Kognitionswissenschaft (1994) 4: 47–54

DOI: 10.20378/irb-90973

Kognitionswissenschaft

Programmieren lernen: Unterstützung des Erwerbs rekursiver Programmiertechniken durch Beispielfunktionen und Erklärungstexte

U. Schmid

Institut für Psychologie, TU Berlin, Dovestraße 1–5, D-10587 Berlin

Supporting the acquisition of recursive programming skills by examples and explanatory texts

Summary. Learning recursive techniques often provides the greatest difficulties for programming novices. Therefore the selection of adequate instructional material is highly relevant for facilitating the learning process. In this paper a study is presented where the impact of example programs and explanatory texts on the acquisition of recursive programming techniques was investigated. The results show that the process of learning by doing is significantly better supported by explanations than by examples. But despite the apparent disadvantage of subjects learning with examples, treatment with the two kinds of instruction materials does not result in differences with respect to transfer scores.

Zusammenfassung. Beim Erwerb von Programmiertechniken ist das Lernen rekursiver Kontrollstrukturen für Anfänger meist die größte Hürde. Die Auswahl von geeignetem Lehrmaterial zur Unterstützung des Lernprozesses ist deshalb gerade für die Vermittlung rekursiver Programmierung relevant. Es wird eine Studie vorgestellt, in der der Einfluß von zwei Lehrmaterialien – Beispielprogrammen und erklärenden Texten – auf den Erwerb rekursiver Programmiertechniken untersucht wurde. Es zeigte sich, daß Lernen mit Erklärungen und mit Beispielen zu vergleichbaren Transferleistungen führt, die Performanz während des Lernprozesses ist bei Unterstützung durch Erklärungen allerdings wesentlich besser als bei Unterstützung durch Beispiele.

1. Einleitung

Während sich zahlreiche empirische Untersuchungen zum Erwerb von Programmierwissen mit der Identifikation von Fehlkonzeptionen von Programmieranfängern befassen (Kahney 1989, Bonar & Soloway 1989), gibt es nur wenige Untersuchungen zum Prozeß des Wissenserwerbs und zur möglichen Unterstützung des Lernprozesses. Solche Untersuchungen könnten eine empirische Grundlage für die Modellierung des Wissensstandes von Lernenden sowie für den Entwurf tutorieller Strategien im Rahmen intelligenter tutorieller Systeme (z. B. Anderson, Conrad & Corbett 1989, Bonar & Cunningham 1989) liefern. Einige Aufschlüsse über den Prozeß des Programmierenlernens erhält man über Einzelfallstudien, bei denen mit Protokollen lauten Denkens gearbeitet wird (Anderson, Farrell & Sauers 1984, Pirolli & Anderson 1985). Ein experimenteller Zugang zum Prozeß der Wissensakquisition ist die Vermittlung neuer Fertigkeiten oder Kenntnisse mit verschiedenen Lehrmaterialien, über deren Einfluß auf den Lernprozeß unterschiedliche Annahmen getroffen werden können. Untersuchungen dieser Art werden im Bereich Programmieren vor allem zum Erwerb von syntaktischen Konzepten von Programmiersprachen (Schmalhofer, Boschert & Kühn 1990) durchgeführt. Der Einfluß von Lehrmaterialien auf den Erwerb prozeduraler Fertigkeiten wird vorwiegend beim mathematischen Problemlösen untersucht (Reed & Bolstad 1991, Novick & Holyoak 1991).

Als unterschiedliche Lehrmaterialien werden sowohl in der Instruktionspsychologie als auch im Bereich „Maschinelles Lernen" (Buchanan, Mitchell, Smith & Johnson 1977) Beispiele (learning from examples) und Erklärungen (learning by being told) verwendet. In beiden Bereichen ist das größte Augenmerk auf den Nutzen von Beispielen für die Wissensakquisition gerichtet (Mitchell, Keller & Cedar-Cabelli 1986, Novick & Holyoak 1991). Allerdings liegt auch in diesen Bereichen der Fokus auf dem Lernen von Konzepten und nicht auf dem Erwerb prozeduraler Fertigkeiten. Beim Konzeptlernen steht die Frage nach der Auswahl geeigneter positiver und negativer Beispiele für den zu lernenden Begriff im Mittelpunkt. Beim Erwerb prozeduraler Fertigkeiten ist die kritische Frage dagegen nicht das Finden von Kriterien zur Konstruktion angemessener Beispiele für den Lerngegenstand, sondern die Voraussetzungen, unter denen ein Beispiel korrekt auf eine Aufgabenstellung übertragen werden kann (Pirolli & Anderson 1985, Samuel 1963, Novick

& Holyoak 1991). Als Beispiele werden hier Lösungen zu der Aufgabe ähnlichen Problemstellungen verwendet. Solche Beispiellösungen können auch bereits gelöste Probleme aus der eigenen Lerngeschichte sein (Ross & Kennedy 1990, Weber, Waloszek & Wender 1988). Prinzipiell ist damit beim Lernen von Problemlöseprozeduren das Lernen mit Beispielen mit dem Ansatz des learning by doing (Anderson et al. 1989) gleichzusetzen (Ross & Kennedy 1990).

