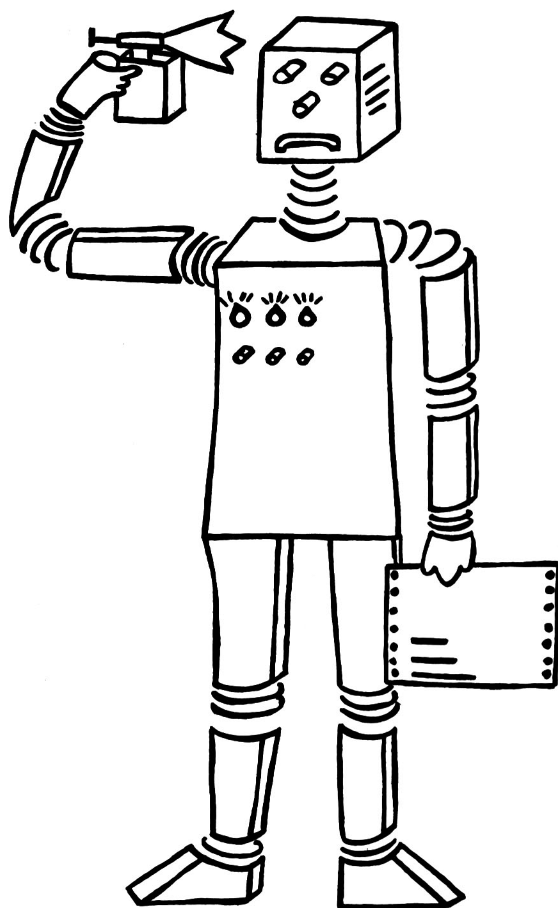
Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern 1, W. Germany

SEKI-Working Paper



Entwurf und Implementierung einer Interaktionsumgebung für LISPLOG

Manfred Meyer

SEKI Working-Paper SWP-87-02

Entwurf und Implementierung einer Interaktionsumgebung fuer LISPLOG

SEKI Working Paper

SWP-87-02

Januar 1987

(2. erweiterte Auflage: April 1988)

Manfred Andreas Meyer

Sonderforschungsbereich 314
Fachbereich Informatik
Universitaet Kaiserslautern
D-6750 Kaiserslautern
West-Germany

Mail-Adresse:

unido!uklirb!meyer
- oder -
meyer@uklirb.UUCP

Diese Arbeit ist teilweise im Sonderforschungsbereich 314, "Kuenstliche Intelligenz - Wissensbasierte Systeme", in Kaiserslautern entstanden.

Abstract

This paper deals with the design and implementation of an interactive programming environment for an integration of LISP and PROLOG called LISPLOG, which is currently developed at the Universitaet Kaiserslautern.

This programming environment consists of a box model tracer with backtrace possibilities and a very comfortable break-package.

Two innovative interactive tools for non-deterministic languages that have been explored in LISPLOG are a "cut-indicator" and a "manual cutter".

In addition, an extension of the box model for the functional part of LISPLOG is discussed and prototypically implemented.

Finally, several improvements of the user interface are roughly outlined, and the current implementation is critically reviewed once again.

Zusammenfassung

Diese Arbeit behandelt den Entwurf und die Implementierung einer Interaktionsumgebung fuer eine LISP/PROLOG-Vereinheitlichung, die unter der Bezeichnung LISPLOG an der Universitaet Kaiserslautern entwickelt wird.

Diese Umgebung besteht derzeit aus einem auf dem Box-Modell von Byrd basierenden Tracer mit Backtrace-Moeglichkeit und Stepper sowie einem umfangreichen Break-Paket zur interaktiven Unterstuetzung des Benutzers beim Debugging.

Zu den innovativen interaktiven Werkzeugen fuer nicht-deterministische Sprachen, die in LISPLOG erprobt worden sind, gehoeren ein "Schnitt-Anzeiger" und ein "Hand-Schneider".

Ausserdem werden eine Erweiterung des Box-Modells auch auf den funktionalen Teil von LISPLOG diskutiert und eine prototypische Implementation beschrieben.

Abschliessend werden Ansaetze fuer weitere Verbesserungen der Benutzeroberflaeche grob skizziert und die vorliegende Implementierung insgesamt einer kritischen Betrachtung unterzogen.

Inhalt

1. Allgemeines zum LISPLOG-Projekt 3

2. LISPLOG als Software-Entwicklungsumgebung 4

3. Anforderungen an eine Interaktionsumgebung 5

4. Auswahl eines geeigneten Beschreibungsmodells 7

5. Das Box-Modell 8
5.1. PROLOG - eine Sprache ohne explizite Kontrollstrukturen .. 8
5.2. Abbildung des Kontrollflusses im Box-Modell 10
5.3. Ein Beispiel mit Froeschen und Prinzen 13

6. Der LISPLOG-Interpreter 21
6.1. Die Kernfunktionen and-process und or-process 21
6.2. Extraktion der relevanten Box-Modell-Informationen 23
6.3. Implementation des Box-Modells: CALL und FAIL 26
6.4. Erweiterungen am Kern des LISPLOG-Interpreters 29
Exkurs: Das Resolutionsverfahren 29
6.5. Implementation des Box-Modells: EXIT und REDO 35

7. Die LISPLOG-Interaktionsumgebung 39
7.1. Die Schnittstelle zum LISPLOG-Interpreter 39
7.2. Konservierung der Ablaufinformationen fuer Backtrace 41
7.3. Selektion der auszugebenden Informationen (Debugger) 43
7.4. Moeglichkeiten zur Programmunterbrechung 46
7.5. Die Kommandoebene LISPLOG-Break-Level 52
7.6. Ausblenden von Teilauswertungen (Skipper) 59
7.7. Blaettern im Trace: Stepper und Backtrace 61
7.8. Cut-Anzeiger und Hand-Schneider 66

8. Erweiterung des Box-Modells fuer Funktionen 76
8.1. Ein Box-Modell fuer Funktionen 76
8.2. LISP-Durchgriffe als PROLOG-Goals 77
8.3. LISP-Ausdruecke auf der rechten Seite vom is-Operator ... 78
8.4. Integration in die bestehende Interaktionsumgebung 79

9. Modularisierung der Interaktionsumgebung 86
9.1. Auslagern des Trace-Pakets 86
9.2. Auslagern des Break-Pakets 90

10. Integration mit anderen LISPLOG-Interpretern 92

11. Ausblick: Erweiterungen und Verbesserungen 93

12. Literatur 103

Anhang A: LISPLOG-Interaktionsumgebung auf einen Blick 108
Anhang B: Listings 109

1. Allgemeines zum LISPLOG-Projekt

Die dieser Arbeit zugrundeliegende LISP/PROLOG-Vereinheitlichung (LISPLOG) soll die Paradigmen der funktionalen (LISP) und der logischen Programmierung (PROLOG) zusammenfuehren, um so ein integriertes Werkzeug fuer die Software-Entwicklung im Bereich der Kuenstlichen Intelligenz bereitzustellen.

Dazu wurde ein sehr kompakter Interpreter fuer pures PROLOG in purem LISP (vgl. [Kahn 1983/84]) in FRANZ LISP implementiert, weiter minimiert und leicht verbessert (LISPLOG.0, vgl. [Kammermeier 1985]). Dieser Interpreter wurde anschliessend in Richtung einer echten Integration von LISP und PROLOG erweitert:

Neben der Zurueckgabe der ersten n PROLOG-Loesungen nach LISP (Integration von PROLOG in LISP) wurde auch die Verwendung von (geschachtelten) LISP-Praedikaten als PROLOG-Goals ermoeglicht und ein verallgemeinerter is-Operator mit allgemeinen LISP-Ausdruecken auf der rechten Seite eingefuehrt (Integration von LISP in PROLOG).
Ausserdem wurden weitere PROLOG-Primitive (not, var, nonvar) und ein "initialer" cut-Operator eingefuehrt sowie die Datenbasis zur Effizienzsteigerung nach dem Praedikatnamen indexiert.
Die so entstandene Version LISPLOG.1, die in [Boley & Kammermeier et. al. 1985] ausfuehrlich dokumentiert ist, stellt die Grundlage dar, fuer die eine Interaktionsumgebung zu entwickeln ist. [1]

In weiteren Arbeiten wurde neben der Einfuehrung eines sekundaeren Datenbasis-Indexierungsverfahrens mit freier Wahl der Indexierungsstelle (vgl. [Bernardi 1986]) und der Bereitstellung eines Modulsystems fuer LISPLOG (vgl. [Dahmen 1986]) vor allem aus Effizienz- und Kompatibilitaetsgruenden die Moeglichkeit geschaffen, die in LISPLOG entwickelten funktional/logischen Programme (mit gewissen Einschraenkungen) nach CPROLOG oder (teilweise) nach LISP zu uebersetzen, um so durch Nutzung der entsprechenden effizienteren Laufzeitumgebung bessere Ausfuehrungszeiten fuer die Produktionsversionen zu erreichen.

---

[1] Parallel zu den vorgenommenen konzeptuellen und effizienzsteigernden Erweiterungen wurde bereits eine erste Version einer Interaktionsumgebung implementiert, die ebenfalls in [Boley & Kammermeier et. al. 1985] beschrieben wird. Die Schnittstelle zwischen dem Kern des LISPLOG-Interpreters und der Interaktionsumgebung sowie die prinzipielle Organisation der notwendigen Datenstrukturen konnten spaeter jedoch stark vereinfacht werden, so dass es sinnvoll erscheint, fuer eine schrittweise und vollstaendige Darstellung der Interaktionsumgebung auf einer LISPLOG.1-Version ohne jegliche Interaktionsmoeglichkeiten aufzubauen.

Dazu wurden ein Verfahren zur Klauselcompilation (vgl. [Herr 1985]), ein Uebersetzer von LISPLOG nach CPROLOG (vgl. [Hinkelmann & Morgenstern 1985] bzw. [Hinkelmann 1986]) sowie das Gegenstueck von CPROLOG nach LISPLOG (vgl. [Dahmen 1985]) entwickelt.
Diese Uebersetzer koennen natuerlich nur eine Teilmenge der entsprechenden Quellsprache semantikerhaltend in ihre Zielsprache abbilden. Diese Teilmenge ist allerdings ausreichend, um auch komplexere Anwendungen in LISPLOG entwickeln und bei Bedarf nach CPROLOG portieren zu koennen, was durch die Erstellung eines Modell-Expertensystems (micro-UNIXPERT) zur Diagnose von Druckproblemen in einer UNIX-Umgebung (vgl. [Lessel 1986] bzw. [Lessel & Boley 1987]) gezeigt werden konnte.

2. LISPLOG als Software-Entwicklungs-Umgebung

Wie bereits dargestellt, soll LISPLOG als Werkzeug zur Software-Entwicklung im KI-Bereich eingesetzt werden. Das heisst, dass LISPLOG primaer zur Entwicklung von KI-Anwendungen und weniger fuer den spaeteren routinemaessigen Einsatz von bereits entwikkelten Systemen verwendet werden soll.
Die Interaktionsumgebung soll also den KI-Programmierer bei der Entwicklung von funktional/logischen Programmen (LISPLOG-Programmen) unterstuetzen, die sich von der Phase der ersten prototypischen Implementation einiger Funktionen bzw. Praedikate ("Rapid Protoyping") bis hin zur eventuellen Fehlersuche und Fehlerkorrektur (Debugging) erstreckt.

Durch die angesprochene Moeglichkeit zur Trennung von Entwicklungsumgebung (LISPLOG) und Produktionsumgebung (LISP bzw. CPROLOG) wird deutlich, dass speziell fuer die Entwicklungsphase eine komfortable interaktive Unterstuetzung des Programmierers notwendig ist. Der Einsatz des fertigen Programms kann dann in einer in diesem Sinn weniger komfortablen Umgebung geschehen, da auf eine Interaktion zwischen Laufzeitsystem und Anwender im allgemeinen verzichtet werden kann bzw. diese bewusst nicht angeboten werden soll.
Selbstverstaendlich kann auch LISPLOG selbst als Produktionsumgebung eingesetzt werden, wobei man allerdings unabhaengig von der angebotenen Interaktionsmoeglichkeit, die natuerlich auch die Laufzeiteffizienz negativ beeinflusst, bereits fuer die Zweisprachigkeit (PROLOG und LISP) einen mehr oder weniger grossen Effizienzverlust in Kauf nehmen muss, der auch davon abhaengt, wie gross der PROLOG-Anteil an dem konkreten LISPLOG-Programm ist. Daran aendert grundsaetzlich auch die Tatsache nichts, dass mit dem in [Dahmen 1987] beschriebenen iterativen LISPLOG-Interpreter inzwischen eine wesentlich effizientere LISPLOG-Version (LISPLOG.2) zur Verfuegung steht.

Dennoch sollen natuerlich auch Effizienzgesichtspunkte beim Entwurf und der Implementation der Interaktionsumgebung beruecksichtigt werden.

3. Anforderungen an eine Interaktionsumgebung

Besonders im Bereich der Kuenstlichen Intelligenz werden haeufig Theorien, die in Teilen noch weiterentwickelt werden, durch Programme dargestellt bzw. ausgetestet ("Experimentelles Programmieren"). Auch groessere KI-Systeme werden haeufig im Kern zunaechst nur prototypisch realisiert, um verschiedene Implementationsmoeglichkeiten auszuprobieren. Dabei werden dann sowohl zur Darstellung von Testergebnissen als auch beim Debugging oftmals spezielle Ausgabeanweisungen in das Programm eingefuegt. Dieses Vorgehen kann natuerlich vor allem wegen des damit verbundenen Aufwandes nicht befriedigen, so dass es bald sinnvoller erscheint, eine allgemeine problemunabhaengige Programmierumgebung zu entwickeln, die folgende Werkzeuge zur Verfuegung stellen sollte:

(1) Tracer und Debugger:

Der Tracer soll in einfacher Form Informationen ueber den Programmablauf ausgeben. Ein wichtiges Entwurfsziel ist dabei die leichte Erlernbarkeit: Der Tracer soll ein einfaches Hilfsmittel sein [2]. Das hat jedoch zunaechst den Nachteil, dass teilweise auch Informationen ausgegeben werden, die in der konkreten Situation nicht von Interesse sind. Fuer komplexere Anwendungen benoetigt man daher ein Instrument, mit dem man gezielt Informationen ueber den Programmablauf ermitteln kann: einen Debugger.
Dieser soll dem Benutzer so wenig wie moeglich - aber natuerlich auch so viel wie noetig - Informationen ueber den Programmablauf ausgeben, um ihn so beim Debuggig effektiv zu unterstuetzen [3].

In unserem System ist der Uebergang zwischen Tracer und Debugger fliessend: Zwar liefert der Tracer - wenn er aktiviert ist - immer ein vollstaendiges Protokoll des Programmablaufs [4]. Der Benutzer hat jedoch durch einfache Kommandos die Moeglichkeit, die Teilmenge der tatsaechlich auszugebenden Informationen flexibel einzuschraenken und auch wieder auszuweiten.

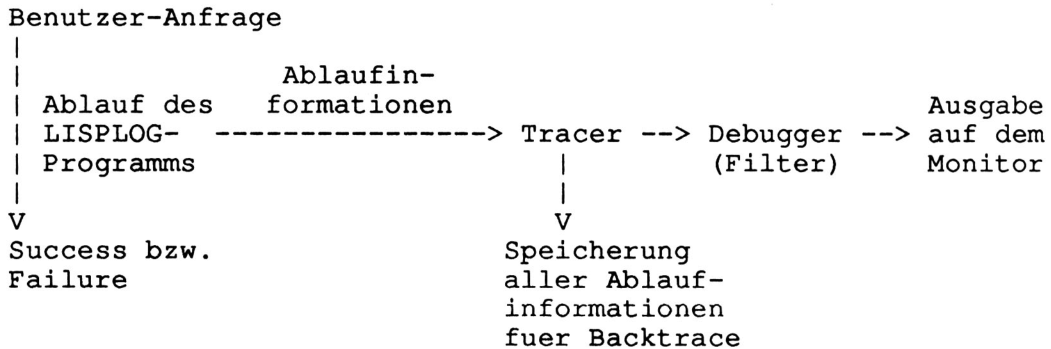
---

[2] In der hier beschriebenen Interaktionsumgebung ist der Umgang mit dem Tracer zunaechst denkbar einfach: Das Kommando "spy" schaltet den Tracer ein, das Kommando "nospy" schaltet ihn wieder aus.

[3] Er soll also "maximal informativ" sein. Vgl. [Grice 1967].

[4] "vollstaendig" natuerlich bezueglich des zugrundeliegenden Modells (s. Kapitel 4).

Im Prinzip ist der Debugger also ein vom Benutzer steuerbarer Filter fuer die vom Tracer zur Verfuegung gestellten Ablaufinformationen, wie die folgende Skizze des Informationsflusses zeigt:



(2) Eingriffsmoeglichkeiten in den Programmablauf (Break-Paket):

Zu diesen Eingriffsmoeglichkeiten gehoeren z.B. die Ausgabe von Informationen ueber den aktuellen Programmzustand (aktuelle Bindungen, noch zu beweisende (Teil-)Ziele, gerade bearbeitetes (Teil-)Ziel), aktueller Pfad im Suchbaum), die Aenderung von Daten und Programmteilen nach Fehlern, die Veraenderung des oben angesprochenen Filters (Debugger) sowie das (Zurueck-)Blaettern im Trace auch mit jederzeit aenderbarem Informationsfilter (Backtrace).
All diese Eingriffe sollen in einer besonderen Kommandoebene (Break-Level) erfolgen koennen, die ausserdem natuerlich auch alle anderen Kommandos der Benutzerschnittstelle (Verwaltung der Datenbasis, Kommandos des Modulsystems, Steuerung der Interaktionsumgebung usw.) zur Verfuegung stellt. Diese (Break-)Kommandoebene soll dann sowohl mittels spezieller Breakpoints, die ebenfalls sehr einfach zu verwalten sein muessen [5], als auch durch jederzeit moegliche Programmunterbrechung (Interrupt) aktiviert werden koennen.

Diese zweite Moeglichkeit der direkten Programmunterbrechung ist besonders wichtig fuer Programmlaeufe, bei denen man zunaechst auf die Ausgabe von Ablaufinformationen ganz verzichten, bei Bedarf aber ueber alle Moeglichkeiten der Interaktionsumgebung, auch ueber die Ausgabe aller Ablaufinformationen mittels Backtrace, verfuegen moechte.

---

[5] Analog zu den Kommandos "spy" und "nospy" gibt es hierzu die Kommandos "brk" und "nobrk", mit denen Breakpoints gesetzt bzw. geloescht werden koennen.

4. Auswahl eines geeigneten Beschreibungsmodells

Wie im letzten Abschnitt dargestellt, soll also durch die zu entwickelnde Interaktionsumgebung dem Benutzer die Moeglichkeit gegeben werden, die Abarbeitung seines LISPLOG-Programmms auch schrittweise (Stepper) gezielt mitverfolgen zu koennen (Tracer/Debugger) und ggfs. speziell beim Debugging den Ablauf auch unterbrechen und aktiv auf die Abarbeitung Einfluss nehmen zu koennen (Break-Moeglichkeit).

In welcher Weise aber soll der Benutzer die Abarbeitung des LISPLOG-Programms mitverfolgen koennen?

Dazu muessen waehrend des Programmablaufs geeignete Informationen ueber den aktuellen Stand der Abarbeitung des LISPLOG-Programms bereitgestellt werden, aus denen der Benutzer dann Hinweise ueber den korrekten oder fehlerhaften Programmablauf beziehen kann. Diese Bereitstellung von Informationen kann sicherlich auf verschiedenen Ebenen geschehen:

So waere es zum Beispiel denkbar, die Abarbeitung eines LISPLOG-Programms anhand der den LISPLOG-Interpreter realisierenden LISP-Funktionen, also durch einen LISP-Trace, darzustellen.
Der LISPLOG-Anwender erhaelt dadurch sicherlich umfassende Informationen ueber die Abarbeitung seines Programms, vorausgesetzt allerdings, dass ihm die Implementation des LISPLOG-Interpreters bis in die letzten Details bekannt ist. Abgesehen davon, dass diese Voraussetzung natuerlich voellig unrealistisch ist, wuerde der Anwender durch die Unmenge an Detailinformationen selbst bei sehr kleinen LISPLOG-Programmen sehr schnell den Ueberblick ueber den logischen Ablauf des eigentlichen LISPLOG-Programms verlieren.

Das Problem liegt also darin, dass ein Trace der implementierenden LISP-Funktionen eben kein geeignetes Beschreibungsmodell fuer den Ablauf eines LISPLOG-Programms darstellt. Dieses Modell ist zwar geeignet, die Abarbeitung des funktional codierten Teils des LISPLOG-Programms (LISP-Funktionen) angemessen darzustellen. Es ist aber ungeeignet zur Beschreibung der Abarbeitung und damit der operationalen Semantik des in Klauselform codierten Teils, da bei der Abarbeitung durch die Korekursionen der Interpretationsfunktionen and- und or-process die eigentlich n-aeren Verzweigungen des Und/Oder-Baums nunmehr in binaerer Form repraesentiert werden (vgl. Kapitel 6).

Die grundsaetzliche Voraussetzung fuer den Entwurf und die Implementierung einer Interaktionsumgebung fuer LISPLOG besteht also in der Wahl eines geeigneten Modells zur Beschreibung der operationalen Semantik von LISPLOG-Programmen. Dieses Modell muss sowohl den logischen als auch den funktionalen Teil eines LISPLOG-Programms angemessen repraesentieren.

Fuer die hier nun zu entwickelnde Interaktionsumgebung verwenden wir das von L. Byrd eingefuehrte Box-Modell (vgl. [Bowen, Byrd, Pereira, Pereira & Warren 1981] oder [Clocksin & Mellish 1984]), das auch in anderen PROLOG-Systemen, wie z.B. IF-PROLOG (vgl. [IF-PROLOG 1984]), als Beschreibungsmodell verwendet wird. Es erlaubt zunaechst jedoch nur eine Ablauf-Darstellung des in Klauselform repraesentierten Teils eines LISPLOG-Programms, also nur eine partielle Abbildung der operationalen LISPLOG-Semantik. Um auch fuer LISPLOG als grundlegendes Beschreibungsmodell fuer eine darauf aufbauende Interaktionsumgebung dienen zu koennen, wird es spaeter auch fuer den funktionalen Teil erweitert werden.

Zunaechst werden wir uns aber beim weiteren Entwurf der Interaktionsumgebung auf das im folgenden Abschnitt noch naeher beschriebene Box-Modell von Byrd beschraenken. Erst spaeter werden wir dann einhergehend mit der Ausdehnung des Box-Modells auf den funktionalen Teil von LISPLOG auch die Interaktionsumgebung entsprechend erweitern - ein Vorgehen, das einerseits auch dem tatsaechlichen Vorgehen bei der Implementierung entspricht, andererseits aber auch durch die natuerliche schrittweise Darstellung das Verstaendnis der Implementation erleichtern soll.

5. Das Box-Modell

Das Box-Modell von L. Byrd dient, wie gesagt, zur dynamischen Beschreibung der operationalen Semantik von PROLOG.

Bevor wir dieses Modell im Detail beschreiben, wollen wir noch kurz darstellen, welche speziellen Anforderungen dieses Modell erfuellen muss, die sich aus dem besonderen Charakter von PROLOG als stark deklarativer Programmiersprache ergeben.

5.1. PROLOG - eine Sprache ohne explizite Kontrollstrukturen

Besonderes Merkmal von PROLOG ist, dass diese KI-Sprache fast ohne Kontrollstrukturen auskommt. Abgesehen von built-in-Praedikaten (hier speziell dem Cut-Operator, der in LISPLOG jedoch auf sein initiales Auftreten beschraenkt bleibt [6]), beruht PROLOG ausschliesslich auf Unifikation und Backtracking.

Die Ausfuehrung eines PROLOG-Programms besteht somit im Prinzip nur aus dem Unifizieren von Klauseln und dem Durchlaufen des damit aufgebauten Suchbaums.

---

[6] Zur Diskussion des "initialen" Cut-Operators sei hier auf [Boley & Kammermeier et. al. 1985] verwiesen.

So kann man die Klauseln fuer ein einfaches Praedikat "delete" fuer Zahlenlisten in beliebiger Reihenfolge schreiben, ohne etwa Kontrollstruktuen wie IF-THEN-ELSE zu verwenden: [7], [8]

```
(1)   (ass (delete _x nil nil))
      (ass (delete _x (_x . _l) _m) (delete _x _l _m))
      (ass (delete _x (_y . _l) (_y . _m)) (lessp _x _y)
                                           (delete _x _l _m))
      (ass (delete _x (_y . _l) (_y . _m)) (greaterp _x _y)
                                           (delete _x _l _m))
```

Dazu kommt ein einfaches Variablenkonzept: Es gibt in PROLOG keine globalen Variablen (Gueltigkeitsbereich ist immer die Klausel, in der die Variable auftritt) und grundsaetzlich nur "single-assignment"-Variablen, d.h. keine Moeglichkeit, den Wert bereits belegter ("gebundener") Variablen zu ueberschreiben, wie das in herkoemmlichen Programmiersprachen ueblich ist.

Diese Sprachstruktur bietet eine Reihe von Vorteilen. PROLOG basiert direkt auf der Praedikatenlogik (in Form von Hornklauseln) und laesst sich deshalb auch als (ausfuehrbare) Spezifikation verwenden. Daten und Programme werden gemeinsam als Praedikate dargestellt: Vom Programm erzeugte Daten koennen selbst als Teil des Programms ausgefuehrt werden.

---

[7] Die Syntax von LISPLOG, die sich aufgrund der Wahl von LISP als Implementierungssprache an der LISP-Syntax orientiert, weist einige Unterschiede zu der anderer PROLOG-Implementationen (z.B. CPROLOG) auf, die hier zum Verstaendnis kurz zusammengefasst werden:
Variable werden in LISPLOG durch ein vorangestelltes "_" gekennzeichnet. Gross-/Kleinschreibung ist nicht signifikant. Strukturen werden in der LISP-typischen Cambridge-Polnischen-Praefix-Notation dargestellt. Statt "likes(mary,john)" verwendet LISPLOG die Notation "(likes mary john)". Datenbasen sind Listen von Klauseln, die entweder Regeln oder Fakten sind. In LISPLOG werden Fakten als Regeln ohne Praemissen dargestellt. Ausserdem koennen Klauseln insgesamt mit einem initialen Cut (vorangestelltes "!" vor der Konklusion) versehen werden:

```
<klausel>    ::= (ass <konklusion> <praemisse>*)
<konklusion> ::= <struktur> | !<struktur>
<praemisse>  ::= <struktur>
```

[8] Die Wahl der Praedikatnamen "lessp" bzw. "greaterp" deutet bereits an, dass hier ein LISP-Durchgriff stattfindet. Ausser den PROLOG-Primitiven "var", "nonvar", "is" und "not" werden alle anderen Primitive mittels Durchgriffen auf (primitive) LISP-Funktionen realisiert.

Das hat jedoch fuer den ungeuebten Leser den Nachteil, dass der Kontrollfluss aus dem Programm selbst schwer erkennbar ist. So haben z.B. die schon vorgestellte Version (1) und die folgende Version (2) von "delete" denselben Effekt:

```
(2)   (ass (delete _x nil nil))
      (ass !(delete _x (_x . _l) _m) (delete _x _l _m))
      (ass (delete _x (_y . _l) (_y . _m)) (delete _x _l _m))
```

Die letzte Klausel in Version (2) gewinnt ihre Bedeutung aber nur aufgrund des vorausgegangenen Cuts in der zweiten Klausel. Diese Klauselmenge ist kein reines Axiomensystem mehr: die Reihenfolge der Klauseln und damit der Kontrollfluss muessen mitberuecksichtigt werden.

5.2. Abbildung des Kontrollflusses im Box-Modell

Die Schwierigkeit bei einer Betrachtung der statischen Datenbasis besteht im Wesentlichen darin, die Wirkung von Failure und Backtracking zu erkennen:

Das Praedikat "delete" verwendet kein IF-THEN-ELSE (obwohl sich auch das leicht in PROLOG formulieren laesst), sondern es wird in drei bzw. vier Klauseln aufgeteilt. Die nicht explizit gemachte Kontrollstruktur ergibt sich erst durch das Prinzip des Backtracking bei Failure von Teilzielen.

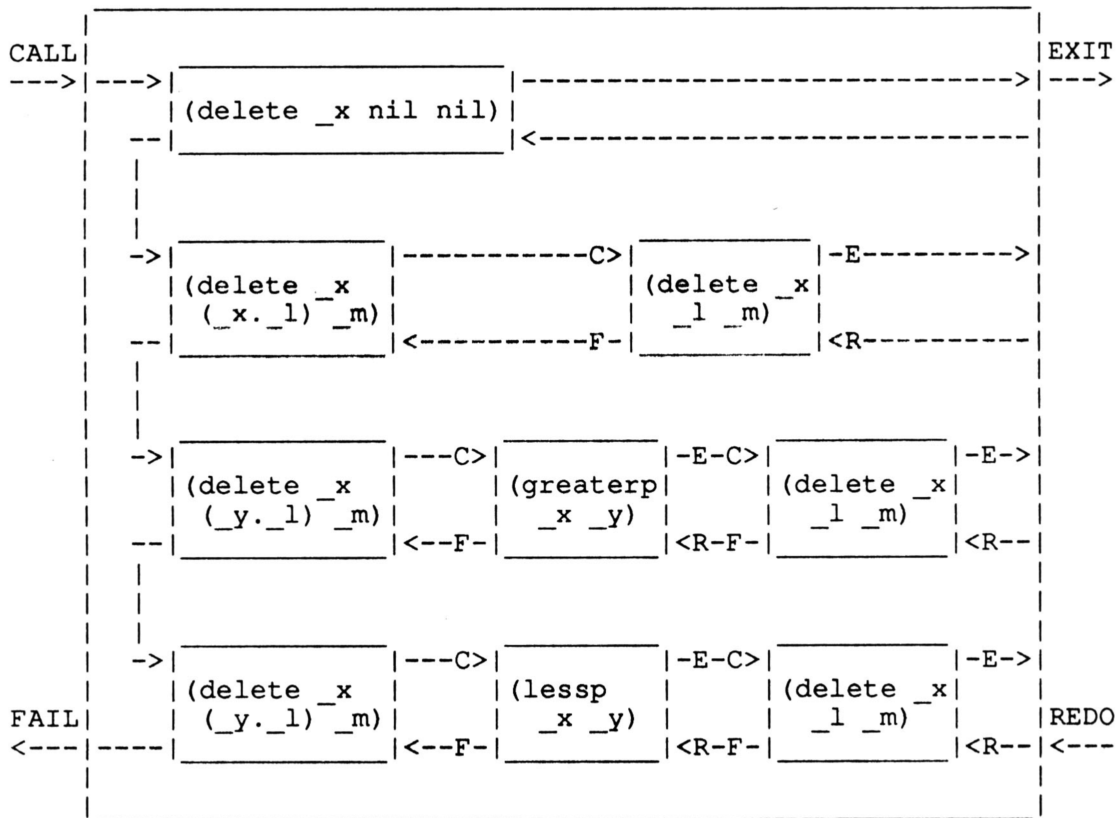
Das Beschreibungsmodell muss deshalb Moeglichkeiten bereitstellen, diese elementaren Ablaufstrukturen angemessen darzustellen.

Dazu wird die Menge aller Klauseln eines Praedikates zu einer sogenannten "Prozedur" zusammengefasst. Auf dieser Ebene setzt das in [Bowen, Byrd, Pereira, Pereira & Warren 1981] eingefuehrte Box-Modell nun an, indem es eine sog. "Procedure-Box" als Kontrollflussmodell einfuehrt.

Eine solche Procedure-Box (hier im folgenden "Box" genannt) wird als Kasten mit zwei Eingaengen (CALL und REDO) und zwei Ausgaengen (EXIT und FAIL) dargestellt. Die einzelnen Klauseln in einer Procedure-Box sind dann ihrerseits wieder aus aehnlichen Boxes aufgebaut. Fuer das Beispiel "delete"

```
(1)   (ass (delete _x nil nil))
      (ass (delete _x (_x . _l) _m) (delete _x _l _m))
      (ass (delete _x (_y . _l) (_y . _m)) (lessp _x _y)
                                           (delete _x _l _m))
      (ass (delete _x (_y . _l) (_y . _m)) (greaterp _x _y)
                                           (delete _x _l _m))
```

erhaelt man z.B. fuer eine beliebige Anfrage die folgende Box bestehend aus vier Klauseln (einem Faktum und drei Regeln):



C,R = CALL- bzw. REDO-Eingaenge einer Box (Praemisse)
E,F = EXIT- bzw. FAIL-Ausgaenge einer Box (Praemisse)

Wird nun eine Anfrage an den LISPLOG-Interpreter gestellt, so wird also das entsprechende Praedikat aufgerufen, d.h. die zugehoerige Procedure-Box durch den CALL-Eingang betreten.

Es wird dann versucht, eine "passende" Klausel, konkret also einen mit der Anfrage unifizierbaren Klauselkopf (Faktum oder Konklusion einer Regel) zu finden. Gelingt dies nicht, so wird die ganze Box durch den FAIL-Ausgang verlassen; die Anfrage konnte nicht verifiziert werden.
Gibt es aber eine "passende" Klausel, so wird nun die Box der ersten Praemisse betreten bzw. bei fehlenden Praemissen (bei Fakten) die Box des ganzen Praedikats durch den EXIT-Ausgang verlassen; die Anfrage konnte also verifiziert werden.

Konnte eine Praemisse durch den EXIT-Ausgang verlassen und damit erfolgreich abgeschlossen werden, so wird anschliessend die naechste Praemisse durch den CALL-Eingang betreten.
Wenn alle Praemissen so durch ihren EXIT-Ausgang verlassen werden konnten, wird auch die ganze Box des Praedikats durch den EXIT-Ausgang verlassen.

Scheitert aber eine Praemisse, d.h. wird eine Praemisse durch den FAIL-Ausgang verlassen, so wird die unmittelbar letzte zuvor noch erfolgreich durchlaufene und durch den EXIT-Ausgang verlassene Praemissen-Box nochmals betreten, jetzt allerdings durch den REDO-Eingang (Backtracking).
Scheitert so schliesslich auch die erste Praemisse einer Klausel, d.h. wird die erste Praemissen-Box einer Klausel einmal durch den FAIL-Ausgang verlassen, so wird erneut versucht, aus der noch verbleibenden Klauselmenge des Praedikats eine "passende" Klausel zu finden und deren Praemissen zu beweisen.

Das allerdings nur, wenn die zuvor gescheiterte Klausel nicht mit einem "initialen" Cut versehen war, der ja gerade aussagt, dass es zu der betreffenden Klausel, wenn sie einmal als "passend" ermittelt wurde, keine Alternativen mehr gibt: Alle nachfolgenden Klauseln werden an dieser Stelle "abgeschnitten" (cut!).
Gibt es dann schliesslich keine "passende" Klausel mehr, so wird die ganze Box durch den FAIL-Ausgang verlassen.

Beim Wiederbetreten einer Box durch den REDO-Eingang werden alle unmittelbar zuvor durch den EXIT-Ausgang verlassenen Procedure-Boxes erneut durch den REDO-Eingang betreten. So gelangt der Kontrollfluss schliesslich zurueck zu dem letzten Faktum, mit dem unifiziert wurde.
Hier wird nun versucht, aus den auf dieses Faktum folgenden Klauseln in der entsprechenden Box erneut eine "passende" Klausel zu finden, und die Abarbeitung so fortgesetzt, als waere dieses Faktum selbst fehlgeschlagen. Der Prozess des Backtracking, also des Zuruecksetzens zum letzten Auswahlpunkt ("choice point"), wo zuletzt eine Auswahl einer "passenden" Klausel vorgenommen wurde, wird also im Box-Modell durch das sukzessive Wiederbetreten (REDO) aller Procedure-Boxes repraesentiert, die aufgrund der letzten Unifikation (notwendigerweise mit einem Faktum!) unmittelbar durch den EXIT-Ausgang verlassen werden konnten.

Mit diesen Procedure-Boxes wird nun ein dynamisches Modell aufgebaut: Bei rekursiven Praedikaten beispielsweise, wie dem als Beispiel benutzten Praedikat "delete", wird fuer jede Rekursionsstufe eine neue Box erzeugt.
Das bedeutet, dass das Box-Modell den Ablauf des Beweises einer Anfrage wiedergibt und somit keine neue Repraesentation der (statischen) Datenbasis ist. Eine konkrete Auspraegung des Box-Modells kann es nur zu einer konkreten Datenbasis und einer konkreten Anfrage geben.

Somit ist es dann die Aufgabe des Tracers, bei der Abarbeitung einer konkreten Anfrage jeden Durchgang durch einen Ein- oder Ausgang im Box-Modell zusammen mit dem gerade bearbeiteten (Teil-)Ziel mit instanziierten Variablen zu protokollieren.
Ausserdem sollten dabei die Procedure-Boxes von Praedikaten auf verschiedenen Aufrufebenen unterscheidbar sein, was wir durch eine der Aufrufebene entsprechende Einrueckung der jeweils auszugebenden Trace-Informationen erreichen.

5.3. Ein Beispiel mit Froeschen und Prinzen

Das folgende Beispiel veranschaulicht die Abbildung des Kontrollflusses im Box-Modell bei der Abarbeitung eines konkreten LISPLOG-Programms.
Dazu sei folgende Datenbasis gegeben:

```
(3)   (ass (prinz _x) (frosch _x) (verzaubert _x))
      (ass (prinz charles))
      (ass (frosch kermit))
      (ass (verzaubert kermit))
      (ass (erstgeboren charles))
      (ass (thronfolger _x) (prinz _x) (erstgeboren _x))
```

Wird nun nach dem Thronfolger gesucht, also die Anfrage

(thronfolger _gesuchte-person)

gestellt, so ergibt sich der folgende Programmablauf und die entsprechende Repraesentation im Box-Modell (fuer genau diese Anfrage bei dieser konkreten Datenbasis):

Zunaechst wird die Box "thronfolger" durch den CALL-Eingang betreten. Der Tracer sollte diese Tatsache durch Ausgabe von

"|CALL (thronfolger _gesuchte-person)"

protokollieren. Es wird dann eine "passende" Klausel zu dieser Anfrage gesucht und dann mit der Klausel "(ass (thronfolger _x) (prinz _x) (erstgeboren _x))" auch gefunden. Dabei entsteht die Bindung der Variablen _gesuchte-person an die Variable _x aus der "thronfolger"-Klausel, die wir mit "_gesuchte-person --> _x-1" notieren. [9]

Nach dieser erfolgreichen Unifikation kann nun im naechsten Schritt die Box der ersten Praemisse dieser Klausel betreten werden, was der Tracer mit

"| CALL (prinz _x-1)"

melden soll, wobei diese Ausgabe - der Aufrufebene entsprechend - gegenueber der vorherigen Ausgabe eine Spalte weiter eingerueckt stehen soll.

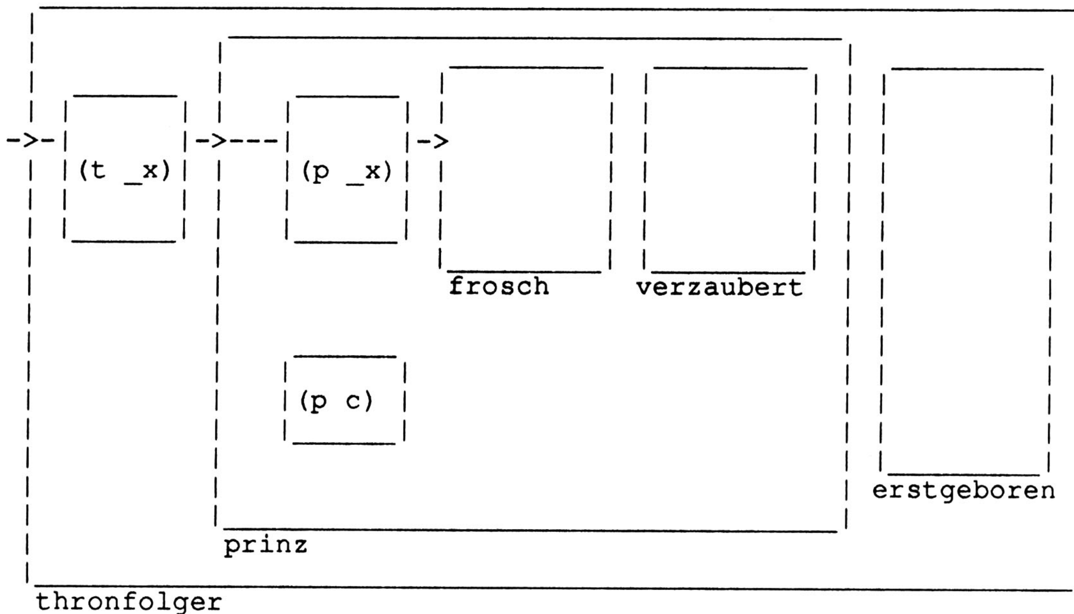
---

[9] Um Variablen gleichen Namens (hier: _x) auf verschiedenen Aufrufebenen unterscheiden zu koennen, werden sie bei der Auswahl einer Klausel aus der Datenbasis durch Anhaengen der Nummer der aktuellen Aufrufebene aufsteigend ab 1 bzw. 0 (bei der Benutzeranfrage) entsprechend umbenannt.

Auch in der Box "prinz" gibt es eine "passende" Klausel fuer das Teilziel "(prinz _x-1)", naemlich die Klausel "(ass (prinz _x) (frosch _x) (verzaubert _x))". Da diese Regel in der Datenbasis (3) vor dem Faktum "(ass (prinz charles))" steht, das auch mit dem Teilziel unifizierbar waere, wird zunaechst die Regel ausgewaehlt und erst spaeter bei Fehlschlagen der Regel eventuell auf das Faktum zurueckgegriffen (Backtracking).
Es wird also jetzt zunaechst die erste Praemisse der soeben ausgewaehlten Klausel durch den CALL-Eingang betreten, was der Tracer wiederum durch Ausgabe von

"| CALL (frosch _x-2)"

meldet. Wenn wir nun wieder die oben eingefuehrte graphische Darstellung heranziehen, so ergibt sich zu diesem Zeitpunkt der Abarbeitung die folgende Modelldarstellung: [10]



Zu der neuen Praemisse "(frosch _x-2)" gibt es auch tatsaechlich eine "passende" Klausel: das Faktum "(frosch kermit)".

