Lisplog
Benutzerhandbuch

Ansgar Bernardi, Michael Dahmen,
Manfred Meyer
SEKI Working Paper 87-01

LISPLOG

Benutzerhandbuch
Rev. 7 10-Feb-88 FRANZ LISP

Ansgar Bernardi

Michael Dahmen

Manfred Meyer

Fachbereich Informatik
SFB 314 Bau 14
Universitaet Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern 1
W. Germany

uucp: unido!uklirb!lisplog
oder lisplog@uklirb.UUCP

SEKI-Working-Paper SWP-87-01

Januar 1987

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

Abstract
========

This paper is intended to serve as a users manual to LISPLOG.

It gives a short description of the syntax of LISPLOG, together with an explanation of all commands available on the toplevel of LISPLOG.
The concepts and the commands of the LISPLOG tracer/breaker are also described.

This manual should give you the ability to use the LISPLOG system.
It doesn't explain the concepts of the LISP/PROLOG-integration or its implementation in detail.

Inhalt
======

0. Vorbemerkung: Die im Handbuch verwendete Syntax

In diesem Handbuch wird die Syntax von Kommandos, Anfragen und sonstigen Eingaben in EBNF angegeben. Dabei bedeuten:

[] eckige Klammern
Der von ihnen eingeschlossene Ausdruck ist optional
Beispiel: save-all [<dateiname>]

{}* geschweifte Klammern, Stern:
Der von ihnen eingeschlossene Ausdruck kann beliebig oft wiederholt werden, er kann auch entfallen.
Beispiel: save {<Modulname>}*

{}+ geschweifte Klammern, Plus:
Der von ihnen eingeschlossene Ausdruck kann beliebig oft wiederholt werden, er muss jedoch mindestens einmal auftreten.
Beispiel: create {<Modulname>}+

| senkrechter Strich:
Er trennt Alternativen

Die in spitze Klammern eingeschlossenen Ausdruecke sind immer Variable, an deren Stelle beim Aufruf das gewuenschte Argument einzusetzen ist.

Beispiel:
Erzeugen eines Moduls.
im Handbuch steht:

create {<Modulname>}+

um das Modul 'Modul1' zu erzeugen, muss

create Modul1

eingegeben werden.

Die nicht in Klammern eingeschlossenen Ausdruecke hingegen sind Kommandos, die wie angegeben eingetippt werden muessen.

1. Aufruf des LISPLOG-Systems

Das LISPLOG-System ist in FRANZ-LISP eingebettet und wird mit diesem zusammen durch das Kommando

Lisplog

geladen. Dieses Kommando aktiviert eine Kommandosequenz, die folgendes bewirkt:

1. Eine Lademeldung wird ausgegeben
2. Eine in der aktuellen Directory (.) eventuell vorhandene Datei mit dem Namen ".lisprc.l" wird in ".oldlisprc.l" umbenannt. Sodann wird eine speziell auf LISPLOG zugeschnittene Datei als .lisprc.l im aktuellen Directory gespeichert. Nach dem Verlassen des LISPLOG-Systems wird der alte Zustand wieder hergestellt.
3. Das FRANZ-LISP-System wird gestartet und der LISPLOG-Interpreter geladen.

Nach dem Laden des Systems befindet sich der Benutzer auf dem FRANZ-LISP-Toplevel. Mit dem Aufruf (lisplog) wird das LISPLOG-System gestartet, der Benutzer befindet sich dann auf dem LISPLOG-Toplevel. Dieser kann an dem Promptzeichen "*" erkannt werden.
Der Benutzer kann nun Anfragen eingeben oder mit Hilfe der im folgenden beschriebenen Kommandos Datenbasen bearbeiten oder sich naehere Informationen ueber einen Beweisablauf verschaffen. Kommandos haben die Form

<Kommandoname> <Argumente>*

Anfragen haben entweder die Form

<Praedikat> <Argumente>*

oder

(<Praedikat-1> <Argumente>*) (<Praedikat-2> <Args>*) ...

wobei die Praedikate logisch und-verknuepft sind. Das System versucht zunaechst eine Eingabe als LISPLOG-Toplevel-Kommando zu interpretieren, wenn dies nicht moeglich ist als LISPLOG-Anfrage.

Mit dem Kommando "lisp" verlaesst der Benutzer das LISPLOG-System und kehrt auf den FRANZ-LISP-Toplevel zurueck. Der Wechsel zwischen LISPLOG-Toplevel und FRANZ-LISP-Toplevel aendert den Zustand des LISPLOG-Systems nicht.

Beachte :
Das LISPLOG-System liest die Benutzereingabe mit der FRANZ-LISP-Funktion "lineread" ein. Diese Funktion liest solange S-Ausdruecke ein, bis die Eingabe mit <RETURN> abgeschlossen wird;

unmittelbar vor dem <RETURN> darf kein Leerzeichen, TAB o. ae. stehen! (Steht unmittelbar vor <RETURN> ein Leerzeichen, so wartet lineread auf weitere Eingaben. Damit ist es moeglich, Eingaben zu machen, die laenger als eine Zeile sind. Wird ein solches Leerzeichen irrtuemlich eingegeben, so wartet der Benutzer anschliessend vergeblich auf eine Antwort des Systems.)

2. Syntax von LISPLOG

Die Syntax von LISPLOG ist der LISP-Syntax angeglichen, sie weicht daher etwas von der ueblichen PROLOG-Syntax wie sie z.B. in [Clocksin & Mellish 1981/84] beschrieben ist ab. Die Unterschiede werden am schnellsten an einigen Beispielen klar; jeweils [Clocksin & Mellish 1981/84] (1) und LISPLOG (2).

```
(1)  likes(john,X):-likes(Y,X),likes(john,Y).
(2)  ((likes john _x) (likes _y _x) (likes john _y))
```

Eine LISPLOG-Klausel ist eine Liste, deren erstes Element der Kopf der Klausel (die Konklusion) ist, die uebrigen Elemente sind die Praemissen der Klausel. Kopf und Praemissen sind S-Ausdruecke, deren erstes Element jeweils das Praedikat ist, die uebrigen Elemente sind dann die Argumente des Praedikats. Variable werden in LISPLOG durch LISP-Atome dargestellt, denen ein Unterstrich ('_') vorangestellt ist.

```
(1) likes(mary,_)
(2) ((likes mary ID))
```

Anonyme Variable werden durch das reservierte Atom 'ID' dargestellt. Eine Klausel ohne Praemissen ist ein Fakt.

Anmerkung: Intern werden die (nichtanonymen) Variablen in Form von Listen repraesentiert. Das erste Element einer solchen Liste ist ein Fragezeichen, das zweite Element der Name der Variablen. Die Umwandlung zwischen externer und interner Darstellung geschieht durch read/print-Macros bzw. Umwandlungsroutinen in LISP.

```
(1) absolut(X,X):-0<X.
    absolut(0,0):-!.
    absolut(X,Y):-0>X,Y is 0-X.
(2) ((absolut _x _x) (lessp 0 _x))
    (! (absolut 0 0))
    ((absolut _x _y) (greaterp 0 _x) (is _y (sub 0 _x)))
```

PROLOG Built-In-Praedikate werden durch LISP-Funktionsaufrufe (hier "lessp", "greaterp") realisiert. Die zweite Praemisse der letzten Klausel nutzt das LISPLOG-Primitive "is". Damit ist die Evaluation beliebiger LISP-Ausdruecke moeglich. Details dazu in Abschnitt 7.

LISPLOG erlaubt nur den initialen Cut, der in Cprolog als 'Cut als erste Praemisse einer Klausel' auftritt (im Beispiel die zweite Klausel).

Syntax: (!conclusion {premisse}*)

Generell wird also bei Klauseln die (immer vorhandene) Konklusion conclusion mit einem "!" praefigiert, also in den Cut-Ausdruck eingebettet. Damit soll ausgedrueckt werden, dass sich der initiale cut-Operator auf die gesamte Klausel bezieht und wirksam wird sobald die Anwendbarkeits-Unifikation zwischen einer Anfrage und der Konklusion erfolgreich war.
Intern wird eine Klausel mit initialem Cut in Form einer Liste dargestellt. Die Umwandlung erfolgt wie bei der Darstellung von Variablen ueber Read-Print- Macros.

```
(1) append([],X,X).
    append([H|T],X,[H|R]):-append(T,X,R).
(2) ((append nil _x _x))
    ((append (_h . _t) _x (_h . _r)) (append _t _x _r))
```

In PROLOG werden Listen als spezielle Terme dargestellt. In LISPLOG kann zur Darstellung der Listen die LISP Liste benutzt werden. Das LISP-Konstrukt der 'dotted-list' erlaubt dabei auch die Definition von Praedikaten variabler Aritaet, was in PROLOG nicht moeglich ist.

```
(2) ((addiere-n 0))
    ((addiere-n _sum _next . _rest)
     (addiere-n _part-sum . _rest)
     (is _sum (add _next _part-sum)))
```

Dieses Praedikat bestimmt die Summe des 2. bis n-ten Arguments als Wert des ersten Arguments.

Auf Dateien werden die LISPLOG-Klauseln nicht als einfache Klauseln abgespeichert, sondern als Aufrufe der LISP-Funktion 'ass', die das Einfuegen von Klauseln in die Datenbasis uebernimmt.
Beispiel:
Die Klausel ((likes john mary) (likes mary john)) wird in einer Datei als (ass (likes john mary) (likes mary john)) gespeichert.
Wird eine Datei erstellt, die LISPLOG-Klauseln enthaelt, oder wird eine solche Datei editiert, so muessen Klauseln ebenfalls in dieser Form eingegeben werden.
Diese Form einer Klausel, also die Klausel mit vorangestellter Funktion 'ass', wird im weiteren Text als 'ass-Ausdruck' bezeichnet.

