

# Automatisierte Unterstützung beim Lösen von LISP-Aufgaben

Studienarbeit zur Erlangung des Bachelor-Grades in  
Cognitive Science

Jan Plate, Universität Osnabrück

Oktober 2001

# Inhaltsverzeichnis

<b>0</b>	<b>Einleitung</b>	<b>3</b>
<b>1</b>	<b>Intelligente Tutorielle Systeme</b>	<b>3</b>
1.1	Zur Geschichte . . . . .	3
1.2	Wünschenswerte Eigenschaften von Programmier-ITS . . . . .	5
1.2.1	Kompetenzbereich . . . . .	5
1.2.2	Adaptivität der Lösungshinweise . . . . .	7
1.2.3	Verdeutlichung von Zusammenhängen . . . . .	8
1.2.4	Unterstützung beim Programmieren und Debuggen . . . . .	8
1.2.5	Benutzerführung und Benutzeroberfläche . . . . .	9
1.3	Allgemeiner Aufbau von Programmier-ITS . . . . .	9
<b>2</b>	<b>LISP</b>	<b>10</b>
2.1	Kurze Einführung in die Programmiersprache . . . . .	10
2.1.1	Zur Geschichte . . . . .	10
2.1.2	Grundzüge . . . . .	11
2.1.3	Grundlegende Datenstrukturen . . . . .	12
2.1.4	<i>Auswertung</i> . . . . .	13
2.1.5	Wichtige LISP-Operatoren und -Konstanten . . . . .	13
2.2	Typische Fehler von LISP-Anfängern . . . . .	14
2.2.1	Klammerungsfehler . . . . .	15
2.2.2	Parallelismen . . . . .	15
2.2.3	‘Komplizierte’ Operatoren . . . . .	17
<b>3</b>	<b>Andersons LISP-Tutor</b>	<b>18</b>
3.1	Theoretische Grundlagen . . . . .	18
3.2	Prinzipien des Designs von LISPITS . . . . .	19
3.3	Eigenschaften von LISPITS . . . . .	21
3.4	Kritik . . . . .	25
<b>4</b>	<b>SULLA</b>	<b>28</b>
4.1	Eigenschaften von SULLA . . . . .	29
4.1.1	Grundsätzliche Benutzungsweise . . . . .	29
4.1.2	Kompetenzbereich und Konfigurierung . . . . .	29
4.1.3	Überprüfung von Klammerung und Layout . . . . .	32
4.1.4	Evaluierung von Aufgabenlösungen . . . . .	32
4.1.5	Lösungshinweise . . . . .	35
4.2	Ausblick: Möglichkeiten der Weiterentwicklung . . . . .	37
<b>A</b>	<b>Der Algorithmus zur Überprüfung der Layout-Konsistenz</b>	<b>40</b>
A.1	Erstellung der ‘Prä-Konsistenzrepräsentation’ . . . . .	41
A.2	Erstellung der ‘Konsistenzrepräsentation’ . . . . .	42
A.3	Reparatur der gefundenen Inkonsistenzen . . . . .	44
<b>B</b>	<b>Erstellung von Musterlösungsdateien</b>	<b>49</b>

## 0 Einleitung

Das Problem, mit dem sich die vorliegende Arbeit befaßt, die „automatisierte Unterstützung bei der Lösung von LISP-Aufgaben“, wobei zu ergänzen wäre: „im Rahmen eines Programmierkurses an einer Schule oder Hochschule“, offenbart schon bei kurzem Hinsehen zwei Hauptaspekte: einen allgemeinen, nämlich die Unterstützung beim Lösen von im Rahmen eines (Hoch-)Schulkurses gestellten Programmieraufgaben überhaupt, und einen spezielleren, der in der Übertragung von Ansätzen zum allgemeinen Teil des Problems auf die Programmiersprache LISP besteht.

Beide Aspekte haben, relativ unabhängig voneinander, einen eigenen weiteren Kontext. Die aktive Unterstützung bei Programmieraufgaben ist nur ein kleiner, wenn auch recht zentraler, Teil des viel umfassenderen Problems der Entwicklung von intelligenten tutoriellen Systemen (abgekürzt ITS). Diesen ist der erste Abschnitt gewidmet, in welchem zunächst ein kurzer Abriß der historischen Entwicklung von ITS gegeben wird, bevor versucht wird, zu bestimmen, welche Anforderungen ein ITS zum Erlernen des Programmierens in einer bestimmten Sprache erfüllen sollte.

Der zweite, speziellere Aspekt ist natürlich eingebettet in den Kontext der Programmiersprache LISP. Nach einer kurzen Einführung in diese Sprache werden im zweiten Abschnitt diejenigen Besonderheiten von LISP behandelt werden, die bei Programmier-Anfängern typischerweise zu Fehlern führen. Aus diesen Fehlern resultieren bestimmte weitere Anforderungen an LISP-Tutorsysteme.

Im dritten und vierten Abschnitt dieser Arbeit werden zwei solche Systeme vorgestellt werden: Zunächst mit J. R. Andersons LISPITS das wohl bekannteste Programmier-ITS überhaupt, und im vierten Abschnitt dann SULLA, das im Rahmen dieser Arbeit entwickelte „System zur Unterstützung beim Lösen von LISP-Aufgaben“. Der Grund dafür, daß SULLA erst nach einer Beschreibung und Kritik von LISPITS vorgestellt wird, liegt zum einen darin, daß der Hintergrund, vor dem SULLA zu sehen ist, deutlich gemacht werden soll – und dieser Hintergrund wird nun einmal weitgehend von LISPITS beherrscht. Der zweite Grund besteht darin, daß eine Kritik von LISPITS auf der Basis der im ersten Abschnitt vorgebrachten Anforderungen in gewisser Weise die Existenzberechtigung für SULLA liefert. Ein in den vierten Abschnitt integrierter Ausblick wird schließlich als Abschluß der Arbeit einige Möglichkeiten diskutieren, wie SULLA weiterentwickelt werden kann und sollte.

## 1 Intelligente Tutorielle Systeme

### 1.1 Zur Geschichte

In den folgenden Absätzen soll dem Leser ein kurzer Überblick über die historische Entwicklung von intelligenten tutoriellen (oder Tutor-) Systemen gegeben werden, damit der allgemeine Kontext, in dem diese Arbeit entstanden ist, deutlicher hervortritt. Zugleich wird mein Ziel sein, dem Leser ein Bild davon zu vermitteln, was unter ITS im allgemeinen zu verstehen ist. Der Begriff „Intelligente Tutorielle Systeme“ (Intelligent Tutoring Systems) ist 1982 durch Sleeman und Brown geprägt worden ([SB82]), und zwar in einem Bericht über die bisherigen Forschungsarbeiten und Projekte zum Thema des Rechnergestützten Unterrichts (Computer- Aided Instruction, CAI). Mit CAI-Systemen hat denn auch die Geschichte der ITS angefangen, noch bevor man den Programmen das Attribut ‘intelligent’ zuschreiben konnte. Als sie während der 70er Jahre dann zunehmend anspruchsvoller, anpas-

sungsfähiger und in diesem Sinne eben ‘intelligenter’ wurden, hat sich der Begriff der ICAI, der *Intelligent Computer-Aided Instruction* eingebürgert. Jedoch ist dieser Begriff im Laufe der 80er Jahre von ITS“ verdrängt worden. Wenger, der 1987 eine in mancher Hinsicht beeindruckende Übersicht über den damaligen Stand der Technik in diesem Bereich gegeben hat ([Wen87]), rechtfertigt die Bevorzugung von „ITS“ gegenüber „ICAI“ damit, daß „die Bedeutung der Verschiebung in der Forschungsmethodik in vielerlei Hinsicht über die Hinzufügung eines ‘T’ zu ‘CAI’ hinausgeht“ (S. 3).

Wie gesagt, hat die Geschichte von ITS relativ bescheiden als „rechnergestützter Unterricht“ begonnen, und zwar bald nach dem Zweiten Weltkrieg in den Vereinigten Staaten. Folgt man der Darstellung von Urban-Lurain ([UL96]), scheint die Entwicklung von CAI-Systemen hauptsächlich von drei Faktoren angestoßen bzw. begünstigt worden: Erstens erfuhren die Ausbildungsinstitute der USA durch die aus dem Krieg heimkehrenden GIs, die dann auf Staatskosten die Colleges besuchen durften, eine ungeahnte Ausdehnung; Zweitens begann in den späten fünfziger und frühen sechziger Jahren, ebenfalls in den USA und mit enormer Euphorie, die Forschung auf dem Gebiet der Künstlichen Intelligenz, und Drittens hatte zur selben Zeit in der Psychologie die Sichtweise des Behaviorismus, die auch den Vorgang des Lernens auf einfache Stimulus-Reaktions-Verbindungen zurückführen wollte, eine unbestrittene Vormachtstellung inne.

Die Verbindung dieser drei Faktoren ließ den Gedanken, die Effizienz der Ausbildung in den Colleges und Universitäten mit Hilfe von Computern zu beschleunigen, nur logisch erscheinen. Die ersten, in den sechziger Jahren entstandenen Systeme zum rechnergestützten Unterricht bezogen sich vorwiegend auf Wissensgebiete wie Arithmetik und Fremdsprachen-Vokabularen, in denen sich auf einfache Weise Aufgaben generieren ließen, die von einem Schüler nur richtig oder falsch beantwortet werden konnten. Diese Antworten wurden von dem jeweiligen System in der Regel gesammelt und am Ende der Sitzung zusammengefaßt als Ergebnis ausgegeben. Während diese frühen System noch in keiner nennenswerten Weise die Präsentation des Lernstoffs bzw. den Typ und die Schwierigkeit von Aufgaben an die Fähigkeiten des Schülers anpaßten, begannen Forscher in den späten sechziger und frühen siebziger Jahren erstmals damit, den Ablauf von rechnerbasierten Lernprogrammen dynamisch den in seinen Antworten sich zeigenden Fähigkeiten des Schülers anzupassen. Es muß aber gesagt werden, daß in dieser Zeit die Entwicklung von CAI-Systemen noch ganz in der Hand von KI-Fachleuten oder sehr stark an der KI orientierten Kognitionswissenschaftlern lag. Auch wenn instruktionale Prinzipien bei der Entwicklung eine gewisse Rolle spielten, fand eine Beteiligung durch Spezialisten für Lernpsychologie oder eine ernstzunehmende Evaluation in Hinblick auf Lernerfolg nur selten statt, und hieran sollte sich laut Seidel und Park auch bis in die Mitte der 80er Jahre nicht viel ändern ([SP94]).

Ein klassisches ITS zum Erlernen einer Programmiersprache ist ab den frühen 80er Jahren von J.R. Anderson entwickelt worden: LISPITS, oder auch einfach LISP-Tutor. Dieses System ist entstanden im Rahmen eines Projektes zum Testen der Lernprinzipien von Andersons ACT\*-Theorie<sup>1</sup> ([And83, CA92]). Dementsprechend leiten viele der tutoriellen Prinzipien, die in LISPITS zum Tragen kommen, sich direkt von Annahmen dieser Theorie ab: ein Beispiel für den wachsenden Einfluß der Kognitionswissenschaft auf dem Gebiet der ITS in jener Zeit (s. a. [UL96]). Im dritten Abschnitt wird Andersons System näher behandelt werden. Die zweite Hälfte der 80er Jahre brachte den Höhepunkt des Interesses an ITS. Seidel und Park stellen für 1988 eine Zahl von 90 zu diesem Thema veröffentlichten Artikeln

---

<sup>1</sup>Die Buchstaben ACT stehen für *Adaptive Control of Thought*.

fest. In der Folge jedoch schwand die Aktivität auf dem Gebiet beträchtlich. Als Gründe hierfür lassen sich den Autoren zufolge sowohl das erstmalige Erscheinen von systematischen Kritiken an der bisher geübten Methodik (wie z.B. [PPS87]) anführen, als auch der Umstand, daß die bis dahin existierenden ITS nur einen beschränkten praktischen Nutzen vorweisen konnten. Zudem haben die mit der Entwicklung von ITS verbundenen Kosten und die benötigte technische Expertise viele Wissenschaftler und Geldgeber aus diesem Forschungsgebiet herausgedrängt.

Für den mit den frühen 90er Jahren beginnenden Zeitraum lassen sich in der Erforschung von Lehrstrategien und der ITS-Entwicklung laut [SP94] (S. 115ff.) fünf Trends erkennen, von denen drei als in unserem Rahmen relevant hier wiedergegeben werden sollen: Zum einen ist der Trend zu nennen, menschliches Lernen und Denken mit zunehmender Präzision in ITS zu simulieren. Bestrebungen in diese Richtung gibt es ja schon seit Andersons ersten Versuchen, seine ACT\*-Theorie als Grundlage verschiedener neuer ITS zu nehmen. In jüngerer Zeit hat jedoch auch Alan Newell in [New90] mit seinem SOAR-System einen Beitrag hierzu geleistet. Zweitens äußern sich Seidel und Park dahingehend, daß es sinnvoll wäre, in einem ITS nicht nur den Schüler zu modellieren, sondern auch zu erforschen, wie ein entsprechendes Lehrermodell aussehen könnte. Ansätze in diese Richtung erkennen sie beispielsweise bei Berliner ([Ber91]). Ein dritter Trend schließlich bestehe in einer Bewegung hin zu vermehrtem Einsatz von ‘multimedia environments’, die neben Text auch Geräusche, hochauflösende und farbreiche Graphik, interaktive Videosequenzen und sogar virtuelle Realität beinhalten können.

Als vierter Trend ließe sich dieser Liste hinzufügen, daß die Verfügbarmachung von ITS für den Zugriff per Internet unter dem Stichwort ‘web-based education’ während der 90er Jahre bis heute sehr viel Aufmerksamkeit auf sich gezogen hat ([Bru99]). Als Beispiel-Systeme zu nennen wären hier das an der Universität Osnabrück im Rahmen eines Studentenprojektes entstandene PLOT (Prolog-Lernen mit Online-Tutor, s. [BGH<sup>+</sup>99]) und zweitens der LISP-Tutor ELM-ART II (ELM Adaptive Remote Tutor, s. [WS97]), der an der Universität Trier entwickelt worden ist.

## 1.2 Wünschenswerte Eigenschaften von Programmier-ITS

Der Zweck eines ITS besteht im weitesten Sinne darin, einem Schüler mit den Mitteln der Informationstechnik möglichst effizient gewisse Kenntnisse und Fähigkeiten zu vermitteln. Bezogen auf die Domäne des Programmierens in einer bestimmten Sprache bedeutet dies, daß das System dem Schüler fundiertes Wissen über Syntax, Semantik und Techniken dieser Programmiersprache beibringt wie auch die Fähigkeit, in dieser Sprache gut und sicher zu programmieren. Von dieser Grundanforderung ausgehend, läßt sich sicherlich eine kaum überschaubare Zahl von Eigenschaften finden, die ein ideales und vollständiges Programmier-ITS aufweisen müßte. Die folgende Darstellung soll sich daher nur auf diejenigen beschränken, die von Relevanz für die vorliegende Arbeit, also für die Unterstützung bei der Lösung von Programmieraufgaben im Kontext eines Kurses an einer Schule oder Hochschule, sind.

### 1.2.1 Kompetenzbereich

Der Bereich (oder die Bereiche) der Fähigkeiten und Kenntnisse, die ein ITS zu vermitteln imstande ist, sollten möglichst nicht unnatürlich beschränkt sein.<sup>2</sup> So

---

<sup>2</sup>Dazu, was unter einer ‘unnatürlichen Beschränkung’ zu verstehen ist, kann man sicherlich sehr unterschiedlicher Auffassung sein. Eine sinnvolle Mindestanforderung stellt in diesem Zusam-

sollte ein ITS zum Erlernen einer bestimmten Programmiersprache Aufgaben zu möglichst allen Features dieser Sprache enthalten oder zumindest ohne größere Schwierigkeiten seitens des Anwenders, also z.B. des Übungsleiters oder Dozenten, in diese Richtung erweiterbar sein. Dies begründet sich zum einen darin, daß der Nutzen eines ITS offensichtlich von der Anzahl und Bedeutung der durch es vermittelten Fähigkeiten und Kenntnisse abhängt, und andererseits darin, daß es wenig wünschenswert wäre, wenn durch eine leichte Änderung oder Erweiterung des Lehrplans möglicherweise schon der Bedarf nach einem anderen System mit geeigneterem (und nach Möglichkeit umfassenderem) Kompetenzbereich entstünde.

Man könnte hier einwenden, daß ein umfassender Kompetenzbereich es einem Programmier-ITS unmöglich machen würde, eine angemessene Analyse von Schülerprogrammen durchzuführen, da die Anzahl der den Schülern zur Verfügung stehenden Operatoren einfach zu groß wäre. Diese Kritik würde aber übersehen, daß es auch bei einem Programmier-ITS, das den gesamten Umfang einer Sprache abdeckt, möglich wäre, bei weniger fortgeschrittenen Aufgaben den Gebrauch von bestimmten Operatoren zu unterbinden und die entsprechenden Schülerprogramme dadurch einer automatischen Analyse zugänglich zu halten.

Die Forderung nach einem möglichst nicht unnatürlich beschränkten Kompetenzbereich betrifft nicht nur die explizit im Kurrikulum eines Programmierkurses auftauchenden Punkte wie „Rekursion“, „Zeigeroperationen“ o.ä., sondern ganz besonders auch solche Aspekte des Programmierens, die im wesentlichen nur durch Übung erworben werden können, wie Problemanalyse, Algorithmenentwicklung (im folgenden unter ‘Programmmentwurf’ bzw. ‘-Planung’ zusammengefaßt) und Debugging. Ein ideales Programmier-ITS sollte daher seinem Benutzer beibringen, sein Programm selbständig zu entwerfen und zu korrigieren, oder ihm zumindest die Gelegenheit hierzu geben. Hierbei ist es allerdings angebracht, zwischen verschiedenen Arten von Tätigkeiten zu differenzieren, die bei diesen Prozessen erforderlich sind. Insoweit nämlich Entwicklungsumgebungen verfügbar sind, die eine bestimmte Art von Tätigkeit zuverlässig (also in *jeder* Programmiersituation und nicht nur im Kontext einer bestimmten Übungsaufgabe) ausführen können, wird auch die Fähigkeit des Programmierers zu solchen Tätigkeiten weniger wichtig.

Allerdings erscheint es utopisch anzunehmen, daß die Kapazitäten solcher Systeme in absehbarer Zeit über die allersimpelsten für Entwurf und Debugging von Programmen relevanten Fähigkeiten hinausgehen werden. In bezug auf Debugging etwa (von Programmmentwurf ist gar nicht erst zu reden) kann für die nächste Zeit kaum erwartet werden, daß es Programmierungsumgebungen geben wird, die nennenswert mehr leisten als die Bereitstellung von Informationen über den Ablauf eines Programmes (*trace*, *single-step* u. ä.) oder beispielsweise Erkennung von Schreibfehlern und Fehlern bezüglich der Art und Anzahl von Funktionsparametern. Algorithmen, die mehr leisten als dies, sind zwar entwickelt worden (s. z.B. [Sha83]), stellen aber an die zu analysierenden Programme einschränkende Anforderungen, die ihre Nutzung im allgemeinen impraktikal erscheinen lassen. Solange dies so ist, bleiben Programmplanung und Debugging ein wertvoller Bestandteil der Fähigkeiten eines Programmierers und sollte daher einem Schüler nicht durch Tutorssysteme aus der Hand genommen werden.

---

menhang wohl das Kriterium der praktischen Anwendbarkeit dar. Wenn der Kompetenzbereich eines ITS derart eingeschränkt ist, daß dadurch seine Anwendbarkeit im Rahmen eines gewöhnlichen Schul- oder Hochschulkurses beeinträchtigt wird, könnte man den Kompetenzbereich des Systems als ‘unnatürlich eingeschränkt’ bezeichnen.

### 1.2.2 Adaptivität der Lösungshinweise

Die Hinweise, die ein Tutorsystem geben kann, wenn ein Schüler bei der Lösung einer Aufgabe stockt, lassen sich i. a. in ein Spektrum einordnen, das von ‘sehr allgemein und abstrakt’ bis ‘äußerst konkret’ reicht. Mit der Konkretheit von Lösungshinweisen wächst in der Regel auch der Nutzen, die ein Schüler daraus für die Lösung seiner Aufgabe ziehen kann (wenn auch nicht unbedingt sein Nutzen in bezug auf seinen Lernerfolg). So wäre ein extrem abstrakter Hinweis etwa „Ihr Lösungsvorschlag ist nicht optimal“, wohingegen der denkbar konkreteste Hinweis wohl in der Bereitstellung der Musterlösung bestünde. Ein Tutorsystem sollte also den Grad der Konkretheit seiner Lösungshinweise an die Fähigkeiten des Schülers anpassen, womit allerdings weniger dessen allgemeine Programmierfähigkeiten gemeint sind, als vielmehr seine Fähigkeit, zum konkret gegebenen Zeitpunkt eben die Aufgabe zu bewältigen, bei deren Lösung er ins Stocken geraten ist. Dies bedeutet, daß das System zunächst nach Maßgabe seiner Einschätzung der Fähigkeiten des Schülers mehr oder weniger abstrakte Lösungshinweise geben sollte: umso abstrakter, je höher es diese Fähigkeiten einschätzt. Scheitert der Schüler erneut an der Aufgabenstellung, muß es seine Einschätzung nach unten revidieren und kann einen entsprechend konkreteren Hinweis geben. Dieses Muster der Interaktion zwischen Schüler und Tutorsystem entspricht weitgehend den Beobachtungen, die Weber und Waloszek ([WWW86]) in Übungssitzungen mit menschlichen Tutoren gemacht haben. Des weiteren sollte das Tutorsystem seine Lösungshinweise an die spezifischen Schwierigkeiten, die der Schüler bei seinen Lösungsversuchen zu haben scheint, anpassen. Diese Schwierigkeiten lassen sich nur zum Teil aus dem gescheiterten Lösungsversuch und dem Muster der Interaktion des Schülers mit dem ITS (wie etwa dem Zeitraum zwischen zwei aufeinanderfolgenden Abgaben von Lösungsversuchen) ableiten. Effizienter und darum wünschenswerter wäre hierbei die Fähigkeit des Tutorsystems, den Schüler selbst auf seine Schwierigkeiten hin zu befragen, also einen intelligenten Dialog über das zur Debatte stehende Problem zu führen.