---

[10] Dabei wurden folgende Abkuerzungen gebraucht, um die Darstellung einigermassen uebersichtlich zu halten:
Praedikate: t = thronfolger, p = prinz, f = frosch,
v = verzaubert, e = erstgeboren
Konstanten: c = charles, k = kermit

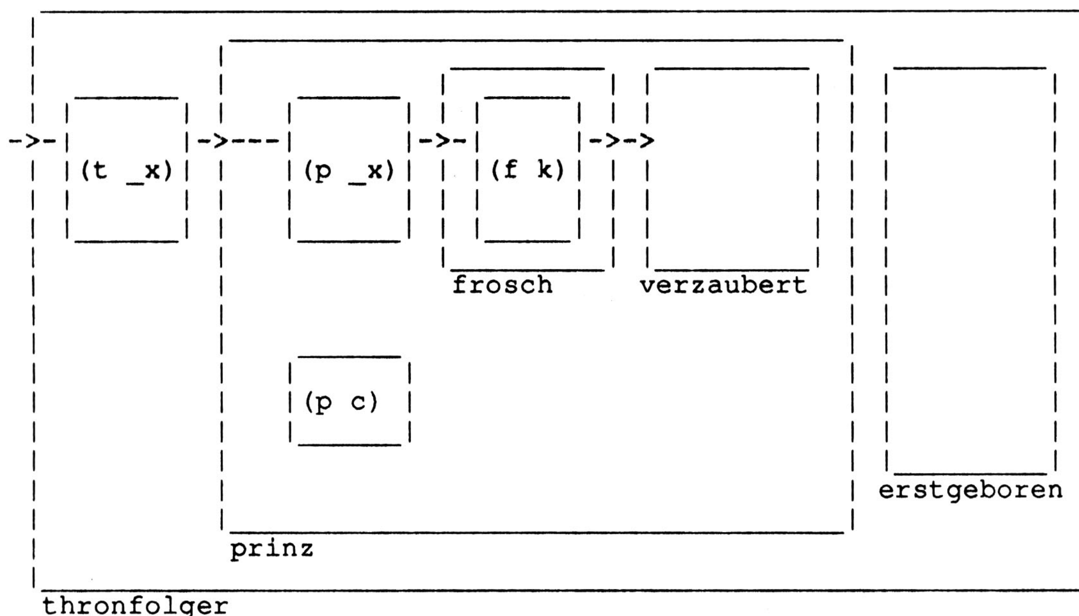
Damit entsteht zum ersten Mal eine "echte" Variablenbindung, waehrend zuvor nur immer eine Variable an eine andere gebunden wurde. Jetzt aber ist ueber die Zwischenbindungen "_gesuchte-person --> _x-1" und "_x-1 --> _x-2" die Anfragevariable _gesuchte-person an die Konstante kermit gebunden.
Damit ist der Beweis der Anfrage natuerlich noch nicht beendet; es stehen ja noch weitere Praemissen aus, die mit dieser Bindung zunaechst erfolgreich zu durchlaufen sind, bevor die Benutzeranfrage tatsaechlich mit dieser Bindung beantwortet werden kann.

Zumindest aber das Teilziel "(frosch _x-2)" ist erfolgreich, so dass die entsprechende Box durch den EXIT-Ausgang verlassen und die der naechsten Praemisse durch den CALL-Eingang betreten werden kann. Durch die eben entstandene Bindung heisst die naechste Praemisse nunmehr "(verzaubert kermit)", da in ihr die gleiche Variable _x-2 steht, die beim Beweis der ersten Praemisse an die Konstante kermit gebunden worden war.

Der Tracer soll auch diesen Vorgang entsprechend melden:

```
"|  EXIT (frosch kermit)"
"|  CALL (verzaubert kermit)"
```

In der graphischen Darstellung erhalten wir nun folgendes Bild:



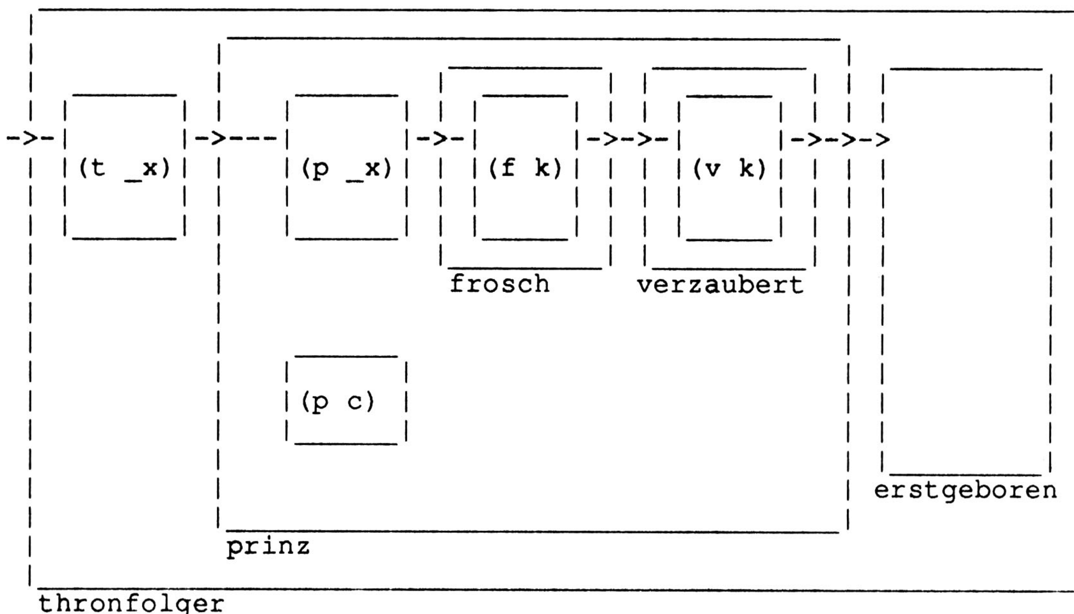
Auch diese zweite Praemisse "(verzaubert kermit)" kann bewiesen werden, da die Box des entsprechenden Praedikats eine "passende" Klausel enthaelt, naemlich gerade das Faktum "(verzaubert kermit)". Dabei entstehen natuerlich keine weiteren Bindungen.

Es wird also die Box "verzaubert" nun durch den EXIT-Ausgang verlassen. Da aber die gerade bewiesene Praemisse "(verzaubert kermit)" zugleich die letzte Praemisse der Regel "(ass (prinz _x) (frosch _x) (verzaubert _x))" ist, kann auch die Box "prinz" durch den EXIT-Ausgang verlassen werden.

Damit ist also auch die erste Praemisse der "thronfolger"-Regel bewiesen, und der Kontrollfluss gelangt jetzt zur zweiten Praemisse "(erstgeboren _x)", die also durch den CALL-Eingang betreten wird.
Der Tracer macht diese drei Schritte in der folgenden Weise

```
"|  EXIT (verzaubert kermit)"
"| EXIT (prinz kermit)"
"| CALL (erstgeboren kermit)"
```

deutlich, wobei man durch die unterschiedliche Einrueckung der Informationen leicht erkennen kann, dass durch den Beweis von "(verzaubert kermit)" auch das uebergeordnete Praedikat "(prinz _x-1)" mit der Instanziierung "_x-1 --> kermit" verlassen werden konnte.
Die folgende Abbildung zeigt dann eine graphische Darstellung zu dem bisherigen Beweisablauf:



Koennte nun auch das Praedikat "(erstgeboren kermit)" bewiesen werden, so waere damit auch die Benutzeranfrage verifiziert, fuer die Anfragevariable _gesuchte-person wuerde der Wert "kermit" ausgegeben werden.

In der Datenbasis und somit in der Box des Praedikats "erstgeboren" gibt es aber leider keine Klausel, die mit dem Teilziel "(erstgeboren kermit)" unifizierbar waere. Hier tritt also zum ersten Mal ein Failure auf; die "erstgeboren"-Box muss durch den FAIL-Ausgang verlassen werden, was den Tracer zur Ausgabe der folgenden Meldung veranlassen soll:

"| FAIL (erstgeboren kermit)"

An dieser Stelle der Abarbeitung hat sich nun herausgestellt, dass eine der zuvor als "passend" ausgewaehlten Klauseln einen falschen Weg im Suchbaum eingeschlagen hat: diese Auswahl muss rueckgaengig gemacht werden (Backtracking).
In LISPLOG geschieht dieses Backtracking ganz einfach dadurch, dass die zuletzt getroffene Auswahl, also die letzte erfolgreiche Unifikation, rueckgaengig gemacht wird. Dieses Vorgehen heisst auch "chronologisches Backtracking" im Gegensatz zu anderen "intelligenteren" Verfahren [11]. Im Abschnitt 5.2. haben wir bereits beschrieben, wie der Prozess des Backtracking im Box-Modell dargestellt wird: es werden alle unmittelbar zuvor noch erfolgreich verlassenen Boxes wieder durch den REDO-Eingang betreten bis der Kontrollfluss so bei der letzten zuvor erfolgreichen Unifikation ankommt. In diesem Fall war dies die Unifikation des Teilziels "(verzaubert kermit)" mit dem Faktum "(ass (verzaubert kermit))", wobei dabei allerdings ueberhaupt keine neuen Bindungen entstanden sind:

"| REDO (prinz kermit)"
"| REDO (verzaubert kermit)"

Diese Unifikation, also die Auswahl der entsprechenden Klausel, muss nun zurueckgenommen werden, und es muss in der zugehoerigen Box nach einer anderen "passenden" Klausel gesucht werden.

Leider gibt es in der "verzaubert"-Box keine weitere und deshalb auch keine neue "passende" Klausel, so dass auch diese Box durch den FAIL-Ausgang verlassen und die erste Praemisse "(frosch _x)" nunmehr durch den REDO-Eingang betreten wird:

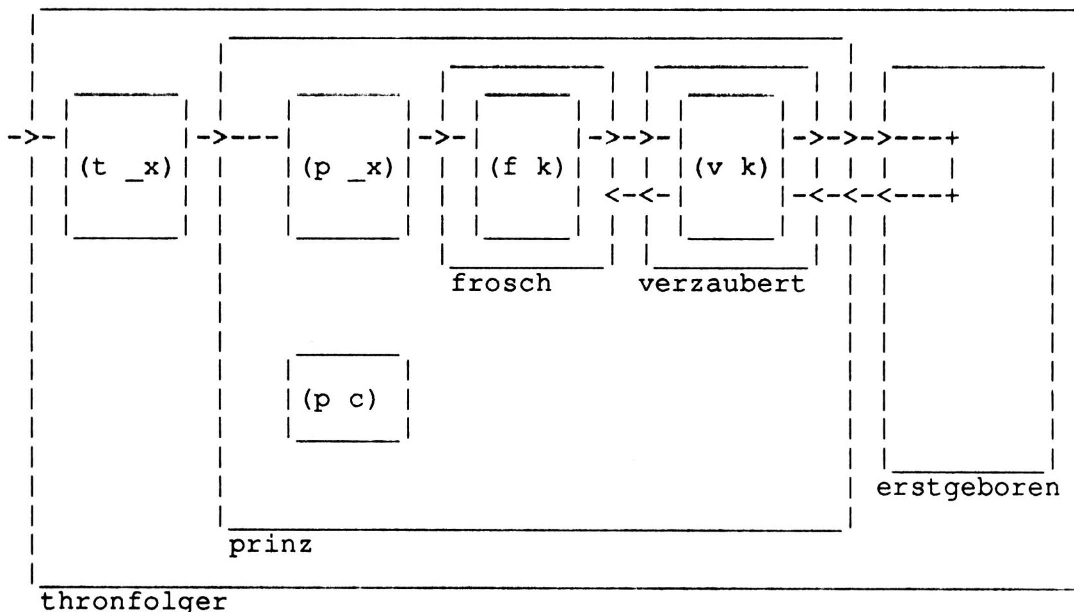
"| FAIL (verzaubert kermit)"
"| REDO (frosch _x-2)"

In der graphischen Darstellung sehen wir, dass im Zuge des Back-

---

[11] "Intelligente" Backtracking-Verfahren versuchen, gezielt die Auswahl (und natuerlich alle davon abhaengigen Schritte) zurueckzunehmen, die das spaetere Scheitern des Beweises (mit)verursacht haben ("dependency-directed backtracking"). Auch fuer die Einfuehrung eines solchen Verfahrens in LISPLOG wurden Vorarbeiten geleistet, ohne dass jedoch zum gegenwaertigen Zeitpunkt bereits eine prototypische Implementierung vorliegt.

tracking die zuvor erfolgreich (CALL --> EXIT) durchlaufenen Boxes jetzt gerade in umgekehrter Richtung (FAIL <-- REDO) betreten und verlassen werden:



Auch in der "frosch"-Box gibt es keine weitere Klausel fuer das zu beweisende Teilziel "(frosch _x-2)", so dass auch diese Praemisse durch ihren FAIL-Ausgang verlassen werden muss:

"| FAIL (frosch _x-2)"

Damit ist schliesslich die ganze Klausel "(ass (prinz _x) (frosch _x) (verzaubert _x))" gescheitert, denn "(frosch _x)" war die erste Praemisse dieser Klausel. Es muss nun also auch hier die bei der Unifikation mit dem Klauselkopf "(prinz _x-2)" entstandene Bindung "_x-1 --> _x-2" zurueckgenommen und nach einer anderen "passenden" Klausel gesucht werden.
Und tatsaechlich folgt auf die fehlgeschlagene noch eine weitere anwendbare Klausel: das Faktum "(prinz charles)". Da diese Klausel keine Praemissen hat, kann nun die Box des Praedikats "prinz" direkt durch den EXIT-Ausgang verlassen werden. Der Tracer zeigt dann auch an, dass in dieser Box eine neue Variableninstantierung gefunden wurde:

"| EXIT (prinz charles)"

Wie schon nach dem ersten Durchlaufen von "prinz" muss nun noch die zweite Praemisse der "thronfolger"-Regel bewiesen und dazu zunaechst durch ihren CALL-Eingang betreten werden.
Im Gegensatz zum ersten Aufruf dieser Box, bei dem die Variable

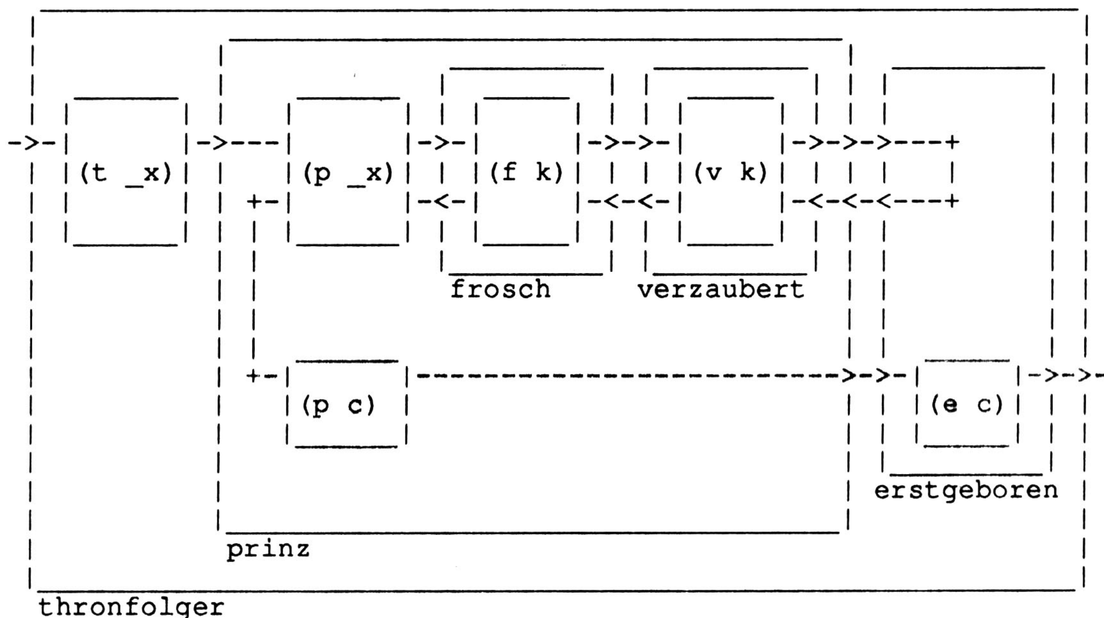
_x-2 an die Konstante kermit gebunden wurde und es deshalb keine "passende" Klausel mehr gab, verlaeuft dieser zweite Anlauf erfolgreich: Die Variable _x-2 ist jetzt an die Konstante charles gebunden, der eigentliche Aufruf der Praemisse lautet also "(erstgeboren charles)", und dazu gibt es sehr wohl eine *"passende" Klausel, naemlich das entsprechende Faktum "(ass* (erstgeboren charles))". Damit kann also endlich auch die "erstgeboren"-Box durch den EXIT-Ausgang verlassen werden.

```
"| CALL (erstgeboren charles)"
"| EXIT (erstgeboren charles)"
```

Weil die Praemisse "(erstgeboren _x)" aber zugleich auch die letzte Praemisse der angewandten "thronfolger"-Regel ist, kann damit auch die "thronfolger"-Box durch den EXIT-Ausgang verlassen werden: die Benutzeranfrage ist damit erfolgreich bewiesen, wobei die Anfragevariable _gesuchte-person schliesslich den Wert charles erhaelt:

```
"|EXIT (thronfolger charles)"
"success!"
"_gesuchte-person = charles"
```

Zum Ueberblick wenden wir uns nun auch noch einmal der graphischen Box-Modell-Darstellung zu, die jetzt fuer die gesamte Abarbeitung der Anfrage folgende Gestalt hat:



Insgesamt erhalten wir also bei der Abarbeitung der Anfrage "(thronfolger _gesuchte-person)" mit der o.a. Datenbasis (3) die folgende Ausgabe des Tracers:

```
|CALL (thronfolger _gesuchte-person)
| CALL (prinz _x-1)
|  CALL (frosch _x-2)
|  EXIT (frosch kermit)
|  CALL (verzaubert kermit)
|  EXIT (verzaubert kermit)
| EXIT (prinz kermit)
| CALL (erstgeboren kermit)
| FAIL (erstgeboren kermit)
| REDO (prinz kermit)
|  REDO (verzaubert kermit)
|  FAIL (verzaubert kermit)
|  REDO (frosch _x-2)
|  FAIL (frosch _x-2)
| EXIT (prinz charles)
| CALL (erstgeboren charles)
| EXIT (erstgeboren charles)
|EXIT (thronfolger charles)
```

Aus diesen Informationen kann der Benutzer sich natuerlich ueber einfache Kommandos die fuer ihn interessanten herausfiltern.

Bevor wir uns aber diesen Moeglichkeiten zuwenden, soll uns zunaechst grundsaetzlich die Implementation des Box-Modell-Tracers beschaeftigen. Zentraler Gegenstand des nun folgenden Abschnitts wird also die Frage sein, wie wir diese Trace-Informationen gemaess dem Box-Modell aus den Informationen erhalten koennen, die dem LISPLOG-Interpreter bei der Abarbeitung des Programms vorliegen. Da der LISPLOG-Interpreter in LISP realisiert ist, suchen wir somit also eine Abbildung von einem LISP-Trace des LISPLOG-Interpreters in die elementaren Informationseinheiten des Box-Modells.

Dazu muessen wir uns zunaechst einmal grundsaetzlich mit dem Kern des LISPLOG-Interpreters auseinandersetzen, um die Bedeutung der einzelnen Datenstrukturen und der Aufrufhierarchien zu erkennen. Eine ausfuehrliche Darstellung des LISPLOG-Interpreters wuerde allerdings den Umfang dieser Arbeit doch erheblich uebersteigen, so dass hier fuer eine solche ausfuehrliche Darstellung auf [Boley & Kammermeier et. al 1985] verwiesen sei.

Wir konzentrieren uns deshalb im folgenden nur auf die fuer die Extraktion der fuer das Box-Modell relevanten Informationen wichtigen Interpretationsfunktionen and-process und or-process:

6. Der LISPLOG-Interpreter

Der Beweis einer Anfrage an den LISPLOG-Interpreter erfolgt durch den Aufruf einer LISP-Funktion and-process, die als Argumente die Liste der zu beweisenden Ziele, eine zunaechst leere Bindungsumgebung (environment) und den Wert 1 fuer die aktuelle Tiefe des Beweises erhaelt.

Diese Funktion bedient sich einer weiteren LISP-Funktion or-process, deren Aufgabe es ist, ein ihr uebergebenes Ziel mit einer Klausel eines ebenfalls als Argument uebergebenen Datenbankausschnitts zu unifizieren und den Beweis der Praemissen (falls vorhanden) sowie der noch ausstehenden Ziele wiederum an die Funktion and-process zu uebertragen.

Diese Funktionen and-process und or-process stellen zusammen mit der Funktion unify, die die angesprochene Unifikation realisiert, den Kern des LISPLOG-Interpreters dar und sollen deshalb noch etwas ausfuehrlicher behandelt werden:

6.1. Die Kernfunktionen and-process und or-process

Die erste Kernfunktion and-process

```
(def and-process
  (lambda (list-of-goals environment level)
    (cond ((null list-of-goals)
            (print-bindings environment environment)
            (not (y-or-n-p "More? ")))
          (t (let ((goal (first list-of-goals))
                   (goals-left (rest list-of-goals)))
                (cond
                 ((is-user-defined goal)
                  (or-process (get (first goal) 'clauses)
                              goals-left
                              goal
                              environment
                              level))
                 ((is-primitive goal)
                  (prolog-primitive goals-left
                                    goal
                                    environment
                                    level))
                 ((is-lisp-defined goal)
                  (lisp-predicates goals-left
                                   goal
                                   environment
                                   level)))))))
```

beweist eine Liste von Zielen (list-of-goals), indem sie

- falls diese (inzwischen) leer ist, den Erfolg des Beweises durch Ausgabe der Variablenbindungen anzeigt und nachfragt, ob weitere Loesungen gewuenscht werden (and-process liefert dann nil = Failure, sonst t);

- ansonsten zunaechst einmal fuer das erste Ziel mittels or-process alle Klauseln zu dem entsprechenden Praedikatnamen durchsucht, wobei or-process bei Erfolg wiederum and-process fuer eine Liste zumindest der restlichen Ziele aufruft (Korekursion). Ist das erste Ziel ein PROLOG-Primitiv oder ein LISP-Praedikat, so wird statt or-process die Funktion prolog-primitive bzw. lisp-predicates aufgerufen, die bei erfolgreichem Beweis ebenfalls and-process fuer die noch ausstehenden Teilziele aufruft. Gelingen die nachfolgenden Beweise, so liefert and-process den Wert t; scheitern diese oder werden sie durch den Wunsch nach weiteren Loesungen als gescheitert betrachtet, so liefert and-process als Ergebnis nil.

Die andere bereits angesprochene Kernfunktion or-process

```
(def or-process
  (lambda (database-left goals-left goal environment level)
    (cond
     ((null database-left) nil)
     (t (let* ((assertion (rename-variables (first database-left)
                                            (list level)))
               (new-environment (unify goal
                                       (s-conclusion assertion)
                                       environment)))
              (cond
               ((null new-environment)
                (or-process (rest database-left)
                            goals-left
                            goal
                            environment
                            level))
               ((and-process (append (s-premisses assertion)
                                     goals-left)
                             new-environment
                             (add1 level)))
               ((not (cut-p assertion))
                (or-process (rest database-left)
                            goals-left
                            goal
                            environment
                            level))))))))
```

probiert fuer ein Ziel (goal) jede Assertion des uebergebenen Datenbankausschnitts (database-left), indem sie

- nil (Misserfolg) zurueckgibt, falls der Datenbankausschnitt keine Klauseln (mehr) enthaelt,

- andernfalls die Variablen der ersten Klausel des Datenbankausschnitts umbenennt und danach das Ziel und die Konklusion der Klausel mittels der Funktion unify konsistent zur alten Bindungsumgebung (environment) zu unifizieren versucht, sowie schliesslich
  - bei Unifikations-Misserfolg eine "Nichtanwendbarkeits-Rekursion" mit dem urspruenglichen Ziel und den restlichen Klauseln durchfuehrt,
  - bei Unifikations-Erfolg mittels and-process wieder eine Liste von Zielen beweisen laesst, welche die Praemissen der ausgewaehlten Klausel anstelle des ersten, jetzt erledigten and-process-Zieles enthaelt, und
  - bei Misserfolg dieses Beweises durch and-process in dem Fall, dass die Klausel (assertion) nicht mit einem "Cut" markiert war, eine "Backtracking-Rekursion" mit dem urspruenglichen Ziel und den restlichen Klauseln durchgefuehrt und andernfalls direkt Misserfolg (nil) liefert.

Die Funktion unify, die hier nicht ausfuehrlich dargestellt werden soll, liefert bei Misserfolg nil und bei erfolgreicher Unifikation ein nicht-leeres neues environment, welches das als Argument uebergebene alte environment konsistent um null oder mehr Bindungen erweitert.

Die Funktionen s-conclusion und s-premisses selektieren die Konklusion bzw. die Liste der Praemissen aus einer Klausel.

6.2. Extraktion der relevanten Box-Modell-Informationen

Zur Implementation eines Box-Modell-Tracers benoetigen wir, wie in Abschnitt 5.3. bereits dargestellt, eine Abbildung von einem LISP-Trace des LISPLOG-Interpreters in die elementaren Informationseinheiten des Box-Modells. Diese bestehen gerade aus dem Passieren der Ein- und Ausgaenge der einzelnen Procedure-Boxes, also aus den Informationen ueber CALL, EXIT, FAIL und REDO.

Um zu erkennen, wie diese Informationen aus dem LISP-Trace der Interpreterfunktionen and-process und or-process gewonnen werden koennen, betrachten wir einmal einen solchen LISP-Trace von and- und or-process fuer das bekannte "thronfolger"-Beispiel aus Abschnitt 5.3.:

```
(3)   (ass (prinz _x) (frosch _x) (verzaubert _x))
      (ass (prinz charles))
      (ass (frosch kermit))
      (ass (verzaubert kermit))
      (ass (erstgeboren charles))
      (ass (thronfolger _x) (prinz _x) (erstgeboren _x))
```

Der folgende LISP-Trace stellt nun alle Aufrufe und Rueckgaben von Werten der Implementationsfunktionen and-process (AND) und or-process (OR) dar, wie sie bei der Bearbeitung der Anfrage "(thronfolger _wer)" mit o.a. Datenbasis (3) stattfinden: [12]

```
enter AND ((t _w)) {} 1
|enter OR (((t _x) (p _x) (e _x))) () (t _w) {} 1
||enter AND ((p _x-1) (e _x-1)) {_w/_x-1} 2
|||enter OR (((p _x) (f _x) (v _x)) ((p c))) ((e _x-1))
||||         (p _x-1) {_w/_x-1} 2
||||enter AND ((f _x-2) (v _x-2) (e _x-1)) {_x-1/_x-2,_w/_x-1} 3
|||||enter OR (((f k))) ((v _x-2) (e _x-1)) (f _x-2)
||||||         {_x-1/_x-2,_w/_x-1} 3
||||||enter AND ((v _x-2) (e _x-1)) {_x-2/k,_x-1/_x-2,_w/_x-1} 4
|||||||enter OR (((v k))) ((e _x-1)) (v k)
||||||||         {_x-2/k,_x-1/_x-2,_w/_x-1} 4
||||||||enter AND ((e _x-1)) {_x-2/k, _x-1/_x-2, _w/_x-1} 5
|||||||||enter OR (((e c))) () (e k) {_x-2/k,_x-1/_x-2,_w/_x-1} 5
||||||||||enter OR () () (e k) {_x-2/k,_x-1/_x-2,_w/_x-1} 5
||||||||||exit OR with result: nil
|||||||||exit OR with result: nil
||||||||exit AND with result: nil
||||||||enter OR () ((e _x-1)) (v k) {_x-2/k,_x-1/_x-2,_w/_x-1} 4
||||||||exit OR with result: nil
|||||||exit OR with result: nil
||||||exit AND with result: nil
||||||enter OR () ((v _x-2) (e _x-1)) (f _x-2)
|||||||         {_x-1/_x-2,_w/_x-1} 3
||||||exit OR with result: nil
|||||exit OR with result: nil
||||exit AND with result: nil
||||enter OR (((p c))) ((e _x-1)) (p _x-1) {_w/_x-1} 2
|||||enter AND ((e c)) {_x-1/c,_w/_x-1} 3
||||||enter OR (((e c))) () (e c) {_x-1/c,_w/_x-1} 3
|||||||enter AND () {_x-1/c,_w/_x-1} 4
|||||||exit AND with result: t
||||||exit OR with result: t
|||||exit AND with result: t
||||exit OR with result: t
|||exit OR with result: t
||exit AND with result: t
|exit OR with result: t
exit AND with result: t
```

Um nun die gewuenschte Abbildung vom LISP-Trace in die Box-

---

[12] Zur uebersichtlicheren Darstellung werden Konstanten und Praedikatnamen wie bei der graphischen Darstellung des Box-Modells in 5.3. abgekuerzt. Fuer die aktuelle Bindungsumgebung (environment) wird eine benutzerfreundliche Darstellung verwendet, auf die wir spaeter noch eingehen werden.

Modell-Darstellung (mittels CALL, EXIT, FAIL und REDO) zu finden, kann man diesen LISP-Trace und die von einem Box-Modell-Tracer zu erwartende Ausgabe (vgl. Abschnitt 5.3) gegenueberstellen. Auf diese Weise kann man dann die einzelnen Box-Modell-Uebergaenge wie folgt im LISP-Trace identifizieren:

(1) Ein Aufruf der Funktion and-process (AND) mit nicht-leerer list-of-goals entspricht im Box-Modell dem Betreten genau einer Procedure-Box durch den CALL-Eingang.

(2) Das Scheitern des or-process-Aufrufs (OR) innerhalb der Funktion and-process - angezeigt durch Rueckgabe des Werts nil = Failure - entspricht im Box-Modell dem Verlassen von genau einer Procedure-Box durch den FAIL-Ausgang.

(3) Eine erfolgeiche Unifikation mit einem Faktum im or-process entspricht im Box-Modell dem Verlassen mindestens einer Procedure-Box durch den EXIT-Ausgang. Es koennen hierbei auch gleich mehrere Boxen erfolgreich verlassen werden!

(4) Das Scheitern des Beweises der noch ausstehenden Ziele mit and-process nach einer erfolgreichen Unifikation mit einem Faktum (wiederum angezeigt durch Rueckgabe des Werts nil) entspricht dann im Box-Modell analog dem Wiederbetreten der zuvor noch ueber EXIT verlassenen Procedure-Boxes durch den REDO-Eingang, also dem Betreten von ebenfalls mindestens einer Procedure-Box durch den REDO-Eingang.

Entsprechend dieser Zuordnung koennen wir jetzt zunaechst den LISP-Trace um die zugehoerigen Box-Modell-Uebergaenge erweitern, so dass wir nun folgenden (erweiterten) Lisp-Trace erhalten, fuer den jedoch noch nicht geklaert ist, wie die eingefuegten Box-Modell-Informationen aus den Aufrufen bzw. Wertrueckgaben konstruiert werden koennen:

```
enter AND ((t _w)) {} 1
|                   --> |CALL (thronfolger _wer)
|enter OR (((t _x) (p _x) (e _x))) () (t _w) {} 1
||enter AND ((p _x-1) (e _x-1)) {_w/_x-1} 2
|||                 --> | CALL (prinz _x-1)
|||enter OR (((p _x) (f _x) (v _x)) ((p c))) ((e _x-1))
||||        (p _x-1) {_w/_x-1} 2
||||enter AND ((f _x-2) (v _x-2) (e _x-1)) {_x-1/_x-2,_w/_x-1} 3
|||||               --> |  CALL (frosch _x-2)
|||||enter OR (((f k))) ((v _x-2) (e _x-1)) (f _x-2)
||||||        {_x-1/_x-2,_w/_x-1} 3
||||||              --> |  EXIT (frosch kermit)
||||||enter AND ((v _x-2) (e _x-1)) {_x-2/k,_x-1/_x-2,_w/_x-1} 4
|||||||             --> |  CALL (verzaubert kermit)
|||||||enter OR (((v k))) ((e _x-1)) (v k)
||||||||        {_x-2/k,_x-1/_x-2,_w/_x-1} 4
||||||||            --> |  EXIT (verzaubert kermit)
||||||||                | EXIT (prinz kermit)
```

```
||||||||enter AND ((e _x-1)) {_x-2/k, _x-1/_x-2, _w/_x-1} 5
|||||||||             --> | CALL (erstgeboren kermit)
|||||||||enter OR (((e c))) () (e k) {_x-2/k,_x-1/_x-2,_w/_x-1} 5
||||||||||enter OR () () (e k) {_x-2/k,_x-1/_x-2,_w/_x-1} 5
||||||||||exit OR with result: nil
|||||||||exit OR with result: nil
|||||||||             --> | FAIL (erstgeboren kermit)
|||||||||                 | REDO (prinz kermit)
|||||||||                 |  REDO (verzaubert kermit)
||||||||exit AND with result: nil
||||||||enter OR () ((e _x-1)) (v k) {_x-2/k,_x-1/_x-2,_w/_x-1} 4
||||||||exit OR with result: nil
|||||||exit OR with result: nil
|||||||               --> |  FAIL (verzaubert kermit)
|||||||                   |  REDO (frosch _x-2)
||||||exit AND with result: nil
||||||enter OR () ((v _x-2) (e _x-1)) (f _x-2)
|||||||          {_x-1/_x-2,_w/_x-1} 3
||||||exit OR with result: nil
|||||exit OR with result: nil
|||||                 --> |  FAIL (frosch _x-2)
||||exit AND with result: nil
||||enter OR (((p c))) ((e _x-1)) (p _x-1) {_w/_x-1} 2
|||||                 --> | EXIT (prinz charles)
|||||enter AND ((e c)) {_x-1/c,_w/_x-1} 3
||||||                --> | CALL (erstgeboren charles)
||||||enter OR (((e c))) () (e c) {_x-1/c,_w/_x-1} 3
|||||||               --> | EXIT (erstgeboren charles)
|||||||                   |EXIT (thronfolger charles)
|||||||enter AND () {_x-1/c,_w/_x-1} 4
|||||||exit AND with result: t
||||||exit OR with result: t
|||||exit AND with result: t
||||exit OR with result: t
|||exit OR with result: t
||exit AND with result: t
|exit OR with result: t
exit AND with result: t
```

6.3. Implementation des Box-Modells: CALL und FAIL

Anhand dieser Gegenueberstellung koennen wir nun versuchen, die gewuenschte Abbildung vom LISP-Trace in die elementaren Informationseinheiten des Box-Modells (CALL, EXIT, FAIL und REDO), die oben bereits informell skizziert wurde, nun konkret zu machen, also die konkrete Konstruktion der Box-Modell-Informationen mittels Aenderungen am Code von and- und or-process zu realisieren.

Die Realisierung von CALL und FAIL erscheint sehr einfach und fuehrt unmittelbar zu folgenden Erweiterungen an der Funktion and-process: [13], [14]

```
(def and-process
  (lambda (list-of-goals environment level)
    (cond ((null list-of-goals)
           (print-bindings environment environment)
           (not (y-or-n-p "More? ")))
          (t (let ((goal (first list-of-goals))
                   (goals-left (rest list-of-goals)))
;;
;; Behandlung von CALL:
;; --------------------
                  (princ "CALL ")
                  (print (ultimate-inst goal environment))
                  (terpri)
                  (or
                   (cond
                    ((is-user-defined goal)
                     (or-process (get (first goal) 'clauses)
                                 goals-left
                                 goal
                                 environment
                                 level))
                    ((is-primitive goal)
                     (prolog-primitive goals-left
                                       goal
                                       environment
                                       level))
                    ((is-lisp-defined goal)
                     (lisp-predicates goals-left
                                      goal
                                      environment
                                      level)))
;; Behandlung von FAIL:
;; --------------------
                   (princ "FAIL ")
                   (print (ultimate-inst goal environment))
                   (terpri))))))
```

---

[13] Die Funktion ultimate-inst fuehrt eine weitestmoegliche Instantiierung eines Terms (hier: goal) in der entsprechenden Bindungsumgebung (environment) durch.

[14] Beachte: Die Funktionen print und terpri liefern als Wert stets nil, so dass der eigentliche Wert der Funktion and-process (t = Success, nil = Failure) durch diese Aenderung nicht beeinflusst wird.

Bei dem Versuch, fuer EXIT und REDO auf aehnliche Weise durch Einfuegen entsprechender "Ausgabeanweisungen" zu einer Implementation zu kommen, muessen wir nun allerdings feststellen, dass dies so nicht moeglich ist, da die fuer die entsprechende Ausgabe notwendigen Informationen nicht vollstaendig vorliegen. Das folgende Beispiel aus dem oben dargestellten LISP-Trace zeigt dies: Man betrachte den Aufruf von or-process:

```
|||||||enter OR (((v k))) ((e _x-1)) (v k)
||||||||         {_x-2/k,_x-1/_x-2,_w/_x-1} 4
||||||||             --> |  EXIT (verzaubert kermit)
||||||||                 | EXIT (prinz kermit)
||||||||enter AND ((e _x-1)) {_x-2/k, _x-1/_x-2, _w/_x-1} 5
|||||||||            --> | CALL (erstgeboren kermit)
```

Das zu beweisende Ziel "(v k)" = "(verzaubert kermit)" kann mit dem Kopf (Konklusion) der ersten Datenbank-Klausel "((v k))" erfolgreich unifiziert werden. Da es sich bei dieser Klausel um ein Faktum handelt, wird im Box-Modell entsprechend die Procedure-Box fuer "(v k)" durch den EXIT-Ausgang verlassen. Bei naeherer Betrachtung der "Vorgeschichte" dieses Aufrufs stellen wir allerdings fest, dass das zu beweisende Ziel "(v k)" zugleich aber auch die letzte noch zu beweisende Praemisse einer zuvor angewandten Klausel darstellt. Demnach ist auch die dieser Klausel entsprechende Procedure-Box, hier: "(p _x-1)", durch den EXIT-Ausgang zu verlassen, bevor der nun folgende Aufruf von and-process fuer die noch verbleibenden Ziele (goals-left) zu einem CALL fuer die naechste Procedure-Box "(e k)" fuehrt.

Auf welche Weise kann aber die hier mit "Vorgeschichte" umschriebene Information innerhalb dieses Aufrufs hergeleitet werden?

Offensichtlich reichen dazu die beim Aufruf an or-process uebergebenen Argumente nicht aus. Gleiches gilt natuerlich auch fuer die Aufbereitung der REDO-Informationen, die ja ggfs. beim Scheitern des nachfolgenden Beweises durch and-process ebenfalls innerhalb dieses or-process-Aufrufs vorzunehmen ist.

Ein denkbarer Weg zur Schliessung dieser "Informationsluecke", der auch tatsaechlich zunaechst eingeschlagen wurde (vgl. [Herr & Meyer 1985]), besteht in der Einfuehrung neuer Datenstrukturen zur direkten Repraesentation des abzuarbeitenden Und/Oder-Baums. Betrachtet man die Implementierung der Funktionen and-process und or-process, so stellt man fest, dass jeweils nur die Information ueber den noch zu bearbeitenden Teil des Suchraums in Form einer linearen Liste aller noch zu beweisenden Teilziele (Argument list-of-goals bzw. goals-left) vorliegt. Die Information ueber bereits bewiesene Teilziele, vor allem aber ueber Ziele, deren direkte oder indirekte Praemissen gerade bewiesen werden sollen ("aktive Ziele"), wird nicht mit uebergeben. Diese Information wird aber gerade zur Erzeugung der Box-Modell-Repraesentation (fuer geschachtelte EXIT-REDO-Uebergaenge) benoetigt.

Hier setzt der in [Herr & Meyer 1985] beschriebene Vorschlag an, indem zusaetzlich globale Datenstrukturen trace-stack und redo-list angelegt werden, die zusammengefasst den jeweils aktuell expandierten Suchbaum repraesentieren. Aus diesen globalen Daten koennen dann die gewuenschten Box-Modell-Informationen extrahiert werden.
Neben der Tatsache, dass die Verwaltung dieser globalen Daten einen grossen Aufwand erfordert, hat diese Realisierung vor allem den Nachteil, dass Informationen, die aufgrund der korekursiven Implementation des LISPLOG-Interpreters bereits implizit im LISP-Rekursions-Stack vorhanden sind, nochmals explizit verwaltet werden muessen.

Deshalb wurde bei der spaeteren Ueberarbeitung und Erweiterung der LISPLOG-Interaktionsumgebung eine andere Implementierung gewaehlt, die ohne diese globalen Datenstrukturen trace-stack und redo-list und somit ohne die damit verbundene Redundanz auskommt.

6.4. Erweiterungen am Kern des LISPLOG-Interpreters

Wesentlich fuer die Vermeidung dieser Redundanz ist eine Erweiterung der Liste der noch zu beweisenden Ziele (list-of-goals), wie sie als Argument an and-process und - aufgespalten in goal und goals-left - weiter an or-process uebergeben wird.
In diese Liste, die - wie wir gleich sehen werden - auch als die aktuelle Resolvente in einem Resolutionsbeweis verstanden werden kann, sollen in der hier zu beschreibenden Implementierung nicht nur alle noch zu beweisenden Ziele, sondern auch die bereits "aktiven" Ziele aufgenommen werden.

Zur Klaerung dieser Begriffe sowie zum erweiterten Verstaendnis des LISPLOG-Interpreters erscheint es sinnvoll, die Abarbeitung eines LISPLOG-Programms, bei dem wir uns wie bekannt zunaechst auf den praedikativen Teil beschraenken wollen, als die Konstruktion eines Resolutionsbeweises fuer eine gegebene Anfrage in einer bestimmten Klauselmenge (Datenbasis) aufzufassen.

Nach einer kurzen Darstellung des Resolutionsverfahrens, bei der wir jedoch bewusst auf die Details der Herleitung (vgl. [Robinson 1965]) verzichten wollen, betrachten wir nochmals das bekannte "thronfolger"-Beispiel - dieses Mal jedoch als Resolutionsbeweis.

Exkurs: Das Resolutionsverfahren

Das Grundprinzip des Resolutionsverfahrens besteht in der Widerlegung des negierten Satzes (engl. refutation procedure). Daraus folgt, dass im Fortgang des Beweises versucht wird, immer kuerzere Klauseln herzuleiten, bis es schliesslich gelingt, die leere Klausel - sie entspricht dem Widerspruch, also der Widerlegung des negierten Satzes - zu erzeugen. Diese Verkuerzung von Klauseln beruht darauf, in einer Klausel ein

Literal A ohne Negation, in einer anderen Klausel dasselbe Literal mit Negation, also -A [15], zu finden. Aus beiden Klauseln wird nun eine neue Klausel als Resultat gebildet, die alle Literale aus diesen "Elternklauseln" bis auf die beiden ausgewaehlten komplementaeren Literale enthaelt. Diesen Schritt bezeichnet man als Resolution der beiden Elternklauseln auf dem ausgewaehlten Literal A, die dabei entstehende neue Klausel als Resolvente. Diese Schlussform der Resolution beruht auf der einfachen aussagenlogischen Beziehung

(A v B) ^ (-A v C) <=> (B v C).

Dass die leere Klausel mit dem Widerspruch gleichbedeutend ist, sehen wir aus

A ^ -A <=> Widerspruch.

Wir haben also eine Klausel, die nur ein (positives) Literal und eine, die nur dasselbe (aber negierte) Literal enthaelt. Daraus koennen wir mittels Resolution die leere Klausel bilden; diese zeigt also den Widerspruch an. Dieser Widerspruch wurde aus der Negation des zu beweisenden Satzes und der Konjunktion aller Axiome (Klauseln der Datenbasis) erzeugt, so dass damit die Gueltigkeit des Satzes nachgewiesen ist.

Nun liegen die Dinge aber nicht ganz so einfach, weil Literale auch Variablen enthalten koennen. Dann kann es vorkommen, dass Literale zwar einander aehnlich, aber vielleicht nicht identisch sind. Betrachten wir dazu die thronfolger-Klauselmenge [16]

```
(4)   {(prinz _x), -(frosch _x), -(verzaubert _x)}
      {(prinz charles)}
      {(frosch kermit)}
      {(verzaubert kermit)}
      {(erstgeboren charles)}
      {(thronfolger _x), -(prinz _x), -(erstgeboren _x)}
```

und die Anfrage (thronfolger _wer), die also den zu beweisenden Satz darstellt. Um mit der Negation der Anfrage, also der Klausel {-(thronfolger _wer)}, sowie einem geeigneten Axiom aus der Klau-

---

[15] Die Negation eines Literals soll hier in der weiteren Darstellung aus drucktechnischen Gruenden durch das Voranstellen des Zeichens "-" ausgedrueckt werden.