Der Nutzen von Beispielen für den Wissenserwerb wird allgemein darin gesehen, daß das Lernen mit Beispielen den Aufbau allgemeiner Lösungsschemata unterstützt (Novick & Holyoak 1991). Die Annahme, daß Experten über Schemawissen verfügen, ist in bezug auf rekursive Programmierung psychologisch plausibel (Vorberg & Goebel 1991). Während das Finden von iterativen Algorithmen zur Lösung eines Problems viele Freiheitsgrade zuläßt, ist die rekursive Lösung für ein Problem und ihre Realisierung in einer funktionalen Sprache meist eindeutig. Rekursive Lösungen von Experten sind relativ uniform (Haack, Hahn & Wagner 1989). Andererseits ist gerade das rekursive Programmieren besonders schwer zu lernen (Pirolli & Anderson 1985, Kahney 1989). Würde sich die Problemlöseexpertise auf das Vorhandensein adäquater Rekursionsschemata beschränken, müßte eine Vorgabe dieser Schemata den Lernprozeß erleichtern. Empirische Befunde zeigen jedoch, daß Anfänger solche abstrakten Informationen nicht nutzen können (Anderson, Farrell & Sauers 1984).

Dies spricht dafür, daß neben dem Aufbau von Schemata auch Regeln zur Auswahl und Anwendung dieser Schemata erworben werden müssen. Zudem müssen zum Finden einer rekursiven Lösung für ein nicht rekursiv definiertes Problem zunächst Regeln zur Bildung der Rekursion angewendet werden, bevor das angemessene Rekursionsschema ausgewählt werden kann. Solche Regeln zur Bildung von Rekursionen werden zum Beispiel im System DEDALUS zum automatischen Programmieren (Manna & Waldinger 1975; recursion-formation rules) und von Goebel & Vorberg (1991; Restselektorstrategie) vorgeschlagen.

Im folgenden wird die Auffassung vertreten, daß die Regeln zur Bildung einer rekursiven Lösungsstruktur aus einer nicht rekursiv gegebenen Problemstellung maßgeblich für eine erfolgreiche Problemlösung sind, während das Vorhandensein von Strukturinformation in Form von Rekursionsschemata nur die Effektivität der Problemlösung beeinflußt.

Im folgenden werden zunächst die Lehrmaterialien Beispiele und Erklärungen und ihr möglicher Einfluß auf den Erwerb von Rekursionsschemata und von Regeln zur Bildung von rekursiven Strukturen diskutiert. Danach wird über eine Studie berichtet, in der Programmieranfänger über 8 Stunden im Programmieren rekursiver Funktionen unterrichtet wurden, wobei eine Gruppe von Probanden durch Beispiele, eine zweite durch Erklärungstexte und eine dritte durch beide Materialien unterstützt wurden.

2. Wissenserwerb mit Beispielen und Erklärungen

2.1. Wissensbereich: rekursive Programmiertechniken

Zum besseren Verständnis des untersuchten Wissensbereichs „rekursive Programmiertechniken" folgt zunächst eine kurze informelle Einführung in das Konzept der Rekursion (siehe z. B. Perl 1979). Am anschaulichsten wird das Konzept am Beispiel rekursiver Definitionen mathematischer Funktionen, z. B. der Fakultätsfunktion: Die Fakultät einer Zahl *(n!)* ist das Produkt der Zahl mit allen Vorgängern, die Fakultät von null ist als eins definiert. So ist 5! gleich $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$. Die allgemeine, nicht rekursive Definition der Funktion ist:

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

für alle $n \in \mathbb{N}$.

Charakteristisch für rekursive Definitionen ist die Verwendung der zu definierenden Operation auf beiden Seiten der Gleichung (rekursiver Fall). Zusätzlich muß immer mindestens ein Fall angegeben werden, für den eine direkte Lösung existiert (Rekursionsverankerung). Solche Definitionen sind möglich, wenn sich ein Problem in strukturgleiche Teilprobleme zerlegen läßt. Bei der Fakultätsfunktion ist der direkte Fall als $0! = 1$ definiert, die rekursive Umformung der Produktformel ist $n! = n \cdot (n-1)!$. Die vollständige mathematische rekursive Definition des Problems lautet:

Fakultät: $\mathbb{N} \rightarrow \mathbb{N}$

$$\text{Fakultät}(n) = \begin{cases} 1 & \text{wenn } n = 0 \\ n \cdot (n-1)! & \text{sonst} \end{cases}$$

für alle $n \in \mathbb{N}$.

Hier wird, wie in algebraischen Definitionen von Funktionen üblich (Ehrig & Mahr 1985), zusätzlich der Argument- und der Wertebereich der Funktion angegeben: Die Fakultätsfunktion erhält als Argument ein Element der natürlichen Zahlen und liefert als Ergebnis eine natürliche Zahl.

Solche Definitionen können in funktionalen Programmiersprachen analog als Programme übernommen werden. Das heißt, daß bei rekursiven Programmen nicht die einzelnen Schritte der Problemlösung definiert werden, sondern ein Lösungsplan angegeben wird, der vom System (Interpreter) ausgeführt wird. Im folgenden wird mit einer einfachen funktionalen Sprache gearbeitet, die vom Autor im Rahmen einer Lernumgebung zum Erwerb rekursiver Programmiertechniken (LEAR, s. unten) entwickelt wurde. Die Fakultätsfunktion wird in dieser Sprache durch folgendes Programm realisiert:

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

Die Unterschiede zur rekursiven mathematischen Definition sind rein syntaktischer Art: Name, Argument- und Wertebereich der Funktion werden in der ersten Zeile (Funktionskopf) kodiert; dann wird die Bedingung für den direkten Fall angegeben; das Funktionsergebnis für den direkten Fall und die Berechnungsvorschrift für den rekursiven Fall werden in den beiden folgenden Zeilen kodiert. Alle Operationen der Programmiersprache werden dabei in Präfixnotation geschrieben *(name(argumente))* und statt der mathematischen Symbole und verbalen Beschreibungen werden die Schlüsselwörter der Programmiersprache verwendet (z. B. *equal* für = und *if* für *wenn*).

