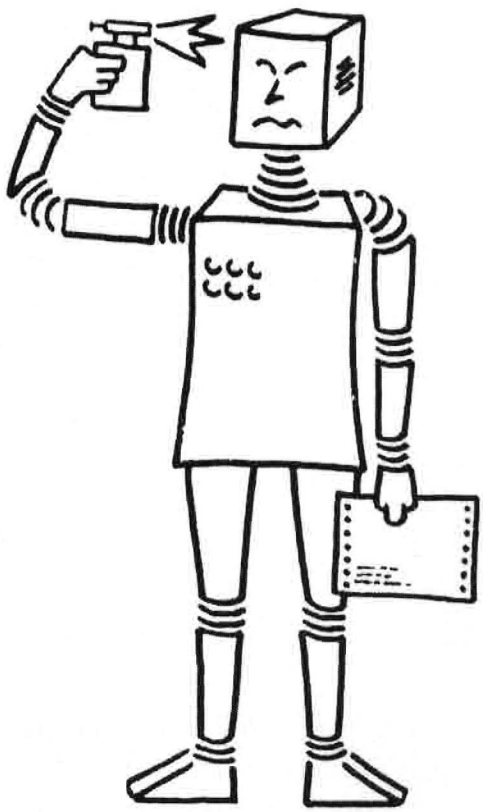
Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern 1, W Germany

SEKI-PROJEKT

SEKI MEMO



LISPLOG

Beiträge zur LISP/PROLOG-Vereinheitlichung

Michael Dahmen, Jürgen Herr
Knut Hinkelmann, Harry Morgenstern

MEMO SEKI-85-10 November 1985

L I S P L O G

BEITRAEGE ZUR LISP/PROLOG-VEREINHEITLICHUNG

Michael Dahmen
Juergen Herr
Knut Hinkelmann
Harry Morgenstern

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

uucp: !unido!uklirb!lisplog

Memo SEKI-85-10

November 1985

LISPLOG

Beiträge zur LISP/PROLOG-Vereinheitlichung

Michael Dahmen, Jürgen Herr
Knut Hinkelmann, Harry Morgenstern

MEMO SEKI-85-10 November 1985

Abstract

This paper includes three sections, discussing in detail extensions and supplementary tools for the LISPLOG system.

The first contribution by Michael Dahmen discusses a translator from CPROLOG to LISPLOG, which effects a partial transformation of PROLOG-programs. This transformation is mainly based upon pattern matching, comparing structures of the program to be transformed with given patterns and substituting structures of the target language for them.
This translator is written in PROLOG, which is suited very well for this task because of its built-in capability for unification. Therefore this translator is an example of a task, which is simply and naturally solvable in PROLOG.

The second contribution by Knut Hinkelmann and Harry Morgenstern discusses the translation of programs written in LISPLOG to standard PROLOG.
In order to translate a LISPLOG program into pure PROLOG, it is necessary to flatten nested LISP function calls into a conjunction of relationcalls and to translate the corresponding LISP functions into PROLOG clauses. The flattening of a nested LISP function call is treated in the first part.
The second part treats the translation of a LISP function into a conjunction of PROLOG clauses using the syntax of LISPLOG. The LISP functions are translated into pure LISPLOG.

These two contributions describe tools which enable developers of PROLOG programs to vary between the implementations of CPROLOG and LISPLOG without too much effort.

The third contribution by Juergen Herr treats the compilation of LISPLOG clauses into LISP functions and breadth-first search as an alternative control structure for LISPLOG. The first part discusses advantages and disadvantages of breadth-first search and introduces an implementation.
The discussion of possible constructions in clause heads and the development of concepts for translating LISPLOG clauses are the main themes in the second part. Furthermore an implementation of some of these concepts is described in the form of a hornclause compiler written in LISP, which assembles predefined function patterns.

This research was supported by the
Deutsche Forschungsgemeinschaft, Sonderforschungsbereich 314,
"Kuenstliche Intelligenz - Wissensbasierte Systeme".

Ueberblick

Dieses Memo enthaelt drei Beitraege, die Erweiterungen und ergaenzende Werkzeuge des LISPLOG-Systems behandeln.

Der erste Beitrag von Michael Dahmen behandelt einen Translator von CPROLOG nach LISPLOG, der eine partielle Transformation von PROLOG-Programmen vornimmt. Diese Transformation ist weitgehend "Pattern-basiert" d.h. die Strukturen des zu transformierenden Programms werden mit vorgegebenen Mustern verglichen und durch Strukturen der Zielsprache substituiert.
Dieser Translator wurde in PROLOG geschrieben, denn PROLOG ist wegen der eingebauten Faehigkeit zum Mustervergleich hierfuer besonders gut geeignet. Der Translator ist daher auch ein Beispiel fuer eine Aufgabe, die sich in PROLOG besonders einfach und natuerlich loesen laesst.

Der zweite Beitrag von Knut Hinkelmann und Harry Morgenstern beschaeftigt sich mit der Uebersetzung eines LISPLOG-Programmes in Standard-PROLOG.
Will man ein LISPLOG-Programm in pures PROLOG uebersetzen, muss man die geschachtelten LISP-Funktionaufrufe in Konjunktionen von Relationsaufrufen abflachen und die zugehoerigen Definitionen der LISP-Funktionen in PROLOG-Klauseln uebersetzen. Die Abflachung eines Aufrufs einer geschachtelten LISP-Funktion wird im ersten Teil behandelt.
Im zweiten Teil geht es um die Uebersetzung der Definition einer LISP-Funktion in eine Konjunktion von PROLOG-Klauseln. Fuer die PROLOG-Klauseln wird die LISPLOG-Syntax benutzt. Die LISP-Funktion wird also nach LISPLOG ohne LISP-Durchgriff uebersetzt.

Diese beiden Beitraege beschreiben Werkzeuge, die es dem Entwicker von PROLOG-Programmen ermoeglichen, mit weitgehender Systemunterstuetzung zwischen der CPROLOG- und LISPLOG-Implementierung zu wechseln.

Der dritte Beitrag von Juergen Herr beschaeftigt sich mit Breitensuche als alternativer Kontrollstruktur fuer LISPLOG, und der Kompilation von LISPLOG-Klauseln nach LISP. Im ersten Teil werden Vor- und Nachteile dieser Kontrollstruktur angesprochen, sowie eine Implementierung vorgestellt und erlauetert. Im zweiten Teil dieses Beitrags werden erlaubte Konstrukte in Klauselkoepfen besprochen und Konzepte zur Uebersetzung von LISPLOG-Klauseln entwickelt. Weiterhin wird die Implementierung eines Teils dieser Konzepte in LISP beschrieben.

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

L I S P L O G

EIN TRANSLATOR VON LISPLOG NACH CPROLOG

Michael Dahmen

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

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

Projektarbeit
(Betreuer: Harold Boley)

November 1985

Gliederung :

1. Aufgabenbeschreibung

1.1 Projektziele

Ziel des Projektes ist es, einen Translator zu erstellen der CPROLOG Programme in LISPLOG Programme uebersetzt. Eine solche Uebersetzung kann, wie weiter unten ausgefuehrt ist, nur partiell sein. Dies gilt ebenso fuer den in [6] beschriebenen Uebersetzer von LISPLOG nach CPROLOG. Da an dem LISPLOG System z.Z. noch gearbeitet wird, ist noch mit Aenderungen an Syntax und Umfang der LISPLOG Primitive zu rechnen. Aus diesem Grund ist es wichtig, das der Translator wartungsfreundlich ist. Dieses Ziel ist durch eine gute Modularisierung und einen uebersichtlichte Programmierung erreichbar.

Im Gegensatz zu einem ueblichen Compiler, dessen Ausgabe nicht weiter verarbeitet sondern ausgefuehrt wird, wird die Ausgabe des Translators oft mit Editoren weiter bearbeitet. Deshalb soll der Translator auch die Kommentare in der Eingabe mit in die Zielsprache uebersetzen und eine gut lesbar formatierte Ausgabe erzeugen (Pretty-Printing). Da der Translator vergleichsweise selten eingesetzt wird, ist eine hohe Verarbeitungsgeschwindigkeit nicht so wichtig. Daher kann der Translator durchaus in CPROLOG geschrieben werden.

1.2 Die Quellsprache

CPROLOG [1] ist eine Implementierung von Edinburgh PROLOG auf VAX Anlagen. Der CPROLOG Interpreter ist in der Programmiersprache C geschrieben, die Sprache C ist jedoch von CPROLOG aus nicht sichtbar. Der CPROLOG Programmierer kann daher nicht auf eine andere Programmiersprache durchgreifen um nicht vorhandene Funktionen zu simulieren oder um zeitkritische Funktionen in einer prozeduralen Sprache zu formulieren. Deshalb umfasst das CPROLOG System eine grosse Zahl von eingebauten Praedikaten unter anderem fuer die Bereiche :

- Ein/Ausgabe
- Arithmetik
- Datenbankverwaltung

Die Syntax von CPROLOG entspricht weitgehend der Syntax des Praedikatenkalkuels 1. Stufe. Funktionen- und Praedikataufrufe werden also in der mathematischen Notation dargestellt (d.h. Funktionsname(Argumente)) wobei zusaetzlich Infix Operatoren definiert werden koennen. Einige Infix Operatoren (z.B. fuer Arithmetik) sind bereits definiert (Built-In); ausserdem ist eine spezielle Syntax fuer Listen definiert.

1.3 Die Zielsprache

LISPLOG [3,7] ist eine Integration von PROLOG in FRANZ-LISP [2], bei der u.a. von PROLOG Praedikaten aus LISP Funktionen benutzt werden koennen. Somit koennen alle LISP Funktionen als PROLOG Primitive aufgefasst werden. Die Menge der PROLOG Primitive ist also praktisch beliebig erweiterbar. Dennoch koennen nicht alle Built-In Praedikate von CPROLOG im LISPLOG System simuliert werden, da der LISPLOG Interpreter sich von dem CPROLOG Interpreter unterscheidet. So sind beispielsweise viele Datenbankoperationen des CPROLOG Systems in LISPLOG nicht simulierbar. Die Syntax von LISPLOG weicht deutlich von der CPROLOG Syntax ab, da sie der LISP Syntax angepasst wurde. Insbesondere gibt es in LISPLOG keine Infix Operatoren und alle Praedikate werden in der LISP Notation (Cambridge Praefix) formuliert.

2. Aufbau des Translators

2.1 Die Einteilung in Phasen

Ein PROLOG Programm besteht aus einer Folge von Klauseln. Diese Klauseln sind voellig unabhaenig voneinander, denn es gibt keine Daten- oder Typdeklarationen, die alle Klauseln beeinflussen. Deshalb ist es sinnvoll die Klausel als Grundeinheit der Uebersetzung zu waehlen. Der Ablauf der Uebersetzung folgt also folgendem Grundmuster :

```
while noch-nicht-alle-Klauseln-uebersetzt do
      lese-naechste-Klausel
      transformiere-Klausel
      gebe-transformierte-Klausel-aus
```

Der Translator zerfaellt somit in mehrere Phasen (Einlesen,mehrere Transformationen,Ausgeben), und in eine Steuerung (while-Schleife), die die einzelnen Phasen nacheinander aktiviert.

2.2 Phase 1 :

Einlesen der Klauseln

Es gibt in CPROLOG zwei verschiedene Arten Programmtexte zu verarbeiten :

- zeichenweise
  Die Zeichen der Eingabedatei werden einzeln mit 'get' eingelesen und zu einer Liste von Zeichen zusammengefasst. Aus dieser Liste erzeugt ein in CPROLOG geschriebener Parser dann einen Ableitungsbaum, der durch weitere Praedikate in die gewuenschte Struktur ueberfuehrt wird.

- Term orientiert
  Von der Eingabedatei werden immer ganze Terme bzw. Klauseln eingelesen. Dazu steht in CPROLOG ein 'read' Praedikat zur Verfuegung. Dieses Praedikat benutzt den Parser des CPROLOG Systems, um aus den Zeichen der Eingabedatei (externe Repraesentation) eine interne Repraesentation zu gewinnen. Diese interne Repraesentation kann dann mit anderen CPROLOG Praedikaten weiter bearbeitet werden.

Beide Ansaetzte haben Vor- und Nachteile. Da der Parser des CPROLOG Systems in C geschrieben ist, ist er wesentlich schneller als ein in CPROLOG geschriebener Parser. Der Nachteil des eingebauten Parsers ist, dass er eine interne Repraesentation erzeugt, die fuer die Zwecke des Translators nicht ausreicht. In der internen Repraesentation fehlen die Kommentare, und die Variabeln sind nicht mehr durch ihren Namen sondern nur noch durch fortlaufende Nummern representiert.

Um dennoch den eingebauten Parser benutzten zu koennen muss der zu uebersetzende Text zunaechst vorbehandelt werden. Die Klauseln werden von der Eingabedatei (CPROLOG Darstellung) zeichenweise gelesen (mit 'get') und auf eine Hilfsdatei kopiert. Dabei werden Kommentare direkt auf die Ausgabedatei (LISPLOG Darstellung) kopiert, und auf die LISP Syntax umgestellt d.h. aus '% Kommentar in CPROLOG' wird '; Kommentar in LISP bzw. LISPLOG'. Ausserdem werden die Variabelnnamen so veraendert, dass sie nach dem Einlesen mit 'read' noch verfuegbar sind. Nachdem eine Klausel auf die Hilfsdatei kopiert wurde, wird sie mit 'read' von der Hilfsdatei eingelesen, und liegt dann in der internen Repraesentation des CPROLOG Systems vor.

Diese erste Phase des Translators ist mehr durch prozedurale als durch logische Programmierung gekennzeichnet (viele Cuts und Seiteneffekte), was durch die komplizierte Ein- / Ausgabe bedingt ist.

2.3 Phase 2 :
Umwandlung des Terms in eine Listen Repraesentation
Der Term ist mit 'read' eingelesen worden und liegt nun in der internen Repraesentation des CPROLOG Systems vor. Durch wiederholte Anwendung des Build-In Praedikats '=..' wird der Term aus der internen Repraesentation in eine Listen Repraesentation transformiert, die der LISPLOG Darstellung bereits sehr aehnlich ist.
Definition der Listen Repraesentation :

```
<Term>           ::= <Atom> |
                     '[' <Funktor> '|' <Argumentliste> ']'
<Funktor>        ::= <Atom>
<Argumentliste>  ::= '[]'    |
                     '[' <Term> '|' <Argumentliste> ']'
```

Ausserdem werden alle Atome,Funktoren und Variablen in die LISPLOG Syntax ueberfuehrt. In dieser Phase wird also beruecksichtigt dass LISPLOG Variablen mit einem '_' beginnen und dass die Bedeutung der einfachen und doppelten Hochkommata in CPROLOG und LISPLOG verschieden ist.

2.4 Phase 3 :
Transformation der Listen Repraesentation :
Die durch Phase 2 erzeugte Struktur unterscheidet sich noch in mehreren Punkten von der in LISPLOG benutzten Struktur. In Phase 3 werden die notwendigen Strukturaenderungen vorgenommen, waehrend in Phase 4 im wesentlichen die Namen der CPROLOG Primitive durch geeignete LISP oder LISPLOG Primitive ersetzt werden. Strukturaenderungen sind an folgenden Stellen notwendig :

2.4.1 Listendarstellung
In CPROLOG werden Listen durch Schachtelung von

Praedikaten dargestellt. Dafuer wird das zweistellige Praedikat '.' benutzt, mit der Bedeutung '.(H,T)' entspricht einer Liste mit Kopf 'H' und Rest 'T'. In LISPLOG wird dagegen die Listendarstellung von LISP benutzt, d.h. das Praedikat '.' besitzt keine Entsprechnung in LISPLOG. Das Zeichen '.' wird in LISP/LISPLOG fuer das CPROLOG Zeichen '|' benutzt.

Beispiel (a,b,c stehen fuer beliebige Terme) :

externe Notation	[a , b | c]
interne Notation	.(a , .(b,c))
nach Phase 2	[. , a , [. , b , c]]
nach Phase 3	[. , a , b , c]
LISPLOG	(a b . c)

2.4.2 Klauseldarstellung

In CPROLOG sind die Praemissen einer Hornklausel durch Komma oder Semikolon getrennt wobei diese als dyadische Operatoren aufgefasst werden. Praemissen und Konklusion sind durch den ':-' Operator getrennt. Nach dieser Transformation sind ':-' ';' und ',' Operatoren mit beliebige vielen Argumenten. Die Operatoren ';' und ',' werden in der naechsten Phase in die LISPLOG Praedikate 'or' und 'and' uebersetzt ('and' und 'or' sind in der Datei 'lisplog.ergaenzungen' definiert). Der ':-' Operator wird bei der Ausgabe in der letzten Phase beruecksichtigt.

Beispiele :

externe Notation	a :- b , c , d
interne Notation	:-(a , ,(b , ,(c,d)))
nach Phase 2	[:- ,a,[, , b , [, ,c,d]]]
nach Phase 3	[:- , a , b , c , d]
LISPLOG	(a b c d)

externe Notation	a ; b ; c ; d
interne Notation	;(a , ;(b , ;(c,d)))
nach Phase 2	[; ,a,[; , b , [; ,c,d]]]
nach Phase 3	[; , a , b , c , d]
LISPLOG	(or a b c d)

2.4.3 Cut - Operator

Der initiale Cut - Operator, der als einziger z.Z. von LISPLOG verarbeitet wird, wird von der CPROLOG Syntax in die LISPLOG Darstellung ueberfuehrt. Diese Transformation des initialen Cut kann durch Setzten der Option 'no_initial_cut' unterdrueckt werden. Alle nicht initialen Cuts fuehren zu einer Warnung

an den Benutzer.

Beispiel :

```
externe Notation   a :- ! , b
interne Notation   :-(a , ,( ! , b ))
nach Phase 2       [ :- , a , [ , , ! , b ] ]
nach Phase 3       [ :- , [ cut a ] , b ]
```

2.4.4 Nullstellige Praedikate
In CPROLOG sind nullstellige Praedikate als Atome dargestellt, in LISPLOG als Einerlisten. Diese Transformation wird nur fuer Praedikate als Argumente durchgefuehrt, also beispielsweise fuer das Argument von 'not'.

2.5 Phase 4 :
Uebersetzung der CPROLOG Bezeichner in LISPLOG Bezeichner
Nachdem in den vorherigen Phasen die Struktur der Klausel von der CPROLOG in die LISPLOG Form ueberfuehrt wurde, werden in dieser Phase die Namen der CPROLOG Primitive in LISP und LISPLOG Funktionen und Praedikate uebersetzt. Diese Uebersetztung ist partiell, denn fuer viele CPROLOG Primitive ist eine adaequate Uebersetztung nicht moeglich. In diesen Faellen werden Warnungen ausgegeben, die dem Benutzer anzeigen, welche Stellen von Hand transformiert werden muessen.
Nicht direkt uebersetzbar sind, beim derzeitigen Umfang von LISPLOG, die Praedikate aus den Bereichen

- Bestimmung aller Loesungen ('setof')
- Dateioperationen und Ein- / Ausgabe von Termen
- Allgemeiner Cut Operator
- Ordnung auf Termen und Sortierpraediate
- Datenbankoperationen mit Datenbankreferenzen

Um welche Praedikate es sich dabei im einzelnen handelt, kann man dem Programmtext von Phase 4 (s. Anhang 2) entnehmen.
Die Uebersetzung wird fuer alle Funktoren d.h. alle Strukturen der Form [Funktor | Argumentliste] versucht. Einzelne Atome werden nicht uebersetzt. Atome wie z.B. 'fail' oder 'repeat', die im Kontext von logischen Verknuepfungen uebersetzt werden muessen, wurden von Phase 3 in die Form von Einerlisten gebracht und sind somit von der Uebersetzung betroffen.
Die Uebersetzung erfolgt durch Anwendung des Praedikats

```
translate(CPROLOG_Praedikat,LISPLOG_Praedikat,
          Argumentliste,Kontext)
```

auf alle Funktoren der Klausel. Da einige Uebersetzungen nur in bestimmten Kontexten sinnvoll sind, wird der

Kontext mitgefuehrt. Der Kontext ist wie folgt definiert :

Kontext ist eine Liste von CPROLOG Praedikaten, bei der fuer jedes Listenelement L gilt :

Der Nachfolger N von L in der Liste ist das Praedikat, bei dem L als Argument aufgetreten ist.

Das erste Listenelement ist das Praedikat, bei dem das aktuell zu uebersetzende Praedikat als Argument aufgetreten ist.

Zu den Praediaten ist jeweils angegeben, wo das LISP/LISPLOG Aequivalent zu dem CPROLOG Praedikat definiert ist. Es bedeuten :

1 - LISP Primitiv :
Die Funktion ist ein FRANZ-LISP Primitiv.

2 - LISP Funktion :
Die Funktion ist in FRANZ-LISP geschrieben, entweder als Funktion im LISPLOG Interpreter oder in der Datei 'lisplog.ergaenzungen'.

3 - LISPLOG Primitiv :
Die Funktion ist in FRANZ-LISP geschrieben und dem LISPLOG Interpreter als LISPLOG Primitive bekanntgegeben. Siehe Dateien 'primitives' (LISPLOG Interpreter) und 'lisplog.ergaenzungen'.

4 - LISPLOG Praedikat :
Das Praedikat ist in LISPLOG definiert. Der Text befindet sich in der Datei 'lisplog.ergaenzungen'.

5 - alles andere :
Zu diesem CPROLOG Primitiv wurde bislang keine aequivalente Definition in LISPLOG erstellt. Die Praediakte die als zweites Argument von 'translate' auftreten sind entweder nicht definiert oder ihre Definition entspricht nicht der Semantik des CPROLOG Primitivs. In diesem Fall gibt der Translator im Anschluss an die uebersetzte Klausel entsprechende Warnungen aus.

2.6 Phase 5 :
Ausgabe der Klauseln in LISPLOG Notation
Die vorherigen Phasen haben die Listen Repraesentation (wie in Abschnitt 2.3 definiert erzeugt), die jetzt ausgegeben wird. Funktoren und Atome werden mit dem CPROLOG 'write' ausgegeben, Terme der Form '[Funktor | Argumentliste]' werden als '(Funktor Argumentfolge)' ausgegeben. Die beiden Funktoren ':-' und '.' werden gesondert behandelt :

':-' Dieser Funktor wird unterdrueckt, die Argumente werden ausgegeben, wobei jedoch die sie umschliessenden Klammern unterdrueckt werden; d.h.
'[:- , a1 ,a2 ... aN]' wird als
'a1 a2 ... aN' ausgegeben, und nicht als
'(a1 a2 ... aN)' .

'.' Dieser Funktor wird unterdrueckt, statt dessen wird vor der Liste ein einfaches Hochkomma ausgegeben (bewirkt in LISPLOG eine Quote). Die Ausgabe des Quote vor der Liste kann durch Setzen der Option 'no_quote' unterdrueckt werden. Ausserdem wird das letzte Argument der Liste besonders behandelt :

- ist es '[]' , so wird es unterdrueckt
- sonst wird es durch '.' (LISP Dot Operator) getrennt

In die Ausgabe ist ein Pretty-Printer integriert. Der Pretty-Printer wirkt aehnlich wie der in LISP. Mit dem Faktum 'line_length(x)' kann er auf eine bestimmte Zeilenlaenge der Ausgabedatei eingestellt werden (siehe Abschnitt 3).
Der Pretty-Printer erzeugt Ausgaben folgender Form :

```
(Funktor Argument-1          fuer Praedikate bzw.
         Argument-2          Funktionsaufrufe
         ...
         Argument-n)

'(Element-1                  fuer Listen
  Element-2
  ....
  Element-n)
```

2.7 Ergaenzungen zum LISPLOG System

Um zumindest die haeufig gebrauchten CPROLOG Primitve uebersetzen zu koennen, sind einige neue LISP Funktionen und LISPLOG Praedikate erforderlich. Diese sind in der Datei 'lisplog.ergaenzungen' definiert. Diese Datei muss in das LISPLOG System geladen werden, damit ein mit dem Translator uebersetztes Programm korrekt ablaeuft. Diese Funktionen und Praedikate veraendern den LISPLOG Interpreter nicht.

3. Benutzung des Translators

Der Translator ist in CPROLOG geschrieben und steht in der Datei '/usr/users/lisplog/translator.cpll' zur Verfuegung. Die Uebersetzung durch den Translator ist nur auf syntaktisch korrekten CPROLOG Programmen definiert. Sicherheitshalber sollte man daher zunaechst sein Programm in das CPROLOG System laden, wobei eventuell vorhanden Syntaxfehler angezeigt werden.

Der Translator ist ein zeitintensives Programm das etwa 50 CPU-Sekunden pro DIN A4 Seite benoetigt. Es ist daher ratsam, den Translator nur als sog. Hintergrund Prozess laufen zu lassen.

Der Translator benoetigt eine temporaere Datei, die er in dem Directory anlegt, in dem er aufgerufen wird. Der Name der temporaeren Datei kann durch die Option 'dateiname' beeinflusst werden (Default ist '.temp'; s. Abschnitt 4).

Beispiel :
Uebersetzt werden soll die Datei 'myprogram.cprolog' in LISPLOG Syntax.

```
% Prolog                    Aufruf des Prolog Systems

['myprogram.cprolog'].      Laden des zu uebersetzenden Programms,
                            um eventuell vorhandene Syntaxfehler
                            zu entdecken.

halt.                       zurueck zum UNIX Betriebsystem

% vi prozess                erstellen der Eingabedatei fuer den
                            Prozess
```

Inhalt der Datei 'prozess' :

```
['/usr/users/lisplog/translator.cpll']

 setzen von Optionen (s. Abschnitt 4)

 einlesen von Mustern zur Uebersetzung von
 Praedikaten in Funktionen (s. Abschnitt 5)

translate_to_lisplog('myprogram.cprolog','myprogram.lisplog').
halt.
```

In der Datei stehen Kommandos an das CPROLOG System zum Laden und Starten des Translators. Vor dem Start des Translators koennen Optionen gesetzt werden (s. Abschnitt 4) und Uebersetzungen von Praedikaten in Funktionen spezifiziert werden (s. Abschnitt 5).

Da die Uebersetzung einige Klauseln des Translators veraendert, sollte das Praedikat 'translate_to_lisplog' nur einmal aufgerufen werden. Weitere CPROLOG Programme sollten mit seperaten Prozessen uebersetzt werden.

