))
5*5+4*4+3*3+2*2+1*1+0=55              FIN

Beispiel IV: II+III
Die Funktion bsp berechnet die Summe  FUN bsp(x:NAT):NAT
einer quadrierten Zahl mit all ihren  IF EQUAL(MINUS(x,1),0)
quadrierten Vorgängern ohne 1.        THEN 0
                                      ELSE PLUS(MULT(x,x),
Beispiel: x=5 ergibt                     bsp(MINUS(x,1)))
5*5+4*4+3*3+2*2=54                    FIN

Beispiel V: IV+
        zusätzliche Erweiterung von Abbruchbedingung oder direktem Fall
Die Funktion bsp berechnet die Summe  FUN bsp(x:NAT):NAT
einer quadrierten Zahl mit all ihren  IF EQUAL(MINUS(x,1),0)
quadrierten Vorgängern ohne 1.        THEN MINUS(x,1)
                                      ELSE PLUS(MULT(x,x),
Beispiel: x=5 ergibt                     bsp(MINUS(x,1)))
5*5+4*4+3*3+2*2=54                    FIN

Beispiel VI: Änderung des Rekursionstyps (nicht im Experiment verwendet)
Die Funktion bsp prüft, ob eine Zahl  FUN bsp(x:NAT):NAT
gerade oder ungerade ist. Ist die     IF GREATER(2,x)
Zahl gerade, liefert sie 0, sonst 1.  THEN x
                                      ELSE bsp(MINUS(x,2))
Beispiel: x=5 ergibt 5-2-2=1          FIN
```

Da die Zeit, in der sich die Probanden in der Untersuchung mit rekursiven Programmiertechniken auseinandersetzen, relativ gering ist, ist nicht anzunehmen, daß sie am Ende der Untersuchung bereits eine der Expertenhierarchie ähnliche Repräsentation rekursiver Funktionen erworben haben. In dem in Kapitel 6.3 dargestellten Modell des Wissenserwerbs wird die Annahme formuliert, daß sich die Hierarchie "von unten nach oben" differenziert. Das heißt, daß an das unvollständige allgemeine Rekursionsschema zunächst Schemata gebunden werden, die relativ speziell sind und Merkmale wie Datentyp und Anzahl von Abbruchbedingungen enthalten. Eine Abstraktion zu einem allgemeinen End- bzw. Teil-Rest-Rekursions-Schema soll sich entsprechend der Modellannahmen erst entwickeln, wenn die Probanden bereits Erfahrung mit rekursiven Funktionen der entsprechenden Klasse hatten, die sich bezüglich dieser Merkmale unterscheiden.

Entsprechend wird zusätzlich zur Distanz zur Expertenhierarchie ein zweites, wesentlich schwächeres Maß verwendet, bei dem lediglich geprüft wird, welche Merkmale die Probanden zur Klassifikation der Funktionen verwenden.

Distanz zur Expertenhierarchie. Die Distanz zur Expertenhierarchie wird ermittelt, in dem alle Sortierungen in eine Distanzmatrix über die *28* Funktionen überführt werden. Die Distanz zwischen je zwei Funktionen wird dabei über die Anzahl der Kanten ermittelt, die durchlaufen werden müssen, um von einer Funktion zur anderen zu gelangen. Die Kantenzahl wird am längsten Weg im Baum relativiert, um für alle Sortierungen vergleichbare Distanzmaße zu erhalten. Entsprechend wird für die Expertensortierung und die Sortierungen der Probanden aller drei Durchgänge jeweils eine symmetrische *28 x 28* Matrix ermittelt, deren Zellen Distanzen zwischen je zwei Funktionen enthalten. Zwei Funktionen haben eine minimale Distanz von null zueinander, wenn sie sich im selben Blatt befinden und eine maximale Distanz von eins, wenn ihre Entfernung in der Hierarchie dem längsten Weg in der Hierarchie entspricht.

Um diese Analyse durchzuführen, müssen alle Sortierungen in eine Baumstruktur überführt werden. Bei Probanden, die eine hierarchische Sortierung wählten, wurden diese Hierarchien verwendet. Für Probanden, die die Funktionen lediglich in mehrere Gruppen einteilten, wurde eine Hierarchie gebildet, die lediglich aus der Wurzel und den entsprechenden Gruppen als Blätter bestand. Die Distanzen zwischen Funktionen können für solche Sortierungen also nur die Extremwerte null (Funktionen in einem Blatt) und eins (Funktionen in verschiedenen Blättern) annehmen.

Als abhängige Variable "Distanz zur Expertenhierarchie" wurde die euklidische Distanz zwischen jeder Probanden-Matrix und der Experten-Matrix verwendet.

Erfassung der verwendeten Sortiermerkmale. Um zu erfassen, welche Merkmale die Probanden zur Klassifikation der Funktionen verwendeten, wurde ein Verfahren entwickelt, daß reziprok zu Methoden des *conceptual clustering* (vgl. Michalski & Stepp 1983; Kap.

4.1) funktioniert. Jede der *28* Funktionen wurde mit einem Merkmalsvektor beschrieben. Dabei wurden die durch die Expertenhierarchie vorgegebenen Merkmale verwendet und zusätzlich das Merkmal "Typ der boolschen Anweisung" eingeführt, da sich zeigte, daß dieses Merkmal von einigen Probanden verwendet wurde. Der Merkmalsvektor zur Beschreibung der Funktionen wurde als Fünf-Tupel mit folgenden Ausprägungen definiert:

- rekursive Funktion: 1=nein, 2=ja
- Rekursionstyp: 0 für nicht-rekursive Funktionen
  1=Endrekursion
  2=Teil-Rest-Rekursion
  3= Simultan-Rekursion
  4= End- und Teil-Rest-Rekursion
- Anzahl der Abbruchbedingungen: 1, 2
- Datentyp: 1= nur NAT, 2= NATLIST
- boolsche Anweisung: 1=equal, 2=greater, 3=empty, 4= NOT(...)

Auf eine feinere Unterteilung der Teil-Rest-Rekursion, so wie sie in der Expertenhierarchie vorgenommen wurde, wurde verzichtet, da diese Diskrimination von keinem Probanden vorgenommen wurde.

Die von den Probanden gebildeten Funktionsgruppen, also die Blätter der gebildeten Hierarchie, wurden dahingehend geprüft, welche Merkmale alle Funktionen einer Gruppe gemeinsam haben. Um ein vergleichbares Maß über alle Probanden zu erhalten, wurde für jedes Merkmal die Anzahl der Funktionsgruppen, in denen dieses Merkmal verwendet wurde an der Anzahl aller gebildeten Gruppen relativiert. Für das Merkmal Rekursionstyp wurde zur Relativierung die Anzahl aller gebildeten Gruppen verwendet, in denen nur rekursive Funktionen enthalten sind.

Ergebnisse und Interpretation

Zunächst wird ein kurzer Überblick über die Ausprägung der erhobenen Variablen in der Stichprobe gegeben.

Weniger als ein Drittel der Probanden (*29.09%*) hatte Mathematik in der gymnasialen Oberstufe als Schwerpunktfach belegt. Die letzte Mathematiknote ($AM = 2.65$, $sd = 1.12$), sowie die Selbsteinschätzung der Begabung ($AM = 2.82$, $sd = 0.82$; auf einer *5*-stufigen Skala von *1*=gar nicht begabt bis *5*=sehr begabt) liegen im mittleren Bereich.

Im Syntaxtest erzielten die Probanden im Schnitt *16.36* von *28* Punkten ($sd = 5.58$). Damit verfügen die Probanden über ausreichende Syntaxkenntnisse zur Bewältigung der nachfolgenden rekursiven Aufgaben.

Von den sechs Zielaufgaben, die nur mit Unterstützung der gemäß den Lernbedingungen vorgegebenen Beispielen bearbeitet wurden, lösten die Probanden fast *70* Prozent (*AM = 4.14, sd = 1.95*).

Die im Abschlußtest gemessene Transferleistung liegt knapp unter der Hälfte der maximal erreichbaren *22* Rohwertpunkte (*AM = 10.64, sd = 4.15*).

Die Kontrollvariable Syntaxtest ist über die Lernbedingungen (Beispielähnlichkeit x Mapping) hinweg leider nicht vergleichbar. Sowohl der Haupteffekt Beispielähnlichkeit, als auch die Interaktion von Beispielähnlichkeit und Mapping werden bei einer Irrtumswahrscheinlichkeit von *25%* signifikant (*n = 55*; Mapping: *F = 0.004, p = 0.95*; Beispielähnlichkeit: *F = 2.04, p = 0.11*; Mapping x Beispielähnlichkeit: *F = 2.13, p = 0.09*). Am deutlichsten unterscheiden sich die Lernbedingungen der Beispielähnlichkeiten *II* und *III* bezüglich der Mapping-Variation (siehe Abb. 17). Aus diesem Grund wird der Syntaxtest bei den folgenden statistischen Analysen als Kovariate mitberücksichtigt.

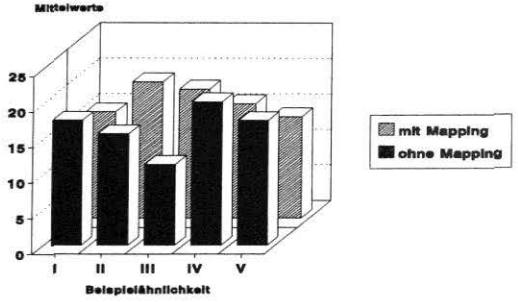


Abb. 17: Vergleichbarkeit der Syntaxtestergebnisse über die Lernbedingungen

Aufgabenlösungserfolg. Die Lernbedingungen beeinflussen die Anzahl gelöster Aufgaben signifikant (*n = 49, 6* missing values; Mapping: *F = 5.98, p = 0.02*; Beispielähnlichkeit: *F = 4.00, p = 0.01*; Mapping x Beispielähnlichkeit: *F = 1.5, p = 0.22*; Kovariate Syntaxtest: *F = 21.48, p < 0.0001*). Die Vorgabe der Ergebnisse des *Mapping*-Prozesses, sowie hohe Beispielähnlichkeit führen zu einer erhöhten Anzahl gelöster Aufgaben (siehe Abb. 18).

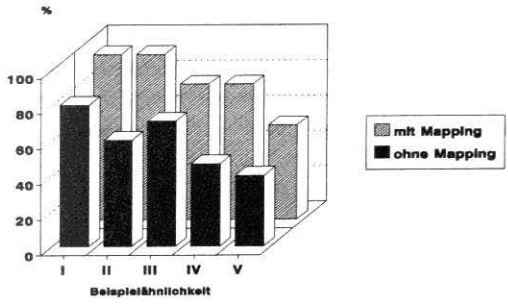


Abb. 18: Einfluß der Lenbedingung auf die Anzahl gelöster Aufgaben

Wissenserwerb I: Lerntransfer. Um eine zuverlässige Skala zur Beschreibung des Lerntransfers zu erhalten, wurde der Abschlußtest raschskaliert. Nach Ausschluß eines Probanden der Beispielbedingung *II*, ohne Mapping, konnte das Modell angepaßt werden ($n = 59$; Itemmodell: $chi^2 = 402.63$, $df = 378$, $z = 0.90$; Personenmodell: $chi^2 = 1042.96$, $df = 986$, $z = 1.27$). Die Werte des Personenmodells liefern den Kennwert für die Transferleistung der Probanden. Alle statistischen Analysen wurden mit den Raschskalenwerten berechnet.

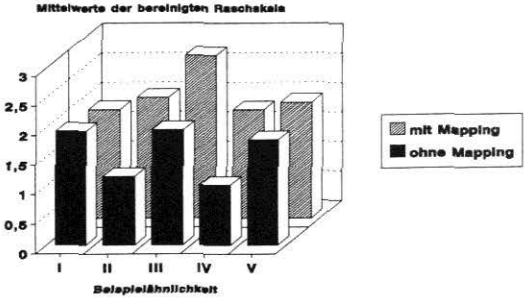
Zusätzlich zu den raschskalierten Werten des Gesamttests wurden auch die Rohwertsummen (vom Syntaxtest bereinigt) der vier Untertestgruppen des Abschlußtests betrachtet. Die Korrelationen der Untertests mit der Gesamtskala sind durchweg signifikant. Die Interkorrelationen der Untertests sind ebenfalls signifikant, weisen jedoch geringere Koeffizienten auf (siehe Tab. 10).

TABELLE 10
Korrelation der Itemgruppen des Abschlußtests mit dem Gesamttestwert und Interkorrelationen

	Wissen	*Semantik*	E/A	Programm.
Gesamttest	0.648 p<0.0001	0.781 p<0.0001	0.871 p<0.0001	0.670 p<0.0001
Wissen		0.316 p=0.02	0.341 p=0.01	0.326 p=0.02
Semantik			0.545 p<0.0001	0.434 p=0.001
E/A				0.606 p<0.0001

Anmerkungen.
n=55 bzw. bei Korrelation mit Gesamttest n=54
Gesamttest: Personenskala des Raschmodells.
Wissen, Semantik, Ein-/Ausgabe, Programm(ierung): Rohwertsummen der Itemgruppen.

Die Lernbedingungen haben keinen bedeutsamen Einfluß auf den Lerntransfer ($n = 54$; Mapping: $F = 1.9$, $p = 0.18$; Beispielähnlichkeit: $F = 0.45$, $p = 0.77$; Mapping x Beispielähnlichkeit: $F = 0.35$, $p = 0.84$; Kovariate Syntaxtest: $F = 18.346$, $p < 0.0001$). Tendenziell zeigen jedoch Probanden, denen die Ergebnisse des *Mapping*-Prozesses vorgegeben wurden, eine bessere Transferleistung (siehe Abb. 19).



Anmerkung.
Kennwert = Fähigkeitsskala des raschskalierten Abschlußtests (vom Syntaxtest bereinigt und auf positiven Bereich tranformiert).

Abb. 19: Einfluß der Lernbedingungen auf den Transfer

Dieser Eindruck wird durch signifikant bessere Leistungen dieser Probandengruppe bei der Beantwortung der Wissensitems (Rangvarianzanalyse: $n = 54$, $chi^2 = 5.1$, $p = 0.024$) sowie der Lösung der Programmieraufgaben ($chi^2 = 8.5$, $p = 0.004$) bestätigt (siehe Abb. 20 und Abb. 21).

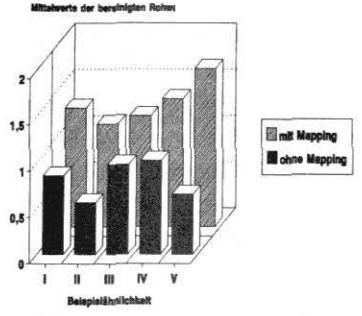


Abb. 20: Einfluß der Lernbedingung auf das erworbene abstrakte Wissen

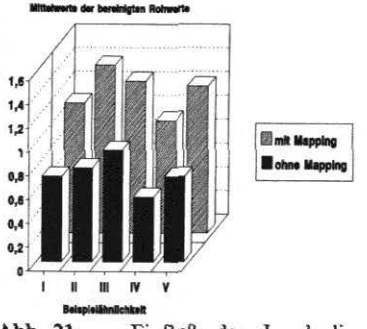


Abb. 21: Einfluß der Lernbedingung auf den Transfer (Programmierung)

Wissenserwerb II: Schemaerwerb. Die Distanz zur Expertenhierarchie ist im ersten Durchgang am höchsten ($n = 54$, $AM = 8.16$), im zweiten Sortierdurchgang am geringsten ($n = 53$, $AM = 7.66$) und steigt beim dritten Durchgang wieder an ($n = 52$, $AM = 8.04$). Die Distanzen zur Expertenhierarchie interkorrelieren über alle drei Sortierdurchgänge signifikant, wobei die jeweils benachbarten Durchgänge höher korrelieren (siehe Tab. 11).

TABELLE 11
Interkorrelationen Distanzen zur Normhierarchie

	Durchg.1	Durchg.2
Durchg.2	.529 (n=53) p<.0001	
Durchg.3	.282 (n=52) p=.021	.68 (n=52) p<.0001

Leider ergab sich für den ersten Sortierdurchgang, der vor der experimentellen Bedingungsvariation durchgeführt wurde, ein bedeutsamer Unterschied für die Beispielbedingungen ($n = 54$; Mapping: $F = 10.11$, $p = 0.003$; Beispielähnlichkeit: $F = 2.44$, $p = 0.06$; Mapping x Beispielähnlichkeit: $F = 1.87$, $p = 0.13$; Kovariate Syntaxtest: $F = 16.46$, $p < 0.0001$).

Wird für die beiden folgenden Sortierversuche zusätzlich zum Syntaxttest die Distanz zur Expertenhierarchie im ersten Durchgang als Kovariate verwendet, so ergeben sich keine signifikanten Unterschiede zwischen den Beispielbedingungen ($n = 53$; Durchgang 2: Mapping: $F = 2.23$, $p = 0.14$; Beispielähnlichkeit: $F = 0.31$, $p = 0.87$; Mapping x Beispielähnlichkeit: $F = 0.57$, $p = 0.69$; Kovariate Syntaxtest: $F = 7.69$, $p = 0.01$; Kovariate Distanz bei Durchgang 1: $F = 3.28$, $p = 0.08$; Durchgang 3: Mapping: $F = 0.98$, $p = 0.33$; Beispielähnlichkeit: $F = 0.47$, $p = 0.76$; Mapping x Beispielähnlichkeit: $F = 0.57$, $p = 0.69$; Kovariate Syntaxtest: $F = 10.39$, $p = 0.003$; Kovariate Distanz bei Durchgang 1: $F = 0.11$, $p = 0.74$).

Generell ist über alle drei Sortierdurchgänge festzustellen, daß Probanden ohne *Mapping*-Vorgabe tendenziell ähnlicher zur Expertenhierarchie sortieren. Abbildung 22 zeigt dies exemplarisch für Durchgang 2.

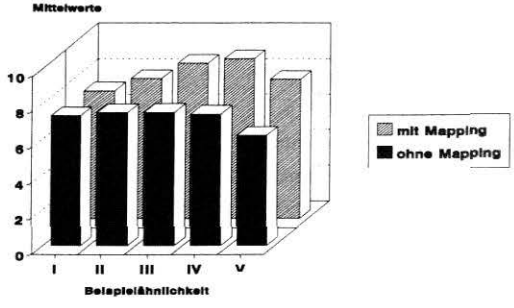


Abb. 22: Einfluß der Lernbedingung auf den Schemaerwerb (Distanz z. Expertenhierarchie)

Die Probanden verwendeten insgesamt fünf Merkmale zur Sortierung der *28* Funktionen (siehe Tab. 12).

TABELLE 12
Verwendete Sortiermerkmale

	Rang	Durchg.1	Durchg.2	Durchg.3
Rekursion ja/nein	3	Mod = 1 Md = .66 AM = .62	Mod = 1 Md = .75 AM = .66	Mod = 1 Md = .55 AM = .59
Rekursionstyp	5	Mod = 0 Md = 0 AM = .12	Mod = 0 Md = 0 AM = .10	Mod = 0 Md = 0 AM = .10
Anzahl der Abbruchbed.	2	Mod = 1 Md = .69 AM = .65	Mod = 1 Md = .75 AM = .70	Mod = 1 Md = .70 AM = .63
Datentypen	1	Mod = 1 Md = 1 AM = .78	Mod = 1 Md = .89 AM = .75	Mod = 1 Md = .89 AM = .71
boolsche Anweisung	4	Mod = 1 Md = .63 AM = .59	Mod = 1 Md = .67 AM = .62	Mod = 1 Md = .57 AM = .55

Anmerkung. Die Maße der zentralen Tendenz sind über die relativen Häufigkeiten der Verwendung der Merkmale berechnet.

Vier dieser Merkmale sind auch der Expertenhierarchie zugrundegelegt. Während in Kapitel 6.2 jedoch angenommen wird, daß Experten rekursive Funktionen vor allem nach dem Merkmal Rekursionstyp beurteilen, wird dieses Kriterium nur von einer geringen Probandenzahl verwendet. Die Probanden orientieren sich beim Vergleich der Funktionen eher an Oberflächenmerkmalen, wie Datentypen und Anzahl von Abbruchbedingungen. Bei einer nur viertägigen Lernzeit ist jedoch auch keine Expertise im Umgang mit rekursiven Funktionen zu erwarten. Daß Anfänger eher nach Oberflächenmerkmalen klassifizieren, ist ein gut belegter Befund (z.B. Adelson 1981; Novick 1988).

Tendenziell verwendeten Probanden ohne Vorgabe des *Mapping*-Prozesses das Merkmal Rekursionstyp häufiger als Probanden mit Mapping-Vorgabe. Probanden dieser Gruppe sortierten auch ähnlicher zur Expertenhierarchie. Deutlich wird dieser Unterschied für Sortierdurchgang 2, hier verwendeten acht Probanden ohne Mapping-Vorgabe und zwei Probanden mit Mapping-Vorgabe das Merkmal Rekursionstyp (die Wahrscheinlichkeit, daß die Zufallsvariable Rekursionstyp in der Binomialverteilung für $p = 10/54$ einen Wert größer gleich *8* annimmt beträgt *0.02* und daß sie einen Wert kleiner gleich *2* annimmt beträgt *0.11*). In Durchgang *3* verringert sich der Unterschied zu fünf Probanden ohne und vier Probanden mit Mapping-Vorgabe.

Zusammenhänge zwischen Programmierfertigkeit und Wissenserwerb. Während sich ein bedeutsamer Einfluß der Beispielbedingungen auf die Anzahl gelöster Aufgaben zeigt, kann ein solcher Einfluß für die Maße des Wissenserwerbs nur teilweise nachgewiesen werden. Im folgenden werden Zusammenhänge zwischen Aufgabenlösung und erworbenem Wissen unabhängig von den Beispielbedingungen betrachtet.

Zwischen der Anzahl gelöster Aufgaben und dem Lerntransfer, sowie den Itemgruppen Semantik und Programmierung, bestehen signifikante Zusammenhänge. Die Distanz zur Expertenhierarchie im dritten Sortierdurchgang steht ebenfalls in signifikanten Zusammenhang mit der Aufgabenlösung (siehe Tab. 13).

TABELLE 13
Zusammenhang von Aufgabenlösung und Lerntransfer

	Abschlußtst	Wissen	Semantik	E/A	Programm.
Aufgabenlösung	.408 (n=48) p=0.004	.267 (n=49) p=0.064	.444 (n=49) p=0.001	.280 (n=49) p=0.051	.319 (n=49) p=0.026

	Durchg.1	Durchg.2	Durchg.3
Aufgabenlösung	-0.041 (n=48) p=0.779	-0.109 (n=48) p=0.463	-0.331 (n=48) p=0.021

Zwischen den beiden Maßen des Wissenserwerbs - Lerntransfer und Nähe zur Expertensortierung - bestehen dagegen keine bedeutsamen Zusammenhänge. Es zeigt sich jedoch, daß Probanden, die im zweiten Durchgang das Merkmal "Rekursionstyp" zur Sortierung der Funktionen verwendeten, signifikant höheren Lerntransfer aufwiesen, als Probanden, die dieses Merkmal nicht verwendeten (Rekursionstyp verwendet: $n = 12$, $AM = 0.81$ vs. nicht verwendet: $n = 41$, $AM = -0.59$; die Mittelwerte wurden über die Raschskala berechnet; Rangvarianzanalyse: $chi^2 = 5.79$, $p = 0.016$).

In Untersuchung II konnten die Befunde von Reed und Bolstad (1991) zum Einfluß der Beispielähnlichkeit auf den Lösungserfolg von einfachen Mathematikaufgaben im Bereich rekursives Programmieren repliziert werden. Je größer die strukturelle Ähnlichkeit von Aufgabe und Beispiel ist, also je weniger Tranformationen bei der Adaptation des Beispiels an die Aufgabe notwenig sind, desto größer ist die Wahrscheinlichkeit, daß die Aufgabe bewältigt wird. Ebenso konnten die Befunde von Novick und Holyoak (1991) zum Einfluß der Vorgabe der Ergebnisse des *Mapping*-Prozesses auf den Aufgabenlösungserfolg repliziert werden. Werden die Ergebnisse des *Mapping*-Prozesses, also die Identifikation der struktur-

gleichen Anteile in Aufgabe und Beispiel, vorgegeben, so erhöht sich die Lösungswahrscheinlichkeit für die Aufgaben.

Bei der Prüfung des Einflusses der Beispielbedingungen auf den Wissenserwerb sprechen die Ergebnisse weniger deutlich für die eingangs aufgestellten Hypothesen. Für die Vorgabe der Ergebnisse des *Mapping*-Prozesses ließ sich ein signifikanter Einfluß auf den Lösungserfolg der Transferaufgaben nachweisen. Dies bestätigt die aus den Befunden von Novick und Holyoak (1991) abgeleitete Hypothese, daß *Mapping*-Vorgabe den Schemaerwerb positiv beeinflußt. Für die direktere Erfassung des Schemaerwerbs mittels der Sortierversuche zeigt sich jedoch ein umgekehrter Trend: Probanden, die den Vergleich von Aufgabe und Beispiel selbst durchführen mußten, haben in der Tendenz eine der Expertenhierarchie ähnlichere Schemastruktur erworben, als Probanden, denen der *Mapping*-Prozeß abgenommen wurde. Zudem verwenden sie das Merkmal Rekursionstyp signifikant häufiger zur Klassifikation von Funktionen. Eine selbständige Durchführung der Vergleichsprozesses scheint also den Aufbau von generalisierten Schemastrukturen zu begünstigen. Jedoch scheinen die an die Schemata gebundenen Regeln bei einer Vorgabe der Vergleichsprozesse besser inferierbar zu sein.

Ein Einfluß der Beispielähnlichkeit auf den Wissenserwerb konnte nicht eindeutig nachgewiesen werden. In der Tendenz lösen zwar Probanden, die mit Beispielen mittlerer Ähnlichkeit gearbeitet haben, mehr Aufgaben im Abschlußtest und zeigen vor allem bezüglich der Programmieraufgaben den besten Transfer. Dieser Befund ist allerdings nicht signifikant. Da aufgrund der kleinen Zellenbesetzungen und der ungünstigen Verteilung der Syntaxkenntnisse jedoch nur sehr starke Effekte in statistischen Analysen signifikant werden könnten, sind hier sicher weitere Untersuchungen notwendig, um eindeutigere Aufschlüsse über die Effekte der Beispielähnlichkeit auf den Lernstransfer zu erhalten.

Der fehlende Einfluß der Beispielähnlichkeiten auf den durch die Sortieraufgabenerfaßten Schemaerwerb kann mehrere Ursachen haben. Erstens könnte das Lösen der Aufgaben selbst den Wissenserwerbsprozeß dominiert haben und damit hätten die Aufgabenlösungen (*learning by doing*) die Wirkung der Beispiele überlagert. Zweitens ermöglichen - auch nach dem in Kapitel 6.3 vorgestellen Modell des Wissenserwerbs - alle Aufgaben-Beispielkombinationen die Inferenz der in der Expertenhierarchie verwendeten Merkmale. Die unterschiedlichen Generalisierungen, die von der strukturellen Ähnlichkeit von Beispiel und Aufgabe abhängen, sind wohl auf einem wesentlich feineren Niveau, als es mit den verwendeten Kennwerten zur Beschreibung der Schemainduktion erfaßbar ist. Möglicherweise könnten detaillierter Analysen des Lösungsverhaltens über die sechs rekursiven Aufgaben hier genaueren Aufschluß über die unterschiedliche Wirkung der Beispiele geben.

Die von Novick und Holyoak (1991) berichteten Zusammenhänge zwischen Aufgabenlösung, Transfer und Schemainduktion konnten nicht repliziert werden. Die Aufgabenlösung korreliert zwar sowohl mit Schemaerwerb (3. Sortierdurchgang) als auch mit Lerntransfer signifikant. Jedoch konnte der berichtete hohe Zusammenhang von Schemaerwerb und Transfer in dieser Untersuchung nicht eindeutig repliziert werden. Es konnte lediglich gezeigt werden, daß die Verwendung des Merkmals "Rekursionstyp" einen signifikanten Einfluß auf den Lerntransfer hat.

8.4 Zusammenfassung und Diskussion der Ergebnisse

Um die aus dem in Kapitel 6 vorgestellten Modell abgeleiteten Annahmen zum Erwerb rekursiver Programmiertechniken mit Beispielen empirisch zu prüfen, wurde die Lernumgebung LEAR entwickelt. Diese Lernumgebung ermöglicht es, Lernprozesse über längere Zeiträume experimentell kontrolliert zu erfassen (vgl. Kap. 7). Die verwendete einfache funktionale Sprache, die Lernumgebung, sowie das Curriculum haben sich in beiden Untersuchungen bewährt. Obwohl es sich bei den Probanden um Psychologiestudenten mit wenig Computervorerfahrung handelte, konnte in beiden Untersuchungen ein genereller Lernfortschritt nachgewiesen werden.

In der ersten Untersuchung wurden recht allgemeine, aus dem Modell ableitbare Annahmen geprüft. Es konnte nachgewiesen werden, daß das analoge Lernen mit Beispielen im Vergleich zu einer direkten Vorgabe der Problemlöseregeln einen höheren kognitiven Aufwand erfordert. Die beim Vergleich von Beispiel und Aufgabe, sowie bei der Anpassung des Beispiels an die Aufgabe notwendigen Inferenzprozesse sind für Personen mit wenig Vorwissen zeitintensiv und fehleranfällig. Dennoch werden, wie im Modell angenommen, aus den inferierten Problemlösungen und den Beispiellösungen allgemeinere Schemata gebildet, so daß ein Transfer des Gelernten auf neue Aufgaben möglich ist.