2.2. Äquivalenz von Lehrmaterialien

Schmalhofer, Boschert & Kühn (1990) haben für das Lernen von syntaktischen Konzepten der Programmiersprache LISP gezeigt, daß Erklärungen und Beispiele zu vergleichbaren Transferleistungen führen. Um Lehrmaterialien bezüglich ihres Informationsgehaltes zu vergleichen, wird ein Konzept zum Vergleich der Informationsäquivalenz von Repräsentationen (Larkin & Simon 1987) verwendet: Zwei Lehrmaterialien sind informationsäquivalent, wenn alle Informationen, die in einem Lehrmaterial explizit gegeben sind, aus dem jeweils anderen Material inferiert werden können.

Für den Erwerb prozeduraler Fertigkeiten, wie dem Programmieren rekursiver Funktionen, gibt es keine eindeutigen Befunde über die Vergleichbarkeit der Lehrmaterialien. Beim Lösen mathematischer Textaufgaben zeigten sich Beispiellösungen deutlich überlegen (Reed & Bolstad 1991), beim Lernen deduktiver und statistischer Schlüsse (Cheng, Holyoak, Nisbett & Oliver 1986, Fong, Krantz & Nisbett 1986) waren Beispiele und verbal formulierte Erklärungen von Problemlöseregeln gleich hilfreich. Solche Ergebnisse sagen jedoch nie etwas eindeutig über die Angemessenheit eines Typs von Lehrmaterial aus, da der Unterstützungswert des Materials stark davon abhängt, wie der Experimentator sein Material auswählt.

Neben dem Konzept der Informationsäquivalenz führen Larkin & Simon (1987) auch das Konzept der komputationalen Äquivalenz ein: Zwei Materialien sind komputational äquivalent, wenn sie informationsäquivalent sind und zudem die explizite Information eines Materials „schnell und einfach" aus dem anderen Material inferiert werden kann. Diese Definition ist sehr vage. Eine mögliche Opeationalisierung wäre der Vergleich von Lösungszeiten in Abhängigkeit vom Lehrmaterial. Zudem müßte die Performanz bei Materialien, aus denen lösungsrelevante Informationen inferiert werden müssen, unter der von Materialien liegen, die die direkte Anwendung der relevanten Information ermöglichen.

Für die angemessene Vermittlung von Techniken rekursiven Problemlösens muß sowohl der Erwerb von Regeln zur Bildung von rekursiven Problemlösungen als auch der Aufbau von Rekursionsschemata unterstützt werden. Wie oben erwähnt, wird dabei angenommen, daß die Verfügbarkeit von Regeln zur Umsetzung eines Problems in einen rekursiven Lösungsplan wesentlich für den Problemlöseerfolg ist.

Gibt man Beispielfunktionen vor, wird das Finden der rekursiven Struktur des Problems direkt unterstützt, die Regeln zur Bildung der Rekursion aus einer nicht rekursiven Definition müssen dagegen inferiert werden. Eine mit der Fakultätsfunktion strukturgleiche Beispielfunktion ist die Multiplikation durch Addition, die folgendermaßen definiert ist:

multipliziere: $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$

$$\text{multipliziere}(x, y) = \begin{cases} 0 & \text{wenn } x = 0 \\ y + \text{multipliziere}((x-1), y) & \text{sonst} \end{cases}$$

für alle $x, y \in \mathbb{N}$.

Als der Fakultätsberechnung strukturgleiche Funktion würde entsprechend folgendes Beispiel vorgegeben:

```
FUN multipliziere(x:NAT,y:NAT):NAT
IF EQUAL(x,0)
THEN 0
ELSE PLUS(y,multipliziere(minus(x,1),y))
FIN.
```

Um das Beispiel auf die Lösung der Fakultätsfunktion zu übertragen, muß inferiert werden, daß zur Berechnung von *n!* die Zahl *n* zu minimieren ist, dann kann die Minimierung analog der Beispielfunktion erfolgen, und die Abbruchbedingung kann übernommen werden. Die Struktur des rekursiven Aufrufs: *„verknüpfe (wert, rekursiver Aufruf)"* kann ebenfalls übernommen werden, wenn erkannt ist, daß statt der Multiplikation die Addition verlangt wird und das neutrale Element die Zahl Null ist (weitere Beispiele siehe Anhang A und B).

Gibt man natürlichsprachige Erklärungen zur Problemstellung, kann die jeweilige Regel zur Bildung der Rekursion direkt gegeben werden, indem erläutert wird, welcher Parameter zu minimieren ist (z. B. *n* bei der Fakultätsfunktion) und mit welcher Operation die Minimierung erfolgen muß ($n - 1$), damit der direkte Fall ($n = 0$) erreicht werden kann (weitere Erklärungstexte siehe Anhang A und B):

Um die Fakultät einer Zahl zu berechnen, muß im Fall, daß die Zahl gleich Null ist, der hierfür definierte Wert (1) ausgegeben werden. Anderenfalls muß das Produkt der Zahl mit der Fakultät der um eins erniedrigten Zahl berechnet werden (rekursiver Fall). Die Erniedrigung der aktuellen Zahl gewährleistet gleichzeitig, daß der direkte Fall eintreten kann, also daß die Funktion terminiert.