Starten des Hintergrundprozesses :

```
% nohup cat prozess | Prolog > prozess.protokoll &
```

Wenn der Prozess beendet ist, findet man in der Datei 'myprogram.lisplog' das uebersetzte Programm in LISPLOG Syntax. CPROLOG Praedikate, die nicht korrekt uebersetzt werden konnten, sind hinter der Klausel in der sie auftraten jeweils aufgefuehrt. In der Datei 'prozess.protokoll' findet man eine Statistik ueber den Zeitbedarf der Uebersetzung von 'myprogram.cprolog' sowie eine Zusammenfassung aller nicht uebersetzbaren Praedikate von 'myprogram.cprolog'.
Damit das uebersetzte Programm in LISPLOG laeuft, muss die Datei 'lisplog.ergaenzungen' in das LISPLOG System geladen werden (s. Abschnitt 2.7).

In CPROLOG Programmtexten kommen neben Klauseln (Fakten und Regeln) manchmal auch Kommandos vor (Syntax : ':- Kommando'). Die Kommandos dienen i.a. zur Definition von Operatoren (z.B. ':-op(700,xfx,=)') oder zum Einlesen von Dateien (z.B. ':-consult(datei)'). Solche Kommandos muessen aus dem zu uebersetzenden Programm entfernt werden. Sind im Programm benutzerdefinierte Operatoren verwendet, so muessen die Definitionen (':-op(...)') vor dem Start des Translators in das CPROLOG System geladen werden. Kommandos zum Einlesen von Dateien ersetzt man am besten durch den Inhalt der entsprechenden Dateien, damit dieser mit uebersetzt wird.

4. Optionen des Translators

4.1 Zeilenlaenge der Ausgabedatei

Die Zeilenlaenge der Ausgabe ist einstellbar (Default = 79 Zeichen), wird eine andere Zeilenlaenge gewuenscht, so ist 'asserta(line_length(x)).' einzugeben, wobei 'x' die gewuenschte Zeilenlaenge ist. Diese Zeilenlaenge bezieht sich nicht auf die von der Eingabedatei kopierten Kommentare, sondern nur auf die Ausgabe der Klauseln in LISPLOG Notation.

4.2 Ausgabe der Warnungen

Die Warnungen, die bei nicht uebersetzbaren Praedikaten im Anschluss an die Klausel ausgegeben werden. koennen durch die Option 'no_warn' unterdrueckt werden. Dazu wird 'assert(option(no_warn))' eingegeben.

4.3 Quoten von Listen

Normalerweise werden alle Listen vom Translator gequotet, um eine Interpretationals Funktionsaufruf zu verhindern. Wird dies nicht gewuenscht, so kann man 'assert(option(no_quote))' eingeben.

4.4 Initialer Cut

Der initiale Cut wird vom Translator uebersetzt, waehrend alle anderen Cuts zu einer Warnung fuehren. Wenn in LISPLOG ein allgemeiner Cut implementiert wuerde, waere vermutlich die jetzige Uebersetzung des initialen Cuts unbrauchbar. Durch die Eingabe von 'assert(option(no_initial_cut))' kann die derzeitige Uebersetzung des initialen Cuts dann gesperrt werden.

4.5 Kommentare im Programm

Normalerweise werden Kommentare aus dem CPROLOG Programm in das LISPLOG Programm uebernommen. Durch Eingabe von 'assert(option(no_comment))' wird die Uebertragung der Kommentare verhindert.

4.6 Temporaere Datei

Der Translator benoetigt eine temporaere Datei, die er in dem Directory anlegt, in dem er aufgerufen wird. Der Name der temporaeren Datei ist '.temp'. Durch Eingabe der Option 'asserta(option(dateiname,<Name>))' wird '<Name>' als Dateiname verwendet. Dies ist notwendig, wenn mehrere Translator - Prozesse (quasi-) gleichzeitig aktiv sind.

5. Uebersetzung von Praedikaten in Funktionen

Beim Uebergang von CPROLOG auf LISPLOG besteht die Moeglichkeit, einige Praedikate nicht nach LISPLOG sondern direkt nach LISP zu uebersetzen, wodurch sich eine erhebliche Steigerung der Verarbeitungsgeschwindigkeit ergibt. Die Praedikate, die man in LISP Funktionen uebersetzen moechte, gibt man dem Translator bekannt. Der Translator fuehrt also nur die syntaktischen Transformationen durch, und ermittelt nicht selber, welche Praedikate durch Funktionen ersetzt werden koennen.
Dies soll an einem Beipiel erlaeutert werden. In dem zu uebersetzenden Programm seien folgende Praedikate benutzt und definiert :

```
% Argumente 1 und 2 sind instanziiert zu Listen
append([],L,L).
append([X|L1],L2,[X|L3]):-append(L1,L2,L3).

% Argumente 1 und 2 sind instanziiert zu Zahlen
geordnet(A,B,A,B):-A =< B.
geordnet(A,B,B,A):-B < A.
```

Werden diese Praedikate nur so benutzt werden, das die angegeben Variablen wirklich instanziiert sind, so sind die folgenden Definitionen zu den obigen aequivalent.

```
append(L1,L2,L3):-nonvar(L1),nonvar(L2),append_1(L1,L2,L3).

append_1([],L,L).
append_1([X|L1],L2,[X|L3]):-append(L1,L2,L3).

geordnet(A,B,C,D):-nonvar(A),nonvar(B),geordnet_1(A,B,C,D).

geordnet_1(A,B,A,B):-A =< B.
geordnet_1(A,B,B,A):-B < A.
```

In diesem Fall koennen die Praedikate durch die folgenden LISP Funktionen ersetzt werden.

```
(def append
  (lambda (L1 L2)
          (cond ( (null L1) L2 )
                ( t (cons (car L1) (append (cdr L1) L2)) ) )))

(def maximum
  (lambda (A B)
          (cond ( (greaterp A B) A )
                ( t              B ) )))

(def minimum
   (lambda (A B)
           (cond ( (lessp A B) A )
                 ( t           B ) )))
```

Damit der Translator diese Ersetzung durchfuehrt, muss man ihm folgende Muster angeben.

```
translate_praed_funk([append,L1,L2,L3],
                     [is,L3,[append,L1,L2]]).

translate_praed_funk([geordnet,A,B,C,D],
                     [and,[is,C,[minimum,A,B]],
                          [is,D,[maximum,A,B]]]).
```

Das 'and' zur Verbindung der beiden 'is' Aufrufe im zweiten Muster ist erforderlich, da hier ein Ziel ('geordnet') durch zwei andere ersetzt werden soll.

Diese CPROLOG Klauseln werden nach dem Laden des Translators, in die CPROLOG Datenbasis aufgenommen (s. Abschnitt 3). Das Praedikat 'translate_praed_funk' hat als erstes Argument das Muster des zu uebersetzenden CPROLOG Praedikates, als zweites Argument das Muster des LISPLOG/LISP Aequivalents. Wie man im Beispiel sieht, ist es nicht erforderlich, die Argumente (L1,L2,L3,A,B,C,D) der zu uebersetzenden Praedikate ebenfalls zu uebersetzen, denn auf das LISPLOG/LISP Muster ([is,L3,[append,L1,L2]]) wird der Uebersetzungsprozess erneut angewandt. Zur Spezifikation des LISPLOG/LISP Musters koennen alle in LISP,LISPLOG und CPROLOG definierten Funktionen und Praedikate benutzt werden.

6. Ein Beispiel fuer eine Uebersetzung

Das folgende CPROLOG Programm wurde mit dem Translator nach LISPLOG uebersetzt, wie im Abschnitt 3 beschrieben. Dabei waren die Optionen (siehe Abschnitt 4) 'no_quote' und 'no_comment' gesetzt. Ausserdem war ein Muster zur Uebersetzung von Praedikaten in Funktionen spezifiziert.

Eingabedatei des CPROLOG Prozesses :

```
['/usr/users/lisplog/translator.cpll'].
assert(option(no_quote)).
assert(option(no_comment)).
assert(translate_praed_funk([union,X,Y,Z],
                            [is,Z,[union,X,Y]])).
translate_to_lisplog('beispiel.cp','beispiel.ll').
halt.
```

Ausgabedatei des CPROLOG Prozesses :

```
CProlog version 1.1
| ?- /usr/users/lisplog/translator.cpll consulted 28536 bytes 13.91666

yes
| ?- |
yes
| ?- |
yes
| ?- |     |
X = _0
Y = _1
Z = _2
yes
| ?- Translator Statistik :
uebersetzt wurden 13 Klauseln
Zeitbedarf in CPU-Sekunden :
fuer Phase 1 : 30.849930
fuer Phase 2 : 9.433319
fuer Phase 3 : 2.216736
fuer Phase 4 : 3.666733
fuer Phase 5 : 9.566574
Gesamtbedarf : 55.733276

folgende Praedikate sind nicht korrekt uebersetzbar :
; !  allgemeiner Cut
; read
; retract

yes
| ?- |
[ Prolog execution halted ]
```

Zu uebersetzendes CPROLOG Programm :

```
% dies ist nur ein Beispiel, um die Funktion
% des Translators zu zeigen; die Praedikate
% sind nicht unbedingt sinnvoll

append([],X,X).
append([H1|T1],L2,[H1|T3]):-append(T1,L2,T3).

flatten([],[]).
flatten([H|L1],[H|L2]):-atom(H),H \== [],flatten(L1,L2).
flatten([H|T],Y):-flatten(H,B),flatten(T,Z),append(B,Z,Y).

last_in_list([E] , E).
last_in_list([_|T], E) :- last_in_list(T , E).

make_set([],[]).
make_set([H|T],A):-member(H,T),!,make_set(T,A).
make_set([H|T],[H|A]):-make_set(T,A).

subset(A,B):-union(A,B,B).

startup:-write('Programm gestartet'),
         assert(programm_running),
         assert(union([],[],[])),
         repeat,
         read(X),
         (X == fertig ;
          (X == done ,
           retract(programm_running)
          )).

logical([X,Y]):-X ;
                true ;
                fail ;
                call(Y) ;
                (true,fail,repeat) ;
                test([true,fail,true]).
```

Uebersetztes LISPLOG Programm :

```
(ass (append nil _X _X ) )
(ass (append (_H1 . _T1 ) _L2 (_H1 . _T3 ) ) (append _T1 _L2 _T3 ) )
(ass (flatten nil nil ) )
(ass (flatten (_H . _L1 ) (_H . _L2 ) )
     (and (not (numberp _H ) ) (atom _H ) )
     (not (equal _H nil ) )
     (flatten _L1 _L2 )
     )
(ass (flatten (_H . _T ) _Y )
     (flatten _H _B )
     (flatten _T _Z )
     (append _B _Z _Y )
     )
(ass (last-in-list (_E ) _E ) )
(ass (last-in-list (_a0 . _T ) _E ) (last-in-list _T _E ) )
(ass (make-set nil nil ) )
(ass (make-set (_H . _T ) _A )
     (member _H _T )
     (general-cut )
     (make-set _T _A )
     )
; folgende Praedikate sind nicht korrekt uebersetztbar :
; !  allgemeiner Cut
(ass (make-set (_H . _T ) (_H . _A ) ) (make-set _T _A ) )
(ass (subset _A _B ) (is _B (union _A _B ) ) )
(ass (startup )
     (patom "Programm gestartet" )
     (ass (programm-running ) )
     (ass (is nil (union nil nil ) ) )
     (repeat )
     (is _X (read ) )
     (or (equal _X fertig )
         (and (equal _X done ) (rex (programm-running ) ) )
         )
     )
; folgende Praedikate sind nicht korrekt uebersetztbar :
; read
; retract
(ass (logical (_X _Y ) )
     (or (call _X )
         t
         nil
         (call _Y )
         (and t nil (repeat ) )
         (test (true fail true ) )
         )
     )
```

7. Zur Frage der Selbstuebersetzung des Translators
   Da der Translator von CPROLOG nach LISPLOG uebersetzt, und selbst in CPROLOG geschrieben ist, erwartet man zunaechst, dass der Translator in der Lage ist, sich selbst nach LISPLOG zu uebersetzen. Diese Selbstuebersetzung ist jedoch nicht moeglich. Der Grund dafuer ist, das der Translator die folgenden CPROLOG Build-In Praedikate benutzt :

   see,seen,tell,told,write,read,setof,!

   Waehrend die Dateioperationen (see, seen, tell, told, write) verhaeltnismaessig leicht durch wenige Programmaenderungen realisiert werden koennen, sind die anderen Praedikate nur mit sehr hohem Aufwand in LISPLOG realiserbar. Fuer 'setof' und den allgemeinen Cut ('!') muss der LISPLOG Interpreter veraendert werden, fuer 'read' muss der Parser des CPROLOG Systems in LISP oder LISPLOG realisiert werden.

8. Anpassung an Aenderungen von LISPLOG

Im LISPLOG System werden noch laufend Aenderungen vorgenommen, die zum Teil auch Einfluss auf den Translator haben. Insbesondere werden neue LISP Funktionen und LISPLOG Praedikate geschrieben, wodurch mehr CPROLOG Praedikate nach LISPLOG uebersetzt werden koennen.
Damit der Translator ein CPROLOG Praedikat in eine andere LISP Funktion oder ein anderes LISPLOG Praedikat uebersetzt, muss die entsprechende Klausel in der Phase 4 des Translators geaendert werden. Das entscheidende Praedikat ist 'translate', das folgenden Aufbau hat :

```
translate(CPROLOG_Praedikat,LISPLOG_Praedikat,
          Argumente,Kontext)
```

Das Argument 'Kontext' kann i.a. ignoriert werden; mit dem Argument 'Argumente' kann spezifiziert werden, welche Stelligkeit das CPROLOG Praedikat hat. In den beiden ersten Argumenten werden die Namen der sich entsprechenden Praedikate in CPROLOG bzw. LISPLOG eingetragen.
Obiges Praedikat ist nur fuer eine eins-zu-eins Ersetzung von Praedikatnamen geeignet. Wo auch eine Aenderung der Argumente in Abhaenigkeit vom Praedikat erforderlich ist, kann das Praedikat 'translate_list' eingesetzt werden.

```
translate_list(CPROLOG_Term,LISPLOG_Term,Kontext)
```

Terme sind geschachtelte Listen; der 'Kontext' kann wiederum meist ignoriert werden.
Im Translator sind sowohl 'translate' als auch 'translate_list' oft benutzt, in Zweifelsfaellen vergleiche man daher eine neue Klausel mit den bereits vorhandenen.

Anhang 1 : Literatur

[1] Fernando Pereira :
CPROLOG User's Manual
University of Edinburgh

[2] John K. Foderaro :
The FRANZ LISP Manual
University of California , 1980

[3] Harold Boley, Franz Kammermeier et al. :
Momentaufnahmen einer LISP/PROLOG - Vereinheitlichung
Workshop Logisches Programmieren und LISP
TU Berlin 17./18. Juni 1985

[4] W. Clocksin, C. Mellish :
Programming in PROLOG
Springer Verlag , 1984

[5] S.R. Bourne :
The UNIX System
Addison - Wesley Publishing Company , 1982

[6] Knut Hinkelmann :
LISPLOG-CPROLOG-Uebersetzer
in : [3]

[7] Harold Boley, Franz Kammermeier und die LISPLOG Gruppe :
LISPLOG : Momentaufnahmen einer
LISP/PROLOG - Vereinheitlichung
MEMO SEKI-85-03, Fachbereich Informatik,
Universitaet Kaiserslautern,
August 1985 (LISPLOG - Projekt)