Wenger ([Wen87], S. 408ff.) hat darauf hingewiesen, daß die Schwierigkeiten und Defizite eines Schülers sich grob auf drei Ebenen aufteilen lassen: die verhaltensbezogene, die epistemische und die individuelle. Letztere sollte vielleicht eher als die ‘psychologische’ bezeichnet werden, denn es geht hier vor allem um solche Aspekte wie Motivation, Lernattitüden und kognitives Selbstmodell. Die Hinweise, die ein Tutorsystem für die Lösung von Programmieraufgaben geben kann, sind dementsprechend ebenfalls jeweils einer dieser Ebenen zuzuordnen. Typischerweise bewegen sich diese Hinweise auf der ersten, verhaltensbezogenen Ebene, etwa wenn das ITS einen Schüler auf einen Fehler in seinem Programm hinweist, das ja als Manifestation des Schülerverhaltens aufgefaßt werden kann. Doch sind die Grenzen zur epistemischen Ebene fließend. Wenn nämlich der Lösungshinweis darin besteht, dem Schüler zu erklären, worin genau sein Fehler liegt und wie ähnliche Probleme im allgemeinen angegangen werden, ist schon nicht mehr nur das Verhalten, sondern bereits das Wissen des Schülers<sup>3</sup> das didaktische Ziel des Lösungshinweises. Dieser wäre damit der epistemischen Ebene zuzuordnen.

Schärfer jedoch scheint die Grenze zur psychologischen Ebene zu sein. Sogar soweit, daß Lösungshinweise, die sich explizit auf die Psychologie des Schülers beziehen, gemeinhin nicht mehr als ‘Lösungshinweise’ bezeichnet werden können. Als Beispiel im Kontext eines Programmier-ITS kann man sich hier den Rat vorstellen,

<sup>3</sup>Mit ‘Wissen’ ist hier nicht einfach nur eine Menge von Propositionen gemeint, sondern viel allgemeiner auch die Organisation, Vernetzung, Zugänglichkeit und Art der Repräsentation solcher Propositionen. Wenger selbst spricht daher (S. 410) nicht nur von *knowledge*, sondern von *knowledge state*.

der Schüler möge doch länger über seinen Lösungsvorschlag nachdenken, bevor er wieder das System um eine Diagnose bittet. Obwohl bei einem solchen Rat die Verbindung zum anstehenden Programmierproblem für gewöhnlich eine sehr lockere ist, handelt es sich dennoch um eine Hilfe bei der Lösung dieses Problems und spricht in unserem Beispiel in der mangelnden Bereitschaft des Schülers zur Reflexion über seine Lösungsversuche ein echtes und nicht zu verachtendes Defizit an. Zusammenfassend läßt sich also sagen: Ein ideales Programmier-ITS paßt die Hinweise und Ratschläge, die es dem Schüler für die Lösung einer Aufgabe gibt, sowohl an die aktuellen Fähigkeiten des Schülers an als auch an die Art seiner Schwierigkeiten (gleich ob epistemisch oder psychologisch).

### **1.2.3 Verdeutlichung von Zusammenhängen**

Ein System, das in der Lage ist, seinen Schüler beim Lösen von Programmieraufgaben durch flexibel an seine jeweiligen Schwierigkeiten angepasste Lösungshinweise unterstützt, ist nach heutigen Maßstäben bestimmt kein schlechtes. Was allerdings auch in dem Fall u.U. noch zu vermissen wäre, ist der Zusammenhang mit Aufgaben und Lektionen, die der Schüler bereits früher zu bewältigen hatte. Die bei solchen Gelegenheiten womöglich zunächst erfahrene Frustration und das schließliche Erfolgserlebnis, wenn die Lösung endlich gefunden worden ist, prägen die hierbei erlangten Kenntnisse und Fertigkeiten viel tiefer ein als dies bei der Lektüre von Lehrbuchtexten zu erwarten wäre. Diese Kenntnisse und Fertigkeiten sind daher prädestiniert, den Kern der vom Schüler zu erwerbenden Expertise zu bilden und verdienen somit besondere Aufmerksamkeit. In diesem Zusammenhang haben Weber und Waloszek ([WW86]) den interessanten Vorschlag gemacht, dem Schüler beim Auftreten von Fehlern in seinem Lösungsversuch Hinweise in Form von ‘Analogien’ oder ‘Erinnerungen’ an Situationen zu geben, in denen sich der Schüler früher schon einmal befunden hat. Auf diese Weise kann zum einen der Blick für die Zusammenhänge zwischen verschiedenen Problembereichen und Aspekten einer Programmiersprache geschärft werden, was an sich schon ein erstrebenswertes Ziel darstellt. Zum anderen bietet sich so die Möglichkeit, die in den einzelnen Aufgabensituationen erworbenen Kenntnisse und Fähigkeiten durch Erinnerung an die jeweilige Situation weiter zu festigen. Und drittens kann auch das neu zu erwerbende Wissen möglicherweise gründlicher verankert werden, wenn Assoziationen zu früheren Erfahrungen und bereits bestehenden Kenntnissen hergestellt werden.

### **1.2.4 Unterstützung beim Programmieren und Debuggen**

Immer, wenn es darum geht, daß jemand sich neue Fähigkeiten und Kenntnisse aneignen soll, ist es wichtig, daß er sich ganz auf den Lerninhalt konzentrieren kann. So hat sich gezeigt ([AJ85]), daß eine Überbeanspruchung des Kurzzeitgedächtnisses zu erheblicher Verlangsamung des Lernprozesses führen kann. Im Kontext von Programmier-ITS und speziell der Unterstützung beim Lösen von Programmieraufgaben bedeutet dies, daß ein ideales System dem Schüler die nebensächlichen, keine Expertise erfordernden Tätigkeiten, die mit dem Programmieren verbunden sind, so weit wie möglich abnimmt. Hierzu gehört beispielsweise die Überprüfung der Code-Syntax. Ein gutes Beispiel dafür, wie diese Tätigkeit von einem ITS übernommen werden kann, ist der in Andersons LISP-Tutor zum Einsatz kommende Editor, der den Code eigenständig strukturiert und für die Einhaltung einer (im wesentlichen) korrekten LISP-Syntax sorgt. Zusätzlich zu dem mächtigen Editor bietet dieses System auch ein Kontext-Fenster, das den Schüler über die bei einer



Programmieraufgabe zu lösenden Unterprobleme auf dem laufenden hält ([Wen87], S. 298f.).

### 1.2.5 Benutzerführung und Benutzeroberfläche

Es ist sicherlich wünschenswert, daß Lösungshinweise, auch wenn sie sich explizit nur auf das Verhalten oder Wissen des Schülers beziehen, ihre Wirkung auf die Psychologie, und hier vor allem auf die Motivation, des Schülers, nicht außer Acht lassen. Dies bezieht sich beispielsweise auf einen angemessenen (und nicht etwa herablassenden o.ä.) Ton, in dem Hinweise und sonstige Instruktionen zu halten sind, oder auch auf ihre Häufigkeit.

Abgesehen von angezeigten Texten tragen sicherlich auch eine ansprechende graphische Gestaltung sowie eine einfache, intuitive Bedienung der Benutzeroberfläche zur Motivation des Schülers bei. Zudem ist es vorstellbar, daß explizites Lob von seiten des Systems bei passenden Gelegenheiten den Schüler unter Umständen erheblich motivieren könnte, etwa bei einer gelungenen Aufgabenlösung, oder wenn der Schüler auf dem richtigen Weg zu einer Lösung scheint, oder wenn sich die Qualität seiner Aufgabenlösungen wesentlich verbessert hat.

## 1.3 Allgemeiner Aufbau von Programmier-ITS

Ausgehend vom in dieser Arbeit angenommenen Zweck und Anwendungskontext eines Programmier-ITS sowie den im vorigen Abschnitt ausgeführten wünschenswerten Eigenschaften legt sich eine allgemeine Architektur solcher Systeme nahe, die im wesentlichen folgende Komponenten umfaßt:

- Eine Benutzeroberfläche für den Kursleiter, in dem dieser die nötigen Voreinstellungen vornehmen kann, um das System für seine spezifischen Unterrichtszwecke einzurichten.
- Eine zweite Benutzeroberfläche für die Interaktion mit dem Schüler.
- Ein Experten-Modul, d.h. eine (wie auch immer geartete) Datenbank zur Repräsentation von Kenntnissen über die betreffende Programmiersprache. Diese können bereits in Form von Lektionen repräsentiert sein.
- Eine Sammlung von Informationen über im Rahmen des Kurses zu bearbeitende Übungsaufgaben, die entweder von vornherein Bestandteil des Systems ist oder vom Kursleiter bereitgestellt wird. Sie sollte durch letzteren zumindest editierbar sein (s. 1.2.1). Neben den Aufgabentexten sind hierin vor allem Anweisungen dazu enthalten, wie die Aufgaben dem Schüler präsentiert werden und wie das System auf deren Bearbeitung durch den Schüler reagieren soll (etwa in welchen Situationen welche Art von Lösungshinweisen oder Fehlermeldungen gegeben werden soll).
- Ein wichtiger Teil dieser Informationen zu den verfügbaren Aufgaben sind oftmals die entsprechenden Musterlösungen. Diese werden vor allem vom Diagnose-Module benötigt, um die vom Schüler erzeugte Lösung zu testen. Auf Musterlösungen kann nur dann verzichtet werden, wenn bereits aufgrund des gespeicherten Expertenwissens in jedem Fall eine hinreichende Bewertung von Schülerprogrammen generiert werden kann. Dies würde jedoch entweder einen sehr hohen Aufwand bei der Bereitstellung des Expertenwissens oder eine starke Beschränkung der als Input akzeptierten Schülerprogramme

implizieren. Beides ist aufgrund von Abschnitt 1.2.1 als problematisch einzuschätzen. Weiter unten (vor allem in Abschnitt 3.4) wird hiervon noch zu sprechen sein.

- Ein Diagnose-Modul zum Testen und Bewerten von Schüler-Programmen sowie zur Lokalisierung von Programmier- und ggf. Layoutfehlern.
- Ein Interpreter oder Compiler für die betreffende Programmiersprache. Auf diesen greift das Diagnose-Modul zu, um den Output sowie ggf. Speicher- und Zeitaufwand von Muster- und Schülerlösung miteinander zu vergleichen.
- Ein Didaktik-Modul. Dieses steht insofern im Zentrum des Systems, als hier die gesamte Ausgabe an den Schüler zusammengestellt wird und außerdem Anfragen des Schülers hier bearbeitet werden. So erstellt das Didaktik-Modul beispielsweise auf Anfrage des Schülers hin Lektionen anhand der im Experten-Modul repräsentierten Informationen (sofern diese nicht schon die Form von Lektionen haben), oder, ggf. unter Einbeziehung der Musterlösungen oder anderer Informationen zu den Übungsaufgaben, Hinweise zu deren Lösung. Des weiteren wird hier die vom Diagnose-Modul erzeugte Bewertung des Schüler-Programms für die Ausgabe aufgearbeitet.
- Bei all diesen Tätigkeiten wird das Didaktik-Modul vom Schülermodell beeinflusst. Dieses hat die Aufgabe, neben anderen Informationen über den Schüler vor allem dessen bisherigen Kenntnisstand zu repräsentieren, damit sich das ITS möglichst gut an die spezifischen Bedürfnisse des Schülers anpassen kann (s. Abschnitt 1.2.2).

Abbildung 1 veranschaulicht diese Gliederung sowie die genannten Beziehungen, die zwischen den einzelnen Komponenten bestehen.

## 2 LISP

### 2.1 Kurze Einführung in die Programmiersprache

Der folgende Abschnitt soll einen kurzen Überblick über die wichtigsten Züge von LISP geben für diejenigen Leser, die mit dieser Programmiersprache bisher noch keine Erfahrung hatten. Elementare Kenntnisse der Informatik werden jedoch vorausgesetzt.

#### 2.1.1 Zur Geschichte

Der Name ‘LISP’ ist als Abkürzung für *list processing* entstanden und steht für die nach FORTRAN älteste noch immer gebräuchliche Programmiersprache der Welt. Bis Anfang der 90er Jahre ist LISP die unangefochten führende KI-Programmiersprache gewesen und behauptet als solche neben PROLOG auch heute noch ihren Platz. Die Entwicklung von LISP begann in der zweiten Hälfte der 50er Jahre unter der Führung von John McCarthy, im wesentlichen am berühmten *Massachusetts Institute of Technology* (MIT). In den folgenden Jahrzehnten hat sich die Sprache in eine Vielzahl von Dialekten aufgesplittert, ist aber Ende der 80er in ANSI COMMON LISP standardisiert worden (s. [Ste90]). Wenn in dieser Arbeit also außerhalb eines historischen Kontexts von LISP die Rede ist, ist damit COMMON LISP gemeint. Ausführlichere Darstellungen der Entwicklung von LISP findet man in [McC81] und [SG93].

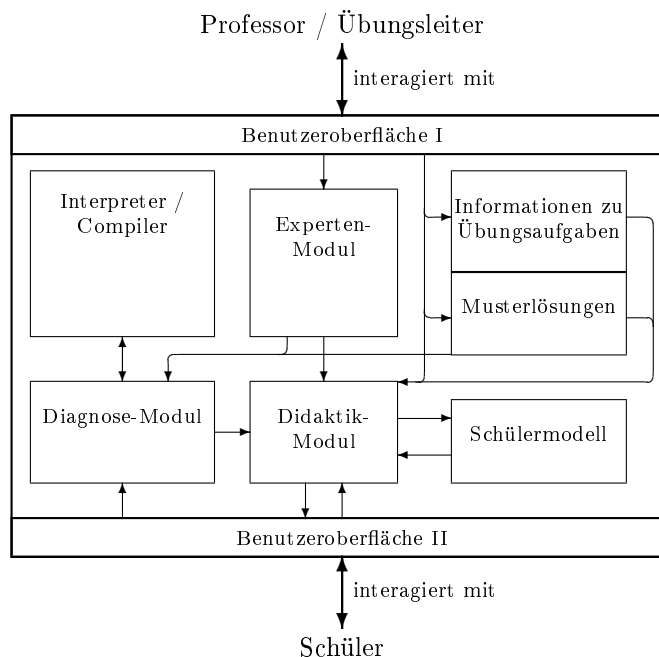


Abbildung 1: Allgemeiner Aufbau eines Programmier-ITS

### 2.1.2 Grundzüge

Der traditionell vorherrschende Programmierstil in LISP ist der des *funktionalen Programmierens*, weshalb LISP auch oft als ‘funktionale Programmiersprache’ bezeichnet wird. Entsprechend sind die meisten Operatoren in LISP seiteneffektfreie Funktionen<sup>4</sup>. Es ist allerdings auch durchaus möglich, (sog. *destruktive*) Funktionen zu definieren, die ihre Argumente, globale (in LISP *speziell* genannte) Variablen oder sogar die Arbeitsweise bereits definierter Funktionen verändern.

Im folgenden sollen kurz einige weitere Grundzüge von LISP aufgezählt werden, die diese Sprache von anderen abheben:

- **Umfang.** LISP ist eine von der Anzahl der Operatoren und Konzepte her ausgesprochen umfangreiche Programmiersprache. Die Liste der im Index von [Ste90] aufgeführten Funktionen, Makros, Spezialoperatoren, globalen Variablen und ‘loop-Klauseln’ umfaßt 994 Einträge, und so gut wie alle fallen in die drei Bereiche Ein- und Ausgabe, Datenmanipulation und Kontrollstrukturen. Bibliotheken für die Programmierung von Benutzeroberflächen, Datenbanken, Netzwerken usw. sind im LISP-Standard dagegen auf absehbare Zeit nicht enthalten.
- **Schwaches Typenkonzept.** In LISP ist es nicht nötig, den Typ einer verwendeten Variable anzugeben. (Man kann es aber tun, um dem Compiler bei seiner Aufgabe zu helfen, effizienten und sicheren Code zu produzieren.)

<sup>4</sup>Daß eine Funktion keine Seiteneffekte hat, bedeutet erstens, daß sie ihre Argumente nicht verändert, sondern vielmehr auf andere, ggf. neu erzeugte, Objekte *abbildet*. Zweitens muß eine solche Funktion, wenn sie ein weiteres Mal mit den gleichen Argumenten aufgerufen wird, auch die gleichen Rückgabewerte liefern.

- Umfassendes und erweiterbares Objekt/Typensystem. Schlichtweg alles in LISP läßt sich in derselben Hierarchie von Objekten einordnen. Zudem hat der Programmierer die Möglichkeit, fast beliebig neue Objektklassen oder Typen zu definieren.
- Reflexivität. LISP bietet sehr viele Möglichkeiten zur Analyse und Manipulation von Programmen. Dies liegt vor allem daran, daß Funktionen in LISP lediglich eine bestimmte Art von Datenstruktur sind. LISP-Programme können sich somit zur Laufzeit selbst analysieren und verändern (wenn dies denn gewünscht wird).
- Syntaktische Flexibilität. Über Makros und Veränderungen der `read`-Tabelle ist es möglich, die Syntax des eigenen LISP-Codes fast völlig frei zu gestalten.

### 2.1.3 Grundlegende Datenstrukturen

Daß LISP ausgeschrieben und übersetzt schlicht ‘Listenverarbeitung’ bedeutet, hängt damit zusammen, daß die wohl wichtigste Datenstruktur in LISP-Programmen die (einfach verkettete) Liste ist. Dies zeigt sich schon in der Einfachheit, wie in LISP solche Listen repräsentiert werden: Eine Liste, die nacheinander die Elemente  $x$ ,  $y$  und  $z$  enthält, wird als  $(x\ y\ z)$  ausgegeben. Wird umgekehrt eine solche Darstellung, etwa als Teil eines Programms, vom LISP-Interpreter gelesen, wird die entsprechende Liste im Speicher erzeugt. Listen spielen in LISP tatsächlich eine eminent wichtige Rolle: Nicht nur werden sie, zumindest in traditionellen LISP-Programmen, für die verschiedensten Zwecke herangezogen und auch dort benutzt, wo Programmierer anderer Sprachen viel eher zu *Arrays* oder Hashtabellen greifen würden (wenngleich in modernen LISP-Implementationen auch solche Datenstrukturen nicht fehlen), sondern *jeder einzelne Prozedur-Aufruf* ist nichts anderes als ein – oftmals tief verschachtelter – Listenausdruck. Da weiterhin Programmdateien in LISP lediglich aus aufeinanderfolgenden Prozeduraufrufen bestehen, haben LISP-Programme also einfach die Form einer Ansammlung hintereinander geschriebener Listenausdrücke und sehen damit beispielsweise so aus (meistens nur viel länger):

```
(defvar *euro->$* 0.92)

(defun euro->$ (euro)
  (* euro *euro->$*))
```

Der Grund dafür, daß Listenausdrücke gleichzeitig Prozedur-Aufrufe sein können, liegt darin, daß jedesmal, wenn der LISP-Interpreter eine ‘Top-level’-Repräsentation eines Objekts eingelesen hat (d.h. eine, die nicht in einen anderen Ausdruck eingeschachtelt ist), diese dann rekursiv *ausgewertet* wird. Die Auswertung einer Liste geschieht dabei so, daß das erste Element, falls vorhanden, als ein Operator aufgefaßt wird, und die restlichen Elemente dessen Argumente darstellen.

Im Normalfall handelt es sich bei dem Operator um ein *Symbol*. Symbole sind in LISP nicht einfach irgendwelche ‘Zeichen’, sondern stellen in LISP vielmehr neben den Listen die wichtigsten Datenstrukturen dar. Ebenso wie eine Liste wird ein noch nicht im Speicher vorhandenes Symbol einfach dadurch erzeugt, daß man seine Repräsentation vom Interpreter einlesen läßt.<sup>5</sup> Das obige Kode-Beispiel etwa enthält Repräsentationen für folgende Symbole: `defvar`, `*euro->$*`, `defun`,

<sup>5</sup>Die Analogie endet allerdings schon beim Einlesen einer zweiten Repräsentation des Symbols. Diese wird nämlich so interpretiert, daß sie sich auf exakt dasselbe bezieht, und nicht etwa auf ein neu zu erzeugendes Symbol mit demselben Namen. Im Unterschied dazu erzeugt jeder Ausdruck der Form  $(\dots)$  eine neue Liste, auch wenn zwischen den Klammern exakt dasselbe steht.

`euro->$`, `euro` und `*`. Auf die Komponenten, aus denen ein Symbol besteht, soll hier nicht weiter eingegangen werden. Im nächsten Abschnitt wird aber von ‘Funktionen’ und ‘Makros’ die Rede sein, die mit einem Symbol ‘verknüpft’ werden können, sowie von dem Wert, zu dem sich ein Symbol auswertet, wenn es als ‘spezielle Variable’ auftaucht. Übrigens macht LISP bei der Repräsentation von Symbolen im Normalfall keinen Unterschied, ob diese groß oder klein geschrieben werden. Als Programmierer benutzt man der Lesbarkeit halber meistens Kleinbuchstaben, aber in der Regel verwendet LISP für die internen Namen dieser Symbole statt dessen Großbuchstaben.

#### 2.1.4 Auswertung

Wenn eine nicht-leere Liste ausgewertet wird, faßt, wie schon erwähnt, der LISP-Interpreter ihr erstes Element als Operator auf. Bezüglich der Art dieses Operators gibt es nun drei Möglichkeiten. Wenn er auf eine Funktion verweist (der häufigste Fall), werden die restlichen Elemente der Liste ausgewertet, und die Ergebnisse dieser Auswertung bilden dann die an die Funktion zu übergebenden Argumente. Ist der Operator dagegen mit einem Makro verknüpft, werden die Elemente direkt an dieses übergeben. Die dritte Möglichkeit bilden ‘Spezial-Operatoren’, die auf besondere und nicht unbedingt in LISP selbst definierbare Prozeduren verweisen.