In der zweiten Untersuchung wurde der Prozeß des Erwerbs von rekursiven Problemlösetechniken mit Beispielen detaillierter betrachtet. Während in Untersuchung I mehrere Maße zur Beschreibung des Verhaltens bei der Aufgabenlösung erhoben wurden, konzentrierte sich die zweite Untersuchung auf die Analyse des erworbenen Wissens. Das Modell postuliert, daß beim analogen Lernen eine Schemahierarchie, in der strukturelle Aspekte und Regeln integriert repräsentiert werden, inferiert wird. Aus den Annahmen des Modells kann abgeleitet werden, daß die strukturelle Ähnlichkeit eines Beispiels zur Aufgabe die Generalität des inferierten Schemas beeinflußt. Entsprechend sollten Beispiele mit einer "mittleren" strukturellen Ähnlichkeit zur Aufgabe sich besonders günstig auf den Aufbau der Schemahierarchie auswirken, da in diesem Fall einerseits allgemeinere Strukturen als bei

sehr ähnlichen Beispielen inferiert werden können und andererseits noch sinnvolle Inferenzen gezogen werden können, was bei zu unähnlichen Beispielen nicht mehr gewährleistet ist.

Diese Annahmen konnten leider nicht eindeutig bestätigt werden. Im Bezug auf die indirekte Erfassung des Schemaerwerbs durch Transferaufgaben zeigt sich zwar in der Tendenz ein Vorteil für Probanden, die Beispiele mittlerer Ähnlichkeit erhielten, der Einfluß ist jedoch nicht signifikant. Für die direktere Erfassung der Schemahierarchie durch Sortieraufgaben konnte kein Einfluß der Beispielähnlichkeit nachgewiesen werden. Eine indirekte Bestätigung liefert jedoch der Befund, daß Personen, die zur Klassifikation der Funktionen das für Experten als relevant postulierte Merkmal "Rekursionstyp" verwenden, signifikant höhere Leistungen bei Transferaufgaben erzielen, als Probanden, die dieses Merkmal nicht verwendet haben.

Hier sind weitere Untersuchungen notwendig, bei denen folgende Probleme berücksichtigt werden sollten: Erstens wurde bei der Untersuchung ein Kompromiß zwischen Lerndauer und der Möglichkeit der experimentellen Kontrolle geschlossen. Es ist jedoch anzunehmen, daß sich eine Hierarchie von Rekursionsschemata über wesentlich längere Zeiträume entwickeln muß, entsprechend wäre es sinnvoll, den Lernprozeß über einen längeren Zeitraum mit zunehmend anspruchsvolleren Aufgabenstellungen zu untersuchen. Zweitens waren die Beispiel-Aufgabenkombinationen so konstruiert, daß die in den Sortieraufgaben erfaßten Merkmale für alle Kombinationen inferierbar waren. Hinweise auf einen Einfluß der Beispielähnlichkeit auf die Generalität der inferierten Schemata sind also auf einem wesentlich detaillierteren Betrachtungsniveau zu erwarten, als es durch die Sortieraufgaben erfaßt werden konnte. Entsprechend könnten die in Untersuchung II erhobenen *logfiles* dahingehend analysiert werden, welches Lösungsverhalten die Probanden über die sechs rekursiven Aufgaben hinweg an den Tag legen. Entsprechend der vorliegenden Beispielbedingung sollten bestimmte Inferenzen zur Aufgabenlösung besser oder schlechter gelingen. In einer weiteren Untersuchung könnte auch versucht werden, Aufgaben und Beispiele so zu konstruieren, daß nach den Annahmen des Modells nur in einigen Bedingungen die für Experten postulierten Strukturmerkmale inferierbar sind.

9 Diskussion

Ziel der Arbeit war es, einen Beitrag zur Forschung zum induktiven Erwerb komplexer Problemlösefertigkeiten zu liefern. Dabei wurde als Lerngegenstand das Erstellen rekursiver Funktionen in einer funktionalen Programmiersprache gewählt. Die betrachtete Variante des induktiven Lernens ist das analoge Lernen (Lernen mit Beispielen).

Da im Bereich induktives Lernen sowohl im Bereich Maschinelles Lernen als auch in der Kognitiven Psychologie vorrangig der Begriffserwerb betrachtet wird, gibt es relativ wenig Arbeiten zu diesem Thema. Ausnahmen sind die Arbeiten von Anderson (Anderson 1982; Anderson & Thompson 1989) und Carbonell (1983; 1986). Die Annahmen dieser und anderer Autoren (Holland, Holyoak, Nisbett & Thagard 1986) zum analogen Wissenserwerb wurden von mir übernommen. Lernen mit Beispielen wird als vierstufiger Prozeß betrachtet: Bei gegebener Aufgabenstellung wird (1) ein dieser Aufgabenstellung möglichst strukturähnliches Beispiel abgerufen, (2) Beispielstruktur und Aufgabenstruktur werden miteinander verglichen und (3) der Lösungsweg des Beispiels an die Aufgabe angepaßt. (4) Vergleichs- und Anpassungsprozesse führen zur Inferenz eines Schemas, das die gemeinsamen Merkmale von Aufgabe und Beispiel repräsentiert. Der Schemaerwerb bedingt, daß bei Fortschreiten des Lernprozesses neue Aufgaben zunehmend ohne Rückgriff auf externe Beispiele gelöst werden können.

Abweichend von den Annahmen der genannten Autoren wurde jedoch ein Repräsentationsformat vorgeschlagen, das Schemata und Produktionsregeln integriert (vgl. Holyoak & Nisbett 1989). Es wurde skizziert, daß bei Verwendung dieses Repräsentationsformats verschiedene Expertenleistungen mit einer einheitlichen Wissensstruktur modellierbar sind. Das Repräsentationsformat wurde verwendet, um Expertenwissen über rekursive Programmierung zu modellieren. Analoges Lernen wurde dann als Prozeß des allmählichen Aufbaus einer Expertenhierarchie beschrieben. Dabei wurde angenommen, daß erste Informationen über Rekursion direkt (über Lehrbücher oder Vorlesungen) vermittelt werden und damit ein rudimentäres allgemeines Rekursionsschema zur Verfügung steht, wenn die Lernenden beginnen, rekursive Aufgaben mit Hilfe von Beispielen zu lösen. Jedes Aufgaben-Beispielpaar wird dann zu einem abstrakteren Schema generalisiert. Dieses Schema behält Aufgabe und Beispiel als Kinder und das bei der Aufgabenlösung verwendete Schema als Vater. Auf diese Art entwickelt sich eine immer differenziertere Hierarchie von Schemata. Scheitern Lernende bei der Lösung von Transferaufgaben, so kann dies entweder an einer unvollständigen Repräsentation der rekursiven Strukturen oder an einer unvollständigen Repräsentation der Problemlöseregeln liegen.

Das vorgeschlagene Modell wurde bisher nur informell dargestellt. Es wäre aber möglich, und ist anzustreben, das Modell soweit zu präzisieren, daß es implementierbar wird. Der Schwerpunkt dieser Arbeit lag auf der empirischen Prüfung von aus dem Modell ableitbaren Annahmen über analoges Lernen rekursiver Programmiertechniken.

Zu diesem Zweck wurde eine Lernumgebung entwickelt, die auf einem Interpreter

für eine einfache funktionale Sprache basiert. Mit dieser Lernumgebung war es möglich, längerfristige Lernprozesse unter experimentell kontrollierten Bedingungen zu erfassen. Die verwendete Sprache, die Lernumgebung und ein erstelltes Curriculum erwiesen sich als geeignet, Personen ohne Programmiererfahrung in wenigen Tagen erfolgreich die Grundlagen rekursiver Programmierung zu vermitteln.

In zwei umfangreichen Untersuchungen wurden Annahmen des Modells überprüft. Eine erste Untersuchung ergab, daß Lernen mit Beispielen zwar kognitiv aufweniger ist, als eine direkte Vorgabe von Regeln aber dennoch zu vergleichbarem Wissenserwerb führt. In einer zweiten Untersuchung wurde der Einfluß der Beispielähnlichkeit auf das analoge Lernen betrachtet. Hier wurde die Annahme, daß sowohl strukturelle als auch regelhafte Wissensstrukturen erworben werden dadurch geprüft, daß sowohl Sortieraufgaben als auch Transferaufgaben zur Analyse des erworbenen Wissens verwendet wurden. Obwohl der Einfluß der Beispielähnlichkeit auf die Generalität der erworbenen Wissensstrukturen nur teilweise nachgewiesen werden konnte, zeigte sich, daß Personen, die angemessene Schemastrukturen erworben haben, auch eher in der Lage sind, Transferaufgaben erfolgreich zu lösen.

Die empirischen Befunde der beiden Untersuchungen sind zwar vielversprechend, doch läßt vor allem die zweite Untersuchung wesentliche Fragen noch ungeklärt. Eine Durchführung weiterer Untersuchungen mit der entwickelten Lernumgebung wäre deshalb sinnvoll. Der Grad der empirischen Bestätigung der im Modell formulierten Annahmen scheint mir jedoch auszureichen, um eine Präzisierung des Modells anzustreben. Der Versuch einer Implementation des Modells könnte wesentlich zur Aufdeckung von Inkonsistenzen und bisher fehlenden Prozeßannahmen beitragen.

Die Arbeit ist jedoch nicht nur als ein Beitrag zur kognitionspsychologischen Modellbildung zu verstehen. Es lassen sich Prinzipien für die Programmierdidaktik ableiten, die unter anderem auch Anwendung in intelligenten Tutorensystemen finden könnten. Es scheint sinnvoll zu sein, rekursive Programmiertechniken mit Hilfe von Beispielen einzuführen, wobei besonderer Wert auf die Verdeutlichung der strukturellen Übereinstimmung verschiedener Funktionen gelegt werden sollte. Ein weiteres Anwendungsgebiet, das von dieser Arbeit berührt wird, ist das automatische Programmieren. Das vorgeschlagene integrierte Repräsentationsformat von Schemata und Regeln bietet die Möglichkeit, einer effizienten Wissensrepräsentation. Aufgrund einer vorgegebenen Spezifikation kann ein Schema aktiviert werden. Der Strukturteil des Schemas stellt bereits eine partielle Aufgabenlösung dar. Die noch offenen Ausdrücke können mit Hilfe der an das Schema gebundenen Regeln instantiiert werden. Durch die formulierten Annahmen zum sequentiellen Aufbau einer solchen Schemahierarchie durch analoges Aufgabenlösung wäre es unter Umständen möglich, zu versuchen, ob

Ansätze des automatischen Programmierens mit einer Lernkomponente kombinierbar sind. Ein solches Projekt scheint mir für beide Bereiche - maschinelles Lernen und automatisches Programmieren - fruchtbar.

Abelson, H. & Sussman, G.J. with Sussman, J. (1985). *Structure and Interpretation of Computer Programs*. Cambridge, Mas.: MIT Press.

Adelson, B. (1981). Problem solving and the development of abstract categories in programming languages. *Memory & Cognition, 9* (4), 422-433.

Amarel, S. (1972). Representation and modelling in problems of program formation. In B. Metzler and D. Michie (Ed.), *Machine Intelligence* (Vol. 6, pp. 411-466). New York: American Elsevier.

Amarel, S. (1983). Problems of representation in heuristic problem solving: Related issues in the development of expert systems. In R. Groner, M. Groner and W.F. Bishof (Ed.), *Methods of Heuristics*. Hillsdale, N.J.: Erlbaum.

Amarel, S. (1986). Program synthesis as a theory formulation task: Problem representations and solution methods. In R.S. Michalski, J.G. Carbonell & T.M. Mitchell (Eds.), *Machine Learning - An Artificial Intelligence Approach* (Vol. 2, pp. 499-570). Los Altos, Cal.: Morgan Kaufmann.

Amthauer, R. (1971). *Intelligenz-Struktur-Test I-S-T-70*. Göttingen: Hogrefe.

Anderson, J.R. (1976). *Language, Memory, and Thought*. Hillsdale, N.J.: Erlbaum.

Anderson, J.R. (1978). Arguments concerning representations for mental imagery. *Psychological Review, 85*, 249-277.

Anderson, J.R. (1981). *Cognitive Skills and Their Acquisition*. Hillsdale, N.J.: Lawrence Erlbaum.

Anderson, J. R. (1982). Aquisition of cognitive skill. *Psychological Review, 89*, 369-406.

Anderson, J. R. (1983). *The Architecture of Cognition* (Cognitive Science Series 5). Cambridge, Mass.: Havard Univ. Press.

Anderson, J.R. (1986). Knowledge compilation: A general learning mechanism. In R.S. Michalski, J.G. Carbonell and T.M. Mitchell (Ed.), *Machine Learning - An Artificial Intelligence Approach* (Vol. 2, pp. 289-310). Palo Alto, Cal: Tioga.

Anderson, J.R. & Bower, G. (1973). *Human Associative Memory*. Washington, D.C.: Winston.

Anderson, J. R., Conrad, F. G. & Corbett, A. T. (1989). Skill acquisition and the LISP tutor. *Cognitive Science, 13*, 467-505.

Anderson, J.R., Corbett, A.T. & Reiser, B.J. (1987). *Essential LISP*. Reading, Mass.: Addison-Wesley.

Anderson,J. R., Farrell, R. & Sauers, R. (1984). Learning to program in LISP. *Cognitive Science, 8*, 87-129.

Anderson, J.R. & Thompson, R. (1989). Use of analogy in a production system architecture. In S. Vosniadou and A. Ortony (Ed.), *Similarity and Analogical Reasoning* (pp. 267-297). Cambridge: Cambridge University Press.

Anderson, R.C. & Pearson, P.D. (1984). A schema-theoretic view of basic processes in reading comprehension. In P.D. Pearson (Ed.), *Handbook of Reading Research* (pp. 255-291). New York: Longman.

Anzai, Y. & Simon, H.A. (1979). The theory of learning by doing. *Psychological Review, 86*, 124-140.

Atkinson, R.C. & Shiffrin, R.M. (1968). Human memory: a proposed system and its control processes. In K.W. Spence and J.T. Spence (Ed.), *The Psychology of Learning and Motivation: advances in research and theory* (Vol. 2, pp. 242-293). New York: Academic Press.

Backus, J. (1978). Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs. *Communications of the ACM, 21*, 613-641.

Backus, J.W. (1959). The Syntax and Semantics of the Proposed International Algebraic Language of the Zürich ACM-GAMM Conference. *Proceedings of the International Conference on Information Processing* UNESCO, 125-132.

Bandura, A. & Walters, R. (1963). *Social Learning and Personality Development*. New York: Holt, Rinehart & Winston.

Barr, A. & Feigenbaum, E.A. (1982). *The Handbook of Artificial Intelligence* (Vol. 2). Los Altos, Cal.: William Kaufmann.

Barstow, D. (1977). A knowledge-based system for automatic program construction. *IJCAI, 5*, 382-388.

Bartlett, F.C. (1932). *Remembering: An Experimental and Social Study*. Cambridge: Cambridge University Press.

Beckstein, C. (Kap.Hrsg.). (1993). KI-Programmierung. In G. Görz (Ed.), *Einführung in die künstliche Intelligenz* (pp. 883-1035). Bonn: Addison-Wesley.

Bonar, J. & Soloway, E. (1989). Preprogamming knowledge: A major source of misconceptions in novice programmers. In E. Soloway and J.C. Spohrer (Ed.), *Studying the Novice Programmer* (pp. 325-354). Hilldale, N.J.: Lawrence Erlbaum.

Bourne, L.E., Jr. (1974). An inference model of conceptual rule learning. In R. Solso (Ed.), *Theories in Cognitive Psychology* (pp. 393-395). Washington, DC: Erlbaum.

Brachman, R.J. & Schmolze, J.G. (1985). An overview of the KL-ONE knowledge representation language. *Cognitive Science, 9*, 171-216.

Bransford, J.D. & Johnson, M.K. (1972). Contextual prerequisites for understanding: Some investigations of comprehension and recall. *Journal of Verbal Learning and Verbal Behavior, 11*, 717-726.

Bruner, J.S., Goodnow, J.J. & Austin, G.A. . (1956). *A study of thinking*. New York: Wiley.

Burstall, R.M., MacQueen, D.B. & Sanella, D.T. (1980). *HOPE: An experimental applicative language* (Vols. CSR-62-80). Department of Computer Science, University of Edinburgh.

Carbonell, J.G. (1983). Learning by analogy: Formulating and generalizing plans from past experience. In R.S. Michalski, J.G. Carbonell and T.M. Mitchell (Ed.), *Machine Learning - An Artificial Intelligence Approach* (Vol. 1, pp. 137-162). New York: Springer.

Carbonell, J.G. (1986). Derivational analogy: A theory of reconstructive problem solving and expertise acquistion. In R.S. Michalski, J.G. Carbonell and T.M. Mitchell (Ed.), *Machine Learning - An Artificial Intelligence Approach* (Vol. 2, pp. 371-392). Los Altos, CA: Morgan Kaufmann Pub.

Carey, S. (1985). *Conceptual Change in Childhood*. Boston: MIT Press.

Carnap, R. & Stegmüller, W. (1959). *Induktive Logik und Wahrscheinlichkeit*. Wien: Springer.

Chase, W. G. & Simon, H. A. (1973). Perception in chess. *Cognitive Psychology, 4*, 55-81.

Cheng, P.W., Holyoak, K.J., Nisbett, R.E. & Oliver, L.M. (1986). Pragmatic versus syntactic approaches to training deductiv reasoning. *Cognitive Psychology, 18*, 293-328.

Chi, M.T.H., Feltovich, P.J. & Glaser, R. (1981). Categorization and representation of physics problems by experts and novices. *Cognitive Science, 5*, 121-152.

Cohen, P.R. & Feigenbaum, E.A. (1982). *The Handbook of Artificial Intelligence* (Vol. 3). Los Altos, Ca: William Kaufmann.

Collins, A.M. & Quillian, M.R. (1969). Retrieval time from semantic memory. *Journal of Verbal Learning and Verbal behavior, 8*, 240-247.

Corno, L. & Haertel, E. (1982). *Selection and validation of a problem solving test for use in Amdahl PS & S technical personnel selection*. Sunnyvale, Cal.: Amdahl Corporation.

Craik, K. (1943). *The Nature of Explanation*. Cambridge, Mass: Cambridge University Press.

Davis, M. (1958). *Computability and Unsolvability*. New York: McGraw-Hill.

DeGroot, A.D. (1965). *Thought and Choice in Chess*. DenHaag: Mouton.

DeJong, G. (1986). An approach to learning from observation. In R.S. Michalski, J.G. Carbonell and T.M. Mitchell (Ed.), *Machine Learning - An Artificial Intelligence Approach* (Vol. 2, pp. 571-590). Los Altos, Cal.: Morgan Kaufmann Pub.

Dershowitz, N. (1986). Programming by analogy. In R.S. Michalski, J.G. Carbonell and T.M. Mitchell (Ed.), *Machine Learning - An Artificial Intelligence Approach* (Vol. 2, pp. 393-422). Los Altos, Ca: Morgan Kaufmann.

Dershowitz, N. & Manna, Z. (1977). The evolution of programs: Automatic program modification. *IEEE Transaction on Software Engineering, SE-3* (6), 377-385.

Dietterich, T.G. & Michalski, R.S. (1981). Inductive learning of structural descriptions: Evaluation criteria and comparative review of selected methods. *Artificial Intelligence, 16*, 257-294.

Ebbinghaus, H.D., Flum, J. & Thomas, W. (1986). *Einführung in die mathematische Logik*. Darmstadt: Wissenschaftliche Buchgesellschaft.

Eckes, Th. & Roßbach, H. (1980). *Clusteranalysen*. Stuttgart: Kohlhammer.

Ehrig, H. & Mahr, B. (1985). *Fundamentals of Algebraic Specification 1 - Equations and Initial Semantics*. Berlin: Springer.

Elgot, C.C. (1971). Algebraic theories and program schemes. In E. Engeler (Ed.), *Lecture Notes in Mathematics: Symposium on Semantics of Algorithmic Languages* (Vol. 188, pp. 71-88). Berlin: Springer.

Engelfriet, J. (1974). *Simple Program Schemes and Formal Languages*. Berlin: Springer.

Essler, W.K. (1980). Induktion. In J. Speck (Ed.), *Wissenschaftstheoretische Begriffe* (Vol. 2, pp. 297-307). Göttingen: UTB.

Estes, W.K. (1959). The statistical approach to learning theory. In S. Koch (Ed.), *Psychology: A Study of a Science* (pp. 380-491). New York: McGraw-Hill.

Eyferth, K. (1988). Die Psychologie der elektronischen Datenverarbeitung. *Eröffnungsvortrag des 36. Kongresses der DGfP in Berlin.*

Field, A.J. & Harrison, P.G. (1988). *Functional Progamming*. Reading, Mas.: Addison-Wesley.

Fikes, R.E. & Nilsson, N.J. (1971). STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2 (3), 189-208.

Fodor, J.A. (1975). *The Language of Thought*. Crowell, New York:.

Fodor, J.A. (1980). Fixation of belief and concept acquisition. In M. Piattelli-Palmarini (Ed.), *Language and Learning: The Debate Between Jean Piaget and Noam Chomsky*. Cambridge, Mass: Havard University Pres.

Fodor, J. A. & Pylyshyn, Z. W. (1988). Connectionism and cognitive architecture: A critical analysis. *Cognition, 28*, 3-71.

Fong, G.T., Krantz, D.H. & Nisbett, R.E. (1986). The effects of statistical training on thinking about everyday problems. *Cognitive Psychology, 18*, 253-292.

Forbus, K.D. & Gentner, D. (1986). Learning physical domains: Toward a theoretical framework. In R.S. Michalski, J.G. Carbonell and T.M. Mitchell (Ed.), *Machine Learning - An Artificial Intelligence Approach* (Vol. 2, pp. 311-348). Los Altos, Cal.: Morgan Kaufmann.

Gentner, D. (1983). Structure-mapping: a theoretical framework for analogy. *Cognitive Science, 7*, 155-170.

Glenberg, A.M. & Meyer, M. & Lindem, K. . (1987). Mental Models Contribute to Foregrounding during Text Comprehension. *Journal of Memory and Language, 26*, 69 - 83.

Goebel, R. & Vorberg, D. (1991). Das Lösen rekursiver Programmierprobleme: Ein Simulationsmodell. *Kognitionswissenschaft, 2*, 27-36.

Goodman, N. (1965). *Fact, Fiction, and Forecast*. Indianapolis: Bobbs-Merrill.

Green, C.C. & Barstow, D.R. (1978). On program synthesis knowledge. *Artificial Intelligence, 10*, 241-279.

Greibach, S.A. (1975). *Theory of Program Structures: Schemes, Semantics, Verification*. Berlin: Springer.

Gugerty, L. & Olson, G.M. (1987). Comprehension differences in debugging by skilled and novice programmers. In E. Soloway and S. Iyengar (Ed.), *Empirical Studies of Programmers* (pp. 13-27). Norwood, N.J.: Ablex.

Haak, U., Hahn, K. & Wagner, K.U. (1989). Programmieren als Problemlösen: Die Struktur von Expertenwissen. *Zeitschrift für Psychologie, 197*, 247-262.

Harel, D. (1987). *Algorithmics - The Spirit of Computing*. Wokingham, Eng.: Addison-Wesley.

Heller, R. (1987). Different LOGO teaching styles: Do they really matter. In E. Soloway and S. Iyengar (Ed.), *Empirical Studies of Programmers* (pp. 117-127). Norwood, N.J.: Ablex.

Hempel, G.G. (1965). *Aspects of Scientific Explanation*. New York: The Free Press.

Hilbert, D. & Ackermann, W. (1959). *Grundzüge der theoretischen Logik*. Berlin: Springer.

Hilpinen, R. (1975). Carnap's inductive logic. In J. Hintikka (Ed.), *Rudolf Carnap, Logical Empiricist* (pp. 333-360). Dordrecht-Holland: D. Reidel.

Hintikka, J. & Niinilouto, I. (1976). An axiomatic foundation for the logic of inductiv generalization. In M. Przelecki, U. Szaniawski and R. Wojcicki (Ed.), *Formal Methods of the Methodology of Science* (pp. 57-81). Wroclaw: Ossolineum.

Hirschberger, J. (1963). *Geschichte der Philosophie* (Vol. 1,2). Basel: Herder.

Holland, J.H., Holyoak, K.J., Nisbett, R.E. & Thagard, P.R. (1986). *Induction - Processes of Inference, Learning, and Discovery*. Cambridge, Mas.: MIT Press.

Holyoak, K.J. & Thagard, P. (1989). Analogical mapping by constraint satisfaction. *Cognitive Science, 13*, 295-355.

Holyoak, K.J. & Thagard, P.R. (1989). A computational model of analogical problem solving. In S. Vosniadou and A. Ortony (Ed.), *Similarity and Analogical Reasoning* (pp. 242-266). Cambridge: Cambridge University Press.