Anhang 2 : Programmlisting

```
% ************************ Phasensteuerung ************************
% Die fuenf Phasen  des Translators werden nacheinander aufgerufen,
% wodurch  eine Klausel eingelesen, uebersetzt und ausgegeben wird.
% Ausserdem werden  die waehrend Phase 4 ggf. gesammelten Warnungen
% ausgegeben. Mit dem Praedikat 'count(N,Ziel)' wird ein Zaehler
% aktiviert, der die Zeit zum Erreichen des Ziels 'Ziel' misst.
% Diese Zeiten werden fuer die 5 Phasen des Translators getrennt
% aufsummiert,  und im Anschluss an die Uebersetzung auf die Ausga-
% bedatei ausgegeben.

%-----------------------------------------------------------------
translate_to_lisplog(Eingabe_datei,Ausgabe_datei):-
        repeat,
          count(1,read_clause(Eingabe_datei,
                              Ausgabe_datei,Klausel)),
          translate_to_lisplog_1(Ausgabe_datei,Klausel),
        !,
        clauses(N0),
        count_sum(1,N1),
        count_sum(2,N2),
        count_sum(3,N3),
        count_sum(4,N4),
        count_sum(5,N5),
        NS is N1 + N2 + N3 + N4 + N5,
        write('Translator Statistik :'),nl,
        write('uebersetzt wurden '),
        write(N0),write(' Klauseln'),nl,
        write('Zeitbedarf in CPU-Sekunden :'),nl,
        write('fuer Phase 1 : '),write(N1),nl,
        write('fuer Phase 2 : '),write(N2),nl,
        write('fuer Phase 3 : '),write(N3),nl,
        write('fuer Phase 4 : '),write(N4),nl,
        write('fuer Phase 5 : '),write(N5),nl,
        write('Gesamtbedarf : '),write(NS),nl,
        nl,
        ausgabe_warning_lang,
        close(Eingabe_datei),
        close(Ausgabe_datei).

%-----------------------------------------------------------------
% 'translate_to_lisplog_1' wendet die Phasen 2-5 auf die aktuelle
% Klausel an.
translate_to_lisplog_1(_,'end_of_file').
translate_to_lisplog_1(Ausgabe,Klausel_1):-
             clauses(N),
             N1 is N + 1,
             retract(clauses(N)),
             assert(clauses(N1)),
             abolish(warn,1),
             count(2,transform_term(Klausel_1,Klausel_2)),
```



```
             count(3,flach_term(Klausel_2,Klausel_3)),
             count(4,translate_term(Klausel_3,Klausel_4,[])),
             tell(Ausgabe),
             count(5,write_term([ ass , Klausel_4 ],0)),nl,
             ausgabe_warning,
             !,
             fail.

%-----------------------------------------------------------------
% Sammeln und Ausgeben der Warnungen, die in Phase 4 erzeugt wurden.
ausgabe_warning:-
    option(no_warn).
ausgabe_warning:-
    setof(X,warn(X),L),
    write('; folgende Praedikate sind nicht korrekt uebersetztbar :'),
    nl,
    ausgabe_warning_1(L).
ausgabe_warning.

%-----------------------------------------------------------------
% Ausgabe einer Liste von Warnungen.
ausgabe_warning_1([]).
ausgabe_warning_1([H|T]):-
                write('; '),write(H),nl,
                ausgabe_warning_1(T).

%-----------------------------------------------------------------
% Sammeln und Ausgeben aller erzeugten Warnungen.
ausgabe_warning_lang:-
    setof(X,warn_lang(X),L),
    write('folgende Praedikate sind nicht korrekt uebersetzbar :'),
    nl,
    ausgabe_warning_1(L).
ausgabe_warning_lang.

%-----------------------------------------------------------------
% Berechnen und Merken der fuer die einzelnen Phasen benoetigten
% Zeiten.
count(Phase,Aim):-
     Tstart is cputime,
     call(Aim),
     Tende is cputime,
     count_sum(Phase,Talt),
     Tneu is Talt + Tende - Tstart,
     retract(count_sum(Phase,Talt)),
     assert(count_sum(Phase,Tneu)),
     !.

%-----------------------------------------------------------------
% Speicherpraedikate fuer die benoetigten Zeiten
count_sum(1,0).
count_sum(2,0).
count_sum(3,0).
```

```
count_sum(4,0).
count_sum(5,0).

%-----------------------------------------------------------------
% Speicherpraedikat fuer die Anzahl der uebersetzten Klauseln.
clauses(0).

%-----------------------------------------------------------------
% Standardpraedikate

% append (A,B,C) <=>
% C = APPEND (A,B)  d.h. Liste B an Liste A angehaengt
append([],X,X).
append([H1|T1],L2,[H1|T3]):-append(T1,L2,T3).

% member (E,L) <=> E ist Element der Liste L
member(X, [X|_]).
member(X, [_|Y]):- member(X,Y).

%-----------------------------------------------------------------
% next_number(S) <=>
% S ist eine String, der eine Integerzahl  repraesentiert.  Erzeugt
% bei jedem Aufruf eine weitere Zahl.
next_number(S):-
          next_number_memory(N),
          !,
          abolish(next_number_memory,1),
          N1 is N + 1,
          assert(next_number_memory(N1)),
          name(N1,S).
next_number("0"):-
          assert(next_number_memory(0)).
```



```
% ************************** Phase 1 ******************************
% Einlesen einer Klausel :
% Die  Klauseln  werden  von  der  Eingabedatei   zeichenweise
% gelesen  (mit  'get') und auf eine Hilfsdatei kopiert. Dabei
% werden Kommentare direkt auf die Ausgabedatei  kopiert,  und
% auf  die LISP Syntax umgestellt d.h. aus 'Kommentar in CPRO-
% LOG' wird '; Kommentar in LISP bzw. LISPLOG'
% Die Klauseln bestehen aus folgenden Elementen :
%
%     - Strings in " "
%       diese bleiben unveraendert und werden beim Einlesen mit
%       'read' zu Listen von Integern
%     - Atome in ' '
%       da die ' ' nach dem Einlesen nicht mehr vorhanden  sind
%       werden diese Atome durch ein Praefix " gekennzeichnet
%     - Atome (ohne '')
%        bleiben unveraendert
%     - Variabeln
%       werden in ' ' gesetzt
%     - Zahlen
%       real und integer Zahlen bleiben unveraendert
%     - Operatoren und Klammern
%       bleiben unveraendert
%
% Die oben aufgefuehrten Aenderungen  bewirken  das  Variabeln
% und  Atome mit und ohne ' ' als Atome eingelesen werden, und
% das die Variablen von  den  Atomen  in  ' '  unterscheidbar
% bleiben. Die Uebersetzung der Variabeln und Atome in LISPLOG
% Syntax  erfolgt  in  der  naechsten  Phase  (Phase  2)   des
% Translators.   Nachdem eine Klausel auf die Hilfsdatei kopi-
% ert  wurde,  wird  sie  mit  'read'  von  der  Hilfsdatei
% eingelesen,  und  liegt dann in der internen Repraesentation
% des CPROLOG Systems vor.

%-----------------------------------------------------------------
% Default Hilfsdatei
option(dateiname,'.temp').

%-----------------------------------------------------------------
% 'read_clause' liest die naechste Klausel der Eingabedatei als Term
% ein; eventuell vorhandene Kommentare werden auf die  Ausgabedatei
% kopiert; die Eingabe  und  Ausgabedateien  sind danach wieder die
% Terminaldateien.
read_clause(Eingabe,Ausgabe,Clause):-
          option(dateiname,Temp),
          see(Eingabe),
          tell(Temp),
          repeat,
             get0(X),read_char_sp(Ausgabe,Temp,X),
          told,
          see(Temp),
          read(Clause),
          seen,
```

```
        !.

%-----------------------------------------------------------------
% Aktionen nach Lesen eines Zeichens :
% Erkennen der Bestandteile der Klausel,
% des Klausel- und des Dateiendes
read_char_sp(_,_,X):-read_sym(eof,X).     % Dateiende
read_char_sp(_,_,X):-[X] = ".",put(X).    % Ende einer Klausel
read_char_sp(_,_,X):-X < 32,!,fail.       % Steuerzeichen
read_char_sp(_,_,32):-put(32),!,fail.     % Blank
read_char_sp(A,T,X):-                     % Kommentar
            [X] = "%",
            option(no_comment),
            repeat,
               get0(Y),
               read_sym(lf,Y),    % bis zum Zeilenende
            !,
            fail.
read_char_sp(A,T,X):-             % Kommentar
            [X] = "%",
            tell(A),              % wird auf die Ausgabedatei kopiert
            write(';'),           % beginnt in LISP mit ;
            repeat,
               get0(Y),           % weitere Zeichen werden kopiert
               put(Y),
               read_sym(lf,Y),    % bis zum Zeilenende
            tell(T),              % weitere Ausgabe auf Hilfsdatei
            !,
            fail.
read_char_sp(_,_,X):-             % Strings in " "
            [X] = """",
            put(X),
            repeat,
               get0(Y),           % werden unveraendert kopiert
               put(Y),
               [Y] = """",
            !,
            fail.
read_char_sp(A,T,X):-             % gequotete Atome
            [X] = "'",
            put(X),               % werden zur spaeteren Unterscheidung
            write('"'),           % von Variabeln und ungequoteten
            repeat,               % Atomen mit Praefix " versehen
               get0(Y),
               put(Y),
               [Y] = "'",
            !,
            fail.
read_char_sp(A,T,X):-             % Zahlen  Integer + Real
            char_is_ziffer(X),
            read_integer(A,T,X).  % werden unveraendert kopiert
read_char_sp(A,T,X):-             % Atome ohne ' '
            char_is_klein(X),
```



```
            read_atom(A,T,X).
read_char_sp(A,T,X):-             % Variabeln
            char_is_start_var(X),
            write(''''),          % werden in ' ' gesetzt
            read_var(A,T,X).
read_char_sp(A,T,X):-             % Operatoren
            char_is_op(X),
            read_op(A,T,X).
read_char_sp(A,T,X):-             % Klammersymbole
            char_is_klammer(X),
            put(X),
            !,
            fail.

%-----------------------------------------------------------------
read_integer(A,T,X):-             % Integerzahlen
            char_is_ziffer(X),
            !,
            put(X),
            get0(Y),
            read_integer(A,T,Y).
read_integer(A,T,X):-             % gefolgt von . ist real Zahl
            [ X ] = "." ,
            !,
            put(X),
            get0(Y),
            read_real_1(A,T,Y).
read_integer(A,T,X):-             % gefolgt von e ist real Zahl
            [ X ] = "e" ,
            !,
            put(X),
            get0(Y),
            read_real_3(A,T,Y).
read_integer(A,T,Y):-
            read_char_sp(A,T,Y).

%-----------------------------------------------------------------
read_real_1(A,T,X):-              % Real Zahl .
           not(char_is_ziffer(X)). % auf Punkt folgt keine Ziffer
read_real_1(A,T,X):-
            read_real_2(A,T,X).   % auf Punkt folgt Ziffer

read_real_2(A,T,X):-              % Real Zahl  Nachkommateil
            char_is_ziffer(X),
            !,
            put(X),
            get0(Y),
            read_real_2(A,T,Y).
read_real_2(A,T,X):-              % auf Nachkommateil folgt e
            [ X ] = "e" ,
            !,
            put(X),
            get0(Y),
```

```
          read_real_3(A,T,Y).
read_real_2(A,T,X):-              % Ende des Nachkommateils
          read_char_sp(A,T,X).

read_real_3(A,T,X):-              % Exponentialteil  Vorzeichen
          member(X,"-+"),
          !,
          put(X),
          get0(Y),
          read_real_4(A,T,Y).
read_real_3(A,T,X):-
          read_real_4(A,T,X).

read_real_4(A,T,X):-              % Expontentialteil  Ziffern
          char_is_ziffer(X),
          !,
          put(X),
          get0(Y),
          read_real_4(A,T,Y).
read_real_4(A,T,Y):-
          read_char_sp(A,T,Y).

%-----------------------------------------------------------------
read_atom(A,T,X):-
         char_is_bu_zi(X),
         !,
         put(X),
         get0(Y),
         read_atom(A,T,Y).
read_atom(A,T,Y):-
         read_char_sp(A,T,Y).

%-----------------------------------------------------------------
read_var(A,T,X):-
        char_is_bu_zi(X),
        !,
        put(X),
        get0(Y),
        read_var(A,T,Y).
read_var(A,T,Y):-
        write(''''),
        read_char_sp(A,T,Y).

%-----------------------------------------------------------------
read_op(A,T,X):-
       char_is_op(X),
       !,
       put(X),
       get0(Y),
       read_op(A,T,Y).
read_op(A,T,Y):-
       read_char_sp(A,T,Y).
```



```
%-----------------------------------------------------------------
% Symbole End_Of_File und Line_Feed
read_sym(eof,26).
read_sym(lf,10).

%-----------------------------------------------------------------
% Testpraedikate fuer Einzelzeichen und Strings

string_is_integer([]).
string_is_integer([H|T]):-char_is_ziffer(H),string_is_integer(T).

char_is_bu_zi(Char):- char_is_ziffer(Char).
char_is_bu_zi(Char):- char_is_start_var(Char).
char_is_bu_zi(Char):- char_is_klein(Char).

char_is_start_var(Char):-char_is_gross(Char).
char_is_start_var(Char):-[ Char ] = "_".

char_is_ziffer(C):-[A] = "0",[B] = "9",A =< C,C =< B.
char_is_gross(C):-[A] = "A",[B] = "Z",A =< C,C =< B.
char_is_klein(C):-[A] = "a",[B] = "z",A =< C,C =< B.
char_is_op(C):-member(C,"@^*-+=:;\|,<.>/").
char_is_klammer(C):-member(C,"!()[]").
```

```
% ************************** Phase 2 ****************************
% Umwandlung des Terms in eine Listen Repraesentation :
% Der Term ist mit 'read' eingelesen worden und liegt nun in
% der internen Repraesentation des CPROLOG Systems vor. Durch
% wiederholte Anwendung des Build-In Praedikats '=..' wird der
% Term aus der internen Repraesentation in eine Listen
% Repraesentation transformiert, die der LISPLOG Darstellung
% bereits sehr aehnlich ist.
%
% Definition der Listen Repraesentation :
%   <Term>          ::= <Atom> |
%                       '[' <Funktor> '|' <Argumentliste> ']'
%   <Funktor>       ::= <Atom>
%   <Argumentliste> ::= '[]'    |
%                       '[' <Term> '|' <Argumentliste> ']'
%
% Ausserdem werden alle Atome,Funktoren und Variablen in die
% LISPLOG Syntax ueberfuehrt, wozu die Praedikate
%     'transform_funktor'
%     'transform_name'
%     'transform_name_atom' verwendet werden.
%
% Aufgerufen wird das Praedikat
%    'transform_term(interne_Rep.,Listen_Rep.)'

%-----------------------------------------------------------------
% 'transform_term(interne_Rep.,Listen_Rep.)'
% Fuehrt die Transformation fuer einen Term durch.
transform_term(Term,Atom_LISPLOG):-
              Term =.. [ Atom_CPROLOG ],
              name(Atom_CPROLOG,Atom_CP_list),
              transform_name(Atom_CP_list,Atom_LL_list),
              name(Atom_LISPLOG,Atom_LL_list).
transform_term(Term,[Funktor_LISPLOG|Args2]):-
              Term =.. [Funktor_CPROLOG|Args1],
              name(Funktor_CPROLOG,Funktor_CP_list),
              transform_funktor(Funktor_CP_list,Funktor_LL_list),
              name(Funktor_LISPLOG,Funktor_LL_list),
              transform_term_list(Args1,Args2).

%-----------------------------------------------------------------
% 'transform_term_list(interne_Rep.,Listen_Rep.)'
% Wendet 'transform_term' auf eine Liste von Termen an.
transform_term_list([],[]).
transform_term_list([H1|T1],[H2|T2]):-
                  transform_term(H1,H2),
                  transform_term_list(T1,T2).

%-----------------------------------------------------------------
% 'transform_funktor(CPROLOG_Syntax,LISPLOG_Syntax)'
% Transformiert einen Funktor von CPROLOG in LISPLOG Syntax.
transform_funktor([A|B],[A|Z]):-   % CPROLOG Atom als Funktor
                char_is_klein(A),
```



```
                transform_name_atom(B,Z).
transform_funktor(A,A).        % CPROLOG Operatoren

%-----------------------------------------------------------------
% 'transform_name(CPROLOG_Syntax,LISPLOG_Syntax)'
% Transformiert ein beliebiges Argument von CPROLOG in LISPLOG
% Syntax.

% Anonyme Variable in CPROLOG
% Die anonyme Variable erhaelt einen noch nicht benutzten Namen,
% denn es gibt keine CPROLOG Variable, die mit 'a' beginnt.
transform_name("_",Z):-
              next_number(Y),
              append("_a",Y,Z).
% CPROLOG Variablen, die mit '_' beginnen
transform_name(A,Z):-
              append("_",B,A),
              transform_name_atom(B,Y),
              append("_n",Y,Z).
% alle anderen CPROLOG Variablen
transform_name([A|B],Z):-
              char_is_gross(A),
              transform_name_atom([A|B],Y),
              append("_",Y,Z).
% CPROLOG Atome ohne ''
transform_name([A|B],[A|Z]):-
              char_is_klein(A),
              transform_name_atom(B,Z).
% CPROLOG Atome mit '' jedoch in LISPLOG ohne ""
transform_name([A|B],Z):-
              [A] = """" ,
              transform_name_atom(B,Z).
% CPROLOG Atome mit '' in LISPLOG mit ""
transform_name([A|B],Z):-
              [A] = """" ,
              transform_name_atom_qu(B,Y),
              Z = [A|Y].
% Alles andere; Zahlen haben gleiche Syntax in LISPLOG und CPROLOG
transform_name(A,A).

%-----------------------------------------------------------------
% Ungequotete Atome und Variabeln bestehen aus Buchstaben und
% Ziffern; dem CPROLOG '_' entspricht in LISPLOG das '-'.
transform_name_atom([],[]).
transform_name_atom([H1|T1],[H2|T2]):-
                  [H1] = "_" ,
                  [H2] = "-" ,
                  transform_name_atom(T1,T2).
transform_name_atom([H|T1],[H|T2]):-
                  char_is_bu_zi(H),
                  transform_name_atom(T1,T2).

%-----------------------------------------------------------------
```

```
% Bei gequoteten Atomen ist zu beruecksichtigen, das in den CPROLOG
% Atomen das " auftreten kann, woraus ein "" werden muss, und das
% in CPROLOG Atomen ein ' auftreten kann, was durch Phase 1 zu '"
% geworden ist.
transform_name_atom_qu([],"""").
transform_name_atom_qu([DQ|T1],[DQ,DQ|T2]):-
                       [DQ] = """",
                       transform_name_atom_qu(T1,T2).
transform_name_atom_qu([SQ,DQ|T1],[SQ|T2]):-
                       [SQ] = "'",
                       [DQ] = """",
                       transform_name_atom_qu(T1,T2).
transform_name_atom_qu([H|T1],[H|T2]):-
                       transform_name_atom_qu(T1,T2).
```



```
% ************************** Phase 3 ****************************
% Transformation der Listen Repraesentation :
% Die durch Phase 2 erzeugte Struktur unterscheidet sich noch in
% mehreren Punkten von der in LISPLOG benutzten Struktur.
% Es sind dies :
%    - Nullstellige Praedikate sind noch als Atome dargestellt und
%      nicht wie in LISPLOG als Einerlisten. Diese Transformation
%      wird nur fuer Praedikate als Argumente durchgefuehrt, also
%      fuer die Argumente von
%         :-      Implikation
%         ,       logisches und
%         ;       logisches oder
%         ->      IF - THEN
%         not     Negation
%         \+      Negation
%         spy     Spy Point setzen
%         nospy   Spy Point loeschen
%         assert  hinzufuegen Klausel
%         retract entfernen Klausel
%
%      Die obige Transformation wirkt nicht auf Klauseln, die nur
%      aus einer nullstelligen Konklusion bestehen, dieser Fall
%      wird daher getrennt behandelt.
%
%    - Listen sind noch als Dotted Pairs representiert ( '.' als
%      dyadischer Operator). Nach dieser Transformation ist '.' ein
%      Operator mit beliebig vielen Argumenten.
%      Beispiel (a,b,c,d stehen fuer beliebige Terme) :
%      CPROLOG externe Notation        [ a , b | c ]
%      CPROLOG interne Notation        .(a , .(b,c))
%      Notation nach Phase 2           [ . , a , [ . , b , c ] ]
%      Notation nach Phase 3           [ . , a , b , c ]
%
%    - die Praemissen einer Hornklausel sind noch durch Komma oder
%      Semikolon getrennt wobei diese als dyadische Operatoren auf-
%      gefasst werden; ebenso sind Praemissen und Konklusion durch
%      den ':-' Operator getrennt. Nach dieser Transformation sind
%      ':-',';' und ',' Operatoren mit beliebig vielen Argumenten.
%      Beispiele :
%      CPROLOG externe Notation        a :- b , c , d
%      CPROLOG interne Notation        :-(a , ,(b , ,(c,d)) )
%      Notation nach Phase 2           [ :- ,a,[ , ,b,[ , ,c,d]]]
%      Notation nach Phase 3           [ :- , a , b , c , d ]
%
%      CPROLOG externe Notation        a , b , c , d
%      CPROLOG interne Notation        ,(a , ,(b , ,(c,d)) )
%      Notation nach Phase 2           [ , ,a,[ , ,b,[ , , c , d ]]]
%      Notation nach Phase 3           [ , , a , b , c , d ]
%
%    - der initiale Cut - Operator, der als einziger Cut z.Z. von
%      LISPLOG verarbeitet wird, wird von der CPROLOG Syntax in die
%      LISPLOG Darstellung ueberfuehrt. Diese Transformation kann
%      durch setzen der Option 'no-initial-cut' unterdrueckt werden.
```

```
%       Beispiel :
%       CPROLOG externe Notation        a :- ! , b
%       CPROLOG interne Notation        :-(a , ,( ! , b ))
%       Notation nach Phase 2           [ :- , a , [ , , ! , b ] ]
%       Notation nach Phase 3           [ :- , [ cut a ] , b ]
%
%       Beispiel ('no-initial-cut' ist gesetzt) :
%       CPROLOG externe Notation        a :- ! , b
%       CPROLOG interne Notation        :-(a , ,( ! , b ))
%       Notation nach Phase 2           [ :- , a , [ , , ! , b ] ]
%       Notation nach Phase 3           [ :- , a , ! , b ]
%
% Alle anderen Funktoren sowie Atome bleiben unbeeinflusst.

% Aufgerufen wird das Praedikat 'flach_term(alte_Rep.,neue_Rep.)'
% 'flach_term_list' ist ein Hilfspraedikat

%-----------------------------------------------------------------------
% Sonderfall : Klausel besteht nur aus einer nullstelligen
% Konklusion
flach_term(Atom,[Atom]):-atom(Atom).
flach_term(Term,Result):-flach_term_1(Term,Result).

%-----------------------------------------------------------------------
% Verarbeitung nullstelliger Praedikate
flach_term_1([ Funktor , A | B ] , Result):-
            funktor_hat_praed_arg(Funktor,1),
            atom_or_cut(A),
            flach_term_1([ Funktor , [ A ] | B ] , Result).
flach_term_1([ Funktor , A , B ] , Result):-
            funktor_hat_praed_arg(Funktor,2),
            atom_or_cut(B),
            flach_term_1([ Funktor , A , [ B ] ] , Result).
flach_term_1([ Funktor , A | B ] , Result):-
            funktor_hat_praed_arg(Funktor,1),
            atom(A),
            name(A,[F|_]),
            [F] = "_",
            flach_term_1([ Funktor , ['call',A] | B ] , Result).
flach_term_1([ Funktor , A , B ] , Result):-
            funktor_hat_praed_arg(Funktor,2),
            atom(B),
            name(B,[F|_]),
            [F] = "_",
            flach_term_1([ Funktor , A , ['call',B] ] , Result).

%-----------------------------------------------------------------------
% Verarbeitung der Listenrepraesentation
flach_term_1([ '.' , A , [ '.' | B ] ],[ '.' , X | Y ]):-
            flach_term_1(A,X),
            flach_term_1([ '.' | B ],[ '.' | Y ]).
flach_term_1([ '.' , A , B ],[ '.' , X , Y ]):-
            flach_term_1(A,X),
```



```
            flach_term_1(B,Y).

%-----------------------------------------------------------------------
% Verarbeitung der Klausel und Cut Darstellung
flach_term_1([ (':-') , A , [ ',' , '!' | B ] ],
                [ (':-') , [ cut , X ] | Y ]):-
            not(option(no_initial_cut)),
            flach_term_1(A,X),
            flach_term_1([ ',' , '!' | B ],[ ',' , [ '!' ] | Y ]).
flach_term_1([ (':-') , A , [ ',' | B ] ],[ (':-') , X | Y ]):-
            flach_term_1(A,X),
            flach_term_1([ ',' | B ],[ ',' | Y ]).
flach_term_1([ (':-') , A , [ '!' ] ],[ (':-') , [ cut , X ] ]):-
            not(option(no_initial_cut)),
            flach_term_1(A,X).

%-----------------------------------------------------------------------
% Verarbeitung von ',' (logisches und)
flach_term_1([ ',' , A , [ ',' | B ] ],[ ',' , X | Y ]):-
            flach_term_1(A,X),
            flach_term_1([ ',' | B ],[ ',' | Y ]).
flach_term_1([ ',' , A , B ],[ ',' , X , Y ]):-
            flach_term_1(A,X),
            flach_term_1(B,Y).

%-----------------------------------------------------------------------
% Verarbeitung von ';' (logisches oder)
flach_term_1([ ';' , A , [ ';' | B ] ],[ ';' , X | Y ]):-
            flach_term_1(A,X),
            flach_term_1([ ';' | B ],[ ';' | Y ]).
flach_term_1([ ';' , A , B ],[ ';' , X , Y ]):-
            flach_term_1(A,X),
            flach_term_1(B,Y).

%-----------------------------------------------------------------------
% rekursive Anwendung auf alle Argumente
flach_term_1([ Func |  Args1 ],[ Func | Args2 ]):-
            flach_term_list(Args1,Args2).

%-----------------------------------------------------------------------
% Catch - All
flach_term_1(X,X).

%-----------------------------------------------------------------------
% 'flach_term_1' auf Liste angewandt
flach_term_list([],[]).
flach_term_list([H1|T1],[H2|T2]):-
               flach_term_1(H1,H2),
               flach_term_list(T1,T2).

%-----------------------------------------------------------------------
% Funktoren, deren Argumente Praedikate sind
funktor_hat_praed_arg((':-'),_).
```

```
funktor_hat_praed_arg(('->'),_).
funktor_hat_praed_arg(('\+'),1).
funktor_hat_praed_arg(',',_).
funktor_hat_praed_arg(';',_).
funktor_hat_praed_arg(not,1).
funktor_hat_praed_arg(spy,1).
funktor_hat_praed_arg(nospy,1).
funktor_hat_praed_arg(assert,1).
funktor_hat_praed_arg(asserta,1).
funktor_hat_praed_arg(assertz,1).
funktor_hat_praed_arg(retract,1).

%-----------------------------------------------------------------------
% Argument ist ein Atom oder ein '!'
atom_or_cut('!').
atom_or_cut(A):-atom(A),name(A,[F|_]),char_is_klein(F).
```



```
% ************************** Phase 4 **************************
% Uebersetzung der CPROLOG Bezeichner in LISPLOG Bezeichner
%
% Diese Phase ist die eigentliche Uebersetzung. Uebersetzt
% werden alle Funktoren d.h. alle Strukturen der Form [ Funk-
% tor | Argumentliste ]. Einzelne Atome werden nicht ueber-
% setzt. Atome wie z.B. 'fail' oder 'repeat', die im Kontext
% von logischen Verknuepfungen uebersetzt werden muessen, wur-
% den von Phase 3 in die Form von Einerlisten gebracht und
% sind somit von der Ubersetzung betroffen.
%
% Bei einigen Funktoren aendert sich bei der Uebersetzung auch
% die Syntax der Argumente; diese Transformation wird von
% 'translate_list' durchgefuehrt.
%
% Die Uebersetzung von Praedikaten in Funktionen erfolgt durch
% das Praedikat 'translate_praed_funk'.

%-----------------------------------------------------------------------
translate_term([Funktor|Args],Result2,Kontext):-
              kontext_is_praed(Kontext),
              translate_praed_funk([Funktor|Args],Result1),
              translate_term(Result1,Result2,Kontext).
translate_term([Funktor1|Args1],Result,Kontext):-
              translate_list([Funktor1|Args1],Result,Kontext).
translate_term([Funktor1|Args1],[Funktor2|Args2],Kontext):-
              translate(Funktor1,Funktor2,Args1,Kontext),
              translate_term_list(Args1,Args2,[Funktor1|Kontext]).
translate_term(Atom,Atom,_).

%-----------------------------------------------------------------------
% wendet 'translate_term' auf jedes Element der Liste an
translate_term_list([],[],_).
translate_term_list([H1|T1],[H2|T2],Kontext):-
                   translate_term(H1,H2,Kontext),
                   translate_term_list(T1,T2,Kontext).

%-----------------------------------------------------------------------
% Translate Praedikat
% 'translate(CPROLOG_Praedikat,LISPLOG_Praedikat,
%            Argumentliste,Kontext)'
% zu den uebersetzbaren Praediaten ist jeweils angegeben,
% wo das LISP/LISPLOG Aequivalent zu dem CPROLOG Praedikat
% definiert ist. Es bedeuten :
%
%    1 - LISP Primitiv :
%        die Funktion ist ein FRANZ-LISP Primitive.
%    2 - LISP Funktion :
%        die Funktion ist in FRANZ-LISP geschrieben
%        entweder als Funktion im LISPLOG Interpreter oder
%        in der Datei 'lisplog.ergaenzungen'.
%    3 - LISPLOG Primitiv :
%        die Funktion ist in FRANZ-LISP geschreiben und
```

```
%       dem LISPLOG Interpreter als LISPLOG Primitive
%       bekanntgegeben.Siehe Datei 'lisplog.ergaenzungen'
%    4 - LISPLOG Praedikat :
%       das Praedikat ist in LISPLOG definiert; Text in
%       Datei 'lisplog.ergaenzungen'.
%    5 - alles andere :
%       zu diesem CPROLOG Primitiv wurde bislang keine
%       aequivalente Definition in LISPLOG erstellt. Die
%       Praediakte die als zweites Argument von
%       'translate' auftreten sind entweder nicht
%       definiert oder ihre Definition entspricht nicht
%       der Semantik des CPROLOG Primitiv. In diesem Fall
%       gibt der Translator im Anschluss an die ueber-
%       setzte Klausel entsprechende Warnungen aus.

%-----------------------------------------------------------------
% Sets
translate(setof,setof,[_,_,_],_):-warning(setof).
translate(bagof,bagof,[_,_,_],_):-warning(bagof).
translate('^',exist,[_,_],[setof|_]):-warning('^ in setof').

%-----------------------------------------------------------------
% Arithmetic
translate(Funktor_CP,Funktor_LL,_,Kontext):-
        translate_arith(Funktor_CP,Funktor_LL),
        kontext_is_arith(Kontext).
translate(Funktor_CP,Funktor_LL,_,Kontext):-
        translate_arith(Funktor_CP,Funktor_LL),
        not(kontext_is_arith(Kontext)),
        warning(Funktor_CP).
translate('.',car,[_],Kontext):-
        kontext_is_arith(Kontext).
translate(is,is,[_,_],_).                    % LISPLOG Primitiv
translate('=:=',equal,[_,_],_).              % LISP Primitiv
translate_list([_'=\=',A,B],[not,[equal,X,Y]],Kontext):-
             translate_term(A,X,[_'=\='|Kontext]),
             translate_term(B,Y,[_'=\='|Kontext]).
translate('<',lessp,[_,_],_).                % LISP Primitiv
translate('>',greaterp,[_,_],_).             % LISP Primitiv
translate_list([_'=<',A,B],[not,[greaterp,X,Y]],Kontext):-
             translate_term(A,X,[_'=<'|Kontext]),
             translate_term(B,Y,[_'=<'|Kontext]).
translate_list([_'>=',A,B],[not,[lessp,X,Y]],Kontext):-
             translate_term(A,X,[_'>='|Kontext]),
             translate_term(B,Y,[_'>='|Kontext]).
translate_list([length,A,B],[is,Y,[length,[quote,X]]],Kontext):-
             option(no_quote),
             translate_term(A,X,[length|Kontext]),
             translate_term(B,Y,[length|Kontext]).
translate_list([length,A,B],[is,Y,[length,X]],Kontext):-
             translate_term(A,X,[length|Kontext]),
             translate_term(B,Y,[length|Kontext]).
```



```
%-----------------------------------------------------------------
% Input / Output
% Das Dateikonzept in CPROLOG unterscheidet sich so deutlich von
% dem FRANZ-LISP Konzept (Ports), dass fuer viele CPROLOG Praedi-
% kate eine Abbildung auf LISP Funktionen nicht moeglich ist.
translate(consult,dskin,[_],_):-warning(consult).
translate(reconsult,dskin,[_],_):-warning(reconsult).
translate(see,see,[_],_):-warning(see).
translate(seeing,seeing,[_],_):-warning(seeing).
translate(seen,seen,[],_):-warning(seen).
translate(tell,tell,[_],_):-warning(tell).
translate(telling,telling,[_],_):-warning(telling).
translate(told,told,[],_):-warning(told).
translate(close,close,[],_):-warning(close).
translate(fileerrors,fileerrors,[],_):-warning(fileerrors).
translate(nofileerrors,nofileerrors,[],_):-warning(nofileerrors).
translate(exist,probef,[_],_).               % LISP Primitiv
translate(rename,rename,[_,_],_):-warning(rename).

%-----------------------------------------------------------------
% Da das 'read' Praedikat von CPROLOG in CPROLOG Syntax einliest,
% ist die Uebersetzung in das LISP Primitiv 'read' nicht korrekt,
% sofern nicht die Daten, die verarbeitet werden von CPROLOG auf
% LISP Syntax umgestellt werden.
translate_list([read,X],[is,X,[read]],_):-warning(read).

%-----------------------------------------------------------------
% Die verschiedenen Ausgabepraedikate von CPROLOG sind in LISP nicht
% verfuegbar, ausserdem geben CPROLOG Programme Terme in CPROLOG
% Syntax aus, wahrend LISPLOG Programme in LISP Syntax ausgeben.
% Alle CPROLOG Term - Ausgabepraedikate werden deshalb in das LISP
% Primitiv 'patom' uebersetzt, was allerdings nur fuer gequotete und
% ungeqoutete CPROLOG Atome und Zahlen mit Sicherheit korrekt ist.
translate(Funktor,patom,[Arg],_):-
        member(Funktor,[write,display,print]),
        atom(Arg),
        name(Arg,[H|_]),
        ( char_is_klein(H) ; [ H ] = """" ; char_is_ziffer(H) ).
translate(Funktor,patom,[_],_):-
        member(Funktor,[write,display,writeq,print]),
        warning(Funktor).

%-----------------------------------------------------------------
% Zeichenweise Ein- Ausgabe
translate_list([nl],[not,[terpr]],_).
translate_list([get0,X],[is,X,[tyi]],_).
translate_list([get,X],[is,X,['tyi-noblank']],_).
translate(skip,skip,[_],_).                  % LISP Funktion
translate(put,tyo,[_],_).                    % LISP Primitiv
translate(tab,spaces,[_],_).                 % LISP Funktion

%-----------------------------------------------------------------
% Convenience
```

```
translate(',',and,_,_).                          % LISPLOG Praedikat
translate(';',or,_,_).                           % LISPLOG Praedikat
translate_list([true],t,Kontext):-
              kontext_is_praed(Kontext).
translate('=',unify,[_,_],_).                    % LISPLOG Praedikat

%-----------------------------------------------------------------
% Extra Control
translate('!','general-cut',[],_):-warning('!  allgemeiner Cut').
translate(not,not,[_],_).                        % LISPLOG Primitive
translate('\+',not,[_],_).                       % LISPLOG Primitive
translate('->',and,[_,_],_):-warning('->').
                          % benoetigt allgemeinen Cut
translate(repeat,repeat,[],_).                   % LISPLOG Praedikat
translate_list([fail],nil,Kontext):-
              kontext_is_praed(Kontext).

%-----------------------------------------------------------------
% Meta Logical
translate(var,var,[_],_).                        % LISPLOG Primitiv
translate(nonvar,nonvar,[_],_).                  % LISPLOG Primitiv
translate_list([atom,X],[and,[not,[numberp,X]],[atom,X]],_).
translate(number,'numberp',[_],_).               % LISP Funktion
translate(integer,'fixp',[_],_).                 % LISP Funktion
translate(atomic,'atom',[_],_).                  % LISP Funktion
translate(name,name,[_,_],_).                    % LISPLOG Praedikat
translate(call,call,[_],_).                      % LISPLOG Primitiv

%-----------------------------------------------------------------
% die naechsten 3 Praedikate beziehen sich auf  das  Verhaeltnis
% zwischen interner und Listen- Repraesentation eines Praedikats
translate(functor,functor,[_,_,_],_).            % LISPLOG Praedikat
translate_list([arg,I,T1,X],[is,X,[nthelem,I,[cdr,T2]]],Kontext):-
              translate_term(T1,T2,[arg|Kontext]).
translate('=..',unify,[_,_],_).                  % LISPLOG Praedikat

%-----------------------------------------------------------------
% Comparison of Terms
% Eine Ordnung auf Termen ist in LISP nicht vorhanden; deshalb
% koennen die entsprechenden CPROLOG Primitive nicht uebersetzt
% werden. Die Praedikate '==' und '\==' koennen jedoch in  die
% LISP  Funktionen 'equal' uebersetzt werden.
translate('==',equal,[_,_],_).                   % LISP Primitiv
translate_list(['\==',A,B],[not,[equal,X,Y]],Kontext):-
              translate_term(A,X,['\=='|Kontext]),
              translate_term(B,Y,['\=='|Kontext]).
translate('@<','term-order-ls',[_,_],_):-warning('@<').
translate('@>','term-order-gr',[_,_],_):-warning('@>').
translate('@=<','term-order-le',[_,_],_):-warning('@=<').
translate('@>=','term-order-ge',[_,_],_):-warning('@>=').
translate(compare,compare,[_,_,_],_):-warning(compare).
translate(sort,sort,[_,_],_):-warning(sort).
translate(keysort,keysort,[_,_],_):-warning(keysort).
```



```
%-----------------------------------------------------------------
% Modification of the Program
% die zweistelligen 'assert' Praedikate  arbeiten mit CPROLOG
% Datenbank Referenzen, was in LISPLOG so nicht moeglich ist.
translate(assert,ass,[_],_).                     % LISP Funktion
translate(assert,ass,[_,_],_):-warning('assert/2').
translate(asserta,'ass-anfang',[_],_).           % LISP Funktion
translate(asserta,'ass-anfang',[_,_],_):-warning('asserta/2').
translate(assertz,ass,[_],_).                    % LISP Funktion
translate(assertz,ass,[_,_],_):-warning('assertz/2').
translate(clause,clause,[_,_],_):-warning(clause).
translate(clause,clause,[_,_,_],_):-warning(clause).
translate(retract,rex,[_],_):-warning(retract).
 % retract(X) <=> entferne die erste unifizierbare Klausel
 % rex(X)     <=> entferne die erste gleiche Klausel
translate(abolish,abolish,[_,_],_).              % LISP Funktion

%-----------------------------------------------------------------
% Information about the State of the Program
translate(listing,listing,[],_).
translate(listing,listing,[_],_).
translate(current_atom,current_atom,[_],_):-
        warning(current_atom).
translate(current_funktor,current_funktor,[_,_],_):-
        warning(current_funktor).
translate(current_predicate,current_predicate,[_,_],_):-
        warning(current_predicate).

%-----------------------------------------------------------------
% Internal Database
% die Praedikate arbeiten mit CPROLOG Datenbank Referenzen,
% was in LISPLOG so nicht moeglich ist.
translate(recorded,recorded,[_,_,_],_):-warning(recorded).
translate(recorda,recorda,[_,_,_],_):-warning(recorda).
translate(recordz,recordz,[_,_,_],_):-warning(recordz).
translate(erase,erase,[_],_):-warning(erase).
translate(erased,erased,[_],_):-warning(erased).
translate(instance,instance,[_,_],_):-warning(instance).

%-----------------------------------------------------------------
% Debugging
% die Trace und Debugging Praedikate sind in CPROLOG nicht
% alle implementiert insbesondere die Praedikate 'spy' und
% 'nospy' wirken nicht.
% Ihre  geplante  Wirkung (laut CPROLOG Manual) entspricht
% aber etwa der Wirkung des LISPLOG 'spy' Praedikats.
translate(debug,spy,[],_).
translate(nodebug,nospy,[],_).
translate(trace,spy,[],_).
translate(spy,spy,[_],_).
translate(nopsy,nospy,[_],_).
translate(debugging,debugging,[],_):-warning(debugging).
```

```
%-----------------------------------------------------------------
% Environmental
% Eine Uebersetzung dieser Praedikate ist kaum sinnvoll,
% da sie zur Steuerung der CPROLOG Umgebung dienen.
translate('NOLC','NOLC',[],_):-warning('NOLC').
translate('LC','LC',[],_):-warning('LC').
translate(op,op,[_,_,_],_):-warning(op).
translate(break,break,[],_):-warning(break).
translate(abort,abort,[],_):-warning(abort).
translate(save,save,[_],_):-warning(save).
translate(prompt,prompt,[_,_],_):-warning(prompt).

%-----------------------------------------------------------------
% Catch - All
translate(X,X,_,_).

%-----------------------------------------------------------------
% Merken von Warnungen
% Merken bis Ende der Uebersetzung der aktuellen Klausel.
warning(X):-assert(warn(X)),warning_lang(X).
% Merken bis Ende der Uebersetzung des Programms.
warning_lang(X):-warn_lang(X).
warning_lang(X):-assert(warn_lang(X)).

%-----------------------------------------------------------------
% Testet auf arithmetischen Kontext
kontext_is_arith([ TopFunc | _ ]):-
                member(TopFunc,[ 'is','=:=','=\=',
                                 '<','>','=<','>=']).
kontext_is_arith([ NextFunc | RestFunc ]):-
                member(NextFunc,[ '+','-','*','/','//',mod,
                                  exp,log,log10,sqrt,sin,cos,
                                  tan,asin,acos,atan,floor,'^',
                                  '/\','\/','<<','>>','\']),
                kontext_i _ ith(Rest_Func).

%-----------------------------------------------------------------
% Testet auf Praedikat Kontext
kontext_is_praed([ L ]):-
                member(L,[(':-'),',',';','->']).
kontext_is_praed([H|T]):-
                member(H,[(':-'),',',';','->']).
                kontext_is_praed(T).

%-----------------------------------------------------------------
% Uebersetzung arithmetischer Funktionen
% Die Funktionen sind alle LISP primitiv; die Funktionen,  die
% zu Warnungen fuehren  sind bislang nicht implementiert, oder
% die Implementierung entspricht nicht der CPROLOG Semantik.
translate_arith('+',plus).
translate_arith('-',diff).
translate_arith('*',times).
translate_arith('/',quotient).
```



```
translate_arith('//','int-div').
translate_arith(mod,mod).
translate_arith(exp,exp).
translate_arith(log,log).
translate_arith(log10,log10).
translate_arith(sqrt,sqrt).
translate_arith(sin,sin).
translate_arith(cos,cos).
translate_arith(tan,tan).
translate_arith(asin,asin).
translate_arith(acos,acos).
translate_arith(atan,atan):-warning(atan).
translate_arith(floor,floor):-warning(floor).
translate_arith('^',expt).
translate_arith('/\',boole1):-warning('/\').
translate_arith('\/',boole2):-warning('\/').
translate_arith('<<',boole3):-warning('<<').
translate_arith('>>',boole4):-warning('>>').
translate_arith('\',boole5):-warning('\').

%-----------------------------------------------------------------
% Uebersetung von Praedikaten in Funktionen
% 'translate_praed_funk(CPROLOG_Term,LISPLOG_Term)'
% wird bei Bedarf vom Benutzter definiert, siehe Dokumenation
% Abschnitt 5

% Beispiel :
% translate_praed_funk([append,X,Y,Z],[is,Z,[append,X,Y]]).
```

```
% ************************* Phase 5 *************************
% Ausgabe der Klauseln in LISPLOG Notation
% die Funktoren ':-' und '.' werden gesondert behandelt
%
%  ':-'  dieser Funktor wird unterdrueckt, die Argumente wer-
%        den ausgegeben, wobei jedoch die sie umschliessenden
%        Klammern unterdrueckt werden; d.h.
%        '[ :- , a1 ,a2  ... aN ]'  wird als
%        'a1 a2 ... aN'             ausgegeben, und nicht als
%        '( a1 a2 ... aN )' .
%
%  '.'   dieser Funktor wird unterdrueckt, statt dessen wird
%        vor der Liste ein Quote ' ausgegeben und das letzte
%        Argument wird besonders behandelt :
%         - ist es '[]' , so wird es unterdrueckt
%         - sonst wird es durch '.' (LISP Dot Operator)
%           getrennt
%        Die Ausgabe des Quote vor der Liste kann durch Setzen
%        der Option 'no_quote' unterdrueckt werden.
%
% In die Ausgabe ist ein Pretty-Printer integriert. Der
% Pretty-Printer wirkt aehnlich wie der in LISP. Mit dem Fak-
% tum 'line_length(x)' kann er auf eine bestimmte Zeilenlaenge
% der Ausgabedatei eingestellt werden. Um die Laenge in
% druckbaren Zeichen eines Terms festzustellen, ist ein
% Praedikat 'lang_term' vorhanden. Es ist aehnlich wie
% 'write_term' aufgebaut, und ermittelt die Laenge des Terms
% unter der Vorraussetzung, dass mit entsprechend ein und das
% Praedikat 'indent' berechent die neue Einrueckung aus alter
% Einrueckung und Lange von Funktor und Argumenten.
% Der Pretty-Printer erzeugt Ausgaben folgender Form :
%
%    (Funktor Argument1          fuer Praedikate bzw.
%           Argument2          Funktionsaufrufe
%           ...
%           Argument-n)
%
%    '(Element1                  fuer Listen
%      Element2
%      ....
%      Element-n)

%-----------------------------------------------------------------
% Pretty Printer
write_term([(':-')|Args],Indent):-
          write_term_list(Args,Indent).
write_term([('.'),Arg1|Args],Indent):-
          option(no_quote),
          write('('),
          lang_term([Arg1|Args],L),
          indent(Indent,0,L,Indent_neu),
          write_term(Arg1,Indent_neu),
          write_list(Args,Indent_neu),
```



```
          write(') '),
          write_indent(Indent).
write_term([('.'),Arg1|Args],Indent):-
          write('''('),
          lang_term([Arg1|Args],L),
          indent(Indent,1,L,Indent_neu),
          write_term(Arg1,Indent_neu),
          write_list(Args,Indent_neu),
          write(') '),
          write_indent(Indent).
write_term([Funktor|Args],Indent):-
          write('('),
          write(Funktor),
          write(' '),
          lang_term(Funktor,L1),
          lang_term(Args,L2),
          indent(Indent,L1,L2,Indent_neu),
          write_term_list(Args,Indent_neu),
          write(') '),
          write_indent(Indent).
write_term([],Indent):-
          write('nil '),
          write_indent(Indent).
write_term(Atom,Indent):-
          write(Atom),
          write(' '),
          write_indent(Indent).

%-----------------------------------------------------------------
% Wendet 'write_term' auf eine Liste von Termen an.
write_term_list([],_).
write_term_list([H|T],Indent):-
               write_term(H,Indent),
               write_term_list(T,Indent).

%-----------------------------------------------------------------
% Gibt eine Liste von Termen aus.
write_list([ [] ],_).
write_list([Last],Indent):-
          write('. '),
          write_term(Last,Indent),
          write_indent(Indent).
write_list([H|T],Indent):-
          write_term(H,Indent),
          write_list(T,Indent).

%-----------------------------------------------------------------
% Ausgabe von Zeilenvorschub und Einrueckung
write_indent(0).
write_indent(X):-nl,tab(X).

%-----------------------------------------------------------------
% Berechnet ob, und wenn ja, wieviel eingerueckt werden muss.
```

```
indent(Indent_old,L_funk,L_args,Indent_neu):-
      line_length(LL),
      L_args > LL - Indent_old - L_funk - 1,
      Indent_neu is Indent_old + L_funk + 1.
indent(_,_,_,0).

%---------------------------------------------------------------
% Berechnet die Anzahl der Zeichen beim Ausdruck eines  Terms,
% wobei angenommen wird, dass mit 'write_term' ausgegeben wird.
lang_term([(':-')|Args],L):-
          lang_term_list(Args,L).
lang_term(['.',Arg1|Args],L):-
          option(no_quote),
          lang_term(Arg1,L1),
          lang_list(Args,L2),
          L is L1 + L2 + 3.
lang_term(['.',Arg1|Args],L):-
          lang_term(Arg1,L1),
          lang_list(Args,L2),
          L is L1 + L2 + 4.
lang_term([Funktor|Args],L):-
          lang_term(Funktor,L1),
          lang_term_list(Args,L2),
          L is L1 + L2 + 4.
lang_term([],4).
lang_term(Atom,L):-
         name(Atom,Liste),
         length(Liste,L1),
         L is L1 + 1.

%---------------------------------------------------------------
% Wendet 'lang_term' auf eine Liste von Termen an.
lang_term_list([],0).
lang_term_list([H|T],L):-
               lang_term(H,L1),
               lang_term_list(T,L2),
               L is L1 + L2.

%---------------------------------------------------------------
% Anzahl der Zeichen des Ausdrucks einer Liste
lang_list([ [] ],0).
lang_list([Last],L):-
          lang_term(Last,L1),
          L is L1 + 2.
lang_list([H|T],L):-
          lang_term(H,L1),
          lang_list(T,L2),
          L is L1 + L2.

%---------------------------------------------------------------
% Default Zeilenlaenge
line_length(79).
```



```
;******************** LISP / LISPLOG Ergaenzungen *****************
; Diese Datei enthaelt LISP Funktionen und LISPLOG Praedikate, fuer
; einige der CPROLOG Praedikate, die in LISPLOG oder LISP noch kein
; Aequivalent besitzen. Die Datei kann in das FRANZ-LISP System mit
; 'dskin' eingelesen  werden, n a c h d e m  das LISPLOG System ge-
; laden wurde.

;---------------------------------------------------------------
; CPROLOG : get
(def tyi-noblank
 (lambda ()
         (cond ( (lessp (tyipeek) 33) (tyi) (tyi-noblank) )
               ( t (tyi) )
         )
))

;---------------------------------------------------------------
; CPROLOG : skip
(def skip
 (lambda (x)
     (cond ( (lessp x 1) )
           ( t  (tyi) (skip (subl x)) )
     )
))

;---------------------------------------------------------------
; CPROLOG : tab
(def spaces
 (lambda (x)
     (cond ( (lessp x 1) )
           ( t  (tyo 32) (spaces (subl x)) )
     )
))

;---------------------------------------------------------------
; CPROLOG : , ( logisches und zwischen Praedikaten )
(ass (and))
(ass (and _x . _y) _x (and . _y))

;---------------------------------------------------------------
; CPROLOG : ; ( logisches oder zwischen Praedikaten )
(ass (or _x . _y) _x)
(ass (or _x . _y) (or . _y))

;---------------------------------------------------------------
; CPROLOG : =
(ass (unify _x _x))

;---------------------------------------------------------------
; CPROLOG : !
(ass (general-cut))

;---------------------------------------------------------------
```

```
; CPROLOG : repeat
(ass (repeat))
(ass (repeat) (repeat))

;-----------------------------------------------------------------
; CPROLOG : name
(ass (name _x _y) (var-p _x) (non-var-p _y)
                  (is _x (implode _y)) )
(ass (name _x _y) (atom _x) (var-p _y)
                  (is _y (exploden _x)) )
(ass (name _x _y) (atom _x) (non-var-p _y)
                  (equal _x (implode _y)) )

;-----------------------------------------------------------------
; CPROLOG : call
(ass (call _x) (symbolp _x) (call (_x)))
(ass (call _x) (not (symbolp _x)) _x)

;-----------------------------------------------------------------
; CPROLOG : functor
(ass (functor _t _f _n) (non-var-p _t)
                        (is _f (car _t))
                        (is _n (subl (length _t))))
(ass (functor _t _f _n) (integer _n)
                        (atom _f)
                        (nvars-list _n _v)
                        (unify _t (_f . _v)))
(ass (nvars-list 0 nil))
(ass (nvars-list _n (_a . _rest)) (is _nl (subl _n))
                                  (nvars-list _nl _rest))

;-----------------------------------------------------------------
; CPROLOG : asserta
(def ass-anfang-l
  (lambda (assertion)
    (let ((pred-l (first (s-conclusion assertion))))
         (putprop pred-l
                  (cons assertion (get pred-l 'clauses))
                  'clauses)
         (cond ((not (member pred-l predicates))
                (setq redo-list (append (list (list pred-l))
                                        redo-list))))
         (setq predicates (union predicates (list pred-l))))))
(def ass-anfang
  (nlambda (assertion)
    (ass-anfang-l assertion)))

;-----------------------------------------------------------------
; CPROLOG //
(def int-div
 (lambda (x y)
         (car (Divide x y))
))
```



```
;-----------------------------------------------------------------
; CPROLOG log10
(def log10
 (lambda (x)
         (quotient (log x) (log 10))
))

;-----------------------------------------------------------------
; CPROLOG tan
(def tan
 (lambda (x)
         (quotient (sin x) (cos x))
))
```