Im Gegensatz zu Spezial-Operatoren lassen sich Funktionen und Makros aber definieren, und ein Beispiel einer solchen Funktionsdefinition ist der `defun`-Ausdruck in dem oben gezeigten Miniaturprogramm. Das Symbol `defun` steht für ein Makro, das im wesentlichen nichts weiter tut, als ein übergebenes Symbol mit einer in demselben Makro-Ausdruck spezifizierten Funktion zu verknüpfen. So wird in unserem Beispiel das Symbol `euro->$` mit einer Funktion verknüpft, die ihr Argument, hier dargestellt durch das Symbol `euro`, abbildet auf den Resultatwert des Ausdrucks `(* euro *euro->$*)`. Die Liste `(euro)` ist dazu da, anzugeben, daß `euro` in der folgenden Definition als Platzhalter für das Argument der Funktion verwendet wird. Bei dem Ausdruck `(* euro *euro->$*)` schließlich handelt es sich um einen Aufruf der Funktion `*`, die das Produkt ihrer (numerischen) Argumente liefert. Im vorliegenden Fall ist dies zum einen das Argument der Funktion `euro->$`, zum anderen der Umrechnungsfaktor, zu dem sich das Symbol `*euro->$*` auswertet. Mit dem `defvar`-Ausdruck am Anfang des ‘Programms’ wird `*euro->$*` das Zahlobjekt 0.92 zugewiesen, aber diese Zuweisung könnte genauso gut auch am Ende des Programms stehen, da die Auswertung des in der Funktionsdefinition von `euro->$` stehenden Symbols `*euro->$*` *dynamisch*, also jeweils zum Zeitpunkt des Aufrufs von `euro->$` erfolgt. Man sagt, daß das Symbol `*euro->$*` an dieser Stelle die Rolle einer *dynamischen* oder auch (genauer) einer *speziellen* Variable spielt. Dagegen taucht `euro` hier nur als *lokale* Variable, sozusagen als reiner Platzhalter, auf. Das Thema der verschiedenen Arten von Variablen und der Regeln, die bestimmen, unter welchen Umständen in LISP welche Objekte referenziert werden können, ist verhältnismäßig umfangreich und sehr abstrakt. Das hier dazu Gesagte wird aber für das Verständnis der vorliegenden Arbeit genügen.

#### 2.1.5 Wichtige LISP-Operatoren und -Konstanten

In diesem Abschnitt sollen schließlich einige der wichtigsten LISP-Operatoren und -Konstanten vorgestellt werden, da deren Kenntnis für das Verständnis von Beispielen in den folgenden Abschnitten hilfreich sein wird. Die Operatoren `defun`, `defvar` und `*` sind bereits hinreichend erläutert worden. Unter Konstanten versteht

man globale Variablen, deren Werte nicht verändert werden können. Darunter fallen zum größten Teil solche Variablen, die bestimmte Eigenschaften des jeweiligen Interpreters ausdrücken (z.B. `most-negative-fixnum`), aber die wichtigsten sind sicherlich `nil` und `t`:

- `nil`. Diese Konstante spielt in LISP die besondere Rolle, sowohl als Synonym für die leere Liste `()` als auch für den Wahrheitswert *falsch* zu stehen. Ihr Wert ist sie selbst (d.h. das Symbol `nil`).
- `t`. Wenn `nil` für *falsch* steht, sollte es auch ein anderes Objekt geben, das *wahr* symbolisiert. Zwar steht in LISP alles, was ‘non-nil’ ist, für *wahr*, aber üblicherweise wird hierfür das eigens zu diesem Zweck als Konstante deklarierte Symbol `t` verwendet. Wie `nil` wertet sich `t` zu sich selbst aus.
- `quote`. Dies ist ein Spezialoperator, der nichts anderes tut, als sein Argument unausgewertet zurückzugeben. Er ist so nützlich, daß er durch das Zeichen ‘*abgekürzt*’ werden kann, derart, daß ‘*Ausdruck*’ gleichbedeutend ist mit `(quote Ausdruck)`.
- `car` und `cdr`. Diese beiden Funktionen sind die beiden Operatoren, mit denen in LISP traditionell auf das erste Element bzw. den Rest einer Liste zugegriffen wird. Synonyme sind `first` und `rest`.
- `cons`. Eine Funktion mit zwei Argumenten, die diese zu einer Liste zusammenfügt, sofern das zweite bereits eine Liste ist. Das erste Argument wird zum ‘*car*’ (d.h. zum ersten Element) der resultierenden Liste, das zweite bildet ihren Rest.
- `cond`. Dieses Makro wird meistens herangezogen, wenn in LISP eine bedingte Programmverzweigung implementiert werden soll. In einem `cond`-Ausdruck stehen nach dem `cond` typischerweise mehrere sog. *Klauseln*, worunter hier einfach Listen auswertbarer Ausdrücke zu verstehen sind. Bei der Auswertung werden nacheinander die jeweils ersten Ausdrücke dieser Klauseln ausgewertet, bis zu der ersten Klausel, deren erster Ausdruck nicht `nil` ergibt. Von dieser Klausel werden dann auch die restlichen in ihr enthaltenen Ausdrücke ausgewertet, und der `cond`-Ausdruck gibt genau das zurück, was der letzte Ausdruck in dieser Klausel liefert. Wenn in keiner Klausel der erste Ausdruck etwas anderes als `nil` ergibt, liefert der `cond`-Ausdruck `nil`, aber aus stilistischen Gründen sollte dies vermieden werden.
- `eq`. Eine Vergleichsfunktion, die genau dann nicht `nil` liefert, wenn ihre beiden Argumente identisch sind, also dasselbe Objekt sowohl ihr erstes als auch ihr zweites Argument ist. Objekte, die auf dieselbe Weise repräsentiert werden, sind darum noch keineswegs ‘*eq*’.
- `eq1`. Ebenfalls eine Vergleichsfunktion, aber etwas ‘liberaler’ als `eq`.
- `equal`. Grob gesprochen, sind zwei Objekte genau dann `equal`, wenn sie durch dieselbe Zeichenkette repräsentiert werden.

## 2.2 Typische Fehler von LISP-Anfängern

In diesem Abschnitt soll untersucht werden, welche Anforderungen an ein LISP-ITS sich aus den spezifischen Eigenschaften dieser Programmiersprache (im Unterschied zu anderen Sprachen) ergeben. Im Hinblick auf die im vorigen Abschnitt aufgeführten Anforderungen an Programmier-ITS, und zwar insbesondere die Punkte

1.2.2 und 1.2.4, stellt sich hier also vor allem die Frage, welche besonderen Schwierigkeiten einem Programmier-Schüler im Umgang mit LISP begegnen.

### 2.2.1 Klammerungsfehler

J. R. Anderson und R. Jeffries ([AJ85]) haben 1985 eine Studie veröffentlicht, in der Probanden, die zuvor noch gar keine oder nur sehr wenig Erfahrung mit LISP hatten, einfache Aufgaben mit Listenfunktionen (`cons`, `list` und `append`) zu lösen hatten. Dabei hat sich gezeigt, daß die Hauptfehlerursache bei den untersuchten Aufgabentypen<sup>6</sup> in einer Überlastung des Arbeitsspeichers der betreffenden Versuchsperson zu suchen ist. Auffällig war, daß den Probanden bei den Aufgabentypen, bei denen dies möglich war, vor allem Klammerungsfehler unterlaufen sind, und zwar solche, die in einem Weglassen oder Hinzufügen von Paaren korrespondierender Klammern bestanden. Und „sogar die als ‘sonstige’ klassifizierten Fehler involvieren vorrangig falsche Klammerung“ (S. 112). Dies zeigt relativ deutlich, daß für LISP-Anfänger (und womöglich auch erfahrenere Programmierer) die hohe Anzahl von Klammern, die das Schreiben von LISP-Programmen typischerweise – und in deutlich höherem Maß als das Programmieren in irgendeiner anderen Sprache – erfordert, besondere Schwierigkeiten birgt.

Im Hinblick auf die Frage, wie ein LISP-ITS diesen Schwierigkeiten gerecht werden kann, sind mehrere Ansätze denkbar. Zunächst ist aber wohl zu bedenken, daß das Vermeiden, Finden und Korrigieren von Klammerungsfehlern keine Tätigkeit ist, die zum Kern dessen gehört, was es heißt, in zu LISP programmieren. Wie das im Rahmen dieser Arbeit vorgelegte System SULLA demonstriert, kann diese Aufgabe außerdem wenigstens zu einem großen Teil automatisch erledigt werden. Darum erscheint es vom Standpunkt der Entwicklung eines ITS weniger wichtig, daß die Benutzer dahingehend trainiert werden, weniger Klammerungsfehler zu begehen bzw. diese schneller zu finden, sondern vielmehr sollte ein ITS seine Benutzer so gut wie möglich darin *unterstützen*, derartige Fehler zu vermeiden und ggf. zu korrigieren. Dafür, wie dies zu erreichen ist, bieten sich wiederum verschiedene Möglichkeiten an, die hier aber nicht weiter behandelt werden sollen.

### 2.2.2 Parallelismen

R. Pea ([Pea86]) berichtet, daß bei Programmieranfängern unabhängig von der jeweils verwendeten Sprache bestimmte Arten konzeptueller Fehler auftreten. Er unterscheidet drei Klassen solcher Fehler: ‘Parallelismen’ (*parallelism bugs*), ‘Egozentrismen’ (*egocentrism bugs*) und ‘Intentionalismen’ (*intentionality bugs*). Alle drei Klassen sind laut Pea auf einen gemeinsamen Grundfehler (*superbug*) zurückzuführen, nämlich den Umstand, daß Programmieranfänger in schwierigen Situationen mangels Erfahrung dazu neigen, auf die Analogie der Kommunikation mit einem Menschen zurückzugreifen.

Unter ‘Parallelismen’ versteht Pea in diesem Zusammenhang Fehler, die sich daraus ergeben, daß der Programmierschüler fälschlicherweise annimmt, mehrere Zeilen eines Programms seien gleichzeitig ‘aktiv’. Er verdeutlicht dies u.a. am Beispiel des folgenden kurzen BASIC-Programms:

---

<sup>6</sup>Die von Anderson und Jeffries untersuchten Aufgabentypen sind zwar nicht eigentlich als ‘Programmieraufgaben’ zu bezeichnen, kommen solchen aber hinreichend nahe, um für die gegenwärtige Betrachtung relevant zu sein. Im einzelnen handelt es sich dabei um verschiedene Arten der Ergänzung von ‘LISP-Gleichungen’ der Form „Der Aufruf (*Operator Argumente*) liefert *Rückgabewert*“. Je nach Aufgabentyp haben die Probanden aus den gegebenen beiden ‘Unbekannten’ einer solchen Gleichung entweder auf den Operator, auf die Argumente oder auf den Rückgabewert zu schließen.

```

IF SIZE = 10, THEN PRINT "HELLO
FOR SIZE = 1 to 10, PRINT "SIZE
NEXT SIZE

```

Pea berichtet von einer Studie, in der die Versuchspersonen (Schüler einer amerikanischen High School im zweiten Jahr Informatik) in 8 von 15 Fällen der Meinung waren, daß am Ende der FOR-Schleife dieses ‘Programms’ tatsächlich HELLO ausgegeben würde. Dies ist natürlich nicht der Fall, da die IF-Anweisung vor der Schleife steht und BASIC-Programme streng sequentiell abgearbeitet werden. Den häufigen Irrtum der Probanden erklärt sich Pea mit einer Überbewertung der Ähnlichkeit zwischen dem IF-Operator und der englischen Konjunktion ‘if’. In natürlicher Sprache nämlich würden Sätze wie „Wenn Du zur Schule mußt, kann ich dich fahren“ i.a. durchaus auch dann noch gelten, wenn bereits etwas ganz anderes gesagt worden ist.

Mit ‘Intentionalismen’ (*intentionality bugs*) bezeichnet Pea Fehler, die durch die Neigung entstanden sind, bei der Analyse eines Programms zu vermuten, daß ein Computer den Zweck zumindest von Teilen dieses Programms erkennen kann und sich danach verhalten wird. ‘Egozentrismen’ können dann als die Kehrseite dieses Phänomens betrachtet werden, da diese ebenso wie ‘Intentionalismen’ durch die Vermutung verursacht werden, der Computer könnte den Zweck eines Programms oder Programmfragments erkennen. Im Gegensatz zu ‘Intentionalismen’ treten ‘Egozentrismen’ aber beim *Schreiben* von Programmen auf. Beispielsweise begeht ein Programmierer einen ‘Egozentrismus’, wenn er bestimmte Programmzeilen wegläßt im Glauben, der Computer könne seine Absichten aus dem Geschriebenen herleiten.

Sowohl ‘Intentionalismen’ als auch ‘Egozentrismen’ sind für die gegenwärtige Betrachtung eher irrelevant, da diese Fehlerarten, wie es scheint, an keine bestimmten Elemente einer Programmiersprache wie LISP geknüpft werden können. Sie tragen daher nichts zur Klärung der oben gestellten Frage bei, welche spezifischen Schwierigkeiten damit verbunden sind, in LISP programmieren zu lernen (wenn sie auch im weiteren Kontext der Entwicklung von Programmier-ITS durchaus Beachtung verdienen). Anders verhält es sich mit Peas sog. *parallelism bugs*. Wie schon angedeutet, liegt die Parallelität, auf die diese Bezeichnung verweist, in der Ansicht des Programmierschülers, daß zwei oder mehr Anweisungen eines Programms zugleich (und in diesem Sinne ‘parallel’) ‘aktiv’ seien. Betrachtet man aber die tiefere Ursache dieses Phänomens, die auch Pea selbst nennt, nämlich die, daß der Programmierschüler die Namen der involvierten Operatoren (IF, WHILE etc.) allzu wörtlich nimmt, dann erscheint es angebracht, die Bedeutung von ‘Parallelismus’ zunächst auszudehnen auf alle Fälle, für die ebendies zutrifft. Der ‘Parallelismus’ bezöge sich dann auf die Annahme einer übertriebenen Ähnlichkeit – ‘Parallelität’ – zwischen der Semantik eines Operators der Programmiersprache und der namensgebenden Entsprechung in der natürlichen Sprache (zumeist Englisch).

Was nun diese Art von Parallelismen angeht, finden sich in LISP in der Tat einige, wenn auch wenige, Operatoren, die das Auftreten derartiger Fehler begünstigen. LISP ist zwar, wie gesagt, eine eher funktionale Programmiersprache, aber es ist durchaus möglich und üblich, in LISP prozedural zu programmieren in dem Sinne, daß hintereinander geschriebene Ausdrücke nacheinander ausgewertet werden. Daher ist zu vermuten, daß ein in einem solchen Kontext auftretender if- oder when-Ausdruck bei Programmieranfängern zu denselben Schwierigkeiten führt, wie Pea sie im Fall des obigen BASIC-‘Programms’ beobachtet hat. Außerdem gibt es in LISP zumindest einen Operator, der etwas anderes tut, als sein



Name auf den ersten Blick vermuten läßt, nämlich `last`. Hierbei handelt es sich um eine Funktion, die, grob gesprochen, zu einer Liste deren letztes Segment liefert, d.h. der Ausdruck `(last '(a b c))` beispielsweise wertet sich zu `(C)` aus. Als LISP-Anfänger neigt man nach der Erfahrung des Autors aber zu der Annahme, daß `last` vielmehr das letzte *Element* der übergebenen Liste (in dem Beispiel also das Symbol `C`) liefert.

Parallelismen sind zwar üblicherweise mit den Namen bestimmter Operatoren verknüpft, weshalb die Orte möglicher Fehler dieser Art von einem ITS relativ leicht eingegrenzt werden können. Allerdings erfordert eine Entscheidung darüber, ob an einem solchen Ort auch tatsächlich ein Parallelismus vorliegt, fast immer die Kenntnis der Absichten, die der Programmierer an der betreffenden Stelle verfolgt hat. Wie sonst sollte beispielsweise entschieden werden, ob ein Schüler an einer bestimmten Stelle seines Programms statt `(last x)` vielmehr `(car (last x))` gemeint hat? Allein anhand des produzierten Codes ist dieses Problem technisch kaum zu lösen. Die einzigen Möglichkeiten, wie man dem Phänomen der Parallelismen gerecht werden kann, scheinen darin zu liegen, diesen Fehlern entweder in Form von entsprechend ausgewähltem Unterrichtsmaterial vorzubeugen, oder ein ITS zu implementieren, das sich durch eine enge Interaktion mit dem Schüler einen hinreichenden Überblick über dessen jeweilige Absichten verschafft.

### 2.2.3 'Komplizierte' Operatoren

Zusätzlich zu Klammerungsfehlern und Parallelismen sind drittens unter den typischen LISP-Anfängerfehlern solche zu nennen, die schlicht und einfach durch die Kompliziertheit der Anwendung bestimmter Operatoren verursacht werden. Nach der Erfahrung des Autors stellt vor allem der Operator `defmacro`, der in LISP zur Definition von Makros benutzt wird, die meisten LISP-Schüler anfangs vor erhebliche Probleme. Ein anderer häufig zu Schwierigkeiten führender Operator, der oft bei der Definition von Makros eingesetzt wird, ist das sog. *Backquote* (```), ein Read-Makro zum Erstellen von Listenstrukturen. Die kognitive Schwierigkeit bei dem Gebrauch von `defmacro` und *Backquote* ist allerdings nicht durch die Syntax der Operatoren selbst verschuldet, sondern dadurch, wozu diese eingesetzt werden. Die Definition eines Makros erfordert nämlich (wie in anderen Sprachen auch, sofern dort überhaupt Makros vorgesehen sind) die Berücksichtigung von zwei verschiedenen Ebenen der Ausführung: erstens die Konstruktion eines neuen Ausdrucks und zweitens die Auswertung desselben. Entsprechend komplizierter ist es, Makros zu definieren, deren Zweck in der Definition weiterer Makros besteht.

Ein ganz anderes Beispiel für einen 'schwierigen' LISP-Operator ist die Funktion `format`, die zum formatierten Ausgeben eines oder mehrerer LISP-Objekte verwendet wird. Die Schwierigkeit bei dieser Funktion besteht in der relativ esoterischen Syntax der sog. *Format-Strings*, in denen das Format der Ausgabe festgelegt wird. Als abschreckendes Beispiel für die syntaktische Unübersichtlichkeit, die sich beim Gebrauch von `format` nur zu leicht einstellt, mag die folgende Definition einer Funktion `list2string` dienen, die zu einer Liste einen entsprechenden 'deutschsprachigen' *String* liefert (derart, daß etwa der Ausdruck `(list2string '("Ernst" "Karl" "Otto"))` sich zu "Ernst, Karl und Otto" auswertet):

```
(defun list2string (l &optional (none "Keiner"))
  (format nil "~:[~a~;::~*~{~a~~~#[~; und ~;:, ~]~}" 1 none))
```

Obwohl in solchen Fällen das Lesen und Verstehen der Ausdrücke möglicherweise ein größeres Problem darstellt als ihre Konstruktion, ist doch zu vermuten, daß sich

aufgrund der besagten Unübersichtlichkeit auch bei der Konstruktion des öfteren Fehler einschleichen.

Das Erkennen von Fehlern, die durch semantisch oder syntaktisch anspruchsvolle Operatoren hervorgerufen werden, erfordert (wenn es sich nicht nur um einfache Syntaxfehler handelt) i.a. die Kenntnis der Absichten des Programmierers, was, wie schon angemerkt wurde, allein anhand des vorliegenden Codes kaum zu verwirklichen ist. Nur gelegentlich gibt der Code auf automatische Weise auswertbare Hinweise darauf, was intendiert ist.<sup>7</sup> Folglich ist wohl auch diesen Fehlern, wie bereits den Parallelismen, letztendlich nur dadurch zu begegnen, daß entweder durch entsprechend gestalteten Unterricht ihnen vorgebeugt oder durch eine intensive Interaktion mit dem Schüler dessen bei jedem Programmierschritt verfolgte Absichten aufgezeichnet werden.

Den letzteren Weg haben J. R. Anderson und seine Mitarbeiter gewählt und in LISPITS verwirklicht. Mit diesem System wird sich der nun folgende Abschnitt auseinandersetzen.

### 3 Andersons LISP-Tutor

Das wohl bekannteste Tutorsystem für LISP ist das seit Anfang der 80er Jahre von Anderson und anderen entwickelte LISPITS. Dessen Darstellung wird den Hintergrund liefern für die im nächsten Abschnitt folgende Beschreibung von SULLA, dem „System zur Unterstützung beim Lösen von LISP-Aufgaben“, das im Rahmen der vorliegenden Arbeit entwickelt worden ist.

Bei LISPITS handelt es sich ursprünglich um eine Testanwendung von Andersons ACT (*adaptive control of thought*)-Theorie. Im selben Rahmen ist auch ein ITS für die Konstruktion von geometrischen Beweisen (GEOMETRY) entwickelt worden ([ABY85]). Die ACT-Theorie hat sich mit der Zeit weiterentwickelt, von der frühen ACT\* von 1983 ([And83]) bis zur 1993 in [And93] veröffentlichten (und im Vergleich zur Vorgängerin in mancher Hinsicht einfacheren) ACT-R, und so gibt es auch mehr als eine Version des LISP-Tutors. Im folgenden sollen nun die theoretischen Grundlagen dieses Systems kurz dargestellt werden; da die ACT-Theorie schon in ihrer anfänglichen Form ziemlich umfangreich ist, kann hier nur ein sehr skizzenhafter Überblick gegeben werden. Für umfassende Darstellungen sei auf [And83] bzw. [And93] verwiesen.

#### 3.1 Theoretische Grundlagen

Im Mittelpunkt der ACT-Theorie steht die Auffassung, daß prozedurales Wissen in Form von *Produktionen* organisiert sei, also in Form von Regeln wie „Wenn du in diesem Zustand bist, dann tu dies“. Unter dem Zustand eines Systems sind dabei neben dem empfangenen Input und gewissen Parametern vor allem dessen gegenwärtige Ziele zu denken. Ein anderes wesentliches Merkmal der Produktionen neben ihrer Zielbezogenheit ist ihre Modularität ([ACKP95]). Im Gegensatz zu Produktionen wird deklaratives Wissen und der Zugriff darauf als unabhängig von Zielsetzungen angesehen.

---

<sup>7</sup>So dürfte es beispielsweise ohne allzuviel Aufwand möglich sein, aus dem jeweiligen Kontext, in dem in einer Makrodefinition verwendete Variablen definiert sind, auf Fehler bezüglich der Ausführungsebene zu schließen. Wenn nämlich eine Variable in einem anderen Kontext definiert ist als dem, in dem sie ausgewertet wird, kann mit großer Wahrscheinlichkeit davon ausgegangen werden, daß ein Fehler bezüglich der in dem betreffenden Fall verwendeten Ausführungsebene vorliegt.

Die zweite wichtige Annahme betrifft den Mechanismus des Lernens. Vom Standpunkt der ACT-Theorie aus läßt sich menschliches Erlernen von Fähigkeiten als „Wissenskompilierung“ (*knowledge compilation*) beschreiben. Das deklarative Wissen, daß ein Schüler durch Unterricht erwirbt, kann dieser z.B. durch das Befolgen von Instruktionen oder das Herstellen analogischer Bezüge bei der Lösung von Problemen einsetzen. ‘Wissenskompilierung’ wird nun derjenige Prozeß genannt, der dieses auf deklarativem Wissen basierendes Vorgehen umformt in Produktionen, die es ermöglichen, ähnliche Probleme zu einem späteren Zeitpunkt leichter und sicherer zu lösen.