[16] Eine fuer LISPLOG durch "(ass concl prem[1] ... prem[n])" ausgedrueckte Implikation "prem[1] ^ ... ^ prem[n] => concl" wird dabei umgeformt zu "concl v -prem[1] v ... v -prem[n]". Diese Disjunktionen werden hier nun als Mengen von Literalen notiert.

selmenge, wobei hier nur die Klausel

{(thronfolger _x-1), -(prinz _x-1), -(erstgeboren _x-1)} [17]

in Betracht kommt, resolvieren zu koennen, muss zuvor eine entsprechende Ersetzung der Variablen vorgenommen werden, um anschliessend wirklich zwei <u>exakt</u> <u>komplementaere</u> Literale entfernen zu koennen. Die Menge dieser Variablenersetzungen wird zu einer <u>Substitution</u> zusammengefasst. Sei L ein Literal und S = {v[1]/t[1], v[2]/t[2], ...} eine Substitution. Dann heisst S(L), das man durch gleichzeitige Ersetzung jeder Variablen v[i] in L durch das zugehoerige t[i] erhaelt, eine <u>Instanz</u> von L. Eine Substitution S heisst dann ein <u>Unifikator</u> fuer ein Literalpaar {L1,L2}, wenn gilt: S(L1) = S(L2).

Fuer unser thronfolger-Beispiel bedeutet das, dass wir, bevor wir auf den "fast komplementaeren" Literalen "(thronfolger _x-1)" und "-(thronfolger _wer)" ueberhaupt resolvieren koennen, zunaechst versuchen muessen, diese Literale durch eine entsprechende Substitution bis auf die Negation exakt gleich zu machen: sie also zu unifizieren. Dabei erhalten wir den Unifikator

S = {_wer/_x-1}. [18]

Diese Substitution S muessen wir nun nicht nur auf die beiden komplementaeren Literale, sondern auch auf die beiden bei diesem Resolutionsschritt beteiligten Elternklauseln

K1 = {-(thronfolger _wer)} und
K2 = {(thronfolger _x-1), -(prinz _x-1), -(erstgeboren _x-1)}

anwenden, da von dieser Substitution S betroffene Variablen auch in weiteren Literalen der beteiligten Klauseln K1 und K2 vorkommen koennen, wie es hier mit der Variablen _x-1 der Fall ist.

---

[17] Die einzelnen Axiome der Klauselmenge (4) sind logisch voneinander unabhaengig. Deshalb sind Variablennamen immer relativ zu der Klausel, in der sie auftreten. Zwischen den Variablen _x in der ersten und dritten Klausel der "thronfolger"-Datenbasis (4) besteht kein Zusammenhang! Darum werden bei <u>jeder</u> Resolution mit einer Klausel aus der Datenbasis zunaechst <u>alle</u> darin auftretenden Variablennamen durch Umbenennung (in LISPLOG durch Anhaengen der Nummer des Resolutionsschritts an den Variablennamen) "einmalig gemacht".

[18] Lies: "Die Variable _wer wird durch den Term _x-1, der in diesem Fall wiederum eine Variable darstellt (aber nicht darstellen muss) ersetzt."

Damit erhalten wir schliesslich als Resolvente der Klauseln K1 und K2 die Klausel

R = {-(prinz _x-1), -(erstgeboren _x-1)},

mit der der Resolutionsbeweis dann im naechsten Schritt fortgesetzt werden kann. [19]

Betrachten wir dazu die Klauselmenge (4), dann erkennen wir, dass es nun mehrere Moeglichkeiten gibt, zu der aktuellen Resolvente R eine zweite Klausel K aus der Datenbasis zu finden, so dass dann im naechsten Schritt auf R und K resolviert werden kann:

Zum einen gibt es zwei Moeglichkeiten, aus R ein Literal auszuwaehlen, zu dem dann eine Klausel mit "fast komplementaerem" Literal gesucht werden soll. In LISPLOG ist diese Auswahl wie bei anderen Standard-PROLOG-Implementationen auf das jeweils erste/linkeste Literal der aktuellen Resolvente festgelegt, in diesem Fall also (zunaechst) auf das Literal "-(prinz _x)".
Aber auch jetzt gibt es wiederum zwei Moeglichkeiten, aus der Datenbasis eine Klausel mit einem moeglicherweise mit "(prinz _x)" unifizierbaren Literal auszuwaehlen. [20], [21] In LISPLOG entscheidet hier jetzt die Reihenfolge der Klauseln in der Daten-

---

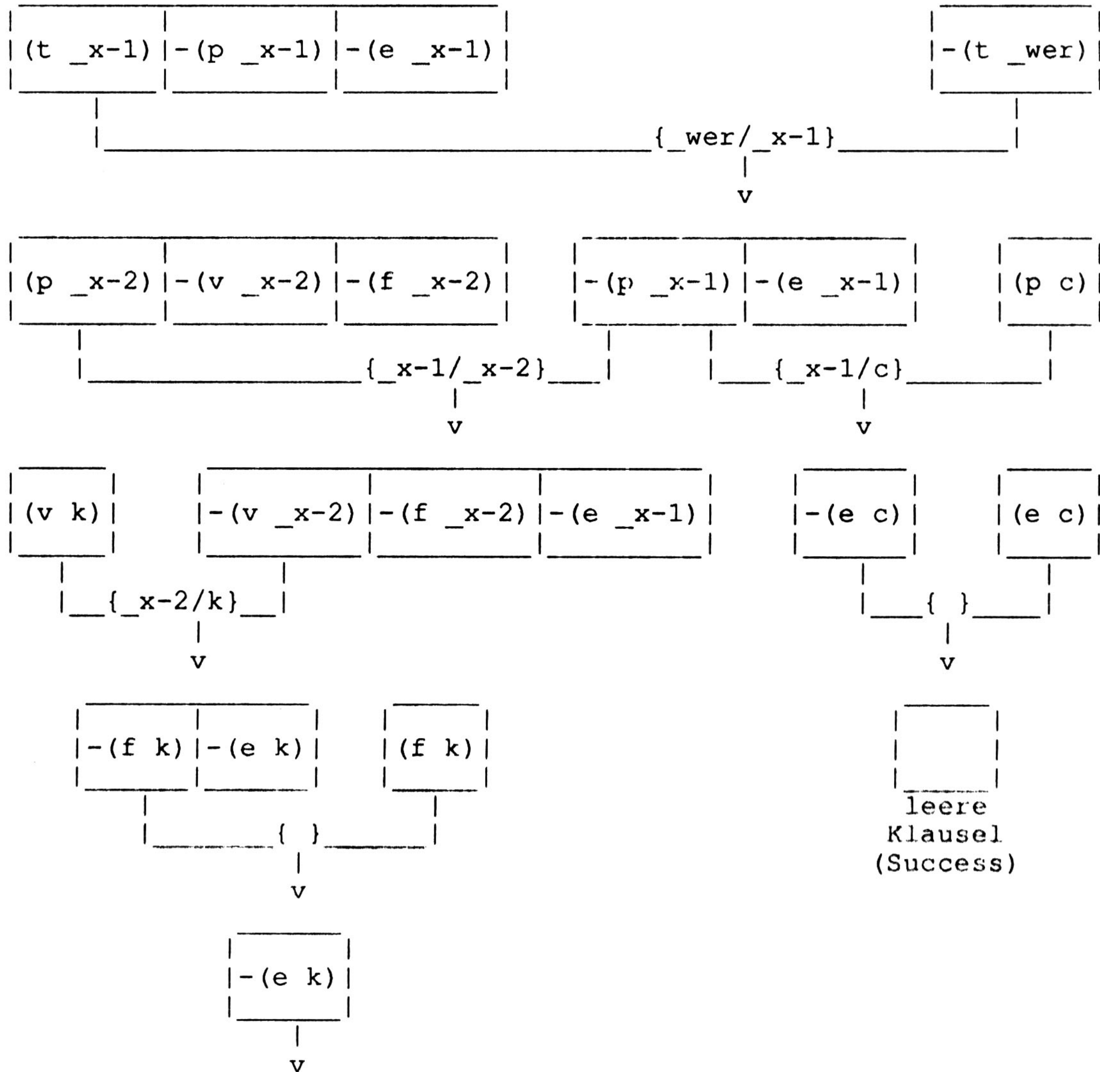
[19] Ein Resolutionsverfahren wie das der PROLOG zugrundeliegenden SLD-Resolution, bei dem grundsaetzlich zunaechst versucht wird, die aktuelle Resolvente im naechsten Schritt wiederum als Elternklausel zu verwenden, heisst linear.

[20] Fuer die Effizienz eines PROLOG-Systems ist es wichtig, die Menge der moeglicherweise unifizierbaren und somit fuer die Auswahl in Frage kommenden Klauseln zwar vollstaendig aber moeglichst klein zu bestimmen, um den Aufwand fuer vergebliche Unifikationsversuche moeglichst gering zu halten. Dieses Verfahren zum gezielten Zugriff auf die Datenbasis, die "Datenbasis-Indexierung", beschraenkt sich in dem hier dargestellten Code auf den Praedikatnamen, d. h. alle Klauseln, die die Negation eines Literals mit gleichem Praedikatnamen enthalten, werden als moegliche Elternklauseln betrachtet und koennen ggfs. erst durch Fehlschlagen der Unifikation ausgeschlossen werden. In [Bernardi 1986a] werden jedoch ein sehr maechtiges und flexibles Indexierungsverfahren und seine Implementation beschrieben.

[21] Diese Suche nach einem solchen (positiven) Literal in einer Klausel der Datenbasis kann in PROLOG auf das erste Literal einer Klausel (Konklusion der PROLOG-Regel) beschraenkt bleiben, da hier nur Klauseln mit hoechstens einem positiven Literal, sog. Hornklauseln, vorkommen (vgl. Abb. auf der naechsten Seite).

basis ("von oben nach unten"). Erst wenn sich der durch die Auswahl einer Klausel eingeschlagene Weg als nicht zur leeren Klausel fuehrend erweist, wird die naechste in Frage kommende Klausel (falls vorhanden) probiert (Backtracking).

Auf diese Art und Weise erhalten wir fuer unser "thronfolger"-Beispiel schliesslich den folgenden (SLD-)Beweisbaum, der neben einem blinden Pfad, der nicht zum Erfolg in Form der leeren Klausel fuehrt, auch tatsaechlich einen Erfolgspfad von der Anfrage bis zur leeren Klausel enthaelt:



Die Vereinigung aller entlang dieses Erfolgspfades vorgenommenen Substitutionen

S1 = {_wer/_x-1}, S2 = {_x-1/charles}, S3 = {}

liefert uns dann auch eine Einsetzung fuer die Variablen, die in der Anfrage enthalten waren: Durch sukzessive Anwendung dieser Substitutionen erhalten wir als "Loesung" fuer die Anfrage "(thronfolger _wer)" die <u>Antwortsubstitution</u>

```
S3(S2(S1({(thronfolger _wer)})))
= S3(S2({(thronfolger _x-1)}))
= S3({(thronfolger charles)})
= {(thronfolger charles)}.
```

Vergleichen wir nun die Schritte in diesem Resolutionsbeweis mit dem Trace der Funktionen and- und or-process (vgl. Seite 24):

Offensichtlich entspricht jeder Aufruf von and-process gerade einem Resolutionsschritt. Die Funktion and-process selektiert mit "(first list-of-goals)" das erste Literal aus der aktuellen Resolvente (list-of-goals), sucht zu diesem Literal mit Hilfe der Indexierung nach dem Praedikatnamen "(get (first goal) 'clauses)" die moeglicherweise fuer eine Resolution brauchbaren Klauseln zusammen und laesst die eigentliche Resolution dann von der Funktion or-process durchfuehren. Diese versucht zunaechst mit Hilfe der Unifikation eine wirklich brauchbare Klausel auszuwaehlen, bildet die neue Resolvente und leitet dann durch einen erneuten Aufruf von and-process mit der neuen Resolvente (list-of-goals) den naechsten Resolutionsschritt ein. Auch das Backtracking aus einem blinden Pfad im Beweisbaum und Fortsetzen des Beweises mit einer alternativen Elternklausel wird in der Funktion or-process organisiert.

Kommen wir noch einmal zurueck zum Beweisbaum. Mit seiner Hilfe koennen wir nun auch den schon frueher als "Ziel, dessen direkte oder indirekte Praemissen gerade bewiesen werden sollen", eingefuehrten Begriff des "aktiven Ziels" definieren:

Im Gegensatz zu den noch zu beweisenden Zielen, die ja gerade die aktuelle Resolvente bilden, wollen wir unter den "bearbeiteten Zielen" zu einem bestimmten Zeitpunkt des Beweises gerade die Literale verstehen, nach denen entlang des aktuellen Pfades im Beweisbaum bisher resolviert wurde. Dies sind also jene Ziele bzw. Literale, die bei den entsprechenden Resolutionsschritten aus der aktuellen Resolvente "herausgeschnitten" und durch die verbleibenden Literale der anderen Elternklausel, die den Praemissen der angewandten PROLOG-Regel entsprechen, ersetzt wurden. Unter diesen "bearbeiteten Zielen" befinden sich nun solche Ziele, die bereits vollstaendig bewiesen worden sind ("passive Ziele"), und solche, deren Beweis noch nicht abgeschlossen ist ("aktive Ziele"):

Wird nun in einem Resolutionsschritt auf der aktuellen Resolvente und einer einelementigen Elternklausel resolviert, so ist das betreffende Literal (Ziel) damit bewiesen und somit ein "passives Ziel". Enthaelt die aus der Datenbasis entnommene Elternklausel neben dem Literal L, nach dem resolviert wird, noch weitere Literale L2 bis Ln, so gilt L erst dann als "passives Ziel", wenn auch alle L2 bis Ln bewiesen, also "passive Ziele" sind. Und schliesslich: "Bearbeitete Ziele", die noch nicht "passiv" sind, heissen "aktive Ziele". Demnach ist ein LISPLOG-Beweis genau dann erfolgreich, wenn es keine "aktiven Ziele" mehr gibt.

Nach dieser Definition gelten zum Zeitpunkt unmittelbar <u>vor</u> dem letzten Resolutionsschritt im blinden Zweig des Beweisbaumes (Resolution auf "(frosch kermit)") die Literale/Ziele

"(thronfolger _wer)" und "(prinz _x-1)"

als "aktive Ziele", waehrend das Ziel "(verzaubert _x-2)" bereits bewiesen und damit "passiv" ist. <u>Nach</u> der Resolution auf dem Literal "(frosch kermit)" gilt dann natuerlich dieses Ziel sofort als "passiv", da die Datenbank-Klausel keine weiteren Literale enthaelt. Ausserdem ist mit "(verzaubert _x-2)" und "(frosch kermit)" jetzt aber auch "(prinz _x-1)" bewiesen, so dass nunmehr nur noch das Literal "(thronfolger _wer)" als "aktives Ziel" gilt.

<u>6.5. Implementation des Box-Modells: EXIT und REDO</u>

Durch Aufnahme auch der "aktiven Ziele" in die list-of-goals sind wir nun in der Lage, auch die Informationen ueber EXIT und REDO von Procedure-Boxes korrekt aus den jeweils vorliegenden Informationen (ausschliesslich Argumente der Funktionen and-process und or-process) herzuleiten. Dazu muessen natuerlich die "aktiven Ziele" in der list-of-goals auch als solche eindeutig gekennzeichnet werden, um sie von den eigentlichen noch zu beweisenden Zielen unterscheiden zu koennen: Ein "aktives Ziel" G tragen wir deshalb in der Form

```
(G . active)
```

in die list-of-goals ein. Der Eintrag eines solchen "aktiven Ziels" erfolgt sinnvollerweise genau dort, wo bisher das als naechstes zu bearbeitende Ziel aus der list-of-goals entnommen wurde. Diese Entnahme des naechsten zu bearbeitenden Ziels (goal) aus der list-of-goals entspricht ja gerade dem Beginn eines Beweises fuer dieses Ziel, das damit jetzt also nach der Definition in 6.4. "aktiv" ist. Deshalb tritt an die Stelle der bisherigen Selektion

```
(let ((goal (first list-of-goals))
      (goals-left (rest list-of-goals)))
      ...)
```

von "goal" (dem als naechstes zu beweisenden Ziel) und "goals-left" (der Liste der anschliessend noch zu beweisenden Ziele) nun der folgende Ausdruck:

```
(let* ((goal (first new-list-of-goals))
       (goals-left (cons (cons goal 'active)
                         (rest new-list-of-goals))))
    ...)
```

Das Hinzufuegen der Praemissen einer angewandten Regel beim Aufruf von and-process durch

```
(append (s-premisses assertion) goals-left)
```

bleibt von diesen Aenderungen natuerlich unberuehrt.

Das Entfernen der "aktiven Ziele" aus der list-of-goals erfolgt im Zusammenhang mit der Aufbereitung der EXIT-Informationen: Bereits in Abschnitt 6.2. haben wir festgestellt, dass ein Verlassen einer Procedure-Box durch den EXIT-Ausgang immer im Zusammenhang steht mit einer erfolgreichen Unifikation eines Ziels mit einem Faktum, korrekt: mit dem Kopf einer Regel ohne Praemissen. Je nach der "Vorgeschichte" koennen dabei aber auch gleich mehrere Procedure-Boxes mit EXIT verlassen werden. Die hier mit "Vorgeschichte" umschriebene Information liegt nun im Gegensatz zur urspruenglichen Version (vgl. 6.1.) in Form der Zusammensetzung der list-of-goals aus "aktiven" und noch zu beweisenden Zielen vor:

Beginnt die an and-process uebergebene list-of-goals naemlich mit einem "aktiven Ziel", dann bedeutet dies, dass im rufenden or-process genau dieses Ziel erfolgreich mit einem Faktum (einer Regel ohne Praemissen) unifiziert werden konnte. Haette die dort angewandte Regel naemlich noch Praemissen, dann stuenden diese jetzt am Beginn der list-of-goals und waeren natuerlich nicht als "aktiv" markiert. Beginnt die list-of-goals darueberhinaus sogar mit mehreren "aktiven Zielen", ist also von der Form

```
((G[1] . active) ... (G[n] . active) G[n+1] ...),
```

dann sind alle Ziele G[1] bis G[n] durch die gerade zuvor in or-process erfolgte Unifikation von G[1] mit einem Faktum bewiesen worden, weil jedes Ziel G[i] (fuer i=1,...,n) jeweils die letzte Praemisse einer Regel ist, deren Kopf mit dem Ziel G[i+1] unifizierbar war und die deshalb angewendet wurde. Somit werden beim Aufruf von and-process alle entsprechenden Procedure-Boxes fuer die am Beginn der list-of-goals stehenden "aktiven Ziele" G[1] bis G[n] durch den EXIT-Ausgang verlassen.

Natuerlich sind nach dem Erzeugen der entsprechenden EXIT-Meldungen die betreffenden "aktiven Ziele" aus der list-of-goals zu entfernen. Zu diesem Zweck fuehren wir eine neue Funktion <u>exit-boxes</u> ein, die aus der an and-process uebergebenen list-of-

goals fuer alle zu Beginn stehenden "aktiven Ziele" G[1] bis G[n] entsprechende EXIT-Meldungen erzeugt:

```
(def exit-boxes
  (lambda (goals-left environment)
    (cond ((null goals-left) nil)
          ((eq (rest (first goals-left)) 'active)
           (princ "EXIT ")
           (print (ultimate-inst (first goals-left) environment))
           (terpri)
           (exit-boxes (rest goals-left) environment))
          (t goals-left))))
```

Der Wert dieser Funktion exit-boxes ist die um die zu Beginn stehenden "aktiven Ziele" verkuerzte list-of-goals. Mit dieser neuen list-of-goals ("new-list-of-goals") kann dann im Rumpf von and-process wie bisher weitergearbeitet werden.

Nachdem nun die Herleitung der EXIT-Informationen geklaert ist, wenden wir uns nun der Erzeugung der REDO-Informationen zu: Bereits in 6.2. haben wir festgestellt, dass nach dem Verlassen einer Box durch den FAIL-Ausgang genau die Procedure-Boxes durch ihren REDO-Eingang wiederbetreten werden, die unmittelbar zuvor noch ueber ihren EXIT-Ausgang erfolgreich verlassen werden konnten. Liefert also der Aufruf von or-process (bzw. prolog-primitive oder lisp-predicates bei PROLOG-Primitiven oder LISP-Durchgriffen) an den rufenden and-process den Wert nil (fuer Failure) zurueck, so werden damit nicht nur die entsprechende Procedure-Box des nicht beweisbaren Zieles G[n+1] durch den FAIL-Ausgang verlassen, sondern ausserdem die unmittelbar zuvor mit EXIT verlassenen Procedure-Boxes fuer G[n] bis G[1] durch ihren REDO-Eingang wiederbetreten. Auch hier fuehren wir eine weitere Funktion <u>redo-boxes</u> sowie zwei Hilfsfunktionen

```
(def redo-boxes
  (lambda (goals-left environment)
    (redo-boxes-1 goals-left nil environment)))

(def redo-boxes-1
  (lambda (goals-left redolist environment)
    (if (eq (rest (first goals-left)) 'active)
        (redo-boxes-1 (rest goals-left)
                      (cons (first (first goals-left)) redolist)
                      environment)
        (redo-boxes-2 redolist environment))))

(def redo-boxes-2
  (lambda (redolist environment)
    (cond ((null redolist) nil)
          (t (princ "REDO ")
             (print (ultimate-inst (first redolist) environment))
             (terpri)
             (redo-boxes-2 (rest redolist) environment)))))
```

ein, die wiederum mit der urspruenglich an and-process uebergebenen list-of-goals aufgerufen werden. Diese Funktion redo-boxes liefert als Wert stets nil, erzeugt aber fuer alle "aktiven Ziele" am Anfang der list-of-goals die entsprechenden REDO-Meldungen; natuerlich in umgekehrter Reihenfolge als beim Verlassen der Procedure-Boxes mit EXIT.

Mit diesen Erweiterungen fuer die Herleitung der EXIT- und REDO-Informationen erhalten wir nun schliesslich die folgende Version der Funktion and-process:

```
(def and-process
  (lambda (list-of-goals environment level)
    (let ((new-list-of-goals
;;
;; Behandlung von EXIT:
;; --------------------
           (exit-boxes list-of-goals environment)))
      (cond
       ((null new-list-of-goals)
        (print-bindings environment environment)
        (not (y-or-n-p "More? ")))
       (t (let* ((goal (first new-list-of-goals))
                 (goals-left (cons (cons goal 'active)
                                   new-list-of-goals)))
            (princ "CALL ")
            (print (ultimate-inst goal environment))
            (terpri)
            (or (cond
                 ((is-user-defined goal)
                  (or-process (get (first goal) 'clauses)
                              goals-left
                              goal
                              environment
                              level))
                 ((is-primitive goal)
                  (prolog-primitive goals-left
                                    goal
                                    environment
                                    level))
                 ((is-lisp-defined goal)
                  (lisp-predicates goals-left
                                   goal
                                   environment
                                   level)))
                (princ "FAIL ")
                (print (ultimate-inst goal environment))
                (terpri)
;;
;; Behandlung von REDO:
;; --------------------
                (redo-boxes list-of-goals environment)))))))
```

7. Die LISPLOG-Interaktionsumgebung

Mit den in 6.3. (CALL und FAIL) und 6.5. (EXIT und REDO) beschriebenen Aenderungen am Code der Funktion and-process haben wir im Prinzip einen einfachen Box-Modell-Tracer in den bestehenden LISPLOG-Interpreter integriert. Die Abbildung vom LISP-Trace in die elementaren Informationseinheiten des Box-Modells liegt also vor.

Dieser so implementierte Box-Modell-Tracer erfuellt natuerlich noch nicht die in Kapitel 3 beschriebenen Anforderungen. Insbesondere laesst sich bei der bisherigen Realisierung die Ausgabe der Trace-Informationen nicht unterdruecken, und es fehlt vor allem eine Moeglichkeit, den Umfang der auszugebenden Ablaufinformationen flexibel steuern zu koennen, naemlich die Komponente, die wir in Kapitel 3 als Filter fuer die Trace-Informationen bzw. als Debugger bezeichnet haben.

7.1. Die Schnittstelle zum LISPLOG-Interpreter

Natuerlich sollen diese Erweiterungen nicht direkt am Kern des LISPLOG-Interpreters, z.B. an der Funktion and-process, vorgenommen werden, denn damit wuerden vor allem die Chancen fuer eine modulare Gestaltung der Interaktionsumgebung und fuer eine Integration mit anderen Implementationen des LISPLOG-Interpreters, insbesondere mit dem iterativen LISPLOG-Interpreter (vgl. [Dahmen 1986]) vergeben werden. Benoetigt wird also eine Schnittstelle zwischen der Interaktionsumgebung und dem LISPLOG-Interpreter.

Da nun die gesamte Interaktionsumgebung auf den Ablauf-informationen entsprechend dem Box-Modell aufbaut, erscheint es sinnvoll, die angesprochene Schnittstelle bei der Uebergabe dieser elementaren Box-Modell-Informationen (Informationen ueber CALL, REDO, EXIT oder FAIL fuer Procedure-Boxes) an die Interaktionsumgebung anzusiedeln. Dazu sind in der bisherigen Implementation (vgl. S. 37/38) die direkten Ausgaben der Ablaufinformationen, z.B. fuer CALL mittels

```
(princ "CALL ")
(print (ultimate-inst goal environment))
(terpri),
```

durch Aufrufe entsprechender Schnittstellenfunktionen box-call, box-exit, box-fail bzw. box-redo zu ersetzen, die grundsaetzlich als Wert nil liefern sollen, damit die gleiche Wert-Neutralitaet wie in der bisherigen Loesung durch "(terpri)" (s. oben) erreicht wird.

Mit Blick auf die Integration mit dem iterativen LISPLOG-Interpreter wollen wir diese Schnittstellenfunktionen so definieren, dass sie als Argument bereits das instantiierte Ziel

erhalten. [22] Wir erhalten also die folgende Schnittstelle zur Interaktionsumgebung:

```
(box-call <goal>)
(box-exit <goal>)
(box-fail <goal>)
(box-redo <goal>)
```

Dementsprechend muessen wir die Funktionen and-process, exit-boxes und redo-boxes-2 aendern und erhalten:

```
(def and-process
  (lambda (list-of-goals environment level)
    (let ((new-list-of-goals
            (exit-boxes list-of-goals environment)))
      (cond
       ((null new-list-of-goals)
        (print-bindings environment environment)
        (not (y-or-n-p "More? ")))
       (t (let* ((goal (ultimate-inst (first new-list-of-goals)
                                      environment))
                 (goals-left (cons (cons goal 'active)
                                   new-list-of-goals)))
            (box-call goal)
            (or (cond
                 ((is-user-defined goal)
                  (or-process (get (first goal) 'clauses)
                              goals-left
                              goal
                              environment
                              level))
                 ((is-primitive goal)
                  (prolog-primitive goals-left
                                    goal
                                    environment
                                    level))
                 ((is-lisp-defined goal)
                  (lisp-predicates goals-left
                                   goal
                                   environment
                                   level)))
                (box-fail goal)
                (redo-boxes list-of-goals environment)))))))

(def exit-boxes
```

---

[22] Damit ist die Implementation der Interaktionsumgebung unabhaengig von der gewaehlten Repraesentation der Bindungsumgebung ("environment"). Im iterativen Interpreter (LISPLOG.2) werden die Variablenbindungen natuerlich anders repraesentiert als im rekursiv formulierten LISPLOG-Interpreter (LISPLOG.II).

```
  (lambda (goals-left environment)
    (cond
     ((null goals-left) nil)
     ((eq (rest (first goals-left)) 'active)
      (box-exit (ultimate-inst (first goals-left) environment))
      (exit-boxes (rest goals-left) environment))
     (t goals-left))))

(def redo-boxes-2
  (lambda (redolist environment)
    (cond
     ((null redolist) nil)
     (t (box-redo (ultimate-inst (first redolist) environment))
        (redo-boxes-2 (rest redolist) environment)))))
```

Nach diesen Aenderungen koennen wir nun die Interaktionsumgebung im Prinzip unabhaengig von der konkreten Implementation des LISPLOG-Interpreters entwickeln.
Sollte dabei dennoch die Bezugnahme auf die Implementation des LISPLOG-Interpreters (rekursive oder iterative Version?) notwendig werden, so werden wir versuchen, auch dabei wieder von den konkreten Details nach Moeglichkeit zu abstrahieren und weitere moeglichst allgemeine Schnittstellen zwischen Interpreter und Interaktionsumgebung einfuehren.

7.2. Konservierung der Ablaufinformationen fuer Backtrace

Bereits in Kapitel 3 haben wir angesprochen, dass grundsaetzlich alle Ablaufinformationen in einem Backtrace-Stack aufgehoben werden sollen, um so spaeter die Moeglichkeit zu haben, bei einer Unterbrechung des Programmablaufs ueber einen vollstaendigen Backtrace von der aktuellen Situation bis zum Start der LISPLOG-Anfrage zu verfuegen. Beim Backtrace koennen so zur Fehlersuche etwa auch Informationen ueber Praedikate angezeigt werden, fuer die beim urspruenglichen Programmlauf keine Informationen an den Benutzer ausgegeben wurden.

Dazu fuehren wir eine globale Variable backtrace-stack ein, an die zu jeder Zeit die Liste aller bei der Bearbeitung der aktuellen LISPLOG-Anfrage gesammelten Ablaufinformationen (ueber CALL, EXIT, FAIL und REDO von Procedure-Boxes) gebunden ist. Diese Variable muss dann jeweils beim Starten einer LISPLOG-Anfrage mit nil initialisiert werden.

Ausserdem benoetigen wir zur Realisierung der gewuenschten Einrueckung der Ablaufinformationen entsprechend der Tiefe des Beweises (vgl. "Tracer-Ausgabe" auf S. 20) eine weitere globale Variable printlevel, an die die aktuelle Einrueckungstiefe gebunden wird und die beim Starten des LISPLOG-Systems mit Null initialisiert wird.

Demnach koennen wir die Schnittstellenfunktionen box-call, box-exit, box-fail und box-redo als einen Aufruf einer Funktion push-backtrace-stack definieren:

```
(def box-call                                    ;; box-redo analog!
  (lambda (goal)
    (push-backtrace-stack printlevel 'call goal)
    (setq printlevel (add1 printlevel))))
```

bzw.

```
(def box-exit                                    ;; box-fail analog!
  (lambda (goal)
    (setq printlevel (sub1 printlevel))
    (push-backtrace-stack printlevel 'exit goal)))
```

Die Funktion push-backtrace-stack nimmt dann neben der Aktualisierung des backtrace-stack ("push") auch die Ausgabe der Ablaufinformationen an den Benutzer vor: [23], [24]

```
(def push-backtrace-stack
  (lambda (printlevel port goal)
    (setq backtrace-stack
          (cons (list printlevel port goal) backtrace-stack))
    (princ "|")
    (tab printlevel)
    (princ (get-text-for port))
    (print-external goal)
    (terpri)))
```

Mit Hilfe dieser Liste von Ablaufinformationen koennen wir dann im Zuge des weiteren Ausbaus der Interaktionsumgebung schliesslich auch die gewuenschte Backtrace-Moeglichkeit realisieren.
Doch bevor wir darauf im Abschnitt 7.7. eingehen werden, wollen wir zunaechst betrachten, wie der Benutzer Einfluss auf den direkt auszugebenden Teil dieser Informationen (Trace) nehmen kann:

---

[23] Die Funktion get-text-for liefert zu einem Box-Modell-Ein/Ausgang (Port) einen entsprechenden String ("CALL ", "EXIT " usw.), der dann ausgegeben werden kann. Somit ist die externe Darstellung der Box-Modell-Informationen von der internen Darstellung (z.B. im backtrace-stack) unabhaengig und leicht zu veraendern.

[24] Die Funktion print-external druckt die externe Repraesentation einer Form aus. Dadurch werden z.B. LISPLOG-Variablen extern als "_x-5" anstelle "(? x 5)" (interne Repraesentation) dargestellt.

7.3. Selektion der auszugebenden Informationen (Der Debugger)

Bei der bisherigen Loesung werden immer noch alle Ablaufinformationen direkt auf den Monitor bzw. die Standard-Ausgabe ausgegeben. Zur gezielten Steuerung dieser Ausgabe haben wir schon in Kapitel 3 einen Filter vorgesehen, der aus den ueber die Schnittstellen box-call, box-exit, box-fail und box-redo an die Interaktionsumgebung gelieferten Ablaufinformationen gerade diejenigen auswaehlt, die auch direkt ausgegeben werden sollen. Diesen Filter realisieren wir durch eine weitere globale Variable tracemode: An diese Variable ist eine Liste von Praedikatnamen gebunden, so dass dann alle Ablaufinformationen, die sich auf Procedure-Boxes dieser Praedikate beziehen, "den Filter passieren" und direkt ausgegeben werden. Zugleich soll die Bindung tracemode = nil, mit der diese Variable beim Start des LISPLOG-Systems auch initialisiert wird, anzeigen, dass die gesamte Interaktionsumgebung inaktiv ist, wodurch dann eine effizientere Abarbeitung des LISPLOG-Programms moeglich ist. Deshalb werden in diesem Fall auch keine Ablaufinformationen im backtrace-stack konserviert, so dass dieser Fall der inaktiven Interaktionsumgebung direkt in den Schnittstellenfunktionen beruecksichtigt werden kann.

```
(def box-call                                    ;; box-redo analog!
  (lambda (goal)
    (cond (tracemode (push-backtrace-stack printlevel 'call goal)
                     (setq printlevel (add1 printlevel))
                     nil))))
```

bzw.

```
(def box-exit                                    ;; box-fail analog!
  (lambda (goal)
    (cond (tracemode (setq printlevel (sub1 printlevel))
                     (push-backtrace-stack printlevel 'exit goal)
                     nil))))
```

Ausserdem muss vor der Ausgabe der Informationen in der Funktion push-backtrace-stack nun ueberprueft werden, ob das zugehoerige Praedikat - "(first goal)" liefert den Praedikatnamen - in der Liste tracemode enthalten ist:

```
(def push-backtrace-stack
  (lambda (printlevel port goal)
    (setq backtrace-stack
          (cons (list printlevel port goal) backtrace-stack))
    (cond ((memq (first goal) tracemode)
           (princ "|")
           (tab printlevel)
           (princ (get-text-for port))
           (print-external goal)
           (terpri)))))
```

Natuerlich muss der Benutzer die Moeglichkeit haben, den Filter "tracemode" jederzeit veraendern zu koennen. Dazu werden zwei neue Kommandos im LISPLOG-Toplevel (vgl. [Bernardi, Dahmen & Meyer 1987]) bereitgestellt: Das Kommando

```
spy pred1 pred2 ...
```

"weitet" den Filter, indem es die angegebenen Praedikatnamen der Liste tracemode hinzufuegt. Umgekehrt "verengt" das Kommando

```
nospy pred1 pred2 ...
```

den Filter durch Entfernen der angegebenen Praedikatnamen aus der Liste tracemode. Beim Aufruf von "spy" ohne Argumente werden alle benutzerdefinierten Praedikatnamen [25] sowie die "built-in"-Praedikate "is", "not", "var" und "nonvar" in die an tracemode gebundene Liste aufgenommen, beim Aufruf von "nospy" ohne Argumente werden alle Praedikatnamen entfernt, die Interaktionsumgebung wird also "ausgeschaltet".

Zur Verdeutlichung der Wirkung dieser beiden Kommandos "spy" und "nospy" betrachten wir einen Beispieldialog, wie er mit dem bis jetzt realisierten Teil der LISPLOG-Interaktionsumgebung gefuehrt werden koennte: [26]

```
1.<lisplog>                          [Aufruf von LISPLOG aus FRANZ-LISP]
*<consult thronfolger.db>               [Laden der Beispiel-Datenbasis]
[load thronfolger.db]
*<listing>                                   [Auflisten aller Klauseln]
(ass (prinz _x) (frosch _x) (verzaubert _x))
(ass (prinz charles))
(ass (frosch kermit))
(ass (verzaubert kermit))
(ass (erstgeboren charles))
(ass (thronfolger _x) (prinz _x) (erstgeboren _x))
*<spy>                                    [Trace fuer alle Praedikate]
(prinz frosch verzaubert erstgeboren thronfolger is not var
 nonvar)                                                [= tracemode]
*<(thronfolger _gesuchte-person)>                    [LISPLOG-Anfrage]
|CALL (thronfolger _gesuchte-person)
| CALL (prinz _x-1)
|  CALL (frosch _x-2)
```

---

[25] Das LISPLOG-System fuehrt (nicht nur zu diesem Zweck) eine Liste <u>predicates</u> mit den Namen von allen vom Benutzer definierten Praedikaten.

[26] Benutzer-Eingaben sind in spitze Klammern eingeschlossen. Dazu sind rechtsbuendig einige Kommentare eingefuegt.

```
|   EXIT (frosch kermit)
|   CALL (verzaubert kermit)
|   EXIT (verzaubert kermit)
|  EXIT (prinz kermit)
|  CALL (erstgeboren kermit)
|  FAIL (erstgeboren kermit)
|  REDO (prinz kermit)
|   REDO (verzaubert kermit)
|   FAIL (verzaubert kermit)
|   REDO (frosch kermit)
|   FAIL (frosch _x-2)
|  EXIT (prinz charles)
|  CALL (erstgeboren charles)
|  EXIT (erstgeboren charles)
|EXIT (thronfolger charles)
success!
(_gesuchte-person = charles)
More? (y or p) <y>                [Weitere "Loesungen" gewuenscht]
|REDO (thronfolger charles)
|  REDO (erstgeboren charles)
|  FAIL (erstgeboren charles)
|  REDO (prinz charles)
|  FAIL (prinz _x-1)
|FAIL (thronfolger _gesuchte-person)
nil                            [Es gibt keine weiteren "Loesungen"]
*<nospy erstgeboren verzaubert>              [Trace einschraenken]
(prinz frosch thronfolger is not var nonvar)        [= tracemode]
*<(thronfolger _gesuchte-person)>                [LISPLOG-Anfrage]
|CALL (thronfolger _gesuchte-person)
|  CALL (prinz _x-1)
|   CALL (frosch _x-2)
|   EXIT (frosch kermit)
|  EXIT (prinz kermit)
|  REDO (prinz kermit)
|   REDO (frosch kermit)
|   FAIL (frosch _x-2)
|  EXIT (prinz charles)
|EXIT (thronfolger charles)
success!
(_gesuchte-person = charles)
More? (y or p) <n>          [Keine weiteren "Loesungen" gewuenscht]
t                              [= Wert des aeussersten and-process]
*<nospy>                   [Interaktionsumgebung "ausgeschaltet"]
nil                                         [= Wert von tracemode]
*<(thronfolger _gesuchte-person)>                [LISPLOG-Anfrage]
success!
(thronfolger = charles)
More? (y or p) <y>                [Weitere "Loesungen" gewuenscht]
nil                            [Es gibt keine weiteren "Loesungen"]
*<lisp>                          [Verlassen des LISPLOG-Toplevel]
2.                               [Zurueck im FRANZ-LISP-Toplevel]
```

7.4. Moeglichkeiten zur Programmunterbrechung

Bei der Darstellung der Anforderungen an eine Interaktionsumgebung haben wir neben Tracer und Debugger als zweites wichtiges Werkzeug ein Break-Paket mit Moeglichkeiten zum Eingriff in den Programmablauf genannt. Ausserdem haben wir dort bereits zwischen zwei Arten der Programmunterbrechung unterschieden, die beide in der LISPLOG-Interaktionsumgebung realisiert werden sollen:

(1) Unterbrechung beim Erreichen von Breakpoints
(2) Interaktive Programmunterbrechung

Kommen wir zunaechst zur ersten Unterbrechungsart durch zuvor gesetzte Breakpoints. Damit ist gemeint, dass der Programmablauf an vorher genau definierten Stellen (Breakpoints) unterbrochen und erst auf entsprechende Eingabe des Benutzers hin wieder fortgesetzt wird.

Ausserdem soll der Benutzer waehrend einer solchen Unterbrechung die Moeglichkeit haben, allgemeine Kommandos des LISPLOG-Toplevel (vgl. [Bernardi, Dahmen & Meyer 1987]) - z.B. zur Veraenderung des "Filters" tracemode mit den Kommandos "spy" bzw. "nospy" - ausfuehren zu koennen und sich mit speziellen Kommandos einen Ueberblick ueber den aktuellen Zustand der gerade bearbeiteten LISPLOG-Anfrage verschaffen zu koennen.

Die Bearbeitung dieser Break-Kommandos muss dabei innerhalb des aktuellen Aufrufs von and-process bzw. or-process erfolgen, da nur innerhalb dieses Aufrufs die Informationen ueber das gerade bearbeitete (Teil-)Ziel, den aktuellen Pfad im Suchbaum ("aktive Ziele"), die noch zu beweisenden Ziele (list-of-goals) oder die aktuelle Bindungsumgebung vorliegen. [27] Deshalb erfolgt die Bearbeitung dieser Break-Kommandos nicht im LISPLOG-Toplevel, sondern in einer eigenen Kommandoebene: dem LISPLOG-Break-Level.

Da die gesamte LISPLOG-Interaktionsumgebung auf dem Box-Modell bzw. den Informationen ueber das Betreten und Verlassen der entsprechenden Procedure-Boxes aufbaut, erscheint es sinnvoll, auch das Einfuegen von Breakpoints am Box-Modell zu orientieren.

---

[27] Daraus wird bereits klar, dass solche Break-Kommandos bei einer Integration der Interaktionsumgebung in den iterativen LISPLOG-Interpreter nicht oder nicht ohne erheblichen Aufwand verfuegbar sein werden. Innerhalb der rekursiven Implementierung liegen die entsprechenden Informationen aber direkt vor, so dass es sicher sinnvoll ist, wenigstens in der rekursiven Version diese auch dem Benutzer zur Verfuegung zu stellen, auch wenn damit die Leistungen der Interaktionsumgebung nicht mehr unabhaengig von der Implementation des LISPLOG-Interpreters sind.

Konkret geschieht das so, dass der Benutzer grundsaetzlich die Moeglichkeit erhaelt, an jeden beliebigen Port (oder an alle Ports) von jeder beliebigen Procedure-Box einen Breakpoint zu setzen. Beim Durchlaufen des entsprechenden Ports waehrend des Programmablaufs erfolgt dann eine Programmunterbrechung sowie der Aufruf der speziellen Break-Kommandoebene (LISPLOG-Break-Level). Erst nach Eingabe des Break-Kommandos "continue" wird dann die Abarbeitung des LISPLOG-Programms fortgesetzt.

Implementierungstechnisch koennen wir analog zur Variablen tracemode eine weitere globale Variable breakmode einfuehren, an die immer die Liste aller aktuellen Breakpoints gebunden sein soll und die beim Start des LISPLOG-Systems mit nil initialisiert wird. Diese Liste breakmode ist jedoch jetzt keine lineare Liste von Praedikatnamen mehr (wie bei tracemode), sondern enthaelt fuer jedes Praedikat, fuer das mindestens ein Breakpoint gesetzt ist, einen Eintrag der Form

    (<Praedikatname> <Port1> <Port2> ...).

Falls dabei fuer ein Praedikat an allen Ports der zugehoerigen Procedure-Box ein Breakpoint gesetzt ist, so soll die Darstellung "(<Praedikatname> call exit fail redo)" durch die Form

    (<Praedikatname> . all)

abgekuerzt werden.