EIN VERFAHREN ZUR TRANSFORMATION VON LISP-FUNKTIONEN IN PROLOG-RELATIONEN

Knut Hinkelmann, Harry Morgenstern

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

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

Projektarbeit
(Betreuer: Harold Boley)

November 1985

Aufgabenstellung

Die Sprache LISPLOG [Boley & Kammermeier et al. 1985] ist eine Vereinheitlichung von LISP und PROLOG. Sie ist in FRANZ LISP implementiert.

Ein LISPLOG-Programm ist eine Menge von Horn-Klauseln. Der Beweis einer Konjunktion von Goals geschieht in purem PROLOG durch fortlaufende Resolution bis die leere Konjunktion erreicht ist. In LISPLOG koennen als Goals neben PROLOG-Praedikaten (Relationen) aber auch LISP-Praedikate auftreten, die sogar geschachtelt sein duerfen. Diese werden nicht durch Resolution bewiesen sondern durch Evaluierung der LISP-Funktion.

Will man ein LISPLOG-Programm in pures PROLOG uebersetzen, muss man die geschachtelten LISP-Funktionen abflachen und die zugehoerigen Definitionen der LISP-Funktionen in PROLOG-Klauseln uebersetzen. Die Abflachung eines Aufrufs einer geschachtelten LISP-Funktion wird im ersten Teil behandelt.

Im zweiten Teil geht es um die Uebersetzung der Definition einer LISP-Funktion in eine Konjunktion von PROLOG-Klauseln. Fuer die PROLOG-Klauseln wird die LISPLOG-Syntax benutzt. Die LISP-Funktion wird also nach LISPLOG mit eingeschraenktem LISP-Durchgriff uebersetzt (erlaubt sind LISP-Funktionen fuer Arithmetik). Mit dem schon vorhandenen LISPLOG-CPROLOG-Translator [Hinkelmann 85] koennen diese Klauseln dann direkt nach CPROLOG uebersetzt werden.

Das hier vorgestellte System ist vollstaendig in FRANZ LISP implementiert.

Inhalt:

Teil I

Optimierendes Abflachen von LISP-Funktionsschachtelungen

Harry Morgenstern

1. LISP-Funktionsaufrufe

Ein (geschachtelter) LISP-Funktionsaufruf ist eine Liste der Form

(f a1 a2 ... an)

wobei f ein LISP-Funktionssymbol ist und die Argumente ai (i=1,...,n) LISP-Ausdruecke sind (von denen mindestens eines wieder ein Funktionsaufruf sein muss).

Beispiele: a. ungeschachtelt:

(plus 1 2)

b. geschachtelt:

(plus (times (+ 2 3)) (- 8 4))

c. geschachtelt:

(reverse (list (plus 2 1) a b))

2. Abflachen von geschachtelten LISP-Funktionsaufrufen

Unter dem Abflachen einer LISP-Funktionsaufrufen kann man sich das Herausziehen von Funktionsaufrufen aus Funktionsaufrufen vorstellen. Jeder n-stellige Funktionsaufruf wird zu einem n+1-stelligen Relationsaufruf, wobei das n+1-te Argument eine Ergebnisvariable ist, die spaeter den Funktionswert annimmt. Die "herausgezogenen" Funktionsaufrufe werden durch die Ergebnisvariable ersetzt. Zusaetzlich wird jedem LISP-Funktionssymbol in der abgeflachten Form ein '*r' angehaengt. Nach dem Abflachen erhaelt man eine Konjunktion von Relationen, hier als Liste dargestellt,

die keine eingebetteten Funktionsaufrufe mehr enthalten.
Zunaechst soll der Einfachheit wegen nur das Abflachen von nicht-arithmetischen LISP-Funktionsaufrufen betrachtet werden.

2.1 Abflachen nicht-arithmetischer LISP-Funktionsaufrufe

Wie das Abflachen von LISP-Funktionsaufrufen vor sich geht, soll jetzt an einigen Beispielen gezeigt werden.

Ein einfaches Beispiel ist folgendes:

Bsp.2.1.1: (f a1 ... an)

Hier sollen saemtliche ai keine Funktionsaufrufe sein. Dann koennen alle ai unveraendert in die abgeflachte Form uebernommen werden. Es muss nur noch eine neue Ergebnisvariable hinzugefuegt werden.
Demnach erhaelt man nach dem Abflachen von obigem Beispiel folgende einelementige Konjunktion:

((f*r a1 ... an res))

Die neue Ergebnisvariable ist res.

Im folgenden Beispiel sind einige Argumente Funktionsaufrufe.

Bsp.2.1.2:

(f1 (f2 b1 b2) a2 (f3 c1 c2) a4)

Man geht hier von links nach rechts vor. Der Funktionsaufruf f2 muss zunaechst "herausgezogen" und durch die Ergebnisvariable in f1 ersetzt werden. Dasselbe muss dann mit f3 geschehen.

Man erhaelt also:

```
((f2*r b1 b2 res1) (f3*r c1 c2 res2)
                   (f1*r res1 a2 res2 a4 res3))
```

res3 ist hier die Ergebnisvariable von f1.

Wenn die Funktionsaufrufe tiefer geschachtelt sind, geht man auf aehnliche Weise von links nach rechts und von innen nach aussen vor.

Bsp.2.1.3:

```
(f1 (f2 a1 (f3 a2 a3)) a4 (f4 a5 (f5 a6 a7)))
```

Dies ergibt:

```
((f3*r a2 a3 res1) (f2*r a1 res1 res2) (f5*r a6 a7 res3)
 (f4*r a5 res3 res4) (f1*r res2 a4 res4 res5))
```

Auch hier wurden wieder Argumente, welche Funktionsaufrufe waren, durch ihre Ergebnisvariablen ersetzt.

Aus diesen Betrachtungen kommt man zu folgendem Algorithmus zum Abflachen von geschachtelten Funktionsaufrufen. Die Funktion behandelt zuerst das erste Element des Funktionsaufrufs und ruft sich dann rekursiv mit dem Rest auf.

Algorithmus 2.1:

1. Ist die Liste leer, so erzeuge eine neue Ergebnisvariable.

Bsp.2.1.4:

```
() --> ((resnew))
```

2. Das erste Element der Liste ist ein Funktionssymbol oder ein Argument, welches kein Funktionsaufruf ist.

   a) Flache den Rest der Liste ab.

   b) Fuege das erste Element in das letzte Element der abgeflachten Liste ein (das letzte Element einer abgeflachten Liste ist immer die gerade betrachtete Top-Level-Funktion).

Bsp.2.1.5:

```
    (f1 (f2 a1 a2) a3)

--> ((f2*r a1 a2 res1) (res1 a3 res2))
a
    (* Rest der Liste abgeflacht *)

--> ((f2*r a1 a2 res1) (f1*r res1 a3 res2))
b
    (* f1 (= erstes Element) eingefuegt *)
```

3. Das erste Element ist ein Funktionsaufruf.

   a) Flache das erste Element und den Rest der Liste ab.

   b) Haenge die beiden Listen zusammen und ersetze den Funktionsaufruf durch die Ergebnisvariable.

Bsp.2.1.6:

```
        ((f1 a1 a2) (f2 b1 b2) a3)

    --> ((f1*r a1 a2 res1)) und ((f2*r b1 b2 res2)
    a                            (res2 a3 res3))

        (* Abgeflachter Kopf und
           abgeflachter Rest der Liste *)

    --> ((f1*r a1 a2 res1) (f2*r b1 b2 res2)
    b                      (res1 res2 a3 res3))

        (* Zusammenhaengen und einfuegen von res1 *)
        (* res3 ist Ergebnisvariable der Funktion,
           in der f1 und f2 aufgerufen wurden *)
```

2.2 Abflachen von arithmetischen LISP-Funktionsaufrufen

Sind in den geschachtelten LISP-Funktionsaufrufen auch arithmetische Funktionsaufrufe enthalten, so brauchen diese nicht abgeflacht zu werden, da geschachtelte arithmetische Funktionsaufrufe als rechte Seite des is-Operators dem PROLOG-Standard entsprechen.
Innerhalb von arithmetischen Funktionsaufrufen koennen jedoch wieder nicht-arithmetische Funktionsaufrufe enthalten sein. Dies macht das Abflachen etwas komplizierter. Im folgenden soll das Abflachen arithmetischer Funktionsaufrufe kurz "arithmetisches Abflachen" genannt werden.

Bsp.2.2.1:

```
(plus 1 (plus (length (a b c)) 2))
```

Man muss hier den nicht-arithmetischen Funktionsaufruf aus den arithmetischen Funktionsaufrufen herausziehen und ihn durch seine Ergebnisvariable ersetzen, jedoch die arithmetischen Funktionsaufrufe unveraendert geschachtelt lassen. Das Resultat des arithmetischen Funktionsaufrufs wird einer Variablen durch die Operation 'is' zugewiesen.

Man kommt zu folgendem Ergebnis:

```
((length*r (a b c) res1) (is res2 (plus 1 (plus res1 2))))
```

Dies fuehrt zu folgenden Erweiterungen von Algorithmus 2.1 :

In Punkt 2 und Punkt 3 muss noch zusaetzlich abgefragt werden, ob das Funktionssymbol arithmetisch ist. Ist dies der Fall, so wird *die Liste in Punkt 2 arithmetisch abgeflacht, bzw. in Punkt 3:*

a) der Kopf der Liste arithmetisch abgeflacht und der Rest der Liste abgeflacht
b) die beiden Listen zusammengehaengt, sowie der Funktionsaufruf durch die Ergebnisvariable ersetzt.

Bsp.2.2.2: (* zu Punkt 2 *)

```
(+ 3 (* 4 2)) --> ((is res1 (+ 3 (* 4 2))))
```

Bsp.2.2.3: (* zu Punkt 3 *)

```
((+ 3 (* 4 2)) (reverse (a b)))

--> ((is res1 (+ 3 (* 4 2)))) und ((reverse*r (a b) res2)
a                                  (res2 res3))

    (* Arithmetisch abgeflachter Kopf
       und abgeflachter Rest *)

--> ((is res1 (+ 3 (* 4 2))) (reverse*r (a b) res2)
b                             (res1 res2 res3))

    (* Zusammenhaengen und Einfuegen von res1 *)
```

Bevor wir zum Algorithmus zum Abflachen von arithmetischen Funktionsaufrufen kommen, noch einige Erklaerungen:

Vorbemerkung:

Sind in dem arithmetisch abzuflachenden Funktionsaufruf auch nicht-arithmetische Funktionsaufrufe enthalten, so befinden sich diese nachher in einer Liste, welche als erstes Element in der Ergebnisliste steht. Waren also nur arithmetische Funktionsaufrufe in der Schachtelung enthalten, so hat die Ergebnisliste die Laenge 1, sonst die Laenge 2.