Die Syntax von LISPLOG kann in EBNF wie folgt beschrieben werden:

```
LISPLOG-Program    ::= { clause | function }*

clause             ::= ( conclusion { premise }* )

conclusion         ::= cut-conclusion |
                       simple-conclusion

premise            ::= predicate-call |
                       function-call  |
                       primitive-call

predicate-call     ::= (predicate {argument}* [.variable])
primitive-call     ::= (is argument argument) |
                       (not premise) |
                       (varp argument) |
                       (nonvarp argument)

predicate          ::= predicate-constant |
                       predicate-variable
predicate-constant ::= LISP-atom
predicate-variable ::= simple-variable
                   (muss beim Aufruf gebunden sein)

cut-conclusion     ::= !simple-conclusion
simple-conclusion  ::= (predicate-constant
                              {argument}* [.variable])

argument           ::= variable | LISP-expression
variable           ::= anonymous-variable | simple-variable

simple-variable    ::= _LISP-atom
anonymous-variable ::= ID


Lisp-expression    ::= LISP-atom | list
list               ::= ({LISP-expression}* [.LISP-expression])
```

LISP-atome duerfen !, _ oder ? nicht enthalten, da diese Zeichen vom LISPLOG-System ueber read/print-Macros speziell interpretiert werden.

3. Module in LISPLOG

In LISPLOG gibt es ein Modulsystem, das aehnlich arbeitet wie entsprechende Systeme in MODULA-2, ADA oder COMMON-LISP. Das Modulsystem soll groessere LISPLOG-Entwicklungen unterstuetzen.

3.1. Leistungen des Modulsystems

Das Modulsystem ermoeglicht beliebig viele Module in einem LISPLOG-Programm. Jedes Modul bildet einen eigenen Namesraum fuer die Praedikatnamen d.h. ein Praedikatsname referiert in verschiedenen Modulen auf verschiedene Praedikatsdefinitionen. Durch seperate Import- und Exportdeklarationen kann ein Praedikatname in mehreren Modulen auf dieselbe Definition referieren.

Ein Modul ist somit eine LISPLOG (Teil-) Datenbasis. Betrachtet man einige Praedikate in dem Modul als dynamisch (d.h. waehrend der Programmlaufzeit) veraenderbar, so hat jedes Modul einen eigenen Zustand naemlich die Klauseln eben dieser Praedikate. Dadurch ist es moeglich, mit einem Modul eine abstrakte (Zustands-) Maschine zu realisieren.

3.2. Aufbau eines Moduls

Das Modulsystem ordnet jedem Modul eine Datei zu, in der die Klauseln und Deklarationen des Moduls gespeichert werden. Wird die Datei zu einem spaeteren Zeitpunkt in ein LISPLOG-System geladen, so wird der Zustand des Moduls zum Zeitpunkt der Abspeicherung wiederhergestellt. Um dies zu erreichen, erhaelt die Datei neben den Klauseln und Deklarationen noch einige Kommandos. Eine Datei hat damit typischerweise folgenden Aufbau :

```
(destroy <Modulname>)
(create <Modulname>)
(switch-to-modul <Modulname>)

(ass (epxort$ <Export-Deklaration>))
(ass (import$ <Import-Deklaration>))

(ass <LISPLOG-Klauseln>)
```

Erlaeuterungen :
Die ersten drei Zeilen sind Kommandos die bewirken dass

- ein ggf. vorhandes Modul gleichen Names geloescht wird
- ein neues Modul kreiert wird
- dieses neue Modul zum aktuellen Modul wird

Die nachfolgenden Export- und Importdeklarationen bilden die Schnittstelle dieses Moduls zu den anderen Modulen im System. Der genaue Aufbau der Deklarationen wird weiter unten erklaert. Danach folgen die LISPLOG-Klauseln dieses Moduls in der ueblichen LISPLOG-Notation.

3.3. Semantik der Export- und Importdeklarationen

Mit einer Exportdeklaration wird festgelegt, welche Praedikate aus diesem Modul exportiert werden. Diese Praedikate koennen von anderen Modulen importiert werden und dann in Klauseln dieser Module benutzt werden. Alle nicht exportierten Praedikate sind ausserhalb des definierenden Moduls unsichtbar und koennen nicht angesprochen werden.

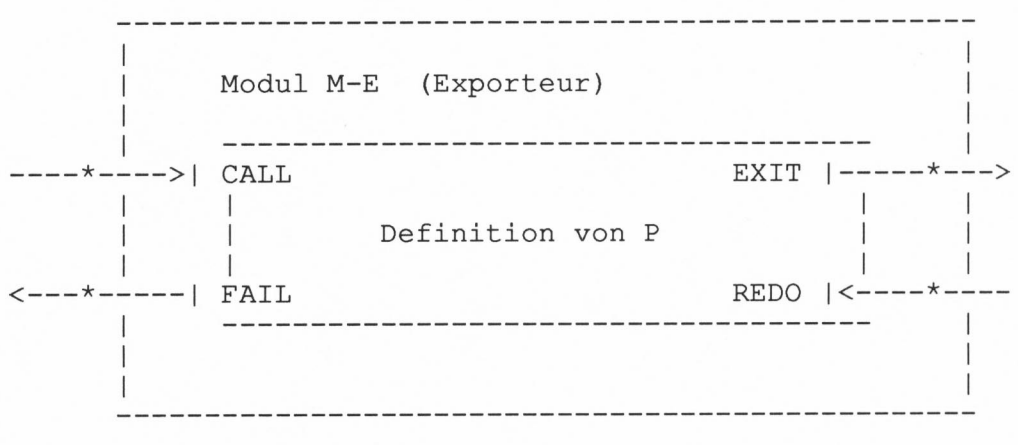
Mit einer Importdeklaration wird angegeben, aus welchem Modul welche Praedikate importiert werden sollen. Dabei koennen natuerlich nur Praedikate aus einem Modul importiert werden, die in dem exportierenden Modul in der Exportliste auftreten. Diese Ueberpruefung geschieht zur Laufzeit des Programms. Ausserdem kann diese Ueberpruefung statisch fuer alle Module durchgefuehrt werden. Da die statische Ueberpruefung nur auf Anforderung des Benutzers geschieht, sind waehrend der Programmerstellung momentan inkonsistene Deklarationen moeglich.

Die importierten Praedikate muessen voneinander und von den im Importeur definierten Praedikaten verschieden sein. Eine Ueberlagerung von Klauseln eines Praedikats aus verschiedenen Modulen ist nicht zugelassen, da die Klauseln eines Praedikats als unteilbare Einheit (Prozedur) betrachtet werden.

3.4. Ablauf eines Beweises ueber Modulgrenzen

Waehrend eines LISPLOG-Beweises oder einer interaktiven Benutzung des LISPLOG-Systems ist immer ein Modul der aktuelle Modul. Alle Operationen auf der Datenbasis (ass, rex, listing usw.) beziehen sich auf diesen aktuellen Modul. Dies gilt insbesondere auch fuer dynamische Aenderungen der Datenbasis.

Bei der Durchfuehrung eines Beweises wird immer der aktuelle Modul mitgefuehrt. Soll ein Teilziel bewiesen werden, so kann aufgrund der Importdeklarationen des aktuellen Moduls zunaechst ermittelt werden, ob das Praedikat P des Teilziels importiert ist und wenn ja aus welchem Modul importiert wird. Ist das Praedikat nicht importiert, arbeitet der LISPLOG-Interpreter wie ueblich. Andernfalls wird der aktuelle Modul gewechselt. Dieser Vorgang laesst sich am besten an einer Modifikation des Box-Modells darstellen.



Die innere Box stellt die Definition des Praedikats P im Modul M-E (Exporteur) dar. Das Praedikat P wird vom Modul M-I (Importeur) aus aufgerufen. Die aeussere Box symbolisiert die Grenze zwischen den Modulen M-E und M-I. An den mit * markierten Stellen schneidet der Kontrollfluss die Modulgrenzen; dort wird der aktuelle Modul gewechselt. Der Wechsel erfolgt also (zeitlich) vor CALL und REDO bzw. nach EXIT und FAIL. Bei Verwendung des Tracers werden beim Wechsel des aktuellen Moduls entsprechende Meldungen ausgegeben.

3.5. Der Modul "lisplog-basis"

Der Modul mit dem Namen "lisplog-basis" ist fuer Praedikate vorgesehen, die in allen Modulen direkt (d.h. ohne explizite Importdeklaration) verfuegbar sein sollen. In dem Modul befinden sich insbesondere Praedikate, die als LISPLOG-Primitive angesehen werden. Der Beweisablauf unter Einbeziehung von LISPLOG-Basis erfolgt nicht in der oben beschriebenen Art. Bei Verwendung von Praedikaten aus "lisplog-basis" wird der aktuelle Modul nicht gewechselt. Dies kann man auch so darstellen, als ob die Praedikate von "lisplog-basis" sich auch in jedem anderen Modul befinden.
Verschiedene Lisplog-Primitive sind in diesem Modul definiert. Da der Loesch-Befehl "destroy-all" (siehe unten) auch dieses Modul loescht, die Primitve aber weiterhin definiert sein sollen, werden sie bei Bedarf aus einer speziellen Datei neu geladen.
Diese Datei traegt den Namen "lisplog.prelude.2.1". Sie ist dem Benutzer nicht zugaenglich, sondern gilt als Bestandteil des Systems.
Der Benutzer kann jedoch nach Bedarf weitere Definitionen zum Modul "lisplog-basis" hinzufuegen oder auch Definitionen loeschen. Derartige Aenderungen gehen jedoch beim Loeschen des Moduls verloren.