In der ACT\*-Theorie, nicht aber in der späteren ACT-R Theorie, fällt unter Wissenskompilierung neben diesem Prozeß der ‘Prozeduralisierung’ noch eine zweite Art von Vorgang: die „Regelzusammenfassung“ (*rule composition*). Hierunter versteht man einen Prozeß des Zusammenfassens mehrerer Produktionen, die in einem bestimmten Kontext hintereinander anzuwenden sind, zu einer einzelnen. Der Vorteil dieser Zusammenfassung wurde darin gesehen, daß die resultierende einzelne Produktion auf eine effizientere Weise zur Ausführung der entsprechenden Aktionen führt, als dies bei den ursprünglichen Produktionen der Fall gewesen ist.

Als dritte Annahme postuliert Anderson in ACT-R einen Prozeß der „Stärkung“ (*strengthening*) von sowohl deklarativem als auch prozeduralem Wissen. Diese sehr alte und intuitiv einleuchtende Annahme besagt einfach, daß durch Übung, wie etwa das Lösen von Programmieraufgaben, das dabei genutzte Wissen auf Dauer leichter zugänglich wird. Dadurch werden dann künftige Operationen, die dieses Wissen erfordern, leichter und sicherer vonstatten gehen.

Schließlich ist im Zusammenhang von ITS noch eine vierte Annahme zu nennen, nämlich die, daß das Arbeits- oder Kurzzeitgedächtnis eines Menschen beschränkt ist, also nur begrenzt viele neue, noch nicht in das (im Unterschied zum Arbeitsgedächtnis als praktisch unbeschränkt angenommene) Langzeitgedächtnis übergegangene Produktionen gespeichert werden können. Die Bedeutung dieser Annahme für die Eigenschaften von LISPITS werden in den nächsten Unterabschnitten deutlicher werden.

### 3.2 Prinzipien des Designs von LISPITS

In [ABFR87] werden aus der ACT\*-Theorie mehr oder weniger streng acht Prinzipien abgeleitet, die die weitere Entwicklung des LISP-Tutors leiten sollten. In [ACKP95] fassen Anderson und seine Mitarbeiter diese zusammen. Hier eine nochmals geraffte Aufzählung:

1. Repräsentation von Kompetenz als Menge von Produktionen. Die grundlegende Idee hierbei ist, daß die zu erlernenden Fähigkeiten in einem kognitiven Modell repräsentiert werden müssen, damit ein ITS für die Erlernung dieser Eigenschaften sinnvoll spezifiziert und implementiert werden kann. Insbesondere ermöglicht das kognitive Modell, Zielsetzungen im Lernverlauf zu setzen und die Reaktionen des Schülers besser zu interpretieren. Dies allein impliziert allerdings noch keine Forderungen an eine Benutzer-Schnittstelle und die Art der Interaktion mit dem Schüler. Solche Aspekte werden erst durch die folgenden Grundsätze behandelt, die sich auf der Grundlage der ACT\*-Theorie sowie teilweise von empirischen Befunden mit der Frage beschäftigen, wie im Hinblick auf dieses erste Prinzip eine sinnvolle Lehrstrategie beschaffen sein müßte.
2. Unterrichtung des Schülers über die dem Problemlöseprozeß zugrundeliegende Zielstruktur. Eine der stets beibehaltenen Annahmen der ACT-Theorie

besteht in der Auffassung, daß ein Teil jedes Problemlösevorgangs die Dekomposition des betreffenden Problems in Ziele und Unterziele ist. Unter dem Eindruck empirischer Untersuchungen, die laut Anderson *et al.* darauf hindeuten, daß Problemlöseschwierigkeiten teilweise mit einer Unkenntnis der zugrundeliegenden „Zielstrukturen“ zusammenhängen, ist dann der Schluß gezogen worden, daß eine explizite Darstellung solcher Zielstrukturen während des Problemlösens dem Schüler eine wertvolle Hilfe bei der Bearbeitung seiner Aufgaben sein kann.

3. Instruktionen im Kontext des Problemlösens. Dieses Prinzip basiert auf dem empirischen Befund, daß Lernen kontextabhängig ist ([And90], Kap. 7). Hieraus ergibt sich zwar noch nicht, worin genau der Kontext besteht, in dem die Instruktionen am effektivsten zu geben sind. Aus den Erfahrungen, die Anderson und sein Team mit ihrem LISP-Tutor gemacht haben, ziehen sie aber den Schluß, daß der beste Ort für Instruktionen jeweils vor den einzelnen Lernabschnitten des Tutor-Programms liegt. Versuche mit Instruktionen, die exakt zu der Stelle gegeben werden, in der der Schüler sie während des Problemlösens benötigt, haben gezeigt, daß sie dort eher störend wirken. Dennoch sollten die Schüler, während sie eine Aufgabe lösen, auf die Instruktionen zugreifen können, für den Fall, daß sie diese benötigen.
4. Förderung eines abstrakten Verständnisses der beim Problemlösen erworbenen Kenntnisse. Die Motivation für dieses Prinzip liegt in der von Andersons Team gemachten Beobachtung, daß Schüler oftmals nur sehr spezifische Schlußfolgerungen aus ihren Erfahrungen mit den Übungsaufgaben gezogen haben. Dieses Prinzip konkretisiert sich im Rahmen des LISP-Tutors als der Versuch, im Text der Hilfe- und Fehlermeldungen des Systems den gewünschten Abstraktionsgrad zu gebrauchen und damit den Schülern aufzuzeigen.
5. Minimierung der Belastung des Arbeitsgedächtnisses. Laut der ACT-Theorie erfordert das Lernen einer neuen Produktion, daß alle relevanten Informationen gleichzeitig im Arbeitsgedächtnis aktiv sind. Weitere Informationen, die gleichfalls aktiv sind, könnten daher mit dem Lernen der Produktion interferieren. Zudem habe Sweller ([Swe88]) gezeigt, daß eine hohe Belastung des Arbeitsgedächtnisses den Lernvorgang beeinträchtigt. Hieraus ergibt sich, daß Instruktionen möglichst nicht dann präsentiert werden sollten, wenn der Schüler gerade dabei ist, eine Aufgabe zu bearbeiten. Außerdem sollte mit dem Erlernen neuer Produktion erst dann begonnen werden, wenn alle bisher zu lernenden Produktionen bereits ins Langzeitgedächtnis übergegangen sind, also das Arbeitsgedächtnis nicht mehr belasten, und es sollten nur wenige Produktionen gleichzeitig gelernt werden.
6. Unverzügliches Anzeigen von Fehlern. Laut der ACT\*-Theorie besteht der Mechanismus des Erzeugens von Produktionen direkt aus der repräsentierten Erfahrung des Problemlösens (in Andersons Ausdrucksweise *problem-solving trace*). Je länger also im Fall eines Fehlers die entsprechende Meldung und Korrektur auf sich warten läßt, desto mehr Information umfaßt diese Erfahrung, und umso schwieriger wird die Aufgabe, diese zu einer Produktion zu verdichten. Folglich haben Fehlermeldungen in der Regel möglichst prompt zu erfolgen. In der ACT-R-Theorie ist jedoch im Unterschied zu ACT\* nicht mehr die repräsentierte Erfahrung selbst die Grundlage für das Bilden von Produktionen, sondern erst das Produkt des Problemlösens, also im Fall eines Programmier-ITS typischerweise der fertige Code. Obwohl damit die direkte

Basis für eine unmittelbare Anzeige von Fehlermeldungen entfällt, sprechen Ergebnisse einer Untersuchung, die Andersons Team mit Studenten und verschiedenen Varianten von LISPITS durchgeführt haben, durchaus für eine solche Form von Rückmeldung.

7. Anpassung der Granularität von Instruktionen an den Lernverlauf. Dieses Prinzip verdankt sich ursprünglich dem in ACT\* postulierten Regelzusammenfassungsverfahren. Im Grunde genommen besagt es, daß die Problemlösefähigkeit des Schülers vom ITS in zunehmend größeren Einheiten analysiert werden sollte. Mit ACT-R ist dieses Prinzip ebenfalls konsistent, folgt dort aber aus anderen Mechanismen. Es ist nur einmal in der Benutzerschnittstelle eines von Andersons Team entwickelten ITS angewandt worden (s. [ABCL90]) und hat sich dort als nicht sehr erfolgreich erwiesen.
8. Erleichterung von allmählicher Annäherung an die zu lernenden Fertigkeiten. Die Grundlage für dieses Prinzip bildet die Beobachtung, daß Schüler zunächst nicht immer alle für die Lösung eines Problems notwendigen Schritte gleich gut beherrschen. Die Annäherung an das Lernziel wird durch ein ITS dadurch erleichtert, daß Schritte, zu denen ein Schüler noch nicht fähig ist, von dem System ergänzt werden. Nach weiteren Übungsaufgaben, so die Annahme, wird der Schüler dann allmählich imstande sein, auch diese Schritte selbst auszuführen, und das ITS tritt mehr und mehr in den Hintergrund.

Im Hinblick auf die in 1.2 aufgeführten Eigenschaften, über die ein ITS idealerweise verfügen sollte, stellt sich hier die Frage, inwieweit sich die soeben genannten Prinzipien mit jenen Eigenschaften in Beziehung setzen lassen. Diese Frage soll im übernächsten Abschnitt behandelt werden. Zunächst dient es der Anschauung, einen Blick auf die konkreten Eigenschaften von LISPITS zu werfen.

### 3.3 Eigenschaften von LISPITS

In diesem Abschnitt soll nun verdeutlicht werden, auf welche Weise LISPITS seinen Benutzer bei der Bearbeitung von LISP-Aufgaben unterstützt. Während der Student an seiner Aufgabe arbeitet, wird dieser – in Übereinstimmung mit dem Prinzip 6 – dabei ständig überwacht, d.h. nach jedem eingegebenen Symbol wird überprüft, ob er sich erkennbar auf einen richtigen Weg zur Lösung befindet. Solange dies der Fall ist, hält sich das System im Hintergrund und unterstützt den Schüler lediglich durch das Bereitstellen von Syntax-Schablonen. Wenn dem Schüler aber ein Fehler unterläuft, reagiert der Tutor sofort mit einer entsprechenden Fehlermeldung und läßt den Schüler erst dann fortfahren, wenn dieser ein als sinnvoll erkanntes Symbol eingibt. Eine Ausnahme macht der Tutor dann, wenn der Schüler ernste Schwierigkeiten zu haben scheint (etwa, wenn er denselben Fehler mehrmals wiederholt). In diesem Fall gibt das System den nächsten Schritt vor, versehen mit einer entsprechenden Erklärung.

Zusätzlich zu den Fehlermeldungen verfügt LISPITS auch über die Möglichkeit, dem Schüler bei Bedarf Lösungshinweise zu geben. Diese beziehen sich zwar nicht auf spezielle Fehler, fußen aber auf demselben, auf Produktionen basierenden Kompetenzmodell wie die Fehlermeldungen. Anderson und seine Mitarbeiter haben mit unterschiedlichen Strategien hinsichtlich der Lösungshinweise experimentiert, und dabei – mit gewissen Vorbehalten – gefunden, daß diese Hinweise nur auf explizite Anforderung gegeben werden sollten (und nicht etwa schon dann, wenn der Schüler in Schwierigkeiten zu stecken scheint).

Im folgenden wird als Beispiel für die Interaktion zwischen LISPITS und einem Schüler ein Ausschnitt aus einer fiktiven, [ACKP95] entnommenen LISPITS-Sitzung wiedergegeben. Aus Platzgründen wird dabei die zeitliche Sequenz des sich auf dem Bildschirm in verschiedenen Fenstern (Editor, LISP-Interpreter, Tutor-Fenster) abspielenden Geschehens in rein textueller Form dargestellt. Am Anfang steht die Aufgabenstellung, die während der gesamten Bearbeitungsdauer der Aufgabe im Tutor-Fenster zu sehen ist, wenn dieses nicht gerade von einer Fehlermeldung des Tutors beansprucht wird. Darauf folgen neun Zyklen der Schüler-Tutor-Interaktion. Am Anfang jedes Zyklus wird der vom Schüler bisher produzierte Code gezeigt, wobei die in dem jeweiligen Zyklus vorgenommenen Ergänzungen in *kursiver Schreibmaschinenschrift* hervorgehoben sind. Darauf folgen jeweils die entsprechenden Reaktionen des Tutors. Wenn nach den neun Zyklen das Programm fertig ist, wird abschließend in das LISP-Fenster gewechselt, wo der Schüler die gerade definierte Funktion testen kann.

Es sollte noch gesagt werden, daß es sich bei der hier abgedruckten Darstellung um keine vollkommen treue Übersetzung handelt. Der Code des fiktiven Schülerprogramms enthält in [ACKP95] (und auch schon in [And93], in der dieses Interaktionsbeispiel ebenfalls, allerdings ohne Hervorhebungen, vorkommt) einige offensichtliche Klammerungsfehler, und es schien nicht angebracht, diese hier zu wiederholen. Außerdem bestehen bezüglich der Hervorhebungen in [ACKP95] einige Unstimmigkeiten, die ich hier versucht habe, aufzulösen. Dabei erschien es sinnvoll, noch eine zweite Art der Hervorhebung einzuführen, nämlich für diejenigen Stellen im Code, die in dem betreffenden Interaktionszyklus automatisch vom LISPITS-Editor eingefügt wurden. Diese sind hier in **grau** dargestellt.

#### Ein Beispiel für die Interaktion mit LISPITS

Definieren Sie eine Funktion mit dem Namen "CREATE-LIST", die genau ein Argument akzeptiert. Dieses Argument muß eine natürliche Zahl sein. Die Funktion liefert eine Liste, die in aufsteigender Reihenfolge alle natürlichen Zahlen zwischen 1 und dem Wert des Arguments enthält. Beispiel:

(create-list 8) liefert (1 2 3 4 5 6 7 8).

Sie sollten in dieser Funktion abwärts zählen, so daß Sie einfach jede neue Zahl an den Anfang der Ergebnisvariable stellen können.

(Es folgen die Interaktionsschritte zwischen Tutor und Student.)

1.

```
(defun create-list (num)
  (loop))
```

Sie werden in dieser Funktion lokale Variablen benötigen. Bevor Sie also irgendetwas anderes tun, gibt es einen speziellen LISP Operator, den Sie benutzen sollten.

*Ein Beispiel für die Interaktion mit LISPITS (Fortsetzung)*

2.

```
(defun create-list (num)
  (let ( )
    <BODY>))
```

Welches der folgenden Dinge möchten Sie tun?

1) Eine Variable initialisieren zur Speicherung der Liste von Ziffern zwischen 1 und dem Wert des Parameters.

2) Eine Zählervariable initialisieren, um die Anzahl der Schleifenumläufe zu speichern.

3) Den Tutor wählen lassen.

Menüwahl: 2

3.

```
(defun create-list (num)
  (let ((count 1)
        <OTHER-INITIALIZATIONS>)
    <BODY>))
```

In dieser Aufgabe wollen wir mit dem Zählen nicht bei jedem Aufruf der Funktion mit dem selben Wert beginnen, also können wir die Zählervariable nicht mit einer Konstanten wie 1 initialisieren. Wie wird der Anfangswert an die Funktion übergeben?

4.

```
(defun create-list (num)
  (let ((count num)
        (numlist (list num)))
    (loop)))
```

Meinten Sie loop (j oder n)? *j*

5.

```
(defun create-list (num)
  (let ((count num)
        (numlist (list num)))
    (loop
      (cond ((equal count 1) numlist))
      <LOOP-ACTIONS2>)))
```

Sie wollen hier die Variable NUMLIST zurückgeben, aber dies wird nicht einfach dadurch erreicht, daß man sie in das Programm schreibt. Welcher LISP Operator erlaubt Ihnen, etwas aus einem LOOP-Konstrukt zurückzugeben?

6.

```
(defun create-list (num)
  (let ((count num)
        (numlist (list num)))
    (loop
      (cond ((equal count 1) (return numlist)))
      <LOOP-ACTIONS2>)))
```

In numerischen Iterationen ist das erste, was Sie nach dem Test der Abbruchbedingung tun sollten, eine Aktualisierung der Zählervariable. Sie können SETQ hierfür benutzen.

7.

```
(defun create-list (num)
  (let ((count num)
        (numlist (list num)))
    (loop
      (cond ((equal count 1) (return numlist)))
      (setq count (1- count))
      (cons)))
  ...)
```

Sie sind auf dem richtigen Weg, da Sie die Funktion CONS benutzen müssen, aber erinnern Sie sich daran, daß Sie versuchen, den neuen Wert der Zählervariable in die Liste einzufügen. Welche LISP Funktion erlaubt Ihnen, die Ergebnisse dieses Funktionsaufrufs einer lokalen Variable zuzuweisen?

8.

```
(defun create-list (num)
  (let ((count num)
        (numlist (list num)))
    (loop
      (cond ((equal count 1) (return numlist)))
      (setq count (1- count))
      (setq numlist (list))))
```

Wenn Sie Argumente wie 7 und (8) zusammenLISTen, erhalten Sie (7 (8)). Was Sie aber wollen, ist (7 8). Probieren Sie eine andere Funktion, die Dinge zusammenfügt.

9.

```
(defun create-list (num)
  (let ((count num)
        (numlist (list num)))
    (loop
      (cond ((equal count 1) (return numlist)))
      (setq count (1- count))
      (setq numlist (cons count numlist)))))
```



### Ein Beispiel für die Interaktion mit LISPITS (Fortsetzung)

```
---- Sie sind fertig. Geben Sie NEXT ein, um weiterzumachen, ----
---- nachdem Sie die eben definierte Funktion getestet haben. ----

(LISP Fenster)

=> (create-list 10)
(1 2 3 4 5 6 7 8 9 10)

=> next
```

Das zugrundeliegende Verfahren, nach dem LISPITS entscheidet, ob eine Fehlermeldung auszugeben ist (und wenn ja, welche), nennt Anderson *model tracing*. Hierbei handelt es sich um den Versuch, die Programmierschritte des Schülers anhand eines aus mehreren hundert Produktionen bestehenden kognitiven Modells nachzuvollziehen. Neben denjenigen Produktionen, die korrektes Lösungsverhalten repräsentieren, enthält dieses Modell vor allem auch sog. ‘Fehlproduktionen’ (*mal-rules*), die dazu dienen, abweichendes Verhalten zu erklären.<sup>8</sup> In LISPITS geschieht das *model tracing* simultan zur tatsächlichen Interaktion mit dem Schüler. Nach jedem von diesem eingegebenen LISP-Symbol wird vom System überprüft, ob die Eingabe im Rahmen des Modells durch eine korrekte Produktion, eine Fehlproduktion oder gar nicht erklärt werden kann. Entspricht die Eingabe einem im Modell repräsentierten Lösungsweg, läßt LISPITS den Benutzer stillschweigend fortfahren. Im Fall einer Fehlproduktion dagegen reagiert LISPITS mit einer entsprechenden Fehlermeldung; und falls das System die Eingabe schließlich gar nicht in sein kognitives Modell einordnen kann, versucht es, den Schüler mithilfe von Hinweisen wieder auf einen repräsentierbaren Lösungsweg zurückzuführen.

### 3.4 Kritik

Es kann in diesem Abschnitt nicht versucht werden, Andersons LISP-Tutor in allen seinen Aspekten einer gründlichen Kritik zu unterziehen, und sei es nur in den Aspekten, die die Unterstützung beim Lösen von Aufgaben betreffen. Ein solcher Versuch würde schon wegen der zugrundeliegenden ACT-Theorie beinahe ein ganzes Buch füllen. Was aber versucht werden kann und sollte (auch wenn die dabei produzierten Argumente keineswegs zum ersten Mal vorgebracht werden), ist, die Eigenschaften von LISPITS, wie sie in den vorigen Abschnitten beschrieben worden sind, mit den in 1.2 aufgeführten „wünschenswerten Eigenschaften eines Programmier-ITS“ in Beziehung zu setzen. Dies ist nicht zuletzt deshalb wichtig, weil auch das im Rahmen der vorliegenden Arbeit entwickelte System SULLA sich an dem in 1.2 aufgestellten Maßstab wird messen lassen müssen.

Da die wesentlichsten Aspekte von LISPITS in den im vorletzten Abschnitt benannten Design-Prinzipien liegen, erscheint es sinnvoll, diese in den Mittelpunkt der folgenden Beurteilung zu stellen. Das wichtigste dieser Design-Prinzipien ist sicherlich das erste, da sich die restlichen mehr oder weniger stark darauf beziehen. Dieses Prinzip, das die „Repräsentation von Kompetenz als Menge von

<sup>8</sup> Für die auf ACT\* basierende frühe Fassung von LISPITS ([AR85]) nennt Wenger ([Wen87]) eine Zahl von 325 korrekten und sogar 475 Fehl-Produktionen.

Produktionen“ verlangt, entzieht sich zunächst einer direkten Bewertung anhand der Ausführungen in 1.2, da diese sich gewissermaßen nur auf den ‘Phänotyp’ eines Programmier-ITS beziehen. Allenfalls scheint dieses Prinzip in die Nähe der in 1.2.2 ausgeführten Forderung nach „Adaptivität der Lösungshinweise“ gerückt zu werden, nämlich dadurch, daß laut Anderson der Nutzen der Repräsentation von Kompetenz in der von ihm beschriebenen Form u. a. darin liegt, das ITS zu besseren Interpretationen von Schüler-Reaktionen zu befähigen. Betrachtet man aber die Auswirkungen, die dieses Prinzip auf die tatsächliche Anwendbarkeit im Kontext eines (Hoch)schulkurses hat, fällt sofort auf, daß eine Konfigurierbarkeit in Richtung auf neue Lehrinhalte bei LISPITS so gut wie nicht gegeben ist, da dies erfordern würde, zunächst die nötigen Produktionen und Fehlproduktionen zu erarbeiten, dann geeignete Übungsaufgaben zu entwerfen und diese schließlich mit den entsprechenden Fehlermeldungen und Lösungshinweisen zu versehen. Anderson und seine Mitarbeiter haben zwar Entwicklungsumgebungen für ACT-basierte Tutorsysteme geschaffen, aber die Umsetzung einer Anpassung eines auf Produktionen und *model tracing* beruhenden ITS an einen neuen Lehrplan bleibt ein äußerst aufwendiger Vorgang. Dies gilt besonders in bezug auf die zu erstellenden Produktionen (für eine Vorstellung von der benötigten Anzahl s. Fußnote auf S. 25). Und es kann auch keine Rede davon sein, daß durch LISPITS etwa schon alles abgedeckt sei, was es über LISP zu wissen gibt. Auch nach zehn Jahren Entwicklungsarbeit dient das System immer noch zu nicht mehr als einem bloßen Einführungskurs. Man kann also sagen, daß dieses erste Prinzip in einem nicht geringen Maße mit der in 1.2.1 ausgesprochenen Forderung nach Abdeckung bzw. Konfigurierbarkeit des Kompetenzbereichs kollidiert.