In Algorithmus 2.1 ruft man, falls man einen arithmetischen Funktionsaufruf erkannt hat, nicht direkt die Funktion zum arithmetischen Abflachen auf, sondern man schaltet eine Funktion davor, die

a) mit Hilfe der Funktion zum arithmetischen Abflachen die Liste abflacht und

b) den is-Operator und die Ergebnisvariable in die abgeflachte Liste einfuegt.
Beim Einfuegen muss man darauf achten, ob nicht-arithmetische Funktionsaufrufe in der geschachtelten Funktion enthalten waren. Dies erkennt man an der Laenge der Liste (s. Vorbemerkung).

Bsp.2.2.4.a:

```
        (+ (* 2 3) 6)

    --> ((+ (* 2 3) 6))
    a
        (* Arithmetisches Abflachen *)

    --> ((is res (+ (* 2 3) 6)))
    b
        (* Einfuegen des is-Operators
           und der Ergebnisvariablen *)
```

Bsp.2.2.4.b:

```
        (+ (length (a b)) (* 2 3))

    --> (((length*r (a b) res1)) (+ res1 (* 2 3)))
    a
        (* Arithmetisches Abflachen *)

    --> ((length*r (a b) res1) (is res (+ res1 (* 2 3))))
    b
        (* Einfuegen des is-Operators
           und der Ergebnisvariablen *)
```

Nun die in den vorigen Beispielen schon benutzte Funktion zum arithmetischen Abflachen.

Algorithmus 2.2:

1. Die Eingabeliste ist leer. Dann erzeuge die leere Liste.

Bsp.2.2.5:

```
    () --> (())
```

2. Das erste Element ist ein arithmetischer Funktionsaufruf.

   a) Flache den Kopf und den Rest arithmetisch ab.

   b) Haenge die beiden Listen zusammen.
      Hier koennen beim Zusammenhaengen vier verschiedene Faelle auftreten, was wieder davon abhaengt, ob nicht-arithmetische Funktionsaufrufe im Kopf bzw. im Rest enthalten waren.

Bsp.2.2.6.a:

```
        ((+ 2 1) (* 3 (- 6 2)))
```

```
    --> ((+ 2 1)) und ((* 3 (- 6 2)))
    a
        (* Arithmetisch abgeflachter Kopf und
           arithmetisch abgeflachter Rest *)

    --> (((+ 2 1) (* 3 (- 6 2))))
    b
        (* Zusammenhaengen *)
```

Bsp.2.2.6.b:

```
        ((+ 2 1) (* 3 (length (a b))))

    --> ((+ 2 1)) und (((length*r (a b) res)) (* 3 res))
    a
        (* Arithmetisch abgeflachter Kopf und
           arithmetisch abgeflachter Rest *)

    --> (((length*r (a b) res)) ((+ 2 1) (* 3 res)))
    b
        (* Zusammenhaengen *)
```

Bsp.2.2.6.c:

```
        ((+ 2 (length (a b))) (* 3 2) 3)

    --> (((length*r (a b) res)) ((+ 2 res)) und (((* 3 2) 3))
    a
        (* Arithmetisch abgeflachter Kopf und
           arithmetisch abgeflachter Rest *)

    --> (((length*r (a b) res)) ((+ 2 res) (* 3 2) 3))
    b
        (* Zusammenhaengen *)
```

Bsp.2.2.6.d:

```
        ((+ 3 (length (a))) (* (length (a b)) 2))

    --> (((length*r (a) res1)) ((+ 3 res1)))
    a    und (((length*r (a b) res2)) ((* res2 2)))

        (* Arithmetisch abgeflachter Kopf und
           arithmetisch abgeflachter Rest *)

    --> (((length*r (a) res1) (length*r (a b) res2)) ((+ 3 res1)
    b                                                 (* res2 2)))

        (* Zusammenhaengen *)
```

3. Das erste Element ist ein nicht-arithmetischer Funktionsaufruf.

a) Flache das erste Element mit Algorithmus 2.1 ab und flache den Rest arithmetisch ab.

b) Haenge die beiden Listen zusammen und fuege die Ergebnisvariable ein.
Hier koennen beim Zusammenhaengen zwei Faelle auftreten.

Bsp.2.2.7.a:

```
        ((length (a b)) (* (+ 2 1) 5))

    --> ((length*r (a b) res)) und (((* (+ 2 1) 5)))
    a
        (* Abgeflachter Kopf und
           arithmetisch abgeflachter Rest *)

    --> (((length*r (a b) res)) (res (* (+ 2 1) 5)))
    b
        (* Zusammenhaengen und Einfuegen von res *)
```

Bsp.2.2.7.b:

```
        ((length (a b)) (* (+ 2 1) (length (a))))

    --> ((length*r (a b) res1)) und (((length*r (a) res2))
    a                                ((* (+ 2 1) res2)))

        (* Abgeflachter Kopf und
           arithmetisch abgeflachter Rest *)

    --> (((length*r (a b) res1) (length*r (a) res2))
    b    (res1 (* (+ 2 1) res2)))

        (* Zusammenhaengen und Einfuegen von res1 *)
```

4. Trifft keiner der drei vorherigen Faelle zu, dann ist das erste Element ein Datum oder ein arithmetisches Funktionssymbol.

a) Flache den Rest der Liste arithmetisch ab.

b) Fuege das erste Element in die abgeflachte Liste ein.
Auch hier koennen beim Einfuegen zwei Faelle auftreten.

Bsp.2.2.8.a:

```
        (plus (* 3 2) 1)

    --> (((* 3 2) 1))
    a
        (* Arithmetisch abgeflachter Rest *)

    --> ((plus (* 3 2) 1))
    b
```

```
        (* Einfuegen von 'plus' (= erstes Element) *)
```

Bsp.2.2.8.b:

```
        (* (+ 5 (length (a b))) 2)

    --> (((length*r (a b) res)) ((+ 5 res) 2))
    a
        (* Arithmetisch abgeflachter Rest *)

    --> (((length*r (a b) res)) (* (+ 5 res) 2))
    b
        (* Einfuegen von '*' (= erstes Element) *)
```

2.3 Sonderfaelle

Ist das Funktionssymbol ein "quote", so wird der gequotete Ausdruck nicht weiter abgeflacht.

Bsp.2.3.1.a:

```
    (quote (car (a b))) --> ((quote*r (car (a b)) res))
```

Bsp.2.3.1.b:

```
        (cons '(car (a b)) (c d))

    --> ((quote*r (car (a b)) res1) (cons*r res1 (c d) res2))
```

Steht die geschachtelte Funktion in einem LISPLOG-Programm bereits auf der rechten Seite von einem is-Operator, so darf man fuer die Top-Level-Funktion keine neue Variable benutzen, da die zum is-Operator gehoerende Variable noch in anderen Praemissen der gleichen Klausel vorkommen kann.

Man geht daher so vor:

a) Flache den geschachtelten Funktionsaufruf wie gewoehnlich nach Algorithmus 2.1 ab.

b) Ersetze die von Algorithmus 2.1 erzeugte neue Ergebnisvariable durch die zum is-Operator gehoerende Variable.

Bsp.2.3.2.a

```
        (is res (reverse (a b)))

    --> ((reverse*r (a b) resl))
    a
        (* Abgeflachte Liste *)

    --> ((reverse*r (a b) res))
    b
        (* resl durch res ersetzt *)
```

Bsp.2.3.2.b:

```
        (is res (+ 3 2))

    --> ((is resl (+ 3 2)))
    a
        (* Abgeflachte Liste *)

    --> ((is res (+ 3 2)))
    b
        (* resl durch res ersetzt *)
```

Tritt die Top-Level-Funkton als Goal eines LISPLOG-Programms auf, so braucht fuer sie keine neue Ergebnisvariable erzeugt zu werden. Der Wert wird als success bzw. failure interpretiert. In diesem Fall wird ein '*p' an das LISP-Funktionssymbol gehaengt. Fuer diesen Fall steht eine eigene Funktion zur Verfuegung. Der Benutzer ist also dafuer verantwortlich, ob der Funktionsaufruf als Aufruf einer Relation oder einer allgemeinen Funktion interpretiert wird (vgl. auch Teil II).

Bsp.2.3.3:

```
        (null (reverse (a b)))

    --> ((reverse*r (a b) res) (null*p res))
```

Ist der Ausdruck ein Aufruf der Funktion "not" so wird das Symbol "not" vor dem abgeflachten Argument, das eine Relation ist, eingefuegt.

Bsp.2.3.4:

```
        (not (null (reverse (a b))))

    --> ((reverse*r (a b) res) (not (null*p res)))
```

3. Optimierung von abgeflachten Funktionen

Um zu verhindern, dass spaeter im LISPLOG-Programm gleiche Goals doppelt bewiesen werden, kann man die abgeflachte Funktion nach gleichen Ausdruecken durchsuchen (Gleichheit wird bis auf die Ergebnisvariable der Funktion geprueft).

Bsp.3.1:

```
(f1 a (f2 a (f3 b c)) (f2 a (f3 b c)))
```

Dies wuerde nicht optimiert folgendes liefern:

```
((f3*r b c res1) (f2*r a res1 res2)
 (f3*r b c res3) (f2*r a res3 res4) (f1*r a res2 res4 res5))
```

Die Goals (f3*r b c res1) und (f3*r b c res3), bzw. (f2*r a res1 res2) und (f2*r a res3 res4) sind gleich, da sie mit freien Ergebnisvariablen aufgerufen werden. Sie wuerden doppelt bewiesen werden.
Man kann dies verhindern, indem man (f3*r b c res3) und (f2*r a res3 res4) streicht und die Ergebnisvariable res3 durch res1 und res4 durch res2 ersetzt.

Man wuerde dann folgendes erhalten:

```
((f3*r b c res1) (f2*r a res1 res2) (f1*r a res2 res2 res5))
```

Man hat also zwei Goals gespart.

Algorithmus zur Optimierung

1. Nimm das erste Element und pruefe ob dieses Element in dem Rest der Liste enthalten ist.
Ist dies der Fall, so loesche die gefundenen gleichen Elemente und ersetze die Ergebnisvariablen der geloeschten Elemente - welche in dem Rest der Liste nach dem geloeschten Element noch genau einmal vorkommen muessen - durch die Ergebnisvariable des ersten Elements.
In unserem Beispiel oben wuerde man in diesem Schritt (f3*r b c res3) loeschen und dann in dem Rest (hier: ((f2*r a res3 res4) (f1*r a res2 res4 res5))) res3 durch res1 ersetzen.
Man wuerde also als Ergebnis nach dem ersten Schritt erhalten:

```
((f3*r b c res1) (f2*r a res1 res2) (f2*r a res1 res4)
 (f1*r a res2 res4 res5))
```

2. Fahre auf die gleiche Weise mit dem 2. Element usw. fort, bis

die Liste leer ist.

Bemerkung:

Um erfolgreicher optimieren zu koennen, werden die aequivalenten LISP-Funktionen

(+, sum, add, plus) zu plus,
(*, product, times) zu times,
(-, difference, diff) zu diff,
(/, quotient, divide) zu divide,
(remainder, mod) zu mod,
(|1+|, add1) zu add1 und
(|1-|, sub1) zu sub1

normalisiert.

4. Implementierung

4.1 Hauptfunktionen

(opt-flattenfunc 'l_list)
liefert als Ergebnis die optimierte abgeflachte Liste von l_list.
L_list hat die Form wie in 1. beschrieben oder hat die Form (is res F), wobei F wieder die Form von 1. hat.
"opt-flattenfunc" flacht mit "flattenfunc" l_list ab und wendet darauf die Optimierungsfunktion "opt" an.
Ist der Kopf von l_list gleich 'is', so wird wie in den Beispielen 2.3.2.a-b vorgegangen und dann optimiert.

(opt-flattenrel 'l_list)
liefert als Wert die abgeflachte und optimierte Liste von l_list. L_list hat die Form wie in 1. beschrieben. "opt-flattenrel" flacht mit "flattenrel" l_list ab und wendet darauf die Optimierungsfunktion "opt" an. Diese Funktion wird nur benutzt, wenn man weiss, dass die Top-Level-Funktion eine Relation ist.

(flattenfunc 'l_list)
liefert als Wert die abgeflachte Form von l_list, wie in Algorithmus 2.1 beschrieben.
Alle beim Abflachen vorkommenden LISP-Funktionsnamen werden in den globalen Variablen mind-func und still-func gespeichert. Zusaetzlich wird allen Funktionsnamen in der abgeflachten Form ein '*r' angehaengt (siehe "newn").
Wird ein arithmetischer Funktionsaufruf erkannt, wird die Funktion "arith-func" aufgerufen.
Beim Abflachen muss noch gesondert abgefragt werden, ob eine Liste eine LISPLOG-Variable ist, da das Fragezeichen in z.B. (? res) sonst als Funktionssymbol erkannt wuerde, und somit auch eine Variable "abgeflacht" wuerde.

(flattenrel 'l_list)
erzeugt fuer die Top-Level-Funktion von l_list keine Ergebnisvariable. Sonst ist "flattenrel" aequivalent zu "flattenfunc".
Der Relationsname wird in den globalen Variablen mind-rel und still-rel gespeichert. In der abgeflachten Form wird an den Relationsnamen ein '*p' gehaengt (siehe "newrel").

(arith-func 'l_list)
ruft "flatten-arithfunc" auf und fuegt den is-Operator in die von "flatten-arithfunc" abgeflachte Form von l_list ein, wie in Beispiel 2.2.4.a-b.

(flatten-arithfunc 'l_list)
flacht einen arithmetischen Funktionsaufruf so ab, wie in Algorithmus 2.2 beschrieben.

(opt 'l_list)
optimiert die abgeflachte Liste l_list wie im Algorithmus zur Optimierung beschrieben. "opt" ruft die Funktion "delete-member" auf.

4.2 Hilfsfunktionen

(insert-arg 'g_arg 'l_list)
fuegt das Element g_arg in die abgeflachte Liste l_list ein, wie in Beispiel 2.1.5.

(append-and-insert-resvar-1 'l_listl 'l_list2)
haengt die abgeflachten Listen l_listl und l_list2 zusammen und fuegt eine Ergebnisvariable ein, wie in Beispiel 2.1.6.

(insert-is-resvar 'g_isresvar 'l_list)
fuegt den is-Operator sowie die Ergebnisvariable in die arithmetisch abgeflachte Liste l_list ein, wie in den Beispielen 2.2.4.a-b.

(append-arith-flatten-lists 'l_listl 'l_list2)
haengt die arithmetisch abgeflachten Listen l_listl und l_list2 aneinander, wie in den Beispielen 2.2.6.a-d.

(append-and-insert-resvar-2 'l_listl 'l_list2)
haengt die abgeflachte Liste l_listl und die arithmetisch abgeflachte Liste l_list2 aneinander und fuegt eine Ergebnisvariable ein, wie in den Beispielen 2.2.7.a-b.

(insert-arith-arg 'g_aritharg 'l_list)
fuegt das Element g_arith-arg in die arithmetisch abgeflachte Liste l_list ein, wie in den Beispielen 2.2.8.a-b.

(append-and-insert-arith-resvar 'l_listl 'l_list2)
haengt die arithmetisch abgeflachte Liste l_listl und die abgeflachte Liste l_list2 zusammen und fuegt eine Ergebnisvariable ein, wie in Beispiel 2.2.3.

(insert-not 'g_sym 'l_list)
fuegt das Symbol 'not' in die Liste l_list ein, wie in Beispiel 2.3.4.

(replace-resvar 'g_resvar 'l_list)
ersetzt in der abgeflachten Liste l_list ein Element durch das Element g_resvar, wie in Beispiel 2.3.2.a.

(replace-arith-resvar 'g_arith-resvar 'l_list)
ersetzt in der arithmetisch abgeflachten Liste l_list ein Element durch das Element g_arith-resvar, wie in Beispiel 2.3.2.b.

(begin 'l_list)
liefert als Wert die Liste l_list ohne ihr letztes Element.

(last-l 'l_list)
liefert als Wert das letzte Element von l_list.

Bsp: (last-l '(a b c)) --> c

(laast-l 'l_list)
hat als Argument eine Liste von Listen und liefert als Wert das letzte Element des letzten Elements.

Bsp.: (laast-l '((a b) (c d))) --> d

(create-var)
erzeugt eine neue Variable der Form (((? res00x))). Die doppelte Klammerung ist notwendig, damit die Variable die fuer die Abflachungsfunktion notwendige Form hat.

(create-arith-var)
erzeugt den is-Operator und eine neue Variable der Form (is (? res00x)).

(newn 's_funcname)
haengt an s_funcname '*r' an.

(newrel 's_relname)
falls s_relname in der Liste (eq equal greaterp > >& < <& lessp = =& litatom atom numberp) enthalten ist, fuer die es in Standard-PROLOG Entsprechungen gibt, bleibt s_relname gleich, sonst wird '*p' angehaengt.

(arith-func-sym 'g_sym)
prueft nach, ob g_sym ein arithmetisches Funktionssymbol ist.

(not-arith-func-sym 'g_sym)
prueft nach, ob g_sym ein nicht-arithmetisches Funktionssymbol ist.

(quote-sym 'g_sym)
prueft nach, ob g_sym ein das Symbol 'quote' ist.

(delete-member 'l_elem 'l_list)
ruft "delete-arith-member" auf, falls der Kopf von l_elem 'is' ist, sonst wird "delete-notarith-member" aufgerufen. Diese Unterscheidung erfolgt aus Effizienzgruenden.

(delete-notarith-member 'l_elem 'l_list)
loescht die Vorkommen von l_elem in l_list, wie im Algorithmus zur Optimierung in Schritt 1 beschrieben; ruft dabei

"search" auf.

(delete-arith-member 'l_elem 'l_list)
behandelt nur l_elem der Form '(is res (...)), sonst aequivalent zu "delete-notarith-member".

(search 'g_x 'g_y 'l_list)
l_list ist eine Liste von Listen. "Search" sucht in den Listen mit Hilfe von "member-l" nach dem ersten Vorkommen von g_x und ruft dann "replace" auf.

(replace 'g_x 'g_y 'l_list)
ersetzt in einer beliebig geschachtelten Liste das Element g_x durch das Element g_y.

(member-l 'g_x 'l_list)
prueft nach, ob g_x in der beliebig geschachtelten Liste l_list enthalten ist.

(make-uniform 'g_x)
vereinheitlicht aequivalente LISP-Funktionen wie im Kapitel zur Optimierung beschrieben.

4.3 Globale Variablen

arith-functions
hat als Wert eine Liste der arithmetischen Funktionen.

still-func
hat als Wert eine Liste von LISP-Funktionen, die noch uebersetzt werden muessen.

still-rel
hat als Wert eine Liste von LISP-Praedikaten, die noch zu uebersetzen sind.

mind-func
hat als Wert eine Liste aller bisher gefundenen Funktionen.

mind-rel
hat als Wert eine Liste aller bisher gefundenen Praedikate.

Fuer jede LISP-Funktion, die der Abflachungsalgorithmus findet, testet er, ob die Funktion vorher schon aufgetreten war. In diesem Fall muss ihr Name in der Liste 'mind-func' bzw. 'mind-rel' enthalten sein. Ist der Funktionsname dort nicht enthalten, wird er sowohl zu 'mind-func' bzw. 'mind-rel' als auch zu der entsprechenden Liste 'still-func' bzw. 'still-rel' hinzugefuegt. Mit Hilfe der Listen 'mind-func' und 'mind-rel' wird also verhindert, dass Funktionen mehrfach uebersetzt werden.

Der Uebersetzungsalgorithmus (Teil II) greift auf die Variablen 'still-func' und 'still-rel' zu. Er uebersetzt jeweils eine Funktion und loescht sie aus der entsprechenden Liste.

Teil II

Umwandlung von lambda-Definitionen zu Horn-Klauseln

Knut Hinkelmann

1. Grundlagen

1.1 Grundlagen der Uebersetzung

Angenommen wir wollen eine Funktion (f x y) uebersetzen, die folgende Definition hat:

```
(def f
     (lambda (x y)
             (cons (car x) y)))
```

Diese kann nach [Kowalski 1985] folgendermassen in eine Horn-Klausel uebersetzt werden:

```
      (f x y) = (cons (car x) y)

<-->  (f*r x y (cons (car x) y))                          (E1)

<-->  (f*r x y $res)  if  (cons (car x) y) = $res        (E2)

<-->  (f*r x y $res)  if  (cons*r (car x) y $res)        (E1)

<-->  (f*r x y $res)  if  (cons*r xl y $res)
                      and (car x) = xl                   (E2)

<-->  (f*r x y $res)  if (cons*r xl y $res)
                      and (car*r x xl)                   (E1)
```

Man muss also den Rumpf der Definition in eine Konjunktion von Relationen abflachen, die die Praemissen der Horn-Klausel bilden.

1.2 Die Konklusion

Das Aufrufmuster der LISPLOG-Klausel, also die Konklusion, gewinnt man aus dem Aufrufmuster der LISP-Funktion wie folgt:

Die Konklusion ist eine Liste mit dem Praedikat als erstem Element. Das Praedikat gewinnt man aus dem LISP-Funktionsnamen durch Anhaengen von '*p' bei einem LISP-Praedikat bzw. durch Anhaengen von '*r' bei einer allgemeinen LISP-Funktion.

Als naechste Elemente folgen die Argumente, die man durch Umwandlung der Argumente der LISP-Funktion in LISPLOG-Variablen erhaelt.

Falls es sich nicht um ein LISP-Praedikat handelt, wird als letztes Argument noch '_$res' eingefuegt, das als Wert das Resultat der Funktion annehmen wird.

Beispiel 1.2.1:

```
(append l1 l2) --> (append*r _l1 _l2 _$res)

(member a l)   --> (member*p _a _l)
```

1.3 Die Praemissen

Es werden nur Funktionen uebersetzt, die als Rumpf genau einen Funktionsaufruf haben, der natuerlich beliebig geschachtelt sein kann (siehe 1.9). Der Rumpf wird umgewandelt zu den Praemissen der LISPLOG-Klausel.

Als erstes wird der LISP-Funktionaufruf durch den in Teil I beschriebenen Algorithmus abgeflacht. Das Resultat der LISP-Funktion ist entweder der Rumpf selbst, wenn der Rumpf ein konstantes Atom ist (z.B. 'nil' oder eine Zahl), oder es steht als letztes Argument in der abgeflachten Form der top-level-Funktion des Rumpfes. Dieses Resultat wird mit dem letzten Argument '_$res' der Konklusion unifiziert.

Beispiel 1.3.1:

Die LISP-Funktion

```
(def f
     (lambda (x y)
             (cons (car x) y)))
```

wird nach 1.1 zu

```
(f*r x y $res)  if  (car*r x xl)  and  (cons*r xl y $res)
```

Dies kann man folgendermassen umformen:

```
(f*r x y $res) if      (car*r x xl)
                  and (cons*r xl y x2)
                  and (= $res x2)
```

wobei '=' Unifikation bedeutet. Diese Unifikation ist notwendig, weil der Abflachungsalgorithmus aus Teil I eine Variable fuer das Resultat des Rumpfes erzeugt (hier x2). Unabhaengig davon wird die Konklusion gebildet, die als zusaetzliches Argument die Variable $res bekommt. Bei der Optimierung (siehe 1.7 a)) wird die Unifikation wieder aus den Praemissen geloescht, indem man $res durch die Ergebnisvariable (hier x2) ersetzt.

Schreibt man obige Klausel in LISPLOG-Syntax um, so erhaelt man:

```
(ass (f*r _x _y _$res) (car*r _x _xl)
                       (cons*r _xl _y _x2)
                       (= _$res _x2))
```

Das ganze kann man folgendermassen lesen:

"Der Relationsaufruf (f*r _x _y _$res) ist erfolgreich mit einer Bindung von _$res, wenn der Relationsaufruf (car*r _x _xl) und der Relationsaufruf (cons*r _xl _y _x2) erfolgreich sind und _$res mit _x2 unifizierbar ist." (die PROLOG-Lesart)

oder:

"(f x y) hat den Wert $res, wenn (car x) den Wert xl hat, (cons xl y) den Wert x2 hat und x2 gleich $res ist." (eine LISPLOG-Lesart)

oder abgekuerzt:

"(f x y) hat den Wert (cons (car x) y)." (die LISP-Lesart)

Treten innerhalb des Rumpfes LISP-Funktionen auf, die noch nicht nach LISPLOG uebersetzt sind, so werden diese in einer Variablen gespeichert und spaeter noch uebersetzt.

1.4 Die LISP-Funktion 'cond'

Ist der Rumpf ein Aufruf der Funktion 'cond', so wird in LISP eine Aktion nur ausgefuehrt, wenn die entsprechende Bedingung einen non-nil-Wert hat.

Beispiel 1.4.1:

LISP-Funktion:

```
(def f
    (lambda (x)
       (cond ((bed1 x) (akt1 x))
             ((bed2 x) (akt2 x)))))
```

Aus dieser Funktion mit zwei 'cond'-Klauseln ergibt sich nach [Kowalski 1985] folgende Uebersetzung in eine Relation mit zwei Horn-Klauseln:

```
a)     (f x)  =  ((bed1 x) (akt1 x))

  <--> (f*r x ((bed1 x) (akt1 x)))

  <--> (f*r x $res)  if  ((bed1 x) (akt1 x)) = $res

  <--> (f*r x $res)  if  (bed1*p x)  and  (akt1 x) = $res      (*)

  <--> (f*r x $res)  if  (bed1*p x)  and  (akt1*r x $res)


b)     (f x)  =  ((bed2 x) (akt2 x))

  <--> (f*r x ((bed2 x) (akt2 x)))

  <--> (f*r x $res)  if  ((bed2 x) (akt2 x)) = $res

  <--> (f*r x $res)  if  (bed2*p x)  and  (akt2 x) = $res      (*)

  <--> (f*r x $res)  if  (bed2*p x)  and  (akt2*r x $res)
```

LISPLOG-Entsprechung des Uebersetzungsresultats:

```
(ass (f*r _x _$res) (bed1*p _x) (akt1*r _x _r1) (= _$res _r1))
(ass (f*r _x _$res) (bed2*p _x) (akt2*r _x _r2) (= _$res _r2))
```

In Worten:

"(f x) hat den Wert $res, wenn Bedingung (bed1 x) erfuellt ist, (akt1 x) den Wert r1 hat und $res = r1 ist, oder
(f x) hat den Wert $res, wenn Bedingung (bed2 x) erfuellt ist, (akt2 x) den Wert r2 hat und $res = r2 ist."