4. Die Benutzung des Systems

Um eine Klausel auf dem LISPLOG-Toplevel einzugeben, muss zunaechst das entsprechende Kommando eingegeben werden. Das Kommando heisst 'ass', die Kurzform (zum schnelleren Tippen auf dem Toplevel) heisst '+'.
Hinter diesem Kommando wird dann die Klausel eingetippt. Dank der oben beschriebenen Eigenschaft von lineread, das die Eingabe als Liste zusammenfasst, muss das auessere Klammerpaar einer Klausel nicht eingetippt werden.
Beispiel:
Um die Klausel ((likes john mary) (likes mary john)) in die Datenbasis einzufuegen, muss
+ (likes john mary) (likes mary john)[RETURN]
eingetippt werden. Die Klausel wird an das Ende der Datenbasis angefuegt. Das Praedikat des Klauselkopfes darf keine Variable sein. Die Praedikate der Praemissen koennen Variable sein, es muss aber sichergestellt sein, dass die Variablen zum Zeitpunkt des Aufrufs der Klausel gebunden sind.
Beispiel:
Die Klausel ((_x john mary)) ist unzulaessig.
Die Klausel ((besteht _beziehung john mary) (_beziehung john mary)) ist als solche zulaessig. Allerdings muss das erste Argument beim Aufruf gebunden sein. So liefert der Aufruf (besteht likes john mary) ein Ergebnis, (besteht _y john mary) jedoch einen Fehler, wenn _y nicht gebunden ist.

Die Eingabe der Klauseln direkt auf dem LISPLOG-Toplevel ist nur bei sehr kleinen Beispielen zu empfehlen, da bei dieser Eingabe keinerlei Editierung moeglich ist. Es ist daher besser den Texteditor vi zu Erstellung der Dateibasis zu benutzen und die Textdatei dann mit dem Kommando "consult" zu laden. Dabei ist zu beachten, dass in Dateien die LISPLOG-Klauseln als 'ass-Ausdrucke' notiert werden muessen (vgl. Abschnitt 2).

5. Die LISPLOG-Kommandos

Die nachfolgenden LISPLOG-Kommandos sind als LISP-Funktionen (Typ nlambda) implementiert. Sie sind daher dem LISPLOG-Toplevel und auf dem FRANZ-LISP-Toplevel aufrufbar und auch als Praemissen in LISPLOG-Klauseln moeglich. Die Kurzformen sind nur auf dem LISPLOG-Toplevel moeglich. Auf dem LISPLOG-Toplevel entfaellt ausserdem das auessere Klammerpaar bei der Eingabe.
Verwendet man das Modulsystem nicht, so werden alle Operationen auf dem Default-Modul "clauses" durchgefuehrt. In diesem Fall ist das aktuelle Modul immer "clauses" und zugleich auch die gesammte Datenbasis des LISPLOG-Systems.

5.1. Einfuegen von Klauseln

(ass <Klausel>)
Kurzform: (+ <Klausel>)
Returns : t
Effekt : Die Klausel wird in das aktuelle Modul als letzte Klausel des Praedikats aufgenommen.

(ass-a <Klausel>)
Returns : t
Effekt : Die Klausel wird in das aktuelle Modul als erste Klausel des Praedikats aufgenommen.

5.2. Loeschen von Klauseln

(rex <Klausel>)
Kurzform: (- <Klausel>)
Returns : t wenn Klausel vorhanden war, sonst nil.
Effekt : Die Klausel wird aus der Definition des Praedikats im aktuellen Modul entfernt, sofern sie vorhanden war. Der Vergleich der Klauseln erfolgt mit "equal", es findet also keine Unifikation beim Vergleich statt.

(abolish {<Praedikatname>}+)
Returns : t
Effekt : Die Definitionen der angegebenen Praedikate im aktuellen Modul werden geloescht.

(destroy {<Praedikatname> | <Modulname> | <LISP-Funktionsname>}*)
Returns : t
Default : das aktuelle Modul
Effekt : Die angegebenen Praedikate im aktuellen Modul und die angegebenen Module werden geloescht. Wird das aktuelle Modul geloescht, so wird auf das Modul "clauses" geschaltet, das ggf. zuvor erzeugt wird.
Werden Praedikate oder Module geloescht, so bleiben die

LISP-Funktionen unbeeinflusst. Wird ein LISP-Funktionsname angegeben, so wird die entsprechende Funktion fuer das System unbekannt. Sie bleibt aber weiterhin definiert.

(destroy-all)
Returns : clauses
Effekt : Alle Module werden geloescht. Das Modul "clauses" wird neu erzeugt und zum aktuellen Modul gemacht.
Alle dem System bekannten LISP-Funktionen (vgl. 5.11.) werden dem System unbekannt, bleiben aber weiterhin definiert.

5.3. Anzeigen von Klauseln auf dem Bildschirm

(listing {<Praedikatname>|<Modulname>|<LISP-Funktionsname>}*)
Kurzform: (l {<Praedikatname>|<Modulname>|<LISP-Funktionsname>}*)
Returns : t
Default : das aktuelle Modul & alle LISP-Funktionen
Effekt : Die angegebenen Praedikate im aktuellen Modul und die angegebenen Module werden aufgelistet. Die Klauseln werden auf dem Bildschirm in Form von ass-Ausdruecken dargestellt. Es werden nur LISP-Funktionen aufgelistet, die dem System bekannt sind (vgl. 5.11.).

5.4. Editieren von Klauseln

(edit [<Praedikatname>|<Modulname>|<LISP-Funktionsname>])
Returns : t
Default : das aktuelle Modul
Effekt : Der Texteditor "vi" wird aufgerufen. Er laedt eine Datei, die die Definition des Praedikats, der Funktion bzw. des Moduls enthaelt. Wird ein Modulname angegeben, so enthaelt die Datei zu den Klauseln des Moduls noch alle dem System bekannten LISP-Funktionen (vgl. 5.11.).
Der Texteditor wird mit ":w :q" verlassen, die geaenderte Definition wird eingelesen und ueberschreibt die alte Definition.

5.5. Laden einer Datei

(consult {<Dateiname>}*)
Returns : {<Dateiname>}*
Default : system.wrk.db
Effekt : Die angegebenen Dateien, die LISPLOG Module enthalten, werden geladen. Die Dateien sind Textdateien.

Eine solche Datei kann LISPLOG-Klauseln in Form von ass-Ausdruecken enthalten, ebenso aber auch andere evaluierbare

LISP-Expressions.

Warnung: Von allen LISP-Ausdruecken, die in einer solchen Datei moeglich sind, werden dem System nur mit 'def' erfolgende Funktionsdefinitionen bekannt (vgl. 5.11.)
Nur diese koennen von LISPLOG aus bearbeitet, editiert und insbesondere wieder gespeichert werden.
Zwischen diesem Kommando und dem Gegenstueck "save-all" besteht daher folgende Unsymmetrie: Waehrend "consult" beliebige LISP-Ausdruecke (setq ...) laden kann, speichert "save-all" nur LISPLOG-Klauseln und dem System bekannte Funktionsdefinitionen. Wenn eine Datei also ausser LISPLOG-Klauseln und 'def'-Funktionsdefinitionen noch andere LISP-Ausdruecke enthaelt, so ist sie nach der Folge
consult <datei>
save-all <datei>
veraendert! Entsprechendes gilt fuer die Funktion tell!

(reconsult {<Dateiname>}*)
Returns : {<Dateiname>}*
Default : system.wrk.db
Effekt : Die angegebenen Dateien, die LISPLOG Module enthalten, werden geladen. Die Dateien sind Textdateien. Praedikate, die bereits in der Datenbasis enthalten sind, werden durch die neuen Definitionen ersetzt.

Achtung! Eine Datei, die consult-Aufrufe enthaelt, darf nicht mit reconsult geladen werden. Das Ergebnis eines solchen Aufrufs ist undefiniert, da die in der Datei enthaltenen consult-Aufrufe den reconsult-Modues wieder abschalten!

5.6. Speichern von Klauseln

(save {<Modulname>}*)
Returns : t
Default : das aktuelle Modul
Effekt : Die angegebenen Module werden auf Textdateien gesichert. Die Dateinamen sind fuer jedes Modul unter der Property "dateiname$" gespeichert. Nach "create" ist der Dateiname der Name des Moduls, nach "consult" der Name der Datei, aus der das Modul geladen wurde. Der Dateiname kann mit "(ass (dateiname$ <Dateiname>))" geaendert werden. Dem System bekannte LISP-Funktionen werden mit diesem Kommando nicht gesichert! (vgl. 5.11.)

(save-all [<Dateiname>])
Returns : t
Default : system.wrk.db
Effekt : Alle Module werden gespeichert (siehe "save"). In die angegebene Datei werden die Kommandos zum Laden aller Module geschrieben. Ausserdem werden die dem System bekannten LISP-Funktionen (vgl. 5.11) in diese Datei geschrieben.
Es genuegt also, diese Datei zu laden, um den derzeitigem Zustand des LISPLOG Systems wiederherzustellen.

(ass (dateiname$ <Dateiname>))
Returns : wie "ass"
Effekt : Dem aktuellen Modul wird die Datei <Dateiname> zugeordnet. Die Zuordnung wird beim Speichern des Moduls benutzt. Die Default Zuordnung ist "ll.<Modulname>".