Rekursionsschemata können beim Lernen mit Erklärungen inferiert werden, da jede erfolgreiche Problemlösung ein positives Beispiel für eine rekursive Struktur ist.

Nach diesen Überlegungen sind rekursive Techniken mit beiden Lehrmaterialien vermittelbar, die Lernleistungen nach Abschluß des Curriculums müßten also vergleichbar sein. Zudem dürfte bei informationsäquivalenten Materialien keines einen Lernvorteil bringen. Allerdings muß beim Lernen mit Beispielen die für die Problemlösung zentrale Umsetzung der Problemstellung in eine rekursive Definition selbst geleistet werden, während sie bei den Erklärungstexten explizit vorgegeben wird, die Materialien sind also nicht komputational äquivalent. Dementsprechend sollten die Personen, die mit Beispielen arbeiten, während des Lernzeitraums mehr Probleme mit der Lösung des Trainingsmaterials haben

als Personen, die mit Erklärungen arbeiten. Ein Hinweis darauf, daß die Regel der Rekursionsbildung zentral für die Lösung rekursiver Programmieraufgaben ist, läßt sich zusätzlich aus dem Hilfewahlverhalten von Personen ableiten, die beide Materialien zur Verfügung haben: Erklärungen sollten in diesem Fall meist vor Beispielfunktionen konsultiert werden.

3. Experiment

3.1. Die Lernumgebung LEAR

Um eine realistische Umgebung für die Entwicklung von Programmen zu schaffen, die gleichzeitig für Lernuntersuchungen mit Anfängern geeignet ist, wurde eine Lernumgebung (LEAR) entwickelt. Der Kern der Lernumgebung ist ein Interpreter für die oben erwähnte einfache funktionale Sprache (Field & Harrison 1988). Aus folgenden Gründen wurde statt der bekannten funktionalen beziehungsweise applikativen Sprachen wie LISP oder LOGO eine eigene Sprache verwendet: Zum einen sollte die Syntax der Sprache leicht erlernbar und die Semantik leicht verständlich sein, und zum anderen sollte die Sprache rein funktional und möglichst nah an algebraischen Konzepten orientiert sein (Ehrig & Mahr 1985).

In der Logik funktionaler Programmierung sind Programme eine Sammlung von Funktionen, wobei jede Funktion eine Menge von Eingabewerten (Parameter) auf einen Ausgabewert abbildet. Im Funktionsrumpf steht genau eine – beliebig komplexe – Berechnungsvorschrift, in der vordefinierte sowie selbstdefinierte Funktionen verwendet werden. Sequentielles Abarbeiten von Befehlen ist also nicht möglich. Die Kontrollstrukturen sind bedingte Anweisungen und die Rekursion (Endrekursion, Teil-Rest-Rekursion, indirekte Rekursion, Simultanrekursion). Schleifen und das damit verbundene Konzept der Iterationsvariablen sind nicht vorhanden. Zudem sind die Funktionen, anders als etwa in LISP, typisiert. Damit entfällt zwar ein wesentlicher Vorteil von Sprachen wie LISP und LOGO, die Flexibilität der Programme, andererseits wird der Programmierer gezwungen, die Funktionalität einer Operation klar zu definieren, und ein Erkennen typverletzender Operationen wird erleichtert.

Die in LEAR verwendete Sprache ist klar und überschaubar: Die Syntax beschränkt sich auf drei Datentypen (natürliche Zahlen, lineare Listen und Wahrheitswerte) und die notwendigen Grundoperationen (1) für Bedingungsprüfungen (Gleichheit?, größer als? und leere Liste?), (2) Verknüpfung von Wahrheitswerten (und, nicht), (3) Operationen auf natürlichen Zahlen (Addition, Subtraktion, Multiplikation) und (4) Operationen auf Listen (Listenkonstruktor, Selektion des ersten Elementes und Selektion der Liste ohne erstes Element).

Die Lernumgebung ist in vier Fenster geteilt: In einem Fenster wird eine Aufgabenstellung und Testwerte für die Ausführung der Funktion vorgegeben. In einem Editorfenster wird die Funktion kodiert. Ein weiteres Fenster meldet syntaktische Fehler (auch Typfehler) und semantische Fehler. Semantische Fehler werden durch internes Testen der Funktion mit kritischen Werten erkannt, so können fehlerhafte Berechnungen und Endlosrekursionen rückgemeldet werden. Dabei wird die Fehlerposition im Editorfenster angezeigt. Ist eine Funktion korrekt kodiert, kann sie in einem Interpreterfenster mit konkreten Testwerten ausgewertet werden.

Zudem werden für alle rekursiven Programmierprobleme Beispielfunktionen und/oder Erklärungstexte angeboten. Diese Hilfen werden, anders als bei tutoriellen Systemen, nicht aufgrund von Systementscheidungen gegeben, sondern von den Lernenden selbst durch Tastendruck angefordert. Um aus den aufgezeichneten Interaktionen (logfile recording) eindeutig schließen zu können, wann Hilfen gelesen wurden, müssen die Hilfen abgeschaltet werden, bevor weiter an der Programmieraufgabe gearbeitet werden kann.