Hopcroft, J.E. & Ullman, J.D. (1990). *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie*. Bonn: Addison-Wesley.

Horowitz, E. & Sahni, S. (1978). *Fundamentals of Computer Algorithms*. Potomac, MD: Computer Science Press.

Hussy, W. (1984). *Denkpsychologie: Ein Lehrbuch* (Vol. 1). Stuttgart: Kohlhammer.

Johnson-Laird, P.N. (1983). *Mental Models: Towards a Cognitive Science of Language, Inference, and Consciousness*. Cambridge: Cambridge University Press.

Johnson-Laird, P.N. (1988). Freedom and constraint in creativity. In R.J. Sternberg (Ed.), *Creativity*. Cambridge: Cambridge University Press.

Johnson-Laird, P.N., Byrne, R.M.J. & Schaeken, W. (1992). Propositional reasoning by model. *Psychological Review, 99* (3), 418-439.

Kahney, H. (1989). What do novice programmers know about recursion? In E. Soloway a. J.C. Spohrer (Ed.), *Studying the Novice Programmer* (pp. 209-228) Lawrence Erlbaum.

Klahr, D., Langley, P. & Neches, R. (1987). *Production System Models of Learning and Development*. Cambridge, Mass: MIT Press.

Klix, F. (1980). *Erwachendes Denken*. Berlin: Deutscher Verlag der Wissenschaften.

Kolodner, J.L. (1984). *Retrieval and Organizational Strategies in Conceptual Memory: A Computer Model*. Hillsdale, N.J.: Erlbaum.

Kosslyn, S. (1980). *Image and Mind*. Cambridge, MA: Harvard UP.

Krause, W. & Wysotzki, F. (1984). Computermodelle und psychologische Befunde der Wissensrepräsentation. In F. Klix (Ed.), *Gedächtnis, Wissen, Wissensnutzung*. Berlin: Deutscher Verlag der Wissenschaften.

Kurland, D.M., Pea, R.D., Clement, C. & Mawby, R. (1989). The study of the development of programming ability and thinking skills in high school students. In E. Soloway and J.C. Spohrer (Ed.), *Studying the Novice Programmer* (pp. 83-112). Hillsdale, N.J.: Lawrence Erlbaum.

Kutschera, F. (1972). *Wissenschaftstheorie* (Vol. I, Kap. 2). München: UTB.

Labov, W. (1973). The boundaries of words and their meanings. In C.J.N. Bailey and R.W. Shuy (Ed.), *New Ways of Analysing Variation in English* (pp. 340-373). Washington: Georgetown University Press.

Lakulos, I. (1976). *Proofs and Refutations. The Logic of Mathematical Discovery*. Cambrige: Cambridge University Press.

Langley, P. (1983). Learning search strategies through discrimination. *International Journal of Man-Machine Studies, 18,* 513-541.

Larkin, J.H. & Simon, H.A. (1987). Why a diagram is (somtetimes) worth ten thousand words. *Cognitive Science, 11,* 65-99.

Lebowitz, M. (1987). Experiments with incremental concept formation: UNIMEM. *Machine Learning, 2,* 103-138.

Lenat, D.B. (1980). AM: An artificial intelligence approach to discovery in mathematics as heuristic search. In R. Davis and D.B. Lenat (Ed.), *Knowledge-based Systems in Artificial Intelligence.* New York: McGraw-Hill.

Loeckx, J. & Sieber, K. (1984). *The Foundations of Program Verification.* Stuttgart: Teubner.

Manna, Z. & Waldinger, R. (1975). Knowledge and reasoning in program synthesis. *Artificial Intelligence, 6,* 175-208.

Maxwell, G. (1975). Induction and empiricism: A Bayesian-frequentist alternative. In G. Maxwell and R.M. Anderson Jr. (Ed.), *Minnesota Studies in the Philosophy of Science - Induction, Probability, and Confirmation* (Vol. VI, pp. 106-166). Minneapolis: University of Minnesota Press.

Mayer, R.E. (1988). The psychology of how novices learn computer programming. In E. Soloway and J.C. Spohrer (Ed.), *Studying the Novice Programmer* (pp. 129-160). Hilldale, N.J.: Lawrence Erlbaum.

McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P. & Levin, M.I. (1962). *LISP 1.5 Programmers Manual.* MIT Press.

McDermott, D.V. (1967). Planning and Acting. *Cognitive Science, 2* (2), 71-109.

McKeithen, K.B., Reitman, J.S., Rueter, H.H. & Hirtle, S.C. (1981). Knowledge organization and skill differences in computer programmers. *Cognitive Psychology, 13,* 307-325.

Metzler, J. & Shepard, R.N. (1974). Transformational studies of the internal representation of three-dimensional objects. In R.L. Solso (Ed.), *Theories of Cognitive Psychology: The Loyola Symposium.* Hilldale, N.J.: Erlbaum.

Michalski, R.S. (1986). Understanding the nature of learning. In R.S. Michalski, J.G. Carbonell and T.M. Mitchell (Ed.), *Machine Learning - An Artificial Intelligence Approach* (Vol. II, pp. 471-498). Los Altos, Cal.: Tioga.

Michalski, R.S. (1980). Pattern recognition as rule-guided inductive inference. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 2,* 349-361.

Michalski, R.S. & Stepp, R.E. (1983). Learning from observation: Conceptual clustering. In R.S. Michalski, J.G. Carbonell and T.M. Mitchell (Ed.), *Machine Learning - An Artificial Intelligence Approach* (Vol. 1, pp. 331-363). New York: Springer.

Mill, J.S. (1843/1974). *A system of logic ratiocinative and inductive*. Toronto: Univerity of Toronto Press.

Miller, G.A. (1956). The magical number seven plus or minus two: Some limits on our capacity for processing informations. *Psychological Review, 63*, 81-97.

Milner, R., Tofte, M. & Harper, R. (1990). *The Definition of Standard ML*. Cambridge, Mass.: MIT-Press.

Minsky, M. (1975). A framework for representing knowledge. In P. H. Winston (Ed.), *The Psychology of Computer Vision* (pp. 211-277). New York: McGraw-Hill.

Mitchell, T.M. (1982). Generalisation as search. *Artificial Intelligence, 18* (2) 203-226.

Mitchell, T.M., Utgoff, P.E. & Banerji, R. (1983). Learning by experimentation: Acquiring and refining problem-solving heuristics. In R.S. Michalski, J.G. Carbonell and T.M. Mitchell (Ed.), *Machine Learning - An Artificial Intelligence Approach* (Vol. 1, pp. 163-190). New York: Springer.

Naur, P. et al. (1960). Report on the Algorithmic Language ALGOL 60. *Communications of the ACM, 6* (1), 1-17.

Nebel, B. (1990). *Reasoning and Revision in Hybrid Representation Systems*. Berlin: Springer.

Neches, R., Langley, P. & Klahr, D. (1987). Learning, development, and production systems. In D, Klahr, P. Langley and R. Neches (Ed.), *Production System Models of Learning and Development* (pp. 1-54). Cambridge, Mass: MIT Press.

Neisser, U. & Weene, P. (1962). Hierarchies in concept attainment. *Journal of Experimental Psychology, 64*, 640-645.

Newell, A. (1972). A theoretical exploration of mechanisms for coding the stimulus. In A.W. Melton and E. Martin (Ed.), *Coding Processes in Human Memory*. Washington, DC: Winston.

Newell, A. & Simon, H.A. (1972). *Human problem solving*. Englewood Cliffs, N.J.: Prentice Hall.

Novick, L.R. (1988). Analogical transfer, problem similarity, and expertise. *Journal of Experimental Psychology: Learning, Memory, and Cognition, 14*, 510-520.

Novick, L.R. & Holyoak, K.J. (1991). Mathematical problem solving by analogy. *Journal of Experimental Psychology: Learning, Memory, and Cognition, 17* (3), 398-415.

Opwis, K. (1988). Produktionssysteme. In H. Mandl und H. Spada (Ed.), *Wissenspsychologie* (pp. 74-98). München: Psychologie Verlags Union.

Osherson, D.N. & Smith, E.E. (1982). Gradedness and conceptual combination. *Cognition, 12*, 299-318.

Papert, S. (1980). *Mindstorms: Children, Computers and Powerful Ideas*. New York: Basic Books.

Peltason, C., Schmiedel, A., Kindermann, C. & Quantz, J. (1989). The BACK System Revisited. *KIT-Report*. Berlin: Fachbereich Informatik, *75*.

Perl, J. (1979). *Rekursive Programmierung*. München: Carl Hanser.

Piaget, J. (1936). *The Origins of Intelligence in Children*. In M. Cook (Trans.), New York: International University Press.

Piaget, J. (1952). *The Origins of Intelligence*. New York: International University Press.

Pirolli, P., & Anderson, J.R. (1985). The role of learning from examples in the acquisition of recursive programming skills. *Canadian Journal of Psychology, 39*, 240-272.

Pirolli, P.L. (1986). A cognitive model and computer tutor for programming recursion. *Human-Computer-Interaction, 2*, 319-355.

Pirolli, P.L. & Anderson, J.R. (1985). The role of learning from examples in the acquisition of recursive programming skills. *Canadian Journal of Psychology, 39*, 240-272.

Polya, G. (1957). *How to Solve It*. Princeton, New Jersey: Princeton University Press.

Popper, K. (1984). *Logik der Forschung*. Tübingen: Mohr.

Putnam, R.T., Sleeman, D., Baxter, J.A. & Kuspa, L. (1986). A summary of misconceptions of high school BASIC programmers. *Journal of Educational Computing Research, 2*, 459-472.

Pylyshyn, Z. W. (1984). *Computation and Cognition - Toward a Foundation for Cognitive Science*. Cambridge: Bradford MIT.

Quillian, M.R. (1968). Semantic memory. In M. Minsky (Ed.), *Semantic Information Processing* (pp. 227-270). Cmbridge, MA: MIT Press.

Quinlan, J.R. (1983). Learning efficient classification procedures and their application to chess end games. In R.S. Michalski, J.G. Carbonell and T.M. Mitchell (Ed.), *Machine Learning - An Artificial Intelligence Approach* (Vol. 1, pp. 463-482). Palo Alto: Tioga.

Reed, S.K. & Bolstad, C.A. (1991). Use of examples and procedures in problem solving. *Journal of Experimental Psychology: Learning, Memory, and Cognition, 17* (4), 753-766.

Reiter, R. (1980). A logic for default reasoning. *Artificial Intelligence, 1*, 81-132.

Rosch, E. (1978). Principles of categorisation. In E. Rosch and B.B. Lloyd (Ed.), *Cognition and Categorization*. Hillsdale N.J.: Lawrence Erlbaum.

Rosch, E.H. (1973). Natural categories. *Cognitive Psychology, 4*, 328-350.

Rosenbloom, P.S. & Newell, A. (1986). The chunking of goal hierarchies: A generalized model of practice. In R.S. Michalsik, J.G. Carbonell and T.M. Mitchell (Ed.), *Machine Learning - An Artificial Intelligence Approach* (Vol. 2, pp. 247-288). Los Altos, Cal.: Morgan Kaufmann.

Rosenbloom, P.S. & Newell, A. (1987). The chunking of goal hierarchies: A generalized model of practice. In R.S. Michalski, J.G. Carbonell and T.M. Mitchell (Ed.), *Machinme Learning - An Artificial Intelligence Approach* (Vol. II, pp. 247-288). Los Altos, Cal.: Morgan Kaufmann.

Ross, B.H. & Kennedy, P.T. (1990). Generalizing from the use of earlier examples in problem solving. *Journal of Experimental Psychology: Learning, Memory, and Cognition, 16* (1), 42-55.

Rumelhart, D.E. & McClelland, J.L. (1986). PDP - Parallel Distributed Processing. In J. A. Feldman, P. J. Hayes u. D. E. Rummelhard (Ed.), *Computational Models of Cognition and Perception* (Vol. 1). Cambridge, Mass.: MIT Press.

Rumelhart, D.E. & Norman, D.A. (1978). Accretion, tuning and restructuring: Three modes of learning. In J.W. Cotton and R.L. Klatzky (Ed.), *Semantic Factors in Cognition* (pp. 37-53). Hillsdale, N.J.: Erlbaum.

Rumelhart, D.E. & Norman, D.A. (1981). Analogical processes in learning. In J.R. Anderson (Ed.), *Cognitive Skills and Their Acquisition* (pp. 335-360). Hillsdale, N.J.: Lawrence Erlbaum.

Rumelhart, D.E. & Norman, D.A. (1983). Representation in Memory. In R.C. Atkinson, R.J. Herrnstein, G. Lindzey and R.D. Luce (Ed.), *Handbook of Experimental Psychology*. New York: Wiley.

Russell, B. (1939). Dewey's New 'Logic'. In P.A. Schillp (Ed.), *The Philosophie of John Dewey*. New York: Tudor.

Samuel, A.L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development, 3*, 210-229.

Samurcay, R. (1989). The concept of variable in programming: Its meaning and use in problem solving by novice programmers. In E. Soloway and J.C. Spohrer (Ed.), *Studying the Novice Programmer* (pp. 161-178). Hillsdale, N.J.: Lawrence Erlbaum.

Schank, R.C. (1982). *Dynamic Memory - A Theory of Reminding and Learning in Computers and People*. Cambridge, Mas.: Cambridge University Press.

Schank, R.C. (1983). The current state of AI: One man's opinion. *AI Magazine, 4* (1), 1-8.

Schank, R.C. & Abelson, R.P. (1977). *Scripts, plans, goals, and understanding* (Vols. Kap. 1-3). Hillsdale, N.J.: Erlbaum.

Schild, K. (1989). Towards a Theory for Frames and Rules. *KIT-Report, 76*.

Schmid, U. (1994). Programmieren lernen: Unterstützung des Erwerbs rekursiver Programmiertechniken durch Beispielfunktionen und Erklärungstexte. *Kognitionswissenschaft, 4*, (1), 47-54.

Schmid, U. & Gräbener, J. (1993). Der Einfluß von Lehrmaterial auf das Lösen rekursiver Programmieraufgaben: Eine Fehleranalyse. *Forschungsbericht aus dem Institut für Psychologie*. Berlin: Institut für Psychologie der TU Berlin, *3*.

Schmid, U. & Kaup, B. (1993). *Der Einfluß von Beispielähnlichkeit auf induktive Lernprozesse beim rekursiven Programmieren*. Bericht des Orientierungsprojektes "Programmieren Lernen mit Beispielen", durchgeführt am Institut für Psychologie der TU Berlin.

Schmid, U. & Kindsmüller, M. C. (1994). Modellierungsmethoden der Kognitionswissenschaft. *Skript zur gleichnahmigen Lehrveranstaltung am Institut für Psychologie der TU Berlin.*

Schmid, U. & Ulber, D. (1993). Determinanten des Erwerbs von Programmierkompetenz. *Forschungsbericht aus dem Institut für Psychologie*. Berlin: Institut für Psychologie der TU Berlin, *1*.

Schömann, M. (1991). *Erwerb von Rekursionsschemata in LISP. Lernprozesse, Lösungsstrategien und Wissensorganisation von AnfängerInnen und fortgeschrittenen ProgrammiererInnen* (Unveröffentlichte Dissertation). TU Braunschweig:.

Searle, J.R. (1980). Minds, brains, and programs. *The Behavioral and Brain Sciences, 3*, 417-457.

Shneiderman, B. (1976). Exploratory experiments in programmer behaviour. *International Journal of Computer Information Science, 5*, 123-143.

Shrager, J. & Pirolli, P.L. (1983). *SIMPLE: A Simple Language for Research in Programmer Psychology* [Computer Program]. Pittsburgh: Carnegie-Mellon University, Dept. of Psych.

Siklossy, L. (1976). *Let's Talk LISP*. Englewood Cliffs, N.J.: Prentice-Hall.

Siklossy, L. & Sykens, D.A. (1975). Automatic program synthesis from example problems. *IJCAI, 4*, 268-273.

Simon, H.A. (1983). Why should machines learn? In R.S. Michalski, T.M. Mitchell,, and J. Carbonell (Ed.) *Machine Learning - An Artificial Intelligence Approach* (Vol. I, pp. 25-39). New York: Springer.

Sixtl, F. (1967). *Meßmethoden der Psychologie*. Weinheim: Beltz.

Skinner, B.F. (1951). How to teach animals. *Scientific American, 185*, 26-29.

Smith, E.E. & Medin, D.L. (1981). *Categories and Concepts*. Cambridge, MA: Harvard University Press.

Smith, E.E., Shoben, E.J. & Rips, L.J. (1974). Structure and process in semantic memory: A feature model for semantic decisions. *Psychological Review, 81*, 214-241.

Snoddy, G.S. (1926). Learning and Stability. *Journal of Applied Psychology, 10*, 1-36.

Soloway, E. (1985). From problems to programs via plans: the content and structure of knowledge for introductory LISP programming. *Journal of Educational Research, 1*, 157-172.

Soloway, E., Bonar, J. & Ehrlich, K. (1989). Cognitive strategies and looping constructs: An empirical study. In E. Soloway and J.C. Spohrer (Ed.), *Studying the Novice Programmer* (pp. 191-208). Hilldale, N.J.: Lawrence Erlbaum.

Soloway, E. & Spohrer, J.C. (Eds.). (1989). *Studying the Novice Programmer*. Lawrence Erlbaum.

Spohrer, J.C., Soloway, E. & Pope, E. (1989). A goal/plan analysis of buggy Pascal programs. In E. Soloway and J.C. Spohrer (Ed.), *Studying the Novice Programmer* (pp. 355-400). Hillsdale, N.J.: Lawrence Erlbaum.

Strube, G. (Hrsg.). (1993). Kognition. In G. Görz (Ed.), *Einführung in die künstliche Intelligenz* (pp. 303-366). Bonn: Addison-Wesley.

Sussman, G.J. (1975). *A Computer Model of Skill Acquisition*. New York: American Elsevier.

Unger, S. & Wysotzki, F. (1981). *Lernfähige Klassifizierungssysteme*. Berlin.

Van Lehn, K. (1983). Validating a model of children's arithmetic skills: SIERRA. In R.S. Michalski (Ed.), *Proceedings of the International Machine Learning Workshop, University of Illinois, June 22-24.*

Vorberg, D. & Goebel, R. (1991). Das Lösen rekursiver Programmierprobleme: Rekursionsschemata. *Kognitionswissenschaft, 1*, 83-95.

Vosniadou, S. & Ortony, A. (Eds.). (1989). *Similarity and Analogical Reasoning*. Cambridge: Cambridge University Press.

Wachsmuth, I. (1993). Expertensysteme. In F. Görz (Ed.), *Einführung in die Künstliche Intelligenz* (pp.714-828). Bonn: Addison-Weseley.

Waldinger, R. & Levitt, K.N. (1974). Reasoning about programs. *Artificial Intelligence, 5*, 235-316.

Waterman, D. (1970). Generalization learning techniques for automating the learning of heuristics. *Artificial Intelligence, 1*, 121-170.

Waterman, D.A. & Hayes-Roth, F. (Eds.). (1978). *Pattern-Directed Inference Systems*. New York: Academic Press.

Wechler, W. (1992). *Universal Algebra for Computer Scientists*. Berlin: Springer.

Weimer, W. (1975). The psychology of inference and expectation: Some preliminary remarks. In G. Maxwell and R.M. Anderson Jr. (Ed.), *Minnesota Studies in the Philosophy of Science - Induction, Probability, and Confirmation* (Vol. VI, pp. 430-486). Minneapolis: University of Minnesota Press.

Westermann, R. & Gerjets, P. (1991). Induktives Denken in Philosophie und Psychologie. In D. Frey und G.M. Koehnken (Ed.), *Bericht über den 37. Kongress der Deutschen Gesellschaft für Psychologie in Kiel 1990* (Vol. 2, pp. 475-486). Kiel: Hogrefe.

Widowski, D. & Eyferth, K. (1986). Comprehending and recalling computer programs of different structural and semantic complexity by experts and novices. In H.P. Willumeit (Ed.), *Human Decision Making and Manual Control*. North-Holland: Elsevier.

Wilensky, R. (1983). *Planning and Understanding*. Reading, Mass: Addison-Wesley.

Winograd, T. (1975). Frame representations and the declarative-procedural controversy. In D.G. Bobrow and A.M. Collins (Ed.), *Representation and Understanding: Studies in Cognitive Science*. New York: Academic Press.

Winston, P.H. (1975). Learning structural descriptions from exmaples. In P. H. Winston (Ed.), *The Psychology of Computer Vision* (pp. 157-210). New York: McGraw-Hill.

Winston, P.H. & Horn, B.K.P. (1989). *LISP*. Reading, Mass: Addison-Wesley.

Wirth, N. (1983). *Algorithmen und Datenstrukturen*. Stuttgart: Teubner.

ANHANG

A Mathematische Grundlagen
A.1 Peano-Axiome und Vollständige Induktion (Kap. 2.1)
A.2 Homomorphismen (Kap. 3.5, Kap. 6.4)

B Die Lernumgebung LEAR
B.1 Interpreter für eine einfache funktionale Sprache (Kap. 5.3)
B.2 Module in LEAR (Kap. 7.4)

C Ausführungen zum Modell
C.1 Spezifikation von Aufgaben (Kap. 6.2)
C.2 Schemaspezifische Strukturen und Regeln (Kap. 6.2, Kap. 6.3)

D Zusatzinformation zu den Untersuchungen
D.1 Beispiele und Erklärungen in Untersuchung I (Kap. 8.2)
D.2 Beispiele in Untersuchung II (Kap. 8.3)
D.3 Abschlußtest (Kap. 8.1)
D.4 Sortieraufgaben (Kap. 8.3)

A.1 Peano-Axiome und Vollständige Induktion (Kap. 2.1)

Die Peano-Axiome definieren die natürlichen Zahlen:

N ist eine Menge, auf der eine Abbildung *SUCC* N -> N *definiert ist und für die folgende* Eigenschaften gelten:

(P1) 0 ist eine natürliche Zahl

$0 \in N$

(P2) Jede natürliche Zahl hat genau einen direkten Nachfolger

$\forall n \in N$ gilt $\exists!$ SUCC(n)

$\forall n,m \in N$ mit $n \neq m$ gilt SUCC(n) $\neq$ SUCC(m)

(P3) Der Nachfolger jeder natürlichen Zahl *ist von 0 verschieden*

$\forall n \in N$ gilt SUCC(n) $\neq 0$

(P4) Wenn die Nachfolger zweier natürlicher Zahlen gleich sind, so sind auch die beiden Zahlen *gleich*

$\forall n \in N$ gilt SUCC(n) = SUCC(m) => n=m

(P5) Jede Teilmenge natürlicher Zahlen, die null enthält und abgeschlossen bezüglich der Nachfolgeroperation ist, ist identisch mit der Menge der natürlichen Zahlen.

$\forall S \subseteq N$ mit

$0 \in S$ *und*

$n \in S$ => SUCC(n) $\in S$

gilt S = N

Axiome P1 bis P4 definieren, daß die Struktur der natürlichen Zahlen linear mit einem festen Anfangselement ist:

0 --SUCC--> 1 --SUCC--> 2 --SUCC--> 3 --SUCC--> 4 --> ...

Das Axiom P5 heißt auch *Induktionsaxiom:*

Sei S eine Menge natürlicher Zahlen, für die Eigenschaft E gilt: $S_E = \{n \in \mathbb{N} \mid E(n)\}$

$0 \in S_E$ entspricht der Induktionsverankerung "E(0) ist gültig"

$n \in S_E \Rightarrow SUCC(n) \in S_E$ entspricht dem Induktionsschritt "E(n) => E(SUCC(n))"

Die Schlußfolgerung "E(n) ist gültig für alle $n \in \mathbb{N}$" entspricht $S_E = \mathbb{N}$.

Das Induktionsprinzip ist auch auf nicht-lineare Strukturen anwendbar, für die die Regeln zur Erzeugung (Signatur bzw. Spezifikation) gegeben sind. Solche induktiven Beweise heißen strukturelle Induktion.

Zur Veranschaulichung der vollständigen Induktion sei noch ein kleines Beispiel angegeben:

Behauptung: für alle natürlichen Zahlen n gilt:

$$\sum_{i=0}^{n} i = \frac{n \times (n+1)}{2}$$

Annahme: für eine beliebige aber feste natürliche Zahl k gilt:

$$\sum_{i=0}^{k} i = \frac{k \times (k+1)}{2}$$

Verankerung für n=0 und n=1:

$$\sum_{i=0}^{0} i = 0 = \frac{0 \times (0+1)}{2}$$

$$\sum_{i=0}^{1} i = 1 = \frac{1 \times (1+1)}{2}$$

Induktionsschritt: Schluß von einer beliebigen aber festen nat. Zahl k nach SUCC(k):

$$\sum_{i=0}^{k+1} i = \sum_{i=0}^{k} i + (k+1)$$

$$= \frac{k \times (k+1)}{2} + (k+1)$$

$$= \frac{k \times (k+1) + 2 \times (k+1)}{2}$$

$$= \frac{(k+1) \times (k+2)}{2}$$

$$= \frac{(k+1) \times ((k+1)+1)}{2} \quad \blacksquare$$

A.2 Homomorphismen (Kap. 3.5, Kap. 6.4)

Das Konzept der Homomorphismen ist fundamental für die Algebra.

Eine Algebra ist eine Menge zusammen mit einer Familie von Operationen. So bildet zum Beispiel die Menge der natürlichen Zahlen mit der Addition eine Algebra (N, +). Allgemein bilden zum Beispiel Halbgruppen (eine Menge mit einer assoziativen Operation) eine Algebra $(A, *_A)$.

Die (endliche) syntaktische Repräsentation einer Algebra heißt Signatur beziehungsweise Spezifikation. Eine Signatur besteht aus der Angabe von Sortensymbolen und Operationssymbolen. Eine Spezifikation gibt zusätzlich "Gleichungen" (genauer syntaktische Regeln der Termersetzung) für die Operationssymbole an.

So ist zum Beispiel die Spezifikation für eine Halbgruppe:

```
semigroup =
  sorten: s
  opns  : *: s s -> s
  eqns  : m1,m2,m3 ϵ s                    ; m1,m2,m3 sind Variablen zur Sorte s
          (m1*m2)*m3 = m1*(m2*m3).
```

Ein Homomorphismus ist eine komponentenweise Abbildung der Mengen einer Algebra auf die Mengen einer anderen Algebra, wobei gelten muß, daß der Homomorphismus kompatibel mit den Operationen beider Algebren sein muß:

> Seien A und B Algebren. Dann heißt f: A → B Homomorphismus, wenn
>
> f eine sortenindizierte Familie von Abbildungen mit ist und folgende Eigenschaften hat:
>
> (1) für alle Konstantensymbole N: → s gilt $f_s(N_A)=N_B$
>
> (2) *für alle Operationssymbole* N: $s_1 \ldots s_n \rightarrow s$ und Elemente $a_i \epsilon A_{si}$, i=1..n gilt
>
> $f_s(N_A(a_1..a_n)) = N_B(f_{s1}(a_1)...f_{sn}(a_n))$.

Anmerkungen:
- "sortenindizierte Familie von Abbildungen" bedeutet, daß f aus einer Menge von Abbildungen f_{si} besteht, für alle Sortensymbole, die die Signatur der Algebren enthält.
- Konstantensymbole sind "Operationssymbole ohne Argumentbereich", von der "leeren Sorte" in die Zielsorte.

Veranschaulicht bedeutet Bedingung 1, daß Konstanten einer Algebra auf entsprechende Konstanten der anderen Algebra abgebildet werden und Bedingung 2, daß es keine Rolle spielt, ob die Operationen in Algebra *A* oder Algebra *B* ausgeführt werden.

Betrachten wir ein Beispiel:

Seien A und B Algebren zur Spezifikation semigroup mit:

	A	B
s	N	{true,false}
*	+	<=>

Prüfen wir, ob die Algebren homomorph sind:

Hierzu definieren wir eine Abbildung f: A → B

Wir können alle geraden Zahlen (inklusive 0) aus N auf true und alle ungeraden Zahlen auf false abbilden:

$$f(x) = \begin{cases} true & ;wenn\ x\ gerade \\ false & ;wenn\ x\ ungerade \end{cases}$$

Nun müssen wir prüfen, ob *f* kompatibel mit + und <=> ist. <=> ist die Biimplikation mit A <=> B ist wahr, wenn A und B wahr sind oder A und B falsch sind und falsch, wenn A und B unterschiedliche Wahrheitswerte besitzen. Auch die Biimplikation wird im folgenden mit Präfixnotation <=> *(A,B)* geschrieben.

$$f(+(x,y)) \overset{?}{=} <=>(f(x),f(y))$$

Seien x und y ϵ N gerade, dann ist auch +(x,y) gerade, also bildet f die Summe auf true ab.

Auch f(x) und f(y) bildet beide Komponenten jeweils auf true ab und die Operation <=> liefert true für zwei wahre Argumente.

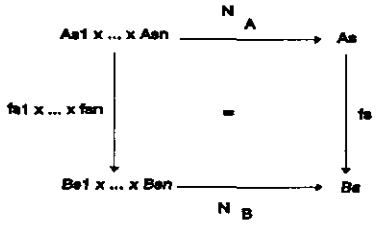
Seien x und y ϵ N ungerade, dann ist +(x,y) gerade, also bildet f die Summe auf true ab.
f(x) und f(y) bildet beide Komponenten jeweils auf false ab und die Operation <=> liefert true für zwei falsche Argumente.

Sei x gerade und y ungerade, dann ist +(x,y) ungerade, also bildet f die Summe auf false ab. Dasselbe gilt für x ungerade und y gerade.
f(x) bildet x auf true ab und f(y) bildet y auf false ab. Die Operation <=> liefert false für zwei Argumente mit unterschiedlichen Wahrheitswerten.

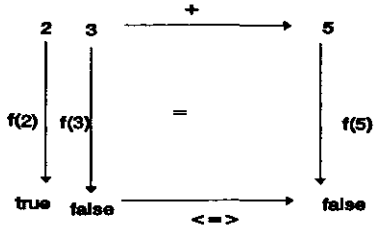
Also ist f: A -> B ein Homomorphismus!

Zudem ist f surjektiv (Epimorphismus), da es zu jedem Element a aus Algebra B jeweils mindestens ein Element x aus Algebra A gibt, für die gilt f(x) = a.

Homomorphismen kann man sich gut in Diagrammen veranschaulichen:



Für ein konkretes Beispiel für den oben dargestellten Homomorphismus ergibt sich:



B.1 Interpreter für eine einfache funktionale Sprache (Kap. 5.3)

Der Interpreter für die in Kapitel 5.3 beschriebene funktionale Sprache ist ein klassischer *eval-apply* Interpreter. Der Interpreter arbeitet auf der als abstrakter Syntaxbaum dargestellten programmierten Funktion.

Der Interpreter ist in TURBO-PASCAL (Version 5.5) programmiert. Die folgenden Ausführungen sind teilweise abstrakt und teilweise als PASCAL-Code angegeben.

Die Datenstruktur für Funktionen ist folgendermaßen aufgebaut:

```
ausdruck =  funktion     : (kopf:deklaration,rumpf:ausdruck) |
            konstante    : (c:name)
            konditional  : (bedingung:ausdruck, dann-anweisung: ausdruck,
                            sonst-anweisung: ausdruck)
            applikation  : (f:name, argumente:argumentliste)

deklaration : (identifyer: name, parameter: parameterliste)
```

Eine korrekter Ausdruck der funktionalen Sprache ist entweder eine selbstdefinierte Funktion, oder eine Konstante, oder eine bedingte Anweisung oder eine Applikation, deren Argumente wieder beliebige korrekte Ausdrücke sein können. Eine selbstdefinierte Funktion besteht aus Kopf und Rumpf. Der Kopf besteht aus der Angabe eines Namens für die Funktion und einer Liste von Parametern, wobei jeder Parameter als Tupel aus Name und Datentyp repräsentiert wird.

Die Datenstruktur ist als Variantenrekord realisiert, wobei jeder Ausdruck der Sprache als Zeigerstruktur repräsentiert wird. Da es sich um eine typisierte Sprache handelt, wird der Ergebnistyp jedes Ausdrucks ebenfalls mitnotiert. Um Fehlerpositionen im Editor anzeigen zu können, werden für jeden Ausdruck die Zeilen- und Spaltenkoordinaten mitangegeben.

Die Datenstruktur ist auf folgende Weise in PASCAL realisiert:

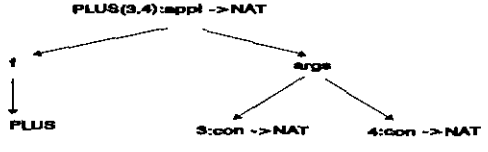
```
nampos    = RECORD
              n   : STRING[10];
              x,y: INTEGER;
            END;

(* --- Funktionskopf ---  *)

parlist   = ^para;      (* Liste der Parameter *)

decl      = RECORD      (* Funktionskopf *)
              fid : nampos;    (* F.-Name   *)
              paras: parlist; (* Parameter *)
            END;

pstruc    = RECORD      (* Struktur eines Param. *)
              pnam: nampos;    (* P.-Name   *)
              ptyp: nampos;    (* Datentyp  *)
            END;
```

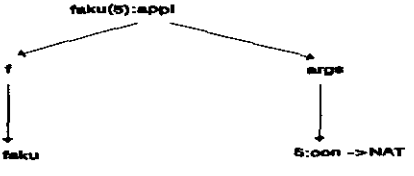
```
para      =  RECORD
               pa : pstruc;     (* Erster P. *)
               rpa: parlist;    (* Nachfolger*)
             END;

(* --- Ausdrücke der funktionalen Sprache --- *)

arglist   =  ^arg;          (* Liste der Argumente *)

arg       =  RECORD
               ex : expr;       (* Erstes A. *)
               rex: arglist;    (* Nachfolger*)
             END;

exprkind  =  (fun, con, cond, appl);
expr      =  ^exp;
exp       =  RECORD
               restyp: nampos; (* Ergebnistyp *)
               CASE kind: exprkind OF
                 fun : (head:decl; body:expr;);
                 con : (c: nampos;);
                 cond: (bed: expr; alt1: expr; alt2: expr;);
                 appl : (f:nampos; args: arglist;);
               END;
```

Wird nun im Interpreter ein Ausdruck eingegeben, zum Beispiel:

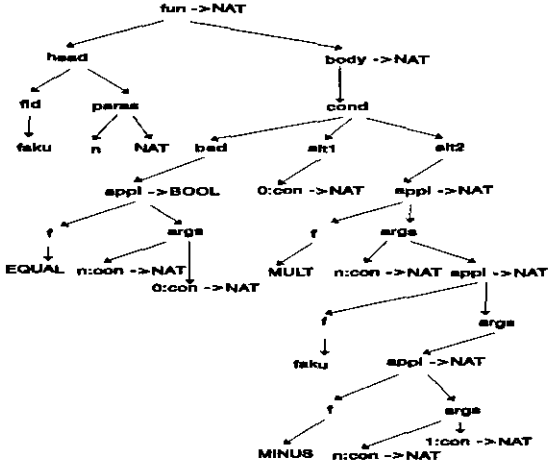
```
> PLUS(3,4)
> faku(5)
```

so wird dieser Ausdruck parsiert und dadurch zum Beispiel folgende Syntaxbäume erzeugt:





Die Operation PLUS ist in der Sprache vordefiniert, *faku* ist eine selbstdefinierte Funktion, für die folgender abstrakter Syntaxbaum erzeugt wird:



Der eval-apply-Interpreter arbeitet nun folgendermaßen auf den abstrakten Syntaxbäumen:

1) eval:

In Abhängigkeit von der Sorte des Ausdrucks (Konstante, Konditional, Applikation) wird entweder ein Ergebnis ausgegeben, die Evaluation von Teilausdrücken vorgenommen oder *apply* aufgerufen.

```
FUNCTION eval(e:expr):expr; (* Hier nur Kern der Funktion,
VAR                           ohne Fehlerbehandlung etc. *)
  aev: arglist;
BEGIN
  CASE e^.kind OF
  con  : eval := e; (* Konstanten werden ausgegeben *)
  cond : IF eval(e^.bed)^.c.n = 'TRUE'
         THEN
           BEGIN
             eval := eval (e^.alt1); (* Auswertung der THEN-Anweisung*)
             dispExpr (e^.alt2); e^.alt2 := NIL; (* Wegschneiden der
           END                                    ELSE-Anweisung aus
         ELSE                                     dem Syntaxbaum *)
           BEGIN
             eval := eval (e^.alt2); (* Auswertung der ELSE-Anweisung*)
             dispExpr (e^.alt2); e^.alt2 := NIL; (* Wegschneiden der
           END;                                     THEN-Anweisung aus
  appl: BEGIN                                     dem Syntaxbaum *)
          aev := evalArgs (e^.args); (* Evaluation der Argumete: eval
                                        wird für jedes Argument wieder
                                        aufgerufen *)
          eval := apply(e^.f.n,aev); (* Anwendung der Funktion f auf die
        END;                            Argumente aev *)
  END; (* CASE *)
END;
```

Bei Applikationen wird für jedes Argument wieder *eval* aufgerufen, so daß die Argumente für *apply* jeweils nur noch Konstanten sein können.

2) apply:

Alle vordefinierten Ausdrücke (NOT, AND, OR, EQUAL, GREATER, EMPTY, PLUS, *MINUS, MULT, HEAD, TAIL, CONS*) werden *mit den (konstanten) Argumenten* ausgewertet. Dabei wird für jeden Ausdruck eine spezifische Auswertungsregel vorgegeben.

Für selbstdefinierte Funktionen (wie *faku*) werden die Argumente, mit denen sie aufgerufen wurden, in den abstrakten Syntaxbaum an die Stelle der Parameter kopiert und dann wird der Rumpf der Funktion an *eval* übergeben.

Die *eval-apply*-Interpreatation soll nun noch an einem Beispiel veranschaulicht werden, dabei werden Konstanten vor der Evaluation mit 'x markiert, um sie von den evaluierten Termen zu unterscheiden.

```
eval (PLUS('3, MINUS('4,'1))) ->
apply (PLUS (eval('3), eval(MINUS('4,'1))) ->
apply (PLUS (3, apply (MINUS (eval('4), eval('1))))) ->
apply (PLUS (3, apply (MINUS (4,1)))) ->
apply (PLUS (3, 4-1)) ->
apply (PLUS (3,3)) ->
3+3 ->
6
```

B.2 Module in LEAR (Kap. 7.4)

Die Lernumgebung LEAR wurde in TURBO-PASCAL (Version 5.5) programmiert[1] und besteht aus folgenden Modulen[2]:

- ADDHELP: Interaktive Einführung in die Bedienung von LEAR sowie interaktives Frage-Antwort-Programm zur Syntax der funktionalen Sprache
- DATSTRUC: Definition der zentralen Konstanten und Typen (Funktionen als Text, als *Token*-Struktur, als abstrakte Syntax)
- EDITOR: Zeileneditor
- ERRORMES: Fehlermeldungen und Curserpositionierung auf im Editor lokalisierte Fehlerstelle (wird von EDITOR, PARSEFUN, TYPFUN und SIMINT benutzt)
- HASHTABL: Speicherung der *Tokens* (Elementarsymbole) von Funktionen in einer Tabelle (wird von SCANFUN benutzt)
- HELPTEXT: Ausgabe der Lernhilfen
- INTERP: *eval-apply*-Interpreter
- PARSINT: Start der Parsierung von Interpretereingaben
- PARSEFUN: Parser (mit Syntaxcheck)
- PROTO: *log-file*-Protokollierung
- SCANFUN: Zerlegt Funktion von *Strings* in *Tokens*
- SIMINT: interne Interpretation von Funktionen mit kritischen Testwerten
- STEUER: Versuchssteuerung (Auslesen der Informationen über aktuellen probanden: Lektion, Lernbedingung und Start bzw. Ende der *log-file*-Protokollierung)
- TIMER: Verwaltung der DOS-Uhr zur Zeitprotokollierung und Zeitmessung
- TYPFUN: Typisierung einer Funktion (mit Fehler-Check)
- TYPING: Start der Typisierung und Verwaltung der Fehlermeldungen
- WINDOWS: Fensterverwaltung

Das Hauptprogramm LEAR kontrollierten den Ablauf des Lernprogramms, so wie es im Flußdiagramm (*Abb. 14) in Kapitel 7.4 dargestellt wurde.*
Zusätzlich wurde ein Programm SIMLEAR erstellt, das ermöglicht, die *log-files* (in Echtzeit) ablaufen zu lassen.

---

[1]Das Programm ist bei der Autorin erhältlich.

[2]Die Module HASHTABL, PROTO und TYPING wurden von Guido Dunker erstellt, der auch wesentliche Anregungen für die Konzeption der Datenstruktur und des Interpreters gab.

Das Programm und seine Module benutzen zusätzlich folgende Textdateien:

- aufg.txt: Aufgabenstellungen
- funcs.txt: korrekte Funktionen
- lsg.txt: Eingabewerte für simulierte Interpretation
- werte.txt: Vorschläge für Testwerte zum Aufruf der Funktionen
- bsp.txt: Beispiele für Untersuchung I
- erk.txt: Erklärungen für Untersuchung I
- bsp_a_x.txt: Beispiele für Untersuchung II (a = mit oder ohne Mapping; x= Ähnlichkeit I bis V).

C.1 Spezifikation von Aufgaben und Beispielen (Kap. 6.2)

Die natürlichsprachigen Aufgabenstellungen werden durch abstraktere Spezifikationen, die die Informationen der Aufgabenstellung erhalten, repräsentiert.

Die Spezifikation wird in einem *3*-stelligen Schema repräsentiert, wobei die erste Stelle den Namen der Funktion angibt, die zweite Stelle Namen und Typen der Parameter und die dritte Stelle die gewünschte Ausgabe zusammen mit dem Ergebnistyp der Funktion.

Schema Aufgabe.Semantik:
fname : name
finput : $\{\text{parametername}_i{:}\text{parametertyp}_i\}_{i=1..n}$
foutput : [ ]: ergebnistyp

Zur Spezifikation der Ausgabe werden dabei folgende Schlüsselwörter verwendet:

count(x) : Markiert eine Zählvariable x
all(x) : gibt an, daß eine Anweisung für alle x durchgeführt werden soll
range(i,j) : gibt für *all* die Unter- und Obergrenzen an
cond(a < b) : gibt eine Bedingung an
length(l) : gibt die Länge einer Liste an.

Listenzerlegungen werden mit l = [x|l'] etc. notiert.

Der Einfachheit halber werden für die Operationen in der Spezifikation direkt die Schlüsselwörter der einfachen funktionalen Sprache übernommen.

Schema Aufgabe.Semantik:
fname : dubx
finput : z:NAT, x:NAT
foutput : [count(x): PLUS(z,z); dubx(0):z]: NAT

Schema Aufgabe.Semantik:
fname : last
finput : l:NATLIST
foutput : [x | l = [l'|x]: NAT

Schema Aufgabe.Semantik:
fname : member
finput : n:NAT, l:NATLIST
foutput : [cond(n= x ∈ all(l): 1; 0]: NAT

Schema Aufgabe.Semantik:
fname : summe
finput : n:NAT
foutput : [PLUS(all(n), range(n,0)]: NAT

Schema Aufgabe.Semantik:
fname : genlist
finput : n:NAT
foutput : [CONS(all(n), range(n,1)]: NAT

Schema Aufgabe.Semantik:
fname : suml
finput : l:NATLIST
foutput : [CONS(all (PLUS(x,y) | l = [x|y|l']]: NATLIST

C.2 Schemaspezifische Strukturen und Regeln (Kap. 6.2, 6.3)

Im folgenden werden die Schemastrukturen und Regeln für folgende Rekursionstypen angegeben:

- allgemein (vgl. Kap. 6.2)
- Endrekursion

und zur Veranschaulichung, wie sich die Strukturen immer weiter vereinfachen und immer stärker vorinstantiiert sind:

- Teil-Rest-Rekursion, Teil=Wert
- Teil-Rest-Rekursion, Teil=Wert, NAT

Es ist anzumerken, daß für die im folgenden angegebenen Strukturen keineswegs der Anspruch erhoben wird, daß diese vollständig sind und für jede beliebige Spezifikation ein korrektes Ergebnis liefern. Die Strukturen dienen lediglich zur Demonstration, daß mit der vorgeschlagenen integrierten Schema-Regel-Struktur prinzipiell die Fähigkeit, rekursive Aufgaben zu programmieren, modellierbar ist.

Schema: Rekursion.allgemein
Kopf: FUN *name* (*parlist* = $\{p_i{:}t_i\}_{i=1..n}$):t_{res}
Rumpf:
Abbruchbedingung: IF *(r=bottom)*
direkter Fall: THEN *wert*
rekursiver Fall: ELSE *verknüpfer(hilfsop*($\{p_i\}_{i\le n}$,*name*([{*hilfsop*$_i$(p_i)}$_{\forall i \text{ mit } p_i \neq r}$, *minop(r)*])) FIN

mit
{} = 1 bis n Elemente
[] = enthält Ausdrücke in möglicherweise anderer Reihenfolge
name = Name der Funktion
parlist = Liste von Paramtern
p_i = Name eines Paramters
t_i = Datentyp eines Parameters
t_{res} = Ergebnistyp
r = zu minimierender Parameter
bottom = kleinstes Teilproblem
wert = Ergebnis im direkten Fall
verknüpfer = Operation, die Ergebnis eines Ausdrucks und des rekursiven Aufrufs verknüpft
hilfsop = beliebige Operation
minop = Minimierungsoperation mit bottom $\le$ minop(r) $<$ r

Kursiv gedruckte Ausdrücke sind Platzhalter, die zu instantiieren sind.
Das aktive Schema wird mit Aufgabe.Rekursion bezeichnet.

Die Regeln sind in einem Pseudo-PASCAL-Code notiert, wobei folgende nicht weiter spezifizierte Anweisungen verwendet werden:

Parametername	: Name eines Parameters aus der Spezifikation(Aufgabenstellung)
Parametertyp	: Datentyp einesParameters aus der Spezifikation
Parameterzahl(finput) -> N	: liefert Anzahl der Parameter aus der Spezifikation
Parametername ϵ (X) IN foutput -> p_i	: liefert Name eines Parameters aus der Spezifikation, der Argument von ALL oder COUNT ist
DatenTyp(x) -> {NAT,NATLIST,BOOL}	: liefert Datentyp eines Parameters
IsArgument(ALL,Operation) -> {TRUE, FALSE}	: prüft, ob ALL als Argument einer Operation in der Spezifikation erscheint
isArgument(Operation,verknüpfer) -> {TRUE,FALSE}	: prüft, ob als Argument der Verknüpfungsoperation eine weitere Operation erscheint
GetOperation -> {vordef. Operation}	: liefert die Operation der Spezifikation, die ungleich der Minimierungsoperation ist
Minimum(range) -> N	: liefert kleineren der beiden Werte in range
isGiven(dF) -> {TRUE,FALSE}	: prüft, ob für kleinsten Wert das Funktionsergebnis (dF) direkt angegeben wurde
Calculate(foutput(r=bottom),ok) -> t_{res}	: setzt in der Spezifikation des Funktionsergebnisses für r bottom ein und versucht, das Ergebnis zu errechnen; ok meldet, ob die Berechnung erfolgreich war.