(tell <Dateiname>)
Returns : t
Default : system.wrk.db
Effekt : Die Klauseln des aktuellen Moduls werden auf der angegebenen Datei gespeichert. Dann werden alle dem LISPLOG-System bekannten Funktionen (vgl. 5.11.) hinzugefuegt.
Wenn das Modulsystem nicht benutzt werden soll, kann mit Hilfe der Kommandos tell und consult gearbeitet werden, als ob das Modulsystem nicht existiert. Alle Aktionen betreffen in diesem Falle das Default-Modul 'clauses'.

5.7. Erzeugen von Modulen

(create {<Modulname>}+)
Returns : {<Modulname>}+
Effekt : Die angegebenen Module werden erzeugt, sofern sie noch nicht vorhanden sind. Neu erzeugte Module sind leer. Das aktuelle Modul wird nicht gewechselt.

5.8. Wechsel des aktuellen Moduls

(switch-to-modul <Modulname>)
Returns : <Modulname>; falls nicht vorhanden nil.
Kurzform: (> <Modulname>)
Effekt : Schaltet auf das angegebene Modul um, sofern es vorhanden ist.

5.9. Import und Export

(ass (export$ {<Praedikatname>}+))
(rex (export$ {<Praedikatname>}+))
Returns : wie "ass" bzw. "rex"
Effekt : Das aktuelle Modul exportiert zusaetzlich (bzw. nicht mehr) die angegebenen Praedikate.

(ass (import$ <Exporteur> {<Praedikat>}+))
(rex (import$ <Exporteur> {<Praedikat>}+))
Returns : wie "ass" bzw. "rex"
Effekt : Das aktuelle Modul importiert zusaetzlich (bzw. nicht mehr) die angegebenen Praedikate aus dem Modul <Exporteur>.

(import-check {<Modulname>}*)
Returns : {<Modulname>}*
Default : das aktuelle Modul
Effekt : Fuer die angegebenen Module wird geprueft, ob alle importierten Praedikate vom Exporteur auch exportiert werden, und ob alle exportierten Praedikate definiert sind d.h. mindestens eine Klausel besitzen. Erkannte Fehler werden auf der Standardausgabe angezeigt.

Die Semantik dieser Kommandos ist im Abschnitt 3 ausfuehrlich erlaeutert.

5.10. Sonstige Kommandos fuer das Modulsystem

(do-on-modul <Modulname> <lispexpr>)
Returns : Ergebnis der Evaluation; nil falls Modul nicht vorhanden.
Effekt : Der Ausdruck <lispexpr> wird in dem angegebenen Modul ausgewertet.

(print-modul-info)
Returns : nil
Kurzform: (i)
Effekt : Ausgabe aller Modulnamen und der zugeordneten Dateien.

(sort-modul {<Modulname>}*)
Returns : t
Default : das aktuelle Modul
Effekt : Die Praedikate sowie die Import- und Exportlisten der angegebenen Module werden alphabetisch sortiert.

(sort-all-modul)
Returns : t
Effekt : Wendet "sort-modul" auf alle Module des Systems an.

5.11. Ein Hinweis zu LISP-Funktionen

Wie bereits oben erwaehnt, koennen Dateien, die in das LISPLOG-System geladen werden, Klausel- und Funktionsdefinitionen enthalten. Dabei ist jedoch folgendes zu beachten:
Funktionen, die von den Verwaltungsfunktionen des LISPLOG-Systems (listing, spy ...) behandelt werden sollen, muessen mit 'def' definiert werden.
Funktionen, die aus dem LISPLOG-System heraus interaktiv definiert werden, werden von den Verwaltungsfunktionen nicht automatisch erkannt.
Das Modulsystem von LISPLOG betrifft nur Klauseln; LISP-Funktionen sind immer fuer alle Module global. Hieraus erklaert sich das Verhalten der verschiedenen Speicherfunktionen (vgl. 5.6).

5.12. Quotierungsautomatik

Die Funktionsweise der Quotierungautomtik ist in Abschnitt 7.4 erlauetert. Sie kann durch "auto-quote-off" abgeschaltet und mit "auto-quote-on" wieder angeschaltet werden. Dies ist nur auf dem LISPLOG-Toplevel moeglich.

5.13. Berechnen weiterer Loseungen

Das Kommando "more" berechnet weitere Loesungen der letzten Anfrage. Es steht nur auf dem LISPLOG-Toplevel zur Verfuegung.

5.14. Verlassen des LISPLOG-Toplevels

Das Kommando "lisp" bewirkt das Verlassen des LISPLOG-Toplevels und die Rueckkehr ins LISP-System.

6. Anfragen an LISPLOG

Anfragen (goals) werden auf dem LISPLOG-Toplevel eingegeben.
Jede Eingabe, die nicht mit einem Kommando beginnt, wird als Anfrage behandelt. Das System versucht zunaechst, die Anfrage mit Hilfe der in der Datenbasis gespeicherten Klauseln zu beweisen. Gelingt dies nicht, d. h. stimmt das Praedikat der Anfrage nicht mit einem Praedikat der Klauseln in der Datenbasis ueberein, so wird die Anfrage als LISP-Funktionsaufruf interpretiert und evaluiert. Liefert dies einen von nil verschiedenen Wert als Ergebnis, so ist die Anfrage ebenfalls bewiesen, anderenfalls ist sie nicht beweisbar.
Wenn eine Anfrage bewiesen werden konnte, so meldet das System 'success'. Dann werden die Bindungen der in der Anfrage enthaltenen freien Variablen ausgegeben. Wenn der Benutzer jetzt das Kommando "more" eingibt, versucht das System, weitere Beweise der Anfrage (und ggf. andere Variablenbindungen) zu finden.
Bei einem fehlgeschlagenen Beweisversuch (oder nach vergeblicher Suche nach weiteren Loesungen, s.o.) meldet das System 'nil'.

Konjunktionen von Anfragen koennen bewiesen werden, indem die Anfragen einfach hintereinander eingegeben werden. Nachdem [RETURN] gedruckt wurde, beginnt der Beweis.

Die Eigenarten der Einlesefunktion (s. o.) ersparen dem Benutzer in vielen Faellen die Eingabe der aeusseren Klammern.
Beispiel:
(likes john _x) [RETURN] Das System versucht, die Anfrage zu beweisen. Wenn eine Loesung gefunden wurde, wird die Bindung der Variablen _x ausgegeben.
likes john _x [RETURN] hat dieselbe Wirkung. Die Einlesefunktion ergaenzt die aeusseren Klammern.
(likes john _x) (likes _x john) [RETURN] Das System versucht, die Konjunktion der Anfragen (likes john _x) und (likes _x john) zu beweisen, d. h. eine Bindung der Variablen _x zu finden, die beide Anfragen erfuellt. Hier koennen keine Klammern weggelassen werden!

7. Der Anschluss an LISP

LISPLOG ist in LISP eingebettet, d. h. es ist nicht nur in LISP realisiert sondern es bietet auch einen Anschluss an LISP. Dieser ist in zwei Richtungen ausgefuehrt. In LISP-Funktionen kann der LISPLOG-Beweiser aufgerufen werden; er liefert dann Loesungsmengen von Anfragen (Abschnitt 7.1). Als LISPLOG-Praemissen koennen andererseits LISP-Praedikate auftreten (Abschnitt 7.2). Ausserdem koennen LISP-Funktionen im "is" Primitiv auftreten (Abschnitt 7.3). Dies kann durchaus zu rekursiven Aufrufen (LISP -> LISPLOG -> LISP -> ..) fuehren.

7.1. Logik in LISP

Es gibt verschiedene Funktionen zum Aufruf des Interpreters, die alle folgendes Grundmuster haben. Es wird eine Anfrage an LISPLOG formuliert, wie dies auch auf dem LISPLOG-Toplevel geschieht. Ausserdem wird ein Resultatterm angegeben, der einige Variablen der Anfrage enthaelt. Eine Loesung der Anfrage ist dann eine Instanziierung des Resultatterms; mehrere Loesungen werden als Liste dargestellt. Alternativ kann auch statt eines Resultatterms eine Assoziationsliste der Variablen die Loesung repraesentieren.

```
(prove <Goallist> <Limit> <Resultatterm>)
```

Diese Funktion berechnet maximal <Limit> viele Loesungen der Anfrage <Goallist>. Zu beachten ist, dass die Funktion ihre Argumente evaluiert, die Argumente muessen daher meist quotiert werden. Ist der <Resultatterm> "nil", so ist jede Loesung eine Assoziationsliste aller Variablen in der <Goallist>. Beispiele :

```
Klauseln :
((append nil _x _x))
((append (_h . _t) _x (_h . _r)) (append _t _x _r))

(prove '((append (a b c) (d e) _x)) 3 '_x)
-->
( (a b c d e) )
```

Die Goallist enthaelt und-verknuepfte Ziele (hier nur eins) und eine Variable; diese Variable ist auch der Resultatterm. Es werden maximal 3 Loesungen berechnet, da die Anfrage aber nur eine Loesung besitzt, ist das Ergebnis eine Liste mit einem Element.

```
(prove '((append _x _y (a b c d)) 3 nil)
-->
( ((_x . nil) (_y . (a b c d)))
  ((_x . (a)) (_y . (b c d)))
  ((_x . (a b)) (_y . (c d))) )
```

Diese Anfrage besitzt mehr als drei Loesungen, berechnet werden die ersten drei. Jede Loesung ist durch eine Assoziationsliste der Variablen in der Anfrage dargstellt, wobei die Assoziationslisten die in LISP uebliche Form haben.

```
(n-solutions <Goal> <Limit>)
-->
(prove (<Goal>) <Limit> nil)
```

Dies ist eine andere Form des Aufrufs (uebernommen aus einer aelteren Version von LISPLOG).