3.2. Methode

Probanden. An der Untersuchung nahmen 49 Psychologiestudenten der TU Berlin teil. 31 Personen hatten Computererfahrung (Textverarbeitung, Statistikpakete), 4 dieser Personen hatten rudimentäre Programmierkenntnisse in imperativen Sprachen (BASIC, PASCAL).

Erfassung von Personendaten. Ein kurzer Fragebogen erfaßte demographische Merkmale der Person sowie ihre letzte Mathematiknote und eine Selbsteinschätzung der mathematischen Begabung. Zusätzlich wurden zwei Untertests des IST-70 (Amthauer 1970), die Zahlenreihen (ZR) und die Rechenaufgaben (RA), durchgeführt, um die Fähigkeit zum induktiven formalen Denken (ZR) und zum deduktiven Denken mit Zahlen (RA) zu kontrollieren.

Instruktionsmaterial. *(1) Bedienung der Lernumgebung.* Die Bedienung der Lernumgebung wurde in einem interaktiven Training eingeübt. *(2) Vermittlung der Syntax.* In die Syntax der Programmiersprache wurde durch einen mündlichen Vortrag eingeführt, der an einem kurzen, den Probanden vorliegenden Handbuch orientiert war. Die Probanden erhielten zusätzlich je ein Merkblatt zur Syntax der Programmiersprache und zur Bedienung der Lernumgebung, das sie während der gesamten Untersuchung benutzen konnten. Zudem wurde die Syntax durch das Programmieren einfacher, nicht rekursiver Funktionen und ein Frage-Antwort-Programm am Rechner eingeübt. *(3) Vermittlung rekursiver Techniken.* Nach einer kurzen mündlichen Einführung in das Konzept der Rekursion arbeiteten die Probanden nur noch mit der Lernumgebung. Die Probanden wurden zufällig einer der drei Lernbedingungen: Erklärungsbedingung ($n1 = 17$), Beispielbedingung ($n2 = 16$) und beide Lehrmaterialien ($n3 = 16$) zugewiesen. *(3a) Beispiele.* Für jede zu programmierende rekursive Funktion konnten zwei strukturgleiche Beispielfunktionen abgerufen werden. Die Beispiele unterschieden sich von der Aufgabenstellung durch Oberflächenmerkmale wie Namen von Funktion und Parametern, verwendete Verknüpfungsoperationen und zum Teil durch die Anzahl der Parameter. *(3b) Erklärungen.* Zu jeder Aufgabe standen zwei Erklärungstexte zur Verfügung. Beide Texte erläuterten die rekursive Struk-

Tabelle 1. Leistungsniveau der Probanden. Angegeben sind arithmetisches Mittel und Standardabweichung in Klammern

		Mathematikbegabung			IST-70	
		Syntaxtest	Note[a]	Selbsteinsch.[b]	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)

[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

Tabelle 2. Transferleistungen der Gesamtstichprobe ($N = 49$) in Abhängigkeit von der geprüften Wissensart

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

[a] Itemschwierigkeit [%] = (Anzahl gelöster Items/N) · 100

tur der Aufgabe mit direktem Fall, Ausgabe im direkten Fall und rekursiver Fall mit Minimierungsoperation. Der erste Text war eine kurzgefaßte formale Erläuterung, der zweite eine etwas ausführlichere Formulierung derselben Inhalte.

Testmaterial. *(1) Syntaxwissen.* Um beurteilen zu können, ob die drei experimentellen Gruppen sich nicht in bezug auf das erworbene Syntaxwissen unterscheiden und das erworbene Syntaxwissen zur Programmierung rekursiver Funktionen ausreichend war, wurde nach der Vermittlung der Syntax der Programmiersprache ein Syntaxtest durchgeführt, den die Probanden ohne Hilfestellung bearbeiten mußten. *(2) Rekursive Techniken. (2a) Performanz beim Lernprozeß.* Die Probanden lösten insgesamt sechs rekursive Aufgaben, drei endrekursive und drei teil-rest-rekursive. Für jede Aufgabe war die Bearbeitungszeit auf 30 Minuten beschränkt, bei Zeitüberschreitung galten die Aufgaben als nicht gelöst. *(2b) Transferleistung.* Das Verständnis rekursiver Programmierung wurde in einem Papier-und-Bleistift-Test erfaßt, der ohne Hilfsmaterial bearbeitet werden mußte. Die 21 Testitems erfaßten vier Aspekte rekursiven Verständnisses: (a) Abstraktes Wissen über Rekursion wurde in multiple-choice-Fragen erfaßt (7 Items). (b) Verständnis rekursiver Funktionen wurde über Auswahl korrekter rekursiver Lösungen für vorgegebene Aufgabenstellungen erfaßt (4 Items). (c) Verständnis des Ablaufs rekursiver Funktionen wurde durch Handsimulationen zur Berechnung von Funktionswerten für vorgegebene Eingabeparameter und zur Angabe von Eingabeparametern bei vorgegebenen Funktionswerten erfaßt (8 Items). (d) Schließlich sollten eine weitere endrekursive und eine teil-rekursive Funktion geschrieben werden (2 Items). Tatsächlich vermittelt wurde nur das Programmieren von Funktionen (Itemtyp d) und die zusätzlich in den Lernhilfen gegebenen Informationen (Itemtyp a). Es wurden also nicht nur deklaratives Wissen und die Anwendung des Gelernten auf weitere Aufgaben, sondern auch Transferleistungen geprüft.