```
R1: Schreibe_Rekursion
IF (Ziel = Schreibe rekursive Funktion)
THEN WITH Aufgabe.Rekursion DO
  verfeinere_Kopf; (* R2 *)
  verfeinere_Rumpf. (* R3 *)

R2: Verfeinere_Kopf
IF (Ziel = Verfeinere Kopf)
THEN WITH Aufgabe. Rekursion DO
  name := Aufgabe.Semantik.fname; (* im Kopf und im rekursiven Aufruf *)
  WITH Aufgabe.Rekursion.Kopf DO
  FOR i := 1 TO n DO
    (pi := Aufgabe.Semantik.finput.parameternamei;
     ti := Aufgabe.Semantik.finput.parametertypi;)
  tres := Aufgabe.Semantik.foutput.ergebnistyp.
```

```
R3: Verfeinere_Rumpf
IF (Ziel = Verfeinere Rumpf)
THEN WITH Aufgabe.Rekursion.Rumpf DO
  verfeinere_rekursiven_Fall; (* R4 *)
  verfeinere_direkten_Fall. (* R9 *)

R4: Verfeinere_rekursiven_Fall
IF (Ziel = Verfeinere rekursiven Fall)
THEN WITH Aufgabe.Rekursion.rekursiver_Fall DO
  finde_rekursiven_Parameter; (* R5 *)
  finde_Minimierung(r); (* R6 *)
  finde_Verknüpfer(t_res); (* R7 *)
  finde_Operation. (* R8 *)

R5: Finde_rekursiven_Parameter
IF (Ziel = Finde rekursiven Parameter)
THEN WITH Aufgabe.Semantik DO WITH Aufgabe.Rekursion DO
  IF Parameterzahl(finput) = 1
  THEN r := Parametername;
  IF Paramterzahl(finput) > 1
  THEN r := Parametername ∈ (ALL OR COUNT) IN foutput.

R6: Finde_Minimierung
IF (Ziel = Finde Minimierung(r))
THEN WITH Aufgabe.Rekursion.rekursiver_Fall DO
  IF DatenTyp(r) = NAT
  THEN minop := MINUS(r,x) mit 0 < x ≤ r; default x = 1;
  IF DatenTyp(r) = NATLIST
  THEN minop := TAIL^x(r) mit 0 < x ≤ Length(r); default x = 1.


R7: Finde_Verknüpfer
IF (Ziel = Finde Verknüpfer)
THEN WITH Aufgabe.Semantik.foutput DO
  IF IsArgument(ALL,Operation)
  THEN Aufgabe.Rekursion.rekursiver_Fall.verknüpfer := Operation
  ELSE verknüpfer := [].

R8: Finde_Operation
IF (Ziel = Finde Operation)
THEN WITH Aufgabe.Semantik.foutput DO WITH Aufgabe.Rekursion.rekursiver_Fall DO
  IF verknüpfer ≠ []
  THEN
    IF isArgument(Operation,verknüpfer)
    THEN hilfsop := Operation
    ELSE hilfsop := id(r);
  IF verknüpfer = []
  THEN
    IF Parameterzahl(Aufgabe.Semantik.finput) > 1
    THEN hilfsop := GetOperation
    ELSE hilfsop := [].
```

```
R9: Verfeinere_direkten_Fall
IF (Ziel = Verfeinere direkten Fall)
THEN WITH Aufgabe.Rekursion.Rumpf DO
  finde_kleinstes_Teilproblem; (* R10 *)
  finde_Prüfbedingung(bottom); (* R11 *)
  finde_Ergebnis(bottom). (* R12 *)

R10: Finde_kleinstes_Teilproblem
IF (Ziel = Finde kleinstes Teilproblem)
THEN WITH Abbruchbedingung DO
  IF DatenTyp(r) = NAT
  THEN bottom := Minimum(Aufgabe.Semantik.foutput.range); default = 0
  IF DatenTyp(r) = NATLIST
  THEN WITH Aufgabe.Semantik.foutput DO
  IF ALL(r) AND hilfsop(r) ≠ TAIL
  THEN bottom := 0;
  IF ALL(r) AND hilfsop(r) = TAIL
  THEN bottom := 1;
  IF verknüpfer = [] AND minop = TAIL^x
  THEN bottom := Length(wert).

R11: Finde_Prüfbedingung
IF (Ziel = Finde Prüfbedingung (bottom))
THEN WITH Aufgabe.Rekursion.Abbruchbedingung
  IF DatenTyp(r) = NAT
  THEN Abbruchbedingung := EQUAL(r,bottom);
  IF DatenTyp(r) = NATLIST
  THEN Abbruchbedingung := EMPTY(TAIL"(r)) mit "=bottom.

R12: Finde_Ergebnis
IF (Ziel = Finde Ergebnis (bottom))
THEN WITH Aufgabe.Rekursion.direkter_Fall DO
  IF isGiven(dF IN Aufgabe.Semantik.foutput)
  THEN wert := dF
  ELSE wert := Calculate(Aufgabe.Semantik.foutput(r=bottom),ok)
    IF NOT ok
    THEN
    IF t_res = NAT THEN wert := 0
    IF t_res = NATLIST THEN wert := NIL.
```

Schema: Endrekursion

Kopf: FUN *name* (*parlist* = $\{p_i{:}t_i\}_{i=1..n}$):t_{res}

Rumpf:

Abbruchbedingung: IF *(r=bottom)*

direkter Fall: THEN *wert*

rekursiver Fall: ELSE name([{hilfsop$_i$(p_i)}$_{\forall i \text{ mit } p_i \neq r}$, *minop(r)*])) FIN

```
R1: Schreibe_Rekursion
IF (Ziel = Schreibe rekursive Funktion)
THEN WITH Aufgabe.Rekursion DO
  verfeinere_Kopf; (* R2 *)
  verfeinere_Rumpf. (* R3 *)

R2: Verfeinere_Kopf
IF (Ziel = Verfeinere Kopf)
THEN WITH Aufgabe. Rekursion DO
  name := Aufgabe.Semantik.fname; (* im Kopf und im rekursiven Aufruf *)
  WITH Aufgabe.Rekursion.Kopf DO
  FOR i := 1 TO n DO
    (p_i := Aufgabe.Semantik.finput.parameternamei;
     t_i := Aufgabe.Semantik.finput.parametertypi;)
  tres := Aufgabe.Semantik.foutput.ergebnistyp.

R3: Verfeinere_Rumpf
IF (Ziel = Verfeinere Rumpf)
THEN WITH Aufgabe.Rekursion.Rumpf DO
  verfeinere_rekursiven_Fall; (* R4 *)
  verfeinere_direkten_Fall. (* R9 *)

R4.end: Verfeinere_rekursiven_Fall
IF (Ziel = Verfeinere rekursiven Fall)
THEN WITH Aufgabe.Rekursion.rekursiver_Fall DO
  finde_rekursiven_Parameter; (* R5 *)
  finde_Minimierung(r); (* R6 *)
  finde_Operation. (* R8 *)

R5: Finde_rekursiven_Parameter
IF (Ziel = Finde rekursiven Parameter)
THEN WITH Aufgabe.Semantik DO WITH Aufgabe.Rekursion DO
  IF Parameterzahl(finput) = 1
  THEN r := Parametername;
  IF Paramterzahl(finput) > 1
  THEN r := Parametername ∈ (ALL OR COUNT) IN foutput.

R6: Finde_Minimierung
IF (Ziel = Finde Minimierung(r))
THEN WITH Aufgabe.Rekursion.rekursiver_Fall DO
  IF DatenTyp(r) = NAT
  THEN minop := MINUS(r,x) mit 0 < x ≤ r; default x = 1;
  IF DatenTyp(r) = NATLIST
  THEN minop := TAIL^x(r) mit 0 < x ≤ Length(r); default x = 1.
```

```
R8.end: Finde_Operation
IF (Ziel = Finde Operation)
THEN WITH Aufgabe.Semantik.foutput DO WITH Aufgabe.Rekursion.rekursiver_Fall DO
    IF Parameterzahl(Aufgabe.Semantik.finput) > 1
    THEN hilfsop := GetOperation
    ELSE hilfsop := ∏.

R9: Verfeinere_direkten_Fall
IF (Ziel = Verfeinere direkten Fall)
THEN WITH Aufgabe.Rekursion.Rumpf DO
  finde_kleinstes_Teilproblem; (* R10 *)
  finde_Prüfbedingung(bottom); (* R11 *)
  finde_Ergebnis(bottom). (* R12 *)

R10.end: Finde_kleinstes_Teilproblem
IF (Ziel = Finde kleinstes Teilproblem)
THEN WITH Abbruchbedingung DO
  IF DatenTyp(r) = NAT
  THEN bottom := Minimum(Aufgabe.Semantik.foutput.range); default = 0
  IF DatenTyp(r) = NATLIST
  THEN WITH Aufgabe.Semantik.foutput DO
  IF ALL(r) AND hilfsop(r) ≠ TAIL
  THEN bottom := 0;
  IF ALL(r) AND hilfsop(r) = TAIL
  THEN bottom := 1;
  IF minop = TAIL*
  THEN bottom := Length(wert).

R11: Finde_Prüfbedingung
IF (Ziel = Finde Prüfbedingung (bottom))
THEN WITH Aufgabe.Rekursion.Abbruchbedingung
  IF DatenTyp(r) = NAT
  THEN Abbruchbedingung := EQUAL(r,bottom);
  IF DatenTyp(r) = NATLIST
  THEN Abbruchbedingung := EMPTY(TAIL*(r)) mit *=bottom.

R12: Finde_Ergebnis
IF (Ziel = Finde Ergebnis (bottom))
THEN WITH Aufgabe.Rekursion.direkter_Fall DO
  IF isGiven(dF IN Aufgabe.Semantik.foutput)
  THEN wert := dF
  ELSE wert := Calculate(Aufgabe.Semantik.foutput(r=bottom),ok)
    IF NOT ok
    THEN
    IF t_res = NAT THEN wert := 0
    IF t_res = NATLIST THEN wert := NIL.
```

Schema: Teil-Rest-Rekursion, Teil = Wert

Kopf: FUN *name* (*parlist* = $\{p_i:t_i\}_{i=1..n}$):t_{res}

Rumpf:

Abbruchbedingung: IF *(r=bottom)*

direkter Fall: THEN *wert*

rekursiver Fall: ELSE *verknüpfer(hilfsop($\{p_i\}_{i\le n}$, name($\{\{pi\}_{\forall i \text{ mit } pi \neq r}$, minop(r)})*) FIN

R1: Schreibe_Rekursion
IF (Ziel = Schreibe rekursive Funktion)
THEN WITH Aufgabe.Rekursion DO
  verfeinere_Kopf; (* R2 *)
  verfeinere_Rumpf. (* R3 *)

R2: Verfeinere_Kopf
IF (Ziel = Verfeinere Kopf)
THEN WITH Aufgabe. Rekursion DO
  name := Aufgabe.Semantik.fname; (* im Kopf und im rekursiven Aufruf *)
  WITH Aufgabe.Rekursion.Kopf DO
  FOR i := 1 TO n DO
    (p_i := Aufgabe.Semantik.finput.parameternamei;
    t_i := Aufgabe.Semantik.finput.parametertypi;)
  tres := Aufgabe.Semantik.foutput.ergebnistyp.

R3: Verfeinere_Rumpf
IF (Ziel = Verfeinere Rumpf)
THEN WITH Aufgabe.Rekursion.Rumpf DO
  verfeinere_rekursiven_Fall; (* R4 *)
  verfeinere_direkten_Fall. (* R9 *)

R4: Verfeinere_rekursiven_Fall
IF (Ziel = Verfeinere rekursiven Fall)
THEN WITH Aufgabe.Rekursion.rekursiver_Fall DO
  finde_rekursiven_Parameter; (* R5 *)
  finde_Minimierung(r); (* R6 *)
  finde_Verknüpfer(t_{res}); (* R7 *)
  finde_Operation. (* R8 *)

R5: Finde_rekursiven_Parameter
IF (Ziel = Finde rekursiven Parameter)
THEN WITH Aufgabe.Semantik DO WITH Aufgabe.Rekursion DO
  IF Parameterzahl(finput) = 1
  THEN r := Parametername;
  IF Paramterzahl(finput) > 1
  THEN r := Parametername ϵ (ALL OR COUNT) IN foutput.

R6: Finde_Minimierung
IF (Ziel = Finde Minimierung(r))
THEN WITH Aufgabe.Rekursion.rekursiver_Fall DO
  IF DatenTyp(r) = NAT
  THEN minop := MINUS(r,x) mit $0 < x \le r$; default x = 1;
  IF DatenTyp(r) = NATLIST
  THEN minop := $TAIL^x$(r) mit $0 < x \le$ Length(r); default x = 1.

```
R7.teil-rest: Finde_Verknüpfer
IF (Ziel = Finde Verknüpfer)
THEN WITH Aufgabe.Semantik.foutput DO
  IF IsArgument(ALL,Operation)
  THEN Aufgabe.Rekursion.rekursiver_Fall.verknüpfer := Operation.

R8.teil=wert: Finde_Operation
IF (Ziel = Finde Operation)
THEN hilfsop := id(r).

R9: Verfeinere_direkten_Fall
IF (Ziel = Verfeinere direkten Fall)
THEN WITH Aufgabe.Rekursion.Rumpf DO
  finde_kleinstes_Teilproblem; (* R10 *)
  finde_Prüfbedingung(bottom); (* R11 *)
  finde_Ergebnis(bottom). (* R12 *)

R10.teil-rest.teil=wert: Finde_kleinstes_Teilproblem
IF (Ziel = Finde kleinstes Teilproblem)
THEN WITH Abbruchbedingung DO
  IF DatenTyp(r) = NAT
  THEN bottom := Minimum(Aufgabe.Semantik.foutput.range); default = 0
  IF DatenTyp(r) = NATLIST
  THEN WITH Aufgabe.Semantik.foutput DO bottom := 0.

R11: Finde_Prüfbedingung
IF (Ziel = Finde Prüfbedingung (bottom))
THEN WITH Aufgabe.Rekursion.Abbruchbedingung
  IF DatenTyp(r) = NAT
  THEN Abbruchbedingung := EQUAL(r,bottom);
  IF DatenTyp(r) = NATLIST
  THEN Abbruchbedingung := EMPTY(TAIL*(r)) mit *=bottom.

R12: Finde_Ergebnis
IF (Ziel = Finde Ergebnis (bottom))
THEN WITH Aufgabe.Rekursion.direkter_Fall DO
  IF isGiven(dF IN Aufgabe.Semantik.foutput)
  THEN wert := dF
  ELSE wert := Calculate(Aufgabe.Semantik.foutput(r=bottom),ok)
    IF NOT ok
    THEN
    IF t_res = NAT THEN wert := 0
    IF t_res = NATLIST THEN wert := NIL.
```

Schema: Teil-Rest-Rekursion, Teil = Wert, nur auf Datentyp NAT
Kopf: FUN *name* (*parlist* = $\{p_i:NAT\}_{i=1..n}$):NAT
Rumpf:
Abbruchbedingung: IF EQUAL(*r,bottom*)
direkter Fall: THEN *wert*
rekursiver Fall: ELSE *verknüpfer*(*hilfsop*($\{p_i\}_{i\le n}$,*name*([$\{pi\}_{\forall i \text{ mit } pi \neq r}$, *minop*(r)])) FIN

```
R1: Schreibe_Rekursion
IF (Ziel = Schreibe rekursive Funktion)
THEN WITH Aufgabe.Rekursion DO
  verfeinere_Kopf; (* R2 *)
  verfeinere_Rumpf. (* R3 *)

R2.NAT: Verfeinere_Kopf
IF (Ziel = Verfeinere Kopf)
THEN WITH Aufgabe. Rekursion DO
  name := Aufgabe.Semantik.fname; (* im Kopf und im rekursiven Aufruf *)
  WITH Aufgabe.Rekursion.Kopf DO
  FOR i := 1 TO n DO
    (p_i := Aufgabe.Semantik.finput.parameternamei)
  tres := Aufgabe.Semantik.foutput.ergebnistyp.

R3: Verfeinere_Rumpf
IF (Ziel = Verfeinere Rumpf)
THEN WITH Aufgabe.Rekursion.Rumpf DO
  verfeinere_rekursiven_Fall; (* R4 *)
  verfeinere_direkten_Fall. (* R9 *)

R4: Verfeinere_rekursiven_Fall
IF (Ziel = Verfeinere rekursiven Fall)
THEN WITH Aufgabe.Rekursion.rekursiver_Fall DO
  finde_rekursiven_Parameter; (* R5 *)
  finde_Minimierung(r); (* R6 *)
  finde_Verknüpfer(t_m); (* R7 *)
  finde_Operation. (* R8 *)

R5: Finde_rekursiven_Parameter
IF (Ziel = Finde rekursiven Parameter)
THEN WITH Aufgabe.Semantik DO WITH Aufgabe.Rekursion DO
  IF Parameterzahl(finput) = 1
  THEN r := Parametername;
  IF Parameterzahl(finput) > 1
  THEN r := Parametername ∈ (ALL OR COUNT) IN foutput.

R6.NAT: Finde_Minimierung
IF (Ziel = Finde Minimierung(r))
THEN WITH Aufgabe.Rekursion.rekursiver_Fall DO
  minop := MINUS(r,x) mit 0 < x ≤ r; default x = 1.

R7.teil-rest: Finde_Verknüpfer
IF (Ziel = Finde Verknüpfer)
THEN WITH Aufgabe.Semantik.foutput DO
  IF IsArgument(ALL,Operation)
  THEN Aufgabe.Rekursion.rekursiver_Fall.verknüpfer := Operation.
```

```
R8.teil=wert: Finde_Operation
IF (Ziel = Finde Operation)
THEN hilfsop := id(r).

R9: Verfeinere_direkten_Fall
IF (Ziel = Verfeinere direkten Fall)
THEN WITH Aufgabe.Rekursion.Rumpf DO
  finde_kleinstes_Teilproblem; (* R10 *)
  finde_Ergebnis(bottom). (* R12 *)

R10.teil-rest.teil=wert.NAT: Finde_kleinstes_Teilproblem
IF (Ziel = Finde kleinstes Teilproblem)
THEN WITH Abbruchbedingung DO
  IF DatenTyp(r) = NAT
  THEN bottom := Minimum(Aufgabe.Semantik.foutput.range); default = 0.