Zum Setzen und Loeschen von Breakpoints fuehren wir ebenfalls analog zu "spy" und "nospy" zwei neue Kommandos "brk" und "nobrk" ein: Mit dem Kommando

    brk box-ports-form1 box-ports-form2 ...

werden Breakpoints gesetzt, wobei jede box-ports-form ein Ausdruck der oben dargestellten Form ist. Dabei kann allerdings ein Ausdruck "(<Praedikatname> . all)" einfach durch den Praedikatnamen abgekuerzt werden, falls an allen Ports einer zugehoerigen Procedure-Box Breakpoints gesetzt werden sollen. Der Aufruf von "brk" ohne Argumente setzt dann Breakpoints an alle Ports von allen benutzerdefinierten Praedikaten. Entsprechend koennen mit dem Kommando

    nobrk box-ports-form1 box-ports-form2 ...

Breakpoints geloescht werden, und ein Aufruf von "nobrk" ohne Argumente loescht dann dementsprechend alle Breakpoints. Waehrend allerdings mit dem Kommando "brk" ohne Argumente nur Breakpoints an Ports von Procedure-Boxes der benutzerdefinierten Praedikate gesetzt werden, werden mit dem Kommando "nobrk" ohne Argumente grundsaetzlich alle Breakpoints geloescht, was in der Implementation einfach durch Bindung von breakmode an nil erfolgt.

Die Integration des Break-Pakets in die bisher vorgestellte Interaktionsumgebung erfolgt sehr einfach durch eine Erweiterung an den Schnittstellenfunktionen box-call, box-exit, box-fail und box-redo:

```
(def box-call                                   ;; box-redo analog!
  (lambda (goal)
    (cond (tracemode (push-backtrace-stack printlevel 'call goal)
                     (setq printlevel (add1 printlevel))
                     (if (break-activated-p goal 'call)
                         (setq breakflag t))
                     (breakpoint)))))

(def box-exit                                   ;; box-fail analog!
  (lambda (goal)
    (cond (tracemode (setq printlevel (sub1 printlevel))
                     (push-backtrace-stack printlevel 'exit goal)
                     (if (break-activated-p goal 'exit)
                         (setq breakflag t))
                     (breakpoint)))))
```

Mit dem Praedikat break-activated-p

```
(def break-activated-p
  (lambda (goal port)
    (let ((ports (rest (assoc (first goal) breakmode))))
         (or (eq ports 'all) (memq port ports)))))
```

wird dabei jeweils abgetestet, ob an dem gerade durchlaufenen Port ein Breakpoint gesetzt ist. In diesem Fall wird in den Funktionen box-call, box-exit, box-fail und box-redo eine weitere globale Variable breakflag auf den Wert t gesetzt. [28] Diese Variable wird jeweils beim Start einer LISPLOG-Anfrage mit dem Wert nil initialisiert. Die Funktion breakpoint

```
(def breakpoint
  (lambda nil
    (if breakflag (break-level))))
```

schliesslich testet, ob die Variable breakflag inzwischen ge-

---

[28] Diese Implementierung mag dem Leser zunaechst recht umstaendlich erscheinen: Warum z.B. wird nicht direkt die Funktion break-level aufgerufen, sondern zunaechst die Variable breakflag auf t gesetzt, dieser Zustand anschliessend in der Funktion breakpoint nochmals ueberprueft und dann erst die Funktion break-level aufgerufen? Der eigentliche Grund fuer die Einfuehrung der Variablen breakflag wird naemlich erst bei der Realisierung einer interaktiven Unterbrechungsmoeglichkeit ersichtlich, wo dann durch Bindung der Variablen breakflag an t eine Programmunterbrechung vorgemerkt wird.

setzt, d.h. an einen Wert ungleich nil gebunden wurde. In diesem Fall erfolgt ein Aufruf der parameterlosen Funktion break-level, die die bereits angesprochene Kommandoebene LISPLOG-Break-Level realisiert.

Beim Verlassen des LISPLOG-Break-Level wird dann in der Funktion break-level die globale Variable breakflag wieder an den Wert nil gebunden. Da so der Funktionswert von break-level immer nil ist, liefern somit die Funktionen box-call, box-exit, box-fail und box-redo auch weiterhin stets als Wert nil zurueck, wie das fuer die beschriebene Integration in den Kern des LISPLOG-Interpreters (and-process) notwendig ist.

Sicherlich ist es fuer manche praktische Anwendung sinnvoll, z.B. zur Fehlersuche in einem LISPLOG-Programm zunaechst einmal auf die Ausgabe von Ablaufinformationen (Trace) ganz zu verzichten und lediglich Breakpoints an die kritischen Ports (z.B. die FAIL-Ausgaenge) der interessierenden Praedikate zu setzen. Bei der bisher beschriebenen Realisierung waeren dann die Variable tracemode an nil und breakmode an die Liste der interessierenden Ports gebunden. In diesem Fall wuerde jedoch ueberhaupt keine Programmunterbrechung erfolgen, da durch die Bindung von tracemode an nil die gesamte Interaktionsumgebung inaktiv ("ausgeschaltet") ist.

Das Problem besteht also darin, dass wir einerseits aus Gruenden der Effizienz ein leicht auszuwertendes Kriterium fuer die Aktivierung der Interaktionsumgebung haben moechten (hier: Test, ob tracemode ungleich nil), andererseits aber auch eben diesen Fall beruecksichtigen muessen, dass zwar kein Preadikatname in tracemode eingetragen ist, aber einige Breakpoints gesetzt sind und somit die Interaktionsumgebung sehr wohl aktiviert ist.

Deshalb erweitern wir den Wertebereich fuer tracemode um den Wert "break-only", der genau diese Situation abdeckt: Da nun tracemode ungleich nil ist, bleibt auch die Interaktionsumgebung aktiv. Andererseits liefert aber der Test

```
(memq (first goal) tracemode)
```

in der Funktion push-backtrace-stack (s.o.) fuer alle Praedikate nach wie vor den gewuenschten Wert nil, so dass also zunaechst keine Ausgabe von Ablaufinformationen erfolgt. Entscheidend ist aber, dass durch die Aktivierung der Interaktionsumgebung dennoch alle anfallenden Ablaufinformationen im backtrace-stack gesammelt werden und somit beim Erreichen eines Breakpoints oder bei einer direkten Programmunterbrechung durch den Benutzer zur Verfuegung stehen.

Bevor wir jedoch darauf eingehen werden, in welcher Weise der Benutzer im LISPLOG-Break-Level auf diese angesammelten Ablaufinformationen sowie andere Informationen ueber den aktuellen Zustand des LISPLOG-Programms zugreifen kann, muessen wir zunaechst noch klaeren, wie neben der Programmunterbrechung mittels zuvor

gesetzter Breakpoints auch die ebenfalls bereits angesprochene Programmunterbrechnung durch ein Interrupt-Signal des Benutzers realisiert werden kann:

Dazu gibt es in FRANZ-LISP die Moeglichkeit, dem Standard-Interrupt-Signal CTRL-C [29], das normalerweise zum FRANZ-LISP-Toplevel zuruecksetzt, eine beliebige parameterlose Funktion zuzuordnen, die dann beim Eintreffen dieses Interrupt-Signals automatisch aufgerufen wird. [30] Davon wollen wir Gebrauch machen und ordnen dem Standard-Interrupt-Signal eine noch zu definierende Funktion lisplog-interrupt zu:

```
(signal 2 'lisplog-interrupt)
```

Beim Eintreffen des Interrupt-Signals CTRL-C wird nun automatisch diese Funktion lisplog-interrupt aufgerufen, die den Benutzer schliesslich in den LISPLOG-Break-Level fuehren soll, wo er die Moeglichkeit hat, sich neben den im backtrace-stack gesammelten Ablaufinformationen auch andere Informationen ueber den Zustand des LISPLOG-Systems bzw. des gerade unterbrochenen LISPLOG-Programms zu beschaffen. Hier koennen dann u.a. auch die aktuelle Bindungsumgebung (environment) inspiziert oder gar beliebige LISP-Ausdruecke in der aktuellen Umgebung evaluiert werden.

Da aber das Interrupt-Signal zu jedem beliebigen Zeitpunkt eintreffen kann, waere es sicher unguenstig, jeweils sofort die Abarbeitung des laufenden LISPLOG-Programms zu unterbrechen und den LISPLOG-Break-Level aufzurufen. [31]
Sinnvoller ist es dagegen, auch diese interaktiven Programmunterbrechungen durch Interrupts genau wie die durch Breakpoints ausgeloesten Unterbrechungen auf die Zeitpunkte des Durchlaufens von beliebigen Ein- oder Ausgaengen im Box-Modell zu beschraenken. Das entspricht auch den bisherigen Darstellungen, dass die Informationen ueber das Betreten oder Verlassen der "Procedure-Boxes" die elementaren Informationseinheiten fuer die Interaktionsumgebung darstellen.
Dementsprechend wird in der Funktion lisplog-interrupt also eine Programmunterbrechung noch nicht direkt vorgenommen, sondern fuer

---

[29] Die Abkuerzung CTRL-C steht hier fuer das durch gleichzeitige Betaetigung der Tasten "CTRL" und "C" erzeugte Signal.

[30] vgl. [Wilensky 1984], S. 269

[31] Es ist wohl leicht vorzustellen, dass eine Inspektion der aktuellen Bindungsumgebung nach einem Interrupt inmitten eines Unifikationsvorgangs oder einer Variablenumbenennung viel eher verwirren als informieren duerfte.

den Zeitpunkt des naechsten Durchlaufens eines Box-Modell-Ports lediglich "vorgemerkt" und der Programmablauf zunaechst durch den Aufruf von "(evalframe nil)" [32] fortgesetzt.

Aus diesem Grund hatten wir bereits bei der Realisierung der Programmunterbrechungen durch Breakpoints die Variable breakflag eingefuehrt. Die Bindung von breakflag an den Wert t entspricht naemlich gerade dem "Vormerken" einer Programmunterbrechung fuer den naechsten Durchgang durch einen beliebigen Box-Modell-Port.

```
(def lisplog-interrupt
  (lambda nil
    (setq breakflag t)
    (terpr)
    (evalframe nil)))
```

Betrachten wir nun nochmals die Funktionen box-call, box-exit usw. (s. S. 48), dann erkennen wir auch, dass jeweils beim Aufruf dieser Schnittstellenfunktionen, also gerade beim Durchlaufen eines Box-Modell-Ports, ueber die Variable breakflag eine Unterbrechung ausgeloest werden kann. Dabei ist es unerheblich, ob das zugehoerige Praedikat in der an tracemode gebundenen Liste enthalten ist. Wichtig ist nur, dass die Interaktionsumgebung ueberhaupt aktiv (tracemode ungleich nil) ist und somit ueberhaupt Ablaufinformationen fuer den LISPLOG-Break-Level zur Verfuegung stehen.

Diese Loesung birgt aber nun noch einen unangenehmen Nebeneffekt: Falls naemlich die Interaktionsumgebung inaktiv ist, so gibt es bei der bisher beschriebenen Realisierung keine Moeglichkeit mehr, ein laufendes LISPLOG-Programm abzubrechen, ohne dass das gesamte LISPLOG-System anschliessend erneut geladen werden muss, wodurch u.U. Aenderungen an Programm oder Datenbasis verloren waeren. Deshalb ist es wuenschenswert, ausser direkt bei der Bearbeitung einer LISPLOG-Anfrage mit aktiver Interaktionsumgebung einen anderen Interrupt-Handler zur Verfuegung zu haben, der den Benutzer direkt auf den LISPLOG-Toplevel zuruecksetzt. Deshalb wird beim Start von LISPLOG durch "(signal 2 'lisplog-reset)" dem Interrupt-Signal CTRL-C zunaechst die Funktion lisplog-reset zugeordnet:

```
(def lisplog-reset
  (lambda nil
    (terpr)
    (princ "RESET to LISPLOG-Toplevel")
    (terpr)
    (lisplog)))
```

Nur fuer den Zeitraum der Bearbeitung einer LISPLOG-Anfrage bei

---

[32] vgl. [Wilensky 1984], S. 363

aktiver Interaktionsumgebung wird dann dieser Interrupt-Handler lisplog-reset durch die Funktion lisplog-interrupt ersetzt. Die Bearbeitung dieser LISPLOG-Anfrage durch die Funktion and-process (beim rekursiven Interpreter, vgl. S. 21) bzw. "prove" (beim iterativen Interpreter, vgl. [Dahmen 1986]) erfolgt dann je nach Aktivierung der Interaktionsumgebung (tracemode) u.U. mit voruebergehender Aenderung des Interrupt-Handlers:

```
(cond (tracemode (signal 2 'lisplog-interrupt)
                 (prove ...) bzw. (and-process ...)
                 (signal 2 'lisplog-reset))
      (t (prove ...) bzw. (and-process ...) ))
```

Damit ist nun auch gewaehrleistet, dass jedes LISPLOG-Programm zu jeder Zeit durch CTRL-C abgebrochen werden kann: Ist die Interaktionsumgebung inaktiv, so besorgt dies der Interrupt-Handler lisplog-reset, ansonsten erfolgt durch die Funktion lisplog-interrupt beim Durchlaufen des naechsten Box-Modell-Ports eine Programmunterbrechung und der Uebergang zum LISPLOG-Break-Level, von dem aus das laufende Programm mit dem Kommando RESET ebenfalls abgebrochen werden kann.

7.5. Die Kommandoebene LISPLOG-Break-Level

Im vorigen Abschnitt wurde bereits angesprochen, dass diese neue Kommandoebene LISPLOG-Break-Level durch die Funktion break-level realisiert werden soll.

Im Prinzip stellt diese Funktion break-level also eine eigene READ-EVAL-PRINT-Schleife dar, wobei bei jedem Durchlauf ein auszuwertender Ausdruck (Break-Kommando) vom Terminal eingelesen (READ) wird, der allerdings eine bestimmte Form haben muss. Nach Bearbeitung (EVAL) dieses Kommandos, was in der Regel zur Ausgabe von Informationen auf dem Bildschirm fuehrt, erfolgt ausser bei den Kommandos RESET und CONTINUE, die ein Verlassen des LISPLOG-Break-Level bewirken, die Ausgabe (PRINT) des von der zugehoerigen LISP-Funktion zurueckgegebenen Wertes und einer Aufforderung zur Eingabe eines weiteren Break-Kommandos sowie ein erneuter Aufruf der READ-EVAL-PRINT-Schleife break-level.

Auf die konkrete Implementation soll hier an dieser Stelle nicht weiter eingegangen werden, da sich hieran kaum wesentliche Konzepte und Entwurfsentscheidungen verdeutlichen lassen. [33]

---

[33] Der interessierte Leser sei hier auf Anhang B verwiesen, in dem sich auch die vollstaendigen Listings der gesamten Interaktionsumgebung befinden.

An dieser Stelle wollen wir uns vielmehr auf die Beschreibung der Break-Kommandos konzentrieren, die im LISPLOG-Break-Level zur Verfuegung stehen:

Dazu gehoeren neben den vor allem im Benutzerhandbuch [Bernardi, Dahmen & Meyer 1987] beschriebenen Toplevel-Kommandos wie "ass", "consult", "tell", "spy", "brk" usw. auch spezielle Kommandos zum Blaettern in den im backtrace-stack gesammelten Informationen. Diese Kommandos ("+P", "-P" bzw. Anzahl der Schritte) werden noch ausfuehrlich in Abschnitt 7.7 beschrieben.
Ausserdem hat der Benutzer im LISPLOG-Break-Level darueberhinaus noch eine Reihe weiterer Break-Kommandos zur Verfuegung, um sich einen Ueberblick ueber den aktuellen Zustand des gerade bearbeiteten LISPLOG-Programms zu verschaffen:

- Das Kommando EVAL <form> wertet einen beliebigen Lisp-Ausdruck ("S-Ausdruck") <form> in der aktuellen Laufzeitumgebung aus und druckt den entsprechenden Wert.

- Das Kommando INFO liefert Informationen ueber die Interaktionsumgebung, also die Bindungen der globalen Variablen tracemode, breakmode, skipmode und skipqmode [34] sowie cutmode [35].

- Das Kommando ENVIRONMENT [36] liefert die aktuelle Bindungsumgebung in einer benutzerfreundlichen Darstellung (vgl. S. 24).

- Das Kommando PATH [36] gibt den aktuellen Pfad im Suchbaum vom aktuellen Ziel bis zur urspruenglichen LISPLOG-Anfrage aus. Dieser Pfad ergibt sich als Liste der augenblicklich "aktuellen Ziele" aus der Variablen goals-left.

- Das Kommando LIST-OF-GOALS [36] stellt die Liste der noch zu beweisenden (Teil-)Ziele dar. Das sind gerade alle Eintraege in der Variablen goals-left, die nicht als "aktiv" markiert sind.

- Das Kommando GOAL [36] schliesslich zeigt das gerade zu beweisende (Teil-)Ziel an.

---

[34] Zur Bedeutung dieser Variablen vgl. Abschnitt 7.6.

[35] Zur Bedeutung der Variablen cutmode vgl. Abschnitt 7.8.

[36] Wegen der unterschiedlichen Repraesentation der Bindungsumgebung (environment) sowie der aktuellen Resolvente (list-of-goals) im iterativen Interpreter LISPLOG.2 (vgl. [Dahmen 1986]) sind die Kommandos ENVIRONMENT, PATH, LIST-OF-GOALS und GOAL zur Zeit nur in der rekursiven Version LISPLOG.II verfuegbar. Grundsaetzlich lassen sich diese Kommandos aber auch fuer LISPLOG.2 realisieren.

- Mit dem bereits angesprochenen Kommando CONTINUE wird die zuvor unterbrochene LISPLOG-Abarbeitung wieder fortgesetzt.

- Das Kommando RESET dagegen bewirkt den Abbruch des LISPLOG-Programms und die Rueckkehr zum LISPLOG-Toplevel.

- Ausserdem gibt es noch ein weiteres Break-Kommando SKIPPING, auf dessen Bedeutung wir noch im naechsten Abschnitt ueber die Ausblendung von Teilberechnungen eingehen werden.

- Im Zusammenhang mit dem Handschneider (Abschnitt 7.8.) oder dem interaktiven Skipper (Abschnitt 7.6.) ist es auch moeglich, den LISPLOG-Break-Level durch Eingabe einer Antwort (JA oder NEIN) auf eine noch ausstehende Frage des Systems, z.B. nach dem Einfuegen eines temporaeren Cuts, zu verlassen.

Zur Vereinfachung der Handhabung dieser Break-Kommandos sind fuer die einzelnen Kommandos Abkuerzungen zugelassen. Die folgende Tabelle gibt eine Uebersicht ueber die zulaessigen Abkuerzungen:

Break-Kommando:	zulaessige Abkuerzungen:
INFO	i I in IN inf INF
CONTINUE	c C co CO con CON cont CONT
RESET	r R re RE res RES
SKIPPING	s S
EVAL	v V
ENVIRONMENT	e E en EN env ENV
PATH	pa PA
LIST-OF-GOALS	L li LI lis LIS list LIST
GOAL	g G go GO goa GOA
(positive Antwort)	y Y yes Yes YES j J ja Ja JA
(negative Antwort)	n N no No NO nein Nein NEIN

Zur Veranschaulchung der Wirkung der beiden neuen Kommandos "brk" und "nobrk" sowie der interaktiven Moeglichkeiten im LISPLOG-Break-Level betrachten wir nochmals einen Beispieldialog, dem zunaechst wieder unsere Standard-Beispiel-Datenbasis

```
(3)   (ass (prinz _x) (frosch _x) (verzaubert _x))
      (ass (prinz charles))
      (ass (frosch kermit))
      (ass (verzaubert kermit))
      (ass (erstgeboren charles))
      (ass (thronfolger _x) (prinz _x) (erstgeboren _x))
```

zugrundeliegt. Im zweiten Teil des folgenden Beispieldialogs wird dann von der Moeglichkeit Gebrauch gemacht, jederzeit auch bei vorlaeufigem Verzicht auf die Ausgabe von Ablaufinformationen die LISPLOG-Abarbeitung durch Eingabe von CTRL-C unterbrechen zu koennen. Das LISPLOG-Programm laeuft also zunaechst genauso ab

wie bei voellig inaktiver Interaktionsumgebung. Erst bei einer Unterbrechung durch den Benutzer (mit CTRL-C), z.B. beim Verdacht einer Endlos-Rekursion, macht sich der Vorteil dieser Vorgehensweise bemerkbar: Durch die im backtrace-stack gespeicherten Ablaufinformationen kann sich der Benutzer jetzt im LISPLOG-Break- Level ein genaues Bild vom aktuellen Zustand des LISPLOG-Programms machen und einer eventuell tatsaechlich vorliegenden Endlos-Rekursion auf die Spur zu kommen. So ist es dann auch leicht, den Fehler in der folgenden nur partiell korrekten Formulierung zur Fakultaetsberechnung

```
(4)   (ass (fak 0 1))
      (ass (fak _n _nfak) (is _n1 (sub1 _n))
                          (fak _n1 _n1fak)
                          (is _nfak (times _n1fak _n)))
```

zu finden, der bei der Anforderung weiterer Loesungen (nach der korrekt gelieferten Fakultaet) zu einer Endlos-Rekursion fuehrt:

```
1.<lisplog>                        [Aufruf von LISPLOG aus FRANZ LISP]
*<consult thronfolger.db>              [Laden der Beispiel-Datenbasis]
[load thronfolger.db]
(thronfolger.db)
*<listing>                                  [Auflisten aller Klauseln]
(ass (prinz _x) (frosch _x) (verzaubert _x))
(ass (prinz charles))
(ass (frosch kermit))
(ass (verzaubert kermit))
(ass (erstgeboren charles))
(ass (thronfolger _x) (prinz _x) (erstgeboren _x))
*<spy>                                    [Trace fuer alle Praedikate]
(prinz frosch verzaubert erstgeboren thronfolger is not var
 nonvar)                                               [= tracemode]
*<brk verzaubert>      [Breakpoint an alle Ports von "verzaubert"]
((verzaubert . all))
*<(thronfolger _wer)>                                [LISPLOG-Anfrage]
|CALL (thronfolger _wer)
| CALL (prinz _x-1)
|  CALL (frosch _x-2)
|  EXIT (frosch kermit)
|  CALL (verzaubert kermit)
***** LISPLOG - Break - Level *****          [Breakpoint erreicht]
Please enter Break-command: <GOAL> [Anzeigen des aktuellen Ziels]
The actual goal is: (verzaubert kermit)
Please enter Break-command: <LIST> [Anzeigen der aktuellen Ziele]
level  active goal              more premises
-----------------------------------------------------------------
0      (thronfolger _wer)       *NONE*
1      (prinz _x-1)             (erstgeboren _x-1)
2      (verzaubert kermit)      *NONE*
Please enter Break-command: <PATH> [Anzeigen des aktuellen Pfads]
Path = (verzaubert kermit) -> (prinz _x-1) -> (thronfolger _wer)
Please enter Break-command: <ENVIRONMENT> [Anzeige der Bindungen]
```

```
Environment = {_x-2/kermit, _x-1/_x-2, _wer/_x-1}
Please enter Break-command: <nospy erstgeboren>
(prinz frosch verzaubert thronfolger is not var nonvar)
Please enter Break-command: <INFO>
spy active for: (prinz frosch verzaubert thronfolger is not var
                 nonvar)
brk active for: ((verzaubert . all))
Please enter Break-command: <nobrk (verzaubert exit fail redo)>
((verzaubert call))                                 [= breakmode]
Please enter Break-command: <brk frosch>    [Breakpoints setzen]
((frosch . all) (verzaubert call))                  [= breakmode]
Please enter Break-command: <INFO>
spy active for: (prinz frosch verzaubert thronfolger is not var
                 nonvar)
brk active for: ((frosch . all) (verzaubert call))
Please enter Break-command: <CONTINUE>    [Programm fortsetzen]
LISPLOG continues...
|  EXIT (verzaubert kermit)
| EXIT (prinz kermit)
| REDO (prinz kermit)
|  REDO (verzaubert kermit)
|  FAIL (verzaubert kermit)
|  REDO (frosch kermit)
***** LISPLOG - Break - Level *****    [Naechster Breakpoint]
Please enter Break-command: <GOAL>
The actual goal is: (frosch _x-2)
Please enter Break-command: <PATH>
Path = (frosch _x-2) -> (prinz _x-1) -> (thronfolger _wer)
Please enter Break-command: <LIST>
level  active goal            more premises
-----------------------------------------------------------------
0      (thronfolger _wer)     *NONE*
1      (prinz _x-1)           (erstgeboren _x-1)
2      (frosch _x-2)          (verzaubert _x-2)
Please enter Break-command: <listing prinz>  [Toplevel-Kommando]
(ass (prinz _x) (frosch _x) (verzaubert _x))
(ass (prinz charles))
Please enter Break-command: <CONTINUE>
LISPLOG continues...
|  FAIL (frosch _x-2)
***** LISPLOG - Break - Level *****    [Ausgang von "frosch"]
Please enter Break-command: <C>                  [= CONTINUE]
LISPLOG continues...
| EXIT (prinz charles)
|EXIT (thronfolger charles)
success
(_wer  =  charles)
More? (y or n) <y>          [Weitere Loesungen gewuenscht]
|REDO (thronfolger charles)
| REDO (prinz charles)
| FAIL (prinz _x-1)
|FAIL (thronfolger _wer)
nil                [Keine weiteren Loesungen vorhanden]
```

```
*<consult fak.db>                   [Laden der Fakultaet-Datenbasis]
[load fak.db]
(fak.db)
*<listing fak>       [Anzeigen aller Klauseln zum Praedikat "fak"]
(ass (fak 0 1))
(ass (fak _n _nfak)
     (is _n1 (sub1 _n))
     (fak _n1 _n1fak)
     (is _nfak (times _n1fak _n)))
*<nospy>              [Keine Ablaufinformationen auf den Bildschirm]
break-only                [Interaktionsumgebung ist noch aktiviert!]
*<(fak 3 _x)>    [LISPLOG-Anfrage: "Was ist die Fakultaet von 3?"]
success
(_x  =  6)                                  [korrekte Loesung: 6]
More? (y or n) <y>                  [Weitere Loesungen gewuenscht]
 .
 .
 .
 .
 .                                [Programm laeuft und laeuft...]
 .
 .
 .
 .
^C                             [Interaktive Programmunterbrechung]
***** LISPLOG - Break - Level *****
Please enter Break-command: <GOAL>          [Wo steckt das System?]
The actual goal is: (is -4 (sub1 -3)) [Wieso negative Argumente?]
Please enter Break-command: <PATH> [Wie kommt es zu diesem Ziel?]
Path = (is _n1-7 (sub1 -3)) -> (fak -3 _n1fak-6) ->
-> (fak -2 _n1fak-5) -> (fak -1 _n1fak-4) -> (fak 0 _n1fak-3) ->
-> (fak 1 _n1fak-2) -> (fak 2 _n1fak-1) -> (fak 3 _x)
Please enter Break-command: <LIS> [Welche Ziele stehen noch aus?]
level  active goal            more premises
-----------------------------------------------------------------
0      (fak 3 _x)             *NONE*
1      (fak 2 _n1fak-1)       (is _nfak-1 (times _n1fak-1 _n-1))
2      (fak 1 _n1fak-2)       (is _nfak-2 (times _n1fak-2 _n-2))
3      (fak 0 _n1fak-3)       (is _nfak-3 (times _n1fak-3 _n-3))
4      (fak -1 _n1fak-4)      (is _nfak-4 (times _n1fak-4 _n-4))
5      (fak -2 _n1fak-5)      (is _nfak-5 (times _n1fak-5 _n-5))
6      (fak -3 _n1fak-6)      (is _nfak-6 (times _n1fak-6 _n-6))
7      (is _n1-7 (sub1 -3))   (fak _n1-7 _n1fak-7)
                              (is _nfak-7 (times _n1fak-7 _n-7))
      [Hier erkennt man klar die (endlose) Rekursion in "fak"!]
Please enter Break-command: <RESET>      [Abbruch des Programms]
user-break:reset-to-LISPLOG-Top-Level
*<spy>                   [Noch einmal mit eingeschaltetem Trace]
(fak is not var nonvar)
*<(fak 3 _x)>                       [Die gleiche LISPLOG-Anfrage]
|CALL (fak 3 _x)
| CALL (is _n1-1 (sub1 3))
| EXIT (is 2 (sub1 3))
```

```
| CALL (fak 2 _n1fak-1)
|  CALL (is _n1-2 (sub1 2))
|  EXIT (is 1 (sub1 2))
|  CALL (fak 1 _n1fak-2)
|   CALL (is _n1-3 (sub1 1))
|   EXIT (is 0 (sub1 1))          [Anzeige von "is" stoert im Trace]
^C                               [Interaktive Programmunterbrechung]
***** LISPLOG - Break - Level *****
Please enter Break-command: <nospy is> ["is" nicht mehr anzeigen]
(fak not var nonvar)                               [= tracemode]
Please enter Break-command: <CONTINUE>   [Abarbeitung fortsetzen]
LISPLOG continues...
|   CALL (fak 0 _n1fak-3)
|   EXIT (fak 0 1)
|  EXIT (fak 1 1)
| EXIT (fak 2 2)
|EXIT (fak 3 6)
success
(_x  =  6)
More? (y or n) <y> [Beim Anfordern weiterer Loesungen passiert's]
|REDO (fak 3 _x)
| REDO (fak 2 _n1fak-1)
|  REDO (fak 1 _n1fak-2)
|   REDO (fak 0 _n1fak-3)
|    CALL (fak -1 _n1fak-4)
|     CALL (fak -2 _n1fak-5)
|      CALL (fak -3 _n1fak-6)
|       CALL (fak -4 _n1fak-7)
|        CALL (fak -5 _n1fak-8)
|         CALL (fak -6 _n1fak-9)
|          CALL (fak -7 _n1fak-10)    [Tatsaechlich: Das ist die]
|           CALL (fak -8 _n1fak-11)   [Endlos-Rekursion in "fak"]
|            CALL (fak -9 _n1fak-12)
|             CALL (fak -10 _n1fak-13)
|              CALL (fak -11 _n1fak-14)
|               CALL (fak -12 _n1fak-15)
|                CALL (fak -13 _n1fak-16)
|                 CALL (fak -14 _n1fak-17)
|                  CALL (fak -15 _n1fak-18)
|                   CALL (fak -16 _n1fak-19)
|                    CALL (fak -17 _n1fak-20)
|                     CALL (fak -18 _n1fak-21)
|                      CALL (fak -19 _n1fak-22)
|                       CALL (fak -20 _n1fak-23)
|                        CALL (fak -21 _n1fak-24)
^C                                           [Unterbrechung...]
***** LISPLOG - Break - Level *****
Please enter Break-command: <RESET>           [...und Abbruch!]
user-break:reset-to-LISPLOG-Top-Level
*<lisp>                           [Verlassen des LISPLOG-Toplevel]
2.                                [Zurueck im FRANZ-LISP-Toplevel]
```

7.6. Ausblenden von Teilberechnungen

Bei der Ablaufverfolgung groesserer LISPLOG-Programme kann es manchmal sehr sinnvoll sein, interaktiv Teilberechnungen wie den Beweis eines bestimmten Teilzieles oder die Auswertung einer bestimmten LISP-Funktion aus den sonst auf dem Bildschirm auszugebenden Ablaufinformationen auszublenden. Dann erscheinen zwar keine Informationen bezueglich dieser Teilberechnung auf dem Bildschirm, dennoch werden alle anfallenden Ablaufinformationen fuer ein eventuelles spaeteres Blaettern im Trace im backtrace-stack gesammelt. Insbesondere bei LISPLOG-Programmen mit direkt oder indirekt rekursiver Formulierung kann so die auszugebende Informationsmenge interaktiv so gesteuert werden, wie es bisher mit den Kommandos "spy/nospy" und "brk/nobrk" nicht moeglich war.

Dazu fuehren wir ein neues Kommando "skipq" sowie eine neue globale Variable skipqmode ein, so dass analog zum Kommando "spy" beim Aufruf von

```
skipq pred1 pred2 ...
```

die im Kommando angegebenen Praedikat- oder Funktionsnamen pred1, pred2 usw. in die an skipqmode gebundene Liste aufgenommen werden. Die Bedeutung dieser Liste besteht nun darin, dass bei jedem nicht-ausgeblendeten Betreten einer Procedure-Box, deren Name sowohl in tracemode als auch in skipqmode eingetragen ist, zunaechst der Benutzer gefragt wird, ob er vielleicht die Auswertung des entsprechenden PROLOG-Teilzieles bzw. der entsprechenden LISP-Funktion, also alle Ablaufinformationen aus dem "Inneren" der entsprechenden Box [37], ausgeblendet haben moechte. Falls ja, werden alle Ablaufinformationen aus dem "Inneren" der aktuellen Box ausgeblendet. Falls waehrend des Ablaufs der ausgeblendeten Teilberechnung kein Breakpoint erreicht oder das Programm interaktiv (durch CTRL-C) unterbrochen wird, so wird erst mit dem Verlassen der aktuellen Procedure-Box durch den EXIT- bzw. FAIL-Ausgang [38] die Ausgabe der Ablaufinformationen auf den Monitor

---

[37] An dieser Stelle greifen wir schon auf die Ergebnisse von Kapitel 8 vor, das eine Erweiterung des Box-Modells fuer den funktionalen Teil von LISPLOG behandelt: Dort werden analog zu den PROLOG-Praedikaten auch fuer die LISP-Funktionen aehnliche Procedure-Boxes eingefuehrt, so dass jedem Aufruf einer (vom Benutzer definierten) LISP-Funktion auch genau eine solche Box entspricht. Diese hat dann allerdings im allgemeinen keine CALL-, REDO-, FAIL- oder EXIT-Ports, sondern entsprechend einen EVAL-Eingang und einen VALUE-Ausgang.

[38] Bei LISP-Funktionen entsprechend beim Verlassen durch den VALUE-Ausgang.

fortgesetzt. Erfolgt aber innerhalb der ausgeblendeten Berechnung ein Aufruf des LISPLOG-Break-Level, so kann dort mit dem Break-Kommando SKIPPING (vgl. Abschnitt 7.5.) die Ausblendung auf die Abarbeitung des aktuellen Zieles (vgl. Break-Kommando GOAL) eingeschraenkt werden. Allgemein entspricht das SKIPPING-Kommando also einem temporaeren (einmaligen) "skip <aktuelles Ziel>".

Im Laufe der Entwicklung groesserer LISPLOG-Programme kann es nun aber auch sehr sinnvoll sein, zum Beispiel bereits ausgetestete Programmteile (Praedikate oder Funktionen) permanent auszublenden, so dass der Benutzer gar nicht mehr jeweils danach gefragt werden muss, ob die entsprechenden Teilberechnungen ausgeblendet werden sollen. Auf diese Weise werden bei der Ablaufprotokollierung die entsprechenden Programmteile einfach uebersprungen. Die Wirkung ist also die gleiche, als waere fuer die entsprechenden Funktionen und Praedikate das Kommando skipq gegeben und auf die Frage nach dem Ausblenden konsequent mit "ja" geantwortet worden.

Zu diesem Zweck fuehren wir noch ein weiteres Kommando "skip"

    skip pred1 pred2 ...

sowie eine weitere globale Variable skipmode ein, die ganz analog zu "skipq" bzw. skipqmode die Namen der permanent auszublendenden Praedikate und Funktionen verwalten.

Da Kommandos des LISPLOG-Toplevels auch zusammen mit den Klauseln und Funktionsdefinitionen in einer Datei stehen und dann mit "consult" in das LISPLOG-System geladen werden koennen, ist es nun auch tatsaechlich moeglich, z. B. bereits ausgetestete LISPLOG-Programmteile durch entsprechende in die Dateien eingefuegte "skip"-Kommandos vom Debugging auszublenden.

Selbstverstaendlich muessen auch "skip"- und "skipq"-Kommandos wieder rueckgaengig gemacht werden koennen. Dazu fuehren wir noch ein weiteres neues Kommando

    noskip pred1 pred2 ...

ein, das die angegebenen Namen aus den an skipmode und skipqmode gebundenen Listen entfernt. Ein Aufruf von "noskip" ohne Argumente bewirkt dann analog zu "nospy" und "nobrk" das Loeschen von skipmode und skipqmode durch Bindung an nil.

Natuerlich kann ein bestimmter Praedikat- oder Funktionsname nur einmal in skipmode oder skipqmode auftreten. Deshalb werden beim Aufruf des Kommandos "skipq pred1 pred2 ..." in skipqmode nur die Namen pred1, pred2 usw. aufgenommen, die nicht bereits in skipmode enthalten sind. Im Gegensatz dazu werden beim Aufruf von "skip pred1 pred2" alle Namen pred1, pred2 usw. in skipmode aufgenommen. Eventuelle Doppeleintraege werden dann aus skipqmode entfernt. Aehnlich wie bei "spy" und "nospy" gelten Aufrufe von

"skip" oder "skipq" ohne Argumente fuer alle dem LISPLOG-System bekannten Namen von Praedikaten oder benutzerdefinierten Funktionen. Ein Aufruf von "skip" ohne Argumente bindet also skipmode an die Liste aller dem System bekannten Praedikat- oder Funktionsnamen und loescht entsprechend skipqmode durch Bindung an nil. Dadurch wird zu jeder LISPLOG-Anfrage - falls "unterwegs" kein Breakpoint oder Interrupt (durch CTRL-C) erreicht wird - nur das Betreten und Verlassen der direkt zur Anfrage gehoerenden Box bzw. Boxes (bei einer Anfrage mit mehreren Teilzielen) auf dem Bildschirm protokolliert. Dennoch stehen alle bei der Abarbeitung angefallenen Ablaufinformationen bei einer Programmunterbrechung durch Breakpoints oder Interrupts im backtrace-stack zur Verfuegung. Somit laesst sich auch ohne Beeinflussung des Filters tracemode durch das Kommando

```
skip
```

fast der gleiche Effekt wie im Dialogbeispiel zum Abschnitt 7.5. mit der fehlerhaften Fakultaetsberechnung erreichen: Das LISPLOG-Programm laeuft zunaechst (fast) ohne Ablaufprotokollierung auf dem Bildschirm; bei ungewoehnlichem Verhaltem - wie einer vermuteten Endlos-Rekursion - stehen nach einer Programmunterbrechung durch CTRL-C im backtrace-stack [39] und den Datenstrukturen des LISPLOG-Interpreters [40] alle Informationen zur Fehlersuche zur Verfuegung.

7.7. Blaettern im Trace: Stepper und Backtrace

Der Stand der Implementation, wie er bis jetzt beschrieben wurde, weist noch zwei Unzulaenglichkeiten auf, deren Beseitigung Gegenstand dieses Abschnitts sein soll:
Erstens besteht bislang noch keine Moeglichkeit, auf die im backtrace-stack konservierten Ablaufinformationen zuzugreifen.
Ausserdem kann es leicht vorkommen, dass der Benutzer durch die umfangreichen Trace-Informationen, die - z.B. nach dem Kommando "spy" ohne weitere Argumente - entsprechend der Abarbeitungsgeschwindigkeit ueber den Bildschirm flimmern, keine Moeglichkeit mehr hat, den aktuellen Stand der Abarbeitung angemessen mitzuverfolgen, um sich so an den interessanten Stellen z.B. ueber den Break-Level weitere Ablaufinformationen zu beschaffen.

---

[39] Davon konnten wir bisher noch keinen Gebrauch machen. Die Verwendung des backtrace-stack zum Blaettern im Trace ist Gegenstand des naechsten Abschnitts 7.7.

[40] Aus diesen Datenstrukturen (goals-left bzw. environment) extrahieren die Break-Kommandos ENVIRONMENT, GOAL, LIST-OF-GOALS und PATH die entsprechenden Informationen.

Dem soll nun dadurch begegnet werden, dass stets nur eine vom Benutzer steuerbare begrenzte Menge von Trace-Informationen ausgegeben wird und anschliessend wieder auf eine Eingabe des Benutzers gewartet wird.

Die Realisierung dieses Konzepts erfordert nun neben der interaktiven Programmunterbrechung und der Unterbrechung durch Breakpoints, die zur Aktivierung des Break-Level fuehren, eine weitere Unterbrechungsart fuer den Fall, dass das vorgegebene Maximum der direkt auszugebenden Trace-Informationen erreicht ist. In diesem Fall wird der Benutzer gefragt, ob mit der Ausgabe weiterer Trace-Informationen fortgefahren werden soll. Bei dieser Gelegenheit hat er aber auch die Moeglichkeit, die Anzahl der maximal direkt auszugebenden Trace-Informationen zu veraendern.

Die Implementation dieses Konzeptes gestaltet sich recht einfach, indem zwei weitere globale Variablen stepcount und linecount eingefuehrt werden. Dabei enthaelt die Variable stepcount die Anzahl der bis zur naechsten Unterbrechung noch maximal auszugebenden Trace-Informationen (Schritte), waehrend linecount die Anzahl der maximal noch auszugebenden Zeilen enthaelt.
Bei jeder Ausgabe von Trace-Informationen werden dann in den Funktionen <u>line-control</u> bzw. <u>step-control</u> diese Variablen entsprechend dekrementiert. Wird dabei der Wert Null erreicht, so erfolgt die angesprochene Unterbrechung durch einen Aufruf der Funktion linebreak:

```
(def line-control
  (lambda nil
    (cond (tracemode (setq linecount (sub1 linecount))
                     (if (zerop linecount) (linebreak))))))

(def step-control
  (lambda nil
    (cond (tracemode (setq stepcount (sub1 stepcount))
                     (if (zerop stepcount) (linebreak)
                                           (line-control))))))
```

Durch Beschraenkung auf die Ausgabe jeweils einer Trace-Information (stepcount = 1) ist damit auch der in Kapitel 4 bereits angekuendigte schrittweise Abarbeitungsmodus (Stepper) realisiert.

Zu Beginn der LISPLOG-Abarbeitung ist der Wert fuer die maximale Anzahl der direkt auszugebenden Trace-Informationen sinnvollerweise so eingestellt, dass sich die erste Unterbrechung nach dem Fuellen genau einer Bildschirmseite ergibt (linecount = 22 [41]), so dass alle ausgegebenen Informationen noch sichtbar sind.

---

[41] Anzahl der darstellbaren Zeilen minus 2 (bei VT100)

Eine Uebersicht ueber alle bei einer solchen Unterbrechung moeglichen Benutzereingaben gibt die folgende Tabelle:

Eingabe:	Wirkung:
P oder +P	Ausgabe max. einer weiteren Bildschirm-Seite an Trace-Informationen
n (pos. Zahl)	Ausgabe max. n weiterer Schritte
0	Abbruch des LISPLOG-Programms
-P	Zurueckblaettern um max. eine Seite
-m (neg. Zahl)	Zurueckblaettern um max. m Schritte
b	Aufruf des LISPLOG-Break-Level
+ zusaetzlich beliebige Kommandos des LISPLOG-Toplevel!	

Analog zum Vorwaertsblaettern um eine Seite (+P) oder n Schritte bietet sich hier auch die Moeglichkeit, im backtrace-stack in aehnlicher Weise auch seitenweise (-P) oder schrittweise (-m) rueckwaerts zu blaettern.
Auf diese Weise kann nun beliebig in den bisher gesammelten Trace-Informationen geblaettert werden. Reichen dabei beim Vorwaertsblaettern die bisher gesammelten Trace-Informationen nicht aus, so wird einfach die zuvor unterbrochene LISPLOG-Abarbeitung fortgesetzt (Meldung: "LISPLOG continues..."), und die weiteren Trace-Informationen werden ausgegeben.