Durchführung. Die Untersuchung fand in 6 Gruppen von etwa 8 Personen über 8 Stunden statt, und zwar in 4 zweistündigen Lektionen, die an aufeinanderfolgenden Tagen stattfanden. In der ersten Lektion wurden demographische Daten erhoben, die Untertests des IST-70 durchgeführt, der Umgang mit der Lernumgebung eingeübt und der erste Teil der Syntax, der Aufbau von Funktionen und das Kodieren einfacher arithmetischer Aufgaben, vermittelt. Es wurden zwei einfache, nicht rekursive Funktionen am Rechner kodiert. In der zweiten Lektion wurde der restliche Teil der Syntax vermittelt und durch ein interaktives Frage-Antwort-Programm sowie vier weitere nicht rekursive Programmieraufgaben eingeübt. In der dritten Lektion wurde zunächst der Syntaxtest durchgeführt und eine kurze Einführung in das Prinzip rekursiver Funktionen gegeben. Danach begann das eigentliche Experiment. Die Probanden kodierten mit ihrer jeweiligen Hilfestellung die drei endrekursiven Aufgaben. In der vierten Lektion kodierten sie die drei teil-rest-rekursiven Aufgaben. Nach einer Pause folgte dann der Abschlußtest zur Erfassung des Wissens über Rekursion.

3.3. Ergebnisse

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 1).

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 zu formalem Denken war bei den Probanden durchschnittlich. Im Schnitt war die letzte Mathematiknote eine 3, die Selbsteinschätzung auf einer fünfstufigen Ratingskala als mittelmäßig begabt, und die Intelligenztestskalen der IST-70-Untertests lagen etwa bei dem für Psychologen

vorausgesetzten Niveau (Skalenwert von 105 für beide Untertests; s. Amthauer 1970). In allen genannten Maßen schnitten die der Beispielbedingung zugeordneten Probanden etwas schlechter ab als die der beiden anderen 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 $z = -0{,}09$ für Itemmodell und $z = 0{,}83$ 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 Wissensaspekte wurden unterschiedlich gut beherrscht, wobei das selbständige Programmieren weiterer rekursiver Funktionen am wenigsten gelang (Tabelle 2).

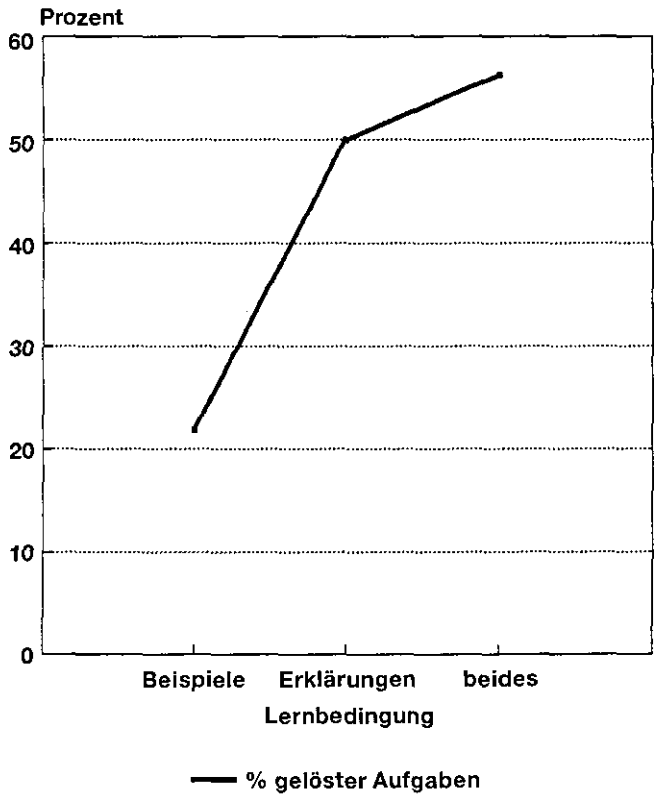

Abb. 1. Anzahl gelöster rekursiver Programmieraufgaben in Abhängigkeit von der Lernbedingung. $n1 = 17$; $n2 = n3 = 16$

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: $F = 54{,}28$, $p < 0{,}001$; Interaktion: $F < 1$).

Performanz beim Lernprozeß. Die Performanz wurde durch die Anzahl der korrekt programmierten rekursiven Funktionen der Lektionen 3 und 4 erfaßt. Bezüglich des Lösungserfolgs der rekursiven Programmieraufgaben bestanden signifikante Unterschiede zwischen den Lernbedingungen ($Chi^2 = 9{,}21$, $p < 0{,}001$), 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 (s. Abb. 1).

Zudem unterscheiden sich die Lerngruppen in der Art der Fehler, die sie beim Versuch, die Aufgaben zu lösen, produzieren: Probanden, die nur Erklärungen zur Verfügung haben, erzielen lediglich bei der Klammersetzung höhere Fehlerwerte (Erklärungsgruppe: 62, Beispielgruppe: 36, Gruppe mit beiden Lernhilfen: 31 Fehler; Kruskal-Wallis H-Test 9,561, $p < 0{,}05$). Zudem erzeugen 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, $p < 0{,}05$).