Die restlichen Design-Prinzipien beziehen sich in stärkerer Weise auf die Eigenschaften von LISPITS, wie sie der Schüler erfährt. Das vierte Prinzip etwa besagt, daß das abstrakte Verständnis der beim Problemlösen erworbenen Kenntnisse gefördert werden sollte, was ohne weiteres als ein Unterpunkt von 1.2.3 („Verdeutlichung von Zusammenhängen“) aufgefaßt werden kann. Ebenso kann die im fünften Prinzip ausgesprochene Forderung nach einer Minimierung der Belastung des Arbeitsgedächtnisses beim Bearbeiten von Aufgaben dem Punkt 1.2.4 („Unterstützung beim Programmieren und Debuggen“) subsumiert werden.

Problematischer fällt die Beurteilung der Prinzipien Nr. 2 und Nr. 6 aus. Weder das eine noch das andere ist durch das in 1.2 Gesagte bestätigt, sondern im Gegenteil stehen beide in einem gewissen Widerspruch zu 1.2.1 („Kompetenzbereich“), und das sechste Design-Prinzip ist sogar auch vom Standpunkt der Benutzer-Motivation (1.2.5) nicht ganz unproblematisch. Zunächst zu Design-Prinzip Nr. 2 („Unterrichtung des Schülers über die dem Problemlöseprozeß zugrundeliegende Zielstruktur“): Andersons Beobachtung, daß Schüler in einer das Lösen von Problemen involvierenden Domäne oft deswegen Schwierigkeiten zeigen, daß sie die „zugrundeliegende Zielstruktur“ nicht kennen, also, mit anderen Worten, nicht wissen, wie sie vorgehen sollen, mag durchaus zutreffen. Dies bedeutet aber nicht automatisch, daß ihnen das Herausfinden der zugrundeliegenden Zielstrukturen, also ein wichtiger Teil der für das Lösen einer Aufgabe nötigen Planungsprozesse, von einem ITS abgenommen werden sollte. Ähnliches läßt sich in bezug auf Debugging vom sechsten Design-Prinzip („Unverzögliches Anzeigen von Fehlern“) sagen. Daß LISPITS dem Benutzer in frühen (bis ca. 1985) Versionen gar nicht und auch später nur eingeschränkt die Möglichkeit läßt, Fehler zu begehen, um diese dann selbständig zu finden und zu korrigieren, mag sich, ähnlich wie die Anzeige von Programmierzielen, positiv auf den mit der Aufgabenbearbeitung verbundenen Zeitaufwand auswirken ([ACKP95], Abschnitt 3), hat auf der anderen Seite aber unweigerlich den Preis, daß dadurch das Einüben von Debugging-Fähigkeiten weit-

gehend verhindert wird. Sowohl Debugging als auch Programmplanung machen aber, wie in 1.2.1 ausgeführt wird, einen wichtigen Teil jeder Programmierfähigkeit aus, weshalb es nicht wünschenswert ist, wenn ein Programmier-ITS seinen Benutzern keine Gelegenheit bietet, diese Fähigkeiten einzuüben. Was das sechste Design-Prinzip angeht, liegt der eigentliche Gegensatz zu 1.2.1 übrigens nicht darin, daß ihm zufolge die Fehlermeldungen *unmittelbar* erfolgen sollen, sondern vielmehr in der Exaktheit, mit der die von LISPITS ausgegebenen Fehlermeldungen den begangenen Fehler benennen und lokalisieren. Denn Fehler zu erkennen und zu lokalisieren ist ja das, was Debugging ausmacht.

In bezug auf die Kritik daran, daß Andersons LISP-Tutor dem Schüler wenig Gelegenheit gibt, seine Programme selbst zu planen, ist hier allerdings eine relativierende Bemerkung angebracht. Der ACT-R-Theorie zufolge findet Lernen nicht als eine ‘Zusammenfassung’ von Problemlösungsvorgängen statt (wie in ACT\*), sondern auf der Basis der Problemlösungsergebnisse (s. [And93], S.160f.). Durch wessen Problemlösetätigkeit diese Ergebnisse zustande gekommen sind, ist dabei nicht per se relevant. Wenn nun also dem Schüler zu einem gegebenen Problem vorgegeben wird, wie er beim Programmieren einer Lösung vorzugehen hat, kann mit ACT-R argumentiert werden, daß er hierbei durchaus Fähigkeiten zur Programmplanung erwirbt. Gleichwohl spricht aber manches (beispielsweise stärkere persönliche Involviertheit) dafür, daß durch Übung solche Fähigkeiten effektiver erworben werden können.

Ein weiterer Aspekt, in dem das sechste Design-Prinzip in bezug auf die in 1.2 genannten Eigenschaften problematisch erscheint, ist die in 1.2.5 erwähnte Motivation des Schülers, die durch ein ITS auch nach Andersons Auffassung (wie sie sich etwa in [ACKP95] ausdrückt) möglichst aufrechterhalten werden sollte. Hierzu berichtet beispielsweise Wenger ([Wen87], S. 300), daß, auch wenn Andersons LISP-Tutor in der Evaluation gewisse Erfolge aufgewiesen hat, der in diesem System verwirklichte, sehr stark direktive Ansatz bei fortgeschrittenen Schülern dazu neigte, eine gewisse Ungeduld hervorzurufen. Darauf, daß dies, zumindest prinzipiell, keine unausweichliche Folge des von Anderson verfolgten Ansatzes des *model tracing* ist, deutet etwa das von Dion und Lelouche ([DL92]) vorgestellte System TACKLE (*Teaching Algorithmics with a Computer using the Karel Language Environment*) hin. Andererseits berichtet Anderson in [ACKP95] über eine Untersuchung, in welcher der Zeitpunkt der Fehlermeldungen variiert wurde, die aber überraschenderweise keine zuverlässigen Unterschiede bezüglich der Akzeptanz der jeweiligen Systemvarianten zeitigte. Es ist jedoch dabei gefunden worden, daß die teilnehmenden Schüler sofort erfolgende Fehlermeldungen umso stärker bevorzugten, je schwieriger die Aufgaben wurden. Aber ein solches Ergebnis könnte man sich auch mit einer natürlichen Tendenz der Schüler erklären, dasjenige System vorzuziehen, das sie die gestellten Aufgaben in der kürzesten Zeit und mit den geringsten Schwierigkeiten lösen läßt. Ob mit einem solchen System auch mehr zu lernen ist, ist sicher eine andere Frage. Insbesondere hätte eine empirische Studie hier zu berücksichtigen, inwieweit durch die unterschiedlichen Ansätze bezüglich der Lösungshinweise die Debugging- und Programmplanungsfähigkeiten der Schüler beeinflusst werden.

Die verbleibenden drei Design-Prinzipien, Nr. 3, Nr. 7 und Nr. 8, stehen mit den in 1.2 genannten Eigenschaften nur in sehr schwacher oder gar keiner Beziehung: Nr. 3 („Instruktionen im Kontext des Problemlösens“) deshalb nicht, weil Instruktionen, also die für das Lösen einer Aufgabe benötigten Kenntnisse, logischerweise vermittelt werden sollten, *bevor* mit der Bearbeitung der Aufgabe begonnen wird. Insofern können Instruktionen also als wesentlicher Bestandteil eines Systems zur Unterstützung beim Lösen von Übungsaufgaben angesehen werden.

Andererseits kann es natürlich sinnvoll sein, deklaratives Wissen in Form eines Lösungshinweises darzubieten, falls der Benutzer bei der Bearbeitung einer Aufgabe entsprechende Schwierigkeiten zeigt. Insofern also würde das dritte Prinzip unter den Punkt der „Adaptivität der Lösungshinweise“ fallen. Das siebte Prinzip („Anpassung der Granularität von Instruktionen an den Lernverlauf“) verdankt sich lediglich spezifischen Annahmen der ACT-Theorie, bezieht sich außerdem lediglich auf die Gestaltung des Kurrikulums und hat damit keinen nennenswerten Bezug zu den in 1.2 genannten Eigenschaften. Ähnliches gilt für das achte Prinzip („Erleichterung von allmählicher Annäherung an die zu lernenden Fertigkeiten“), weil dieses im Grunde nichts weiter besagt, als daß dem Benutzer bei Lösungsschwierigkeiten Hinweise gegeben werden sollten, die ihm über diese hinweghelfen. Dieses Prinzip ist somit schon als implizite Annahme in dem in 1.2 Gesagtem (insbesondere 1.2.2) enthalten.

Zusammenfassend läßt sich festhalten, daß vom Standpunkt der in 1.2 genannten Eigenschaften der hauptsächliche Kritikpunkt an Andersons LISP-Tutorsystem dessen mangelnde Abdeckung seiner Domäne bzw. seine mangelnde Konfigurierbarkeit ist. Es ist dabei bezeichnend, daß diese Schwäche nicht etwa daherrührt, daß dieses Tutorsystem über zu wenige Fähigkeiten verfügt. Sondern vielmehr kann man sagen, daß die Entwickler von LISPITS ihrem System, zumindest vom Standpunkt der genannten Anforderungen an ein Programmier-ITS, eher ein wenig zu viele Fähigkeiten mitgegeben haben. LISPITS auf ein explizites kognitives Modell der Schüler-Kompetenz zu gründen, war notwendig, um die ACT-Theorie einem gründlichen Praxistest zu unterziehen, aber vom Standpunkt der genannten Anforderungen an Programmier-ITS fragt es sich, ob die Vorteile dieses Ansatzes den Aufwand in Entwicklung und Konfigurierung wert sind. Und es fragt sich auch, ob nicht auf bestimmte Fähigkeiten von LISPITS, namentlich die explizite Anzeige von Vorgehensweisen für die Lösung der jeweiligen Aufgaben oder die unmittelbare Rückmeldung bei Programmierfehlern, generell besser verzichtet werden sollte. Empirische Untersuchungen zur Klärung dieser Fragen wären wünschenswert.

## 4 SULLA

Der folgende Abschnitt soll das im Rahmen dieser Arbeit entwickelte System SULLA vorstellen und seine Eigenheiten erläutern. Nach der Darstellung eines Systems wie LISPITS, in welches mehr als eine Dekade Entwicklungsarbeit geflossen ist, muß zunächst die vollkommen andere Größenordnung deutlich gemacht werden, in die SULLA demgegenüber fällt. Die Entwicklungszeit für SULLA betrug ungefähr 6 Wochen, der Code umfaßt etwa 4500 Zeilen (davon ca. 200 in LISP, der Rest in PROLOG). Beides nimmt sich gegen LISPITS in der Tat ausgesprochen bescheiden aus. Aber nach der im letzten Abschnitt geäußerten Kritik an Andersons Tutorsystem könnte gerade hierin ein Verdienst liegen. Tatsächlich ist SULLA vorwiegend im Hinblick auf die Verwendung in einem LISP-Kurs an einer (deutschen) Universität konzipiert worden und darauf ausgelegt, möglichst einfach für einen beliebig großen Teilbereich der Domäne des LISP-Programmierens konfigurierbar zu sein. Zudem fehlen in SULLA Elemente, die in LISPITS (oder auch anderen, hier nicht weiter behandelten LISP-ITS, wie etwa ELM-ART II (s. [WS97]) dem Schüler die Gelegenheit zum eigenständigen Planen und Debuggen seiner Aufgabenlösungen entziehen. Für den Fall, daß der Schüler bei der Lösung einer Aufgabe in größere Schwierigkeiten gerät, kann er zwar Lösungshinweise anfordern. Diese sind aber, um den Schüler zu größtmöglicher Selbständigkeit anzuhalten, jeweils mit einem Punktabzug in der Aufgabenbewertung verbunden. Die folgenden Abschnitte wer-

den auf die einzelnen Merkmale von SULLA näher eingehen. Dabei wird ggf. zugleich auf die in Abschnitt 1.2 ausgeführten Anforderungen eingegangen werden.

## 4.1 Eigenschaften von SULLA

### 4.1.1 Grundsätzliche Benutzungsweise

Im Gegensatz zu LISPITS, wo der gesamte Prozeß des AufgabenlöSENS sich im Tutorssystem selbst abspielt, muß (oder kann) der Benutzer bei SULLA seine Aufgaben in einem beliebigen anderen Editor bearbeiten. Erst das mehr oder weniger fertige Programm wird dann in SULLA eingelesen. Dort kann der Benutzer, wenn das Programm ein konsistentes Layout aufweist und keine Syntaxfehler, aber gültige Aufgabenmarkierungen enthält, eine solche Aufgabe auswählen. Diese wird dann von SULLA anhand der Übereinstimmung ihrer Ergebnisse mit denen der Musterlösung sowie anhand ihrer Effizienz bewertet. Der Benutzer kann daraufhin zu einer anderen Aufgabe (oder einem anderen Programm) übergehen oder, wenn er seine Lösung zuerst noch verbessern möchte, dies tun, um sie dann erneut einlesen und bewerten zu lassen. Falls er bei der Lösung Hilfe benötigt, kann er diese, wie gesagt, gegen Punktabzug vom System anfordern. Darüberhinaus kann er sich eine Auflistung der bisher erzielten Bewertungen seiner Lösungen ansehen, die auch Auskunft über die noch nicht bearbeiteten Aufgaben (und die dort erreichbaren Punkte) gibt.

### 4.1.2 Kompetenzbereich und Konfigurierung

Wie schon gesagt, ist bei der Konzipierung von SULLA in Übereinstimmung mit 1.2.1 viel Wert darauf gelegt worden, daß das System für die Vermittlung eines möglichst großen Teils der mit dem Programmieren in LISP verbundenen Fähigkeiten eingesetzt werden kann. Dies beginnt damit, daß die Prozedur, die SULLA für das Einlesen von LISP-Programmen verwendet, die *gesamte* Standard-Syntax von LISP, also einschließlich etwa der mit # eingeleiteten *Read-Makros*, sinnvoll verarbeiten kann. Bezüglich der in der Musterlösung oder vom Schüler verwendeten LISP-Konstrukte bestehen in SULLA also keinerlei Einschränkungen.

Die Konfigurierung von SULLA für einen bestimmten Bereich der Domäne ‘LISP-Programmierung’ erfolgt einfach durch das Bereitstellen von Musterlösungsdateien. Die Aufgabentexte selbst brauchen in diesen nicht enthalten zu sein, sie sind außerhalb des Systems zu verwalten. Die gesamte Bewertung von Schülerlösungen und auch die Generierung von Lösungshinweisen vollzieht sich einzig anhand der Musterlösungen. In Tabelle 2 ist ein Beispiel für eine (sehr einfache) Musterlösungsdatei mit nur einer einzigen Aufgabe abgebildet. Zum besseren Verständnis ist der Aufgabentext in ihr mit enthalten.

Neben dem Aufgabentext, der Definition einer Hilfsfunktion, etwas erläuterndem Kommentar und der Musterlösung selbst finden sich in dieser Datei fünf verschiedene Arten von sog. *Direktiven*, die SULLA mitteilen, wie mit der Datei umzugehen ist. Hierbei handelt es sich stets um mit ; auskommentierte Zeilen, die als erstes Zeichen nach dem Semikolon einen Rückstrich, gefolgt von einem oder mehreren weiteren Zeichen, enthalten:<sup>9</sup>

- Die A-Direktive ist eine Aufgabenmarkierung. Der gesamte nachfolgende Teil der Datei bis zur nächsten Aufgabenmarkierung wird als zur selben Aufgabe gehörig aufgefaßt. Als Aufgaben-Bezeichnung dient SULLA das jeweils nach dem A stehende ‘Wort’, in diesem Fall ist das die Zeichenkette "1.1".

---

<sup>9</sup>Für weitere Details siehe Anhang B.

```

;;; Musterlösungen zu Blatt 1 (Beispiel)

;;; Hilfsfunktionen

(defun car-eq (l1 l2)
  (eq (car l1) (car l2)))

;;; Aufgaben

;\A 1.1
;\P 3

; Definieren Sie eine Funktion 'second-element', die zu einer
; mindestens zweielementigen Liste deren zweites Element liefert.
; Wenn die Liste weniger als zwei Elemente enthält, soll nil
; zurückgegeben werden.
; Beispiele:
; (second-element '(a b c)) liefert B,
; (second-element nil) liefert NIL.

;\F second-element
;\E car-eq
;\R 50

;\test '()
;\test '('(a))
;\test '('(b (c)))
;\test '('(d (e) f))

(defun second-element (x)
  (car (cdr x)))

```

Abbildung 2: Beispiel für eine Musterlösungsdatei

- Die P-Direktive gibt an, wieviel Punkte für die Aufgabe maximal zu vergeben sind; im vorliegenden Beispiel also 3.
- Mit der F-Direktive wird dem System mitgeteilt, welcher Operator zu testen ist.
- Die E-Direktive erfordert von allen Direktiven die meiste Sorgfalt. In ihr wird angegeben, wie die Listen der Rückgabewerte von Schüler- und Musterlösung miteinander zu vergleichen sind. Für jede zu testende Parameterbelegung gibt es sowohl für die Muster- als auch für die Schülerlösung je eine Liste der Rückgabewerte. Diese Listen sind notwendig, weil Funktionen in LISP mehr als einen (oder auch gar keinen) Rückgabewert aufweisen können. In dem Beispiel der Funktion `second-element` geht es darum, daß die Schülerlösung als (ersten) Rückgabewert exakt dasselbe Objekt liefert wie die Musterlösung, d.h. die jeweils ersten Elemente der beiden Listen von Rückgabewerten müssen `eq` sein. Für diese Art von Vergleich gibt es keinen 'eingebauten' Operator, folglich muß dieser in der Musterlösungsdatei eigens definiert werden. Damit aber die Definition von `car-eq` nicht als Teil der Musterlösung interpretiert wird, ist sie an den Anfang der Datei, in den Bereich vor der ersten Aufgabenmarkierung, zu stellen.
- Die `test`-Direktiven teilen dem System mit, welche Parameterlisten für das Testen der Schülerlösung verwendet werden sollen. Nach jedem `test` erwartet SULLA einen LISP-Ausdruck, der sich zu einer solchen Parameterliste auswertet. Die Parameter haben in dieser Liste genauso zu erscheinen, wie sie auch in einem normalen Aufruf an den zu testenden Operator übergeben würden. Wenn also der Funktionsaufruf `(second-element '(a b))` getestet werden soll, muß der Ausdruck in der entsprechenden `test`-Direktive den Wert `('(a b))` haben.
- Schließlich kann mit der optionalen R-Direktive eingestellt werden, wie oft die Schülerfunktion mit den angegebenen Parameterlisten getestet werden soll. Dies dient lediglich einer zuverlässigeren Bestimmung des Aufwands an Zeit und Speicher, den die Schüler- im Vergleich zur Musterlösung beansprucht. Wenn keine R-Direktive vorhanden ist, werden die Parameterlisten jeweils nur einmal verwendet.

Das hier im Zusammenhang mit der E-Direktive zum vorliegenden Beispiel Gesagte setzt voraus, daß die für die Schülerlösung verwendeten Parameter nicht bloß Kopien derjenigen Objekte sind, auf welche die Funktion der Musterlösung angewendet wird, sondern daß diese vielmehr exakt identisch sind (im Sinne von `eq`). Zugleich liefert dieses Beispiel die Rechtfertigung dafür, daß dies so ist, denn wären die Parameter für Muster- und Schülerlösung bloß Kopien voneinander, könnte anhand der Resultatlisten allein nicht überprüft werden, ob die Funktion der Schülerlösung auch tatsächlich das zweite Element der an sie als Argument übergebenen Liste zurückgibt.

Zum Abschluß dieses Punkts sei noch auf einige Einschränkungen hingewiesen, denen die Konfigurierbarkeit von SULLA unterliegt, und die leider nicht ohne weiteres aufgehoben werden können. So können Aufgaben, die nicht als Definition einer Funktion oder eines Makros definierbar sind, wie das Zeichnen von Diagrammen oder die Beschreibung von Arbeitsweisen einer gegebenen Funktion, in SULLA schlichtweg nicht gestellt werden, auch wenn dies natürlich wünschenswert wäre.

Des weiteren sind Aufgabenstellungen, in denen die zu definierenden Funktionen ihre Argumente verändern, zwar möglich, erfordern aber u. U. einen erheb-

lichen Aufwand bei der Definition der Vergleichsfunktion. Dies liegt daran, daß SULLA für die Testaufrufe von Schüler- und Musterlösung aus den oben genannten Gründen *dieselben* Parameterlisten verwendet, wobei die Aufrufe der Musterlösung zuerst erfolgen. Die an die Schülerlösung übergebenen Parameter sind also durch die Musterlösung bereits verändert. Ein Beispiel, bei dem dies noch ein einfaches Problem darstellt, ist die Aufgabe, eine Funktion zu definieren, die in der an sie übergebenen Liste (*à la nreverse*) die Reihenfolge der Elemente umkehrt und das Ergebnis zurückgibt. Hier bestünde das Problem einfach darin, daß die Vergleichsfunktion lediglich mit Hilfe der Funktion `equal` das Ergebnis der Schülerlösung mit einer Kopie des an die Musterlösungsfunktion übergebenen Parameters (die diese allerdings selbst zu erzeugen und als zusätzlichen Rückgabewert bereitzustellen hätte) vergleichen müßte. In anderen Fällen jedoch kann dieser Vergleich erheblich komplizierter ausfallen. Nachzuprüfen, ob die Schülerlösung auch wirklich ihr Argument verändert, ist dagegen denkbar einfach: die Rückgabewerte von Muster- und Schülerlösung müssen (im Sinne von `eq`) miteinander identisch sein.

#### 4.1.3 Überprüfung von Klammerung und Layout

Einer der wichtigsten Bestandteile des SULLA-Systems und sicherlich der komplexeste von allen ist das Modul zur Überprüfung von Klammerung und Layout. Die vorrangige Aufgabe dieses Moduls ist es, Klammerungsfehler in einem Schülerprogramm anhand von dessen Layout, d.h. der Einrückungsweiten der einzelnen Programmzeilen, möglichst genau einzugrenzen. Zusätzlich ist es fähig, mithilfe gewisser Heuristiken wahrscheinliche Einrückungsfehler aufzuzueigen. Für eine detailliertere Beschreibung der Funktionsweise dieses Moduls sei auf Anhang A verwiesen.