Die obige Form der LISPLOG-Schnittstelle erscheint unangemessen, denn es ist moeglicherweise erst nach Berechnen der ersten Loesungen klar, wieviele weitere Loesungen benoetigt werden. Der Aufruf von "n-solutions" bzw. "prove" sollte also nicht eine Liste aller n Loesungen liefern. Vielmehr sollte bereits nach Berechnen der ersten Loesung die Kontrolle an die aufrufende LISP-Funktion zurueckgehen, die dann allerdings die Moeglichkeit haben muss weitere Loesungen anzufordern.

Dieses Berechnungsmodell kann man mit einer sequentiellen Datei vergleichen, wobei jeder Datensatz der Datei einer Loesung der LISPLOG-Anfrage entspricht. Die einzelnen Loesungen werden aber erst berechnet, wenn sie benoetigt werden. Diese anforderungsgesteuerte Berechnung erlaubt es daher auch Loesungsmengen unendlicher Kardinalitaet darzustellen, die aber natuerlich nicht vollstaendig enumeriert werden koennen. Eine solche Schnittstelle wird durch drei Funktionen des LISPLOG-Interpreters realisiert.

```
(open-lisplog-stream <goallist> <resultatterm> <name>)
(get-lisplog-stream <name>)
(close-lisplog-stream <name>)
```

Das nachfolgende einfache Beispiel zeigt die Moeglichkeiten, die das LISPLOG-System mit diesen Funktionen bietet. Zunaechst einige Klauseln, die die Aufzaehlung aller Primzahlen durch Backtracking organisieren.

```
(ass (primzahl _x) (zahl _x) (not (hat-teiler _x)))

(ass (zahl 1))
(ass (zahl _n) (zahl _k) (is _n (add1 _k)))

(ass (hat-teiler _x)
     (zahl-zwischen 2 (sub1 _x) _y)
     (equal 0 (mod _x _y)))

(ass (zahl-zwischen _low _high _low) (lessp _low _high))
(ass (zahl-zwischen _low _high _x)
     (lessp _low _high)
     (is _low1 (add1 _low))
     (zahl-zwischen _low1 _high _x))
```

Mit diesen Klauseln werden nun die nachfolgenden LISP-Ausdruecke wie folgt evaluiert.

```
(IN)  (open-lisplog-stream '((primzahl _x))
                           '(_x "ist Primzahl")
                           'primes)
(OUT) primes
(IN)  (get-lisplog-stream 'primes)
(OUT) (1 "ist Primzahl")
(IN)  (get-lisplog-stream 'primes)
(OUT) (2 "ist Primzahl")
(IN)  (get-lisplog-stream 'primes)
(OUT) (3 "ist Primzahl")
(IN)  (get-lisplog-stream 'primes)
(OUT) (5 "ist Primzahl")
(IN)  (get-lisplog-stream 'primes)
(OUT) (7 "ist Primzahl")
(IN)  (get-lisplog-stream 'primes)
(OUT) (11 "ist Primzahl")
usw.
```

7.2. LISP-Praedikate

In LISPLOG besteht die Moeglichkeit, beliebig geschachtelte Lispausdruecke als Praemissen zu verwenden. Solche Lispausdruecke werden als Praedikate angesehen. Liefert die Evaluation in LISP den Wert "nil", so hat das Praedikat den Wert falsch ansonsten den Wert wahr. Dies stimmt mit der in LISP ueblichen non-nil Konvention ueberein, so dass viele LISP-Funktionen direkt als LISPLOG-Praedikate verfuegbar sind. Eine Praemisse wird dann als LISP-Praedikat interpretiert, wenn kein LISPLOG-Praedikat gleichen Namens in der Datenbasis existiert, und der auesserste Funktor des LISP-Praedikats eine LISP-Funktionsdefinition besitzt. Vor der Evaluation des LISP-Ausdrucks wird dieser mit einer Quotierungsautomatik behandelt. Diese ist in Abschnitt 7.4 erklaert.

7.3. Uebernahme von LISP-Funktionsergebnissen: is-Primitiv

Das is-Primitiv dient dazu, Werte von LISP-Funktionen in das LISPLOG-System zu uebernehmen. Es hat die Syntax :

(is <LISPLOG-Term> <LISP-Ausdruck>)

Der LISP-Ausdruck wird, nach vorheriger Quotierung (Abschnitt 7.4) evaluiert und dann mit dem LISPLOG-Term unifiziert. In den meisten Faellen ist der LISPLOG-Term eine Variable, die durch diese Operation gebunden wird. Ist die Variable schon gebunden, so werden Funktionsergebnis und Wert der Variablen auf Gleichheit ueberprueft. Ist die Unifikation zwischen dem LISPLOG-Term und der Evalution des LISP-Ausdrucks nicht moeglich, so fuehrt das is-Primitiv zu einem Failure und nachfolgendem Backtracking. Dies ist aber eine unuebliche Nutzung des is-Primitivs.

Beispiel:

```
   Klausel : ((times-p _x _y _z) (is _z (times _x _y)))
   Anfrage : (times-p 2 3 _z)
             --> success mit _z=6
   Anfrage : (times-p 2 3 6)
             --> success ohne Bindung
   Anfrage : (times-p 2 3 7)
             --> failure
   Anfrage : (times-p _x _y _z)
             --> Laufzeitfehler
```

Der Laufzeitfehler bei der letzten Anfrage entsteht, weil die LISP-Funktion "times" nur Zahlen verarbeiten kann, nicht aber ungebundene LISPLOG-Variablen (s. Abschnitt 7.5).

7.4. Quotierungsautomatik

LISP-Ausdruecke, die als Praemissen (Abschnitt 7.2) gebraucht werden oder im is-Primitiv auftreten, werden vor der Evalutation mit einer Quotierungsautomtik behandelt. Es ist daher im allgemeinen nicht notwendig, die Argumente der aufgerufenen Funktionen zu quotieren.

Dieser Vorgang laueft wie folgt ab. Zunaechst werden LISPLOG-Variablen im Lispausdruck mit ihrem Wert instanziiert. Anschliessend wird eine Quotierungsautomatik angewandt, die es dem Benutzer erlauben soll, auf eine explizite Quotierung der Parameter der Lispausdruecke weitgehend zu verzichten. Kern dieser Quotierungsautomatik ist eine Konvention zur Unterscheidung von Datenlisten und Funktionsapplikationen:

Eine Liste wird als Applikation aufgefasst, wenn ihr erstes Element ein Lisp-Funktionsname ist, sonst als Datenliste. Durch explizite Quotierung kann eine Liste immer eindeutig als Datenliste gekennzeichnet werden (wie in Lisp - wichtig, falls der Benutzer nicht alle Funktionsnamen kennt).

Wirkungsweise der Quotierungsautomatik:

(Der gesamte Lispausdruck wird dabei so behandelt, als ob er ein Argument einer lambda-Applikation ist.)

Argumente von lambda-Applikationen
(dazu zaehlen hier auch lexpr-Applikationen) werden, falls sie macro-Applikationen sind, zunaechst expandiert (z.B. (first (a b c)) zu (car (a b c))).
Das (eventuell expandierte) Argument wird quotiert, wenn es entweder ein symbolisches Atom oder eine Liste ist, deren erstes Element keine Funktion darstellt.
Numerische Atome und die leere Liste werden nicht quotiert, da sie selbstevaluierend sind, d.h. bei einer Auswertung sich selbst zum Wert haben. Das gleiche gilt fuer (freie) LISPLOG-Variablen, die dadurch im anschliessenden Test auf freie Variable erkannt werden koennen (siehe unten). Auch explizit quotierte Ausdruecke bleiben unveraendert. So kann eine Liste immer eindeutig als Datenliste gekennzeichnet werden.
In Funktionsapplikationen, d.h. Listen, deren erstes Element ein Funktionsname oder ein lambda- bzw. nlambda-Ausdruck ist, werden nur die Argumente entsprechend der Definitionsart der Funktion behandelt.

Argumente von nlambda-Applikationen
werden ebenfalls zunaechst expandiert. Sie werden nicht quotiert, da nlambda-Funktionen ihre Argumente bei der Parameteruebergabe nicht evaluieren.
Falls sie nicht explizit quotiert sind, wird allerdings ueberprueft, ob sie selbst oder in ihnen eingeschachtelte Ausdruecke Funktionsapplikationen sind. In diesem Fall werden die Argumente dieser Applikationen entsprechend der Definitionsart (lambda oder nlambda) der Funktion behandelt. Dadurch wird z.B. im Ausdruck

```
(cond ((atom (first (a b c))) (first (a b c))))
```

die Liste (a b c) beide Male quotiert (cond ist eine nlambda-Funktion).

Nach abgeschlossener Quotierung wird der Lispausdruck auf freie LISPLOG-Variablen durchsucht, die ausserhalb von quotierten Kontexten (explizit quotierte Ausdruecke, nlambda-Applikationen oder Funktoren von Funktionsapplikationen) auftreten.
Falls keine freie Variable gefunden wurde, wird der Lispausdruck evaluiert, das Resultat wird wie in den vorherigen Abschnitten erlauetert verwendet. Andernfalls wird der Lispausdruck nicht ausgewertet und direkt ein failure erzeugt, mit der Moeglichkeit, dass durch Backtracking eine Bindungsumgebung gefunden wird, in der der instanziierte Lispausdruck keine freien Variablen mehr enthaelt.