Probanden, die nur mit Beispielen arbeiteten, haben dagegen einen sehr hohen Anteil an semantischen Fehlern, die vor allem darauf zurückzuführen sind, daß einerseits relevante Strukturgleichheiten von Beispiel und Aufgabe nicht erkannt wurden und andererseits für die rekursive Struktur irrelevante Anteile der Beispiele übernommen wurden (Tabelle 3). So wird etwa bei der Aufgabe zur Berechnung des Summenwertes (*multipliziere*, s.

Tabelle 3. Vorkommen semantischer Fehler bei der Aufgabenlösung in Abhängigkeit von der Lernbedingung. Es sind absolute Fehlersummen aufgeführt, die aus der Analyse der Aufgabenlösungen von jeweils 13 Personen pro Lernbedingung hervorgingen

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, $p < 0{,}05$
keine oder falsche Minimierung (hätte von Beispielgruppe aus dem Beispiel übernommen werden können)	38	12	17	10,11, $p < 0{,}05$
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.

[a] B = Beispielgruppe, E = Erklärungsgruppe, EB = Gruppe mit beiden Lehrmaterialien
[b] Kruskal-Wallis H-Test

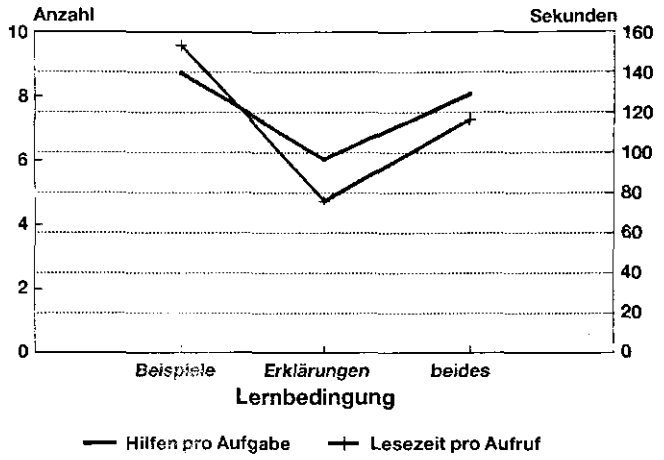

Abb. 2. Aufrufhäufigkeiten und Lesezeiten der Hilfen in Abhängigkeit von der Lernbedingung. $n1 = 17$; $n2 = n3 = 16$

Punkt 2.2) 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 der Aufgabenstellung funktioniert.

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 der Beispielgruppe ihre Hilfen häufiger und länger als die der beiden anderen Gruppen, die Unterschiede sind aber nicht signifikant (s. Abb. 2).

Probanden, die beide Hilfen zur Verfügung hatten, 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 sind für Beispiele 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.

Die Eingangshypothese, daß Erklärungen lösungsrelevantere Informationen enthalten als Beispiele und daß dementsprechend bei freier Auswahl Erklärungen den Beispielen vorgezogen werden, konnte deutlich bestätigt werden ($Chi^2 = 54{,}7$, $p < 0{,}001$). 81% Aufrufe von Erklärungen vor Beispielen stehen 9,5% Aufrufen von Beispielen vor Erklärungen gegenüber (in den verbleibenden 9,5% Fällen wurde keine Hilfe angefordert).

4. Diskussion

Es wurde verglichen, wieweit der Erwerb rekursiver Programmiertechniken durch Vorgabe von Beispielfunktionen und Erklärungstexten, die die rekursive Struktur des Problems erläutern, unterstützt wird. Dabei zeigte sich in einem Transfertest am Ende des Curriculums, daß die beiden Lernhilfen zum Aufbau vergleichbarer Wissensstrukturen führen. Das Arbeiten mit beiden Materialien bringt keinen Vorteil gegenüber der Verwendung nur einer der beiden Hilfearten. Die Materialien waren also informationsäquivalent im Sinne der Definition von Larkin & Simon (1987). Der Befund stimmt auch überein mit Ergebnissen einer Studie (Schmalhofer et al. 1990), bei der der Erwerb von LISP-Syntax aus Erklärungen und Beispielen verglichen wurde.

Andererseits zeigten sich während der Lernphase deutliche Unterschiede in Abhängigkeit vom Lehrmaterial. Personen, die mit Beispielen arbeiten, lösen deutlich weniger Aufgaben in der vorgegebenen Zeit als Personen, die mit Erklärungen oder beiden Materialien arbeiten. Die zur Problemlösung notwendigen Informationen können also aus Erklärungen einfacher inferiert werden als aus Beispielfunktionen. Die Materialien sind somit nicht komputational äquivalent. Dafür sprechen auch die höheren Aufrufhäufigkeiten und die längeren Lesezeiten von Beispielen im Vergleich zu den erklärenden Texten. Für die Notwendigkeit der in den Erklärungen vorgegebenen Regeln zur Bildung einer rekursiven Lösung spricht, daß Personen, die beide Materialien zur Verfügung hatten, fast immer zunächst eine Erklärung anforderten, bevor sie die Struktur der rekursiven Funktion mit Hilfe der Beispielfunktionen generierten. Zudem produzierten Personen, die nur mit Beispielen arbeiten, deutlich mehr semantische Fehler als die beiden anderen Lerngruppen. Diese Fehler kamen vor allem dadurch zustande, daß die Personen nicht in der Lage waren, strukturrelevante Anteile der Beispiele zu identifizieren. So wurden häufig mit der Aufgabe identische Beispielteile nicht übernommen (Abbruchbedingung, Minimierung), sondern stattdessen gerade die Anteile, die sich bei Aufgabe und Beispiel unterscheiden (Rückgabe im direkten Fall, Verknüpfungsfunktion bei der Teil-Rest-Rekursion).