Der Grund, warum SULLA dem Schüler überhaupt die Suche nach Klammerungsfehlern, so gut es geht, abnehmen sollte, liegt vor allem darin, daß es laut Abschnitt 1.2.4 das Lernen sehr erleichtern kann, wenn einfache Routine-, Tätigkeiten vom Tutorsystem übernommen werden. Da die Suche nach Klammerungsfehlern in LISP-Programmen in der Tat eher eine Geduldsübung als eine wichtige und wertvolle Fähigkeit ist, und darüberhinaus Klammerungsfehler gerade bei LISP-Anfängern zu den häufigsten und frustrierendsten Programmierfehlern zählen, liegt eine (Teil-)Automatisierung also nahe. Weiterhin läßt sich der Algorithmus, der in SULLA zur Überprüfung der Klammerung zum Einsatz kommt, unabhängig von jeglichen Musterlösungsdateien auf jedes beliebige LISP-Programm anwenden. Bedenken, die aufgrund von Abschnitt 1.2.1 vor einer zu vermeidenden Einschränkung des Kompetenzbereichs warnen, wären damit gegenstandslos.

Tabelle 3 zeigt in ihrem ersten Teil ein kurzes Schülerprogramm mit Klammerungsfehlern und inkonsistentem Layout. Im zweiten Teil ist der Vorschlag von SULLA zu sehen, wie die gefundenen Fehler und Inkonsistenzen beseitigt werden können.

#### 4.1.4 Evaluierung von Aufgabenlösungen

Aufgabenlösungen werden in SULLA nach folgenden Regeln bewertet: Wenn bei einem Testaufruf der Schülerlösung ein Fehler auftritt, wird sie mit 0 Punkten bewertet; tritt kein Fehler auf, wird die zu erreichende Punktzahl mit dem Anteil der Testaufrufe multipliziert, auf denen sie korrekte Resultate geliefert hat. Zusätzlich wird berücksichtigt, wie effizient die Schülerlösung im Vergleich mit der Musterlösung arbeitet. Je nachdem, ob sie mehr oder weniger Speicher und Zeit verbraucht als die Musterlösung, wird die verbleibende Punktzahl um einen gewissen Faktor verringert oder (sofern die Lösung auch die richtigen Resultate liefert) erhöht.



```
(defun flat (x)
  (cond (null x) nil)
        ((consp (car x))
         (append (flat (car x)) (flat (cdr x)))))
        (T (cons (car x) (flat (cdr x))))))
```

Lade Datei home/l3...

Layout und Klammerung sind nicht konsistent.  
 Z. 2 - 3: Layout irreleitend und/oder Klammerungsfehler:  
 Vermutlich zu viele ')' oder zu wenige '(' (1mal) in: Z. 2  
 Z. 4 sollte 1 Spalte weiter eingerückt sein.

Abbildung 3: Beispiel für Überprüfung von Klammerung und Layout

Wenn der Schüler eine korrekte Lösung abgibt, die zudem effizienter arbeitet als die Musterlösung, kann er also mehr Punkte dafür erhalten, als die P-Direktive angibt.

Um dem Schüler im Fall von Fehlern oder inkorrekten Rückgabewerten die Korrektur seiner Lösung zu erleichtern, werden bei einem Fehler die bei dem entsprechenden Funktionsaufruf verwendete Parameterliste sowie die Fehlermeldung des LISP-Interpreters angezeigt und bei inkorrekten Rückgabewerten die ersten zehn (gemäß Voreinstellung) Parameterlisten, auf denen die Schülerlösung einen falschen Wert geliefert hat.

Die folgende Tabelle gibt zur Veranschaulichung ein Beispiel für die Interaktion mit SULLA im Zusammenhang mit der Bewertung einer Aufgabenlösung.

Tabelle 2: Bewertung von Aufgabenlösungen

<p>Relevanter Ausschnitt aus der Musterlösung</p> <pre>;\A 2.2 ;\P 20 ; Definieren Sie eine Funktion 'rev', die zu einer Liste l1 ; eine neue Liste liefert, in der alle Elemente von l1 in ; umgekehrter Reihenfolge enthalten sind. ; Beispiel: ; (rev '(a b a c)) liefert (C A B A).</pre> <pre>;\F rev ;\E equal ;\R 50 ;\test '(nil) ;\test '('(1 2 (3 4) ((5 6) 7) 8 9)) ;\test '('(r s ,@(make-list 100)))</pre> <pre>(defun rev (l1 &amp;optional r)   (cond ((null l1) r)         (T (rev (cdr l1) (cons (car l1) r)))))</pre>
<p>Funktionsdefinition des Schülers</p> <pre>(defun rev (l1)</pre>

Tabelle 2: Bewertung von Aufgabenlösungen (Fortsetzung)

<pre>(cond ((null l1) nil)       (T (append (rev (cdr l1)) (list (car l1)))) ))</pre>
<p>Interaktion mit SULLA</p> <p>Welche Aufgabe soll untersucht werden?  Die folgenden Nummern stehen zur Verfügung:  2.1  2.2  2.3</p> <p>(a : andere Datei laden; x : zurück zum Anfang)   : 2.2</p> <p>Glückwunsch, Ihre Lösung scheint korrekt zu sein!  Schauen wir uns nun die Effizienzdaten an:  Ihre Lösung benötigt 1.21436mal soviel Zeit wie die Musterlösung.  Das würde Sie 4.23456 Prozent Ihrer verbliebenen Punktzahl kosten.  An Speicher verbrauchen Sie 2.92376mal soviel.  Das würde Sie 22.6409 Prozent Ihrer verbliebenen Punktzahl kosten.  (Vorläufiges) Ergebnis für diese Aufgabe: 14.8167 von 20 Punkten.</p> <p>Optionen:  e : erneut laden; h : Hinweis; a : andere Aufgabe; x : zurück zum Anfang   :</p>
<p>Funktionsdefinition des Schülers</p> <pre>(defun rev (l1 &amp;optional z)   (cond ((null l1) z)         (T (rev (cdr z) (cons (car l1) z)))) ))</pre>
<p>Interaktion mit SULLA</p> <p> : e  Lade Datei 12...</p> <p>Ihre Lösung ist noch nicht korrekt. Für folgende Parameterlisten liefert sie falsche Resultate:</p> <pre>('(#:R #:S NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL   NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL   NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL   NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL   NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL   NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL ) ) '(1 2 (3 4) ((5 6) 7) 8 9))</pre> <p>Damit würden Sie 66.7 Prozent Ihrer Punktzahl für diese Aufgabe verlieren.</p>

Tabelle 2: Bewertung von Aufgabenlösungen (Fortsetzung)

<p>Schauen wir uns nun die Effizienzdaten an:  Ihre Lösung benötigt 0.904729mal soviel Zeit wie die Musterlösung.  An Speicher verbrauchen Sie 0.961439mal soviel.  (Vorläufiges) Ergebnis für diese Aufgabe: 6.66 von 20 Punkten.</p> <p>Optionen:  e : erneut laden; h : Hinweis; a : andere Aufgabe; x : zurück zum Anfang    :</p>
<p>Funktionsdefinition des Schülers</p> <pre>(defun rev (l1 &amp;optional z)   (cond ((null l1) z)         (T (rev (cdr l1) (cons (car l1) z)))   ))</pre>
<p>Interaktion mit SULLA</p> <pre> : e Lade Datei 12...</pre> <p>Glückwunsch, Ihre Lösung scheint korrekt zu sein!  Schauen wir uns nun die Effizienzdaten an:  Ihre Lösung benötigt genau soviel Zeit wie die Musterlösung.  An Speicher verbrauchen Sie 1.00093mal soviel.  Das würde Sie 0.0201792 Prozent Ihrer verbliebenen Punktzahl  kosten.  (Vorläufiges) Ergebnis für diese Aufgabe: 19.996 von 20 Punkten.</p>

#### 4.1.5 Lösungshinweise

Falls der Schüler bei der Bearbeitung einer Aufgabe nicht weiterweiß, kann er von SULLA gegen Punktabzug einen Lösungshinweis anfordern. Gegenwärtig sind zwei Arten von Lösungshinweisen implementiert:

- Anzeige der in der Musterlösung verwendeten Funktionen und Makros. Hierfür werden voreingestellte 8% der jeweils verbliebenen Punkte abgezogen.
- Schrittweise Aufdeckung der Strukturebenen der Musterlösung. Der Punktabzug hierfür ist gleich dem Anteil der aufgedeckten Symbol-Repräsentationen an deren Anzahl in der gesamten Musterlösung.

Zusätzlich kann der Schüler jederzeit einen bereits erhaltenen Hinweis wieder verlangen, ohne dafür Punkte zu verlieren. Die folgende Tabelle zeigt ein Interaktionsbeispiel, das die verschiedenen Arten von Lösungshinweisen veranschaulicht.

Tabelle 3: Lösungshinweise in SULLA

<p>Optionen:  e : erneut laden; h : Hinweis; a : andere Aufgabe; x : zurück zum Anfang    : h  Folgende Lösungshinweise sind verfügbar:  1: Aufdecken der Musterlösung bis zur 0ten Ebene  (Punktabzug: 0%)</p>
---

Tabelle 3: Lösungshinweise in SULLA (Fortsetzung)

<p>2: Auflisten von in der Musterlösung verwendeten Operatoren (Punktabzug: 8%)  : 0</p> <p>Geben Sie bitte eine ganze Zahl zwischen 0 und 3 an: 1</p> <p>Die Musterlösung bis zur 0ten Ebene lautet:</p> <p>(     )</p> <p>Optionen: e : erneut laden; h : Hinweis; a : andere Aufgabe; x : zurück zum Anfang  : h</p> <p>Folgende Lösungshinweise sind verfügbar:</p> <p>1: Aufdecken der Musterlösung bis zur 1ten Ebene (Punktabzug: 33.3333%)</p> <p>2: Erneutes Aufdecken der Musterlösung bis zur 0ten Ebene (Punktabzug: 0%)</p> <p>3: Auflisten von in der Musterlösung verwendeten Operatoren  : 1</p> <p>Die Musterlösung bis zur 1ten Ebene lautet:</p> <p>(defun second-element ( )   (        ))</p> <p>Optionen: e : erneut laden; h : Hinweis; a : andere Aufgabe; x : zurück zum Anfang  : h</p> <p>Folgende Lösungshinweise sind verfügbar:</p> <p>1: Aufdecken der Musterlösung bis zur 2ten Ebene (Punktabzug: 50%)</p> <p>2: Erneutes Aufdecken der Musterlösung bis zur 1ten Ebene (Punktabzug: 0%)</p> <p>3: Auflisten von in der Musterlösung verwendeten Operatoren (Punktabzug: 8%)  : 1</p> <p>Die Musterlösung bis zur 2ten Ebene lautet:</p> <p>(defun second-element (x)   (car (    )))</p> <p>Optionen: e : erneut laden; h : Hinweis; a : andere Aufgabe; x : zurück zum Anfang  : h</p> <p>Folgende Lösungshinweise sind verfügbar:</p> <p>1: Aufdecken der Musterlösung bis zur 3ten Ebene (Punktabzug: 100%)</p> <p>2: Erneutes Aufdecken der Musterlösung bis zur 2ten Ebene (Punktabzug: 0%)</p> <p>3: Auflisten von in der Musterlösung verwendeten Operatoren (Punktabzug: 8%)</p>	
--	--

Tabelle 3: Lösungshinweise in SULLA (Fortsetzung)

: 3
In der Musterlösung verwendete Operatoren:
DEFUN
CAR
CDR
Optionen:
e : erneut laden; h : Hinweis; a : andere Aufgabe; x : zurück zum Anfang
:

Die ‘Adaptivität’ dieser Lösungshinweise (im Sinne von 1.2.2) ist natürlich nicht auf die ‘Intelligenz’ SULLAs zurückzuführen: Es ist der Schüler selbst, der bestimmt, welche Art von Lösungshinweis seinen jeweiligen Schwierigkeiten wahrscheinlich am angemessensten ist. Aber dieser Ansatz scheint nicht der schlechteste zu sein, insofern, als der Schüler über die Art seiner Schwierigkeiten für gewöhnlich selbst am besten informiert ist oder zumindest sein sollte. Und wenn er es nicht ist, kann die Tatsache, daß die einzelnen Hinweise mit einem bestimmten, mehr oder weniger hohen, Punktabzug verbunden sind, ihn womöglich dazu bringen, sich ein klareres Bild über die Art seiner Schwierigkeiten zu machen. Dies allein kann schon ein Schritt zur Lösung seines Problems sein. Dennoch ist nicht abzustreiten, daß das Hinweissystem in SULLA in seiner jetzigen Form ausgesprochen simpel ist und verschiedene Erweiterungsmöglichkeiten sich anbieten. Von solchen ist im nun folgenden Abschnitt die Rede.

## 4.2 Ausblick: Möglichkeiten der Weiterentwicklung

Obwohl SULLA bereits insoweit ein vollständiges Tutorsystem ist, daß es – in einem geeigneten Rahmen und mit einer ausreichenden Anzahl Musterlösungsdateien versehen – zur Unterstützung bei der Bearbeitung von Übungsaufgaben eines LISP-Kurses eingesetzt werden kann, besteht kein Zweifel darüber, daß das System in seiner gegenwärtigen Form für einen ernsthaften Einsatz in mancher Hinsicht einer Erweiterung bedarf. So wäre es etwa sehr wünschenswert, wenn SULLA für den Zugriff über eine Internetverbindung eingerichtet werden könnte, damit das System nur einmal auf einem Server installiert zu werden braucht, um dann von vielen verschiedenen Orten aus zugänglich zu sein.

Aber auch hinsichtlich der in 1.2 aufgeführten wünschenswerten (Kern-)Eigenschaften von Programmier-ITS kann nicht behauptet werden, daß SULLA alle Anforderungen bereits zufriedenstellend erfüllt. Die größten Stärken von SULLA scheinen bisher in der Unterstützung eines verhältnismäßig großen Kompetenzbereichs sowie der Hilfeleistung beim Programmieren und Debuggen (zumindest, was das Finden von Klammerungsfehlern betrifft) zu liegen. Die „Adaptivität der Lösungshinweise“ allerdings liegt, wie bereits angesprochen, noch weitgehend in der Hand des Benutzers selbst. Zunächst wäre hier dem Übungsleiter die Möglichkeit zu geben, für bestimmte Aufgaben Lösungshinweise bestimmter Art zu verbieten oder die damit verbundenen Punktabzüge zu modifizieren. Interessant wäre es auch, dem Übungsleiter die Möglichkeit zu geben, eigene Lösungshinweise (einschließlich Punktabzug) zu definieren, die dann von den Schülern zusätzlich zu den automatisch erstellten Hinweisen abgerufen werden können. Außerdem könnte man darüber nachdenken, wie aus dem produzierten Code des Benutzers oder auch in expliziter Interaktion mit diesem ein klareres Bild über die Art seiner Schwierigkeiten bei der Bearbeitung einer gegebenen Aufgabe gewonnen werden

könnte. Eine Möglichkeit bestünde sicherlich in einem irgendwie gearteten *model tracing*-Verfahren. Jedoch sollte hier darauf geachtet werden, daß durch das Beheben des einen Mangels von SULLA die bestehenden Stärken nicht unterminiert werden. SULLA ist gegenwärtig zugegebenermaßen ein relativ minimalistisches ITS, das aber sehr einfach für verschiedene Lehrpläne konfiguriert werden kann. Durch die Implementation von *model tracing* oder anderer sehr stark von formalisiertem Domänenwissen abhängiger Verfahren würde diese einfache Konfigurierbarkeit verloren gehen.

Zudem hat der gegenwärtig noch sehr geringe Entwicklungsgrad den Vorteil, daß zunächst empirisch untersucht werden kann, wieviel durch ein derartig einfaches System in bezug auf die angestrebte Unterstützung beim Erlernen von LISP zu erreichen ist. Ausgehend von solchen Studien können dann in sehr systematischer Weise verschiedene Erweiterungen des Systems durchgeführt und evaluiert werden. Daher wäre es sicher voreilig, das bestehende System ohne weiteres zugunsten einer Erweiterung in irgendeiner Richtung aufzugeben.

Betrachten wir aber dennoch einige weitere sich anbietende Entwicklungsmöglichkeiten. Die hier bisher noch nicht angesprochenen Anforderungsbereiche von Abschnitt 1.2 betreffen erstens die „Verdeutlichung von Zusammenhängen“ und zweitens die „Benutzerführung und Benutzeroberfläche“. In beiderlei Hinsicht wären Verbesserungen denkbar; was den ersten Punkt angeht, ist in SULLA bisher noch nichts implementiert, das zum Herstellen von Querbezügen zwischen Übungsaufgaben oder gar zu irgendwelchen Lektionen dienen könnte. Ein solches Verfahren müßte auch, wenn der Konfigurationsaufwand nicht erheblich erhöht werden soll, weitestgehend allein auf den zur Verfügung gestellten Musterlösungen basieren und wäre damit entsprechend schwierig zu implementieren. In bezug auf Benutzerführung und -Oberfläche dagegen würde es sich anbieten, eine graphische Benutzeroberfläche zur Verfügung zu stellen. In diese könnte auch ein Editor mit Syntaxunterstützung eingebaut werden, was allerdings sehr leicht Akzeptanzprobleme mit sich bringen könnte, wenn manche Benutzer des Systems bereits jeweils ein bestimmtes Editor-Programm besonders gern verwenden. Hiervon abgesehen wäre jedoch eine graphische Schnittstelle prinzipiell ein sinnvolles Mittel, um die Bedienbarkeit von SULLA sowie seine Akzeptanz bei seinen Benutzern (und damit deren Motivation) zu erhöhen.

Im Hinblick schließlich auf bereits bestehende Programmier-ITS wie z.B. das bereits kurz erwähnte PLOT ([BGH<sup>+</sup>99]), deren Diagnose-Module als wichtigstes Hilfsmittel über Algorithmen zum strukturellen Vergleich von Benutzer- und Musterlösung verfügen, könnte des weiteren argumentiert werden, daß auch in SULLA entsprechende VergleichsprozEDUREN eingebaut werden sollten. Denkbar wäre beispielsweise, daß das System die Lösung des Benutzers mit den für die betreffende Aufgabe angelegten Musterlösungen abgleicht und dann, wenn eine hinreichend ähnliche Musterlösung gefunden worden ist, den Benutzer auf strukturelle Unterschiede hinweist. Solche strukturellen Unterschiede können im konkreten Fall etwa auf falsch geschriebene Symbolnamen oder vertauschte Variablen in Funktionsaufrufen zurückgeführt werden und somit zur Lokalisierung und Behebung durchaus häufig vorkommender Fehler beitragen. So attraktiv die Aussicht auf eine solche Erweiterung der Funktionalität auch erscheint, sollte aber bedacht werden, daß die Voraussetzung für das Funktionieren eines solchen Vergleichs in einer hinreichend sicheren Bestimmung korrespondierender Teile von Benutzer- und Musterlösung besteht. Es muß nicht eigens betont werden, daß für den allgemeinen Fall eine solche Bestimmung nicht immer durchführbar ist, sondern ihrerseits voraussetzt, daß Benutzer- und Musterlösung bereits sehr weit miteinander übereinstimmen. Diese Einschränkung der Anwendungsbedingungen eines Strukturabgleichs mag für sich

genommen noch kein starkes Argument gegen die Implementation eines derartigen Algorithmus sein (sie wird allerdings umso gravierender, je fortgeschrittener die Aufgaben sind, und je vielfältiger damit die möglichen Lösungswege ausfallen). Schwerer wiegt jedoch, daß dieser zwingend auf die Bereitstellung von Musterlösungen angewiesen wäre, also dem Benutzer nur im Kontext einer mit einer Musterlösung versehenen Übungsaufgabe das Debugging zu einem gewissen Grad erleichtern könnte und ansonsten nutzlos wäre. Das bedeutet aber, daß dem Benutzer durch den Einsatz eines Strukturabgleichsalgorithmus im selben Maße die Gelegenheit genommen würde, eine Fähigkeit einzuüben, ohne die er außerhalb des Kontexts von Übungsaufgaben nicht auskommen wird (s. a. S. 6).

Gegen eine strukturelle Analyse des Benutzerprogramms selbst ist damit natürlich noch nichts gesagt. Es gibt vielmehr einige Möglichkeiten, wie dadurch die Funktionalität von SULLA sinnvoll erweitert werden kann. Eine Möglichkeit besteht z.B. darin, zu überprüfen, ob eine Benutzerlösung sich an bestimmte in der Aufgabenstellung enthaltene Vorgaben hält oder nicht. In einem LISP-Kurs kommen häufig Aufgaben vor, in denen der Schüler ein gegebenes Problem ausdrücklich rekursiv oder iterativ lösen soll, oder es gibt Beschränkungen bezüglich der Operatoren, die er verwenden sollte. Um zu überprüfen, ob diese Vorgaben eingehalten werden, wäre in beiden Fällen eine Analyse des Schülerprogramms nötig.

Es ist also, um diese kurze Erörterung zusammenzufassen, einiges Potential zu einer Weiterentwicklung und Erweiterung von SULLA vorhanden. Dennoch ist das System, auch bei seinem gegenwärtigen Entwicklungsstand, nicht wertlos. Bereits jetzt ist es zur Unterstützung bei der Bearbeitung von LISP-Aufgaben, wenn auch in einem etwas eingeschränkten Rahmen, durchaus einsatzfähig, und seine Einfachheit macht es zu einem guten Ausgangspunkt für die Untersuchung der Effektivität verschiedener ITS-Ansätze. Zudem besitzt SULLA den Vorteil einer sehr großen Konfigurierbarkeit für verschiedene Lehrpläne, die es von komplexeren ITS (wie etwa Andersons LISPITS) wesentlich unterscheidet.