An der Stelle (*) ging die Semantik von 'cond'-Klauseln in die Uebersetzung ein. Kowalski verwendet in seinem Beispiel fuer die Umwandlung der Funktion 'length' ebenfalls fuer jede Bedingung eine eigene Klausel, uebernimmt die Bedingung aber in das Aufrufmuster der Konklusion.

Da es bei Bedingungen nur darauf ankommt, ob sie den Wert 'nil' (PROLOG: fail) oder einen non-nil-Wert (PROLOG: success) haben, werden sie als Praedikate ohne zusaetzliches Argument uebersetzt.

Die Reihenfolge der Abarbeitung der Horn-Klauseln ist i.a. nicht beliebig, sondern haengt von der Reihenfolge der 'cond'-Klauseln ab. Wenn in einer 'cond'-Kaskade eine Bedingung erfuellt ist, wird die entsprechende Aktion ausgefuehrt. Die nachfolgenden Bedingungen und Aktionen der 'cond'-Kaskade werden nicht mehr beachtet. Dies wuerde in PROLOG direkt einem 'cut' entsprechen. Dieser muesste nach jeder 'cond'-Bedingung, nachdem sie nach LISPLOG uebersetzt wurde, eingefuegt werden (zur Vermeidung des 'cut' siehe 1.6).

Ohne den 'cut' koennte folgender Fall eintreten: Oftmals ist die letzte 'cond'-Klausel eine sogenannte 'catch-all'-Klausel mit der Bedingung 't'. Ergibt sich durch eine vorherige Klausel eine Bindung fuer _$res, die die nachfolgenden Goals des Beweises nicht erfuellt, so setzt Backtracking ein. Durch die 'catch-all'-Klausel wuerde dann ein weiterer Wert an _$res gebunden werden. Funktionsergebnisse sind jedoch eindeutig. Die Semantik der PROLOG-Relation wuerde also derjenigen der LISP-Funktion nicht mehr entsprechen.

1.5 'cond'-Bedingungen

Bedingungen von 'cond'-Klauseln koennen auch Aufrufe von 'and'- und 'or'- Funktionen sein, die mehrere Praedikate verknuepfen. Dadurch kann eine 'cond'- Klausel zu mehreren LISPLOG-Klauseln fuehren.

Die beiden 'and' aus LISP und LISPLOG kann man identifizieren. Das Uebersetzungsverfahren basiert darauf, dass die DNF-Normalisierung die Reihenfolge der and-or-Argumente erhaelt, die sowohl in LISP als auch in PROLOG von Bedeutung ist.

Beispiel 1.5.1:

LISP-Funktion

```
(def f
     (lambda (x)
             (cond ((and (b1 x) (b2 x)) (a1 x))
                   ((or  (b3 x) (b4 x)) (a2 x))
                   ((and (or (b5 x) (b6 x))
                         (or (b7 x) (b8 x))) (a3 x)))))
```

Uebersetzung:

```
a)   (f*r x $res) if (and (b1 x) (b2 x)) and (a1*r x $res)  (1.4)

<--> (f*r x $res) if (b1*p x) and (b2*p x) and (a1*r x $res)
```

(wegen Assoziativitaet von 'and')

```
b)   (f*r x $res) if (or (b3 x) (b4 x)) and (a2*r x $res)   (1.4)

<--> (f*r x $res) if ((b3*p x) or (b4*p x)) and (a2*r x $res) (x)
```

(Dies ist keine Horn-Klausel, man muss also die Formel durch Anwendung der Distributivgesetze, De Morgan-Regeln und der Regel 'K if P <--> K or not(P)' umformen.)

```
<--> (f*r x $res) if ((b3*p x) and (a2*r x $res))
                  or ((b4*p x) and (a2*r x $res))                (+)

<--> (f*r x $res) or not(   ((b3*p x) and (a2*r x $res))
                         or ((b4*p x) and (a2*r x $res)))

<--> (f*r x $res) or (    not((b3*p x) and (a2*r x $res))
                      and not((b4*p x) and (a2*r x $res)))

<--> (f*r x $res) or (    ( not(b3*p x) or not(a2*r x $res))
                      and ( not(b4*p x) or not(a2*r x $res)))

<--> ((f*r x $res) or not(b3*p x) or not(a2*r x $res))  and
     ((f*r x $res) or not(b4*p x) or not(a2*r x $res))

<--> (f*r x $res) if (b3*p x) and (a2*r x $res)
     (f*r x $res) if (b4*p x) and (a2*r x $res)
```

Auf diese Weise erhaelt man aus einer 'cond'-Klausel zwei Horn-Klauseln. Man beachte, dass die Aktion (a2*r x $res) in jeder der beiden Horn-Klauseln vorkommt.

```
c)   (f*r x $res) if      ((b5*p x) or (b6*p x))
                      and ((b7*p x) or (b8*p x))
                      and (a3*r x $res)

<--> (f*r x $res) if (   ((b5*p x) and (b7*p x))
                      or ((b5*p x) and (b8*p x))
                      or ((b6*p x) and (b7*p x))                 (x)
                      or ((b6*p x) and (b8*p x)))
                     and (a3*r x $res)
```

Durch Anwendung des Distibutivgesetzes erhaelt man (+):

```
(f*r x $res) if    ((b5*p x) and (b7*p x) and (a3*r x $res))
                or ((b5*p x) and (b8*p x) and (a3*r x $res))
                or ((b6*p x) and (b7*p x) and (a3*r x $res))
                or ((b6*p x) and (b8*p x) and (a3*r x $res))
```

Analog zu b) erhaelt man die folgenden Horn-Klauseln:

```
(f*r x $res) if (b5*p x) and (b7*p x) and (a3*r x $res)
(f*r x $res) if (b5*p x) and (b8*p x) and (a3*r x $res)
(f*r x $res) if (b6*p x) and (b7*p x) and (a3*r x $res)
(f*r x $res) if (b6*p x) and (b8*p x) and (a3*r x $res)
```

Auch hier ist die Aktion wieder in jeder Klausel enthalten.

Daraus kann man nun folgenden Schluss ziehen:

Hat man auf der rechten Seite von if, also in den Praemissen, eine disjunktive Normalform (DNF) (siehe (+)), so wird jedes Disjunkt die rechte Seite einer Hornklausel. Aus einer 'cond'-Klausel erhaelt man eine DNF, indem man die Bedingungen in eine DNF umformt, diese mit der Aktion durch 'and' verknuepft (siehe (x)) und das ganze wieder in eine DNF transformiert.

Durch einen eingeschachtelten Aufruf der Funktion 'cond' als Aktion, kann auch aus der Aktion eine disjunktive Normalform entstehen. An der Stelle (x) haette man dann die Verknuepfung zweier DNF durch 'and'. Wandelt man diese Verknuepfung wiederum in eine DNF um, so erhaelt man schliesslich eine DNF, deren Disjunkte jeweils eine rechte Seite einer Horn-Klausel darstellen (siehe (+)).

Die Reihenfolge der Abarbeitung dieser Klauseln ist beliebig, da alle die gleiche Aktion haben und somit das gleiche Ergebnis liefern. Durch Nachforderung oder Backtracking kann also mehrmals der gleiche Wert auftreten. Um dies zu verhindern, kann man eine Reihenfolge beliebig festlegen und nach der Bedingung in jeder Klausel einen 'cut' einfuegen (zur Vermeidung des 'cut' siehe 1.6).

1.6 Der 'cut'

Wenn die zu uebersetzende LISP-Funktion in einem LISPLOG-Programm vorkommt, so sind die Argumente instanziiert (freie Variable hoechstens noch in quotierten Kontexten) und nur der Funktionswert muss berechnet werden. Genau fuer diese Anwendungsrichtung ist auch jede LISP-Funktion geschrieben. Der Wert einer Funktion ist fuer vorgegebene Argumente aber eindeutig, d.h. wenn ein Wert

gefunden ist, kann es keine anderen mehr geben.

In 1.4 und 1.5 wurde angeregt, in jeder Klausel zu einem Praedikat einen 'cut' einzufuegen. Dies hat folgende Gruende:

1. Bei Backtracking koennte der gleiche Funktionswert durch eine nachfolgende Klausel noch einmal berechnet werden (vgl 1.5). Dies waere fuer die Korrektheit zwar nicht schaedlich, aber die dafuer benoetigte Rechenzeit waere vergeudet, da das Goal natuerlich auch im zweiten Versuch mit dem gleichen Wert nicht erfuellt werden kann.

2. Bei Backtracking koennte durch eine nachfolgende Klausel ein anderer Funktionswert berechnet werden. Dies koennte vor allem bei der letzten Klausel, der sogenannten 'catch-all'-Klausel mit der Bedingung 't' der Fall sein, wodurch das LISPLOG-Programm nicht korrekt waere. Dieser Fall muss unbedingt ausgeschlossen werden, was durch einen 'cut' in jeder Klausel, wie in 1.4 vorgeschlagen, auch gelingen wuerde.

Nun ist aber in der Sprache LISPLOG der allgemeine 'cut' nicht erlaubt. Der hier benoetigte 'cut' kann jedoch durch die Verwendung von 'not' ganz vermieden werden (vgl. auch [Clocksin & Mellish 1981]), indem man alle Bedingungen der vorherigen Klauseln negiert, die vor dem 'cut' stehen wuerden (hierzu wird 'not' nur fuer Goals ohne Anfragevariable aufgerufen). Diese Verwendung von 'not' gegenueber 'cut' macht das Programm zwar ineffizienter (mehrfache Auswertung von gleichen Praedikatsaufrufen), aber die Abarbeitung der Klauseln wird von der Reihenfolge unabhaengig (deklarative Lesbarkeit).

a)

Allgemein wuerde die Uebersetzung folgendermassen aussehen:

Beispiel 1.6.1:

LISP-Funktion:

```
(def f
     (lambda (x)
             (cond ((bed1 x) (akt1 x))
                   ((bed2 x) (akt2 x)))))
```

LISPLOG-Kauseln:

```
(ass (f*r _x _$res) (bed1*p _x) (akt1*r _x _r1) (= _$res _r1))
(ass (f*r _x _$res) (not (bed1*p _x))
                    (bed2*p _x)
                    (akt2*r _x _r2)
                    (= _$res _r2))
```

Man beachte, dass die erste Horn-Klausel keine Einschraenkung (not (bed2*p _x)) verwendet, da auch in der ersten 'cond'-Klausel durchaus zusaetzlich zu bed1 auch bed2 gelten kann, wenn akt1 ausgefuehrt wird.

Fuehren die Uebersetzungen von (bed1 x), (bed2 x) und (akt2 x) zu disjunktiven Normalformen mit mehreren Disjunkten, so muessen die Uebersetzung der Negation der vollstaendigen Bedingung (bed1 x) sowie die Uebersetzungen von (bed2 x) und (akt2 x) als DNF durch 'and' verknuepft werden, um eine DNF zur Erzeugung der Horn-Klauseln zu erhalten (vgl 1.5).

b)

Fuehrt eine 'cond'-Klausel nach 1.5 zu mehreren Horn-Klauseln, so muss zu jeder Horn-Klausel noch die Negation der Disjunkte der zu dieser 'cond'-Klausel gehoerenden Bedingung hinzugefuegt werden, die in der Reihenfolge vor dieser Horn-Klausel stehen.

Beispiel 1.6.2:

Sei (or (and a1 a2) (and b1 b2 b3) (and c1 c2))

die DNF einer Bedingung. Zu dem Disjunkt '(and c1 c2)' gehoeren die Negationen der beiden ersten Disjunkte '(not (and a1 a2))' und '(not (and b1 b2 b3))'. Die Umwandlung berechnet sich wie folgt:

```
     (not (or (and a1 a2) (and b1 b2 b3)))

-->  (and (or (not a1) (not a2)) (or (not b1) (not b2) (not b3)))
(1)

-->  (or (and (not a1) (not b1))
(2)      (and (not a1) (not b2))
         (and (not a1) (not b3))
         (and (not a2) (not b1))
         (and (not a2) (not b2))
         (and (not a2) (not b3)))
```

Als letzter Schritt muessen zu jedem Disjunkt die Bedingungen c1 und c2 hinzugefuegt werden. Man erhaelt:

```
     (or (and (not a1) (not b1) c1 c2)
         (and (not a1) (not b2) c1 c2)
         (and (not a1) (not b3) c1 c2)
         (and (not a2) (not b1) c1 c2)
         (and (not a2) (not b2) c1 c2)
         (and (not a2) (not b3) c1 c2))
```

Dazu noch folgende Anmerkungen:

1. Waere im obigen Beispiel b3 = c1, so koennte man b3

streichen, da die Klauseln, in denen (not b3) vorkaeme, nicht erfuellbar waeren.

2. Aus Effizienzgruenden fuer das LISPLOG-Programm sollten auch doppelt vorkommende Terme in einer Klausel vermieden werden (vgl. Teil I). Dies waere z.B. der Fall in der vierten Klausel, falls a2 = b1 waere.

3. Der dritte Fall, der verhindert werden sollte, ist derjenige, dass zwei bis auf die Reihenfolge gleiche Klauseln auftreten. Nehmen wir dazu an, dass a1 = b2 und a2 = b1 ist. Dann stimmen das erste und das fuenfte Disjunkt ueberein.

1.7 Optimierung der Klauseln

a)

Es gibt Bedingungen und Aktionen, die man nicht als Praemissen uebersetzen sollte. Stattdessen sollten die Terme der Klauseln geaendert werden. Dies soll anhand der folgenden Beispiele gezeigt werden (je ein Beispiel fuer alle im Programm vorkommenden Faelle).

Gegeben seien folgende Strukturen (koennen Praemissen oder auch Konklusionen sein):

```
(f*r _x _y _z)  (g*r (_u . _v) _x)
```

Die folgende Auflistung zeigt links vom Pfeil Beispiele von Praemissen, die man durch die Optimierungsfunktion wegfallen lassen kann, indem man die Strukturen zu den auf der rechten Seite angegebenen Instanziierungen der obigen Strukturen aendert. Zu beachten ist, dass die Vorkommen des betreffenden Argumentes in allen Termen der Klausel, also insbesondere auch in der Konklusion, ersetzt werden. Ausserdem muessen die neu hinzukommenden Variablen fuer car oder cdr von Listen (hier _y1, _z1) Namen haben, die sonst nirgendwo in der Klausel vorkommen. Dadurch koennen Bedingungen schon durch Unifikation des Goals mit der Konklusion getestet werden.

Beispiele 1.7.1:

```
(null*p _x)        --> (f*r nil _y _z) (g*r (_u . _v) nil)

(zerop*p _x)       --> (f*r 0 _y _z) (g*r (_u . _v) 0)

(car*r _y _z)      --> (f*r _x (_z . _y1) _z) (g*r (_u . _v) _x)
```

```
(car*r (_u . _v) 2) --> (f*r _x _y _z) (g*r (2 . _v) _x)

(cdr*r _z 5)        --> (f*r _x _y (_zl . 5)) (g*r (_u . _v) _x)

(cdr*r (_u . _v) 2) --> (f*r _x _y _z) (g*r (_u . 2) _x)

(cons*r _a _b _z)   --> (f*r _x _y (_a . _b)) (g*r (_u . _v) _x)

(= _u 4)            --> (f*r _x _y _z) (g*r (4 . _v) _x)

(= _a _y)           --> (f*r _x _a _z) (g*r (_u . _v) _x)
```

Bei der letzten Umformung ist die Instanziierung zu der einfachen Umbenennung entartet.

Es koennte nun auch jemand auf die Idee kommen, den Term '(not (null*p _x))' zu loeschen, indem man _x durch (_xl . _x2) ersetzt. Dadurch koennte das Argument zwar nicht mit der leeren Liste unifiziert werden, da zur Erfuellung der Bedingung die Variable _x aber auch mit einem Atom ungleich 'nil' als Wert instanziiert werden kann, ist die Ersetzung nicht moeglich.

b)

Es gibt Terme, die man als Praemissen loeschen kann, da sie beim Beweis sofort zu 'success' fuehren. Die folgende Aufstellung zeigt solche Terme:

Beispiele 1.7.2:

```
t

5

(null*p nil)

(atom r)

(zerop*p 0)

(not (null*p (_x . _y)))

(not (atom (_x . _y)))

(not (zerop*p 5))
```

Eine Regel degeneriert zu einem Fakt, wenn alle Bedingungen durch die Optimierungsfunktion geloescht und somit in das Aufrufmuster der einzigen noch verbleibenden Struktur, naemlich der

Konklusion, uebernommen werden konnten. Durch die uniforme Notation von Regeln und Fakten in LISPLOG braucht dieser Fall nicht gesondert behandelt zu werden.

Beispiel 1.7.3:

Optimierung einer Funktionsumwandlung:

LISP-Funktion:

```
(def append
     (lambda (l1 l2)
         (cond ((null l1) l2)
               (t (cons (car l1) (append (cdr l1) l2))))))
```

Umformung ohne Optimierung:

```
(ass (append*r _l1 _l2 _$res)
     (null*p _l1)
     (= _$res _l2))
(ass (append*r _l1 _l2 _$res)
     (not (null*p _l1))
     t
     (car*r _l1 _res1)
     (cdr*r _l1 _res2)
     (append*r _res2 _l2 _res3)
     (cons*r _res1 _res3 _res4)
     (= _$res _res4))
```

Optimierung nach a):

```
(ass (append*r nil _l2 _l2))
(ass (append*r (_res1 . _res2) _l2 (_res1 . _res3))
     (not (null*p (_res1 . _res2)))
     t
     (append*r _res2 _l2 _res3))
```

Optimierung nach b):

```
(ass (append*r nil _l2 _l2))
(ass (append*r (_res1 . _res2) _l2 (_res1 . _res3))
     (append*r _res2 _l2 _res3))
```

1.8 Funktionen und Praedikate

Waehrend allgemeine Funktionen einen Wert haben, der zu einem zusaetzlichen Argument im Relationsaufruf fuehrt, ist fuer LISP-Praedikate nur interessant, ob sie den Wert 'nil' (PROLOG: fail) oder einen non-nil-Wert (PROLOG: success) haben. Deshalb bekommen sie kein zusaetzliches Argument im Relationsaufruf. Stattdessen wird, falls ihre Aktion ein LISP-Atom ist, z.B. 'x', ihr Wert durch die zusaetzliche Praemisse '(not (null*p _x))' auf 'nil' oder 'non-nil' getestet.

Ob eine LISP-Funktion als allgemeine Funktion oder als Praedikat interpretiert werden soll haengt nicht von ihrer Definition ab. Entscheidend ist allein, wo sie aufgerufen wird. Wurde die Funktion als Bedingung einer 'cond'-Klausel oder als LISPLOG-Goal aufgerufen, so muss sie als Praedikat interpretiert werden. Selbstverstaendlich werden bei der Umwandlung von Praedikatsdefinitionen deren Aktionen ebenfalls als Praedikate interpretiert. Dagegen wird eine Funktion, deren Aufruf in einen Aufruf einer anderen Funktion geschachtelt ist, immer als allgemeine Funktion interpretiert.

Beispiel 1.8.1:

Gegeben sei folgende lambda-Definition der Funktion member. Sie hat als Wert nicht einfach 't', wenn das Element a in der Liste l enthalten ist, sondern den Rest der Liste, der mit dem ersten Auftreten von a beginnt.

```
(def member
     (lambda (a l)
             (cond ((null l) nil)
                   ((equal (car l) a) l)
                   (t (member a (cdr l))))))
```

Uebersetzung als allgemeine Funktion:

```
(ass (member*r _a nil nil))
(ass (member*r _a (_a . _xl) (_a . _xl)))
(ass (member*r _a (_x2 . _x3) _x4)
     (not (equal _a _x2))
     (member*r _a _x3 _x4))
```

Uebersetzung als Praedikat:

```
(ass (member*p _a nil) nil)
(ass (member*p _a (_a . _xl)))
(ass (member*p _a (_x2 . _x3))
     (not (equal _a _x2))
     (member*p _a _x3))
```

(In der zweiten und dritten Klausel wurden jeweils die Praemissen '(not (null*p (_a . _x1)))' bzw. '(not (null*p (_x2 . _x3)))' durch die Optimierungsfunktion geloescht.)

1.9 Einschraenkungen

Es koennen nicht alle LISP-Funktionen uebersetzt werden, sondern nur solche mit folgenden Einschraenkungen:

1. Die Funktion ist eine 'lambda'-Funktion und keine 'nlambda', 'lexpr' oder 'macro'.

2. Es handelt sich um eine pure Funktion. Der lambda-Rumpf besteht also z.B. nicht aus einer Folge von mehreren Funktionsaufrufen (implizites progn), sondern nur aus einem (geschachtelten) Funktionsaufruf.

3. Ist der Rumpf ein Aufruf der Funktion 'cond', so haben deren Klauseln jeweils genau eine Aktion, die weder ein Aufruf der 'progn'-Funktion (vgl. 2.) noch ein 'let'- oder 'lambda'-Ausdruck sein darf.

4. Compilierte Funktionen sind selbstverstaendlich verboten, z. B. kann man auf die lambda-Definition von Systemfunktionen nicht zugreifen.

2. Benutzeroberflaeche

Es gibt drei Arten, das Programm zu benutzen:

- Umwandlung mehrerer Funktionen bzw. Praedikate
- Umwandlung einer Funktion
- Umwandlung eines Praedikates

(translate 'l_func 'l_pred 's_file)
WERT: t
SEITENEFFEKT: Uebersetzt die in der Liste l_func angegebenen Funktionen und die in der Liste l_pred angegebenen LISP-Praedikate in LISPLOG-Klauseln und schreibt diese auf die Datei s_file. Ebenso werden die in den Ruempfen der LISP-Funktionen und -Praedikate auftretenden Funktionen uebersetzt.

(transfunction 's_functionname)
WERT: t bei erfolgreicher Uebersetzung, nil sonst.
SEITENEFFEKT: Uebersetzt die lambda-Definition von s_functionname in LISPLOG-Klauseln und gibt diese ueber den Port 'prt' aus.
BEACHTE: 1. Die Variable 'prt' hat als Wert einen gueltigen Port, oder nil, falls Ausgabe ueber den default-port gewuenscht wird.
2. Die globalen Variablen 'mind-func', 'mind-rel', 'still-func' und 'still-rel' muessen als Listen beliebigen Wertes instanziiert sein.

(transrelation 's_functionname)
AEQUIVALENT zu 'transfunction', wobei die lambda-Definition von s_functionname als LISP-Praedikat interpretiert wird.

3. Implementierung

3.1 Allgemeines

Wie schon in 1.5 erwaehnt, werden die Bedingungen und Aktionen einer Funktion in disjunktive Normalform umgewandelt. Jedes Disjunkt einer solchen DNF ergibt dann die rechte Seite einer Horn-Klausel, die Praemissen. Eine DNF hat folgende Form:

```
(or (and a1 a2 ... aN)
            .
            .
            .
    (and z1 z2 ... zM))
```

Alle ai ... zi sind Literale. Intern werden die Junktoren 'or' und 'and' weggelassen, so dass sich folgende Form ergibt:

```
((a1 a2 ... aN) ... (z1 z2 ... zM))
```

Wenn im folgenden von disjunktiven Normalformen die Rede ist, so haben diese die oben angegebene interne Form. Die DNF einer einzigen Relation 'a' hat die Form '((a))', da als disjunktive Normalformen immer Listen von Listen erwartet werden.

3.2 Hauptfunktionen

Top-level-Funktion ist, wie in 2. erwaehnt, die Funktion 'translate'.

(translate 'l_func 'l_praed 's_file)
: erzeugt fuer s_file eine Port und instanziiert die globalen Variablen mit den Werten von l_func und l_praed. Danach wird 'transglob' aufgerufen.

(transglob)
: ruft solange die Funktionen 'transrelation' bzw. 'transfunction' auf, bis die Listen 'still-rel' und 'still-func' leer sind. In diesem Fall wird der Port 'prt' geschlossen.

Wie schon erwaehnt, unterscheiden sich allgemeine Funktionen von Praedikaten dadurch, dass ihr Rumpf einen Wert hat, der mit dem zusaetzlichen Argument '$res' im Aufrufmuster unifiziert werden muss.

Hauptfunktion fuer die Uebersetzung einer allgemeinen LISP-Funktion ist die Funktion 'transfunction'.

(transfunction 's_name)
WERT: t bei erfolgreicher Uebersetzung, nil sonst.

Eine Uebersetzung ist moeglich, falls s_name eine Funktionsdefinition hat, die eine lambda-Funktion und keine Systemfunktion ist.
Aus dem Rumpf der Funktion koennen sich nach 1.5 mehrere Horn-Klauseln ergeben. Die Praemissenlisten der Horn-Klauseln werden von der Funktion 'transbody' berechnet. Durch die Funktion 'ausgabe' wird an jede Praemissenliste die Konklusion angefuegt und ausgedruckt.

LISP-Funktionen, die als Praedikate zu interpretieren sind, kommen als Goals in LISPLOG, als Bedingungen in 'cond'-Klauseln oder als top-level-Funktion im Rumpf von Praedikaten vor. Es ist also schon beim Auftreten der Funktion bekannt, ob sie als Praedikat zu interpretieren ist.

Hauptfunktion fuer die Uebersetzung von Praedikaten ist 'transrelation', die analog zu 'transfunction' arbeitet. Einzig die Funktion 'transaction-r' interpretiert den Rumpf abweichend von 'transaction'.

(transrelation 's_name)
WERT: t bei erfolgreicher Uebersetzung, nil sonst.

Die Funktion 'transrelation' arbeitet analog zu 'transfunction'. Die Variable 'relation' wird auf 't' gesetzt, damit in 'transbody' die Funktion 'transaction-r' aufgerufen werden kann.
Die Uebersetzung der Systemfunktion 'null' ist direkt angegeben, da sie beim Test auf non-nil-Wert gebraucht wird.

(transbody sl_action)
WERT: DNF des Funktionaufrufes sl_action.

Ist sl_action ein Aufruf der Funktion 'cond', so wird die DNF als Resultat von 'transcond' geliefert.
Ist die zu uebersetzende Funktion eine Relation (die Variable 'relation' ist in 'transrelation' auf 't' gesetzt), so wird sl_action von 'transaction-r' als Liste von Relationsaufrufen uebersetzt. Um die interne Form einer DNF zu erhalten, ist diese Liste das einzige Element einer Liste (vgl. 3.1).
Die Uebersetzung einer allgemeinen Funktion geschieht entsprechend durch 'transaction'.