Diese Befunde stützen die eingangs formulierte Annahme, daß Strukturinformation nur genutzt werden kann, wenn die Regeln zur Bildung der rekursiven Lösung einer Aufgabe bereits verfügbar sind. Das Lernen rekursiver Programmiertechniken kann nicht nur induktiv über Beispiele für vergleichbare Problemstellungen oder über Beispiele der eigenen Lerngeschichte erfolgen, sondern muß durch die Explikation der Regeln zur Bildung einer Rekursion unterstützt werden.

Ich danke Prof. Dr. Eyferth und zwei anonymen Gutachtern für wertvolle Anregungen, cand. psych. Daniela Ulbert und zahlreichen Studenten des Instituts für Psychologie der TU Berlin für die Unterstützung bei der Durchführung und Auswertung des Experimentes, cand. psych. Jens Grübener für die große Hilfe bei der Durchführung der Fehleranalysen.

Literatur

Amthauer, R. (1970). *Ist-70 – Handanweisung für die Durchführung und Auswertung.* Göttingen: Hogrefe

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

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

Bonar, J. & Cunningham, R. (1988). Bridge: an intelligent tutor for thinking about programming. In: J.Self (ed.), *Artificial Intelligence and Human Learning – Intelligent Computer-Aided Instruction* (pp.391–409). London: Chapman & Hall

Bonar, J. & Soloway, E. (1989). Preprogramming knowledge: A major source of misconceptions in novice programmers. In: E. Soloway & J.C.Spohrer (eds.), *Studying the Novice Programmer* (pp.325–354). Hilldale, NJ: Lawrence Erlbaum

Buchanan, B.G., Mitchell, T.M., Smith, R.G. & Johnson, C.R. (1977). Models of learning systems. In: J.Belzer, A.G.Holzman, & A.Kent (eds.), *Encyclopedia of computer science and technology* (Vol.11, pp.24–51). New York: Marcel Dekker

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

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

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

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

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

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

Kahney, H. (1989). What do novice programmers know about recursion? In: E.Soloway & J.C.Spohrer (eds.), *Studying the Novice Programmer* (pp.209–228). Hilldale: Lawrence Erlbaum

Larkin, J.H. & Simon, H.A. (1987). Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science, 11,* 65–99

Manna, Z. & Waldinger, R. (1975). Knowledge and reasoning in program synthesis. *Artificial Intelligence, 6,* 175–208

Mitchell, T.M., Keller, R.M. & Kedar-Cabelli, S.T. (1986). Explanation-based generalization: A unifying view. *Machine Learning, 1,* 47–80

Novick, L.R. & Holyoak, K.J. (1991). Mathematical problem solving by analogy. *Journal of Experimental Psychology: Learning, Memory, and Cognition, 17* (3), 398–415

Perl, J. (1979). *Rekursive Programmierung.* München: Hanser

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

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

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

Samuel, A.L. (1963). Some studies in machine learning using the game of checkers. In: E.A.Feigenbaum & J.Feldman (eds.), *Computers and Thought* (pp.71–105). New York: McGraw-Hill

Schmalhofer, F., Boschert, St. & Kühn, O. (1990). Der Aufbau allgemeinen Situationswissens aus Text und Beispielen. *Zeitschrift für Pädagogische Psychologie, 4* (3), 177–186

Vorberg, D. & Goebel, R. (1990). Das Lösen rekursiver Programmierprobleme: Rekursionsschemata. *Kognitionswissenschaft, 1,* 83–95

Weber, G., Waloszek, G. & Wender, K.F. (1988). The role of episodic memory in an intelligent tutoring system. In: J.Self (ed.), *Artificial Intelligence and Human Learning – Intelligent Computer-Aided Instruction* (pp.141–155). London: Chapman and Hall

Anhang A: Ein endrekursives Problem und dazugehörige Lernhilfen

Aufgabenstellung: Schreibe eine Funktion member, die prüft, ob ein Element n in einer Liste l enthalten ist. Ist das Element enthalten, dann soll die Funktion TRUE, sonst FALSE zurückliefern.

Beispiel: Die Funktion grex prüft, ob eine Zahl x größer ist als jedes Element einer Liste l:

```
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
```

Erklärung: Um festzustellen, ob eine Liste ein bestimmtes Element n enthält, muß die Liste Element für Element durchsucht werden. Das jeweils erste Listenelement wird auf Gleichheit mit n geprüft. Also muß die Liste im rekursiven Aufruf immer um ein Element verkürzt werden. Wurde die Liste ganz durchsucht – die Liste ist dann leer – und keine Übereinstimmung gefunden, so meldet die Funktion die Nichtmitgliedschaft von n in der Liste. Wird die Gleichheit zwischen dem aktuellen ersten Listenelement und n festgestellt, so ist die Arbeit der Funktion beendet und die Funktion meldet die Mitgliedschaft von n in der Liste.

Lösung:

```
FUN member (n:NAT,l:NATLIST):BOOL
IF EMPTY(l)
THEN FALSE
ELSE IF EQUAL(n,HEAD(l))
     THEN TRUE
     ELSE member(n,tail(l))
FIN.
```

Anhang B: Ein teil-rest-rekursives Problem und dazugehörige Lernhilfen

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

Beispiel: Die Funktion slist erzeugt eine Liste, die nmal 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 1 minimiert.

Lösung:

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