Die Quotierungsautomatik kann vom Toplevel aus mit dem Kommando "auto-quote-off" ausgeschaltet und mit "auto-quote-on" wieder eingeschaltet werden. Normalerweise ist sie angeschaltet.

7.5. Fehler in LISP-Evaluationen

Fuehrt die Evaluation eines LISP-Ausdrucks in Praemisse oder is-Primitiv zu einem LISP-Laufzeitfehler, so wird dieser abgefangen. Der LISPLOG-Interpreter geht dann mit einer entsprechenden Meldung auf den LISPLOG-Toplevel zurueck, ohne den LISP-Debugger aufzurufen.

8. LISPLOG-Primitive

In LISPLOG sind nur ganz wenige Primitive eingebaut, da fast alle Standard PROLOG Praedikate durch LISP-Funktionen realisiert werden koennen (Abschnitt 7.3 und 7.4). Die vorhandenen Primitive sind nachfolgend aufgefuehrt.

8.1. Arithmetische Primitive

Implementierung der arithmetischen Operationen als PROLOG Relationen.
Aufruf-Muster: (arith n_opd1 n_opd2 l_erg)
arith := addit| subit| multit| divit

8.2. not-Primitiv

Dies ermoeglicht die Negation von Konjunktionen von Praedikaten mit der Syntax (not g1 g2 ... gN). Dieses Praedikat ist wahr, wenn es keine Loesung fuer die Konjunktion der Praedikate gibt. Man beachte, dass ein not-Primitiv niemals Variablen binden kann.

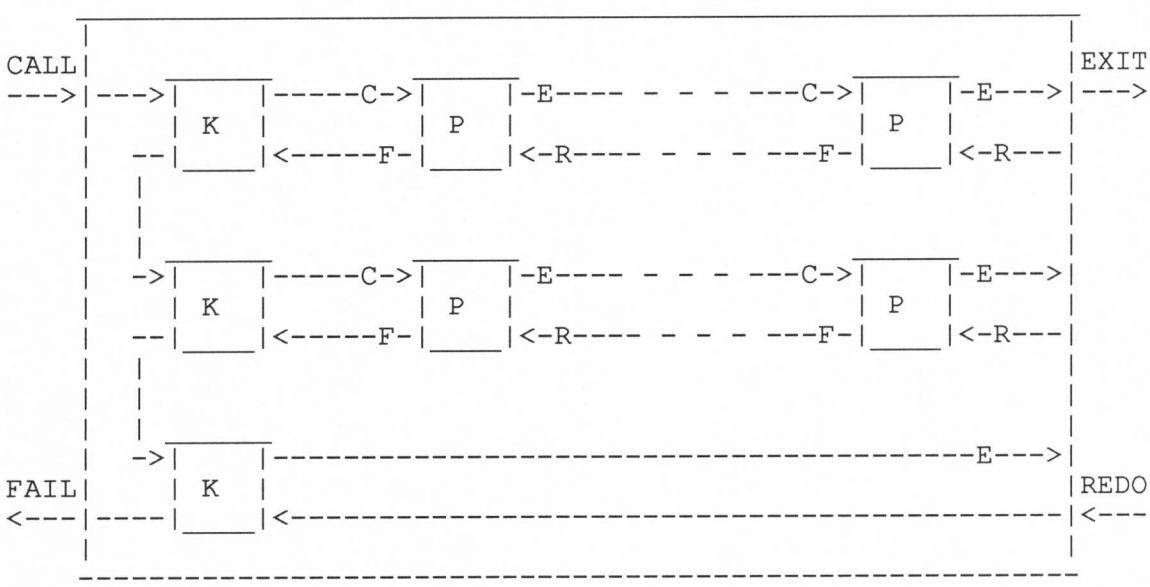
8.3. Test auf Variable

Durch die Einfuehrung von var und nonvar als Primitive wird es dem Benutzer ermoeglicht, festzustellen, ob ein Term eine freie Variable ist, oder nicht. Aufruf-Muster: (var g_term) bzw. (nonvar g_term)

9. Die Interaktionsumgebung

9.1. Hintergrund: Das Boxmodell

Das Box-Modell von Byrd (vgl. [Clocksin & Mellish 1981]) dient zur dynamischen Beschreibung der operationalen Semantik von PROLOG. Diese Semantik beschreibt den Ablauf ('Trace') der Verifizierung oder Falsifizierung einer Aussage. Alle Klauseln eines Praedikates werden hierbei zu einer sog. 'Prozedur' zusammengefasst (entsprechend den 'Moduln', die in LISPLOG.1 unter clauses abgespeichert sind). Eine Prozedur wird als Box mit zwei Eingaengen und zwei Ausgaengen dargestellt, deren Klauseln wiederum aus aehnlichen Boxes aufgebaut sind. Zum Beispiel erhaelt man fuer eine Prozedur aus zwei Regeln gefolgt von einem Fakt folgende Box:



K = Kopf einer Regel bzw. eines Faktums
P = Praemissen einer Regel
C,R = CALL- bzw. REDO-Eingaenge einer Box
E,F = EXIT- bzw. FAIL-Ausgaenge einer Box

Wird nun eine Anfrage an das System gestellt, so wird die entsprechende Prozedurbox durch den CALL-Eingang betreten. Hier wird nun noch zwischen der Box einer Konklusion bzw. einer Praemisse unterschieden. Beim Betreten einer Box durch den CALL-Eingang wird zunaechst die Box der ersten Konklusion betreten. Koennen Konklusion und Anfrage unifiziert werden, so wird die Box der ersten Praemisse durch den CALL-Eingang betreten, ansonsten die Box der naechsten Konklusion. Eine Box

wird durch den EXIT-Ausgang verlassen, wenn alle Praemissen einer Regel durch den EXIT-Ausgang verlassen wurden, bzw. wenn die Anfrage mit einem Faktum unifiziert werden konnte. Eine Box wird durch den FAIL-Ausgang verlassen, wenn keine Klausel erfolgreich angewendet werden konnte. Eine Box wird durch den REDO-Eingang betreten (Backtracking), wenn sie vorher durch den EXIT-Ausgang und die nachfolgende Box durch den FAIL-Ausgang verlassen wurde. PROLOG-Primitive und LISP-Praedikate werden nie durch den REDO-Eingang betreten.
Fuer detailliertere Informationen ueber das Box-Modell ist [Clocksin & Mellish 1981] zu empfehlen.

9.2. Dem Fehler auf der Spur: Der Tracer

Der Tracer gibt waehrend des Beweisablaufs fuer ausgewaehlte Prozeduren die einzelnen Schritte gemaess dem oben beschriebenen Boxmodell aus.

Zur Steuerung des Tracers stehen dem Benutzer zwei Kommando-Funktionen zur Verfuegung:

spy {predicate}*
: bewirkt, dass alle angegebenen Praedikate getraced werden. Wird die Funktion ohne Argumente aufgerufen, so werden alle benutzerdefinierten Praedikate sowie die Primitive is, not, var und nonvar getraced. LISP-Praedikate und andere PROLOG-Primitive muessen explizit angegeben werden.

nospy {predicate}*
: ist gerade invers zu spy. Alle angegebenen Praedikate werden fortan nicht mehr getraced. Wird diese Funktion ohne Argumente aufgerufen, so wird der Tracemodus ganz abgeschaltet.

Weitere Kommandos, die mit dem Tracer verwendet werden koennen, werden zusammen mit den Breaker-Kommandos erklaert (9.4).

9.3. Keine Bewegung! Der Breaker

Der Breaker ermoeglicht es, die Deduktion an vorher bestimmten Stellen oder durch Eingabe eines Kommandos zu beliebiger Zeit zu unterbrechen. Nach der Unterbrechung betritt das System den Break-Level, in dem eine Reihe von Kommandos zur Fehlersuche oder -Korrektur zur Verfuegung stehen.
Warnung! Durch unvorsichtige Benutzung des Breakers ist es moeglich, Beweisablaeufe zu konstruieren, die nicht reproduzierbar sind!

Zur Steuerung des Breakmodus existieren folgende Funktionen:

brk {predicate}*
setzt breakpoints an alle ports der box,

brk {(<predicate> <box-port1> ... <box-portn>)}*
setzt breakpoints nur an die spezifizierten box-ports (call, fail, redo, exit, eval oder value).

Das Setzen eines Breakpints bewirkt, dass alle angegebenen Praedikate eine Unterbrechung bewirken, sobald sie durch einen der angegebenen ports betreten werden. Wird die Funktion voellig ohne Argumente aufgerufen, so werden alle benutzerdefinierten Praedikate so behandelt, als ob fuer sie brk <praedikat> eingegeben worden waere. LISP-Praedikate und PROLOG-Primitive muessen explizit angegeben werden.

nobrk {predicate}*
ist gerade invers zu brk. Alle angegebenen Praedikate bewirken fortan keine Unterbrechung mehr im Falle eines Failure. Wird diese Funktion ohne Argumente aufgerufen, so wird der Breakmodus ganz abgeschaltet.

Neben diesen ueber die Funktionen brk und nobrk steuerbaren Breakpoints an den FAIL-Ausgaengen hat der Benutzer jederzeit die Moeglichkeit, die normale Verarbeitung durch Eingabe von <CTRL> und <C> zu unterbrechen. Er gelangt dann in den LISPLOG-Break-Modus, der ihm diverse Funktionen zur interaktiven Fehlersuche und -behebung sowie zur kontrollierten Fortsetzung der Bearbeitung zur Verfuegung stellt.