(transcond 'l_neglist 'l_condcascade)
WERT: Liste von Praemissenlisten der 'cond'-Klauseln l_condcascade als DNF (s.o.).

ARGUMENTE: l_condcascade ist eine Liste von zu uebersetzenden Klauseln.
l_neglist ist die DNF der Negationen der schon uebersetzten Klauseln der 'cond'-Funktion (siehe 1.6).

Die Funktion ruft 'trclause' zur Uebersetzung der ersten 'cond'-Klausel auf. Die DNF dieser Uebersetzung wird zur Uebersetzung der folgenden Klauseln hinzugefuegt, die durch rekursiven Aufruf von 'transcond' uebersetzt werden. Dabei wird l_neglist von 'trclause' neu berechnet.

(trclause 'l_neglist 'l_condclause)
WERT: Liste mit zwei Elementen. Das erste ist die Uebersetzung der Klausel l_condclause zusammen mit l_neglist als DNF. Das zweite Element ist die DNF der Negationen l_neglist, erweitert um die Negation der Bedingung von l_condclause.

Die Bedingung der Klausel wird in DNF ueberfuehrt. Hat diese DNF mehrere Disjunkte, so muss zu jedem Disjunkt die Negation der vorherigen Disjunkte berechnet werden (siehe 1.6). Man erhaelt also eine neue DNF der Bedingung, die durch 'newcondlist' berechnet wird. Da sowohl l_neglist, als auch die Uebersetzung der Bedingung (berechnet von 'transcondition') und die Uebersetzung der Aktion (berechnet von 'transbody') gelten muessen, werden diese durch 'and' verknuepft und man erhaelt die Uebersetzung der Klausel als DNF.
Die negierten Bedingungen fuer die naechste cond-Klausel ist die DNF l_neglist 'and' die Negation der Bedingung von l_condclause.

(transcondition 'l_conjunction)
WERT: Liste mit den abgeflachten Formen von l_conjunction, wobei l_conjunction eine Liste von LISP-Funktionsaufrufen ist, die als Praedikate zu interpretieren sind.

(transaction 'sl_term)
WERT: Liste mit der abgeflachten Form von sl_term als Folge von Relationsaufrufen und Unifikation des Resultates mit '$res'.

Der Wert von sl_term ist das Resultat der zu uebersetzenden Funktion.
Ist sl_term ein Atom, so wird das Argument '$res' der Konklusion mit dem Wert von sl_term unifiziert.
Ist sl_term kein Atom, sondern ein Funktionsaufruf, so wird dieser abgeflacht und das Resultat mit '$res' unifiziert.

(transaction*p 'sl_term)
WERT: Liste mit der abgeflachten Form von sl_term und Test

des Resultates auf non-nil-Wert.

Ist sl_term gleich 'nil', so fuehrt das Praedikat zu 'failure', also 'nil'.
Ist sl_term gleich 't' oder eine Zahl, so fuehrt es zu 'success', braucht also nicht uebersetzt zu werden, d.h. die Ergebnisliste ist leer.
Ist sl_term ein anderes Atom, so wird es zu einer Variablen uebersetzt und sein Wert durch die Praemisse '(not (null*p term))' auf non-nil-Wert getestet.
Ist sl_term ein Funktionsaufruf, so wird dieser als Praedikatsaufruf durch 'opt-flattenrel' abgeflacht.

(ausgabe 'l_premiselist)
WERT: t

Fuegt in jedes Element von l_premiselist die Konklusion ein und druckt dieses dann als LISPLOG-Klausel ueber den Port 'prt' aus. Dazu verwendet sie die Ausgabefunktionen des LISPLOG-Systems.

3.3 Optimierung

Der Optimierungsteil wird durch die Funktion 'optimize' aufgerufen.

(optimize 'l_conclusion 'l_premiselist)
WERT: optimierte Klausel gemaess 1.7.

Ruft die Funktionen 'shorten' und 'del' auf.

(shorten 'l_conclusion 'l_premiselist-left 'l_premiselist-right)
WERT: Klausel, die keine Praemisse doppelt enthaelt und die gemaess 1.7 a) optimiert ist.

Die Funktion hat 3 Argumente, naemlich die Konklusion, die schon bearbeiteten und die noch zu bearbeitenden Praemissen. Die schon bearbeiteten Praemissen muessen mitgefuehrt werden, weil die Aenderungen in allen Termen einer Klausel durchgefuehrt werden muessen.

(del 'l_clause)
WERT: Klausel l_clause ohne die Praemissen, die sofort zu 'success' fuehren gemaess 1.7 b).

Da nur Klauseln in den Praemissen geloescht werden duerfen, ruft 'del' die Funktion 'dele', die die Loeschungen durchfuehrt, mit den Praemissen als Argument auf.

(dele 'l_premiselist)
WERT: l_premiselist ohne die Klauseln, die sofort zu 'success' fuehren gemaess 1.7 b).

3.4 Aufloesung von and-or-Bedingungen

(transform-dnf 'sl_condition)
WERT: DNF von sl_condition in der oben angegebenen Form.

Ist sl_condition ein Aufruf einer 'and'- 'or'- oder 'not'-Funktion, so werden die Argumente von 'transform-dnf' rekursiv in disjunktive Normalformen umgewandelt und dann durch die entsprechenden Funktionen verknuepft.

(or-connect 'l_disdiscon)
WERT: DNF von l_disdiscon

l_disdiscon ist eine Liste disjunktiver Normalformen, die durch 'or' verknuepft werden sollen. Dies geschieht, indem man das Assoziativgesetz anwendet.

Beispiel:

```
(or (or (and a1 a2) (and b1 b2))
    (or (and c1 c2) (and d1 d2)))

--> (or (and a1 a2) (and b1 b2) (and c1 c2) (and d1 d2))
```

oder in der hier gewaehlten Darstellung:

```
(((a1 a2) (b1 b2)) ((c1 c2) (d1 d2)))

--> ((a1 a2) (b1 b2) (c1 c2) (d1 d2))
```

(and-connect 'l_discon)
WERT: DNF von l_discon

l_discon ist eine Liste disjunktiver Normalformen, die durch 'and' verknuepft werden sollen. Dazu werden die Konjunkte ausmultipliziert.

Beispiel:

```
(and (or (and a1 a2) (and b1 b2))
     (or (and c1 c2) (and d1 d2)))

--> (or (and a1 a2 c1 c2)
        (and a1 a2 d1 d2)
```

```
            (and b1 b2 c1 c2)
            (and b1 b2 d1 d2))
```

oder in der hier gewaehlten Form:

```
    (((a1 a2) (b1 b2)) ((c1 c2) (d1 d2)))

--> ((a1 a2 c1 c2) (a1 a2 d1 d2) (b1 b2 c1 c2) (c1 c2 d1 d2))
```

(not-treatm 'l_neg)
WERT: DNF der Negation l_neg

l_neg ist von der Form '(not x)'.
Ist x ein Atom, so wird die DNF durch eine Doppelliste hergestellt (siehe 3.1).
Ist x eine Konjunktion oder Disjunktion, so werden die De Morgan-Regeln angewendet, um die Negation nach innen zu ziehen. Aus diesem Ergebnis wird die DNF gebildet.
Ist l_neg eine doppelte Negation so wird die DNF vom Argument des inneren not-Aufrufs gebildet.

(demorgan 'l_neg)
WERT: Umformung von l_neg nach den Regeln von De Morgan.

(negation 'sl_expr)
WERT: Negation des Ausdrucks sl_expr.

3.5 Hilfsfunktionen

(car-construct 'sl_term)
WERT: Punktpaar mit sl_term als erstem und einer neuen Variablen als zweitem Element.

(cdr-construct 'sl_term)
WERT: Punktpaar mit einer neuen Variablen als erstem und sl_term als zweitem Element.

(create-variables 'sl_term)
WERT: sl_term, wobei literale Atome zu Variablen geaendert werden.
Beispiel: (member 4 l) --> (member 4 (? l))

(subst-any-level 'g_a 'g_b 'sl_liste)
WERT: sl_liste, wobei g_b auf jedem Level durch g_a ersetzt wurde.

Die folgenden Funktionen berechnen die DNF einer 'cond'-Bedingung, bei der der 'cut' nach 1.6 b) durch Negation der

vorherigen Bedingungen vermieden wird.

(newcondlist 'l_condlist)
WERT: DNF von l_condlist nach 1.6 b).

(negate-and-append 'l_negativum 'l_condlist)
WERT: Eine Liste disjunktiver Normalformen.
ARGUMENTE: l_condlist enthaelt die Disjunkte, zu denen noch die Negationen berechnet werden muessen.
l_negativum enthaelt diejenigen Disjunkte, die fuer das erste Element von l_condlist zu negieren sind.

Die Funktion 'negate-and-append' negiert l_negativum und haengt an jedes Disjunkt das erste Element von l_condlist an (Schritt (3) aus 1.6 b)). Vor der Negation werden die nach Anmerkung 1. aus 1.6 b) ueberfluessigen Bedingungen aus l_negativum geloescht. Mit dem Rest von l_condlist und dem erweiterten l_negativum ruft sie sich rekursiv auf.

(negate 'l_dnf)
WERT: Negation der DNF l_dnf als DNF.

Die Funktionen cnf-to-dnf und neg-elements liefern die negierte Form von l_dnf. Daraus werden die nach den Anmerkungen 2. und 3. aus 1.6 b) ueberfluessigen Elemente geloescht.

(cnf-to-dnf 'l_cnf)
WERT: DNF der konjunktiven Normalform l_cnf.

Das Argument l_cnf ist eine Liste von Listen, die hier als konjunktive Normalform zu interpretieren ist. Durch Ausmultiplizieren wird daraus eine disjunktive Normalform berechnet (Schritt (2) aus 1.6 b)).

(neg-elements 'l_dnf)
WERT: Negation von l_dnf als konjunktive Normalform.

Aus einer DNF erhaelt man die Negation, indem man die Konjunkte negiert und die Junktoren 'and' und 'or' vertauscht (Schritt (1) aus 1.6 b)). Der Wert der Funktion ist eine Liste von Listen, die aber von der Funktion 'cnf-to-dnf' als konjunktive Normalform interpretiert wird.

(delete-all 'l_condition 'l_condlist)
WERT: l_condlist ohne die in l_condition vorkommenden Elemente.

l_condlist ist eine DNF. Die Elemente von l_condition werden in jedem Disjunkt von l_condlist geloescht.

(without-equals 'l_dnf)
WERT: l_dnf, wobei die nach 1.6 b) ueberfluessigen Disjunkte und Konjunkte geloescht wurden.

(remequals 'l_list)
WERT: l_list, wobei doppelt vorkommende Elemente geloescht sind.

(remequalsets 'l_list)
WERT: l_list, wobei doppelt vorkommende Mengen geloescht wurden.

l_list ist eine Liste von Listen. Die Elemente von l_list werden wie Mengen behandelt, d.h. zwei Mengen sind gleich, wenn sie die gleichen Elemente enthalten. Die Reihenfolge der Elemente kann allerdings verschieden sein.

(element 'sl_a 'l_list)
WERT: t, falls sl_a in l_list enthalten ist, nil sonst.

l_list ist eine Liste, deren Elemente als Mengen betrachtet werden.

(eqsets 'l_m1 'l_m2)
WERT: t, falls l_m1 und l_m2 die gleichen Mengen darstellen, nil sonst.

(result 'l_conjunction)
WERT: Das Argument der letzten Relation einer Konjunktion von Relationen, das das Resultat einer Aktion aufnimmt.

(flat-rel 'l_liste)
WERT: Liste der abgeflachten Relationsaufrufe in l_liste.

4. Weiterfuehrende Ueberlegungen

4.1 Globale Variablen

Ein Problem sind die in LISP erlaubten globalen Variablen, die es in PROLOG ja nicht gibt. Tritt in einer PROLOG-Klausel ein Variablenname auf, der schon in einer 'aufrufenden' Klausel auftrat, so ist das fuer den Interpreter eine neue Variable. In LISPLOG wird dies durch den level der Variablen deutlich gemacht.

4.2 Konsistenzbedingung bei Wertzuweisungen

LISP's 'set' und 'setq' ueberschreiben Werte von Variablen. In PROLOG koennen Werte nicht ueberschrieben werden. Der PROLOG-Operator 'is', der Variablen Werte zuweisen kann, unifiziert zwei Werte, falls die Variable auf der linken Seite von 'is' instanziiert ist.

4.3 Prozeduraufrufe mit gebundenen und freien Variablen

Die aus LISP-Funktionen erhaltenen LISPLOG-Praedikate der Form (f*r a1 a2 ... aN-1 aN) sollten nur mit Argumenten aufgerufen werden, die soweit instanziiert sind, dass LISPLOG-Variablen nur in gequoteten Kontexten vorkommen (s.o.). Dies ist fuer die zu uebersetzenden LISPLOG-Quellprogramme gegeben, da diese beim Aufruf der LISP-Funktion ja auch die Argumente soweit instanziiert haben muessen, wenn der Wert berechnet werden soll.

Normalerweise werden die generierten LISPLOG-Relationen nicht von Hand aufgerufen. Wenn ein Anwender eines uebersetzten Programmes dies jedoch versucht, sollte er auf folgendes achten:

Zu einer LISP-Funktion mit dem Aufrufmuster (f x y) erhaelt man die LISPLOG-Konklusion '(f*r _x _y _$res)'. Man koennte also, syntaktisch gesehen, auch Anfragen der Form '(f*r _u _v 5)' mit freien Variablen _u und _v stellen. Da aber die LISPLOG-Uebersetzung nur auf die funktionale Verwendungsweise der LISP-Funktionen abzielt, fuehren solche Aufrufe im allgemeinen nicht zu sinnvollen Ergebnissen.

Auch Anfragen der Form '(f*r _u 3 7)' oder '(f*r _u 3 _z)' mit freien Variablen _u und _z sind syntaktisch moeglich, koennen aber zu falschen Ergebnissen oder gar zu Endlosschleifen fuehren.

Literatur:

[Boley & Kammermeier et al. 1985]
Harold Boley, Franz Kammermeier und die LISPLOG-Gruppe: LISPLOG - Momentaufnahmen einer LISP/PROLOG-Vereinheitlichung, Universitaet Kaiserslautern, Fachbereich Informatik, MEMO-SEKI-85-03, August 1985

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

[Foderaro et al. 1983]
J.K. Foderaro, K.L. Sklower, K. Laver: The FRANZ LISP Manual, University of California, Juni 1983

[Gray 1984]
Peter M.D. Gray: Logic, Algebra and Databases, Ellis Horwood Ltd., 1984

[Kowalski 1983]
Robert Kowalski: Logic Programming in Proc. IFIP, pp. 133-145, Amsterdam, North Holland

[Kowalski 1985]
Robert Kowalski: The Relation between Logic Programming and Logic Specification, in C.A.R. Hoare, J.L. Shepherdson: Mathematical Logic and Programming Languages, Prentice/Hall International, 1985

[Hinkelmann 1985]
Knut Hinkelmann: LISPLOG-CPROLOG-Translator, in [Boley & Kammermeier et al. 1985]

Anhang A: Listings

Teil I

Optimierendes Abflachen von LISP-Funktionsschachtelungen

```
(setq flattenfns
      '(begin insert-arg
              replace-arith-resvar
              append-and-insert-resvar-1
              insert-is-resvar
              append-arith-flatten-lists
              append-and-insert-resvar-2
              insert-arith-arg
              append-and-insert-arith-resvar
              insert-not
              replace-resvar
              flatten-arithfunc
              arith-func
              arith-func-sym
              arith-functions
              create-arith-var
              create-var
              opt-flattenfunc
              opt-flattenrel
              flattenfunc
              flattenrel
              mind-func
              mind-rel
              still-func
              still-rel
              laast-l
              last-l
              make-uniform
              member-l
              delete-member
              delete-notarith-member
              delete-arith-member
              newn
              newrel
              not-arith-func-sym
              opt
              quote-sym
```



```
              replace
              search
              variable-p))

(def begin
  (lambda (list)
    (reverse (cdr (reverse list)))))


(def insert-arg
  (lambda (elem list)
    (append (begin list) (list (cons elem (last-l list))))))


(def replace-arith-resvar
  (lambda (arith-resvar list)
    (append (begin list)
            (list
             (append (begin (last-l list))
                     (list arith-resvar))))))


(def append-and-insert-resvar-1
  (lambda (listl list2)
    (append listl
            (append (begin list2)
                    (list
                     (cons (laast-l listl) (last-l list2)))))))


(def insert-is-resvar
  (lambda (isresvar list)
    (cond ((equal (length list) 1)
           (list (append isresvar list)))
          (t
           (append (car list)
                   (list (append isresvar (last list))))))))


(def append-arith-flatten-lists
  (lambda (listl list2)
    (cond ((equal (length listl) 1)
           (cond ((equal (length list2) 1)
                  (list (cons (car listl) (car list2))))
                 (t
                  (cons (car list2)
                        (list
                         (cons (car listl) (cadr list2)))))))
          (t
           (cond ((equal (length list2) 1)
                  (cons (car listl)
                        (list
                         (cons (cadr listl) (car list2)))))
```

```
                (t
                 (cons (append (car listl) (car list2))
                       (list
                        (cons (cadr listl) (cadr list2))))))))))


(def append-and-insert-resvar-2
  (lambda (listl list2)
    (cond ((equal (length list2) 1)
           (cons listl
                 (list (cons (laast-l listl) (car list2)))))
          (t
           (cons (append listl (car list2))
                 (list (cons (laast-l listl) (cadr list2))))))))


(def insert-arith-arg
  (lambda (aritharg list)
    (cond ((equal (length list) 1)
           (list (cons (make-uniform aritharg) (car list))))
          (t
           (list (car list)
                 (cons (make-uniform aritharg) (cadr list)))))))


(def append-and-insert-arith-resvar
  (lambda (listl list2)
    (append listl
            (append (begin list2)
                    (list
                     (cons (cadr (last-l listl))
                           (last-l list2)))))))


(def insert-not
  (lambda (sym list)
    (append (begin list) (list (cons sym (last list))))))


(def replace-resvar
  (lambda (resvar list)
    (append (begin list)
            (list
             (cons 'is
                   (cons resvar (cddr (last-l list))))))))


(def flatten-arithfunc
  (lambda (list)
    (cond ((null list) (list nil))
          ((and (not (variable-p (car list)))
                (listp (car list))
                (arith-func-sym (caar list)))
```



```
           (append-arith-flatten-lists (flatten-arithfunc
                                        (car list))
                                       (flatten-arithfunc
                                        (cdr list))))
          ((and (not (variable-p (car list)))
                (listp (car list))
                (not-arith-func-sym (caar list)))
           (append-and-insert-resvar-2 (flattenfunc (car list))
                                       (flatten-arithfunc
                                        (cdr list))))
          (t
           (insert-arith-arg (car list)
                             (flatten-arithfunc (cdr list)))))))


(def arith-func
  (lambda (list)
    (insert-is-resvar (create-arith-var)
                      (flatten-arithfunc list))))


(def arith-func-sym
  (lambda (sym)
    (and (symbolp sym) (getd sym) (member sym arith-functions))))


(setq arith-functions
      '(plus times
             divide
             add
             +
             sum
             diff
             difference
             -
             product
             *
             quotient
             /
             mod
             remainder
             addl
             |1+|
             subl
             |1-|
             acos
             asin
             atan
             cos
             exp
             log
             sin
             sqrt))
```

```
(def create-arith-var
  (lambda nil
    (list 'is
          (list '?
                (concat 'res
                        (implode (cdr (explode (gensym))))))))))


(def create-var
  (lambda nil
    (list
     (list
      (list '?
            (concat 'res
                    (implode (cdr (explode (gensym))))))))))


(def opt-flattenfunc
  (lambda (list)
    (cond ((equal (car list) 'is)
           (cond ((member (car (last-l list)) arith-functions)
                  (replace-resvar (cadr list)
                                  (opt
                                   (flattenfunc
                                    (caddr list)))))
                 (t
                  (replace-arith-resvar (cadr list)
                                        (opt
                                         (flattenfunc
                                          (caddr list)))))))
          (t (opt (flattenfunc list))))))


(def opt-flattenrel
  (lambda (list)
    (opt (flattenrel list))))


(def flattenfunc
  (lambda (list)
    (cond ((null list) (create-var))
          ((variable-p (car list))
           (insert-arg (car list) (flattenfunc (cdr list))))
          ((quote-sym (car list))
           (cond
            ((not (member 'quote mind-func))
             (setq mind-func (cons 'quote mind-func))
             (setq still-func
                   (appendl still-func 'quote))))
           (list
            (append (cons 'quote*r (cdr list))
                    (car (create-var)))))
          ((arith-func-sym (car list)) (arith-func list))
```



```
          ((not-arith-func-sym (car list))
           (cond
            ((not (member (car list) mind-func))
             (setq mind-func (cons (car list) mind-func))
             (setq still-func (appendl still-func (car list)))))
           (insert-arg (newn (car list))
                       (flattenfunc (cdr list))))
          ((and (listp (car list)) (arith-func-sym (caar list)))
           (append-and-insert-arith-resvar (arith-func
                                            (car list))
                                           (flattenfunc
                                            (cdr list))))
          ((and (listp (car list))
                (not-arith-func-sym (caar list)))
           (append-and-insert-resvar-l (flattenfunc (car list))
                                       (flattenfunc
                                        (cdr list))))
          (t (insert-arg (car list) (flattenfunc (cdr list))))))


(def flattenrel
  (lambda (list)
    (cond ((null list) (list nil))
          ((variable-p (car list))
           (insert-arg (car list) (flattenrel (cdr list))))
          ((eq (car list) 'not)
           (insert-not 'not (flattenrel (cadr list))))
          ((and (symbolp (car list)) (getd (car list)))
           (cond
            ((not (member (car list) mind-rel))
             (setq mind-rel (cons (car list) mind-rel))
             (setq still-rel (appendl still-rel (car list)))))
           (insert-arg (newrel (car list))
                       (flattenrel (cdr list))))
          ((and (listp (car list)) (arith-func-sym (caar list)))
           (append-and-insert-arith-resvar (arith-func
                                            (car list))
                                           (flattenrel
                                            (cdr list))))
          ((and (listp (car list))
                (not-arith-func-sym (caar list)))
           (append-and-insert-resvar-l (flattenfunc (car list))
                                       (flattenrel (cdr list))))
          (t (insert-arg (car list) (flattenrel (cdr list))))))


(setq mind-func 'nil)

(setq mind-rel 'nil)

(setq still-func 'nil)

(setq still-rel 'nil)
```

```
(def laast-l
  (lambda (list)
    (last-l (last-l list))))


(def last-l
  (lambda (list)
    (car (last list))))


(def make-uniform
  (lambda (x)
    (cond ((not (member x arith-functions)) x)
          ((or (equal x 'add)
               (equal x '+)
               (equal x 'sum))
           'plus)
          ((or (equal x '*) (equal x 'product))
           'times)
          ((or (equal x 'difference) (equal x '-))
           'diff)
          ((or (equal x 'quotient) (equal x '/))
           'divide)
          ((equal x 'remainder) 'mod)
          ((equal x '|1+|) 'add1)
          ((equal x '|1-|) 'sub1)
          (t x))))


(def member-l
  (lambda (x list)
    (cond ((null list) nil)
          ((and (listp (car list))
                (not (variable-p (car list))))
           (or (member-l x (car list)) (member-l x (cdr list))))
          (t (or (equal x (car list)) (member-l x (cdr list)))))))


(def delete-member
  (lambda (list1 list2)
    (cond ((equal (car list1) 'is)
           (delete-arith-member list1 list2))
          (t (delete-notarith-member list1 list2)))))


(def delete-notarith-member
  (lambda (list1 list2)
    (cond ((null list2) nil)
          ((equal (begin list1) (begin (car list2)))
           (delete-notarith-member list1
                                   (search (last-l
                                            (car list2))
                                           (last-l list1)
```



```
                                           (cdr list2))))
          (t
           (cons (car list2)
                 (delete-notarith-member list1 (cdr list2)))))))


(def delete-arith-member
  (lambda (list1 list2)
    (cond ((null list2) nil)
          ((and (equal (caar list2) 'is)
                (equal (caddr list1) (caddar list2)))
           (delete-arith-member list1
                                (search (cadar list2)
                                        (cadr list1)
                                        (cdr list2))))
          (t
           (cons (car list2)
                 (delete-arith-member list1 (cdr list2)))))))


(def newn
  (lambda (funcname)
    (concat funcname '*r)))


(def newrel
  (lambda (relname)
    (cond ((member relname
                   '(eq equal
                        greaterp
                        >
                        >&
                        lessp
                        <
                        <&
                        =
                        =&
                        litatom
                        atom
                        numberp))
           relname)
          (t (concat relname '*p)))))


(def not-arith-func-sym
  (lambda (sym)
    (and (symbolp sym)
         (getd sym)
         (not (member sym arith-functions)))))


(def opt
  (lambda (list)
```

```
    (cond ((null list) nil)
          (t
           (cons (car list)
                 (opt (delete-member (car list) (cdr list)))))))))
```

```
(def quote-sym
  (lambda (sym)
    (and (symbolp sym) (equal sym 'quote))))
```

```
(def replace
  (lambda (x y list)
    (cond ((null list) nil)
          ((and (listp (car list))
                (not (variable-p (car list))))
           (cons (replace x y (car list))
                 (replace x y (cdr list))))
          ((equal x (car list)) (cons y (cdr list)))
          (t (cons (car list) (replace x y (cdr list)))))))
```

```
(def search
  (lambda (x y list)
    (cond ((null list) nil)
          (t
           (cond ((member-l x (car list))
                  (cons (replace x y (car list)) (cdr list)))
                 (t
                  (cons (car list) (search x y (cdr list))))))))))
```