## A Der Algorithmus zur Überprüfung der Layout-Konsistenz

Im folgenden soll, in aller Kürze, aber dennoch verständlich, der Algorithmus vorgestellt werden, den SULLA zum Aufdecken von Klammerungs- und Einrückungsfehlern verwendet. Je nachdem, ob es der Kürze dient, wird dabei die Darstellung mehr oder weniger formal ausfallen. Zunächst benötigen wir einige vorbereitende Definitionen:

Ein *Programm* sei eine Sequenz von Programmzeilen, deren Ordnung durch das Relationszeichen  $\succ$  bezeichnet werden soll. Eine *Zeile* in einem solchen Programm sei eine Sequenz von Zeichen eines Alphabets  $A$ , das eine Untermenge  $L$  von Leerzeichen besitzt. Mit  $z_{P,i}$  bezeichnen wir die  $i$ -te Zeile von Programm  $P$ , und  $indent_z$  sei die Anzahl der Leerzeichen am Anfang der Zeile  $z$  (also deren Einrückungsweite). Als *Klammer* bezeichnen wir im folgenden eine runde Klammer, die nicht innerhalb eines Strings vorkommt oder auf andere Weise ‘geschützt’ ist; also nur solche, die ein listenähnliches Konstrukt einleiten. Hieraus ergibt sich, was im folgenden unter ‘Klammerungsebene’ zu verstehen ist. Weiter benötigen wir den Begriff der ‘Anfangsebene’ einer Zeile. Hierbei handelt es sich um diejenige Klammerungsebene, auf der ein Konstrukt sich befindet, wenn es am Anfang der betreffenden Zeile steht. Die Anfangsebene einer Zeile  $z$  sei durch  $ae_z$  bezeichnet. Schließlich ist noch der Begriff der ‘Layout-Ebene’ einzuführen. Diese Größe, muß mit der Einrückungsebene in einem bestimmten Verhältnis stehen, damit das Layout des Programms konsistent ist. Wir schreiben für die Layout-Ebene von Zeile  $z$   $le_z$  und definieren:

Ist das erste Nicht-Leerzeichen in Zeile  $z$  keine schließende Klammer, gilt  $le_z = ae_z$ ; sonst:  $le_z = ae_z - schl_z$ , wobei  $schl_z :=$  die Anzahl der schließenden Klammern auf Zeile  $z$ , die *nicht* einer ebenfalls in  $z$  enthaltenen öffnenden Klammer entsprechen.

Der Sinn dieser Definition besteht darin, daß auch Codefragmente wie

```
(defun foo ()
  (do-something)
)
```

die Layout-Konsistenz eines Programms nicht stören sollen. Die normale Kodierungsweise für ein solches Konstrukt wäre

```
(defun foo ()
  (do-something))
```

Wenn aber die letzte schließende Klammer als erstes Nicht-Leerzeichen auf eine neue Zeile gesetzt wird, sollte diese in diesem Beispiel genauso weit eingerückt werden wie diejenige Zeile, auf der sich die entsprechende öffnende Klammer befindet.

Definieren wir nun, was unter Layout-Konsistenz zu verstehen ist:

**Definition A.1.** Unter einem ‘layout-konsistenten Programm’ verstehen wir ein Programm  $P$ , für das folgendes gilt:

1.  $\forall x, y \in P : (\neg \exists z \in P : ((x < z < y \vee x > z > y) \wedge (indent_z \leq indent_x \vee indent_z \leq indent_y))) \implies ((le_x < le_y \equiv indent_x < indent_y) \wedge (le_x > le_y \equiv indent_x > indent_y))$
2.  $\forall x \in P : (indent_x = 0 \equiv le_x = 0)$



Der Algorithmus, mit dem SULLA versucht, Klammerungs- und Einrückungsfehler aufgrund dieser Definition zu lokalisieren, besteht, grob gesprochen, darin, Elemente von  $PrK_P$  zu finden, zwischen denen eine Layout-Inkonsistenz im Sinne obiger Definition besteht, und diese Inkonsistenzen durch Einrückungs- oder Klammerungsfehler zu erklären. Er läßt sich dabei in drei größere Teile aufgliedern, die in den folgenden Abschnitten beschrieben werden.

## A.1 Erstellung der ‘Prä-Konsistenzrepräsentation’

Dies ist der notwendige erste Schritt, auf dem die folgenden Teile des Algorithmus aufbauen. Die Prä-Konsistenzrepräsentation enthält alle Zeilen, die auch in  $P$  enthalten sind, mit einer Ausnahme: Solche Zeilen, die auf vorhergehenden Zeilen beginnende Konstrukte abschließen und auf denen zugleich Konstrukte beginnen, die erst auf einer nachfolgenden Zeile abgeschlossen werden, werden gewissermaßen aufgespalten in zwei ‘Pseudo-Zeilen’, für die dies nicht mehr gilt. Die Spaltennummer, an der diese Aufspaltung geschieht, liegt irgendwo zwischen der Spaltennummer der letzten schließenden Klammer, die ein auf einer vorigen Zeile begonnenes Konstrukt abschließt, und der ersten öffnenden Klammer, die ein erst auf einer späteren Zeile abgeschlossenes Konstrukt beginnt. Die hieraus resultierenden Elemente der Prä-Konsistenzrepräsentation werden, obwohl sie jeweils nur ein Anfangs- bzw. Endsegment einer tatsächlichen Zeile von  $P$  repräsentieren, in den folgenden Schritten im wesentlichen genauso behandelt wie die anderen Elemente. (Der wichtigste Unterschied besteht darin, daß bezüglich solcher Elemente, die ein Endsegment einer Programmzeile repräsentieren, in der Ausgabe des Algorithmus kein Einrückungsfehler signalisiert wird. Dies ergäbe auch wenig Sinn, da ja bei Endsegmenten von Zeilen nicht sinnvoll von einer Einrückung gesprochen werden kann. Dennoch erhalten die Elemente der Prä-Konsistenzrepräsentation, die solche Endsegmente repräsentieren, zunächst eine Einrückungsweite zugewiesen, und zwar dieselbe wie die des entsprechenden Anfangssegments.)

Der Grund für die angesprochene Aufspaltung liegt darin, daß der Algorithmus auf dem Vergleich von Einrückungsweite und Layout-Ebene, bezogen auf ganze Zeilen, basiert. Ob also in einer Zeilenrepräsentation eine schließende vor einer öffnenden Klammer steht oder umgekehrt, macht für ihn keinen Unterschied. Durch die Zeilen-Aufspaltung wird verhindert, daß aufgrund dieser Tatsache Klammerungsfehler übersehen werden. Im folgenden haben wir also eine Prä-Konsistenzrepräsentation  $PrK_P$  eines Programms  $P$  als eine Sequenz von Zeilensegmentrepräsentationen zu denken. Wenn  $x$  eine solche Zeilensegmentrepräsentation ist, bezeichnen wir mit  $zn_x$  die Zeilennummer im ursprünglichen Programm, und mit  $spn_x$  die Spaltennummer, auf der der repräsentierte Zeilenabschnitt dort beginnt. Die Relation  $\succ$  bzw.  $\prec$  ist also durch  $zn_x$  und  $spn_x$  auf naheliegende Weise eindeutig definiert für alle Elemente einer Prä-Konsistenzrepräsentation.<sup>10</sup>

Zusätzlich zu den Repräsentationen der in  $P$  enthaltenen Zeilen (bzw. von deren Segmenten) befinden sich in  $PrK_P$  noch Repräsentationen für je eine leere ‘nullte’ Zeile  $z_0$  (mit  $zn_{z_0} = 0 \wedge spn_{z_0} = 0 \wedge indent_{z_0} = 0$ ) und eine letzte Zeile  $z_n$  mit  $n := zn_{z_n} = \max\{i \in \mathbb{N} \mid z_{P,i} \in P\} + 1 \wedge spn_{z_n} = 0 \wedge indent_{z_n} = 0$ . Die Aufnahme dieser zusätzlichen Repräsentationen hat den Zweck, daß die zweite Bedingung von Definition A.1 dadurch genau dann erfüllt ist, wenn auch die erste Bedingung zutrifft. Tabelle 4 veranschaulicht die Beziehung zwischen einem Programm und seiner Prä-Konsistenzrepräsentation. Man beachte, daß die Klammerungsfehler in

<sup>10</sup>In der Implementation werden für die Zeilennummern von Zeilenendsegmenten nicht-ganzzahlige Werte verwendet, also etwa 7,1, wenn die ursprüngliche Zeilennummer 7 ist. Dies dient aber nur der Effizienz und braucht in der gegenwärtigen Betrachtung keine Rolle zu spielen.

$z \in P$			$x \in PrK_P$			
$zn_z$	$le_z$	Inhalt	$zn_x$	$spn_x$	$le_x$	Inhalt
1	0	(defun second (x)	0	0	0	(defun second (x)
2	1	(when (null (cdr x))	1	0	0	(when (null (cdr x))
3	2	(error "too short: ~A"	2	0	1	(when (null (cdr x))
4	3	x))) (((cadr x)	3	0	2	(error "too short: ~A"
			4	0	3	x)))
			4	5	-1	((cadr x)
5	0	)	5	0	0	)
			6	0	0	

Tabelle 4: Ein ‘Programm’ und seine Prä-Konsistenzrepräsentation

Zeile 4 zusammengekommen keine Auswirkung auf die Layout-Ebenen der Programmzeilen zeigen und sich erst in der Prä-Konsistenzrepräsentation bemerkbar machen.

## A.2 Erstellung der ‘Konsistenzrepräsentation’

Hier wird die Prä-Konsistenzrepräsentation auf Inkonsistenzen untersucht, und alle gefundenen Inkonsistenzen werden in einer Liste gespeichert. Bevor wir definieren, was eine Inkonsistenz ist, soll noch eine Relation definiert werden, sie im folgenden recht praktisch sein wird:

**Definition A.2.** Ein Element  $x$  einer Prä-Konsistenzrepräsentation  $PrK_P$  ist *Nachfolger* eines anderen Elements  $y \in PrK_P$  genau dann, wenn

$$x \succ y \wedge \neg \exists z \in PrK_P : (x \succ z \succ y \wedge (indent_z \leq indent_x \vee indent_z \leq indent_y)).$$

Das Symbol für diese Relation sei  $\rightarrow$ . Der Sachverhalt, daß  $x$  *rekursiv* Nachfolger von  $y$  ist, soll durch  $x \rightarrow^* y$  ausgedrückt werden.

Eine Inkonsistenz definieren wir nun wie folgt:

**Definition A.3.** Eine Inkonsistenz  $I_{x,y}$  zwischen zwei Elementen  $x$  und  $y$  einer Prä-Konsistenzrepräsentation (die in dem Fall auch als Anfangs- bzw. End-„Zeile“ der Inkonsistenz bezeichnet und als  $az_I$  bzw.  $ez_I$  geschrieben werden) ist ein Tupel  $(x, y, typ_I, grad_I)$ , von dem wir genau dann sagen, es *existiere*, wenn die folgende Aussage wahr ist:

$$y \rightarrow x \wedge \begin{aligned} &\text{a) } le_x \geq le_y \wedge indent_x < indent_y \vee \\ &\text{b) } le_x \leq le_y \wedge indent_x > indent_y \vee \\ &\text{c) } le_x < le_y \wedge indent_x = indent_y \vee \\ &\text{d) } le_x > le_y \wedge indent_x = indent_y \end{aligned}$$

In den Fällen a) und d) gilt für den *Typ* der Inkonsistenz  $typ_I := -1$ ; sonst ist  $typ_I := +1$ . Die Komponente  $grad_I$  (den *Grad* der Inkonsistenz) berechnen wir

wie folgt:

$$grad_I := \begin{cases} le_x - le_y + 1 & \text{in Fall a)} \\ le_y - le_x + 1 & \text{in Fall b)} \\ le_y - le_x & \text{in Fall c)} \\ le_x - le_y & \text{in Fall d)} \end{cases}$$

Den Sachverhalt, daß zwischen zwei Elementen  $x$  und  $y$  einer Prä-Konsistenzrepräsentation eine Inkonsistenz besteht, kann man symbolisch so ausdrücken:  $x \text{ inc } y$ .

Es ist leicht zu sehen, daß ein Programm  $P$  genau dann layout-konsistent ist, wenn zwischen keinen zwei Elementen von  $PrK_P$  eine Inkonsistenz existiert, und außerdem die 2. Bedingung von Definition A.1 erfüllt ist. Für den nächsten Schritt des Algorithmus müssen wir noch einige weitere Begriffe definieren:

**Definition A.4.** Eine Inkonsistenz  $I_1$  zwischen zwei Elementen  $x_1$  und  $y_1$  von  $PrK_P$  *umspannt* eine andere Inkonsistenz  $I_2$ , die zwischen  $x_2$  und  $y_2$  ( $x_2, y_2 \in PrK_P$ ) besteht, genau dann, wenn

$$(x_1 \preceq x_2 \wedge y_1 \succ y_2) \vee (x_1 \prec x_2 \wedge y_1 \succeq y_2).$$

Formal schreibt man diesen Sachverhalt so:  $I_1 \supset I_2$ . Da die in einem Programm enthaltenen Inkonsistenzen bezüglich der Umspannungsrelation eine Menge von disjunkten Bäumen bilden (d.h. solchen, die paarweise keine gemeinsamen Unterbäume enthalten), wird eine Inkonsistenz, die keine andere Inkonsistenz umspannt, *Blatt-Inkonsistenz* genannt. Zwischen zwei Inkonsistenzen  $I$  und  $J$  besteht ein *Inkonsistenzpfad* von  $I$  nach  $J$ , wenn es Inkonsistenzen  $I_0, \dots, I_n$  gibt, so daß  $I_0 = I$ ,  $I_n = J$  und  $I_0 \supset I_1 \supset \dots \supset I_n$ .

**Definition A.5.** Unter einem *potentiellen Einrückungsfehler*  $E_{P,J}$  innerhalb eines Programms  $P$  verstehen wir einen Sachverhalt, der genau dann existiert, wenn für ein gegebenes Intervall  $J \subset \mathbb{N}$  und eine zusammenhängende Menge von Programmzeilen  $Z_{E_{P,J}} := \{z_{P,i} \mid i \in I\}$  die Summe  $\sum_I grad_I$  über alle Inkonsistenzen innerhalb von  $PrK_P$  kleiner wird, falls die Einrückung aller Zeilen in  $Z_{E_{P,J}}$  um einen Betrag  $|\delta| \neq 0$  verändert wird. Diese Definition überträgt sich analog auf Einrückungsfehler  $E_{PrK_P,J}$  innerhalb einer Prä-Konsistenzrepräsentation  $PrK_P$  mit  $Z_{E_{PrK_P,J}} \in \{Z \mid Z \subset PrK_P \wedge \neg \exists x, y \in Z, z \in PrK_P \setminus Z : (x \prec z \prec y)\}$ .

Hierbei ist zu beachten, daß nur genau denjenigen Einrückungsfehlern  $E_{PrK_P,J}$  in  $PrK_P$  ein Einrückungsfehler in  $P$  entspricht, für die  $Z_{E_{PrK_P,J}}$  mindestens eine Zeile  $z$  mit  $spn_z = 0$  enthält.

**Definition A.6.** Ein *potentieller Klammerungsfehler*  $K_{P,i,j}$  liegt in einem Programm  $P$  genau dann vor, wenn es eine Zeile  $z_{P,i} \in P$  und eine Spalte  $j$  gibt, so daß die Summe  $\sum_I grad_I$  über alle Inkonsistenzen innerhalb von  $PrK_P$  kleiner wird, falls in Zeile  $z_{P,i}$  und Spalte  $j$  eine schließende oder öffnende Klammer hinzugefügt oder entfernt wird.

**Definition A.7.** Ein Element  $x$  einer Prä-Konsistenzrepräsentation  $PrK_P$  ist an ein anderes Element  $y$  genau dann  $\delta$ -eintrückungsgebunden, wenn zwischen  $x$  und  $y$  noch keine Inkonsistenz besteht, aber eine Erhöhung von  $indent_x$  um  $\delta$  zu einer solchen Inkonsistenz (d.h. mit  $az_I = x \wedge ez_I = y$  oder  $az_I = y \wedge ez_I = x$ ) führen würde (was voraussetzt, daß  $x \rightarrow^* y \vee y \rightarrow^* x$ ). Des weiteren ist  $x$  *rekursiv* an  $y$   $\delta$ -eintrückungsgebunden, wenn  $x \rightarrow^* y \vee y \rightarrow^* x$ , noch keine Inkonsistenz  $I$  mit  $az_I \succeq x \wedge ez_I \preceq y$  bzw.  $az_I \succeq y \wedge ez_I \preceq x$  besteht und drittens die Erhöhung von  $indent_x$  um  $\delta$  zu einer Inkonsistenz führt, falls nicht auch  $indent_y$  verändert wird. Das Relationssymbol für  $\delta$ -Einrückungsgebundenheit sei  $\rightsquigarrow_\delta$ .

### A.3 Reparatur der gefundenen Inkonsistenzen

Inkonsistenzen im Layout können auf zweierlei Weise entstehen: Zum einen durch Fehler in der Einrückungsweite, zum anderen durch fehlende oder überzählige Klammern (leider läßt allein anhand des Layouts nicht entscheiden, ob schließende bzw. öffnende Klammern fehlen, oder ob die jeweils komplementären Klammern überzählig sind). Da Layout und Klammerung sich aufeinander beziehen, ist es nicht mit völliger Sicherheit möglich, zwischen den beiden Fällen zu unterscheiden. Es können aber aufgrund bestimmter Heuristiken Vermutungen darüber angestellt werden, welcher Fall bei einer gegebenen Menge von Inkonsistenzen vorliegt. Die Heuristiken, die in SULLA dabei zum Einsatz kommen, sind folgende:

- (H1) Fehler, und zwar sowohl Einrückungs- wie Klammerungsfehler, sind selten genug, daß nicht mehr Fehler angenommen werden sollten, als aufgrund der vorliegenden Inkonsistenzen notwendig ist. Mit anderen Worten: Inkonsistenzen sollten auf möglichst wenige Fehler zurückgeführt werden.
- (H2) Wenn eine Zeile gar nicht eingerückt ist, sollte sie auch nicht eingerückt sein; und wenn eine Zeile eingerückt ist, gehört sie auch eingerückt. Der Grund hierfür ist einfach, daß die 2. Bedingung von Definition A.1 so klar und einleuchtend ist, daß davon ausgegangen werden kann, daß ein LISP-Programmierer, selbst als Anfänger, keinen Fehler diesbezüglich begeht.
- (H3) Andererseits ist zu vermuten, daß Einrückungsfehler, die nicht in diese sehr unwahrscheinliche Kategorie fallen, nicht so selten sind, als daß bei einer gegebenen Menge von Inkonsistenzen, die entweder auf einen solchen Einrückungsfehler oder auf zwei Klammerungsfehler zurückgeführt werden kann, nicht eher von einem Einrückungsfehler auszugehen ist.

Dafür, wie sich ein Einrückungsfehler  $E$  innerhalb einer Prä-Konsistenzrepräsentation in zusätzlichen Inkonsistenzen auswirken kann, gibt es zwei Möglichkeiten:

1.  $E$  erzeugt genau eine Blatt-Inkonsistenz  $I$ . In diesem Fall gibt es zunächst zwei Möglichkeiten:
  - (a) Die Einrückung (u.a.) von  $az_I$  ist zu erhöhen, wenn  $typ_I = -1$ , und sonst zu verringern.
  - (b) Die Einrückung (u.a.) von  $ez_I$  ist zu verringern, wenn  $typ_I = -1$ , sonst zu erhöhen.
2.  $E$  erzeugt genau zwei Blatt-Inkonsistenzen  $I$  und  $J$ . Zu verändern ist die Einrückung von (u.a.)  $ez_I$ , und zwar ist sie zu erhöhen, wenn  $typ_I = +1$ , und zu verringern, wenn  $typ_I = -1$ . Außerdem gilt:
  - (a)  $J$  folgt unmittelbar auf  $I$ , d.h.  $ez_I \succeq az_J$ , und es gibt keine Inkonsistenz  $H$  mit  $ez_I \preceq az_H \wedge ez_H \preceq az_J$ ,
  - (b)  $typ_I \neq typ_J$ .

Wir können also von zwei Inkonsistenzen  $I$  und  $J$ , für die die Bedingungen 2a und 2b gelten, gemäß Heuristik (H3) annehmen, daß diese durch eine falsche Einrückung von mindestens  $ez_I$  erzeugt worden sind. Es ist zwar auch möglich, daß  $I$  und  $J$  durch zwei Einrückungsfehler erzeugt worden sind, die dann mindestens die Zeilen  $az_I$  und  $ez_J$  betreffen, aber diese Möglichkeit wird aufgrund der Heuristik (H1) nicht berücksichtigt.

Für alle anderen Inkonsistenzen ist der Rückschluß auf den verursachenden Fehler komplizierter. Die Bedingung, anhand derer SULLA entscheidet, daß eine Inkonsistenz  $I$ , die nicht die obigen Bedingungen 2a und 2b erfüllt, auf einen Einrückungsfehler zurückzuführen ist, ist folgende:

- (B) Entweder es gibt keine globale Inkonsistenz, die  $I$  umspannt, oder, wenn es eine solche Inkonsistenz  $G$  gibt:  $typ_G \neq typ_I$  oder  $grad_I > grad_G$ .

Der Grund für diese Bedingung liegt darin, daß Nicht-Blatt-Inkonsistenzen, die von SULLA nicht auf einen Einrückungsfehler zurückgeführt werden, wegen der Heuristik (H1) möglichst auf von ihnen umspannte Inkonsistenzen zurückgeführt werden sollen. Betrachten wir hierzu ein einfaches Beispiel. Gegeben sei das in folgender Tabelle dargestellte (fehlerhafte) Programmfragment:

1	0	(defun second (x)
2	1	(when (null (cdr x))
3	2	(error "second: list too short")))
4	0	(cadr x)
5	-2	))

Wie an den in der zweiten Spalte dargestellten Layout-Ebenen der einzelnen Zeilen zu erkennen ist, enthält das Fragment folgende Inkonsistenzen:  $I := (1, 5, -1, 2)$  und  $J := (2, 4, -1, 1)$ .  $I$  ist außerdem global und umspannt  $J$ . Da  $I$  vom selben Typ ist wie  $J$  und der Grad von  $J$  nicht größer ist als der von  $I$ , könnte  $J$  also nach Bedingung (B) nicht durch einen Einrückungsfehler erklärt werden. Damit bliebe nur, einen Klammerungsfehler anzunehmen. Eine Korrektur dieses Klammerungsfehlers, d.h. das Hinzufügen einer öffnenden oder das Wegnehmen einer schließenden Klammer in einer der beiden in Frage kommenden Zeilen (Nr. 2 oder Nr. 3), würde aber nicht nur  $J$  beheben, sondern auch den Grad von  $I$  um 1 absenken. Um dann  $I$  ebenfalls zu beheben, müßte also nur eine Klammer (und nicht zwei) hinzugefügt oder entfernt werden. Wir hätten also  $I$  und  $J$  auf insgesamt zwei Klammerungsfehler zurückgeführt. Würde dagegen  $J$  auf einen Einrückungsfehler zurückgeführt werden, hätten wir zusätzlich zu diesem immer noch zwei Klammerungsfehler, da sich durch die nötige Änderung der Einrückung weder Layout-Ebene noch Einrückungstiefe von Zeile 1 oder Zeile 5 ändern würde, und folglich  $I$  keinerlei Abschwächung erführe.