Selbstverstaendlich sind diese Moeglichkeiten zum Blaettern im Trace auch im Break-Level verfuegbar, wodurch es ja auch erst moeglich wird, durch eine Veraenderung des Filters tracemode beim Backtrace Informationen ausgeben zu lassen, die beim eigentlichen Ablauf des LISPLOG-Programms zuvor nicht auf dem Monitor ausgegeben wurden. In diesem Sinne ist dann die Tabelle der Break-Kommandos auf Seite 54 um obige Tabelle zu erweitern.

Zur Verdeutlichung des dargestellten Konzeptes soll auch das abschliessende Dialogbeispiel dienen, dem wieder unsere Standard-Beispiel-Datenbasis zugrundeliegt. Damit aber ueberhaupt mehr als eine Bildschirm-Seite an Trace-Informationen anfaellt, fuegen wir noch ein paar weitere Fakten hinzu, die lediglich den Suchraum fuer die nach wie vor eindeutig bestimmte Loesung (charles) vergroessern:

```
(5)   (ass (prinz _x) (frosch _x) (verzaubert _x))
      (ass (prinz andrew))
      (ass (prinz philip))
      (ass (prinz charles))
      (ass (frosch fridolin))
      (ass (frosch kermit))
      (ass (verzaubert kermit))
      (ass (erstgeboren charles))
      (ass (thronfolger _x) (prinz _x) (erstgeboren _x))
```

```
1.<lisplog>                          [Aufruf von LISPLOG aus FRANZ-LISP]
*<consult thronfolger.neu.db>       [Laden der Beispiel-Datenbasis]
[load thronfolger.neu.db]
*<listing>                                [Auflisten aller Klauseln]
(ass (prinz _x) (frosch _x) (verzaubert _x))
(ass (prinz andrew))
(ass (prinz philip))
(ass (prinz charles))
(ass (frosch fridolin))
(ass (frosch kermit))
(ass (verzaubert kermit))
(ass (erstgeboren charles))
(ass (thronfolger _x) (prinz _x) (erstgeboren _x))
*<spy prinz frosch erstgeboren thronfolger>  [Tracer einschalten]
(prinz frosch erstgeboren thronfolger is var nonvar)
*<(thronfolger _gesuchte-person)>                  [LISPLOG-Anfrage]
|CALL (thronfolger _gesuchte-person)
| CALL (prinz _x-1)
|  CALL (frosch _x-2)
|  EXIT (frosch fridolin)
|  REDO (frosch fridolin)
|  EXIT (frosch kermit)
| EXIT (prinz kermit)
| CALL (erstgeboren kermit)
| FAIL (erstgeboren kermit)
| REDO (prinz kermit)
|  REDO (frosch kermit)
|  FAIL (frosch _x-2)
| EXIT (prinz andrew)
| CALL (erstgeboren andrew)
| FAIL (erstgeboren andrew)
| REDO (prinz andrew)
| EXIT (prinz philip)
| CALL (erstgeboren philip)
| FAIL (erstgeboren philip)
| REDO (prinz philip)
| EXIT (prinz charles)
| CALL (erstgeboren charles)
More information desired? Enter +p, -p or number of steps: <b>
***** LISPLOG - Break - Level *****       [Aufruf des Break-Level]
Please enter Break-command: <GOAL> [Anzeigen des aktuellen Ziels]
The actual goal is: (erstgeboren charles)
Please enter Break-command: <spy verzaubert>   [Filter ausweiten]
(verzaubert prinz frosch erstgeboren thronfolger is var nonvar)
Please enter Break-command: <-4>      [Letzte 4 Schritte anzeigen]
| FAIL (erstgeboren philip)
| REDO (prinz philip)
| EXIT (prinz charles)
| CALL (erstgeboren charles)
More information desired? Enter +p, -p or number of steps: <-P>
|  CALL (frosch _x-2)               [Ganze Seite zurueckblaettern]
|  EXIT (frosch fridolin)
|  CALL (verzaubert fridolin)
```

```
|   FAIL (verzaubert fridolin)
|   REDO (frosch fridolin)
|   EXIT (frosch kermit)
|   CALL (verzaubert kermit)
|   EXIT (verzaubert kermit)
|  EXIT (prinz kermit)
|  CALL (erstgeboren kermit)
|  FAIL (erstgeboren kermit)
|  REDO (prinz kermit)
|   REDO (verzaubert kermit)
|   FAIL (verzaubert kermit)
|   REDO (frosch kermit)
|   FAIL (frosch _x-2)
|  EXIT (prinz andrew)
|  CALL (erstgeboren andrew)
|  FAIL (erstgeboren andrew)
|  REDO (prinz andrew)
|  EXIT (prinz philip)
|  CALL (erstgeboren philip)
More information desired? Enter +p, -p or number of steps: <10>
|  FAIL (erstgeboren philip)
|  REDO (prinz philip)
|  EXIT (prinz charles)
|  CALL (erstgeboren charles)       [Wieder im "alten" Break-Level]
Please enter Break-command: <GOAL> [Anzeigen des aktuellen Ziels]
The actual goal is: (erstgeboren charles)
Please enter Break-command: <1>  [Einen Schritt weiter = Stepper]
LISPLOG continues...
|  EXIT (erstgeboren charles)
More information desired? Enter +p, -p or number of steps: <1>
|EXIT (thronfolger charles)
success!
(_gesuchte-person = charles)
More? (y or p) <y>                 [Weitere "Loesungen" gewuenscht]
|REDO (thronfolger charles)
More information desired? Enter +p, -p or number of steps: <1>
|  REDO (erstgeboren charles)
More information desired? Enter +p, -p or number of steps: <-4>
|  EXIT (erstgeboren charles)
|EXIT (thronfolger charles)
|REDO (thronfolger charles)
|  REDO (erstgeboren charles)
More information desired? Enter +p, -p or number of steps: <P>
LISPLOG continues...
|  FAIL (erstgeboren charles)
|  REDO (prinz charles)
|  FAIL (prinz _x-1)
|FAIL (thronfolger _gesuchte-person)
nil                          [Es gibt keine weiteren "Loesungen"]
*<lisp>                         [Verlassen des LISPLOG-Toplevel]
2.                              [Zurueck im FRANZ-LISP-Toplevel]
```

7.8. Cut-Anzeiger und Hand-Schneider

Bei der Einfuehrung des Box-Modells hatten wir in Abschnitt 5.1. bereits festgestellt, dass PROLOG als Programmiersprache (fast) ohne explizite Kontrollstrukturen auskommt. Gerade deswegen ist es ja auch so schwierig, aus einem PROLOG-Programm direkt den Kontrollfluss abzulesen bzw. ihn beim Debugging angemessen darzustellen. Hierfuer haben wir mit dem Box-Modell einen Ansatz uebernommen, der den Kontrollfluss bei der Abarbeitung eines PROLOG (LISPLOG)-Programms fuer eine gegebene Anfrage auf der Ebene der einzelnen zu beweisenden Teilziele (Literale) darstellt. Die Ports des Box-Modells haben wir mit dem Start (CALL, REDO) bzw. dem erfolgreichen (EXIT) oder erfolglosen (FAIL) Abschluss des Beweises fuer ein Teilziel identifiziert.

Solange wir bei unseren Betrachtungen von PROLOG-Programmen ohne Verwendung expliziter Kontrollmechanismen ("pure-PROLOG"-Programmen) ausgehen, reicht dieses Modell zur angemessenen Darstellung des Kontrollflusses aus.

Wie verhaelt es sich aber, wenn wir in PROLOG-Programmen eben auch von expliziten Kontrollmechanismen wie dem Cut-Operator Gebrauch machen?

Der Gebrauch solcher Mechanismen zur Steuerung des Kontrollflusses wirkt sich natuerlich auch auf die Beschreibungsebene des Box-Modells aus:

Wenn durch den Cut-Operator ein Teil des Suchraums abgeschnitten wird, dann wird fuer die darin befindlichen Teilziele ja gar kein Beweis versucht; diese Teilziele treten also auf der Ebene des Box-Modells gar nicht mehr auf. Das ist zunaechst auch voellig korrekt und unproblematisch. Das Problem besteht eben nur darin, dass die Bedingungen fuer das Wirksamwerden des Cut-Operators nicht logisch aus den vorangegangenen Beweisen von Teilzielen hergeleitet werden koennen.
Insofern ist also das Box-Modell fuer derartige PROLOG-Programme nicht ausreichend, weil es keine Informationen ueber die Wirkung des Cut-Opeartors enthaelt. Fuer den Benutzer einer nur auf diesem Box-Modell basierenden Interaktionsumgebung ist es dann nicht ohne Probleme moeglich nachzuvollziehen, warum eine eigentlich anwendbare Klausel [42] einmal angewendet und bei einem anderen Aufruf ploetzlich nicht mehr benutzt wird. Aus diesem Grund erscheint es durchaus sinnvoll und notwendig, dem Benutzer neben den eigentlichen Box-Modell-Informationen auch Informationen ueber die Wirkung des Cut-Operators zur Verfuegung zu stellen.

---

[42] Eine Klausel, deren Klauselkopf mit dem aktuell zu beweisenden Teilziel unifizierbar ist.

Die Tatsache, dass der Cut-Operator in LISPLOG auf sein initiales Auftreten (direkt nach dem Klauselkopf; vor den evtl. vorhandenen Praemissen) beschraenkt ist, koennen wir dabei natuerlich auch ausnutzen.

Waehrend vom allgemeinen Cut-Operator eine zweifache Wirkung auf die den Cut enthaltende Klausel - alle Literale zwischen Klauselkopf und Cut werden beim Backtracking uebergangen - sowie die nachfolgenden Klauseln der den Cut enthaltenden Prozedur - diese werden ebenfalls beim Backtracking uebergangen - ausgeht, ist die Semantik des initialen Cut-Operators wesentlich klarer: Durch das initiale Auftreten erstreckt sich seine Wirkung nur auf die nachfolgenden Klauseln. Die Bedeutung eines initialen Cuts in einer Klausel K einer Prozedur P laesst sich also kompakt wie folgt beschreiben: "Wenn die Klausel K anwendbar, ihr Kopf also mit dem aktuellen Teilziel unifizierbar ist, dann verwende K als ausschliessliche Definition des Praedikats P, d.h. vernachlaessige alle alternativen Definitionen (nachfolgende Klauseln)."
In diesem Sinne wird eine mit einem initialen Cut versehene Klausel also als "sichere" Klausel fuer all die Faelle ausgezeichnet, fuer die sie anwendbar, ihr Kopf also unifizierbar ist.

Die Tatsache, dass eine Cut-Klausel angewendet wird, ist an sich noch kein Grund fuer eine besondere Information an den Benutzer. Die Klausel waere ja auch ohne den Cut angewandt worden. Erst wenn eine Cut-Klausel fehlschlaegt und aus diesem Grunde weitere ebenfalls anwendbare Klauseln uebersprungen werden, tritt die Wirkung des Cut-Operators ein. Dieser Umstand sollte dann dem Benutzer in angemessener Weise, etwa durch eine Meldung der Form

"Cutting <n> [<m>] clauses after clause <c>"

mitgeteilt werden. Dabei sollen <c> die den Cut enthaltende Klausel, <n> die Anzahl der durch den Cut uebergangenen Klauseln und <m> die Anzahl der tatsaechlich abgeschnittenen Klauseln bezeichnen. Im Allgemeinen sind ja nicht alle nachfolgenden Klauseln auch anwendbar, so dass auch ohne den Cut-Operator nur ein Teil der <n> nachfolgenden Klauseln als Alternativen fuer die gescheiterte Klausel <c> in Betracht kaemen. Im Falle <m> = 0 kann dann auch auf eine Information des Benutzers verzichtet werden, da der Cut-Operator mangels alternativer anwendbarer Klauseln keine Auswirkung auf den Programmablauf hatte. Im Falle <m> > 0 ist der Benutzer durch Ausgabe dieser Informationen hinreichend ueber die Tatsache, den Ort (Klausel <c>) und den Umfang (<m> anwendbare Klauseln) der Beschneidung des Suchraums informiert.

Auch implementationstechnisch ist diese Erweiterung sehr einfach vorzunehmen:

Zunaechst muessen die Informationen ueber das Wirksamwerden des Cut-Operators von den Kernfunktionen des LISPLOG-Interpreters bereitgestellt werden. Den geeigneten Ansatzpunkt dazu bietet die Funktion or-process (vgl. Seite 22):

Wenn hier nach einem Unifikationserfolg der Beweis der um die Praemissen der angewandten Klausel erweiterten list-of-goals (Aufruf von and-process) fehlschlaegt und Backtracking notwendig wird, dann wird der entsprechende rekursive or-process-Aufruf fuer die noch zur Verfuegung stehenden Klauseln nur dann durchgefuehrt, falls die fehlgeschlagene Klausel nicht mit einem Cut versehen war, wie es in der Bedingung

```
(not (cut-p assertion))
```

zum Ausdruck kommt. Hier kann nun die Cond-Klausel einfach um den komplementaeren Fall, dass durch den Cut-Operator alternative Klauseln abgeschnitten wurden, erweitert [43] werden, in dem dann die Interaktionsumgebung ueber eine neue Schnittstellenfunktion box-cut mit den notwendigen Informationen versorgt wird:

```
(def or-process
  (lambda (database-left goals-left goal environment level)
    (cond
     ((null database-left) nil)
     (t (let* ((assertion (rename-variables (first database-left)
                                            (list level)))
               (new-goals (s-premises assertion))
               (head (s-conclusion assertion))
               (new-environment (unify goal head environment)))
              (cond
               ((null new-environment)
                (or-process (rest database-left)
                            goals-left
                            goal
                            environment
                            level))
               ((box-skip? goal new-goals))
               ((and-process (append new-goals goals-left)
                             new-environment
                             (add1 level)))
;; Cut-Anzeiger:
;; -------------
               ((cut-p assertion)
                (box-cut goal assertion (rest database-left)))
               (t (or-process (rest database-left)
                              goals-left
                              goal
                              environment
                              level)))))))
```

Die neue Schnittstellenfunktion box-cut verarbeitet diese neue

---

[43] Dieser Fall war bisher implizit durch das Fehlen der entsprechenden Cond-Klausel so realisiert, dass vom or-process direkt der Wert nil (= Fehlschlag) zurueckgeliefert wird.

Art von Ablaufinformation ganz analog zu den bekannten Funktionen box-call, box-exit, box-fail, box-redo, box-eval oder box-value, allerdings mit dem Unterschied, dass eine Information nur in dem Falle ausgegeben und im backtrace-stack abgelegt wird, wenn durch den Cut-Operator auch tatsaechlich mindestens eine anwendbare Klausel abgeschnitten wurde: [44]

```
(def box-cut
  (lambda (goal clause clause-list)
    (if tracemode
        (let ((clauses-cut (trace-unify goal clause-list)))
             (if (greaterp clauses-cut 0)
                 (push-backtrace-stack printlevel
                                       (list 'cut
                                             (length clause-list)
                                             clauses-cut
                                             clause)
                                       goal))
             nil))))
```

Auch hierbei ist wieder wichtig, dass der Wert von box-cut auf jeden Fall immer nil ist, da dieser vom aufrufenden or-process ja direkt als Misserfolgsmeldung zurueckgeliefert wird. Auf diese Weise ist der Cut-Anzeiger als ergaenzendes Werkzeug zur Unterstuetzung des Benutzers beim Gebrauch des initialen Cut-Operators vollstaendig in die bestehende Interaktionsumgebung integriert.

Neben diesem Instrument zur reinen Darstellung der Wirkung des initialen Cut-Operators bietet sich aber ausserdem die Bereitstellung eines weiteren Werkzeugs zur interaktiven Steuerung des Kontrollflusses an, um so den Benutzer noch besser bei der Entwicklung von LISPLOG-Programmen unter Einsatz des initialen Cut-Operators zu unterstuetzen. Die zugrundeliegende Idee ist dabei, den Benutzer nicht nur ueber das Abschneiden von Klauseln durch den initialen Cut-Operator zu informieren (Cut-Anzeiger), sondern beim Fehlschlagen einer Klausel vor der Anwendung der alternativen Klausel ("tiefes" Backtracking) den Benutzer zuvor zu fragen, ob diese alternative Klausel tatsaechlich angewendet werden soll. Falls nicht, so kann der Benutzer hier interaktiv den gleichen Effekt erzielen wie im Fall, dass die zuvor fehlgeschlagene Klausel mit einem initialen Cut-Operator versehen gewesen waere: es wird keine alternative Klausel angewandt, Backtracking also verhindert und so der Suchraum wie beim initialen Cut-Operator entsprechend eingeschraenkt bzw. beschnitten (Hand-Schneider).

---

[44] Die Anzahl aller mit einem Teilziel (goal) unifizierbaren Klauselkoepfe aus einer Liste von Klauseln (clause-list) wird durch die Funktion trace-unify berechnet.

Die Einbindung dieses Hand-Schneiders in den bestehenden LISPLOG-Interpreter setzt nun ebenso wie auch der schon beschriebene Cut-Anzeiger an der inneren Cond-Klausel in der Funktion or-process (vgl. Seite 68) an. Die damit naheliegende Verbindung der beiden neuen Werkzeuge Hand-Schneider und Cut-Anzeiger ist auch durchaus sehr sinnvoll: Wenn durch den Hand-Schneider eine Wirkung auf den Kontrollfluss erzielt werden kann, die einem voruebergehend im LISPLOG-Programm eingefuegten Cut-Operator entspricht, so soll diese Beschneidung des Suchraums natuerlich auch durch den Cut-Anzeiger entsprechend dargestellt werden.

Damit ergibt sich auch eine sehr einfache aber elegante Implementationsmoeglichkeit: Als Bedingung fuer den Aufruf des Cut-Anzeigers (box-cut) wird neben dem Test auf Vorhandensein eines Cut-Operators in der entsprechenden Klausel (cut-p) alternativ ein Aufruf einer neuen Schnittstellenfunktion box-cut-p eingefuehrt, die genau dann nil liefert, wenn Backtracking erwuenscht ist, der Benutzer also nicht in den Kontrollfluss eingreifen moechte:

```
(def or-process
  (lambda (database-left goals-left goal environment level)
    (cond
     ((null database-left) nil)
     (t (let* ((assertion (rename-variables (first database-left)
                                            (list level)))
               (new-goals (s-premises assertion))
               (head (s-conclusion assertion))
               (new-environment (unify goal head environment)))
              (cond
               ((null new-environment)
                (or-process (rest database-left)
                            goals-left
                            goal
                            environment
                            level))
               ((box-skip? goal new-goals))
               ((and-process (append new-goals goals-left)
                             new-environment
                             (add1 level)))
               ((or (cut-p assertion)
;;
;; Hand-Schneider:
;; ---------------
                    (box-cut-p goal database-left))
                (box-cut goal assertion (rest database-left)))
               (t (or-process (rest database-left)
                              goals-left
                              goal
                              environment
                              level)))))))))
```

Die Funktion box-cut-p wird also nur dann ueberhaupt aufgerufen, wenn die Anwendung einer Klausel fehlschlaegt und diese Klausel

nicht bereits mit einem Cut-Operator versehen war. In diesem Fall wird Backtracking ja bereits durch den Cut-Operator unterbunden.

Andernfalls muss innerhalb der neuen Funktion box-cut-p zunaechst geprueft werden, ob ein Backtracking-Versuch ueberhaupt sinnvoll ist, d.h. ob es ueberhaupt eine anwendbare alternative Klausel zu der fehlgeschlagenen Klausel gibt. Dieser Test erfolgt aehnlich wie beim Cut-Anzeiger durch Aufruf einer Funktion trace-unify-p, die im Unterschied zu trace-unify nicht die Anzahl der alternativen Klauseln liefert, sondern nur die Feststellung, ob es ueberhaupt alternative anwendbare Klauseln gibt. Falls es naemlich gar keine anwendbaren alternativen Klauseln gibt, dann bleibt auch ein voruebergehend in die fehlgeschlagene Klausel eingesetzter initialer Cut-Operator auf jeden Fall wirkungslos, und somit ist auch eine entsprechende Frage an den Benutzer irrelevant und deshalb zu unterlassen.

Natuerlich soll der Benutzer auch die Moeglichkeit haben, den Hand-Schneider gezielt nur fuer bestimmte Praedikate einzusetzen. Deshalb fuehren wir fuer die Steuerung des Hand-Schneiders noch zwei neue Kommandos "cut" und "nocut" ein, mit denen die Liste der Praedikatnamen verwaltet wird, fuer die der Hand-Schneider aktiviert ist. Diese Liste wird analog zu den anderen Kommandos der Interaktionsumgebung an eine neue globale Variable cutmode gebunden, die beim Start des LISPLOG-Systems sinnvollerweise mit dem Wert nil initialisiert wird.

Somit kann nun der Hand-Schneider durch das Kommando

```
cut pred-name-1 pred-name-2 ...
```

fuer die angegebenen Praedikate bzw. bei fehlenden Argumenten fuer alle benutzerdefinierten Praedikate aktiviert werden. Umgekehrt wird der Hand-Schneider durch das Kommando

```
nocut pred-name-1 pred-name-2
```

fuer die angegebenen Praedikate bzw. fuer alle Praedikate wieder ausgeschaltet.

Anhand der neu eingefuehrten Variablen cutmode muss nun in der Funktion box-cut-p

```
(def box-cut-p
  (lambda (goal database-left)
    (let* ((clause (first database-left))
           (pred (get-pred-of-goal (s-conclusion clause))))
          (cond ((and (memq pred-name cutmode)
                      (trace-unify-p goal (rest database-left)))
                 (manual-cutter clause))))))
```

zusaetzlich noch geprueft werden, ob der Hand-Schneider fuer das entsprechende Praedikat ueberhaupt aktiviert war, bevor schliess-

lich ueber die Funktion manual-cutter dem Benutzer die Moeglichkeit gegeben wird, interaktiv in den Kontrollfluss einzugreifen:

```
(def manual-cutter
  (lambda (clause)
    (princ "Do you want to cut the clause ")
    (print-external (cons 'ass clause))
    (princ "? ")
    (let ((answer (first (lineread t))))
         (if (memq answer '(b B break Break BREAK))
             (cond (tracemode (setq answer (break-loop t)))
                   (t (princ "No Box-Model-Tracer activated...")
                      (terpr))))
         (setq linecount 22)
         (cond ((memq answer '(y Y j J yes Yes YES ja Ja JA))
                (make-cut-permanent? clause)
                t)
               ((memq answer '(n N no No NO nein Nein NEIN)) nil)
               (t (manual-cutter clause))))))
```

Auf die Frage "Do you want to cut..." ist neben der direkten Antwort "Ja" oder "Nein" auch die Eingabe von "Break" bzw. der Abkuerzung "b" moeglich, so dass der Benutzer ueber den LISPLOG-Break-Level die Moeglichkeit hat, zunaechst den aktuellen Stand der Bearbeitung seines LISPLOG-Programms naeher zu untersuchen, bevor er dann durch Eingabe einer entsprechenden positiven oder negativen Antwort (vgl. Tabelle auf Seite 54) den Break-Level verlaesst und bei positiver Antwort die gewuenschte Aktion vom Hand-Schneider ausgefuehrt wird.

Wurde nun die Frage des Hand-Schneiders positiv beantwortet, so verhaelt sich das System genauso, als wenn voruebergehend und nur fuer diesen einen Beweisschritt die entsprechende Klausel mit einem initialen Cut-Operator versehen gewesen waere.
Bei rekursiv formulierten Praedikaten kann sich an dieser Stelle also anschliessend gleich noch mehrmals die gleiche Frage fuer die gleiche Klausel anschliessen, da es sich dabei zwar um die gleiche Klausel, aber um verschiedene Beweisschritte handelt, so dass die urspruengliche Frage des Hand-Schneiders jedes Mal von Neuem entschieden werden kann.

Deshalb erscheint es sinnvoll, ueber den bisher beschriebenen Funktionsumfang hinaus dem Benutzer an dieser Stelle ausserdem die Moeglichkeit zu geben, den vom Hand-Schneider nur voruebergehend bewirkten Cut-Operator bei Bedarf auch permanent in der Datenbasis, also im entsprechenden LISPLOG-Programm einzutragen.

Zu diesem Zweck wird bei positiver Antwort des Benutzers auf die Frage "Do you want to cut the clause..." nicht nur der Wert t von der Funktion manual-cutter und somit auch von box-cut-p zurueckgeliefert, sondern zuvor ueber einen Aufruf einer neuen Funktion make-cut-permanent? dem Benutzer die Moeglichkeit gegeben, die betreffende Klausel permanent mit einem initialen Cut-Operator zu

versehen. Auch hier hat der Benutzer wieder die Moeglichkeit, auf die Frage "Do you want to make the Cut permanent?" direkt oder indirekt (ueber den LISPLOG-Break-Level) zu antworten. Bei einer positiven Antwort wird dann schliesslich die betreffende Klausel direkt durch eine destruktive Manipulation mittels rplaca durch Auswertung des Ausdrucks

```
(rplaca clause (cons 'cut (s-conclusion clause)))
```

mit einem initialen Cut-Operator versehen.

Mit dieser Erweiterung zum interaktiven Einfuegen eines initialen Cut-Operators in das aktuelle LISPLOG-Programm eignen sich die beiden speziell auf den fuer LISPLOG charakteristischen initialen Cut-Operator abgestimmten Werkzeuge Cut-Anzeiger und Hand-Schneider nicht nur zur Fehlererkennung, sondern auch zur interaktiven Fehlerkorrektur. Wenn auch diese Werkzeuge eine interaktive Fehlerkorrektur nur fuer eine eingeschraenkte Klasse von Programmierfehlern (unbeabsichtigtes Backtracking wegen fehlendem Cut-Operator) ermoeglichen, so ist der praktische Nutzen dieser - wie wir gesehen haben - recht einfach zu implementierenden Werkzeuge doch recht hoch einzuschaetzen, da diese Klasse von Programmierfehlern in echten LISPLOG-Programmen nach reinen Syntaxfehlern mit am haeufigsten auftritt.

Dementsprechend wollen wir diesen Abschnitt nun auch mit einem Dialogbeispiel beenden, dem ein LISPLOG-Programm mit einem Fehler zugrundeliegt, der gerade in diese Klasse von Programmierfehlern faellt: Das folgende LISPLOG-Programm (6) definiert ein Praedikat "even-occur" ueber einer Zahl _n und einer Zahlenliste _l, das genau dann erfuellt sein soll, wenn die Zahl _n geradzahlig-oft (also gar nicht, zweimal, viermal usw.) in der Liste _l vorkommt:

```
(6)   (ass (even-occur _n _l) (even-occur-1 _n _l 0))
      (ass (even-occur-1 ID nil 0))
      (ass (even-occur-1 _n (_n . _t) 0) (even-occur-1 _n _t 1))
      (ass (even-occur-1 _n (_n . _t) 1) (even-occur-1 _n _t 0))
      (ass (even-occur-1 _n (_h . _t) _c) (even-occur-1 _n _t _c))
```

Wenn wir dieses LISPLOG-Programm laden und dann die Anfrage

```
(even-occur 2 (2 2 3 2 1 2 2))
```

stellen, so erwarten wir natuerlich, dass der Beweis dieses Ziels fehlschlaegt. Ein Programmierfehler der eben angesprochenen Art bewirkt aber, dass das System die falsche Antwort, naemlich eine erfolgreiche Beweisfuehrung, meldet.

Dem Anwender, der dieses LISPLOG-Programm erstellt hat und nun austestet, ist beim Niederschreiben dieser obigen Klauseln klar, dass jede Klausel, wenn sie angewendet werden kann, auch eine ausschliessliche und korrekte Definition der even-occur-Relation repraesentiert, ein spaeterer Fehlschlag der Praemissen also das

Nichtvorliegen der even-occur-Relation beschreibt. Ein Backtrakking zu einer der nachfolgenden alternativen Klauseln (sog. "tiefes" Backtracking) sollte also eigentlich nicht stattfinden. Und gerade hier liegt auch die Ursache fuer das unerwartete Verhalten des Systems: Waehrend bei einem voll instanziierten Teilziel die ersten drei Klauseln der Prozedur "even-occur-1" sich gegenseitig ausschliessen, dient die letzte Klausel eigentlich nur als sog. "Catch-All-Klausel", die fuer (fast) alle Faelle anwendbar ist und intuitiv nur fuer den Fall niedergeschrieben wurde, wenn gar keine der voranstehenden Klauseln angewendet werden konnte.

Mit dem Einsatz des Hand-Schneiders kann man nun das unerwartete Backtracking leicht erkennen und darueberhinaus den Fehler im LISPLOG-Programm (zwei fehlende Cut-Operatoren) sogar interaktiv noch waehrend des Testlaufs beheben:

```
1.<lisplog>                          [Aufruf von LISPLOG aus FRANZ LISP]
*<consult even-occur.db>                [Laden des Beispiel-Programms]
[load even-occur.db]
(even-occur.db)
*<listing>                                  [Anzeigen aller Klauseln]
(ass (even-occur _n _l) (even-occur-1 _n _l 0))
(ass (even-occur-1 ID nil 0))
(ass (even-occur-1 _n (_n . _t) 0) (even-occur-1 _n _t 1))
(ass (even-occur-1 _n (_n . _t) 1) (even-occur-1 _n _t 0))
(ass (even-occur-1 _n (_h . _t) _c) (even-occur-1 _n _t _c))
*<(even-occur 2 (2 2 3 2 1 2 2))>  [Anfrage muesste fehlschlagen]
success                                   [... liefert aber Erfolg!]
More? (y or n) <n>
t
*<spy>                       [Tracer einschalten, um Fehler zu suchen]
(even-occur even-occur-1 is not var nonvar)
*<cut>                        [Zusaetzlich Hand-Schneider aktivieren!]
(even-occur even-occur-1)
*<(even-occur 2 (2 2 3 2 1 2 2))>   [Nochmal die gleiche Anfrage]
|CALL (even-occur 2 (2 2 3 2 1 2 2))
| CALL (even-occur-1 2 (2 2 3 2 1 2 2) 0)
|  CALL (even-occur-1 2 (2 3 2 1 2 2) 1)
|   CALL (even-occur-1 2 (3 2 1 2 2) 0)
|    CALL (even-occur-1 2 (2 1 2 2) 0)
|     CALL (even-occur-1 2 (1 2 2) 1)
|      CALL (even-occur-1 2 (2 2) 1)
|       CALL (even-occur-1 2 (2) 0)
|        CALL (even-occur-1 2 nil 1)
|        FAIL (even-occur-1 2 nil 1)
|       FAIL (even-occur-1 2 (2) 0)
Do you want to cut the clause
(ass (even-occur-1 _n (_n . _t) 0) (even-occur-1 _n _t 1))? <b>
***** LISPLOG - Break - Level *****
Please enter Break-command: <GOAL>
The actual goal is: (even-occur-1 2 (2) 0)
Please enter Break-command: <LISTING even-occur-1>
(ass (even-occur-1 ID nil 0))
```

```
(ass (even-occur-1 _n (_n . _t) 0) (even-occur-1 _n _t 1))
(ass (even-occur-1 _n (_n . _t) 1) (even-occur-1 _n _t 0))
(ass (even-occur-1 _n (_h . _t) _c) (even-occur-1 _n _t _c))
Please enter Break-command: <Yes>   [Antwort fuer Hand-Schneider]
Do you want to make the cut permanent? <n>        [Erstmal nicht!]
|       Cutting 2 [1] clauses after clause
|       (ass (even-occur-1 _n (_n . _t) 0)
|            (even-occur-1 _n _t 1))
|      FAIL (even-occur-1 2 (2 2) 1)
Do you want to cut the clause
(ass (even-occur-1 _n (_n . _t) 1) (even-occur-1 _n _t 0))? <y>
Do you want to make the cut permanent? <y>
Cut is made permanent!
|      Cutting 1 [1] clauses after clause
|      (ass (even-occur-1 _n (_n . _t) 1) (even-occur-1 _n _t 0))
|     FAIL (even-occur-1 2 (1 2 2) 1)
|    FAIL (even-occur-1 2 (2 1 2 2) 0)
Do you want to cut the clause
(ass (even-occur-1 _n (_n . _t) 0) (even-occur-1 _n _t 1))? <y>
Do you want to make the cut permanent? <y>
Cut is made permanent!
|    Cutting 2 [1] clauses after clause
|    (ass (even-occur-1 _n (_n . _t) 0) (even-occur-1 _n _t 1))
|   FAIL (even-occur-1 2 (3 2 1 2 2) 0)
|  FAIL (even-occur-1 2 (2 3 2 1 2 2) 1)
|  Cutting 1 [1] clauses after clause
|  (ass (even-occur-1 _n (_n . _t) 1) (even-occur-1 _n _t 0))
| FAIL (even-occur-1 2 (2 2 3 2 1 2 2) 0)
| Cutting 2 [1] clauses after clause
| (ass (even-occur-1 _n (_n . _t) 0) (even-occur-1 _n _t 1))
|FAIL (even-occur 2 (2 2 3 2 1 2 2))
nil               [Anfrage nicht beweisbar, Fehler also behoben!]
*<listing>                 [Beachte: Datenbasis wurde geaendert!]
(ass (even-occur _n _l) (even-occur-1 _n _l 0))
(ass (even-occur-1 ID nil 0))
(ass !(even-occur-1 _n (_n . _t) 0) (even-occur-1 _n _t 1))
(ass !(even-occur-1 _n (_n . _t) 1) (even-occur-1 _n _t 0))
(ass (even-occur-1 _n (_h . _t) _c) (even-occur-1 _n _t _c))
*<nospy>                                     [Tracer ausschalten]
nil
*<(even-occur 2 (2 2 3 2 1 2 2))>  [Nochmal die gleiche Anfrage]
nil       [...korrektes Ergebnis und kein "tiefes" Backtracking!]
*<nocut>                             [Hand-Schneider ausschalten]
nil
*<(even-occur 5 (5 4 2 5 3 5 3 5 5 6 5))> [Noch ein anderer Test]
success                          [...auch mit korrektem Ergebnis!]
More? (y or n) <n>
t
*<lisp>                           [Verlassen des LISPLOG-Toplevel]
2.                                [Zurueck im FRANZ-LISP-Toplevel]
```

8. Erweiterung des Box-Modells fuer Funktionen

Alle in den vorstehenden Kapiteln dargestellten Konzepte fuer eine LISPLOG-Interaktionsumgebung unterstuetzen bislang nur die relationale Komponente von LISPLOG.
Bereits in Kapitel 4 haben wir aber festgestellt, dass fuer eine Interaktionsumgebung fuer LISPLOG ein Beschreibungsmodell erforderlich ist, das sowohl den relationalen - in Klauselform notierten Teil eines LISPLOG-Programms - als auch den funktionalen - in Form von LISP-Funktionen codierten - Teil angemessen darstellt.

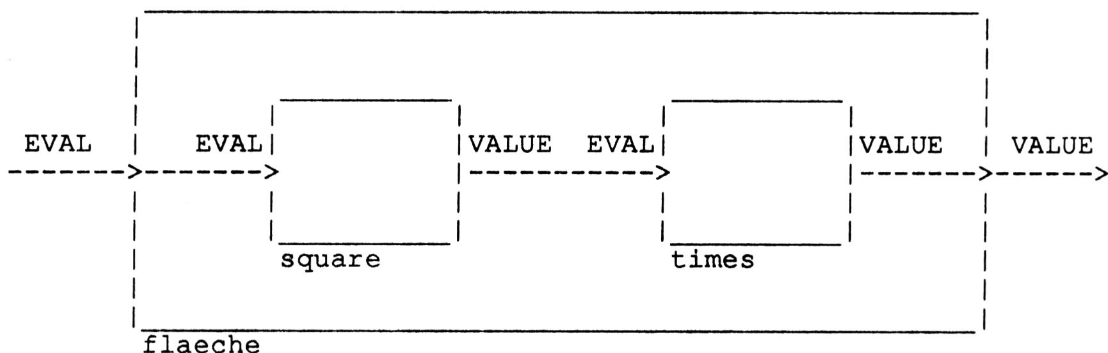
Deshalb werden wir in diesem Kapitel eine Erweiterung des Box-Modells fuer Funktionen vorstellen und anschliessend die Integration in die bisher vorgestellte Interaktionsumgebung beschreiben.

8.1. Ein Box-Modell fuer Funktionen

LISP-Funktionen koennen aehnlich wie PROLOG-Praedikate durch Procedure-Boxes repraesentiert werden, die dann aber aufgrund der Tatsache, dass Funktionen stets deterministisch sind, nur einen Eingang (EVAL) und einen Ausgang (VALUE) haben.
Auf diese Weise koennen auch geschachtelte Funktionen durch solche Boxes dargestellt werden. Fuer einen Aufruf der folgenden LISP-Funktion

```
(defun flaeche (radius)
  (times 3.1415926 (square r)))
```

erhalten wir somit die Box-Modell-Repraesentation



bzw. die folgende Trace-Darstellung:

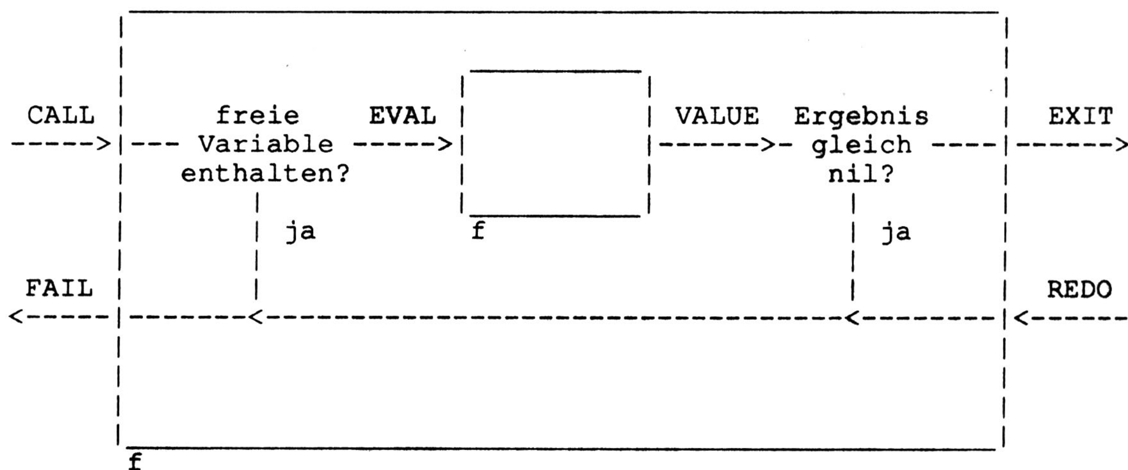
```
|EVAL (flaeche 5)
| EVAL (square 5)
| VALU 25
| EVAL (times 3.1415926 25)
| VALU 78.539815
|VALU 78.539815
```

8.2. LISP-Durchgriffe als PROLOG-Goals

In LISPLOG koennen als Praemissen von PROLOG-Klauseln auch Aufrufe von LISP-Funktionen auftreten, wobei bei dieser Art des LISP-Durchgriffs nur der Wahrheitswert der ausgewerteten LISP-Funktion (nil oder non-nil) in PROLOG verwendet wird (praedikativer LISP-Durchgriff).

Der LISP-Aufruf wird als PROLOG-Goal so interpretiert, dass das PROLOG-Goal genau dann erfolgreich ist, wenn zum Aufrufzeitpunkt alle vorkommenden PROLOG-Variablen gebunden sind [45] und der Wert der aufgerufenen LISP-Funktion ungleich nil ist.

Im Box-Modell laesst sich ein solcher praedikativer LISP-Durchgriff (f x1 x2 ... xn) sehr einfach darstellen, indem in die das entsprechende PROLOG-Goal repraesentierende Procedure-Box mit vier Ports eine funktionale Box eingefuegt wird:



Sind im LISP-Aufruf noch freie Variablen enthalten, so schlaegt

---

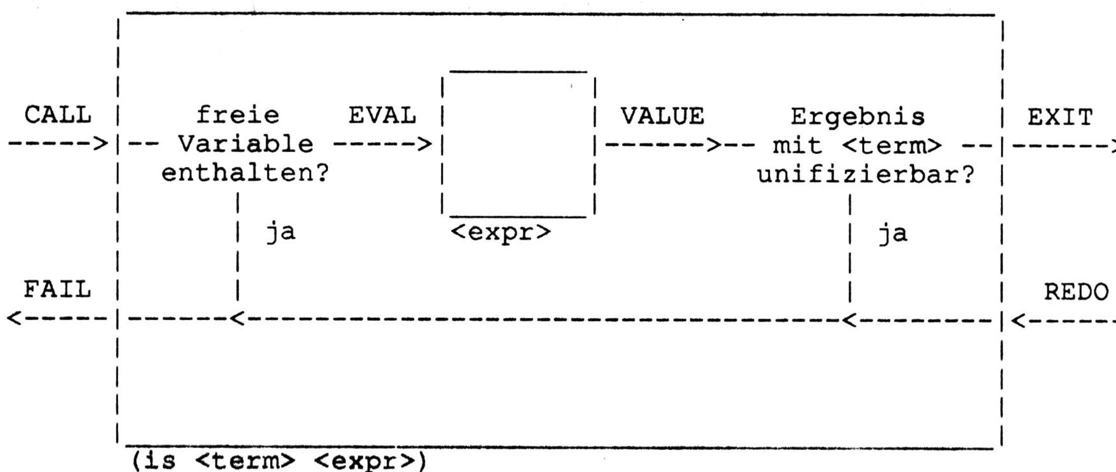
[45] Dieser Test auf freie Variablen (vgl. Funktion <u>containsfreevars</u> in Anhang B) beruecksichtigt auch die Definitionsart (lambda, nlambda, lexpr) der aufgerufenen LISP-Funktion und weiterer evtl. in den Argumentstellen eingeschachtelter LISP-Aufrufe: Aufrufe von nlambda-Funktionen koennen dabei sehr wohl auch freie Variablen enthalten, da die Argumente in der Regel nicht oder nur durch einen expliziten eval-Aufruf ausgewertet werden. Dadurch ist es z.B. moeglich, mit LISP-Funktionen auch PROLOG-Klauseln zu bearbeiten. Auch wenn es bei diesem Vorgehen durchaus noch vorkommen kann, dass bei einem LISP-Durchgriff auf ungebundene PROLOG-Variablen zugegriffen wird und dies natuerlich zu einem Laufzeitfehler fuehrt, erscheint diese Loesung doch besser, als grundsaetzlich freie Variablen im Aufruf zu verbieten.

der LISP-Durchgriff fehl (FAIL). Ansonsten wird abhaengig vom Ergebnis der LISP-Funktion die umgebende Procedure-Box entweder durch den EXIT- oder den FAIL-Ausgang verlassen. Fuer den Fall, dass dieser LISP-Durchgriff ueber den EXIT-Ausgang erfolgreich verlassen wird, das nachfolgende Ziel aber fehlschlaegt, so muss ein Backtracking-Versuch fuer den LISP-Durchgriff (Wiederbetreten durch den REDO-Eingang) direkt zum Scheitern (FAIL) gebracht werden. Funktionen sind deterministisch, ein Backtracking, d.h. eine erneute Auswertung der Funktion, ist also sinnlos.

Ein Beispiel fuer einen praedikativen LISP-Durchgriff haben wir bereits in Abschnitt 5.1. mit dem Praedikat "delete" fuer Zahlenlisten kennengelernt, wenngleich dabei nicht auf benutzerdefinierte LISP-Funktionen, sondern auf die Systempraedikate "greaterp" und "lessp" durchgegriffen wird. In Abschnitt 8.4. werden wir fuer eine lediglich in diesem Punkt abgewandelte Datenbasis (4') auch einen Beispieldialog angeben.