9.4. Der Break-Modus: Kommandos fuer Tracer und Breaker

Die hier beschriebene Kommandoebene wird zusammen mit Breaker und Tracer auf drei moegliche Arten erreicht:

a) durch Erreichen bzw. Ueberschreiten der Seitenlaenge im Trace (CM); Eingabe von 'b' erreicht (BM)

b) durch Eingabe von 'b' auf die Frage des Handschneiders (9.6)

c) durch Erreichen eines Breakpoints bzw. durch Eingabe von CTRL-C (BM)

Auf dieser Ebene stehen nun folgende Kommandos zur Verfuegung:

Eingabe:	Wirkung:	Verfuegbarkeit:
b	Erreicht den Breakmodus BM	CM
-p(age)	Zurueckblaettern im Trace um 1 Seite	BM CM
+p oder p	Vorwaertsblaettern im Trace um 1 Seite bzw. Fortsetzen der Berechnung um 1 Seite	BM CM
-n (n bel.)	Zurueckblaettern im Trace um n Schritte	BM CM
n (n bel.)	Vorwaertsblaettern im Trace um n Schritte bzw. Fortsetzen der Berechnung um n Schritte	BM CM
0	Ruecksprung in LISPLOG-Top-Level	BM CM
r(eturn)	Ruecksprung in LISPLOG-Top-Level	BM
c(ontinue)	Fortsetzen der unterbrochenen Berechnung	BM
v		BM
<beliebiger S-Ausdruck>	Auswerten des S-Ausdrucks; damit Moeglichkeit, Variablen etc. zu aendern (Seiteneffekte)	

Hierbei bedeuten fuer die Verfuegbarkeit der Funktionen:

BM = Diese Funktion ist im LISPLOG-Break-Modus verfuegbar.
CM = Diese Funktion ist bei Unterbrechung wegen Seitenueberlauf bzw. erreichter Schrittzahl (Control-Modus) verfuegbar.

Hinweis: Fuer einen durchlaufenden LISPLOG-Trace ist eine ausreichend grosse Schrittzahl n anzugeben.

Einige der angegebenen Kommandos koennen nur in Verbindung mit dem rekursiven Interpreter aufgerufen werden.

9.5. Der Cut-Anzeiger

In Zusammenarbeit mit Tracer und/oder Breaker liefert der Cut-Anzeiger naehere Informationen ueber die Wirkung von initialen Cuts. Wenn eine mit einem initialen Cut versehene Klausel eines Praedikates, fuer das Tracer bzw. Breaker aktiviert sind, erfolgreich (EXIT) verlassen wird, erscheint die Meldung "Cutting <N>[<M>] Clauses after clause <C>" . <N> gibt dabei die Anzahl der noch nicht angewendeten Klauseln dieses Praedikats an, <M> die Anzahl der davon mit dem Goal unifizierbaren Klauseln, <C> ist die mit dem initialen Cut versehene Klausel.

9.6. Der Handschneider

Dieses Werkzeug erlaubt es dem Benutzer, die Wirkung eines initialen Cuts zunaechst experimentell zu erproben, ehe der Cut wirklich programmiert wird.
Fuer ausgewaehlte Praedikate fragt das System im Falle des Failure einer Klausel des Praedikats, ob diese Klausel so behandelt werden soll, als waere sie mit einem initialen Cut versehen.
Eine bejahende Antwort bewirkt, dass die betreffende Klausel fuer den Rest des Beweisganges als mit einem initialen Cut versehen betrachtet wird.
Die Eingabe des Kommandos 'b' fuehrt in den Break-Modus. Hier kann dann die Frage des Handschneiders noch mit y oder n beantwortet werden.
Der Handschneider wird ueber zwei Kommandos gesteuert:

cut {predicate}*
: aktiviert den Handschneider fuer die angegebenen Praedikate. Ohne Argument aufgerufen, wird der Handschneider fuer alle benutzerdefinierten Praedikate aktiviert.

nocut {predicate}*
: ist gerade invers zu cut: Fuer die angegebenen {bzw. fuer alle benutzerdefinierten) Praedikate wird der Handschneider deaktiviert.

10. Die HELP-Funktion

Das Kommando

help {kommando}

gibt Hilfsinformationen auf dem Bildschirm aus. Wenn kein Kommando angegeben wird, so wird eine Liste aller verfuegbaren Toplevel-Kommandos auf dem Bildschirm ausgegeben.
Wird als Argument ein LISPLOG-Toplevel-Kommando angegeben, so wird zu diesem Kommando eine kurze Beschreibung ausgegeben. In der Regel handelt es sich dabei um Auszuege aus diesem Handbuch.
Ist das Argument kein LISPLOG-Toplevel-Kommando, so wird es an die FRANZ-LISP Help-Funktion weitergereicht.
Im Break-Modus des Tracers (Abschnitt 9.4) steht ebensfalls das Kommando help zur Verfuegung.

11. Indexierung der Klauseln

Um die Verarbeitung grosser Datenbasen zu beschleunigen, koennen Prozeduren, die viele Klauseln enthalten, indexiert werden. Die Indexierung wirkt auf ein Argument der Klauseln und erlaubt dem System, sehr schnell alle Klauseln zu verwerfen, die in diesem Argument mit der Anfrage nicht unifizierbar sind.
Da aber die Indexierung selbst Zeit braucht, lohnt sie sich nur fuer Prozeduren mit vielen (z. B. mehr als 10) Klauseln und fuer Argumente, in denen sich die einzelnen Klauseln voneinander unterscheiden und die in Anfragen normalerweise nicht mit uninstanziierten Variablen besetzt sind.

Beispiel:
Die Datenbasis enthalte folgende Klauseln:

```
(ass (meer atlantischer-ozean))
(ass (meer indischer-ozean))

(ass (kueste atlantischer-ozean brasilien))
(ass (kueste atlantischer-ozean uruguay))
(ass (kueste atlantischer-ozean venezuela))
(ass (kueste atlantischer-ozean zaire))
(ass (kueste atlantischer-ozean nigeria))
(ass (kueste atlantischer-ozean angola))
(ass (kueste indischer-ozean indien))
(ass (kueste indischer-ozean pakistan))
(ass (kueste indischer-ozean iran))
(ass (kueste indischer-ozean somalia))
(ass (kueste indischer-ozean kenia))
(ass (kueste indischer-ozean tansania))
```

Lauten alle Anfragen, die an diese Datenbasis gestellt werden, etwa
"An welches Meer grenzt dieses Land?",
also z. B.

```
(kueste _x indien)
```

so ist eine Indexierung des Praedikats "kueste" nach dem ersten Argument nicht sinnvoll (hier steht in der Anfrage eine uninstanziierte Variable), wohl aber eine solche nach dem zweiten Argument.

Fuer die Anfrage (meer _x) (kueste _x _y) ist hingegen eine Indexierung von "kueste" nach dem ersten Argument sinnvoll, da hier die Variable zum Zeitpunkt des Aufrufs instanziiert ist.

Eine Indexierung von "meer" ist nicht sinnvoll, da sich der Aufwand bei nur 2 Klauseln nicht lohnt.

Dem Benutzer steht zur Steuerung der Indexierung ein Kommando zur Verfuegung:

index <Argumentnummer> {<Praedikatname>}+
indexiert das (bzw. die) angegebene(n) Praedikat(e) nach dem Argument <Argumentnummer>.

Die Indexierung eines Praedikates wird vom Listing-Befehl angezeigt, indem unmittelbar vor den Klauseln des Praedikates der der Indexierung entsprechende index-Befehl ausgedruckt wird. Auch beim Abspeichern auf Dateien wird dieser Indexierungsbefehl mitgespeichert.

Defaultmaessig ist die Indexierung fuer alle Praedikate ausgeschaltet.

12. Uebersetzung von LISPLOG-Programmen nach CPROLOG

Der vorliegende in FRANZ LISP geschriebene Translator ist in der Lage, eine Teilmenge von LISPLOG (LISPLOG mit eingeschraenktem LISP-Durchgriff) nach CPROLOG zu uebersetzen. Da CPROLOG [Pereira ohne Datum] auf Edinburgh PROLOG [Clocksin & Mellish 1981/84] basiert und mit diesem eng kompatibel ist, wurde somit eine Uebergangsmoeglichkeit zum PROLOG-Standard geschaffen.

Ein PROLOG-Programm ist eine Menge von Horn-Klauseln. Der Beweis einer Konjunktion von Goals geschieht in purem PROLOG durch fortlaufende Resolution bis die leere Klausel erreicht ist. Ein LISPLOG-Programm ist eine Menge von Klauseln und Funktionsdefinitionen. In LISPLOG koennen als Goals neben PROLOG-Praedikaten und Primitiven auch LISP-Praedikate auftreten, die sogar geschachtelt sein duerfen. Diese werden nicht durch Resolution bewiesen sondern durch Evaluierung der LISP-Funktion. Bei der Uebersetzung der LISPLOG-Klauseln nach CPROLOG muss man die geschachtelten LISP-Funktionsaufrufe in Konjunktionen von PROLOG-Relationsaufrufen abflachen. Um diese neuen PROLOG-Relationsaufrufe beweisen zu koennen, muss man anschliessend aus den Definitionen der LISP-Funktionen die entsprechenden PROLOG-Prozeduren erzeugen.

12.1 Die Uebersetzung

Das Uebersetzungsprogramm wird nach dem Laden des LISPLOG-Systems aufgerufen. Eine zu uebersetzende LISPLOG-Datenbasis ist eine Datei von Klauseln und Funktionsdefinitionen. Diese Datei wird vom Programm geladen, das dann eine interne Darstellung der Datenbasis als Liste von Klauseln erzeugt. Die Klauseln werden nacheinander uebersetzt und auf eine Datei ausgegeben.