R12: Finde_Ergebnis
IF (Ziel = Finde Ergebnis (bottom))
THEN WITH Aufgabe.Rekursion.direkter_Fall DO
  IF isGiven(dF IN Aufgabe.Semantik.foutput)
  THEN wert := dF
  ELSE wert := Calculate(Aufgabe.Semantik.foutput(r=bottom),ok)
    IF NOT ok
    THEN
    IF t_res = NAT THEN wert := 0
    IF t_res = NATLIST THEN wert := NIL.
```

D.1 Beispiele und Erklärungen in Untersuchung I (Kap. 8.2)

In Untersuchung I waren pro Aufgabe je zwei Erklärungen bzw. Beispiele abrufbar. Die Kennnummern kodieren Lektion/Aufgabe/Hilfenummer.

Erklärungen

3/1/1
Damit die rekursive Funktion terminiert (d.h. nicht endlos läuft) muß zunächst eine Abbruchbedingung geschaffen werden. z soll x-mal verdoppelt werden. Das heißt, wenn x=0 ist, soll die Zahl z unverdoppelt ausgegeben werden. Im rekursiven Aufruf der Funktion muß x also minimiert werden, damit die Abbruchbedingung erreicht wird. Zusätzlich muß im rekursiven Aufruf z verdoppelt werden.
ACHTUNG: Die Reihenfolge der Parameter muß immer identisch sein: Im rekursiven Aufruf muß also die selbe Reihenfolge verwendet werden, wie im Funktionskopf!!!

3/1/2
Im Kopf der Funktion stehen die Zahlen z und x. Das Ergebnis ist ein Zahlenwert. Mit einer IF-THEN-ELSE-Anweisung wird zunächst die Abbruchbedingung (x=0) geschaffen. Ist x=0, dann wird z unverdoppelt ausgegeben. Sonst -das ist der rekursive Fall- wird die Funktion dubx aufgerufen, wobei z verdoppelt wird und x schrittweise minimiert wird, also z+z und x-1. Dabei muß zuerst das verdoppelte z und dann das minimierte x im Funktionsaufruf stehen!

3/2/1
Um das letze Element einer Liste auszugeben, muß die Liste solange verkürzt werden, bis nur noch ein Element vorhanden ist. Dieses Element wird dann ausgegeben. Als Abbruchbedingung muß also gelten, daß die Liste nur ein Element enthält. Damit diese Bedingung erreicht werden kann, muß die Liste im rekursiven Aufruf minimiert werden.

3/2/2
Im Kopf der Funktion steht eine Liste l. Das Ergebnis ist eine Zahl, nämlich das letze Element der Liste. Mit einer IF-THEN-ELSE-Anweisung wird geprüft, ob die Liste nur ein Element enthält. Dies ist der Fall wenn der Tail der Liste leer ist. In diesem Fall wird das Element, also der Kopf der Liste, ausgegeben. Sonst -das ist der rekursive Fall- wird die Funktion mit der um ein Element erniedrigten Liste, also dem TAIL der Liste, wieder aufgerufen.

3/3/1
Um festzustellen, ob eine Liste l ein bestimmtes Element n enthält, muß die Liste von anfang an durchsucht werden. Das jeweils erste Element der Liste wird auf Gleichheit mit n geprüft. Ist die Gleichheit gegeben, ist die Arbeit der Funktion beendet. Damit die Funktion in jedem Fall terminiert, also auch dann, wenn n nicht in l vorkommt, wird eine zweite Abbruchbedingung benötigt: Wenn die Liste bis zum Ende durchsucht wurde und kein mit n übereinstimmendes Element gefunden wurde, soll die Funktion dies melden.

3/3/2
Im Kopf der Funktion steht eine Zahl n und eine Liste l. Die Funktion liefert einen Wahrheitswert: TRUE, wenn n Element von l ist und FALSE, wenn n kein Element von l ist. Die Funktion benötigt zwei Abbruchbedingungen. Es werden also zwei IF-THEN-ELSE-Anweisungen benötigt. Zuerst muß geprüft werden, ob die Liste l leer ist. Ist dies der Fall, ist n nicht in l, die Funktion liefert also FALSE. Sonst folgt die zweite Bedingung (also ELSE IF ...). Hier wird geprüft, ob n gleich dem Kopf der Liste ist. Ist dies der Fall, gibt die Funktion TRUE aus. Anderenfalls - die Liste ist weder leer noch ist n gleich dem Kopf von l - folgt der rekursive Aufruf: Die Funktion wird mit n und der um ein Element minimierten Liste, also dem TAIL der Liste, aufgerufen.

4/1/1
Im Gegensatz zu den Aufgaben in Lektion 3, wird im dann-Teil nicht das Ergebnis ausgegeben, und im sonst-Teil steht nicht der Aufruf der Funktion mit veränderten Parametern. Stattdessen steht im dann-Teil ein fester Wert, für den Fall, daß die Bedingung wahr ist. Im sonst-Teil steht ein Verknüpfer, z.B. PLUS, der als Parameter einen Wert und den rekursiven Aufruf der Funktion enthält.
Um die Summe aller Zahlen von x bis 0 zu erhalten, muß also 0 ausgegeben werden, wenn x=0 ist. Sonst muß x zum Ergebnis der Funktion für x-1 addiert werden, also x + addz (x-1).

4/1/2
Im Kopf der Funktion steht eine Zahl x. Das Ergebnis ist ein Zahlenwert, die Summe der Zahlen x bis 0. In der IF-THEN-ELSE-Anweisung wird geprüft, ob x=0 ist. Ist dies der Fall, ist das Ergebnis 0. Sonst -das ist der rekursive Fall- wird x zum Aufruf der Funktion mit minimiertem x addiert. Im ELSE-Teil steht also PLUS (x, Funkionsaufruf). Der Funktionsaufruf wird durch den Namen der Funktion, sowie dem minimierten Parameter (x-1) in Klammern realisiert. Verknüpft wird also für x=2: 2 + addz(1). addz(1) errechnet sich als 1 + addz (0). addz(0) ergibt 0. Also ergibt sich 0 + 1 + 2.

4/2/1
Um eine Liste absteigender Zahlen von n bis 1 zu erzeugen, wird im Fall, daß n=0 ist, die leere Liste ausgegeben. Anderenfalls wird mit dem Listenkonstruktur die Zahl n mit dem rekursiven Aufruf der Funktion verknüpft. Im rekursiven Aufruf wird die Zahl n minimiert, damit das Abbruchkriterium erreicht wird.

4/2/2
Im Kopf der Funktion steht die Zahl n. Die Funktion liefert eine Liste absteigender Zahlen von n bis 1. In der IF-THEN-ELSE-Anweisung wird geprüft, ob die Zahl n gleich 0 ist. Ist dies der Fall wird NIL, also die leere Liste ausgegeben. Anderenfalls wird mit dem Listenkonstruktor CONS die Zahl n mit dem rekursiven Aufruf der Funktion verknüpft. Die Funktion erhält die um 1 verminderte Zahl n als Parameter.

4/3/1
Um eine Liste zu erzeugen, die die Summen zwischen je zwei aufeinanderfolgenden Zahlen der Parameterliste enthält, muß im Fall einer einelementigen Liste die leere Liste ausgegeben werden, da für eine einzige Zahl keine Summe existiert. Im anderen Fall wird die Summe der ersten beiden Elemente mit dem rekursiven Aufruf der Restliste verknüpft. Die Verknüpfung geschieht mit dem Listenkonstruktor.

4/3/2
Im Kopf der Funktion steht eine Liste l. Die Funktion liefert eine Liste, die die Summenwerte zwischen je zwei aufeinanderfolgenden Elementen der Ursprungsliste enthält. In der IF-THEN-ELSE-Anweisung wird geprüft, ob die Liste nur ein Element enthält. Ist dies der Fall, wird die leere Liste (NIL) ausgegeben. Anderenfalls, das ist der rekursive Fall, wird die Summe der ersten beiden Elemente, also dem Kopf und dem Kopf vom TAIl der Liste, mit dem rekursiven Aufruf der Funktion verknüpft. Dabei wird der Funktion die um das erste Element minimierte Liste, also der TAIl der Liste, übergeben.

Beispiele

3/1/1
Die Funktion add7 addiert x-mal
7 zu einer Zahl z:

```
FUN add7 (z:NAT, x: NAT): NAT
IF EQUAL (x,0)
THEN z
ELSE add7 (PLUS(z,7),MINUS(x,1))
FIN
```

```
add7(3,2) = 17
add7(0,1) = 7
add7(4,0) = 4
```

3/1/2
Die Funktion add addiert eine Zahl
x in Einerschritten zu einer Zahl y:

```
FUN add (x: NAT, y: NAT): NAT
IF EQUAL (x,0)
THEN y
ELSE add (MINUS(x,1),PLUS(y,1))
FIN
```

```
add (1,1) = 2
add (3,1) = 4
add (0,2) = 2
```

3/2/1
Die Funktion length liefert die Länge n einer Liste l. n wird im Aufruf immer mit 0 übergeben:

```
FUN length (l:NATLIST,n:NAT):NAT
IF EMPTY (l)
THEN n
ELSE length (TAIL(l), PLUS(n,1))
FIN
```

```
length (list(1,2,3),0) = 3
length (list(),0)      = 0
```

3/2/2
Die Funktion sumbutlast bildet die Summe x aller Elemente der Liste l ohne das letzte Element. x wird im Aufruf immer mit 0 übergeben:

```
FUN sumbutlast (l:NATLIST,x:NAT):NAT
IF EMPTY (TAIL(l))
THEN x
ELSE sumbutlast (TAIL(l),PLUS (x,HEAD(l)))
FIN

sumbutlast (list(1,2,3),0) = 3
sumbutlast (list(2),0)     = 0
```

3/3/1
Die Funktion grex prüft, ob eine Zahl x größer ist als jedes Element der Liste l. Wenn ja liefert grex TRUE, sonst FALSE:

```
FUN grex (x:NAT,l:NATLIST):BOOL
IF EMPTY (l)
THEN TRUE
ELSE IF GREATER (x, HEAD(l))
    THEN grex (x, TAIL(l))
    ELSE FALSE
FIN

grex (5,list(1,2,3)) = TRUE
grex (1,list(1,2,3)) = FALSE
grex (3,NIL)         = TRUE
```

3/3/2
Die Funktion sumx prüft, ob eine Zahl x identisch mit der Summe zweier aufeinanderfolgender Elemente der Liste l ist. Wenn ja liefert sumx TRUE, sonst FALSE:

```
FUN sumx (x:NAT,l:NATLIST):BOOL
IF EMPTY (l)
THEN FALSE
ELSE IF EQUAL(x,PLUS(HEAD(l),HEAD(TAIL(l))))
    THEN TRUE
    ELSE sumx (x, TAIL(l))
FIN

sumx (4, list (1,2,2,3)) = TRUE
sumx (1, list (1,2,2,3)) = FALSE
sumx (1, NIL)            = FALSE
```

4/1/1
Die Funktion fac ist schon aus der Einführung bekannt. Sie liefert die Fakultät einer Zahl x:

```
FUN fac (x:NAT):NAT
IF NOT (GREATER (x,1))
THEN 1
ELSE MULT (x, fac(MINUS(x,1)))
FIN

fac(0) = 1
fac(1) = 1
fac(4) = 24
```

4/1/2
Die Funktion pot4 multipliziert eine Zahl x viermal mit sich selbst. Dazu wird eine Zählvariable count eingeführt, die im Aufruf immer mit 0 übergeben wird:

```
FUN pot4 (x:NAT, count:NAT):NAT
IF EQUAL (count,x)
THEN x
ELSE MULT (x,pot4(x,PLUS(count,1)))
FIN

pot4(2,0) = 24
pot4(0,0) = 0
```

4/2/1
Die Funktion slist konstruiert eine Liste, die n-mal die Zahl 7 enthält:

```
FUN slist (n:NAT):NATLIST
IF EQUAL (n,0)
THEN NIL
ELSE CONS (7, slist(MINUS(n,1)))
FIN

slist (3) = (7,7,7)
slist (0) = NIL
```

4/2/2
Die Funktion lenght ermittelt die Länge einer Liste l. Allerdings anders als im Beispiel in Lektion 3:

```
FUN lenght (l:NATLIST):NAT
IF EMPTY (l)
THEN 0
ELSE PLUS (1, length(TAIL(l)))
FIN

length (list(1,2,3)) = 3
length (list())      = 0
```

4/3/1
Die Funktion al bildet eine Liste, die die Summenwerte der Elemente zweier Listen l und m enthält. Addiert werden jeweils die Elemente, die bei Liste l und m dieselbe Position haben:

```
FUN al(l:NATLIST,m:NATLIST):NATLIST
IF EMPTY (l)
THEN NIL
ELSE CONS (PLUS (HEAD(l),HEAD(m)),al(TAIL(l),TAIL(m)))
FIN

al (list(1,2,3),list(4,5,6)) = (5,7,9)
al (NIL,list(1,2,3))         = NIL
```

4/3/2
Die Funktion butlast liefert eine Liste l ohne ihr letztes Element:

```
FUN butlast (l:NATLIST):NATLIST
IF EMPTY (TAIL(l))
THEN NIL
ELSE CONS(HEAD(l),butlast(TAIL(l)))
FIN

butlast (list(1,2,3)) = (1,2)
butlast (list(1))     = (1)
```

D.2 Beispiele in Untersuchung II *(Kap. 8.3)*

In Untersuchung II erhielten die Probanden pro Aufgabe ein Beispiel. Dabei waren die für die Aufgabe gleichbleibenden Strukturen fett markiert oder nicht (mit, ohne Mapping) und pro Proband hatten die Beispiele eine bestimmte strukturelle Ähnlichkeit (Beispielähnlichkeit I bis V bzw. VI) zur Aufgabe. Die im folgenden verwendeten Kennzeichnungen kodieren Lektion/Aufgabe/Ähnlichkeit. Alle Beispiele sind in der "Mit-Mapping-Vorgabe"-Bedingung angegeben.

3/1/1
Die Funktion bsp erhöht die Zahl z x-mal um eins.
Beispiel: z=3, x=2
bsp(3,2)=bsp(4,1)=bsp(5,0)= 5

FUN bsp (z:NAT,x:NAT):NAT
IF EQUAL(x,0)
THEN z
ELSE bsp(PLUS(z,1),MINUS(x,1))
FIN

3/1/2
Die Funktion bsp verdoppelt eine Zahl z x-1-mal.
Beispiel: z=3, x=2
bsp(3,2) = bsp(6,1) = 6

FUN bsp (z:NAT,x:NAT):NAT
IF EQUAL(MINUS(x,1),0)
THEN z
ELSE bsp(PLUS(z,z),MINUS(x,1))
FIN

3/1/3
Die Funktion bsp multipliziert eine Zahl z x-mal mit 10. Dies wird realisiert, indem die jeweils verdoppelte Zahl mit 5 multipliziert wird.
Beispiel: z=3 x=2
bsp(3,2)=bsp(30,1)=bsp(300,0)=300

FUN bsp (z:NAT,x:NAT):NAT
IF EQUAL(x,0)
THEN z
ELSE bsp(MULT(PLUS(z,z),5),MINUS(x,1))
FIN

3/1/4
Die Funktion bsp multipliziert eine Zahl z x-1-mal mit 10. Dies wird realisiert, indem die jeweils verdoppelte Zahl mit 5 multipliziert wird.
Beispiel: z=3, x=2
bsp(3,2) = bsp(30,1) = 30

```
FUN bsp (z:NAT,x:NAT):NAT
IF EQUAL(MINUS(x,1),0)
THEN z
ELSE bsp(MULT(PLUS(z,z),5),MINUS(x,1))
FIN
```

3/1/5
Die Funktion bsp multipliziert eine Zahl z x-mal mit 10. Dies wird realisiert, indem die jeweils verdoppelte Zahl mit 5 multipliziert wird.
Beispiel: z=3, x=2
bsp(3,2)=bsp(30,1)=MULT(30,10)=300

```
FUN bsp (z:NAT,x:NAT):NAT
IF EQUAL(MINUS(x,1),0)
THEN MULT(z,10)
ELSE bsp(MULT(PLUS(z,z),5),MINUS(x,1))
FIN
```

3/1/6
Die Funktion bsp addiert eine Zahl z x+1-mal.
Beispiel: z=3, x=2
bsp(3,2) = 3+bsp(3,1)=3+3+bsp(3,0) = 3+3+3 = 9

```
FUN bsp (z:NAT,x:NAT):NAT
IF EQUAL(x,0)
THEN z
ELSE PLUS(z,bsp(z,MINUS(x,1)))
FIN
```

3/2/1
Die Funktion bsp verkürzt eine Liste und gibt, wenn nur noch ein Element übrig ist, die Zahl 1 aus.
Beispiel: l=(1,2,3)
bsp(list(1,2,3))=bsp(list(2,3))=bsp(list(3))=1

```
FUN bsp (l:NATLIST):NAT
IF EMPTY(TAIL(l))
THEN 1
ELSE bsp(TAIL(l))
FIN
```

3/2/2
Die Funktion bsp gibt das um eins erhöhte letzte Listenelement aus.
Beispiel: l=(1,2,3)
bsp(list(1,2,3))=bsp(list(2,3))=bsp(list(3))=PLUS(3,1)=4

FUN bsp (l:NATLIST):NAT
IF EMPTY(TAIL(l))
THEN PLUS(**HEAD(l)**,1)
ELSE bsp(TAIL(l))
FIN

3/2/3
Die Funktion bsp arbeitet auf Listen mit einer ungeraden Anzahl von Elementen und gibt für solche Listen das letzte Element aus.
Beispiel: l=(1,2,3)
bsp(list(1,2,3))=bsp(list(3))=3

FUN bsp(l:NATLIST):NAT
IF EMPTY(TAIL(l))
THEN HEAD(l)
ELSE bsp(TAIL(**TAIL(l)**))
FIN

3/2/4
Die Funktion bsp arbeitet auf Listen mit einer ungeraden Anzahl von Elementen und gibt für solche Listen das um eins erhöhte letzte Element aus.
Beispiel: l=(1,2,3)
bsp(list(1,2,3))=bsp(list(3))=PLUS(3,1)=4

FUN bsp(l:NATLIST):NAT
IF EMPTY(TAIL(l))
THEN PLUS(**HEAD(l)**,1)
ELSE bsp(TAIL(**TAIL(l)**))
FIN

3/2/5
Die Funktion bsp arbeitet auf Listen mit einer geraden Anzahl von Elementen und gibt für solche Listen das um eins erhöhte vorletzte Element aus.
Beispiel: l=(1,2,3,0)
bsp(list(1,2,3,4))=bsp(list(3,4))=*PLUS*(3,1)=4

FUN bsp(l:NATLIST):NAT
IF EMPTY(TAIL(TAIL(l)))
THEN PLUS(**HEAD(l)**,1)
ELSE bsp(TAIL(**TAIL(l)**))
FIN

3/2/6
Die Funktion bsp addiert alle Elemente einer Liste.
Beispiel: l=(1,2,3)
bsp(list(1,2,3))=1+bsp(list(2,3))=1+2+bsp(list(3))=1+2+3=6

```
FUN bsp (l:NATLIST):NAT
IF EMPTY(TAIL(l))
THEN HEAD(l)
ELSE PLUS(HEAD(l),bsp(TAIL(l)))
FIN
```

3/3/1
Die Funktion bsp liefert 1, wenn mindestens ein Element einer Liste größer ist als eine Zahl n, und 0, wenn kein Listenelement größer ist als n.
Beispiel: n=2, l=(1,2,3)
bsp(2,list(1,2,3))=bsp(2,list(2,3))=bsp(2,list(3))=1

```
FUN bsp (n:NAT,l:NATLIST):NAT
IF EMPTY(l)
THEN 0
ELSE IF GREATER(HEAD(l),n)
    THEN 1
    ELSE bsp(n,TAIL(l))
FIN
```

3/3/2
Die Funktion bsp liefert 1, wenn eine Zahl n in einer Liste ohne das letzte Element enthalten ist, und 0, wenn n nicht in der Liste ohne das letze Element enthalten ist.
Beispiel: n=2, l=(1,2,3)
bsp(2,list(1,2,3))=bsp(2,list(2,3))=1

```
FUN bsp (n:NAT,l:NATLIST):NAT
IF EMPTY(TAIL(l))
THEN 0
ELSE IF EQUAL(HEAD(l),n)
    THEN 1
    ELSE bsp(n,TAIL(l))
FIN
```

3/3/3
Die Funktion bsp prüft, ob das erste Listenelement gleich n oder das zweite Element gleich n+1, das dritte gleich n+2 etc. ist. Sobald eine Übereinstimmung gefunden wird, liefert die Funktion 1. Wird keine Übereinstimmung gefunden, liefert die Funktion 0.
Beispiel: n=2, l=(1,2,3)
bsp(2,list(1,2,3))=bsp(3,list(2,3))=bsp(4,list(3))=bsp(5,nil)=0

```
FUN bsp (n:NAT,l:NATLIST):NAT
IF EMPTY(l)
THEN 0
ELSE IF EQUAL(HEAD(l),n)
    THEN 1
    ELSE bsp(PLUS(n,1),TAIL(l))
FIN
```

3/3/4
Die Funktion bsp prüft für alle Listenelemente, außer dem letzten, ob das erste Listenelement gleich n oder das zweite Element gleich n+1, das dritte gleich n+2 etc. ist. Sobald eine Übereinstimmung gefunden wird, liefert die Funktion 1. Wird keine*Übereinstimmung* gefunden, liefert die Funktion 0.
Beispiel: n=2, l=(1,2,3)
bsp(2,list(1,2,3))=bsp(3,list(2,3))=bsp(4,list(3))=0