8.3. LISP-Ausdruecke auf der rechten Seite vom is-Operator

Neben dem praedikativen LISP-Durchgriff gibt es in LISPLOG eine weitere Durchgriffsmoeglichkeit von PROLOG nach LISP, bei der der Wert der aufgerufenen LISP-Funktion auch in PROLOG weiterverwendet werden kann: den verallgemeinerten is-Operator. Im Gegensatz zu Standard-Prolog (vgl. [Clocksin & Mellish 1981/84]) koennen in LISPLOG auf der rechten Seite eines "is" nicht nur arithmetische Ausdruecke, sondern ein beliebiger LISP-Ausdruecke stehen. Dieser wird dann beim Aufruf evaluiert und das Ergebnis mit dem PROLOG-Term der linken Seite des is-Operators zu unifizieren versucht, so dass wir fuer einen Aufruf der Form "(is <term> <expr>)" die folgende Box-Modell-Repraesentation erhalten:



Natuerlich gilt auch hier fuer den LISP-Ausdruck die gleiche Einschraenkung bzgl. ungebundener PROLOG-Variablen, wie dies beim praedikativen LISP-Durchgriff (vgl. Abschnitt 8.2.) bereits ausgefuehrt wurde.

Auch fuer diese zweite Art des LISP-Durchgriffs ist kein Backtracking sinnvoll, da eine erneute Auswertung des LISP-Ausdrucks <expr> stets zum gleichen Resultat fuehrt. [46]

Ein Beispiel fuer eine Anwendung des is-Operators haben wir schon in Abschnitt 7.5. mit der (nur partiell korrekten) Fakultaets-Datenbasis

```
(4)   (ass (fak 0 1))
      (ass (fak _n _nfak) (is _n1 (sub1 _n))
                          (fak _n1 _n1fak)
                          (is _nfak (times _n1fak _n)))
```

kennengelernt. Der folgende Abschnitt 8.4. enthaelt zu der dann auch korrekten Formulierung (4') einen Beispieldialog, der die Wirkung des is-Operators und die erzeugten Ablaufinformationen veranschaulicht.

8.4. Integration in die Interaktionsumgebung

Die Integration des in den vorangegangenen Abschnitten entworfenen Box-Modells fuer den funktionalen Teil von LISPLOG in die zuvor bereits dargestellte Interaktionsumgebung gestaltet sich recht klar und einfach:

Da fuer die Erzeugung der Ablaufinformationen nicht der LISP-Interpreter entsprechend manipuliert werden soll, wie dies fuer den relationalen Teil am PROLOG-Interpreter angebracht war (vgl. die Abschnitte 6.3. und 6.5.), muss die Generierung der Ablaufinformationen von den im LISPLOG-Programm aufgerufenen benutzerdefinierten LISP-Funktionen selbst vorgenommen werden:

Dazu werden die Definitionen der benutzerdefinierten LISP-Funktionen selbst entsprechend manipuliert. Zuvor wird allerdings die urspruengliche Funktionsdefinition fuer die Kommandos "listing", "tell", "save" und ausserdem auch fuer das Kommando "nospy" unter

---

[46] Dabei gehen wir natuerlich davon aus, dass wir es nur mit seiteneffektfreien LISP-Funktionen zu tun haben, was insbesondere fuer Ein-/Ausgabefunktionen nicht zutrifft. Die Konsequenz daraus ist allerdings, dass - wie in anderen PROLOG-Systemen auch - fuer diese in LISPLOG durch LISP-Durchgriffe realisierten Primitive wie Ein-/Ausgabe kein Backtracking moeglich ist.

einem aus dem urspruenglichen Funktionsnamen (z.B. "foo") konstruierten Namen (hier: "foo-with-lisplog-trace") gesichert.

Eine Funktionsdefinition der Form

```
(defun foo (arg[1] ... arg[n])
  expr[1] ... expr[m]])
```

wird dann umgeformt zu: [47]

```
(defun foo (&rest args)
  (box-eval 'foo args)
  (let ((result (eval (cons 'foo-with-lisplog-trace
                           (mapcar 'quote-arg args)))))
       (box-value 'foo result)
       result))
```

Die Funktion quote-arg dient dabei zum Quotieren der Argumente:

```
(def quote-arg
  (lambda (x) (list 'quote x)))
```

Wird nun bei der Abarbeitung eines LISPLOG-Programms eine auf diese Weise manipulierte Funktion aufgerufen, so koennen mit den neuen Schnittstellenfunktionen box-eval und box-value ganz analog

---

[47] Allerdings ist diese Transformation beschraenkt auf Funktionen vom lambda-Typ. Das bedeutet, dass nlambda- oder lexpr-Funktionen (vgl. [Wilensky 1984]) nicht behandelt werden koennen und somit auch keine Ablaufinformationen liefern koennen.
Es gibt jedoch speziell fuer FRANZ-LISP eine sehr elegante Implementierungsmoeglichkeit unter Zuhilfenahme des FRANZ-LISP-Tracers (vgl. [Foderaro et al. 1983]). Dabei wird anstelle einer direkten Manipulation der entsprechenden Funktionsdefinitionen einfach der LISP-Tracer fuer diese Funktionen aktiviert, wobei als Protokollfunktionen (traceenter und traceexit) gerade die Schnittstellenfunktionen box-eval und box-value uebergeben werden. Der wesentliche Unterschied zur hier vorgestellten Vorgehensweise besteht eigentlich nur darin, dass die Aktionen die jetzt vom LISPLOG-System durchgefuehrt werden muessen (Sichern der alten Definition und Aenderung der Funktionsdefinition) im anderen Fall einfach vom LISP-Tracer uebernommen werden. Da diese sicherlich elegante Vorgehensweise allerdings FRANZ-LISP-spezifisch ist, bleiben wir hier bei der weniger eleganten und etwas aufwendigeren Vorgehensweise, die dafuer aber auch auf andere LISP-Dialekte uebertragbar ist. In der im Anhang B abgedruckten Implementation wird jedoch auch der kompakten Darstellung wegen auf die angedeutete elegantere Moeglichkeit zurueckgegriffen, wodurch der Leser leicht beide Realisierungsmoeglichkeiten gegenueberstellen kann.

zu den Funktionen box-call, box-exit, box-fail und box-redo diese Trace-Informationen verarbeitet und im backtrace-stack abgelegt werden:

```
(def box-eval
  (lambda (fct-name fct-args)
    (let ((goal (cons fct-name fct-args)))
         (cond (tracemode
                (push-backtrace-stack printlevel 'eval goal)
                (setq printlevel (add1 printlevel))
                (if (break-activated-p goal 'eval)
                    (setq breakflag t))
                (breakpoint)
                (skip? goal)))
         nil)))

(def box-value
  (lambda (fct-name fct-result)
    (cond (tracemode
           (check-skiplevel)
           (setq printlevel (sub1 printlevel))
           (push-backtrace-stack printlevel
                                 'value
                                 (list fct-name fct-result))
           (if (break-activated-p (list fct-name) 'value)
               (setq breakflag t))
           (breakpoint)
           nil))))
```

Bei der Abarbeitung eines LISPLOG-Programms werden die anfallenden Ablaufinformationen zu allen Praedikaten im backtrace-stack gesammelt, sobald die Interaktionsumgebung aktiv - d.h. tracemode ungleich nil - ist. Dabei spielt es keine Rolle, ob der zugehoerige Praedikatname auch in der an tracemode gebundenen Liste enthalten ist.

Um dies auch fuer alle benutzerdefinierten Funktionen zu realisieren, fuehren wir analog zur Variablen predicates eine neue globale Variable <u>functions</u> ein, die als Wert eine Liste der Namen aller benutzerdefinierten LISP-Funktionen erhaelt. Beim Einlesen einer Datei mit dem Kommando "consult" wird diese Liste dann entsprechend um die Namen der in dieser Datei definierten LISP-Funktionen erweitert.

Ist nun die Interaktionsumgebung ausgeschaltet (tracemode = nil), so werden weder vom LISPLOG-Interpreter (and-/or-process), noch von den benutzerdefinierten LISP-Funktionen Ablaufinformationen erzeugt, so dass eine moeglichst effiziente Abarbeitung des LISPLOG-Programms bei Verzicht auf eventuelle Debugging-Hilfen (Trace, Break-Paket, Backtrace) gewaehrleistet ist.

Wird dann durch ein beliebiges "spy"- oder "brk"-Kommando die Interaktionsumgebung eingeschaltet, so muessen jetzt auch alle

benutzerdefinierten LISP-Funktionen so behandelt werden, dass sie bei einem Aufruf entsprechende Ablaufinformationen ueber die Schnittstellen box-eval und box-value bereitstellen.

Dazu werden bei jedem "spy"-Kommando alle Funktionen, deren Name in der an functions gebundenen Liste enthalten ist, in der zu Beginn dieses Abschnitts beschriebenen Art und Weise manipuliert, falls sie nicht schon zuvor durch fruehere "spy"-Aufrufe entsprechend behandelt worden sind.

Wenn dann irgendwann die Interaktionsumgebung durch "nospy"- oder "nobrk"-Kommandos wieder ausgeschaltet wird, so werden dann auch diese Aenderungen der Funktionsdefinitionen wieder rueckgaengig gemacht, um den hierdurch bedingten erhoehten Laufzeit- und Speicherbedarf wieder zu reduzieren.

Natuerlich koennen die von den benutzerdefinierten LISP-Funktionen gelieferten Ablaufinformationen (EVAL und VALUE) auch analog zu den Informationen ueber die Abarbeitung von PROLOG-Praedikaten (CALL, EXIT, FAIL und REDO) fuer die Ausgabe auf dem Bildschirm mittels der Variable tracemode gefiltert werden. Dazu sind die Beschreibungen der Kommandos "spy" und "nospy" in Abschnitt 7.3. (Seite 44) entsprechend auf Funktionsnamen zu erweitern.

Ein Aufruf von "spy" ohne weitere Argumente bedeutet also, dass alle Namen von benutzerdefinierten Funktionen und Praedikaten sowie die vordefinierten Praedikate "not", "var", "nonvar" und "is" in die an tracemode gebundene Liste aufgenommen werden: [48]

```
(def spy
  (nlambda (names)
    (cond ((check-names names)
           (if (eq tracemode 'break-only) (setq tracemode nil))
           (if (null names)
               (setq tracemode
                     (union (union functions predicates)
                            (union '(is not var nonvar)
                                   tracemode)))
               (setq tracemode (union names tracemode)))
           (trace-lisp-functions functions)
           (if (and (null tracemode) breakmode)
               (setq tracemode 'break-only)
               tracemode)))))
```

---

[48] Die Funktion check-names liefert genau dann nil, wenn die Argumentliste Namen enthaelt, die nicht in functions, predicates oder der Liste der LISPLOG-Primitive ("is", "not", "var" und "nonvar") vorkommen. In einem solchen Fall wird von check-names auch eine entsprechende Fehlermeldung erzeugt.

```
(def nospy
  (nlambda (names)
    (cond ((not (check-names names)) nil)
          ((eq tracemode 'break-only))
          (t (if (null names)
                 (setq tracemode nil)
                 (setq tracemode (remlist names tracemode)))
             (cond ((and (null tracemode) breakmode)
                    (setq tracemode 'break-only))
                   ((null tracemode)
                    (untrace-lisp-functions functions)))))))
```

In aehnlicher Weise kann dann auch das in Abschnitt 7.4. eingefuehrte Konzept der Programmunterbrechung durch Breakpoints auf den funktionalen Teil ausgedehnt werden: Zu den bisherigen Ports CALL, EXIT, FAIL und REDO kommen nun noch die Ports EVAL und VALUE fuer LISP-Funktionen hinzu, an denen auch Breakpoints gesetzt werden koennen. Dementsprechend sind auch die Beschreibungen der Kommandos "brk" und "nobrk" in Abschnitt 7.4. (Seite 47) auch auf LISP-Funktionen und die neuen Ports EVAL und VALUE auszuweiten.

Mit diesen Darstellungen ist die Erweiterung des Box-Modells fuer Funktionen vollstaendig in die bestehende Interaktionsumgebung integriert, wie auch das abschliessende Dialogbeispiel zeigt, dem die schon in den Abschnitten 8.2. und 8.3. angesprochenen Datenbasen zur Fakultaetsberechnung (jetzt korrekt mit Cut!)

```
(4')  (ass !(fak 0 1))
      (ass (fak _n _nfak) (is _n1 (sub1 _n))
                          (fak _n1 _n1fak)
                          (is _nfak (multiply _n1fak _n)))
      (defun multiply (x y) (times x y))
```

und zum Loeschen eines Elements aus einer Zahlenliste

```
(1')  (ass (delete _x nil nil))
      (ass (delete _x (_x . _l) _m) (delete _x _l _m))
      (ass (delete _x (_y . _l) (_y . _m)) (my-lessp _x _y)
                                           (delete _x _l _m))
      (ass (delete _x (_y . _l) (_y . _m)) (my-greaterp _x _y)
                                           (delete _x _l _m))
      (defun my-lessp (x y) (lessp x y))
      (defun my-greaterp (x y) (greaterp x y))
```

zugrundeliegen. Waehrend bei der Fakultaets-Datenbasis (4') auf die Systemfunktion "sub1" durchgegriffen wird, die vom Kommando "spy" unberuehrt bleibt, koennen Informationen ueber die Auswertung der benutzerdefinierten LISP-Funktion "multiply" sowie bei der Abarbeitung der zweiten Datenbasis (1') auch Ablaufinformationen ueber die Auswertung von "my-lessp" und "my-greaterp" ausgegeben werden:

```
1.<lisplog>                        [Aufruf von LISPLOG aus FRANZ LISP]
*<consult fakneu.db>                 [Laden der Fakultaet-Datenbasis]
[load fakneu.db]
(fakneu.db)
*<listing fak>       [Anzeigen aller Klauseln zum Praedikat "fak"]
(ass !(fak 0 1))
(ass (fak _n _nfak)
     (is _n1 (sub1 _n))
     (fak _n1 _n1fak)
     (is _nfak (multiply _n1fak _n)))
*<spy>           [Alle Praedikate und Benutzer-Funktionen "tracen"]
(multiply fak is not var nonvar)                       [= tracemode]
*<(fak 3 _x)>    [LISPLOG-Anfrage: "Was ist die Fakultaet von 3?"]
|CALL (fak 3 _x)
| CALL (is _n1-1 (sub1 3))
| EXIT (is 2 (sub1 3))      [Keine Ablaufinformationen zu "sub1"!]
| CALL (fak 2 _n1fak-1)
|  CALL (is _n1-2 (sub1 2))
|  EXIT (is 1 (sub1 2))
|  CALL (fak 1 _n1fak-2)
|   CALL (is _n1-3 (sub1 1))
|   EXIT (is 0 (sub1 1))
|   CALL (fak 0 _n1fak-3)
|   EXIT (fak 0 1)
|   CALL (is _nfak-3 (multiply 1 1))
|    EVAL (multiply 1 1)    [Aufruf der LISP-Funktion "multiply"]
|    VALU 1          [Ergebnis ist 1; VALUE durch VALU abgekuerzt!]
|   EXIT (is 1 1)
|  EXIT (fak 1 1)
|  CALL (is _nfak-2 (multiply 1 2))
|   EVAL (multiply 1 2)
|   VALU 2
|  EXIT (is 2 2)
| EXIT (fak 2 2)
| CALL (is _nfak-1 (multiply 2 3))
|  EVAL (multiply 2 3)
|  VALU 6
| EXIT (is 6 6)
|EXIT (fak 3 6)
success
(_x  =  6)
More? (y or n) <y> [Datenbasis korrekt: Es gibt nur eine Loesung]
|REDO (fak 3 _x)
| REDO (fak 2 _n1fak-1)
|  REDO (fak 1 _n1fak-2)
|   REDO (fak 0 _n1fak-3)
|   FAIL (fak 0 _n1fak-3)              [Hier wird der Cut wirksam!]
|  FAIL (fak 1 _n1fak-2)
| FAIL (fak 2 _n1fak-1)
|FAIL (fak 3 _x)
nil
*<consult delete.db>       [Laden der zweiten Beispiel-Datenbasis]
(delete.db)
```

```
*<listing my-lessp>             [Anzeigen der Funktion "my-lessp"]
(defun my-lessp (x y) (lessp x y))
*<listing delete>               [Anzeigen des Praedikats "delete"]
(ass (delete _x nil nil))
(ass (delete _x (_x . _l) _m) (delete _x _l _m))
(ass (delete _x (_y . _l) (_y . _m)) (my-lessp _x _y)
                                     (delete _x _l _m))
(ass (delete _x (_y . _l) (_y . _m)) (my-greaterp _x _y)
                                     (delete _x _l _m))
*<spy delete my-lessp>       [Nur "delete" und "my-lessp" ausgeben]
(my-lessp delete multiply fak is not var nonvar)
*<brk (my-greaterp fail eval)>    [Breakpoints an FAIL und EVAL-
((my-greaterp fail eval))                Ports von "my-greaterp"]
*<(delete 7 (2 7 4 9) _ergebnis)>      [7 aus (2 7 4 9) loeschen]
|CALL (delete 7 (2 7 4 9) _ergebnis)
| CALL (my-lessp 7 2)
|  EVAL (my-lessp 7 2)
|  VALU nil
| FAIL (my-lessp 7 2)
|  EVAL (my-greaterp 7 2)       [Anzeige erfolgt wegen Breakpoint]
***** LISPLOG - Break - Level *****          [Breakpoint erreicht]
Please enter Break-command: <GOAL> [Anzeigen des aktuellen Ziels]
The actual goal is: (my-greaterp 7 2)     [Das ist eine Funktion!]
Please enter Break-command: <INFO>
spy active for: (my-lessp delete multiply fak is not var nonvar)
brk active for: ((my-greaterp fail eval))
Please enter Break-command: <nobrk (my-greaterp eval))>
((my-greaterp fail))                                [= breakmode]
Please enter Break-command: <CONTINUE>
LISPLOG continues...
| CALL (delete 7 (7 4 9) _m-1)
|  CALL (delete 7 (4 9) _m-2)
|   CALL (my-lessp 7 4)
|    EVAL (my-lessp 7 4)
|    VALU nil
|   FAIL (my-lessp 7 4)
|   CALL (delete 7 (9) _m-3)
|    CALL (my-lessp 7 9)
|     EVAL (my-lessp 7 9)
|     VALU t
|    EXIT (my-lessp 7 9)
|    CALL (delete 7 nil _m-4)
|    EXIT (delete 7 nil nil)
|   EXIT (delete 7 (9) (9))
|  EXIT (delete 7 (4 9) (4 9))
| EXIT (delete 7 (7 4 9) (4 9))
|EXIT (delete 7 (2 7 4 9) (2 4 9))
success:
(_ergebnis = (2 4 9))
More? (y or n) <n>
t
*<lisp>                         [Verlassen des LISPLOG-Toplevel]
2.                              [Zurueck im FRANZ-LISP-Toplevel]
```

9. Modularisierung der Interaktionsumgebung

In den vorangegangenen Darstellungen haben wir bereits mehrfach Effizienzgesichtspunkte aufgegriffen. Im Vordergrund stand dabei die Unterscheidung zwischen aktiver und inaktiver Interaktionsumgebung ueber die globale Variable tracemode (nil = inaktiv).

Die besondere Behandlung fuer den Fall einer inaktiven Interaktionsumgebung wirkt sich sowohl auf die Speichereffizienz (keine Sammlung von Ablaufinformationen im backtrace-stack) als auch auf die Laufzeiteffizienz (die meisten Aktionen nur bei tracemode ungleich nil) aus.

Damit sind die Moeglichkeiten zur Effizienzsteigerung bei nicht aktiver Interaktionsumgebung aber noch laengst nicht erschoepft:

9.1. Auslagerung des Trace-Pakets

So haben wir in Kapitel 6 den Kern des LISPLOG-Interpreters speziell fuer die Herleitung der Trace-Informationen manipuliert. Durch die Aufnahme der aktiven Ziele in die list-of-goals ist nun aber die neue Version des LISPLOG-Interpreter-Kerns auch bei inaktiver Interaktionsumgebung ineffizienter als die urspruengliche Version (Seiten 21/22). Und dies gilt sowohl bezueglich Speicherbedarf (aktive Ziele in der list-of-goals) als auch Laufzeiteffizienz (laengere Zugriffszeiten und Aufwand zum Entfernen der aktiven Ziele).

Deshalb erscheint es sinnvoll, die Interaktionsumgebung so zu modularisieren, dass bei inaktiver Interaktionsumgebung der urspruengliche Kern des LISPLOG-Interpreters benutzt wird und erst bei Aktivierung der Interaktionsumgebung durch ein beliebiges "spy"- oder "brk"-Kommando auch die Repraesentation der list-of-goals entsprechend geaendert wird.

Konkret geschieht das dadurch, dass die Funktionen redo-boxes und exit-boxes zunaechst bei inaktiver Interaktionsumgebung gemaess ihrer Wirkung im urspruenglichen LISPLOG-Interpreter definiert werden:

```
(def exit-boxes
  (lambda (list-of-goals environment)
    list-of-goals))

(def redo-boxes
  (lambda (list-of-goals environment)
    nil))
```

Ausserdem wird in der Funktion and-process (vgl. Seite 38) jetzt das ausgewaehlte und als naechstes zu beweisende Ziel nicht mehr direkt durch

```
(let* ((goal ...)
       (goals-left (cons (cons goal 'active) new-list-of-goals)))
      ...)
```

als "aktiv" markiert, sondern die Markierung durch Aufruf einer eigenen Funktion get-rest-of-goals ausgelagert:

```
(let* ((goal ...)
       (goals-left (get-rest-of-goals goal new-list-of-goals)))
      ...)
```

Damit kann bei inaktiver Interaktionsumgebung auch die urspruengliche Bedeutung der Funktion get-rest-of-goals (vgl. Seite 21) ohne Aufnahme und Markierung aktiver Ziele verwendet werden:

```
(def get-rest-of-goals
  (lambda (goal list-of-goals)
    (rest list-of-goals)))
```

Erst bei Aktivierung der Interaktionsumgebung werden dann die neue Definition fuer get-rest-of-goals

```
(def get-rest-of-goals
  (lambda (goal new-list-of-goals)
    (cons (cons goal 'active) new-list-of-goals)))
```

sowie die Definitionen der Funktionen exit-boxes und redo-boxes (vgl. Seite 37) nachgeladen.

Auf diese Weise ist dann bei inaktiver Interaktionsumgebung eine effizientere Abarbeitung moeglich, da die urspruengliche Repraesentation der list-of-goals ohne aktive Ziele beibehalten wird.

Natuerlich koennen nun auch die Schnittstellenfunktionen box-call und box-fail entsprechend ohne Aktion definiert werden:

```
(def box-call
  (lambda (goal) nil))

(def box-fail
  (lambda (goal) nil))
```

Die Funktionen box-redo und box-exit werden ohnehin nur von den Funktionen exit-boxes bzw. redo-boxes und somit nur bei aktiver Interaktionsumgebung aufgerufen. Auch die Funktionen box-eval und box-value werden ja nur aufgerufen, falls die entsprechenden Definitionen der benutzerdefinierten LISP-Funktionen schon zuvor durch die Interaktionsumgebung entsprechend manipuliert worden sind (vgl. Abschnitt 8.4.).

Ausserdem kann nun bei einer inaktiven Interaktionsumgebung auch auf das Laden eines Grossteils des Codes fuer die Interaktionsumgebung ganz verzichtet werden, wodurch die Speichereffizienz

noch einmal deutlich verbessert werden kann. Erst bei einer Aktivierung der Interaktionsumgebung sind dann auch noch die restlichen Funktionen zu laden und die Definitionen der Funktionen get-rest-of-goals, exit-boxes, redo-boxes, box-call und box-fail zu ueberlagern. Wird dann die Interaktionsumgebung wieder einmal inaktiv, so kann durch erneute (Rueck-)Ueberlagerung dieser und weiterer nicht mehr benoetigter Funktionen auch wieder Speicherraum freigegeben werden, da eine eventuell notwendig werdende Speicherbereinigung (garbage collection) den somit nicht mehr erreichbaren Code wieder als freien Speicherraum zur Verfuegung stellt.

Die Funktionsdefinitionen fuer die inaktive Interaktionsumgebung werden also in einer Datei "notrace" zusammengefasst und beim Start des LISPLOG-Systems mitgeladen (vgl. Anhang B). Natuerlich muss nun in den Funktionen spy und brk dafuer gesorgt werden, dass die Funktionsdefinitionen fuer die aktive Interaktionsumgebung, die in einer eigenen Datei "trace" zusammengefasst sind, nachgeladen werden:

```
(def spy                                ; brk analog (vgl. Anhang B)
  (nlambda (names)
    (cond ((check-names names)
           (load-tracer)                      ; Laden des Trace-Pakets
           (if (eq tracemode 'break-only) (setq tracemode nil))
           (if (null names)
               (setq tracemode
                     (union (union functions predicates)
                            (union '(is not var nonvar)
                                   tracemode)))
               (setq tracemode (union names tracemode)))
           (trace-lisp-functions functions)
           (if (and (null tracemode) breakmode)
               (setq tracemode 'break-only)
              tracemode)))))

(def load-tracer
  (lambda nil
    (cond (tracer-loaded)
          (toplevel-flag (load "trace")
                         (setq tracer-loaded t)))))
```

Die Kommandos "skip", "skipq" und "noskip" sind von diesen Aenderungen nicht betroffen: durch diese Kommandos wird ja die Interaktionsumgebung nicht aktiviert! Deshalb koennen auch in den fertigen Produktionsversionen eventuelle "skip"-Kommandos in den Dateien verbleiben (vgl. Abschnitt 7.6.), die dann erst beim Debugging wirksam werden.

Auch durch das "cut"-Kommando wird die Interaktionsumgebung nicht direkt aktiviert, so dass keine Erweiterung zum Nachladen der entsprechenden Dateien notwendig ist. Erst wenn der Benutzer vom Hand-Schneider aus in den Break-Level gehen moechte (Eingabe 'b',

vgl. Abschnitt 7.8.), muss schliesslich die volle Interaktionsumgebung [49] nachgeladen werden.

Wird die Interaktionsumgebung dann zu irgendeinem Zeitpunkt durch die Kommandos "nobrk" bzw. "nospy" wieder ausgeschaltet, so koennen dann wieder die alten Funktionsdefinitionen geladen werden, wobei jedoch sichergestellt sein muss, dass gerade kein LISPLOG-Beweis aktiv ist, das entsprechende "nospy" oder "nobrk"-Kommando also auf dem LISPLOG-Toplevel eingegeben [50] wurde:

```
(def nospy                             ; nobrk analog (vgl. Anhang B)
  (nlambda (names)
    (cond ((not (check-names names)) nil)
          ((eq tracemode 'break-only))
          (t (if (null names)
                 (setq tracemode nil)
                 (setq tracemode (remlist names tracemode)))
             (cond ((and (null tracemode) breakmode)
                    (setq tracemode 'break-only))
                   ((null tracemode)
                    (untrace-lisp-functions functions)
                    (unload-tracer)))))))

(def unload-tracer
  (lambda nil
    (cond ((and tracer-loaded toplevel-flag)
           (untrace-lisp-functions functions)
           (load "untrace")
           (setq tracer-loaded nil))
          (tracer-loaded
           (princ "Note: The tracer is no more activated ")
           (princ "for any predicate or function but may be")
           (terpr)
           (princ "      unloaded only when given the ")
           (princ "'nospy' or 'nobrk' command at toplevel!")
           (terpr)))))
```

---

[49] Obwohl der Hand-Schneider selbstverstaendlich mit zur Interaktionsumgebung gehoert, wird der entsprechende Code hierfuer beim Start von LISPLOG direkt mitgeladen. Der Hand-Schneider wird also nicht erst bei Bedarf (z.B. durch das "cut"-Kommando) nachgeladen! Das ist deshalb unproblematisch, weil der Hand-Schneider einerseits keinen Gebrauch von der geaenderten Repraesentation der list-of-goals macht und andererseits auch umfangmaessig nicht ins Gewicht faellt, so dass der entsprechende Code problemlos permanent geladen sein kann.

[50] Dies kann ueber eine entsprechende Variable toplevel-flag entschieden werden (vgl. Anhang B).

Die in diesem Abschnitt beschriebene Modularisierung in Form der Dateien "notrace", "trace" und "untrace" stellt sicher, dass bei einer inaktiven Interaktionsumgebung der Mehraufwand an Speicher und Laufzeit durch die Interaktionsumgebung wirklich auf das absolute Minimum (Bereitstellung der Funktionen spy, nospy, brk usw.) reduziert wird.

Dieses Resultat ist ausserordentlich wichtig, weil damit gewaehrleistet ist, dass die hier vorgestellte LISPLOG-Version inklusive Interaktionsumgebung sowohl als komfortable Entwicklungsumgebung (bei Benutzung der angebotenen Interaktionsumgebung) als auch als effiziente Ablaufumgebung (bei inaktiver Interaktionsumgebung) eingesetzt werden kann (vgl. Kapitel 2).

9.2. Auslagern des Break-Pakets

Fuer den ersten Fall, dass der Anwender von der Interaktionsumgebung Gebrauch macht, bietet sich nun aber noch eine Moeglichkeit zur weitergehenden Modularisierung an, die wir im folgenden auch noch darstellen wollen:

Nach den bisherigen Darstellungen bewirkt jede einfache Ablaufverfolgung (Trace) durch ein "spy"-Kommando sofort das Nachladen der gesamten Interaktionsumgebung. Bei naeherer Betrachtung stellen wir jedoch fest, dass ein grosser Teil davon nur fuer die Realisierung des LISPLOG-Break-Level, also nicht fuer die eigentliche Ablaufprotokollierung auf dem Monitor, benoetigt wird.

Daraus ergibt sich nun ein Ansatz zur weiteren Modularisierung der Interaktionsumgebung, indem aus dem bisher in der Datei "trace" zusammengefassten Code nun der Teil, der nur fuer die Realisierung des LISPLOG-Break-Level benoetigt wird, in eine eigene Datei ausgelagert wird.

Somit braucht bei einem "spy"- oder "brk"-Kommando zunaechst nur das Trace-Paket (die Datei "trace") nachgeladen zu werden. Erst bei einer Aktivierung des LISPLOG-Break-Level durch eine interaktive Programmunterbrechung (CTRL-C) oder beim Erreichen eines Breakpoints wird dann in der Funktion <u>breakpoint</u>

```
(def breakpoint
  (lambda nil
    (cond (breakflag (load-breaker) (breaking)))))
```

die Datei "break" durch die Funktion <u>load-breaker</u> nachgeladen:

```
(def load-breaker
  (lambda nil
    (cond (breaker-loaded)
          (t (load "break")
             (setq breaker-loaded t)))))
```

Dieses Nachladen des Break-Pakets muss natuerlich auch dann erfolgen wenn durch Eingabe von 'b' der LISPLOG-Break-Level direkt vom Hand-Schneider oder Skipper aus aktiviert wird. Dazu sind in den Funktionen manual-cutter, make-cut-permanent? (beide in der Datei "notrace") sowie skip? (in der Datei "trace") entsprechende Aufrufe der Funktion load-breaker einzufuegen. [51]

Selbstverstaendlich macht es keinen Sinn, diese Funktionen des Break-Pakets nach Verlassen des LISPLOG-Break-Level wieder aus dem Speicher zu entfernen, wie es zum Beispiel durch eine Ueberlagerung mit leeren Funktionsdefinitionen der Form

```
(def <function-name> (lambda nil nil))
```

und eine nachfolgende Speicherbereinigung (garbage collection) erfolgen koennte. Bei einer solchen Vorgehensweise wuerde bei einer interaktiven Programmentwicklung ein Grossteil der Abarbeitungszeit durch das Laden und anschliessende Ueberlagern des Break-Pakets verschlungen werden.

Deshalb bleibt das Break-Paket, wenn es einmal geladen wurde, zunaechst auch weiterhin geladen. Da es aber ohnehin nur als Ergaenzung zum Trace-Paket benoetigt und geladen wird, bietet es sich allerdings an, bei einer Deaktivierung der Interaktionsumgebung schliesslich neben den Funktionen des Trace-Pakets auch die Funktionen des Break-Pakets in der oben beschriebenden Weise durch Laden der Datei "untrace" zu entfernen.

Somit erhalten wir fuer die Interaktionsumgebung insgesamt vier Dateien, die auch komplett im Anhang B abgedruckt sind:

- Die Datei "notrace" wird mit dem LISPLOG-System geladen und enthaelt im wesentlichen die Realisierung der Kommandos "spy", "nospy", "brk", "nobrk", "cut", "nocut", "skip", "skipq" und "noskip" sowie die Organisation des Nachladens von Trace- und Break-Paket.

- Die Datei "trace" enthaelt das eigentliche Trace-Paket.

- Die Datei "break" enthaelt das eigentliche Break-Paket.

- Die Datei "untrace" schliesslich enthaelt die Redefinitionen zur Entfernung des Trace- und Break-Pakets.

Eine noch weitergehende Modularisierung bietet sich sicher nicht an, da in diesem Fall der Verwaltungsaufwand (Laden, Entladen) den moeglichen Gewinn deutlich uebersteigen duerfte.

---

[51] Der resultierende Code ist in Anhang B zu finden.

10. Integration mit anderen LISPLOG-Interpretern

Die in den vorangegangenen Kapiteln beschriebene Interaktionsumgebung arbeitet sowohl mit dem urspruenglichen, im Kern rekursiv formulierten (and-process und or-process) Interpreter LISPLOG.II als auch mit dem effizienteren iterativen Interpreter LISPLOG.2 (vgl. [Dahmen 1986]) zusammen, ohne dass dafuer irgendwelche Aenderungen am Code der Interaktionsumgebung (Dateien "notrace", "trace", "break" und "untrace") erforderlich waeren.

Diese Universalitaet der Interaktionsumgebung bzgl. des zugrundeliegenden LISPLOG-Interpreters wurde erst moeglich durch die exakte Definition der Schnittstellen zwischen der Interaktionsumgebung und dem uebrigem LISPLOG-System, was wir bereits in Abschnitt 7.1. angesprochen und dann bei der nachfolgenden Implementierung konsequent durchgefuehrt haben.

Bei den Darstellungen in dieser Arbeit sind wir wegen der kompakteren und vor allem leichter verstaendlicheren Formulierung von einer Integration mit dem rekursiven LISPLOG.II-Interpreter ausgegangen. An einigen Stellen haben wir darueberhinaus angedeutet, welche Besonderheiten bei einer Integration mit dem iterativen LISPLOG.2-Interpreter bzgl. Verfuegbarkeit bestimmter Informationen fuer den LISPLOG-Break-Level und der Verwaltung des Interrupt-Handlers (vgl. Seite 52) zu beruecksichtigen sind.

Grundsaetzlich kann die beschriebene Interaktionsumgebung aber auch mit anderen LISPLOG-Interpretern zusammenarbeiten, soweit diese jeweils

- die von der Interaktionsumgebung definierten Schnittstellenfunktionen mit entsprechenden Argumenten versorgt aufrufen (box-call, box-fail, box-exit, box-redo, box-cut-p, box-cut und tracer-init),

- die von der Interaktionsumgebung benoetigten Zugriffsmoeglichkeiten auf Datenstrukturen (z.B. Bindungsumgebung) und Operationen (z.B. Unifikation) bereitstellen (trace-unify und trace-unify-p),

- dem Benutzer den Aufruf der Kommandofunktionen zur Steuerung der Interaktionsumgebung ermoeglichen (spy, nospy, brk, nobrk, skip, skipq, noskip, cut und nocut),

- eine interaktive Programmunterbrechung (CTRL-C) durch Setzen der globalen Variable breakflag (vgl. S. 51) der Interaktionsumgebung mitteilen,

- die globalen Variablen toplevel-flag (vgl. S. 89), predicates (vgl. S. 44), functions (vgl. S. 81) und version (zur Identifikation des verwendeten LISPLOG-Interpreters) in der von der Interaktionsumgebung erwarteten Weise verwalten und

- ausserdem einige wenige Funktionen zur Verwendung abstrakter Syntax fuer LISPLOG-Programme (s-conclusion, s-premises) und fuer die externe Darstellung bei der Ausgabe von LISPLOG-Code (external-form, print-external) bereitstellen.

Unter Beruecksichtigung dieser Voraussetzungen kann die vorgestellte Interaktionsumgebung also auch mit einem anderen, eventuell erweiterten oder verbesserten LISPLOG-Interpreter zusammenarbeiten.

11. Ausblick: Erweiterungen und Verbesserungen

Die Interaktionsumgebung, wie sie in den Kapiteln 7, 8 und 9 beschrieben wurde, ist vollstaendig implementiert und lauffaehig.

Der praktische Einsatz von LISPLOG und der Interaktionsumgebung im Rahmen von Projektarbeiten und Praktika hat gezeigt, dass die hier entwickelte Interaktionsumgebung einen wichtigen Beitrag zur Unterstuetzung des Benutzers beim Debugging von LISPLOG-Programmen leisten kann.
Zugleich wurde dabei aber auch deutlich, dass die angebotene Programmierumgebung keineswegs perfekt war oder ist. Vielmehr haben praktische Einsaetze ausserhalb des Kreises der LISPLOG-Entwickler auch einige wertvolle Anregungen zur Verbesserung der Interaktionsumgebung hervorgebracht, von denen einige bereits in diese Arbeit eingeflossen und somit schon realisiert worden sind. [52]

Einige andere Moeglichkeiten zur Verbesserung oder Erweiterung der Interaktionsumgebung sollen hier nun abschliessend kurz angesprochen werden, auch wenn ihre Realisierung derzeit noch nicht abgeschlossen ist bzw. noch gar nicht in Angriff genommen wurde:

11.1. Erweiterung zum Trace der Unifikationen

Wegen der hierfuer besonders gut geeigneten Repraesentation der Bindungsumgebung als Liste von Variable-Wert-Paaren bietet sich zumindest fuer den rekursiven Interpreter LISPLOG.II auch eine Darstellung der Unifikationsergebnisse im Trace an. Solche Unifikationsergebnisse koennen sein:

(1) Das aktuelle Teilziel (goal) ist mit einem Klauselkopf unifizierbar. Dabei entsteht ein Unifikator (unifier) der zur bisherigen Bindungsumgebung (environment) hinzugefuegt wird.

---

[52] So geht zum Beispiel die Entwicklung des Skippers zur Ausblendung von Teilberechnungen (vgl. Abschnitt 7.6.) auf eine Anregung aus dem Teilnehmerkreis des UNIXPERT-Praktikums zurueck.

(2) Das aktuelle Teilziel (goal) ist mit keinem Klauselkopf mehr unifizierbar.

(3) Nach dem Fehlschlagen aller Versuche, das aktuelle Teilziel (goal) mit einem entsprechenden Klauselkopf zu unifizieren, erfolgt Backtracking zu einem frueheren "aktiven Ziel" (vgl. Seite 28), wobei die zwischenzeitlich erfolgten Variablen-bindungen, also der letzte Unifikator (unifier) wieder aus der aktuellen Bindungsumgebung entfernt wird.

Der LISPLOG-Interpreter kann in jedem dieser Faelle entsprechende Ablaufinformationen durch einen Aufruf geeigneter Schnittstellen-funktionen

(1) (box-unify-success <goal> <unifier>)

(2) (box-unify-fail <goal>) bzw.

(3) (box-backtrack <goal> <unifier>)

erzeugen. Aus diesen Ablaufinformationen kann dann der Debugger je nach Steuerung durch entsprechende neue oder auch erweiterte LISPLOG-Toplevel-Kommandos die auszugebenden Informationen selektieren und dem Benutzer zur Verfuegung stellen.

Die Erzeugung der Ablaufinformationen fuer diese Erweiterung laesst sich durch geringfuegige Ergaenzungen an der Funktion or-process realisieren, wobei wir davon Gebrauch machen, dass der Unifikator, der bei erfolgreichem Ergebnis der Unifikation entsteht, zur aktuellen Bindungsumgebung hinzugefuegt und bei spaeterem Backtracking ggfs. wieder zurueckgenommen werden muss, durch eine Funktion get-unifier

```
(def get-unifier
  (lambda (new-env old-env)
    (if (eq new-env old-env)
        nil
        (cons (first new-env)
              (get-unifier (rest new-env) old-env)))))
```

als Differenz zwischen neuer und alter Bindungsumgebung erhalten werden kann.

Damit kann jetzt in der Funktion or-process im Anschluss an den Aufruf der Unifikation direkt der dabei entstandene Unifikator ermittelt und an eine Variable unifier gebunden werden. War die Unifikation erfolgreich, die entstandene neue Bindungsumgebung (new-environment) also nicht leer, dann kann der Unifikator mit box-unify-success der Interaktionsumgebung gemeldet werden (1). Beim Backtracking nach dem Scheitern des and-process-Aufrufes fuer die noch verbliebenen Ziele wird dann der gleiche Unifikator (unifier) durch die Funktion box-backtrack der Interaktionsumgebung gemeldet (3).

Ist schliesslich beim Aufruf von or-process die Menge der fuer die Unifikation mit dem aktuellen Ziel noch in Betracht kommenden Klauseln (database-left) erschoepft, so wird dies durch einen Aufruf der Funktion <u>box</u>-<u>unify</u>-<u>fail</u> der Interaktionsumgebung gemeldet (2):

```
(def or-process
  (lambda (database-left goals-left goal environment level)
    (cond
;;
;; Fall (2):
;; ---------
;;
     ((null database-left) (box-unify-fail goal))
     (t (let* ((assertion (rename-variables (first database-left)
                                            (list level)))
               (new-goals (s-premises assertion))
               (head (s-conclusion assertion))
               (new-environment (unify goal head environment))
               (unifier (get-unifier new-environment
                                     environment)))
              (cond
               ((null new-environment)
                (or-process (rest database-left)
                            goals-left
                            goal
                            environment
                            level))
;;
;; Fall (1):
;; ---------
               ((box-unify-success goal unifier))
               ((box-skip? goal new-goals))
               ((and-process (append new-goals goals-left)
                             new-environment
                             (add1 level)))
;;
;; Fall (3):
;; ---------
               ((box-backtrack goal unifier))
               ((or (cut-p assertion)
                    (box-cut-p goal database-left))
                (box-cut goal assertion (rest database-left)))
               (t (or-process (rest database-left)
                              goals-left
                              goal
                              environment
                              level))))))))
```

Diese Erweiterung zum Trace der Unifikationen liegt bereits als prototypische Implementierung vor und kann ausser zur Unterstuetzung beim Debugging auch sehr gut zur Verdeutlichung der Abarbeitungsstrategie von LISPLOG bzw. PROLOG eingesetzt werden.