In einer zweiten Phase werden die in der Datenbasis aufgerufenen sowie die zu deren Berechnung notwendigen LISP-Funktionen uebersetzt. Eventuell in der Datei vorkommende Definitionen anderer Funktionen werden nicht beachtet.

Es gibt drei verschiedene Moeglichkeiten der Anwendung von LISP-Funktionen und somit drei verschiedene Arten der Uebersetzung:

- Fuer LISP-Funktionen, die als Praemissen von LISPLOG-Klauseln oder als Bedingung z.B. in der Funktion 'cond' vorkommen, ist nur von Interesse, ob sie den Wert 'nil' haben oder nicht. Solche Funktionen werden als Praedikate bezeichnet.

- Sind LISP-Funktionen in Aufrufe anderer LISP-Funktionen eingebettet oder werden sie als Argumente des Primitivs 'is' aufgerufen, so wird ihr Wert noch weiterverwendet. Diese

Funktionen werden nicht-praedikative Funktionen oder einfach Funktionen genannt.

- Wird eine LISP-Funktion als eines der ersten n-1 Argumente der Funktion 'progn' aufgerufen, so ist ihr Wert ohne Bedeutung. Ihr Aufruf dient der Durchfuehrung von Seiteneffekten. Ihre CPROLOG-Uebersetzung darf allerdings nie zu einem failure fuehren. Solche Funktionen heissen Pseudo-Funktionen.

Zu beachten ist, dass sich die Funktionen nicht aufgrund ihrer Definition unterscheiden. Jede LISP-Funktion kann sowohl als Praedikat, als nicht-praedikative Funktion oder als Pseudo-Funktion auftreten. Das wiederum bedeutet, dass eine einzige Funktionsdefinition moeglicherweise auf verschiedene Arten uebersetzt werden muss.

Fuer gewisse, als LISPLOG-Primitive verwendete LISP-Praedikate (numberp, atom, litatom, equal, eq, greaterp, >, >&, lessp, <, <&, =, =&), werden entsprechende CPROLOG-Praedikate erzeugt. Zu einer grossen Zahl von LISP-Funktionen (z.B. listp, null, zerop usw. sowie benutzerdefinierte Funktionen) steht in CPROLOG dagegen kein einfaches Aequivalent zur Verfuegung; wenn moeglich generiert der Uebersetzer die notwendigen CPROLOG-Klauseln. Dies gilt allerdings nur fuer nicht-compilierte pure lambda-Funktionen, die zur Zeit der Uebersetzung geladen sein muessen.

Naehere Informationen ueber den Uebersetzer stehen in [Hinkelmann 1986]. Der Uebersetzer greift auf die Programme zur Abflachung von geschachtelten LISP-Funktionen und zur Umwandlung von LISP-Funktionsdefinitionen in LISPLOG-Klauseln aus [Hinkelmann & Morgenstern 1985] zurueck.

12.2 Aufruf des Uebersetzungsprogramms

Normalerweise wird der Uebersetzer durch die Funktion translate in der Form "(translate <lisplogdatei> <cprologdatei>)" aufgerufen, wobei <lisplogdatei> die Quelldatei ist, auf der die zu uebersetzende LISPLOG-Datenbasis steht; <cprologdatei> ist die Zieldatei, auf die die herauskommende CPROLOG-Datenbasis geschrieben wird. Ist <cprologldatei> gleich nil, so erfolgt die Ausgabe ueber den Standard-Output, also im allgemeinen aufs Terminal.

Der Uebersetzer kann auch durch die Funktion translate-db in der Form "(translate-db <cprologdatei>)" aufgerufen werden, wenn die Datenbasis schon geladen ist.

(translate 's_lisplogdatei 's_cprologdatei)
WERT: t, falls die Uebersetzung durchgefuehrt wird; nil sonst.
SEITENEFFEKT: Uebersetzt die in der Datei s_lisplogdatei

stehende LISPLOG-Datenbasis in CPROLOG-Syntax und schreibt sie auf die Datei s_cprologdatei bzw. aufs Terminal, falls s_cprologdatei gleich nil.

BEACHTE: Ist zum Zeitpunkt des Funktionsaufrufes schon eine LISPLOG-Datenbasis geladen, wird ueber den Bildschirm ein Hinweis ausgedruckt. Dadurch soll verhindert werden, dass zwei unabhaengige Datenbasen versehentlich zusammengefuegt werden.

(translate-db 's_cprologdatei)

WERT: t

SEITENEFFEKT: Uebersetzt die schon geladene LISPLOG-Datenbasis in CPROLOG-Syntax und schreibt sie auf die Datei s_cprologdatei bzw. aufs Terminal, falls s_cprologdatei gleich nil.

Zur Erzeugung von CPROLOG-Klauseln aus LISP-Funktionsdefintionen ruft man die Funktion translate-fns auf.

(translate-fns 'l_flist 's_cprologfile)

WERT: t

ARGUMENTE: s_cprologfile ist die Datei, auf die die Klauseln geschrieben werden sollen. Ist s_cprologfile nil, so erfolgt die Ausgabe ueber den Standard-Output.
l_flist ist eine Liste mit den Namen der zu uebersetzenden Funktionen. Wird nur der Name der Funktion angegeben, so fragt das Programm, ob die Funktion als Praedikat, als Funktion oder als Pseudofunktion uebersetzt werden soll. Man kann dies dem Programm aber auch direkt angeben, indem man statt des Funktionsnamens eine Liste eingibt mit dem Funktionsnamen als erstem Element gefolgt von Optionen, die die Art der Uebersetzung bestimmen. Es gibt 3 Optionen:

- p: Praedikat
- f: nicht-praedikative Funktion
- ps: Pseudofunktion

Es koennen mehrere Optionen gleichzeitig angegeben werden. Bei einem einzigen Funktionsaufruf koennen die verschieden Aufrufarten gemischt vorkommen.

Beispiel: (translate-fns '(append (member p f)) nil)

member wird als Praedikat und als Funktion uebersetzt;
bei append fragt das Programm, wie uebersetzt werden soll.

Es gibt eine zweite Version des Uebersetzers, die nach CPROLOG ohne Verwendung des allgemeinen Cut uebersetzt. Dazu ruft man statt der oben angegebenen Funktionen die entsprechende der folgenden Funktionen auf:

```
(translate.nocut 's_lisplogfile 's_cprologfile)

(translate.nocut-db 's_cprologfile)

(translate.nocut-fns 'l_flist 's_cprologfile)
```

13. Literatur

[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. Short version in: B. Nebel (Ed.): Papiere zum Workshop Logisches Programmieren und Lisp. TU Berlin, FB Informatik, KIT-REPORT 31, Dec. 1985, pp. 36-53

[Clocksin & Mellish 1981/84] W. Clocksin & C. Mellish: Programming in PROLOG. Springer Verlag, Berlin Heidelberg New York, 1981. Second Edition 1984

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

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

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

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

[Kahn 1983/84] K. M. Kahn: Pure PROLOG in Pure LISP. Logic Programming Newsletter 5, Winter 83/84, pp.3-4

[Kammermeier 1984] F. Kammermeier: Franz-Lisp Mini Handbuch. Universitaet Kaiserslautern, FB Informatik, Juni 1984

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

[Pereira ohne Datum] F. Pereira (Ed.): CProlog User's Manual. University of Edinburgh, Dept. of Architecture

14. Anhang A: Listen der Kommandos

14.1 Alphabetische Liste der Toplevel-Kommandos

Kommando	Bedeutung
-	Kurzform von rex
+	Kurzform von ass
>	Wechseln des aktuellen Modul
abolish	Loeschen einer Prozedur
ass	Hinzufuegen von Klauseln zur Datenbasis
ass-a	Hinzufuegen von Klauseln am Anfang der Datenbasis
auto-quote-off	Quotierungsautomatik ausschalten
auto-quote-on	- " - einschalten
brk	Breakpoint setzen
consult	Laden von Dateien
create	Erzeugen von Modulen
cut	Handschneider einschalten
destroy	Loeschen eines Moduls oder Praedikats
destroy-all	Loeschen aller Module
do-on-modul	Ausfuehrung auf anderem Modul
edit	Editieren von Klauseln
help	Hilfsinformationen ausgeben
i	Ueberblichk ueber alle Module
index	Indexierung eines Praedikates aendern
import-check	Ueberpruefen der Importdeklarationen
l	Kurzform fuer listing
lisp	Verlassen des LISPLOG-Interpreters
listing	Anzeigen von Klauseln auf dem Bildschirm
more	Berechnung der naechsten Loesung der Anfrage
nobrk	Breakpoint loeschen
nocut	Handschneider ausschalten
nospy	Tracer ausschalten
print-modul-info	Ueberblick ueber alle Module
reconsult	Ueberschreibendes Laden von Dateien
rex	Klauseln loeschen
save	Speichern eines Moduls
save-all	Speichern aller Module
sort-modul	Modul nach Prozeduren sortieren
sort-all-modul	Alle Module nach Prozeduren sortieren
spy	Tracer einschalten
switch-to-modul	Wechseln des aktuellen Modul
tell	Speichern einer Datenbasis auf Datei

14.2 Sonstige Kommandos

Kommando	Bedeutung
ctrl-C	Unterbrechen des Beweises, Uebergang in Breakmodus
(lisplog)	startet in FRANZ LISP das LISPLOG-

System, wenn es geladen ist.