```
FUN bsp (n:NAT,l:NATLIST):NAT
IF EMPTY(TAIL(l))
THEN 0
ELSE IF EQUAL(HEAD(l),n)
    THEN 1
    ELSE bsp(PLUS(n,1),TAIL(l))
FIN
```

3/3/5
Die Funktion bsp prüft für alle Listenelemente, außer dem letzten, ob das erste Listenelement gleich n oder das zweite Element gleich n+1, das dritte gleich n+2 etc. ist. Sobald eine Übereinstimmung gefunden wird, liefert die Funktion die Zahl, für die
die Übereinstimmung gefunden wurde. Wird keine Übereinstimmung gefunden, liefert die Funktion 0.
Beispiel: n=2, l=(1,2,3)
bsp(2,list(1,2,3))=bsp(3,list(2,3))=bsp(4,list(3))=0

```
FUN bsp (n:NAT,l:NATLIST):NAT
IF EMPTY(TAIL(l))
THEN 0
ELSE IF EQUAL(HEAD(l),n)
    THEN n
    ELSE bsp(PLUS(n,1),TAIL(l))
FIN
```

3/3/6
Die Funktion bsp prüft, ob ein Element n in einer Liste vorhanden ist. Sie gibt 0 aus, wenn das Element nicht in der Liste vorkommt, ansonsten die Listenposition des gesuchten Elementes.
Beispiel: n=2, l=(1,2,3)
bsp(2,list(1,2,3))=1+bsp(2,list(2,3))=1+1=2

```
FUN bsp (n:NAT,l:NATLIST):NAT
IF EMPTY(l)
THEN 0
ELSE IF EQUAL(HEAD(l),n)
    THEN 1
    ELSE PLUS(1,bsp(n,TAIL(l)))
FIN
```

4/1/1
Die Funktion bsp berechnet die Summe einer Zahl mit all ihren Vorgängern ohne 2,1 und 0.
Beispiel: x=5 ergibt
5+4+3+2 = 14

```
FUN bsp (x:NAT):NAT
IF EQUAL(x,2)
THEN 0
ELSE PLUS(x,bsp(MINUS(x,1)))
FIN
```

4/1/2
Die Funktion bsp berechnet die Summe einer Zahl mit all ihren Vorgängern ohne 1.
Beispiel: x=5 ergibt
5+4+3+2 = 14

```
FUN bsp (x:NAT):NAT
IF EQUAL(MINUS(x,1),0)
THEN 0
ELSE PLUS(x,bsp(MINUS(x,1)))
FIN
```

4/1/3
Die Funktion bsp berechnet die Summe einer quadrierten Zahl mit all ihren quadrierten Vorgängern.
Beispiel: x=5 ergibt
5*5+4*4+3*3+2*2+1*1+0 = 25+16+9+4+1+0 = 55

```
FUN bsp (x:NAT):NAT
IF EQUAL(x,0)
THEN 0
ELSE PLUS(MULT(x,x),bsp(MINUS(x,1)))
FIN
```

4/1/4
Die Funktion bsp berechnet die Summe einer quadrierten Zahl mit all ihren quadrierten Vorgängern ohne 1.
Beispiel: x=5 ergibt
5*5+4*4+3*3+2*2 = 25+16+9+4 = 54

```
FUN bsp (x:NAT):NAT
IF EQUAL(MINUS(x,1),0)
THEN 0
ELSE PLUS(MULT(x,x),bsp(MINUS(x,1)))
FIN
```

4/1/5
Die Funktion bsp berechnet die Summe einer quadrierten Zahl mit all ihren quadrierten Vorgängern ohne 1.
Beispiel: x=5 ergibt
5*5+4*4+3*3+2*2 = 25+16+9+4 = 54

```
FUN bsp (x:NAT):NAT
IF EQUAL(MINUS(x,1),0)
THEN MINUS(x,1)
ELSE PLUS(MULT(x,x),bsp(MINUS(x,1)))
FIN
```

4/1/6
Die Funktion bsp prüft, ob eine Zahl gerade oder ungerade ist. Ist die Zahl gerade liefert sie 0, sonst 1.
Beispiel: x=5 ergibt
5-2-2=1

```
FUN bsp(x:NAT):NAT
IF GREATER(2,x)
THEN x
ELSE bsp(minus(x,2))
FIN
```

4/2/1
Die Funktion bsp gibt eine Liste absteigender Zahlen von n bis 2 aus.
Beispiel: n = 5 ergibt
(5,4,3,2)

```
FUN bsp (n:NAT):NATLIST
IF EQUAL(n,1)
THEN NIL
ELSE CONS(n,bsp(MINUS(n,1)))
FIN
```

4/2/2
Die Funktion bsp gibt eine Liste absteigender Zahlen von n bis 2 aus.
Beispiel: n = 5 ergibt
(5,4,3,2)

```
FUN bsp (n:NAT):NATLIST
IF EQUAL(MINUS(n,1),0)
THEN NIL
ELSE CONS(n,bsp(MINUS(n,1)))
FIN
```

4/2/3
Die Funktion bsp gibt eine Liste verdoppelter absteigender Zahlen von n bis 1 aus.
Beispiel: n = 5 ergibt
(10,8,6,4,2)

```
FUN bsp (n:NAT):NATLIST
IF EQUAL(n,0)
THEN NIL
ELSE CONS(PLUS(n,n),bsp(MINUS(n,1)))
FIN
```

4/2/4
Die Funktion bsp gibt eine Liste verdoppelter absteigender Zahlen von n bis 2 aus.
Beispiel: n = 5 ergibt
(10,8,6,4)

```
FUN bsp (n:NAT):NATLIST
IF EQUAL(MINUS(n,1),0)
THEN NIL
ELSE CONS(PLUS(n,n),bsp(MINUS(n,1)))
FIN
```

4/2/5
Die Funktion bsp gibt eine Liste verdoppelter absteigender Zahlen von n bis 1 aus.
Beispiel: n = 5 ergibt
(10,8,6,4,1)

```
FUN bsp (n:NAT):NATLIST
IF EQUAL(MINUS(n,1),0)
THEN CONS(1,NIL)
ELSE CONS(PLUS(n,n),bsp(MINUS(n,1)))
FIN
```

4/2/6
Die Funktion bsp gibt eine Liste (0) aus, wenn die eingegebene Zahl geradzahlig ist, ist n ungerade, gibt sie die Liste (1) aus.
Beispiel: n = 5 ergibt (1)

```
FUN bsp (n:NAT):NATLIST
IF GREATER(2,n)
THEN CONS(n,NIL)
ELSE bsp(MINUS(n,2)))
FIN
```

4/3/1
Die Funktion bsp gibt eine Liste aus, die das Produkt von aufeinanderfolgenden Listenelementen ausgibt.
Beispiel: l = (6,4,1) ergibt
(24,4)

```
FUN bsp(l:NATLIST):NATLIST
IF EMPTY(TAIL(l))
THEN NIL
ELSE CONS (MULT(HEAD(l),HEAD(TAIL(l))), bsp(TAIL(l)))
FIN
```

4/3/2
Die Funktion bsp gibt eine Liste aus, die die Summe von aufeinanderfolgenden Listenelementen ohne das letzte Element ausgibt.
Beispiel: l = (6,4,1) ergibt
(10)

```
FUN bsp(l:NATLIST):NATLIST
IF EMPTY(TAIL(TAIL(l)))
THEN NIL
ELSE CONS (PLUS(HEAD(l),HEAD(TAIL(l))),bsp(TAIL(l)))
FIN
```

4/3/3
Die Funktion bsp gibt eine Liste aus, die die Summe von aufeinanderfolgenden Listenelementen durch 0en getrennt ausgibt.
Beispiel: l = (6,4,1) ergibt
(0,10,0,5)

```
FUN bsp(l:NATLIST):NATLIST
IF EMPTY(TAIL(l))
THEN NIL
ELSE CONS (CONS(0,PLUS(HEAD(l), HEAD(TAIL(l)))), bsp(TAIL(l)))
FIN
```

4/3/4
Die Funktion bsp gibt eine Liste aus, die die Summe von aufeinanderfolgenden Listenelementen ohne das letzte Element durch 0en getrennt ausgibt.
Beispiel: l = (6,4,1) ergibt
(0,10)

```
FUN bsp(l:NATLIST):NATLIST
IF EMPTY(TAIL(TAIL(l)))
THEN NIL
ELSE CONS (CONS(0,PLUS(HEAD(l), HEAD(TAIL(l)))), bsp(TAIL(l)))
FIN
```

4/3/5
Die Funktion bsp gibt eine Liste aus, die die Summe von aufeinanderfolgenden Listenelementen durch 0en getrennt ausgibt, und das letzte Element unverändert läßt.
Beispiel: l = (6,4,1) ergibt
(0,10,1)

```
FUN bsp(l:NATLIST):NATLIST
IF EMPTY(TAIL(TAIL(l)))
THEN TAIL(l)
ELSE CONS (CONS(0,PLUS(HEAD(l),HEAD(TAIL(l)))), bsp(TAIL(l)))
FIN
```

4/3/6
Die Funktion bsp liefert eine Liste, die zweimal das letzte Element der Eingangsliste enthält.
Beispiel: l = (6,4,1) ergibt
(1,1)

```
FUN bsp(l:NATLIST):NATLIST
IF EMPTY(TAIL(l))
THEN CONS(HEAD(l),l)
ELSE bsp(TAIL(l))
FIN
```

D.3 *Abschlußtest (Kap. 8.1)*

LEAR Abschlußtest

VP-Nr. ___

1) Kreuze die Aussagen über rekursive Funktionen an, die korrekt sind:

   Eine rekursive Funktion ruft sich unendlich oft wieder auf. ___

   Bei rekursiven Funktionen werden die Parameter im rekursiven Aufruf solange verändert, bis die Abbruchbedingung erfüllt ist. ___

   Rekursion ist der Aufruf einer Funktion durch sich selber. ___

   Bei rekursiven Funktionen wird stets ein Parameter unverändert wieder ausgegeben. ___

2) Gib zu den genannten Eingaben die Ausgaben an, die die Funktion liefert.

```
FUN rek (x:NAT, y: NAT): NAT
IF EQUAL (y, 0)
THEN 0
ELSE PLUS (x, rek (x, MINUS (y,1)))
FIN
```

   a) x = 3, y = 2 : _____

   b) x = 1, y = 4 : _____

   c) x = 7, y = 0 : _____

3) Welche Werte für den Parameter l müssen der Funktion übergeben werden, um folgende Ergebnisse zu erhalten?

```
FUN rek (x: NAT, l:NATLIST): NATLIST
IF EMPTY (l)
THEN NIL
ELSE CONS (PLUS (x, HEAD (l)), rek (x, TAIL (l)))
FIN
```

a) x = 3 Ergebnis = (4,5,6) l = _____

b) x = 1 Ergebnis = (17,4) l = _____

c) x = 0 Ergebnis = (1,2,3,4) l = _____

4) Schreibe eine Funktion, die prüft, ob mindestens ein Element, das kleiner oder gleich einer Zahl x ist, in einer Liste vorhanden ist. Ist dies der Fall, soll die Funktion TRUE ausgeben, sonst FALSE.

5) Kreuze die Aussagen an, die korrekt sind.

Rekursive Funktionen benötigen eine Abbruchbedingung,

damit der rekursive Prozeß beendet werden kann. ___

damit der rekursive Ablauf bei falscher Eingabe unterbrochen wird. ___

damit bei einer vorgegebenen Bedingung der Funktion ein fester Wert zugeordnet werden kann. ___

6) Schreibe eine Funktion, die die Elemente einer Liste addiert und als Ergebnis die Summe aller Listenelemente liefert.

7) Eine Funktion soll alle Elemente einer Liste addieren, die größer als 2 sind, gib an, welche Funktion(en) diese Aufgabe erfüllt/erfüllen und welche nicht. Begründe Deine Wahl, wenn Du der Meinung bist, daß eine Funktion die Aufgabe nicht erfüllt.

a) FUN addl (l:NATLIST):NAT
IF GREATER (HEAD(l),2)
THEN PLUS (HEAD(l), addl(TAIL(l)))
ELSE addl (TAIL(l))
FIN

b) FUN addl (l:NATLIST):NAT
IF EMPTY (l)
THEN 0
ELSE IF GREATER (HEAD(l),2)
THEN PLUS (HEAD(l),addl(TAIL(l)))
ELSE addl (TAIL(l))
FIN

c) FUN addl (x:NAT,l:NATLIST):NAT
IF EMPTY (l)
THEN 0
ELSE IF GREATER (HEAD(l),2)
THEN PLUS (HEAD(l),addl(TAIL(l)))
ELSE addl (TAIL(l))
FIN

d) FUN addl (l:NATLIST):NAT
IF EMPTY (TAIL(l))
THEN HEAD(l)
ELSE addl (CONS (HEAD (TAIL(l)), TAIL(l)))
FIN

a) korrekt? ja___ nein___
Wenn nicht korrekt, warum? _____
_____

b) korrekt? ja___ nein___
Wenn nicht korrekt, warum? _____
_____

c) korrekt? ja___ nein___
Wenn nicht korrekt, warum? _____
_____

d) korrekt? ja___ nein___
Wenn nicht korrekt, warum? _____
_____

8) Gib zu den genannten Eingaben die entsprechenden Ausgaben an, die die Funktion liefert.

```
FUN rek (l: NATLIST, n:NAT): NAT
IF EMPTY (l)
THEN n
ELSE IF GREATER (HEAD(l), n)
    THEN rek (TAIL(l), HEAD(l))
    ELSE rek (TAIL(l), n)
FIN
```

Der Parameter n muß der Funktion immer mit 0 übergeben werden.

a) l = (4,1,8,2) n = 0 : ___

b) l = (3) n = 0 : ___

9) [Nur in Untersuchung II]
Schreibe eine Funktion fib, die die Fibonacci-Zahl einer Zahl n ausgibt.
Die Fibonacci-Funktion ist definiert als:
fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2)

D.4 Sortieraufgaben (Kap. 8.3)

NR. 12

```
FUN test (x:nat,y:nat):nat
IF GREATER(y,x)
THEN x
ELSE test (MINUS(x,y),y)
FIN
```

NR.19

```
FUN test (l:natlist,x:nat):nat
IF EQUAL(x,1)
THEN HEAD(l)
ELSE test(TAIL(l),MINUS(x,1))
FIN
```

NR. 23

```
FUN test(x:nat,y:nat):nat
IF GREATER(10,PLUS(x,y))
THEN 0
ELSE IF EQUAL (10,PLUS(x,y))
    THEN x
    ELSE test(MINUS(x,1),y)
FIN
```

NR. 16

```
FUN test(l:natlist):nat
IF EMPTY(TAIL(l))
THEN HEAD(l)
ELSE IF GREATER(HEAD(l),HEAD(TAIL(l)))
    THEN HEAD(l)
    ELSE test(TAIL(l))
FIN
```

NR. 6

```
FUN test(x:nat,y:nat):nat
IF EQUAL(x,0)
THEN y
ELSE test(MINUS(x,1),y)
FIN
```

NR. 8

```
FUN test(l:natlist):nat
IF EMPTY(TAIL(l))
THEN MULT(HEAD(l),2)
ELSE test(TAIL(l))
FIN
```

NR. 10

```
FUN test(x:nat,y:nat):nat
IF EQUAL(x,0)
THEN 0
ELSE IF GREATER(x,y)
    THEN y
    ELSE test(MINUS(x,y),y)
FIN
```

NR. 4

```
FUN test(l:natlist,m:natlist):nat
IF EMPTY(l)
THEN 0
ELSE IF EQUAL(HEAD(l),HEAD(m))
    THEN 1
    ELSE test(TAIL(l),TAIL(m))
FIN
```

NR. 25

```
FUN test(x:nat,y:nat):nat
IF GREATER(x,y)
THEN 0
ELSE IF EQUAL(x,y)
   THEN y
   ELSE MULT(x,test(PLUS(x,1),y)
FIN
```

NR. 21

```
FUN test(l:natlist):nat
IF EMPTY(l)
THEN 0
ELSE PLUS(1,test(TAIL(l)))
FIN
```

NR. 26

```
FUN test(x:nat):nat
IF NOT(GREATER(x,1))
THEN 1
ELSE PLUS(MULT(x,2),test(MINUS(x,1)))
FIN
```

NR. 15

```
FUN test(l:natlist):nat
IF EMPTY(l)
THEN 0
ELSE PLUS(HEAD(l),test(TAIL(l))
FIN
```

NR. 1

```
FUN test(x:nat,y:nat):nat
IF EQUAL(y,0)
THEN 1
ELSE MULT(x,test(x,MINUS(y,1)))
FIN
```

NR. 27

```
FUN test(m:natlist):nat
IF EMPTY(m)
THEN 1
ELSE MULT(2,test(TAIL(m))
FIN
```

NR. 17

```
FUN test(x:nat,y:nat):nat
IF EQUAL(x,0)
THEN 0
ELSE PLUS(PLUS(x,y),test(MINUS(x,1),y))
FIN
```

NR. 2

```
FUN test(l:natlist,m:natlist):natlist
IF EMPTY(l)
THEN NIL
ELSE IF EMPTY(m)
   THEN HEAD(l)
   ELSE CONS(PLUS(HEAD(l),HEAD(m)),
            test(TAIL(l),TAIL(m)))
FIN
```

NR. 22

```
FUN test(x:nat):nat
IF GREATER(2,x)
THEN 1
ELSE PLUS(test(MINUS(x,1)),test(MINUS(x,2))
FIN
```

NR. 11

```
FUN test(x:nat,y:nat):nat
IF EQUAL(x,y)
THEN 1
ELSE IF EQUAL(y,0)
   THEN 1
   ELSE PLUS(test(MINUS(x,1),MINUS(y,1)),
        test(MINUS(x,1),y))
FIN
```

NR. 7

```
FUN test(x:nat,l:natlist):natlist
IF EMPTY(l)
THEN NIL
ELSE IF EQUAL(HEAD(l))
   THEN test(x,TAIL(l))
   ELSE CONS (HEAD(l),test(x,TAIL(l)))
FIN
```

NR. 3

```
FUN test(x:nat,y:nat):nat
IF GREATER(y,x)
THEN x
ELSE IF GREATER(MULT(2,y),x)
   THEN MINUS(x,y)
   ELSE test(test(x,MULT(2,y)),y)
FIN
```

NR. 24

```
FUN test(x:nat,y:nat):nat
IF NOT(GREATER(x,y))
THEN MINUS(y,x)
ELSE MULT(PLUS(x,10),y)
FIN
```

NR. 18

```
FUN test(x:nat):nat
IF EQUAL(x,0)
THEN 0
ELSE PLUS(x,MULT(x,2))
FIN
```

NR. 9

```
FUN test(l:natlist):nat
IF EMPTY(l)
THEN 0
ELSE HEAD(TAIL(l))
FIN
```

NR. 28

```
FUN test(x:nat,l:natlist):natlist
IF GREATER(HEAD(l),x)
THEN l
ELSE CONS(x,TAIL(l))
FIN
```

NR. 5

```
FUN test(x:nat,y:nat):nat
IF EQUAL(x,0)
THEN 0
ELSE IF GREATER (x,y)
   THEN MINUS(x,y)
   ELSE MULT(x,y)
FIN
```

NR. 14

```
FUN test(x:nat,y:nat):nat
IF EQUAL(x,y)
THEN 0
ELSE IF GREATER(y,x)
   THEN 0
   ELSE MINUS(x,y)
FIN
```

NR. 20

```
FUN test(l:natlist,m:natlist):natlist
IF EMPTY(l)
THEN m
ELSE IF EMPTY(m)
   THEN NIL
   ELSE CONS(HEAD(TAIL(l)),TAIL(m))
FIN
```

NR. 13

```
FUN test(l:natlist,m:natlist):nat
IF EMPTY(l)
THEN 0
ELSE IF EMPTY(m)
   THEN HEAD(l)
   ELSE PLUS(HEAD(l),HEAD(m))
FIN
```