So erhalten wir zum Beispiel fuer unsere "thronfolger"-Datenbasis die folgende um solche Unifikations-Informationen erweiterte Tracer-Ausgabe, aus der insbesondere auch die Verwaltung der Variablenbindungen beim Ablauf des PROLOG-Beweises sehr gut zu erkennen ist:

```
|CALL (thronfolger _gesuchte-person)
| unifier = {_gesuchte-person -> _x-1}.
| CALL (prinz _x-1)
|  unify: unifier is {_x-1 -> _x-2}.
|  CALL (frosch _x-2)
|   unify: unifier is {_x-2 -> kermit}.
|  EXIT (frosch kermit)
|  CALL (verzaubert kermit)
|   unify: unifier is {}.
|  EXIT (verzaubert kermit)
| EXIT (prinz kermit)
| CALL (erstgeboren kermit)
|  mismatch: no clause for (erstgeboren kermit).
| FAIL (erstgeboren kermit)
| REDO (prinz kermit)
|  REDO (verzaubert kermit)
|   backtracking: remove last unifier {}.
|  FAIL (verzaubert kermit)
|  REDO (frosch _x-2)
|   backtracking: remove last unifier {_x-2 -> kermit}.
|   mismatch: no clause for (frosch _x-2).
|  FAIL (frosch _x-2)
|  backtracking: remove last unifier {_x-1 -> _x-2}.
|  unify: unifier is {_x-1 -> charles}.
| EXIT (prinz charles)
| CALL (erstgeboren charles)
|  unify: unifier is {}.
| EXIT (erstgeboren charles)
|EXIT (thronfolger charles)
```

Somit traegt gerade auch diese Erweiterung mit dazu bei, mit der LISPLOG-Interaktionsumgebung zwei getrennte zielgruppenspezifische Zielsetzungen zu erfuellen:

Der LISPLOG-Anwender, der mit dem System seine eigenen Programme erstellt, wird durch die angebotenen Werkzeuge vom Tracer ueber das Break-Paket bis hin zu den Cut-Werkzeugen intensiv beim Debugging seiner funktional/logischen Programme unterstuetzt.

Aber auch ein LISPLOG- oder PROLOG-Neuling kann insbesondere mit den einfach zu bedienenden und sehr umfangreichen Trace-Werkzeugen zunaechst die Abarbeitungsstrategie des LISPLOG-Interpreters und die Wirkung der einzelnen Primitive wie Cut- oder is-Operator an einfachen Beispielen praktisch kennenlernen. An die Stelle der umfangreichen theoretischen Einarbeitung mit allen konkreten Einzelheiten oder des Versuchs, durch Ausprobieren verschiedener Alternativen zu dem gewuenschten Effekt zu kommen, was beides

sehr leicht zur Frustration und somit zu mangelnder Akzeptanz des Systems fuehren kann, kann jetzt das zielgerichtete Kennenlernen und Verstehen des Systems samt der fuer den Benutzer relevanten internen Vorgaenge anhand konkreter Beispiele treten.

So ist neben der allgemeinen Help-Funktion des LISPLOG-Toplevels (vgl. [Bernardi, Dahmen & Meyer 1987]) auch der Tracer ein wichtiges Instrument, um dem Benutzer die Einarbeitung mit dem System zu erleichtern. Und hierzu wiederum ist die Moeglichkeit, den Ablauf eines LISPLOG-Programms bis auf die Ebene der Unifikationen mitverfolgen zu koennen, ein durchaus wichtiger Beitrag.

11.2. Beschraenkung des Backtrace-Stack

Aus Effizienzgruenden ist eine andere Verbesserung interessant, die ebenfalls untersucht wird, ohne dass hier allerdings schon eine konkrete Implementierung (auch aus Platzgruenden) angegeben werden soll:

Bei der bisherigen Realisierung waechst der backtrace-stack proportional mit der Laenge des Beweises: mit jedem Schritt wird bei eingeschalteter Interaktionsumgebung eine weitere Ablaufinformation in den backtrace-stack aufgenommen.

Fuer die Bearbeitung umfangreicherer LISPLOG-Programme und dabei insbesondere bei Anfragen, deren Beweis sehr aufwendig ist, ist dieser Zustand nicht lange tragbar. Hier ist also ein Kompromiss zu finden zwischen dem Speicherbedarf fuer den backtrace-stack und der Moeglichkeit, auf moeglichst viele bzw. auf die wichtigsten und interessantesten Ablaufinformationen beim Backtrace zugreifen zu koennen. Fuer eine geringere Speicherbelastung durch die gesammelten Ablaufinformationen muessen also Einschraenkungen bezueglich der Verfuegbarkeit von Ablaufinformationen beim Backtrace in Kauf genommen werden.

Dazu soll hier ein Ansatz kurz skizziert werden, der einen guten Kompromiss zwischen diesen konkurrierenden Zielen darstellt:

Zur Entlastung des backtrace-stack werden Ablaufinformationen ueber abgeschlossene Teilberechnungen aus dem backtrace-stack ausgelagert, so dass der backtrace-stack im wesentlichen nur noch den Pfad vom gerade bearbeiteten Teilziel bis zur urspruenglichen Anfrage des Benutzers enthaelt.

Dazu werden beim Eintragen einer Ablaufinformation ueber das Verlassen einer Box (EXIT, FAIL oder VALUE) in den backtrace-stack (mit push-backtrace-stack) alle Informationen aus dem Inneren der damit verlassenen Box zu einem sog. "Stackframe" zusammengefasst. Diese aufeinanderfolgenden Eintraege im backtrace-stack werden dann durch eine Adresse ersetzt, unter der dieser Stackframe in einer separaten Datenstruktur aufbewahrt wird.

Ein Beispiel soll dieses Verfahren veranschaulichen: Bei der Abarbeitung der bekannten "thronfolger"-Anfrage (vgl. S. 13) werden die folgenden (hier numerierten) Ablaufinformationen in den backtrace-stack eingetragen:

```
 1   |CALL (thronfolger _gesuchte-person)
 2   | CALL (prinz _x-1)
 3   |  CALL (frosch _x-2)
 4   |  EXIT (frosch kermit)
 5   |  CALL (verzaubert kermit)
 6   |  EXIT (verzaubert kermit)
 7   | EXIT (prinz kermit)
 8   | CALL (erstgeboren kermit)
 9   | FAIL (erstgeboren kermit)
10   | REDO (prinz kermit)
11   |  REDO (verzaubert kermit)
12   |  FAIL (verzaubert kermit)
13   |  REDO (frosch _x-2)
14   |  FAIL (frosch _x-2)
15   | EXIT (prinz charles)
16   | CALL (erstgeboren charles)
17   | EXIT (erstgeboren charles)
18   |EXIT (thronfolger charles)
```

Bei den ersten beiden EXIT-Informationen (fuer "(frosch kermit)" und "(verzaubert kermit)") gibt es keine Informationen aus dem Inneren dieser Boxes; deshalb werden auch keine Stackframes erzeugt. Erst beim Verlassen der "prinz"-Box (Meldung 7) werden die vier obersten Ablaufinformationen auf dem backtrace-stack (Informationen 3 bis 6) zu einem Stackframe zusammengefasst, der in die angesprochene Datenstruktur ausgelagert und im backtrace-stack durch seine Adresse in dieser Datenstruktur ersetzt wird.

Im weiteren Verlauf der Abarbeitung werden dann noch zwei weitere Stackframes gebildet und ausgelagert:

Der naechste Stackframe besteht aus den Ablaufinformationen 11 bis 14 und wird beim naechsten Verlassen einer "prinz"-Box durch den EXIT-Ausgang (Meldung 15) aus dem backtrace-stack ausgelagert.

Beim Verlassen der "thronfolger"-Box schliesslich wird dann der naechste Stackframe gebildet, der nun aus den Ablaufinformationen 2 bis 17 besteht, wobei die Informationen 3 bis 6 und 11 bis 14 bereits durch zwei entsprechende Adressen ersetzt wurden, der neue Stackframe also nur 10 Eintraege enthaelt.

Nach Ersetzen dieser 10 Eintraege im backtrace-stack durch die Adresse des aus ihnen gebildeten Stackframe (Adresse-3) enthaelt der backtrace-stack nur noch 3 Eintraege:

- die CALL-Meldung der "thronfolger"-Box,

- die Adresse, wo naehere Informationen ueber die zugehoerige Teilberechnung zu finden sind, sowie

- die abschliessende EXIT-Meldung ueber den erfolgreichen Beweis der "thronfolger"-Anfrage.

Zusaetzlich zum backtrace-stack haben wir jetzt noch 3 Eintraege in der angesprochenen zusaetzlichen Datenstruktur, die wir jetzt backtrace-hunk nennen wollen:

```
------------------------------------------------------------
Adresse-1      |  CALL (frosch _x-2)
               |  EXIT (frosch kermit)
               |  CALL (verzaubert kermit)
               |  EXIT (verzaubert kermit)
---------------+--------------------------------------------
Adresse-2      |  REDO (verzaubert kermit)
               |  FAIL (verzaubert kermit)
               |  REDO (frosch _x-2)
               |  FAIL (frosch _x-2)
---------------+--------------------------------------------
Adresse-3      | CALL (prinz _x-1)
               | ---> Adresse-1
               | EXIT (prinz kermit)
               | CALL (erstgeboren kermit)
               | FAIL (erstgeboren kermit)
               | REDO (prinz kermit)
               | ---> Adresse-2
               | EXIT (prinz charles)
               | CALL (erstgeboren charles)
               | EXIT (erstgeboren charles)
------------------------------------------------------------
```

Dabei faellt auf, dass innerhalb von Eintraegen im backtrace-hunk bzw. backtrace-stack auch Adressen auftreten koennen, die ihrerseits auf andere Eintraege (Stackframes) referieren.

Der backtrace-stack enthaelt nun nicht mehr nur einfach die angesammelten Ablaufinformationen, sondern kann zugleich auch als Abstraktionsebene ueber diesen Ablaufinformationen verstanden werden. Mehr noch: Durch Umkehrung des Auslagerungsprozesses, also durch Ersetzen der Adressen im backtrace-stack durch ihre zugehoerigen Eintraege im backtrace-hunk, erhalten wir eine Konkretisierung und gelangen so auf eine tiefere Abstraktionsebene.

Mit diesen Mechanismen zur Abstraktion (Auslagerung von Stackframes) und Konkretisierung (Expansion der Adressen zu Stackframes) erhalten wir also zu den Ablaufinformationen einer LISPLOG-Anfrage stets eine wohldefinierte Abstraktionshierarchie, deren Hoehe der maximalen Einrueckungstiefe der Ablaufinformationen, also der maximalen "Tiefe" des Beweises, entspricht.

Fuer die Ablaufinformationen zu unserem "thronfolger"-Beispiel erhalten wir hier also eine dreistufige Abstraktionshierarchie:

Ausgehend vom dreielementigen backtrace-stack am Schluss des Beweises (hoechste Abstraktionsebene) gelangen wir durch Expansion der Adresse-3 zur naechstniedrigeren Abstraktionsebene. Durch weitere Expansion von Adresse-1 und Adresse-2 erhalten wir den vollstaendigen backtrace-stack, sind also auf der niedrigsten Abstraktionsebene angekommen.

Bildlich gesprochen kann man diese Expansionen also mit dem Auflegen einer Lupe auf die Ablaufinformationen gleichsetzen. Da nun auch die Umkehrabbildung (Auslagerung) definiert ist und zur Verfuegung steht, koennen wir diese Erweiterung des Box-Modell-Tracers auch als "Zoom-Box-Model-Tracer" auffassen, der das Wechseln zwischen Abstraktionsebenen beim Debugging gestattet.

Im Prinzip hatten wir einen aehnlichen Effekt, eine Abstraktion auf den Ablaufinformationen, bereits mit dem Skipper realisiert, wenn diese Abstraktion auch von Praedikat zu Praedikat ganz unterschiedlich definiert ("skip") und auch dynamisch noch interaktiv ("skipq") sehr differenziert festgelegt werden kann.

Die hier beschriebene Erweiterung fuehrt diesen Grundgedanken konsequent weiter und ermoeglicht auf der Basis dieser Abstraktionsebenen und der sie realisierenden Speicherungsstruktur auch eine Loesung des eingangs dargestellten Problems: einen annehmbaren Kompromiss zu finden zwischen Speicheraufwand und Informationsangebot beim Backtrace.

Da auch nach diesem ersten Ansatz zur Aenderung der Speicherungsstruktur nach wie vor alle Ablaufinformationen gesammelt und aufbewahrt werden, haben wir unser Ziel der Speicherplatzreduzierung noch nicht erreicht. Allerdings faellt auf, dass die Laenge des backtrace-stack jetzt nicht mehr direkt mit der Laenge des Beweises, sondern nur noch mit der "Tiefe" des Beweises multipliziert mit einem Faktor L fuer die maximale Laenge eines Stackframe waechst. [53]

---

[53] Diese Groesse L laesst sich bestimmen durch L = #P * #D, wobei #P die maximale Anzahl von Praemissen einer Klausel des betrachteten LISPLOG-Programms und #D die in der Praxis maximale Anzahl von Durchlaeufen durch einen beliebigen Box-Modell-Port des entsprechenden Programms bezeichnen. Zwar ist im Gegensatz zu #P die Groesse #D grundsaetzlich nicht beschraenkt; die Verwendung dieser Gleichung L = #P * #D in der hier angestellten Aufwandsabschaetzung laesst aber erkennen, von welchen Groessen nunmehr der Speicherbedarf abhaengt, so dass auch der Grad der Speicherplatzeinsparung sichtbar wird.

Wenn wir nun fuer den backtrace-hunk nur eine beschraenkte Anzahl von Eintraegen (Adressen) zulassen, ihn also zum Beispiel durch ein Array fester Laenge realisieren, so erhalten wir fuer den backtrace-hunk zunaechst einen konstanten Speicherplatzbedarf. Da aber jeder Eintrag im backtrace-hunk einen LISP-Ausdruck (= Pointer) darstellt, der in unserem Fall die Liste der unter einer bestimmten Adresse ausgelagerten Ablaufinformationen, also wiederum einen Stackframe, repraesentiert, haben wir auch hier beim Speicheraufwand noch den Faktor L zu beruecksichtigen.

Nach diesen etwas theoretischen Untersuchungen duerfte dennoch klar geworden sein, dass ganz wesentlich durch die Beschraenkung des backtrace-hunk auf eine feste Anzahl von Eintraegen der Speicheraufwand bei der Abarbeitung umfangreicher LISPLOG-Programme stark reduziert wird und der Aufwand fuer die Speicherung der Backtrace-Informationen (backtrace-stack plus backtrace-hunk) in der Praxis durch die Tiefe des Beweises beschraenkt ist.

Abschliessend ist hier nun noch die Frage zu klaeren, was zu tun ist, wenn bei der Abarbeitung eines groesseren LISPLOG-Programms schliesslich mehr Stackframes ausgelagert werden muessen als Eintraege im backtrace-hunk vorhanden sind.

Die naive Loesung, den backtrace-hunk zunaechst vollstaendig zu fuellen und dann alle spaeter noch auszulagernden Stackframes unberuecksichtigt zu lassen, ist sicher nicht akzeptabel: Hierbei wuerden gerade die juengsten, also fuer den Anwender wahrscheinlich wichtigsten Informationen zugunsten der aelteren Ablaufinformationen "weggeworfen" werden.

Sinnvoll ist dagegen eine zyklische Vergabe der Eintraege, so dass beim Ueberlauf des backtrace-hunk jeweils der aelteste in ihm aufbewahrte Stackframe ueberschrieben wird. Dadurch sind beim Backtrace immer die juengsten Ablaufinformationen vollstaendig vorhanden, waehrend aeltere Stackframes evtl. ueberschrieben werden.

Da es sich bei diesen aelteren Stackframes aber um Informationen ueber laengst abgeschlossene Teilberechnungen handelt und der Pfad von den im backtrace-hunk enthaltenen Stackframes zurueck zur urspruenglichen Benutzeranfrage auf jeden Fall im backtrace-stack enthalten ist, entspricht diese Vorgehensweise ziemlich genau dem, was wir von einer "intelligenten" Speicherverwaltung erwarten: Alte, inzwischen nicht mehr interessante Informationen werden durch neue aktuelle Informationen ersetzt, wobei jedoch die Voraussetzungen, auf denen die neuen Informationen basieren (Pfad im backtrace-stack), dennoch zugaenglich bleiben.

Bei der Implementierung ist hierbei natuerlich noch zu beachten, dass im backtrace-stack oder im backtrace-hunk selbst enthaltene Adressen durch die angesprochene zyklische Ueberlagerung ploetzlich ungueltig werden koennen.

Zur Loesung dieses Problems bietet sich nun ein Verfahren zur Verwaltung des backtrace-hunk und zur Kontrolle der Zugriffe an, das wir abschliessend ebenfalls kurz beschreiben wollen:

Die Adressen, durch die die ausgelagerten Informationen im backtrace-stack ersetzt werden, werden nicht zyklisch entsprechend der tatsaechlichen Speicherung im backtrace-hunk, sondern fortlaufend aufsteigend vergeben. Ausserdem wird in jedem Element des backtrace-hunk neben dem eigentlichen Stackframe auch die fuer ihn im backtrace-stack eingetragene fortlaufende Adresse LFD-ADR gespeichert. Die tatsaechliche Adresse HUNK-ADR im backtrace-hunk ergibt sich dann aus der fortlaufenden Adresse LFD-ADR durch

HUNK-ADR = LFD-ADR modulo MAX-ADR,

wobei MAX-ADR die Groesse des backtrace-hunk bezeichnet und von einer Adressierung des backtrace-hunk von 1 bis MAX-ADR ausgegangen wird.

Bei der Expansion von im backtrace-stack bzw. backtrace-hunk enthaltenen Adressen ADR ist dann zunaechst zu ueberpruefen, ob die im backtrace-hunk unter der Adresse (ADR modulo MAX-ADR) gespeicherte Kontrolladresse mit ADR uebereinstimmt. Ist das nicht der Fall, so ist die Adresse ADR inzwischen ungueltig geworden und die zugehoerigen Ablaufinformationen sind inzwischen ueberschrieben worden. Beim Backtrace kann dieser Umstand dann auch durch Ausgabe eines entsprechenden Hinweises

"Information purged by stack manager."

anstelle der urspruenglichen Ablaufinformationen dargestellt werden. Auch dieser Hinweis hat durchaus einen Informationsgehalt fuer den Benutzer: Aus einer Folge von Ablaufinformationen der Form

```
| ... CALL <goal>
| ...  Information purged by stack manager
| ... EXIT <goal>
```

geht naemlich zumindest hervor, dass das mit CALL aufgerufene Teilziel <goal> nicht durch Vorliegen des entsprechenden Faktums, sondern durch eine weitere moeglicherweise auch recht aufwendige Teilberechnung bewiesen wurde.

Mit dieser hier vorgestellten Erweiterung wird dann schliesslich auch ein Einsatz der Interaktionsumgebung zum Debugging umfangreicherer LISPLOG-Programme denkbar, der bisher aufgrund des direkt mit der Laenge des Beweises wachsenden Speicherbedarfs fuer die Ablaufinformationen kaum moeglich ist.

12. Literatur

[Bernardi 1986]

A. Bernardi: Ein Indexierungskonzept fuer LISPLOG-Datenbasen.
Universitaet Kaiserslautern, FB Informatik,
SEKI Working Paper SWP-86-10, Dezember 1986

[Bernardi, Dahmen & Meyer 1987]

A. Bernardi, M. Dahmen, M. Meyer: LISPLOG-Benutzerhandbuch.
Universitaet Kaiserslautern, FB Informatik,
SEKI Working Paper SWP-87-01, Januar 1987

[Bernardi & Kammermeier 1987]

A. Bernardi, F. Kammermeier: LISPLOG.2 - Implementation einer LISP/PROLOG-Vereinheitlichung.
In: W.-M. Lippe (Ed.): Tagungsband 4. Workshop "Alternative Konzepte fuer Sprachen und Rechner", Bad Honnef, 30.3.-1.4.1987,
Universitaet Muenster, Angewandte Mathematik und Informatik,
Bericht Nr. 2/87-1, Juni 1987

[Bernardi & Meyer 1987]

A. Bernardi, M. Meyer: LISPLOG auf dem IBM 6150.
Workshop "Logikprogrammierung fuer Expertensysteme und Sprachverarbeitung", IBM Heidelberg, 3.-4.11.1987

[Boley 1987a]

H. Boley: Fone and Fall: Forward-with-Backward Chaining in LISPLOG.
Universitaet Kaiserslautern, FB Informatik,
SEKI Working Paper SWP-87-03, Juni 1987

[Boley 1987b]

H. Boley: Goal: Backward-with-Forward Chaining in LISPLOG.
Universitaet Kaiserslautern, FB Informatik,
SEKI Working Paper SWP-87-04, Juni 1987

[Boley 1987c]

H. Boley: Frame and Heir: Clausal Frames and Multiple Inheritance in LISPLOG.
Universitaet Kaiserslautern, FB Informatik,
SEKI Working Paper SWP-87-09, November 1987

[Boley 1987d]

H. Boley (Ed.): A bird's-eye view of LISPLOG:
The LISP/PROLOG integration with initial-cut tools.
Universitaet Kaiserslautern, FB Informatik,
SEKI Working Paper SWP-86-08, Dezember 1986,
3rd edition: Dezember 1987

[Boley & Kammermeier et al. 1985]

H. Boley, F. Kammermeier u. die LISPLOG-Gruppe:
LISPLOG: Momentaufnahmen einer LISP/PROLOG-Vereinheitlichung.
Universitaet Kaiserslautern, FB Informatik,
MEMO SEKI-85-03, August 1985.
Kurzfassung in: B. Nebel (Ed.): Papiere zum Workshop Logisches
Programmieren und Lisp. TU Berlin, FB Informatik, KIT-Report 31,
Dezember 1985, pp. 36-53

[Boley & Meyer 1986]

H. Boley, M. Meyer: Implementation einer LISP/PROLOG-
Vereinheitlichung und ihrer Interaktionsumgebung.
In: F. Simon (Ed.): Implementierung von funktionalen
und logischen Programmiersprachen, Bericht Nr. 8603,
Institut fuer Informatik und Praktische Mathematik,
Christian-Albrechts-Universitaet Kiel, Mai 1986

[Bowen, Byrd, Pereira, Pereira & Warren 1981]

D.L. Bowen, L. Byrd, L.M. Pereira, F.C. Pereira, D.H. Warren:
Prolog on the DECSYSTEM-10, User's Manual.
Department of Artificial Intelligence,
University of Edinburgh, 1981

[Clocksin & Mellish 1981/84]

W. Clocksin, C. Mellish: Programming in PROLOG.
Springer Verlag, Berlin Heidelberg New York, 1981.
Second Edition 1984

[Dahmen 1985]

M. Dahmen: Ein Translator von CPROLOG nach LISPLOG.
in: [Dahmen, Herr, Hinkelmann & Morgenstern 1985], pp. 5-53

[Dahmen 1986]

M. Dahmen: Ein iterativer LISPLOG-Interpreter -
Implementierung, Dokumentation und Evaluation.
Universitaet Kaiserslautern, FB Informatik,
SEKI Working Paper SWP-86-03, Juni 1986

[Dahmen 1987]

M. Dahmen: Module und Streams in LISPLOG.
Universitaet Kaiserslautern, FB Informatik,
SEKI Working Paper SWP-87-06, Juni 1987

[Dahmen, Herr, Hinkelmann & Morgenstern 1985]

M. Dahmen, J. Herr, K. Hinkelmann, H. Morgenstern:
Beitraege zur LISP/PROLOG-Vereinheitlichung.
Universitaet Kaiserslautern, FB Informatik,
MEMO SEKI-85-10, November 1985

[Eimermacher 1983]

M. Eimermacher: Eine Programmierumgebung fuer PROLOG.
TU Berlin, FB Informatik, KIT - Interner Arbeitsbericht 6,
Juli 1983

[Eisenstadt 1985]

M. Eisenstadt: Retrospective Zooming: A knowledge-based
Tracing and Debugging Methodology for Logic Programming.
Proc. 9th IJCAI, Los Angeles, 1985, pp. 717-719

[Foderaro et al. 1983]

J. K. Foderaro, K. L. Sklower, K. Laver:
The FRANZ LISP Manual.
University of California, Berkeley, Juni 1983

[Grice 1967]

H. Grice: Logic and Conversation
in: P. Cole, J. Morgan: Speech Acts.
Academic Press, New York, 1967

[Herr 1985]

J. Herr: Breitensuche und Klauselcompilation fuer LISPLOG.
in: [Dahmen, Herr, Hinkelmann & Morgenstern 1985], pp. 126-172

[Herr 1986]

J. Herr: Ansatz fuer einen LISPLOG-Compiler mit LISP als
Zielsprache.
Universitaet Kaiserslautern, FB Informatik,
SEKI Working Paper SWP-86-06, September 1986

[Herr 1988]

J. Herr: SPADE - A Program Analyzing the Stability of Finite Difference Methods for the Numerical Solution of Linear Partial Differential Equations with Constant Coefficients.
Universitaet Kaiserslautern, FB Informatik, Diplomarbeit, erscheint demnaechst

[Herr & Meyer 1985]

J. Herr, M. Meyer: Interaktionsmoeglichkeiten zum Debugging.
in: [Boley & Kammermeier et. al. 1985], pp. 54-65

[Hinkelmann 1986]

K. Hinkelmann: Uebersetzung von LISPLOG-Programmen nach CPROLOG.
Universitaet Kaiserslautern, FB Informatik,
SEKI Working Paper SWP-86-05, August 1986

[Hinkelmann 1988]

K. Hinkelmann: SASLOG - Eine funktional-logische Sprachintegration mit Lazy Evaluation und semantischer Unifikation.
Universitaet Kaiserslautern, FB Informatik, Diplomarbeit, erscheint demnaechst

[Hinkelmann & Morgenstern 1985]

K. Hinkelmann, H. Morgenstern: Ein Verfahren zur Transformation von LISP-Funktionen in PROLOG-Relationen.
in: [Dahmen, Herr, Hinkelmann & Morgenstern 1985], pp. 54-125

[Hinkelmann, Noekel & Rehbold 1988]

K. Hinkelmann, K. Noekel, R. Rehbold: SASLOG - Lazy Evaluation Meets Backtracking.
Universitaet Kaiserslautern, FB Informatik, erscheint
[IF-PROLOG 1984]

U. Leibrandt, P. Folkjaer, L. Bernhard:
IF/Prolog Users's Manual Version 2.0,
Interface Computer GmbH, Muenchen, September 1984

[Kahn 1983/84]

K. M. Kahn: Pure PROLOG in Pure LISP.
Logic Programming Newsletter 5, Winter 83/84, pp. 3-4

[Kammermeier 1985]

F. Kammermeier: Dokumentation der Prolog-Implementation in Franz-Lisp.
Universitaet Kaiserslautern, FB Informatik, April 1985

[Kammermeier 1986]

F. Kammermeier: LISPLOG im Kontext anderer LISP/PROLOG-Vereinheitlichungen.
Universitaet Kaiserslautern, FB Informatik,
SEKI Working Paper SWP-86-09, Dezember 1986

[Lessel 1986]

M. Lessel: micro-UNIXPERT: Ein wissensbasiertes System zur Behandlung von Problemen bei UNIX-Druckauftraegen.
Universitaet Kaiserslautern, FB Informatik,
SEKI Working Paper SWP-86-04, Mai 1986

[Lessel & Boley 1987]

M. Lessel, H. Boley: micro-UNIXPERT: Diagnosis of Printer Problems.
Universitaet Kaiserslautern, FB Informatik,
SEKI Report SR-87-09, August 1987
Auch in: Proc. 12th Symposium Operations Research, Passau, 9.-11.9.1987.

[Meyer 1987a]

M. Meyer: The LISPLOG Interactive Programming Environment.
Workshop "Logisches Programmieren", Lohmar, 28.-30.9.1987
Auch in: [Boley 1987d], pp. 14-18

[Meyer 1987b]

M. Meyer: Entwicklung wissensbasierter Systeme mit LISPLOG.
Tagung "Wissensbasierte Systeme", Heringsdorf (Usedom) / DDR, 23.-27.11.1987
Erscheint in: Wissenschaftliche Zeitschrift der Technischen Hochschule Leipzig

[Robinson 1965]

J. A. Robinson: A Machine Oriented Logic Based on the Resolution Principle.
Journal of the ACM 12(1), pp. 23-41, 1965

[Wilensky 1984]

R. Wilensky: LISPcraft.
W. W. Norton & Co., New York, London, 1984

Anhang A: Die LISPLOG-Interaktionsumgebung auf einen Blick

(1) Toplevel-Kommandos zur Steuerung der Interaktionsumgebung:

```
spy     {pred|func}*                                  -> S. 44/88
nospy   {pred|func}*                                  -> S. 44/88
brk     {pred|func|(pred{port}*)|(func{port}*)}*      -> S. 47/83
nobrk   {pred|func|(pred{port}*)|(func{port}*)}*      -> S. 47/83
skip    {pred|func}*                                  -> S. 59
skipq   {pred|func}*                                  -> S. 60
noskip  {pred|func}*                                  -> S. 60
cut     {pred}*                                       -> S. 71
nocut   {pred}*                                       -> S. 71
```

mit: pred = Praedikatname oder Name eines LISPLOG-Primitivs
func = Name einer benutzerdefinierten LISP-Funktion
port = Box-Modell-Port (CALL,FAIL,EXIT,REDO,VALU,EVAL)

(2) Moegliche Antworten auf die Frage "More information desired?"

Eingabe:	Wirkung:
P oder +P	Ausgabe max. einer weiteren Bildschirm-Seite an Trace-Informationen
n (pos. Zahl)	Ausgabe max. n weiterer Schritte
0	Abbruch des LISPLOG-Programms
-P	Zurueckblaettern um max. eine Seite
-m (neg. Zahl)	Zurueckblaettern um max. m Schritte
b	Aufruf des LISPLOG-Break-Level

+ zusaetzlich beliebige Kommandos des LISPLOG-Toplevel!

(3) Break-Kommandos und zulaessige Abkuerzungen:

Break-Kommando:	zulaessige Abkuerzungen:
INFO	i I in IN inf INF
CONTINUE	c C co CO con CON cont CONT
RESET	r R re RE res RES
SKIPPING	s S
EVAL	v V
ENVIRONMENT	e E en EN env ENV
PATH	pa PA
LIST-OF-GOALS	L li LI lis LIS list LIST
GOAL	g G go GO goa GOA
(positive Antwort)	y Y yes Yes YES j J ja Ja JA
(negative Antwort)	n N no No NO nein Nein NEIN

+ zusaetzlich beliebige Antworten nach obiger Tabelle (2)
+ zusaetzlich beliebige Kommandos des LISPLOG-Toplevel!

Anhang B: Listings

Zur Ergaenzung der Darstellungen im Textteil wird hier der Quellcode des LISPLOG.II-Systems nochmals zusammenhaengend abgedruckt. Dabei verzichten wir allerdings auf diejenigen Teile des Systems, die nicht in direktem Zusammenhang mit der Interaktionsumgebung stehen, und beschraenken uns somit auf folgende Dateien:

Dateiname	Inhalt
absynt.l	Abstrakte Syntax
andor.l	Kern des LISPLOG-Interpreters
notrace.l	Kernfunktionen der Interaktionsumgebung
trace.l	Nachzuladendes Trace-Paket
break.l	Nachzuladendes Break-Paket
untrace.l	Redefinitionen zum Auslagern von Trace und Break
unify.l	Unifikation
interface.l	Benutzeroberflaeche, LISPLOG-Toplevel
nsolutions.l	Schnittstelle von LISP nach PROLOG
primitives.l	LISPLOG-Primitive: not, var, nonvar
lispeval.l	Durchgriff von PROLOG nach LISP: is-Operator
variable.l	Repraesentation von Variablen und Bindungen

Die hier nun abgedruckte Version des LISPLOG.II-Systems enthaelt bereits einige Erweiterungen, die wir zwar in Kapitel 1 schon einmal angesprochen haben (Indexierung, Modulsystem), deren Einbindung in den Kern des LISPLOG-Interpreters (and-process und or-process) wir aber bei der schrittweisen Darstellung der LISPLOG-Interaktionsumgebung vor allem aus Gruenden der Klarheit und Uebersichtlichkeit nicht beruecksichtigt haben. Aus diesem Grunde ist vor allem der hier abgedruckte Code der Funktion and-process sehr viel umfangreicher (und wohl auch schwerer verstaendlich) als die im Textteil verwendete reduzierte Version ohne Modul- und Indexierungssystem.

Auch am Code der Interaktionsumgebung selbst sind einige Unterschiede zu den Darstellungen im Textteil festzustellen: Einerseits folgt die tatsaechliche Implementierung der Erzeugung von Ablaufinformationen bei LISP-Funktionen dem im Abschnitt 8.4. nur am Rande angesprochenen Konzept auf der Basis des FRANZ-LISP-Tracers (vgl. S. 80). Andererseits sind durch die angesprochenen Erweiterungen auch einige, allerdings sehr geringfuegige, Aenderungen der Dateien "trace", "notrace" und "break" notwendig geworden, deren auffaelligste vielleicht die interne Umbenennung der Kommandofunktionen zur Aktivierung des Hand-Schneiders ("cut" und "nocut") in "mcut" bzw. "nomcut" ist.

Noch ein Hinweis: Der besonders interessierte Leser kann auch den gesamten Quellcode der Systeme LISPLOG.II und LISPLOG.2 (mit iterativem Interpreter) vom Autor erhalten. Auch ein Versand auf Datentraeger ist zum Selbstkostenpreis moeglich.
Das LISPLOG-Team freut sich ueber jede Anfrage!

Listing der Datei absynt.l:

```
(setq absynt.lfns
  '(first second third fourth fifth sixth seventh rest twolist
    consp s-1-ofarith s-2-ofarith s-3-ofarith is-user-defined
    is-primitive is-lisp-defined cut-p s-conclusion s-premises
    remove-cut is-p not-p arith-p is-command))

(defmacro first (x) (list 'car x))

(defmacro second (x) (list 'cadr x))

(defmacro third (x) (list 'caddr x))

(defmacro fourth (x) (list 'cadddr x))

(defmacro fifth (x) (list 'car (list 'cddddr x)))

(defmacro sixth (x) (list 'cadr (list 'cddddr x)))

(defmacro seventh (x) (list 'caddr (list 'cddddr x)))

(defmacro rest (x) (list 'cdr x))

(defmacro twolist (l1 l2) (list 'cons l1 (list 'cons l2 nil)))

(defmacro consp (expr) (list 'dtpr expr))

(defmacro s-1-ofarith (x) (list 'second x))

(defmacro s-2-ofarith (x) (list 'third x))

(defmacro s-3-ofarith (x) (list 'fourth x))

(defmacro is-user-defined (goal)
  (list 'get (list 'first goal) '(quote clauses)))

(defmacro is-primitive (goal)
  (list 'member (list 'first goal) 'primitives))

(defmacro is-lisp-defined (goal) (list 'getd (list 'first goal)))

(defmacro cut-p (l-clause)
  (list 'eq (list 'first (list 'first l-clause)) '(quote cut)))

(defun s-conclusion (l-clause)
  (if (cut-p l-clause) (second (first l-clause))
                       (first l-clause)))

(defmacro s-premises (l-clause) (list 'rest l-clause))

(defmacro remove-cut (l-clause)
  (list 'cons
```

```
        (list 's-conclusion l-clause)
        (list 's-premises l-clause)))

(defmacro is-p (goal)
  (list 'and
        (list 'consp goal)
        (list 'eq (list 'first goal) '(quote is))))

(defmacro not-p (goal)
  (list 'and
        (list 'consp goal)
        (list 'eq (list 'first goal) '(quote not))))

(defun arith-p (x)
  (and (listp x)
       (eq (length x) 4)
       (assoc (first x) arith-predicates)))

(defmacro is-command (x)
  (list 'member
        x
        (list 'append (list 'quote '(+ - l lisp)) 'lisp-coms)))
```

Listing der Datei andor.l:

```
(setq andor.lfns '(and-process is-external or-process))

(defun and-process (list-of-goals environment level)
  (let
    ((new-list-of-goals (exit-boxes list-of-goals environment)))
    (if
      (null new-list-of-goals)
      (cond (intern-flag
             (cond ((eq anzahl 1)
                    (setq all-environment
                      (cons environment all-environment))
                    t)
                   (t (setq all-environment
                        (cons environment all-environment))
                      (setq anzahl (sub1 anzahl))
                      (redo-boxes list-of-goals environment))))
            (t (if tracemode (terpr))
               (setq linecount 22)
               (patom "success")
               (terpr)
               (print-bindings (rest (reverse environment))
                               environment)
               (if (y-or-n-p "More? (y or n) ")
                   (redo-boxes list-of-goals environment)
                   t)))
      (let*
        ((goal
          (ultimate-inst (first new-list-of-goals) environment))
         (goals-left (get-rest-of-goals goal new-list-of-goals)))
        (cond
          ((atom goal)
           (box-call goal)
           (cond
             (goal
              (or
                (and-process goals-left environment (add1 level))
                (box-fail goal)
                (redo-boxes list-of-goals environment)))
             (t (box-fail goal)
                (redo-boxes list-of-goals environment))))
          ((eq '$ (first goal))
           (switch-to-modul-intern (second goal))
           (cond ((and-process goals-left environment level))
                 (t (switch-to-modul-intern (third goal)) nil)))
          ((is-external goal)
           (let* ((new-modul (get (first goal) modul-current))
                  (new-goal
                    (list (list '$ new-modul modul-current)
                          goal
                          (list '$ modul-current new-modul))))
                 (and-process (append new-goal goals-left)
```

```
                              environment
                              level)))
          ((not (atom (get (first goal) modul-current)))
           (box-call goal)
           (or (or-process
                 (inx-get (first goal) modul-current goal)
                 goals-left
                 goal
                 environment
                 level)
               (box-fail goal)
               (redo-boxes list-of-goals environment)))
          ((is-primitive goal)
           (box-call goal)
           (or
             (prolog-primitive goals-left goal environment level)
             (box-fail goal)
             (redo-boxes list-of-goals environment)))
          ((is-lisp-defined goal)
           (box-call goal)
           (or
             (lisp-predicates goal goals-left environment level)
             (box-fail goal)
             (redo-boxes list-of-goals environment))))))))

(defun is-external (goal)
  (let ((exporter (get (first goal) modul-current)))
       (and (atom exporter)
            (member (first goal) (get exporter 'export$)))))

(defun or-process (database-left goals-left goal environment
                   level)
  (if
    (null database-left)
    nil
    (let
      ((assertion
        (rename-variables (first database-left) (list level))))
      (let ((new-environment
             (unify goal (s-conclusion assertion) environment))
            (new-goals (s-premises assertion)))
           (cond ((null new-environment)
                  (or-process (rest database-left)
                              goals-left
                              goal
                              environment
                              level))
                 ((box-skip? goal new-goals))
                 ((and-process (append new-goals goals-left)
                               new-environment
                               (add1 level)))
                 ((or (cut-p assertion)
                      (box-cut-p goal database-left))
```

```
                  (box-cut goal assertion (rest database-left)))
                 (t (or-process (rest database-left)
                                goals-left
                                goal
                                environment
                                level)))))))
```

Listing der Datei notrace.l:

```
(setq notrace.lfns
  '(all-box-ports box-call box-cut box-cut-p box-exit box-fail
    box-redo box-skip? brk check-box-ports check-names
    check-ports exit-boxes get-port-list get-pred-of-goal
    get-rest-of-goals line-control lisplog-interrupt
    lisplog-reset load-tracer make-cut-permanent? manual-cutter
    mcut nobrk nomcut noskip nospy redo-boxes skip skipq spy
    unload-tracer))

(setq all-box-ports '(call exit fail redo eval value))

(defun box-call (goal) nil)

(defun box-cut (goal assertion database-left) nil)

(defun box-cut-p (goal database-left)
  (let* ((clause (first database-left))
         (predicate (get-pred-of-goal (s-conclusion clause))))
        (if (and (memq predicate cutmode)
                 (trace-unify-p goal database-left))
            (manual-cutter clause))))

(defun box-exit (goal) nil)

(defun box-fail (goal) nil)

(defun box-redo (goal) nil)

(defun box-skip? (goal new-goals) nil)

(def brk
  (nlambda (pred-list)
    (cond ((check-ports pred-list)
           (load-tracer)
           (if (null tracemode) (setq tracemode 'break-only))
           (if (null pred-list)
               (brk-1 (get-port-list predicates))
               (brk-1 (get-port-list pred-list)))))))

(defun check-box-ports (ports-list)
  (cond ((null ports-list) 'okay)
        ((memq (first ports-list) all-box-ports)
         (check-box-ports (rest ports-list)))
        (t (princ "Error: '")
           (princ (first ports-list))
           (princ "' is no box-model-port!")
           (terpr)
           nil)))

(defun check-names (names)
  (cond
```

```
    ((null names) 'okay)
    ((memq (first names) predicates) (check-names (rest names)))
    ((memq (first names) functions) (check-names (rest names)))
    ((memq (first names) '(is not var nonvar))
     (check-names (rest names)))
    (t (princ "Error: '")
       (princ (first names))
       (princ "' is no predicate or user-defined function!")
       (terpr)
       nil)))

(defun check-ports (pred-list)
  (cond ((null pred-list) 'okay)
        ((atom (first pred-list))
         (and (check-names (list (first pred-list)))
              (check-ports (rest pred-list))))
        (t (and (check-names (list (first (first pred-list))))
                (check-box-ports (rest (first pred-list)))
                (check-ports (rest pred-list))))))

(defun exit-boxes (list-of-goals environment) list-of-goals)

(defun get-port-list (pred-list)
  (cond ((null pred-list) nil)
        ((atom (first pred-list))
         (cons (cons (first pred-list) 'all)
               (get-port-list (rest pred-list))))
        (t (cons (first pred-list)
                 (get-port-list (rest pred-list))))))

(defun get-pred-of-goal (goal)
  (if (atom goal) goal (first goal)))

(defun get-rest-of-goals (goal list-of-goals)
  (rest list-of-goals))

(defun line-control nil nil)

(defun lisplog-interrupt (x)
  (setq breakflag t)
  (terpr)
  (evalframe nil))

(defun lisplog-reset (x)
  (terpr)
  (princ (ascii 27))
  (princ "#6")
  (princ (ascii 27))
  (princ "[7m")
  (princ "RESET to LISPLOG-Toplevel")
  (princ (ascii 27))
  (princ "[0m")
  (terpr)
```

```
  (lisplog))

(defun load-tracer nil
  (cond (tracer-loaded)
        (toplevel-flag (apply 'dskin (list tracer-file-name))
                       (setq tracer-loaded t))))

(defun make-cut-permanent? (clause)
  (princ "Do you want to make the Cut permanent? ")
  (let ((answer (first (lineread t))))
       (if (memq answer '(b B break Break BREAK))
           (cond (tracemode (load-breaker)
                            (setq answer (break-loop t)))
                 (t (princ "No Box-Model-Tracer activated...")
                    (terpr))))
       (setq linecount 22)
       (cond ((memq answer '(y Y j J yes Yes ja Ja))
              (rplaca clause (list 'cut (first clause)))
              (princ "Cut is made permanent.")
              (terpr))
             ((memq answer '(n N no No nein Nein)))
             (t (make-cut-permanent? clause)))))

(defun manual-cutter (clause)
  (princ "Do you want to cut clause ")
  (cond ((greaterp (flatc (external-form (cons 'ass clause))) 51)
         (terpr)
         (princ "...")))
  (print-external (cons 'ass clause))
  (princ "? ")
  (let ((answer (first (lineread t))))
       (if (member answer '(b B break Break BREAK))
           (cond (tracemode (load-breaker)
                            (setq answer (break-loop t)))
                 (t (princ "No Box-Model-Tracer activated...")
                    (terpr))))
       (setq linecount 22)
       (cond ((member answer '(y Y j J yes Yes ja Ja))
              (make-cut-permanent? clause)
              t)
             ((member answer '(n N no No nein Nein)) nil)
             (t (manual-cutter clause)))))

(def mcut
  (nlambda (pred-list)
    (cond ((not (check-names pred-list)))
          ((null pred-list) (setq cutmode predicates))
          (t (setq cutmode (union pred-list cutmode))))
    cutmode))

(def nobrk
  (nlambda (pred-list)
    (cond
```

```
        ((check-ports pred-list)
         (if (null pred-list) (setq breakmode nil)
                              (nobrk-1 (get-port-list pred-list)))
         (cond ((and (null breakmode) (eq tracemode 'break-only))
                (setq tracemode nil)
                (unload-tracer)
                breakmode)
               (t breakmode))))))

(def nomcut
  (nlambda (pred-list)
    (cond ((not (check-names pred-list)))
          ((null pred-list) (setq cutmode nil))
          (t (setq cutmode (remlist pred-list cutmode))))
    cutmode))

(def noskip
  (nlambda (pred-list)
    (cond
      ((not (check-names pred-list)))
      ((null pred-list) (setq skipmode nil) (setq skipqmode nil))
      (t (setq skipmode (remlist pred-list skipmode))
         (setq skipqmode (remlist pred-list skipqmode))))
    (append skipmode skipqmode)))

(def nospy
  (nlambda (pred-list)
    (cond ((not (check-names pred-list)))
          ((eq tracemode 'break-only))
          (t (if (null pred-list)
                 (setq tracemode nil)
                 (setq tracemode (remlist pred-list tracemode)))
             (cond ((and (null tracemode) breakmode)
                    (setq tracemode 'break-only))
                   ((null tracemode)
                    (untrace-lisp-functions functions)
                    (unload-tracer)))))
    tracemode))

(defun redo-boxes (list-of-goals environment) nil)

(def skip
  (nlambda (pred-list)
    (cond ((not (check-names pred-list)))
          ((null pred-list) (setq skipqmode nil)
                            (setq skipmode
                              (union
                                (union functions predicates)
                                (union '(is not) skipmode))))
          (t (setq skipmode (union pred-list skipmode))
             (setq skipqmode (remlist pred-list skipqmode))))
    skipmode))
```

```
(def skipq
  (nlambda (pred-list)
    (cond ((not (check-names pred-list)))
          ((null pred-list)
           (setq skipqmode
             (remlist
              skipmode
              (union (union functions predicates) '(is not)))))
          (t (setq skipqmode (union pred-list skipqmode))
             (setq skipmode (remlist pred-list skipmode))))
    skipqmode))

(def spy
  (nlambda (pred-list)
    (cond ((check-names pred-list)
           (load-tracer)
           (if (eq tracemode 'break-only) (setq tracemode nil))
           (if (null pred-list)
               (setq tracemode
                 (union
                  (union functions predicates)
                  (union '(is not var nonvar) tracemode)))
               (setq tracemode (union pred-list tracemode)))
           (trace-lisp-functions functions)
           (if (and (null tracemode) breakmode)
               (setq tracemode 'break-only)
              tracemode)))))

(defun unload-tracer nil
  (cond ((and tracer-loaded toplevel-flag)
         (untrace-lisp-functions functions)
         (apply 'dskin (list no-tracer-file-name))
         (setq tracer-loaded nil))
        (tracer-loaded
         (princ "Note: The tracer is no more activated ")
         (princ "for any predicate or function but may be")
         (terpr)
         (princ "      unloaded only when given the ")
         (princ "'nospy' or 'nobrk' command in the top-level!")
         (terpr))))
```