Wenn es dagegen keine unerklärte globale Inkonsistenz wie  $I$  gäbe, die  $J$  umspannt, würde einer 'Erklärung' von  $J$  durch einen Einrückungsfehler nichts im Wege stehen. Es wäre sogar, wieder gemäß (H1), sinnvoller, einen solchen Fehler anzunehmen, denn daß keine globale,  $J$  umspannende Inkonsistenz existiert, bedeutet ja, daß das Hinzufügen oder Entfernen von Klammern zwischen  $az_J$  und  $ez_J$  notwendig eine zusätzliche Inkonsistenz zur Folge hätte. Die anderen beiden in Bedingung (B) genannten Alternativen verdanken sich ebenfalls der Heuristik (H1), denn in beiden Fällen würde ein vollständiges Zurückführen auf Klammerungsfehler in zusätzlichen Inkonsistenzen resultieren, die nicht entstünden, wenn die Inkonsistenz zumindest teilweise auf einen Einrückungsfehler zurückgeführt würde.

Das hier Gesagte setzt voraus, daß die umspannende Inkonsistenz in Bedingung (B) nicht auf einen Einrückungsfehler zurückgeführt werden kann. Dies ist der Grund dafür, daß diese Inkonsistenz eine globale sein muß, denn nur von globalen Inkonsistenzen läßt sich (aufgrund von Heuristik (H2)) sagen, daß sie nicht durch Einrückungsfehler zu erklären sind.

Nach diesen Vorbereitungen kann nun der eigentliche Algorithmus, zumindest im Kern, beschrieben werden. Dies soll der Kürze halber teilweise in Form von

Pseudo-Code geschehen. Der Algorithmus ist zwar in PROLOG, also einer relationalen Programmiersprache implementiert, aber zum besseren Verständnis soll er hier in einem prozeduralen Paradigma dargestellt werden. Einige Details werden dabei aus Platzgründen nicht angesprochen werden können; für diese sei auf die beiliegende Diskette verwiesen, die den gesamten (zusätzlich kommentierten) Quellcode von SULLA enthält.

Funktion *check-consistency*

Parameter:  $P$

Rückgabewert:  $OR$  (Ausgaberepräsentation)

$PrK_P :=$  Prä-Konsistenzrepräsentation von  $P$

$(PrK'_P, OR) := \text{checkconsis}(PrK_P, \{\})$

Für alle  $z_i \in PrK_P$  und  $z'_i \in PrK'_P$  mit  $zn_{z_i} = zn_{z'_i} \wedge spn_{z_i} = spn_{z'_i} = 0$ :  
 $OR := OR \cup x$ , wobei  $x$  die Vermutung repräsentiert, daß bezüglich  $z_i$  ein Einrückungsfehler vorliegt und die Einrückung der entsprechenden Zeile in  $P$  um  $indent_{z'_i} - indent_{z_i}$  verändert werden sollte.

Funktion *checkconsis*

Parameter:  $PrK_P, OR$

Rückgabewert:  $(PrK''_P, OR'')$

$K_P :=$  Konsistenzrepräsentation, erstellt auf der Grundlage von  $PrK_P$

Wenn  $K_P$  leer ist:  $PrK''_P := PrK_P, OR'' := OR$ ; sonst:

$(PrK'_P, OR') := \text{checkconsis}'(PrK_P, K_P, \text{none}, OR)$

$(PrK''_P, OR'') := \text{checkconsis}(PrK'_P, OR')$

Funktion *checkconsis'*

Parameter:  $K_P, PrK_P, G$  : globale Inkonsistenz oder none,

$a, e \in PrK_P, OR$

Rückgabewert:  $(PrK''_P, OR'', G''')$

Nacheinander für alle  $I \in M := \{I \in K_P \mid az_I \succeq a \wedge ez_I \preceq e \wedge \neg(az_I = a \wedge ez_I = e)\}$  mit  $\neg \exists J \in M : (J \supset I)$ :

Wenn  $I$  global ist:  $G' := I$ ;

sonst, wenn  $az_I \succeq ez_G$ :  $G' := \text{none}$ ;

sonst:  $G' := G$ .

$(PrK'_P, OR', G'') := \text{checkconsis}'(K_P, PrK_P, G', az_I, ez_I, OR)$

$(PrK''_P, OR'', G''') := \text{repair-inconsistency}(I, K_P, PrK'_P, G'', OR')$

$G := G'''; PrK_P := PrK''_P; OR := OR''$

Funktion *repair-inconsistency*

Parameter:  $I, K_P, PrK_P, G$  : globale Inkonsistenz oder none,  $OR$

Rückgabewert:  $(PrK'_P, OR', G')$

Sei  $x \in PrK_P$  mit  $zn_x = zn_{az_I} \wedge spn_x = spn_{az_I}$  und

sei  $y \in PrK_P$  mit  $zn_y = zn_{ez_I} \wedge spn_y = spn_{ez_I}$ .

Wenn zwischen  $x$  und  $y$  eine Inkonsistenz  $I'$  existiert:

$(PrK'_P, OR', G') := \text{repair}(I', K_P, PrK'_P, G'', OR')$ ;  
sonst:  $(PrK'_P, OR', G') := (PrK_P, OR, G)$

Funktion `repair`

Parameter:  $I \in K_P$ ,  $K_P$ ,  $PrK_P$ ,  $G$ : globale Inkonsistenz oder `none`,  $OR$

Rückgabewert:  $(PrK'_P, OR', G')$

Wenn es eine Inkonsistenz  $J \in K_P$  gibt, so daß  $J$  unmittelbar auf  $I$  folgt und  $typ_I \neq typ_J$ :

Sei  $x \in PrK_P$  mit  $zn_x = zn_{az_J} \wedge spn_x = spn_{az_J}$  und

sei  $y \in PrK_P$  mit  $zn_y = zn_{ez_J} \wedge spn_y = spn_{ez_J}$ .

Wenn zwischen  $x$  und  $y$  eine Inkonsistenz  $J'$  existiert:

Es wird eine neue Prä-Konsistenzrepräsentation  $PrK'_P$  erzeugt, die  $PrK_P$  entspricht mit der Ausnahme, daß alle Elemente  $z_1, \dots, z_n$ , an die  $x$  rekursiv  $\delta_1$ -einrückungsgebunden ist (wobei o.B.d.A. gelten möge, daß  $z_1 := x$ ), ersetzt werden durch andere  $z'_1, \dots, z'_n$ , so daß für alle  $i \in \{1, \dots, n\}$  sich  $z_i$  von  $z'_i$  nur dadurch unterscheidet, daß  $indent_{z'_i} - indent_{z_i} = \delta_i$ . Die einzelnen  $\delta_i$  werden nun für  $i \in 2, \dots, n$  und in Abhängigkeit von  $\delta_1$  so gewählt, daß

1. ihre Beträge möglichst klein bleiben, und
2.  $\neg \exists j \in \{1, \dots, n\} : (z_j \rightsquigarrow_\delta z_i \wedge z'_j \text{ inc } z'_i)$ .

Der Wert für  $\delta_1$  wird so gewählt, daß

1.  $\delta_1 \neq 0$ ,
2.  $\delta_1, \dots, \delta_n$  nicht gegen die Heuristik (H2) verstoßen, und
3.  $I$  und  $J$  möglichst weit abgeschwächt werden, ohne daß an der Stelle von  $I$  oder  $J$  eine neue Inkonsistenz von komplementärem Typ entsteht.

Wenn ein solches  $\delta_1$  nicht existiert, bleibt  $PrK'_P$  undefiniert;

sonst:  $G' := G$ ,  $OR' := OR$ .

Wenn  $PrK'_P$  noch undefiniert ist und entweder  $G = \text{none}$  ist oder  $typ_G \neq typ_I$  oder  $grad_I > grad_G$  (s. Bedingung (B)):

Im letzteren Fall sei  $J := (az_I, ez_I, typ_I, grad_I)$ , sonst  $J := I$ .

Es wird ein  $x \in \{az_J, ez_J\}$  gewählt und eine neue Prä-Konsistenzrepräsentation  $PrK'_P$  erzeugt, die  $PrK_P$  entspricht mit der Ausnahme, daß analog zu oben alle Elemente, an die  $x$  rekursiv  $\delta_1$ -einrückungsgebunden ist, ersetzt werden durch andere, die sich von den entsprechenden Elementen in  $PrK_P$  nur in ihrer Einrückung unterscheiden. Dabei sind  $x$  und  $\delta_1$  so gewählt, daß

1.  $\delta_1 \neq 0$ ,
2. diese Veränderung nicht gegen die Heuristik (H2) verstößt, und
3. die Inkonsistenz  $J$  möglichst weit abgeschwächt wird, ohne daß an der Stelle von  $J$  eine neue Inkonsistenz von komplementärem Typ entsteht.

Wenn ein solches  $\delta_1$  nicht existiert, bleibt  $PrK'_P$  undefiniert;  
sonst:  $G' := G$ ,  $OR' := OR$ .

Wenn  $PrK'_P$  noch immer undefiniert ist, wird  $I$  auf einen Klammerungsfehler zurückgeführt, und es wird wie folgt vorgegangen:

$S$  := die Menge der ‘verdächtigen Zeilen’, also derjenigen Elemente von  $PrK_P$ , auf denen vermutlich Klammern fehlen oder überflüssig sind. Die Auswahl dieser Elemente von  $PrK_P$  berücksichtigt,

1. ob das Hinzufügen oder Entfernen einer Klammer eine neue Inkonsistenz hervorrufen würde (in diesem Fall würde das betreffende Element nicht ‘verdächtigt’ werden), und
2. für welches Element  $e$  von  $PrK_P$  erstmals  $le_e < 0$  gilt. In diesem Fall gelten, wenn  $typ_I = -1$ ,  $e$  und die nachfolgenden Elemente nicht mehr als verdächtig. Ausnahme: Wenn  $e \in S$  und das erste Nicht-Leerzeichen in der durch  $e$  repräsentierten Zeile eine schließende Klammer ist, gilt  $e$  auch dann noch als verdächtig, wenn  $le_e < 0$ .

Eine neue Prä-Konsistenzrepräsentation  $PrK'_P$  wird erzeugt, die  $PrK_P$  entspricht mit der Ausnahme, daß alle Elemente  $z_1, \dots, z_n \in \{z \in PrK_P \mid z \succ \max_{\prec} S\} \cap M$  mit

$$M := \begin{cases} \{\max_{\prec} S\} & \text{falls das erste Nicht-Leerzeichen von } \max_{\prec} S \\ & \text{eine schließende Klammer ist} \\ \{\} & \text{sonst} \end{cases}$$

ersetzt werden durch  $z'_1, \dots, z'_m$ , so daß für alle  $i \in \{1, \dots, m\}$  sich  $z'_i$  von  $z_i$  nur dadurch unterscheidet, daß  $le_{z'_i} = le_{z_i} + typ_I grad_I$ .

Wenn  $typ_G = typ_I$ :

$G'$  wird so definiert, daß  $G' = G$  mit der Ausnahme, daß  $grad_{G'} := |grad_G - grad_I|$  und, sofern  $grad_G < grad_I$ ,  $typ_{G'} := -typ_G$ ;  
sonst:  $G' = G$

$OR' := OR \cup \{x\}$ , wobei  $x$  die Vermutung repräsentiert, daß auf den Zeilenabschnitten, die den in  $S$  enthaltenen Elementen von  $PrK_P$  entsprechen, ein  $grad_I$ maliger Klammerungsfehler vorliegt. Je nach dem Wert von  $typ_I$  werden dabei entweder (falls  $typ_I = -1$ ) überzählige schließende bzw. fehlende öffnende Klammern vermutet, oder umgekehrt.

□



## B Erstellung von Musterlösungsdateien

In den folgenden Absätzen sollen die Details bezüglich der Erstellung von Musterlösungsdateien, die in Abschnitt 4.1.2 aus Platzgründen übersprungen worden sind, nachgeholt werden. Es werden dabei allerdings dementsprechend tiefere Kenntnisse von LISP vorausgesetzt werden.

Als ein einfaches Beispiel für eine Musterlösungsdatei betrachte man nochmals die bereits in Abschnitt 4.1.2 verwendete Abbildung:

```
;;; Musterlösungen zu Blatt 1 (Beispiel)

;;; Hilfsfunktionen

(defun car-eq (l1 l2)
  (eq (car l1) (car l2)))

;;; Aufgaben

; \A 1.1
; \P 3

; Definieren Sie eine Funktion 'second-element', die zu einer
; mindestens zweielementigen Liste deren zweites Element liefert.
; Wenn die Liste weniger als zwei Elemente enthält, soll nil
; zurückgegeben werden.
; Beispiele:
; (second-element '(a b c)) liefert B,
; (second-element nil) liefert NIL.

; \F second-element
; \E car-eq
; \R 50

; \test '(() )
; \test '('(a))
; \test '('(b (c)))
; \test '('(d (e) f))

(defun second-element (x)
  (car (cdr x)))
```

In dieser Musterlösungsdatei ist, wie schon gesagt, der Aufgabentext nur der Verständlichkeit halber mit enthalten. Was benötigt wird, sind lediglich die Definitionen der Musterlösungsfunktionen, gewisse Direktiven, die SULLA mitteilen, wie mit der Datei umzugehen ist, und ggf. Hilfsfunktionen. Es folgt eine Auflistung der von SULLA interpretierbaren Direktiven:

- Die A-Direktive dient der Aufgabenmarkierung, wie in 4.1.2 bereits ausreichend beschrieben.
- Die P-Direktive gibt an, wieviel Punkte für die Aufgabe maximal zu vergeben sind, im vorliegenden Beispiel also 3.

- Mit der F-Direktive wird dem System mitgeteilt, welcher Operator zu testen ist. Hierbei muß das nach dem F stehende Wort als ein LISP-Symbol interpretierbar sein. Es muß sich nicht darum gekümmert werden, ob die Muster- und die Schülerlösung sich in verschiedenen Packages befinden, da SULLA sich bei der Referenzierung der zu testenden Funktionen bzw. Makros allein nach dem *Namen* desjenigen Symbols richtet, als dessen Repräsentation das nach dem F stehende Wort vom LISP-Interpreter aufgefaßt wird. Da bei diesem Vorgang die standardmäßigen Interpreter-Einstellungen gelten, wird durch die F-Direktive des obigen Beispiels der Name der zu definierenden Funktion mit "SECOND-ELEMENT" festgelegt.
- Mit der E-Direktive wird angegeben, wie die Listen der Rückgabewerte, die durch die Aufrufe der Musterlösung entstehen, mit denen der Schülerlösung zu vergleichen sind. SULLA erwartet nach dem E die Repräsentation eines LISP-Symbols, das spätestens innerhalb der Musterlösungsdatei als Vergleichsoperator definiert ist. Dabei ist folgendes zu beachten:
  1. Der Operator sollte genau dann nicht nil liefern, wenn die Resultatliste der Schülerlösung ein korrektes Ergebnis darstellt.
  2. Das Package, in dem sie definiert ist, sollte auch nach dem Laden der Musterlösungsdatei noch aktiv sein.
  3. Wenn der Vergleichsoperator erst in der Musterlösungsdatei definiert ist, sollte die Definition vor der ersten Aufgabenmarkierung stehen, damit sie von SULLA nicht als Teil der Musterlösung selbst angesehen wird.
- Die test-Direktiven geben dem System die für den Test der Schülerlösung zu verwendenden Parameterlisten an, wie in 4.1.2 beschrieben.
- Die R-Direktive schließlich ist optional und ebenfalls in Abschnitt 4.1.2 ausreichend behandelt.

Abschließend noch eine Bemerkung zur Gestaltung der Aufgabenstellungen. Damit die Schülerlösung nicht auf in der Musterlösungsdatei definierte Operatoren zurückgreifen kann, werden vor dem Laden des Schülerprogramms alle internalen Symbole gelöscht ('de-interniert'). Wenn also die Aufgabenstellung vorsieht, daß der Rückgabewert der zu definierenden Funktion internale, nicht bereits in den Funktionsargumenten vorkommende Symbole enthält, wie z.B. in Abb. 4 dargestellt, wird die Funktion der Schülerlösung niemals dasselbe liefern können wie die der Musterlösung. Die von der Schülerlösung gelieferten Symbole werden mit denen, die die Musterlösung zurückgibt, nur den Namen gemein haben, und dies kann die Aufgabe, einen Vergleichsoperator für die E-Direktive zu definieren, u.U. erheblich komplizierter machen. Günstiger wäre es, auf internale Symbole zugunsten von Schlüsselwort-Symbolen oder Strings zu verzichten. Statt `symbol`, `list` und `other` würde die Aufgabe in Abb. 4 dann beispielsweise `:symbol`, `:list` und `:other` als Rückgabewerte verlangen, was aus stilistischen Gründen ohnehin vorzuziehen wäre.

```
; Definieren Sie eine Funktion type, die je nach dem Typ ihres  
; Arguments 'symbol, 'list oder 'other zurückgibt.  
; Beispiel:  
; (type 'a) liefert 'symbol.
```

Abbildung 4: Beispiel für eine ungünstige Aufgabenstellung (s. Text)

## Literatur

- [ABCL90] ANDERSON, J. R., C. F. BOYLE, A. CORBETT und M. W. LEWIS: *Cognitive modeling and intelligent tutoring*. Artificial Intelligence, 42:7–49, 1990.
- [ABFR87] ANDERSON, J. R., C. F. BOYLE, R. FARRELL und B. J REISER: *Cognitive principles in the design of computer tutors*. In: MORRIS, P. (Herausgeber): *Modeling Cognition*. Wiley, 1987.
- [ABY85] ANDERSON, J. R., C. F. BOYLE und G. YOST: *The geometry tutor*. In: *Proceedings of the International Joint Conference on Artificial Intelligence*, Seiten 1–7, 1985.
- [ACKP95] ANDERSON, J. R., A. T. CORBETT, K. KOEDINGER und R. PELLETIER: *Cognitive tutors: Lessons learned*. The Journal of Learning Sciences, 4:167–207, 1995.
- [AJ85] ANDERSON, J. R. und R. JEFFRIES: *Novice LISP errors: Undetected losses of information from working memory*. Human Computer Interaction, 1:107–131, 1985.
- [And83] ANDERSON, J. R.: *The Architecture of Cognition*. Harvard University Press, Cambridge, Massachusetts, 1983.
- [And90] ANDERSON, J. R.: *Cognitive Psychology and Its Implications*. Freeman, New York, 1990.
- [And93] ANDERSON, J. R.: *Rules of the Mind*. Erlbaum, Hillsdale, New Jersey, 1993.
- [AR85] ANDERSON, J. R. und B. J. REISER: *The LISP Tutor*. Byte, Seiten 159–175, April 1985.
- [Ber91] BERLINER, D.: *Educational Psychology and Pedagogical Expertise: New Findings and New Opportunities for Thinking about Training*. Educational Psychologist, 26:145–155, 1991.
- [BGH<sup>+</sup>99] BÖNNEN, C., J. GERSTENHÖFER, N. HARTMANN, A. HOLZWARTH, K. KOBER, N. LENSCHOW und B. MEISSNER: *Abschlußbericht des Projekts PLOT*. Technischer Bericht, Institut für Semantische Informationsverarbeitung, Universität Osnabrück, 1999.
- [Bru99] BRUSILOVSKY, P.: *Adaptive and Intelligent Technologies for Web-based Education*. In: ROLLINGER, C. und C. PEYLO (Herausgeber): *Special Issue on Intelligent Systems and Teleteaching, Künstliche Intelligenz, 4*, Seiten 19–25. Gesellschaft für Informatik e. V., 1999.
- [CA92] CORBETT, A. T. und J. R. ANDERSON: *The LISP intelligent tutoring system: Research in skill acquisition*. In: LARKIN, J., R. CHABAY und C. SCHEFTIC (Herausgeber): *Computer Assisted Instruction and Intelligent Tutoring Systems: Establishing Communication and Collaboration*. Erlbaum, Hillsdale, New Jersey, 1992.
- [DL92] DION, P. und R. LELOUCHE: *Application de la méthodologie du traçage de modèle à un environnement d'apprentissage utilisant une stratégie pédagogique non directive*. In: FRASSON, C., G. GAUTHIER und

- G. I. MCCALLA (Herausgeber): *Intelligent Tutoring Systems*, Nummer 608 in *Lecture Notes in Computer Science*, Seiten 421–434, Berlin, 1992. Springer.
- [McC81] MCCARTHY, J.: *History of LISP*. In: WEXELBLATT, RICHARD L. (Herausgeber): *History of Programming Languages*. Academic Press, New York, 1981.
- [New90] NEWELL, A.: *Unified Theories of Cognition*. Harvard University Press, Cambridge, Massachusetts, 1990.
- [Pea86] PEA, R.D.: *Language-independent conceptual bugs in novice programming*. *Journal of Educational Computing*, 2:25–36, 1986.
- [PPS87] PARK, O., R. S. PEREZ und R. J. SEIDEL: *Intelligent CAI: Old Wine in New Bottles or a New Vintage?* In: KEARSLEY, G. P. (Herausgeber): *Artificial Intelligence and Instruction: Applications and Methods*, Seiten 19–25. Addison-Wesley, 1987.
- [SB82] SLEEMAN, D.H. und J.S. BROWN (Herausgeber): *Intelligent Tutoring Systems*. Academic Press, London, 1982.
- [SG93] STEELE, G. und J. GABRIEL: *The evolution of Lisp*. In: *The 2nd ACM SIGPLAN History of Programming Languages Conference (HOPL-II)*, Seiten 231–270, 1993.
- [Sha83] SHAPIRO, E. Y.: *Algorithmic Program Debugging*. MIT Press, Cambridge, MA, 1983.
- [SP94] SEIDEL, R.J. und O. PARK: *An historical perspective and a model for evaluation of intelligent tutoring systems*. *Journal of Educational Computing Research*, 10(2):103–126, 1994.
- [Ste90] STEELE, GUY L., JR.: *Common Lisp: The Language*. Digital Press, Bedford, MA, 1990.
- [Swe88] SWELLER, J.: *Cognitive load during problem solving: Effects on learning*. *Cognitive Science*, 12:257–286, 1988.
- [UL96] URBAN-LURAIN, MARK: *Intelligent Tutoring Systems: An Historic Review in the Context of the Development of Artificial Intelligence and Educational Psychology*, 1996.
- [Wen87] WENGER, ETIENNE: *Artificial Intelligence and Tutoring Systems*. Morgan Kaufmann, Los Altos, 1987.
- [WS97] WEBER, G. und M. SPECHT: *User modeling and adaptive navigation support in WWW-based tutoring systems*. In: *Proceedings of User Modeling '97*, Seiten 289–300, 1997.
- [WWW86] WALOSZEK, G., G. WEBER und K.-F. WENDER: *Entwicklung eines intelligenten LISP-Tutors*. Technischer Bericht 2, Institut für Psychologie, TU Braunschweig, 1986.