Listing der Datei trace.l:

```
(setq trace.lfns
  '(box-call box-cut box-eval box-exit box-fail box-ports-diff
    box-ports-union box-redo box-skip? box-value
    break-activated-p breakpoint brk-1 brk-2 check-skiplevel
    exit-boxes get-rest-of-goals get-text-for line-control
    linebreak linebreak-loop load-breaker nobrk-1 nobrk-2
    print-activated-p print-box-information push-backtrace-stack
    redo-boxes redo-boxes-1 redo-boxes-2 skip? step-control
    trace-activated-p trace-lisp-functions tracer-init
    untrace-lisp-functions))

(defun box-call (goal)
  (cond (tracemode
         (push-backtrace-stack printlevel
                               'call
                               (trace-inst goal 'call))
         (setq printlevel (add1 printlevel))
         (if (break-activated-p goal 'call) (setq breakflag t))
         (breakpoint)
         (if (or (eq (first goal) 'not)
                 (and (eq (first goal) 'is)
                      (listp (third goal))
                      (memq (first (third goal)) functions)))
             (skip? goal))))
  nil)

(defun box-cut (goal clause clause-list)
  (if tracemode
      (let
        ((clauses-cut (trace-unify goal database-left)))
        (if (greaterp clauses-cut 0)
            (push-backtrace-stack
              printlevel
              (list 'cut (length clause-list) clauses-cut clause)
              goal))
        nil)))

(defun box-eval (fct-name fct-arglist)
  (let
    ((goal (cons fct-name fct-arglist)))
    (cond (tracemode
           (push-backtrace-stack printlevel 'eval goal)
           (setq printlevel (add1 printlevel))
           (if (break-activated-p goal 'eval) (setq breakflag t))
           (breakpoint)
           (skip? goal)))
    nil))

(defun box-exit (goal)
  (cond (tracemode
         (check-skiplevel)
```

```
          (setq printlevel (sub1 printlevel))
          (push-backtrace-stack printlevel
                                'exit
                                (trace-inst goal 'exit))
          (if (break-activated-p goal 'exit) (setq breakflag t))
          (breakpoint)))
  nil)

(defun box-fail (goal)
  (cond (tracemode
          (check-skiplevel)
          (setq printlevel (sub1 printlevel))
          (push-backtrace-stack printlevel
                                'fail
                                (trace-inst goal 'fail))
          (if (break-activated-p goal 'fail) (setq breakflag t))
          (breakpoint)))
  nil)

(defun box-ports-diff (list1 list2)
  (cond ((null list1) nil)
        ((eq list1 'all) (box-ports-diff all-box-ports list2))
        ((eq list2 'all) (box-ports-diff list1 all-box-ports))
        ((memq (first list1) list2)
         (box-ports-diff (rest list1) list2))
        (t (cons (first list1)
                 (box-ports-diff (rest list1) list2)))))

(defun box-ports-union (list1 list2)
  (cond
    ((eq list1 'all) 'all)
    ((eq list2 'all) 'all)
    (t (let
          ((ports-union (union list1 list2)))
          (if (eq ports-union all-box-ports) 'all ports-union)))))

(defun box-redo (goal)
  (cond (tracemode
          (push-backtrace-stack printlevel
                                'redo
                                (trace-inst goal 'redo))
          (setq printlevel (add1 printlevel))
          (if (break-activated-p goal 'redo) (setq breakflag t))
          (breakpoint)))
  nil)

(defun box-skip?  (goal new-goals)
  (if tracemode (if new-goals (skip? goal))))

(defun box-value (fct-name fct-result)
  (cond
    (tracemode (check-skiplevel)
               (setq printlevel (sub1 printlevel))
```

```
              (push-backtrace-stack printlevel
                                    'value
                                    (list fct-name fct-result))
              (if (break-activated-p (list fct-name) 'value)
                  (setq breakflag t))
              (breakpoint)))
  nil)

(defun break-activated-p (goal flag)
  (if (lessp printlevel skiplevel)
      (let ((ports
              (rest (assoc (get-pred-of-goal goal) breakmode))))
           (or (eq ports 'all) (memq flag ports)))))

(defun breakpoint nil
  (cond (breakflag (load-breaker) (breaking))))

(defun brk-1 (port-list)
  (cond
    ((null port-list) breakmode)
    (t (let ((pred (first (first port-list)))
             (box-ports (rest (first port-list))))
            (if (assoc pred breakmode)
                (setq breakmode (brk-2 pred box-ports breakmode))
                (setq breakmode
                  (cons (first port-list) breakmode))))
        (brk-1 (rest port-list)))))

(defun brk-2 (pred box-ports rest-breakmode)
  (if
    (eq (first (first rest-breakmode)) pred)
    (cons (cons pred
                (box-ports-union box-ports
                                 (rest (first rest-breakmode))))
          (rest rest-breakmode))
    (cons (first rest-breakmode)
          (brk-2 pred box-ports (rest rest-breakmode)))))

(defun check-skiplevel nil
  (if (equal printlevel skiplevel) (setq skiplevel 999999)))

(defun exit-boxes (goals-left environment)
  (cond ((null goals-left) nil)
        ((atom (first goals-left)) goals-left)
        ((eq (rest (first goals-left)) 'active)
         (box-exit (trace-inst (first (first goals-left)) 'exit))
         (exit-boxes (rest goals-left) environment))
        (t goals-left)))

(defun get-rest-of-goals (goal list-of-goals)
  (cons (cons goal 'active) (rest list-of-goals)))
```

```
(defun get-text-for (flag)
  (let ((text
         (assoc flag
                '((call "CALL ")
                  (exit "EXIT ")
                  (fail "FAIL ")
                  (redo "REDO ")
                  (eval "EVAL ")
                  (value "VALU ")))))
       (if text (second text) "*** Unknown flag ***")))

(defun line-control nil
  (cond (tracemode (setq linecount (sub1 linecount))
                   (if (zerop linecount) (linebreak)))))

(defun linebreak nil
  (cond (breakflag (breaking))
        (t (princ "More information desired? ")
           (princ "Enter +p, -p or number of steps: ")
           (linebreak-loop)
           (setq breakflag nil))))

(defun linebreak-loop nil
  (let*
    ((inline (lineread t))
     (answer (first inline))
     (com (second (assoc answer lisp-coms))))
    (cond
      ((null inline) (setq stepcount 1) (setq linecount -1))
      (com (apply com (rest inline)) (linebreak))
      ((memq answer '(b B break BREAK)) (setq breakflag t)
                                        (breakpoint))
      ((numberp answer)
       (cond ((plusp answer) (setq stepcount answer)
                             (setq linecount -1))
             ((minusp answer)
              (load-breaker)
              (backtrace 1
                         (minus answer)
                         (get-length-activated backtrace-stack)
                         backtrace-stack
                         t
                         t)
              (princ "LISPLOG continues...")
              (terpr))
             (t (terpr)
                (throw 'user-break:reset-to-LISPLOG-Top-Level))))
      ((or (eq answer 'p)
           (eq answer 'P)
           (eq answer '+p)
           (eq answer '+P))
       (setq stepcount -1)
       (setq linecount 22))
```

```
      ((or (eq answer '-p) (eq answer '-P))
       (load-breaker)
       (backtrace 1
                  22
                  (get-length-activated backtrace-stack)
                  backtrace-stack
                  t
                  t)
       (princ "LISPLOG continues...")
       (terpr))
      (t (princ "*** Error: Unknown input! ")
         (princ "Enter +p, -p or number of steps: ")
         (linebreak-loop)))))

(defun load-breaker nil
  (cond (breaker-loaded)
        (t (apply 'dskin (list breaker-file-name))
           (setq breaker-loaded t))))

(defun nobrk-1 (port-list)
  (cond
    ((null port-list) breakmode)
    (t (let ((pred (first (first port-list)))
             (box-ports (rest (first port-list))))
            (setq breakmode (nobrk-2 pred box-ports breakmode)))
       (nobrk-1 (rest port-list)))))

(defun nobrk-2 (pred box-ports rest-breakmode)
  (cond ((null rest-breakmode) nil)
        ((eq (first (first rest-breakmode)) pred)
         (let ((new-ports
                (box-ports-diff (rest (first rest-breakmode))
                                box-ports)))
              (if new-ports (cons (cons pred new-ports)
                                  (rest rest-breakmode))
                            (rest rest-breakmode))))
        (t (nobrk-2 pred box-ports (rest rest-breakmode)))))

(defun print-activated-p (text goal)
  (and
    (lessp printlevel skiplevel)
    (cond
      ((or (and (listp text)
                (memq (first text)
                      '(unify-backward unify-forward mismatch)))
           (eq text 'mismatch))
       (and (trace-activated-p goal)
            (trace-activated-p 'lisplog-unify)))
      ((or
         (eq text 'modul)
         (trace-activated-p goal)
         (and (eq text 'undef) (trace-activated-p 'undef))
         (and (eq text 'value) (trace-activated-p (first goal)))
```

```
          (break-activated-p goal text))))))

(defun print-box-information (printlevel text goal)
  (princ "|")
  (tab printlevel)
  (cond ((eq text 'value) (princ (get-text-for 'value))
                          (princ (second goal)))
        ((atom text) (princ (get-text-for text))
                     (print-external goal))
        ((eq (first text) 'cut)
         (princ "Cutting ")
         (princ (second text))
         (princ " [")
         (princ (third text))
         (princ "]")
         (princ " clauses after clause ")
         (cond ((greaterp (flatc (external-form
                                  (cons 'ass (fourth text))))
                          43)
                (terpr)
                (princ "|...")))
         (print-external (cons 'ass (fourth text)))))
  (terpr)
  (if (atom text) (step-control) (line-control)))

(defun push-backtrace-stack (printlevel text goal)
  (setq backtrace-stack
    (cons (list printlevel text goal) backtrace-stack))
  (if (print-activated-p text goal)
      (print-box-information printlevel text goal)))

(defun redo-boxes (goals-left environment)
  (redo-boxes-1 goals-left nil environment))

(defun redo-boxes-1 (goals-left redo-list environment)
  (if (eq (cdr (first goals-left)) 'active)
      (redo-boxes-1
        (rest goals-left)
        (cons (trace-inst (first goals-left) 'redo) redo-list)
        environment)
      (redo-boxes-2 redo-list
                    (cons (first redo-list) goals-left))))

(defun redo-boxes-2 (redo-list goals-left)
  (cond
    ((null redo-list) nil)
    (t (box-redo (first (first redo-list)))
       (if (rest redo-list) (skip? (first (first redo-list))))
       (redo-boxes-2 (rest redo-list)
                     (cons (second redo-list) goals-left)))))

(defun skip? (goal)
  (cond
```

```
    ((and
       (lessp printlevel skiplevel)
       (memq (first goal) tracemode)
       (memq (second (first backtrace-stack)) '(call redo eval)))
     (cond
       ((memq (first goal) skipmode) (setq skiplevel printlevel)
                                     (princ "|")
                                     (tab (sub1 printlevel))
                                     (princ "Skipping...")
                                     (terpr))
       ((memq (first goal) skipqmode)
        (princ "Do you want to skip? ")
        (let ((answer (first (lineread t))))
             (cond ((memq answer '(b B break Break BREAK))
                    (load-breaker)
                    (setq answer (break-loop t))))
             (cond ((memq answer '(j J ja Ja JA y Y yes Yes YES))
                    (setq skiplevel printlevel)
                    (princ "|")
                    (tab (sub1 printlevel))
                    (princ "Skipping...")
                    (terpr))
                   ((memq answer '(n N no No NO nein Nein NEIN)))
                   ((equal skiplevel printlevel))
                   (t (skip? goal))))
        (setq linecount 22)))
     nil)))

(defun step-control nil
  (cond (tracemode
         (setq stepcount (sub1 stepcount))
         (if (zerop stepcount) (linebreak) (line-control)))))

(defun trace-activated-p (goal)
  (if (listp tracemode)
      (member (get-pred-of-goal goal) tracemode)
      nil))

(defun trace-lisp-functions (functions)
  (cond ((null functions))
        (t (let ((fct-name (first functions)))
                (apply 'trace
                       (list (cons fct-name
                                   '(traceenter box-eval
                                     traceexit box-value)))))
           (trace-lisp-functions (rest functions)))))

(defun tracer-init nil
  (setq backtrace-stack nil)
  (setq stepcount -1)
  (setq linecount 22)
  (setq breakflag nil)
  (setq skiplevel 999999)
```

```
  (setq printlevel 1))

(defun untrace-lisp-functions (functions)
  (if functions (apply 'untrace functions)))
```

Listing der Datei break.l:

```
(setq break.lfns
  '(backtrace backtrace-loop backwardp break-loop breaking
    get-length-activated print-backtrace print-backtrace-stack
    print-goal print-list-of-goals print-list-of-goals-1
    print-path print-path-1 print-premises))

(defun backtrace (begin end max stack printp continuep)
  (cond
    ((greaterp begin 0)
     (if printp (print-backtrace (sub1 begin)
                                 (add1 (diff end begin))
                                 stack
                                 nil))
     (cond
       ((or (greaterp begin 1) (backwardp answer))
        (princ "More information desired? ")
        (princ "Enter +p, -p or number of steps: ")
        (backtrace-loop begin end max stack printp continuep))))
    (continuep (if printp (print-backtrace 0 end stack nil))
               (setq stepcount (diff 1 begin))
               (setq linecount -1))
    (t (if printp (print-backtrace 0 end stack nil)))))

(defun backtrace-loop (begin end max stack printp continuep)
  (let
    ((answer (first (lineread t))))
    (cond
      ((or (eq answer '-p) (eq answer '-P)) (setq answer -22))
      ((or (eq answer '+p) (eq answer '+P)) (setq answer 22))
      ((or (eq answer 'p) (eq answer 'P)) (setq answer 22)))
    (cond
      ((numberp answer)
       (cond ((plusp answer) (backtrace (diff begin answer)
                                        (sub1 begin)
                                        max
                                        stack
                                        t
                                        continuep))
             ((minusp answer)
              (if (eq end max)
                  (backtrace begin end max stack nil continuep)
                  (backtrace (add1 end)
                             (if (greaterp (diff end answer) max)
                                 max
                                 (diff end answer))
                             max
                             stack
                             t
                             continuep)))
             (t (terpr)
                (throw 'user-break:reset-to-LISPLOG-Top-Level))))
```

```
         (t
          (princ "*** Error: Unknown input! ")
          (princ "Enter +p, -p or number of steps: ")
          (backtrace-loop begin end max stack printp continuep)))))

(defun backwardp (answer)
  (or (and (numberp answer) (minusp answer))
      (eq answer '-p)
      (eq answer '-P)))

(defun break-loop (reply-outstanding)
  (princ "Please enter Break-command: ")
  (let*
    ((inline (lineread t))
     (answer (first inline))
     (com (second (assoc answer lisp-coms))))
    (cond
      (com (apply com (rest inline))
           (break-loop reply-outstanding))
      ((null inline) (setq stepcount 1) (setq linecount -1))
      ((numberp answer)
       (cond ((plusp answer) (setq stepcount answer)
                             (setq linecount -1)
                             (princ "LISPLOG continues...")
                             (terpr))
             ((minusp answer)
              (backtrace 1
                         (minus answer)
                         (get-length-activated backtrace-stack)
                         backtrace-stack
                         t
                         nil)
              (break-loop reply-outstanding))
             (t (terpr)
                (throw 'user-break:reset-to-LISPLOG-Top-Level))))
      ((or (eq answer 'p)
           (eq answer 'P)
           (eq answer '+p)
           (eq answer '+P))
       (setq stepcount -1)
       (setq linecount 22)
       (princ "LISPLOG continues...")
       (terpr))
      ((or (eq answer '-p) (eq answer '-P))
       (backtrace 1
                  22
                  (get-length-activated backtrace-stack)
                  backtrace-stack
                  t
                  nil)
       (break-loop reply-outstanding))
      ((memq answer '(r R re RE res RES reset RESET))
       (terpr)
```

```
          (throw 'user-break:reset-to-LISPLOG-Top-Level))
         ((memq answer
                '(c C co CO con CON cont CONT continue CONTINUE))
          (princ "LISPLOG continues...")
          (setq linecount 22)
          (terpr))
         ((memq answer '(s S skipping SKIPPING))
          (cond
            ((memq (second (first backtrace-stack))
                   '(call eval redo))
             (setq skiplevel printlevel)
             (princ "Now skipping information about execution of ")
             (princ (third (first backtrace-stack)))
             (terpr))
            (t (princ "Skipping ")
               (princ (third (first backtrace-stack)))
               (princ " makes no sense: box already left.")
               (terpr)))
          (break-loop reply-outstanding))
         ((memq answer '(y Y j J n N yes YES ja JA no NO nein NEIN))
          (cond (reply-outstanding answer)
                (t (princ "There is no reply outstanding...!")
                   (terpr)
                   (break-loop reply-outstanding))))
         ((memq answer '(v V eval EVAL))
          (print (first (errset (eval (second inline)))))
          (terpr)
          (break-loop reply-outstanding))
         ((memq answer '(i I in IN inf INF info INFO))
          (princ "spy active for: ")
          (princ tracemode)
          (terpr)
          (princ "brk active for: ")
          (princ breakmode)
          (terpr)
          (princ "cut active for: ")
          (princ cutmode)
          (terpr)
          (princ "skip active for: ")
          (princ skipmode)
          (terpr)
          (princ "skipq active for: ")
          (princ skipqmode)
          (terpr)
          (break-loop reply-outstanding))
         ((eq (first version) 'rekursiv)
          (cond ((memq answer
                       '(e E en EN env ENV environment ENVIRONMENT))
                 (pp environment)
                 (terpr)
                 (break-loop reply-outstanding))
                ((memq answer '(p P pa PA path PATH))
                 (print-path goals-left)
```

```
               (terpr)
               (break-loop reply-outstanding))
              ((memq answer
                     '(l L li LI lis LIS list LIST list-of-goals
                       LIST-OF-GOALS))
               (print-list-of-goals goals-left)
               (terpr)
               (break-loop reply-outstanding))
              ((memq answer '(g G go GO goa GOA goal GOAL))
               (print-goal goal)
               (terpr)
               (break-loop reply-outstanding))
              (t (princ "*** Unknown break-command ***")
                 (terpr)
                 (break-loop reply-outstanding))))
       (t (princ "*** Unknown break-command ***")
          (terpr)
          (break-loop reply-outstanding)))))

(defun breaking nil
  (princ (ascii 27))
  (princ "#6")
  (princ (ascii 27))
  (princ "[7m")
  (princ "LISPLOG - Break - Level")
  (princ (ascii 27))
  (princ "[0m")
  (terpr)
  (break-loop nil)
  (setq breakflag nil))

(defun get-length-activated (stack)
  (cond ((null stack) 0)
        ((print-activated-p (second (first stack))
                            (third (first stack)))
         (add1 (get-length-activated (rest stack))))
        (t (get-length-activated (rest stack)))))

(defun print-backtrace (depth length stack printstack)
  (cond
    ((and stack
          (print-activated-p (second (first stack))
                             (third (first stack))))
     (cond ((plusp depth) (print-backtrace (sub1 depth)
                                           length
                                           (rest stack)
                                           printstack))
           ((plusp length)
            (print-backtrace depth
                             (sub1 length)
                             (rest stack)
                             (cons (first stack) printstack)))
           (t (print-backtrace-stack printstack))))
```

```
    (stack
     (print-backtrace depth length (rest stack) printstack))
    (t (print-backtrace-stack printstack))))

(defun print-backtrace-stack (printstack)
  (cond ((null printstack))
        (t (print-box-information (first (first printstack))
                                  (second (first printstack))
                                  (third (first printstack)))
           (print-backtrace-stack (rest printstack)))))

(defun print-goal (goal)
  (princ "The actual goal is: ")
  (print-external goal))

(defun print-list-of-goals (goal-list)
  (princ "level")
  (tab 6)
  (princ "active goal")
  (tab 20)
  (princ "more premises ")
  (terpr)
  (princ "----------------------------------------")
  (princ "----------------------------------------")
  (terpr)
  (print-list-of-goals-1 (reverse goal-list) nil 0))

(defun print-list-of-goals-1 (goal-list premises level)
  (cond
    ((null goal-list))
    ((eq (rest (first goal-list)) 'active)
     (princ level)
     (tab 6)
     (print-external (first (first goal-list)))
     (cond (premises (tab 19) (print-premises premises 60))
           (t (tab 20) (princ "none")))
     (terpr)
     (print-list-of-goals-1 (rest goal-list) nil (add1 level)))
    (t (print-list-of-goals-1 (rest goal-list)
                              (cons (first goal-list) premises)
                              level))))

(defun print-path (goal-list)
  (princ "The actual path is: ")
  (print-path-1 goal-list 60))

(defun print-path-1 (goal-list space)
  (cond
    ((null goal-list))
    ((eq (rest (first goal-list)) 'active)
     (let
       ((next-goal (first (first goal-list))))
       (cond ((or (equal space 80)
```

```
                  (greaterp
                    space
                    (plus 4 (flatc (external-form next-goal)))))
              (princ " -- ")
              (print-external next-goal)
              (print-path-1
                (rest goal-list)
                (- space 4 (flatc (external-form next-goal)))))
             (t (terpr) (print-path-1 goal-list 80)))))
    (t (print-path-1 (rest goal-list) space))))

(defun print-premises (premises space)
  (cond ((null premises))
        ((or (equal space 60)
             (greaterp
               space
               (+ 3 (flatc (external-form (first premises))))))
         (princ " - ")
         (print-external (first premises))
         (print-premises
           (rest premises)
           (- space 3 (flatc (external-form (first premises))))))
        (t (tab 19) (print-premises premises 60))))
```

Listing der Datei untrace.l:

```
(setq untrace.lfns
  '(backtrace backtrace-loop backwardp box-call box-cut box-eval
    box-exit box-fail box-ports-diff box-ports-union box-redo
    box-skip? box-value break-activated-p break-loop breaking
    breakpoint brk-1 brk-2 check-skiplevel exit-boxes
    get-length-activated get-rest-of-goals get-text-for
    line-control linebreak linebreak-loop load-breaker nobrk-1
    nobrk-2 print-activated-p print-backtrace
    print-backtrace-stack print-box-information print-goal
    print-list-of-goals print-list-of-goals-1 print-path
    print-path-1 print-premises push-backtrace-stack redo-boxes
    redo-boxes-1 redo-boxes-2 skip? step-control
    trace-activated-p trace-lisp-functions tracer-init
    untrace-lisp-functions))

(defun backtrace (begin end max stack printp continuep) nil)

(defun backtrace-loop (begin end max stack printp continuep) nil)

(defun backwardp (answer) nil)

(defun box-call (goal) nil)

(defun box-cut (goal assertion database-left) nil)

(defun box-eval (fct-name fct-arglist) nil)

(defun box-exit (goal) nil)

(defun box-fail (goal) nil)

(defun box-ports-diff (list1 list2) nil)

(defun box-ports-union (list1 list2) nil)

(defun box-redo (goal) nil)

(defun box-skip?  (goal new-goals) nil)

(defun box-value (fct-name fct-result) nil)

(defun break-activated-p (goal flag) nil)

(defun break-loop (reply-outstanding) nil)

(defun breaking nil nil)

(defun breakpoint nil nil)

(defun brk-1 (port-list) nil)
```

```
(defun brk-2 (pred box-ports rest-breakmode) nil)

(defun check-skiplevel nil nil)

(defun exit-boxes (list-of-goals environment) list-of-goals)

(defun get-length-activated (stack) nil)

(defun get-rest-of-goals (goal list-of-goals)
  (rest list-of-goals))

(defun get-text-for (flag) nil)

(defun line-control nil nil)

(defun linebreak nil nil)

(defun linebreak-loop nil nil)

(defun load-breaker nil nil)

(defun nobrk-1 (port-list) nil)

(defun nobrk-2 (pred box-ports rest-breakmode) nil)

(defun print-activated-p (text goal) nil)

(defun print-backtrace (depth length stack printstack) nil)

(defun print-backtrace-stack (printstack) nil)

(defun print-box-information (printlevel text goal) nil)

(defun print-goal (goal) nil)

(defun print-list-of-goals (goal-list) nil)

(defun print-list-of-goals-1 (goal-list premises level) nil)

(defun print-path (goal-list) nil)

(defun print-path-1 (goal-list space) nil)

(defun print-premises (premises space) nil)

(defun push-backtrace-stack (printlevel text goal) nil)

(defun redo-boxes (list-of-goals environment) nil)

(defun redo-boxes-1 (goals-left redo-list environment) nil)

(defun redo-boxes-2 (redo-list goals-left) nil)
```

```
(defun skip? (goal) nil)

(defun step-control nil nil)

(defun trace-activated-p (goal) nil)

(defun trace-lisp-functions (functions) nil)

(defun tracer-init nil nil)

(defun untrace-lisp-functions (functions) nil)
```

Listing der Datei unify.l:

```
(setq unify.lfns
  '(trace-inst trace-unify trace-unify-p unify ultimate-assoc
    ultimate-inst ultimate-partinst anonymous-p))

(defun trace-inst (goal mode)
  (if (eq mode 'exit) (ultimate-inst goal environment) goal))

(defun trace-unify (goal database-left)
  (cond
    ((null database-left) 0)
    ((unify goal
            (s-conclusion (rename-variables (first database-left)
                                            (list level)))
            environment)
     (add1 (trace-unify goal (rest database-left))))
    (t (trace-unify goal (rest database-left)))))

(defun trace-unify-p (goal database-left)
  (cond
    ((null database-left) nil)
    ((unify goal
            (s-conclusion (rename-variables (first database-left)
                                            (list level)))
            environment))
    (t (trace-unify-p goal (rest database-left)))))

(defun unify (x y environment)
  (let
    ((x (ultimate-assoc x environment))
     (y (ultimate-assoc y environment)))
    (cond
      ((equal x y) environment)
      ((or (anonymous-p x) (anonymous-p y)) environment)
      ((variable-p x) (cons (list x y) environment))
      ((variable-p y) (cons (list y x) environment))
      ((or (atom x) (atom y)) nil)
      (t (let ((new-environment
                (unify (car x) (car y) environment)))
           (and new-environment
                (unify (cdr x) (cdr y) new-environment)))))))

(defun ultimate-assoc (x environment)
  (if (variable-p x)
      (let ((binding (assoc x environment)))
           (if (null binding)
               x
               (ultimate-assoc (second binding) environment)))
      x))

(defun ultimate-inst (x environment)
  (cond ((variable-p x)
```

```
         (let ((binding (assoc x environment)))
              (if (null binding)
                  x
                  (ultimate-inst (second binding) environment))))
        ((atom x) x)
        (t (cons (ultimate-inst (first x) environment)
                 (ultimate-inst (rest x) environment)))))

(defun ultimate-partinst (x environment)
  (let ((x (ultimate-assoc x environment)))
       (if (atom x) x
                  (cons (ultimate-assoc (first x) environment)
                        (rest x)))))

(defun anonymous-p (x) (eq x 'ID))
```

Listing der Datei interface.l:

```
(setq interface.lfns
  '(commands do-exit-lisplog do-external-help do-help help-fname
    lineread lisp-coms lisplog lisplog-loop print-help
    print-help-list translate-command y-or-n-p))

(defun commands (inline)
  ((lambda
    (x)
    (if x (car x) (twolist 'error--lisp:internal '???)))
    (errset
      (let
        ((com (translate-command (first inline) lisp-coms))
         (intern-flag nil))
        (if tracemode (tracer-init))
        (if (not (null com))
            (let ((toplevel-flag t)) (apply com (rest inline)))
            (let ((consult-flag t))
                 (if (not (atom tracemode)) (terpr))
                 (cond (tracemode
                        (signal 2 'lisplog-interrupt)
                        (let ((result
                               (catch
                                 (and-process
                                   (rename-variables inline '(0))
                                   '((bottom-of-environment))
                                   1))))
                             (signal 2 'lisplog-reset)
                             result))
                       (t (catch (and-process
                                   (rename-variables inline '(0))
                                   '((bottom-of-environment))
                                   1))))))))))

(defun do-exit-lisplog nil
  (newintr)
  (throw (concat "You're now back in the LISP-Toplevel! "
                 "Return to LISPLOG with: (lisplog)")
         *leave-prolog*))

(defun do-external-help (argument)
  (cond
    ((null (translate-command (first argument) lisp-coms))
     (apply (function help) argument))
    ((not (null (help-fname (first argument) lisp-coms)))
     (*process
       (implode
         (append
           (append
             (explode '/usr/users/lisplog/tools/LISPLOG-help)
             '(| |))
           (explode (help-fname (first argument) lisp-coms))))))
```

```
    (t (patom "Keine weitere Hilfe verfuegbar") (terpri))))

(def do-help
  (nlambda (argument)
    (if (null argument) (print-help)
                        (do-external-help argument))))

(defun help-fname (com list-of-coms)
  (cond ((null list-of-coms) nil)
        ((eq com (first (first list-of-coms)))
         (if (null (rest (rest (first list-of-coms))))
             nil
             (rest (rest (rest (first list-of-coms))))))
        (t (help-fname com (rest list-of-coms)))))

(defun lineread (single-carriage-return-allowed)
  (if
    single-carriage-return-allowed
    (let ((nextchar (tyi)))
         (cond ((eq nextchar 10) nil)
               (t (untyi nextchar) (cons (read) (lineread t)))))
    (cons (read) (lineread t))))

(setq lisp-coms
  '((ass ass "Einfuegen einer Klausel in die Datenbasis" .
     ass.help)
    (+ ass "Kurz fuer ass" . ass.help)
    (rex rex "Loeschen einer Klausel" . rex.help)
    (- rex "Kurz fuer rex" . rex.help)
    (create create "Erzeugen eines Moduls" . modul.help)
    (abolish abolish "Loeschen eines Praedikates" . abolish.help)
    (destroy destroy "Loeschen eines Moduls" . abolish.help)
    (destroy-all destroy-all "Loeschen der ganzen Datenbasis" .
     abolish.help)
    (do-on-modul do-on-modul "" . modul.help)
    (exit exit "Verlasse LISP (und LISPLOG!)")
    (import-check import-check
     "Pruefen der Import/Export - Listen" . modul.help)
    (save save "Speichern eines Moduls" . save.help)
    (save-all save-all "Speichern aller Module" . save.help)
    (sort-modul sort-modul "Sortieren eines Moduls" . modul.help)
    (switch-to-modul switch-to-modul
     "Umschalten auf ein anderes Modul" . modul.help)
    (> switch-to-modul "Kurzform" . modul.help)
    (listing listing
     "Auflisten der angegebenen Praedikate, nil = alles" .
     listing.help)
    (l listing "Kurz fuer listing" . listing.help)
    (lisp do-exit-lisplog "Verlasse den LISPLOG-Interpreter")
    (edit edit "Editieren des angegebenen Praedikates mit vi")
    (tell tell "Save-all" . save.help)
    (consult consult "Laden der angegebenen Datei" . save.help)
    (spy spy "Tracen der angegebenen Praedikate" . spy.help)
```

```
    (nospy nospy
           "Tracer fuer angegebene Praedikate ausschalten" .
           spy.help)
    (brk brk "Break fuer angegebene Praedikate" . brk.help)
    (nobrk nobrk "Break ausschalten" . brk.help)
    (cut mcut "Hand-Cut fuer angegebene Praedikate" . cut.help)
    (nocut nomcut "Handschneider ausschalten" . cut.help)
    (skip skip "Trace-Information unterdruecken" . skip.help)
    (skipq skipq
           "Trace-Information nach Rueckfrage unterdruecken" .
           skip.help)
    (noskip noskip
            "Keine Trace-Information unterdruecken" .
            skip.help)
    (i print-modul-info "Zeige aktuelles Modul" . modul.help)
    (help do-help "Drucke Hilfsinformationen")))

(defun lisplog nil
  (princ (catch (lisplog-loop) *leave-prolog*))
  (terpr))

(defun lisplog-loop nil
  (signal 2 'lisplog-reset)
  (let ((toplevel-flag nil))
       (do ((leave-prolog nil))
           (leave-prolog)
           (print '*)
           (pp-external-form (commands (lineread nil)))
           (terpr))))

(defun print-help nil
  (patom "Folgende Kommandos stehen zur Verfuegung:")
  (terpri)
  (print-help-list lisp-coms))

(defun print-help-list (list-of-coms)
  (cond
    ((null list-of-coms) t)
    (t (print (first (first list-of-coms)))
       (tabl
        (- 15 (length (explode (first (first list-of-coms))))))
       (patom " : ")
       (patom (first (rest (rest (first list-of-coms)))))
       (terpri)
       (print-help-list (rest list-of-coms)))))

(defun translate-command (com list-of-coms)
  (cond ((null list-of-coms) nil)
        ((eq com (first (first list-of-coms)))
         (first (rest (first list-of-coms))))
        (t (translate-command com (rest list-of-coms)))))
```

```
(defun y-or-n-p (message)
  (patom message)
  (let ((response (read)))
       (if tracemode (terpr))
       (setq linecount 22)
       (cond ((eq response 'y) t)
             ((eq response 'n) nil)
             (t (y-or-n-p message)))))
```

Listing der Datei nsolutions.l:

```
(setq nsolutions.lfns
  '(n-solutions cons-bindings cons-bindings-all))

(defun n-solutions (goal anzahl)
  (let ((all-environment nil) (anzahl anzahl) (intern-flag t))
       (cond ((null goal) t)
             ((equal anzahl 0) nil)
             ((listp goal)
              (and-process (list (rename-variables goal '(0)))
                           '((bottom-of-environment))
                           1)
              (if all-environment
                  (if (equal (first all-environment)
                             '((bottom-of-environment)))
                      all-environment
                      (cons-bindings-all all-environment))
                  nil)))))

(defun cons-bindings (environment-left environment)
  (if
    (rest environment-left)
    (cons (let ((variable (first (first environment-left))))
               (if (zerop (third variable))
                   (cons (twolist '? (second variable))
                         (list (ultimate-inst variable
                                              environment)))))
          (cons-bindings (rest environment-left) environment))))

(defun cons-bindings-all (environment-liste)
  (if (null environment-liste)
      nil
      (cons (remove nil
                    (cons-bindings (first environment-liste)
                                   (first environment-liste)))
            (cons-bindings-all (rest environment-liste)))))
```

Listing der Datei primitives.l:

```
(setq primitives.lfns
  '(prolog-primitive arith-predicates execute-not
    execute-not-intern execute-is primitives arith-predicates))

(defun prolog-primitive (goals-left goal environment level)
  (cond
    ((arith-p goal)
     (and
       (non-var-p (s-1-ofarith goal) environment)
       (non-var-p (s-2-ofarith goal) environment)
       (var-p (s-3-ofarith goal) environment)
       (and-process
         goals-left
         (cons
           (list
             (s-3-ofarith goal)
             (apply (cadr (assoc (first goal) arith-predicates))
                    (list (ultimate-assoc (s-1-ofarith goal)
                                          environment)
                          (ultimate-assoc (s-2-ofarith goal)
                                          environment))))
          environment)
         level)))
    ((is-p goal)
     (let ((new-environment
             (execute-is (second goal) (third goal) environment)))
          (and new-environment
               (and-process goals-left new-environment level))))
    ((not-p goal)
     (execute-not-intern goals-left goal environment level))
    ((eq (first goal) 'var)
     (and (var-p (first (rest goal)) environment)
          (and-process goals-left environment level)))
    ((eq (first goal) 'nonvar)
     (and (non-var-p (first (rest goal)) environment)
          (and-process goals-left environment level)))))

(setq arith-predicates
  '((addit plus) (subit diff) (multit times) (divit quotient)))

(defun execute-not (goals-left goal environment level)
  (let ((all-environment nil) (anzahl 1) (intern-flag t))
       (and-process (list (cadr goal)) environment (add1 level))
       (not all-environment)))

(defun execute-not-intern (goals-left goal environment level)
  (if (execute-not goals-left goal environment level)
      (and-process goals-left environment level)
     nil))
```

```
(defun execute-is (variable lispexpr environment)
  (let
    ((quoted-lispexpr
       (quote-lambda-arg (ultimate-inst lispexpr environment))))
    (if (contains-freevars quoted-lispexpr)
        nil
        (let ((res (errset (eval quoted-lispexpr))))
             (if (null res)
                 (throw
                   (twolist 'error--execute-is:eval-not-possible
                            quoted-lispexpr))
                 (unify variable (first res) environment))))))

(setq primitives '(addit subit multit divit is not var nonvar))

(setq arith-predicates
  '((addit plus) (subit diff) (multit times) (divit quotient)))
```

Listing der Datei lispeval.l:

```
(setq lispeval.lfns
  '(lisp-predicates quoted-p lambda-appl-p nlambda-appl-p
    macro-appl-p application-p discipline contains-freevars
    quote-lambda-arg quote-nlambda-arg quote-application))

(defun lisp-predicates (goal rest-of-goals environment level)
  (let
    ((quoted-goal
      (quote-lambda-arg (ultimate-inst goal environment))))
    (if
      (contains-freevars quoted-goal)
      nil
      (let ((res (errset (eval quoted-goal))))
           (cond
             ((null res)
              (throw (twolist
                      'error--lisp-predicates:eval-not-possible
                      quoted-goal)))
             ((first res)
              (and-process rest-of-goals environment level)))))))

(defmacro quoted-p (expr)
  (list 'and
        (list 'consp expr)
        (list 'eq (list 'first expr) '(quote quote))))

(defmacro lambda-appl-p (expr)
  (list 'cond
        (list (list 'atom expr) nil)
        (list t
              (list 'or
                    (list 'eq
                          (list 'discipline (list 'first expr))
                          '(quote lambda))
                    (list 'eq
                          (list 'discipline (list 'first expr))
                          '(quote lexpr))))))

(defmacro nlambda-appl-p (expr)
  (list 'cond
        (list (list 'atom expr) nil)
        (list t
              (list 'eq
                    (list 'discipline (list 'first expr))
                    '(quote nlambda)))))

(defmacro macro-appl-p (expr)
  (list 'cond
        (list (list 'atom expr) nil)
        (list t
              (list 'eq
```

```
                    (list 'discipline (list 'first expr))
                    '(quote macro)))))

(defun application-p (expr)
  (or (lambda-appl-p expr) (nlambda-appl-p expr)))

(defun discipline (funct)
  (cond ((numberp funct) nil)
        ((atom funct) (let ((def (getd funct) (nil nil . nil)))
                        (cond ((null def) nil)
                              ((atom def) (getdisc def))
                              (t (first def)))))
        (t (first funct))))

(defun contains-freevars (expr)
  (cond
    ((or (atom expr) (quoted-p expr) (nlambda-appl-p expr)) nil)
    ((variable-p expr) t)
    ((lambda-appl-p expr)
     (some (function contains-freevars) (rest expr)))))

(defun quote-lambda-arg (expr)
  (let ((expr (expand-macro expr)))
       (cond ((or (null expr) (numberp expr)) expr)
             ((application-p expr) (quote-application expr))
             (t (twolist 'quote expr)))))

(defun quote-nlambda-arg (expr)
  (let ((expr (expand-macro expr)))
       (cond ((atom expr) expr)
             ((application-p expr) (quote-application expr))
             (t (mapcardot (function quote-nlambda-arg) expr)))))

(defun quote-application (expr)
  (cond
    ((or (variable-p expr) (quoted-p expr)) expr)
    ((lambda-appl-p expr)
     (cons (first expr)
           (mapcar (function quote-lambda-arg) (rest expr))))
    ((nlambda-appl-p expr)
     (cons (first expr)
           (mapcar (function quote-nlambda-arg) (rest expr))))))
```

Listing der Datei variable.l:

```
(setq variable.lfns
  '(variable-p name-of level-of var-p non-var-p print-bindings
    rename-variables))

(defmacro variable-p (expr)
  (list 'and
        (list 'consp expr)
        (list 'eq (list 'first expr) '(quote ?))))

(defmacro name-of (var) (list 'second var))

(defmacro level-of (var)
  (list 'and (list 'eq (list 'length var) 3) (list 'third var)))

(defun var-p (x environment)
  (variable-p (ultimate-assoc x environment)))

(defun non-var-p (x environment) (not (var-p x environment)))

(defun print-bindings (rev-environment-left environment)
  (cond
    (rev-environment-left
     (let ((variable (first (first rev-environment-left))))
          (cond ((zerop (level-of variable))
                 (pp-external-form
                  (list variable
                        '" = "
                        (ultimate-inst variable environment)))
                 (terpr)
                 (line-control))))
     (print-bindings (rest rev-environment-left) environment))))

(defun rename-variables (term listified-level)
  (cond
    ((variable-p term) (append term listified-level))
    ((atom term) term)
    (t (cons (rename-variables (first term) listified-level)
             (rename-variables (rest term) listified-level)))))
```