Teil II

Umwandlung von lambda-Definitionen zu Horn-Klauseln

```
; Hauptfunktionen:
;-----------------
```

```
(setq trans.llfns
      '(translate transglob
                  transfunction
                  transrelation
                  transbody
                  transcond
                  trclause
                  transcondition
                  transaction
                  transaction-r
                  ausgabe))
```

```
(def translate
  (lambda (func rel file)
    (let ((prt (outfile file))
          (still-func func)
          (still-rel (cons 'null rel))
          (mind-func (append '(car cdr cons) func))
          (mind-rel (append '(null equal eq) rel)))
         (transglob))))
```

```
(def transglob
  (lambda nil
    (cond ((and (null still-rel) (null still-func)) (close prt))
          ((null still-func)
           (transrelation (first still-rel))
           (terpr prt)
           (terpr prt)
           (setq still-rel (rest still-rel))
           (transglob))
          (t (transfunction (first still-func))
             (terpr prt)
             (terpr prt)
             (setq still-func (rest still-func))
```

```
            (transglob)))))


(def transfunction
  (lambda (oldname)
    (let ((definition (getd oldname)))
         (setq relation nil)
         (cond ((eq oldname 'quote)
                (pp-external-form
                 '(ass (quote*r (? x) (? x)))))
               ((or (not (dtpr definition))
                    (member (first definition)
                            '(nlambda macro
                                      lexpr)
                             ))
                (terpr)
                (patom
                 (uconcat '"Uebersetzung der Funktion '"
                          oldname
                          '"' nicht moeglich"))
                nil)
               (t
                (ausgabe (cons (newn oldname)
                               (appendl (mapcar (function
                                                 create-variables)
                                                (second
                                                 definition))
                                        '(? $res)))
                         (transbody (third definition))))))))


(def transrelation
  (lambda (oldname)
    (let ((definition (getd oldname)))
         (setq relation t)
         (cond ((eq oldname 'null)
                (pp-external-form '(ass (null*p nil))
                                  prt)
                t)
               ((or (not (dtpr definition))
                    (member (first definition)
                            '(nlambda macro
                                      lexpr)
                             ))
                (terpr)
                (patom
                 (uconcat '"Uebersetzung der Funktion '"
                          oldname
                          '"' nicht moeglich"))
                nil)
               (t
                (ausgabe (cons (newrel oldname)
                               (mapcar (function
```



```
                                        create-variables)
                                       (second definition)))
                         (transbody (third definition)))))))))


(def transbody
  (lambda (action)
    (cond ((and (listp action) (eq (first action) 'cond))
           (transcond nil (rest action)))
          (relation (transaction-r action))
          (t (list (transaction action))))))


(def transcond
  (lambda (neglist condcascade)
    (cond ((null condcascade) nil)
          (t
           ((lambda (x)
                    (append (first x)
                            (transcond (second x)
                                       (rest condcascade))))

            (trclause neglist (first condcascade)))))))


(def trclause
  (lambda (neglist condclause)
    (let ((condlist (transform-dnf (first condclause)))
          (actionlist (transbody (second condclause))))
         (list (and-connect
                (list neglist
                      (mapcar (function transcondition)
                              (newcondlist condlist))
                      actionlist))
               (and-connect
                (list neglist
                      (mapcar (function transcondition)
                              (negate condlist)))))))))


(def transcondition
  (lambda (conjunction)
    (cond ((null conjunction) nil)
          ((eq (first conjunction) t)
           (cons t (transcondition (rest conjunction))))
          ((atom (first conjunction))
           (cons (create-variables (first conjunction))
                 (transcondition (rest conjunction))))
          (t
           (append (opt-flattenrel
                    (create-variables (first conjunction)))
                   (transcondition (rest conjunction)))))))
```

```
(def transaction
  (lambda (term)
    (cond ((null term)
           (list (list 'null*p '(? $res))))
          ((atom term)
           (list
            (cons '=
                  (list '(? $res)
                        (create-variables term)))))
          ((and (symbolp (first term)) (getd (first term)))
           ((lambda (action)
                    (appendl action
                             (cons '=
                                   (cons '(? $res)
                                         (result action)))))

            (opt-flattenfunc (create-variables term)))))))


(def transaction-r
  (lambda (term)
    (cond ((null term) (transform-dnf nil))
          ((eq term t) (list nil))
          ((numberp term) (list nil))
          ((atom term)
           (transform-dnf
            (list 'not
                  (list 'null*p
                        (create-variables term)))))
          ((and (symbolp (first term)) (getd (first term)))
           (mapcar (function flat-rel)
                   (transform-dnf (create-variables term)))))))


(def ausgabe
  (lambda (conclusion premiselists)
    (cond ((null premiselists) t)
          (t (pp-external-form (cons 'ass
                                     (optimize conclusion
                                               (first
                                                premiselists)))
                               prt)
             (ausgabe conclusion (rest premiselists))))))
```



```
;Optimierung:
;------------



(setq optimizationfns
      '(optimize shorten
                 del
                 dele
                 subst-any-level
                 car-treat
                 cdr-treat))

(def optimize
  (lambda (conclusion premiselist)
    (del (shorten (list conclusion nil premiselist)))))


(def shorten
  (lambda (clause)
    (let ((conclusion (first clause))
          (premiselist-left (second clause))
          (premiselist-right (third clause))
          (premise (first (third clause))))
         (cond ((null premiselist-right)
                (cons conclusion premiselist-left))
               ((member premise (rest premiselist-right))
                (shorten
                 (list conclusion
                       premiselist-left
                       (rest premiselist-right))))
               ((atom (first premiselist-right))
                (shorten
                 (list conclusion
                       (appendl premiselist-left
                                (first premiselist-right))
                       (rest premiselist-right))))
               (t
                (let ((functor (first premise)))
                     (cond ((and (eq functor 'null*p)
                                 (member (second premise)
                                         conclusion))
                            (shorten
                             (subst-any-level
                              nil
                              (second premise)
                              (list conclusion
                                    premiselist-left
                                    (rest premiselist-right)))))
                           ((and (eq functor 'zerop*p)
                                 (member (second premise)
                                         conclusion))
```

```
 (shorten
  (subst-any-level
   0
   (second premise)
   (list conclusion
         premiselist-left
         (rest premiselist-right)))))
((and (memq functor
            '(eq equal = =&))
      (variable-p
       (second premise))
      (member (second premise)
              conclusion))
 (shorten
  (subst-any-level
   (third premise)
   (second premise)
   (list conclusion
         premiselist-left
         (rest premiselist-right)))))
((and (memq functor
            '(eq equal = =&))
      (variable-p
       (third premise))
      (member (third premise)
              conclusion))
 (shorten
  (subst-any-level
   (second premise)
   (third premise)
   (list conclusion
         premiselist-left
         (rest premiselist-right)))))
((eq functor 'car*r)
 (car-treat (second premise)
            (third premise)))
((eq functor 'cdr*r)
 (cdr-treat (second premise)
            (third premise)))
((eq functor 'cons*r)
 (shorten
  (subst-any-level
   (cons (second premise)
         (third premise))
   (fourth premise)
   (list conclusion
         premiselist-left
         (rest premiselist-right)))))
(t
 (shorten
  (list conclusion
        (appendl premiselist-left
                 premise)
```



```
                                 (rest
                                  premiselist-right))))))))))))


(def del
  (lambda (clause)
    (cons (first clause) (dele (rest clause)))))


(def dele
  (lambda (premiselist)
    (let ((premise (first premiselist)))
         (cond ((null premiselist) nil)
               ((eq premise t) (dele (rest premiselist)))
               ((atom premise)
                (cons premise (dele (rest premiselist))))
               ((and (eq (first premise) 'null*p)
                     (null (second premise)))
                (dele (rest premiselist)))
               ((and (eq (first premise) 'atom)
                     (atom (second premise)))
                (dele (rest premiselist)))
               ((and (eq (first premise) 'zerop*p)
                     (zerop (second premise)))
                (dele (rest premiselist)))
               ((and (memq (first premise) '(eq equal = =&))
                     (atom (second premise))
                     (atom (third premise))
                     (eq (first premise) (second premise)))
                (dele (rest premiselist)))
               ((eq (first premise) 'not)
                (cond ((and (eq (first (second premise))
                                'null*p)
                            (not
                             (variable-p
                              (second (second premise))))
                            (not
                             (null
                              (second (second premise)))))
                       (dele (rest premiselist)))
                      ((and (eq (first (second premise))
                                'atom)
                            (not
                             (variable-p
                              (second (second premise))))
                            (not
                             (atom
                              (second (second premise)))))
                       (dele (rest premiselist)))
                      ((and (eq (first (second premise))
                                'zerop*p)
                            (numberp
                             (second (second premise)))
```

```
                          (not
                           (zerop
                            (second (second premise)))))
                     (dele (rest premiselist)))
                    ((and (memq (first premise)
                                '(eq equal = =&))
                          (atom (second premise))
                          (atom (third premise))
                          (not (eq (second premise)
                                   (third premise))))
                     (dele (rest premiselist)))
                    (t
                     (cons premise
                           (dele (rest premiselist))))))
            (t (cons premise (dele (rest premiselist))))))))


(def subst-any-level
  (lambda (a b liste)
    (cond ((null liste) nil)
          ((equal liste b) a)
          ((variable-p liste) liste)
          ((listp liste)
           (cons (subst-any-level a b (first liste))
                 (subst-any-level a b (rest liste))))
          (t liste))))


(def car-treat
  (lambda (arg1 arg2)
    (cond ((variable-p arg1)
           (shorten
            (subst-any-level (car-construct arg2)
                             arg1
                             (list conclusion
                                   premiselist-left
                                   (rest premiselist-right)))))
          ((listp arg1)
           (shorten
            (subst-any-level (first arg1)
                             arg2
                             (list conclusion
                                   premiselist-left
                                   (rest premiselist-right))))))))


(def cdr-treat
  (lambda (arg1 arg2)
    (cond ((variable-p arg1)
           (shorten
            (subst-any-level (cdr-construct arg2)
                             arg1
                             (list conclusion
```



```
                                   premiselist-left
                                   (rest premiselist-right)))))
          ((listp arg1)
           (shorten
            (subst-any-level (rest arg1)
                             arg2
                             (list conclusion
                                   premiselist-left
                                   (rest premiselist-right))))))))
```

```
;Aufloesung von and-or-Bedingungen:
;----------------------------------



(setq dnffns
      '(and-connect transform-dnf
                    demorgan
                    negation
                    not-treatm
                    or-connect))

(def and-connect
  (lambda (discon)
    (cond ((null (rest discon)) (first discon))
          ((null (rest (first discon)))
           (mapcar (function
                     (lambda (x)
                             (append (first (first discon)) x))
                     )
                   (and-connect (rest discon))))
          (t
           (append (mapcar (function
                             (lambda (x)
                                     (append (first
                                               (first discon))
                                             x))
                             )
                           (and-connect (rest discon)))
                   (and-connect
                    (cons (rest (first discon))
                          (rest discon)))))))))


(def transform-dnf
  (lambda (condition)
    (cond ((atom condition) (list (list condition)))
          ((eq (first condition) 'or)
           (or-connect
            (mapcar (function transform-dnf) (rest condition))))
          ((eq (first condition) 'and)
           (and-connect
            (mapcar (function transform-dnf) (rest condition))))
          ((eq (first condition) 'not)
           (not-treatm condition))
          (t (list (list condition))))))


(def demorgan
  (lambda (neg)
    (let ((arg (second neg)))
         (cond ((eq (first arg) 'and)
```



```
                (cons 'or
                      (mapcar (function negation) (rest arg))))
               ((eq (first arg) 'or)
                (cons 'and
                      (mapcar (function negation)
                              (rest arg))))))))


(def negation
  (lambda (expr)
    (cond ((null expr) t)
          ((eq expr t) nil)
          ((atom expr) (list 'not expr))
          ((eq (first expr) 'not) (second expr))
          (t (list 'not expr)))))


(def not-treatm
  (lambda (neg)
    (cond ((atom (second neg)) (list (list neg)))
          ((member (first (second neg)) '(and or))
           (transform-dnf (demorgan neg)))
          ((eq (first (second neg)) 'not)
           (transform-dnf (second (second neg))))
          (t (list (list neg))))))


(def or-connect
  (lambda (disdiscon)
    (cond ((null (rest disdiscon)) (first disdiscon))
          (t
           (append (first disdiscon)
                   (or-connect (rest disdiscon)))))))
```

```
;Hilfsfunktionen:
;----------------



(setq hilfsfkttfns
      '(newcondlist negate-and-append
                    negate
                    cnf-to-dnf
                    neg-elements
                    without-equals
                    remequals
                    remequalsets
                    eqsets
                    element
                    delete-all
                    car-construct
                    cdr-construct
                    create-variables
                    result
                    flat-rel))

(def newcondlist
  (lambda (condlist)
    (cond ((null (rest condlist)) condlist)
          (t (or-connect (negate-and-append nil condlist))))))


(def negate-and-append
  (lambda (negativum condlist)
    (cond ((null condlist) nil)
          ((null negativum)
           (cons (list (first condlist))
                 (negate-and-append (list (first condlist))
                                    (rest condlist))))
          (t
           (cons (mapcar (function
                          (lambda (x)
                                  (append x (first condlist)))
                          )
                         (negate
                          (delete-all (first condlist)
                                      negativum)))
                 (negate-and-append (appendl negativum
                                             (first
                                              condlist))
                                    (rest condlist)))))))


(def negate
  (lambda (dnf)
    (without-equals (cnf-to-dnf (neg-elements dnf)))))
```



```
(def cnf-to-dnf
  (lambda (cnf)
    (cond ((null (rest cnf))
           (mapcar (function list) (first cnf)))
          ((null (rest (first cnf)))
           (mapcar (function
                    (lambda (x)
                            (cons (first (first cnf)) x))
                    )
                   (cnf-to-dnf (rest cnf))))
          (t
           (append (mapcar (function
                            (lambda (x)
                                    (cons (first (first cnf))
                                          x))
                            )
                           (cnf-to-dnf (rest cnf)))
                   (cnf-to-dnf
                    (cons (rest (first cnf)) (rest cnf)))))))))


(def neg-elements
  (lambda (dnf)
    (cond ((null dnf) nil)
          (t
           (cons (mapcar (function negation) (first dnf))
                 (neg-elements (rest dnf)))))))


(def without-equals
  (lambda (list)
    (remequalsets (mapcar (function remequals) list))))


(def remequals
  (lambda (list)
    (cond ((null list) nil)
          ((member (first list) (rest list))
           (remequals (rest list)))
          (t (cons (first list) (remequals (rest list)))))))


(def remequalsets
  (lambda (list)
    (cond ((null list) nil)
          ((element (first list) (rest list))
           (remequalsets (rest list)))
          (t (cons (first list) (remequalsets (rest list)))))))


(def eqsets
  (lambda (m1 m2)
    (cond ((and (null m1) (null m2)) t)
```

```
          ((member (first m1) m2)
           (eqsets (rest m1) (remove (first m1) m2)))
          (t nil))))


(def element
  (lambda (a l)
    (cond ((null l) nil)
          ((eqsets a (first l)) t)
          (t (element a (rest l))))))


(def delete-all
  (lambda (condition condlist)
    (cond ((null condition) condlist)
          (t
           (delete-all (cdr condition)
                       (mapcar (function
                                (lambda (x)
                                        (remove (car
                                                 condition)
                                                x))
                                )
                               condlist))))))


(def car-construct
  (lambda (term)
    (cons (create-variables term)
          (create-variables (gensym 'x)))))


(def cdr-construct
  (lambda (term)
    (cons (create-variables (gensym 'x))
          (create-variables term))))


(def create-variables
  (lambda (term)
    (cond ((numberp term) term)
          ((eq term 't) t)
          ((null term) nil)
          ((variable-p term) term)
          ((and (listp term)
                (symbolp (first term))
                (getd (first term)))
           (cons (first term)
                 (mapcar (function create-variables)
                         (rest term))))
          ((listp term) term)
          (t (list '? term)))))
```



```
(def result
  (lambda (conjunction)
    (cond ((eq (first (first (last conjunction))) 'is)
           (list (second (first (last conjunction)))))
          (t (last (first (last conjunction)))))))


(def flat-rel
  (lambda (liste)
    (mapcan (function opt-flattenrel) liste)))
```

Anhang B: Beispiel

Es sollen die beiden folgenden LISPLOG-Klauseln mit LISP-Durchgriff abgeflacht werden:

```
(ass (predicate _liste _element 0)
     (not (member _element _liste)))

(ass (predicate _liste _element _wert)
     (is _wert (quotient (length _liste)
                         (length (member _element _liste)))))
```

Die Definitionen der darin auftretenden LISP-Funktionen lauten wie folgt:

```
(def member
  (lambda (a l)
    (cond ((null l) nil) ((equal a (car l)) l)
          (t (member a (cdr l))))))

(def length
  (lambda (l)
    (cond ((null l) 0)
          (t (plus 1 (length (cdr l)))))))
```

Uebersetzung der ersten Klausel:

Die Konklusion bleibt erhalten:

```
(predicate _liste _element 0)
```

Die Praemisse wird wie folgt abgeflacht:

```
(not (member*p _element _liste))
```

Die LISP-Funktion 'member' wird hier als Praedikat interpretiert, da sie als Argument eines Aufrufs der Funktion 'not' auftritt. Der Name 'member' muss zur Liste 'mind-rel' und 'still-rel' hinzugefuegt werden, weil die Funktion spaeter noch uebersetzt werden muss.

Uebersetzung der zweiten Klausel:

Die Konklusion bleibt erhalten:

```
(predicate _liste _element _wert)
```

Die Praemisse wird wie folgt in eine Liste von Relationsaufrufen abgeflacht:

```
((length*r _liste _res001)
 (member*r _element _liste _res002)
 (length*r _res002 _res003)
 (is _wert (divide _res001 _res003)))
```

Statt der Funktion 'quotient' wird wegen der Vereinfachung der Optimierung die aequivalente Funktion 'divide' aufgerufen. Die Funktion 'length' tritt als Argument eines Aufrufs der Funktion 'quotient' auf und ist deshalb als allgemeine Funktion zu interpretieren. In diesem Fall werden sowohl 'mind-func' als auch 'still-func' um den Namen 'length' erweitert. Die Funktion 'member' wird innerhalb eines Aufrufes von 'length' aufgerufen und ist deshalb ebenfalls als allgemeine Funktion aufzufassen. Auch ihr Name wird zu den Listen 'still-func' und 'mind-func' hinzugefuegt, da er in 'mind-func' noch nicht enthalten ist. Die Funktion 'length' wird auch bei ihrem zweiten Auftreten als allgemeine Funktion interpretiert. Ihr Name wird allerdings nicht nochmals zu 'still-func' und 'mind-func' hinzugefuegt, weil er in 'mind-func' schon enthalten ist.

Umwandlung der LISP-Funktionsdefintionen:

Zuerst werden die in 'still-func' enthaltenen Funktionen uebersetzt. Es sind dies 'length' und 'member'. Man erhaelt also folgende Klauseln:

```
(ass (member*r _a nil nil))
(ass (member*r _res004
               (_res004 . _x008)
               (_res004 . _x008)))
(ass (member*r _a (_res005 . _x009) _res007)
     (not (equal _a _res005))
     (member*r _a _x009 _res007))

(ass (length*r nil 0))
(ass (length*r (_x010 . _x011) _res012)
     (length*r _x011 _res13)
     (is _res012 (plus 1 _res13)))
```

Danach werden die in 'still-rel' enthaltenen Funktionen uebersetzt. Dies ist nur die Funktion 'member', die jetzt aber als

Praedikat zu interpretieren ist. Man erhaelt folgende Klauseln:

```
(ass (member*p _a nil) nil)
(ass (member*p _res014 (_res014 . _x017)))
(ass (member*p _a (_res015 . _x018))
     (not (equal _a _res015))
     (member*p _a _x018))
```

Ergebnis:
=========

Bei der Abflachung der LISPLOG-Klauseln

```
(ass (predicate _liste _element 0)
     (not (member _element _liste)))
(ass (predicate _liste _element _wert)
     (is _wert (quotient (length _liste)
                         (length (member _element _liste)))))
```

erhaelt man also folgendes Programmstueck:

```
(ass (predicate _liste _element 0)
     (not (member*p _element _liste)))
(ass (predicate _liste _element _wert)
     (length*r _liste _res001)
     (member*r _element _liste _res002)
     (length*r _res002 _res003)
     (is _wert (quotient _res001 _res003)))

(ass (member*r _a nil nil))
(ass (member*r _res004
               (_res004 . _x008)
               (_res004 . _x008)))
(ass (member*r _a (_res005 . _x009) _res007)
     (not (equal _a _res005))
     (member*r _a _x009 _res007))

(ass (length*r nil 0))
(ass (length*r (_x010 . _x011) _res012)
     (length*r _x011 _res13)
     (is _res012 (plus 1 _res13)))

(ass (member*p _a nil) nil)
(ass (member*p _res014 (_res014 . _x017)))
(ass (member*p _a (_res015 . _x018))
     (not (equal _a _res015))
     (member*p _a _x018))
```

Breitensuche und Klauselcompilation fuer LISPLOG

Juergen Herr

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

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

Projektarbeit
(Betreuer : Harold Boley)

November 1985

Inhalt

1. Einleitung

Diese Projektarbeit behandelt Erweiterungen zu der urspruenglichen LISPLOG.1-Version (vgl. [BO/KA , 1985]). Als erstes soll ein Breitensuchkonzept fuer LISPLOG entwickelt und implementiert werden. Dazu gehoert ein kurzer Vergleich von Breitensuche und Tiefensuche, welcher Vor- und Nachteile beider Verfahren fuer den praktischen Einsatz untersucht. Dies geschieht im wesentlichen in Kapitel 2.1. In den Kapiteln 2.2 und 2.3 wird dann ein Konzept zur breitenorientierten Suche und eine moegliche neue Definition des Cut-Operators entwickelt. Die Implementierung dieser Konzepte, mit einigen pragmatischen Abstrichen, wird dann in Kapitel 2.4 beschrieben.

Der zweite, umfangreichere Teil dieser Arbeit behandelt das Thema Hornklauselcompilation.
In Kapitel 3.1 wird eine kurze Motivation fuer die Compilation von Hornklauseln in LISP-Funktionen gegeben. In den Kapiteln 3.2, 3.3 und 3.4 wird dann systematisch ein Konzept zur Compilation von Hornklauseln entwickelt. Dort werden eingehend der Aufbau der zu erzeugenden LISP-Funktionen, die moeglichen Konstrukte in Klauselkoepfen und die Bearbeitung des Cut-Operators beschrieben. Diese Compilation von LISPLOG-Klauseln bedeutet im wesentlichen die Bearbeitung der Klauselkoepfe, da die Praemissen einer Regel kaum Ansatzpunkte fuer eine Compilation bieten. Der Aufruf dieser compilierten Klauseln innerhalb des LISPLOG-Interpreters ist das Thema von Kapitel 3.5.
Fuer den reinen Anwender sind sicherlich das Kapitel 3.6 sowie Anhang E von mehr Interesse, da dort die Anwendung des Klauselcompilers beschrieben und anhand eines Beispieles deutlich gemacht wird.

2. Breitensuche

2.1 Motivation fuer Breitensuche

Als Motivation fuer Breitensuche anstelle von Tiefensuche soll hier ein kurzer Vergleich beider Suchstrategien gegeben werden. Eine Beschreibung beider Verfahren findet man in fast jedem einfuehrenden Lehrbuch fuer Kuenstliche Intelligenz, z.B. auch in [RICH , 1983].
Es gibt zunaechst einmal gewichtige Gruende, die fuer ein Tiefensuchverfahren sprechen. Dies waeren z.B.:
(a) Bei Anwendung eines Tiefensuchverfahrens hat man geringeren Buchfuehrungsaufwand. Bei LISPLOG.1 wird dies wegen der rekursiven Aufrufstruktur der Interpreterkernfunktionen zu einem grossen Teil durch das FRANZ-LISP-System erledigt.
(b) Da bei einem Tiefensuchverfahren nur der aktuelle Pfad im Suchbaum abgespeichert werden muss, wird nur Speicherplatz in der Groessenordnung O(n) benoetigt, wobei n die Tiefe im Suchbaum ist. Zum Vergleich: Ein Breitensuchverfahren benoetigt Speicherplatz in der Groessenordnung O(a**n), wobei a die mittlere Verzweigungsrate (= mittlere Zahl anwendbarer Klauseln) im Suchbaum ist.

Es gibt aber auch Gruende, die fuer Breitensuche sprechen. Betrachten wir einmal folgendes Beispiel:

```
(sumto _N _RESULT)(is _N1 (subl _N))
                  (sumto _N1 _RESULT1)
                  (is _RESULT (add _N _RESULT1))
!(sumto 1 1)
```

Dies ist ein triviales LISPLOG-Programm, welches die Summe von 1 bis _N berechnet. Ein mit dem Tiefensuchverfahren arbeitender LISPLOG-Interpreter geraet hier unvermeidlich in eine Endlosschleife, da die zuerst stehende Regel immer wieder angewendet wird, s.d. die an zweiter Stelle stehende Abbruchbedingung nie erreicht wird. Nun kann man dies sicherlich durch eine ganz einfache Programmiermethodik vermeiden, indem man naemlich alle Fakten an den Anfang einer Prozedur schreibt, aber dieses Beispiel macht bereits deutlich, dass die Wahl des Suchverfahrens grossen Einfluss auf die Semantik eines logischen Programmes hat, denn ein mit Breitensuche arbeitender LISPLOG-Interpreter wuerde jeweils versuchen, beide Klauseln mit dem Goal zu unifizieren.
Betrachten wir nun eine andere Klasse von Problemen, naemlich das Durchsuchen von Suchgraphen anstelle von Suchbaeumen. Hier existieren zu manchen Regeln auch inverse Regeln, d.h. Loesungsschritte koennen zurueckgenommen werden, so dass ein Tiefensuchverfahren sich ausschliesslich mit der aufeinanderfolgenden Ausfuehrung von zueinander inversen Regeln beschaeftigen kann, ohne jemals der Loesung auch nur einen Schritt naeherzukommen. Sollen solche Zyklen im Programmablauf vermieden werden, muss der Programmierer explizit

dafuer Sorge tragen (tatsaechlich ist ein haeufiger Einwand gegen KI-Programme, die Loesung sei bereits einkodiert). Ein Beispiel fuer ein solches Problem ist das bekannte Wasserbehaelterproblem: Es sind zwei Wasserbehaelter mit 4 bzw. 3 Litern Fassungsvermoegen vorhanden. Kein Behaelter hat irgendwelche Markierungen zum Messen des Inhalts. Wie bekommt man, mit den nachfolgend aufgefuehrten Aktionen, genau zwei Liter Wasser in den groesseren Behaelter?

1. Fuelle den groesseren Behaelter randvoll!
2. Fuelle den kleineren Behaelter randvoll!
3. Falls der groessere Behaelter nicht leer ist, entleere ihn.
4. Falls der kleinere Behaelter nicht leer ist, entleere ihn.
5. Schuette Wasser aus dem kleineren Behaelter in den groesseren Behaelter, bis dieser voll ist.
6. Schuette Wasser aus dem groesseren Behaelter in den kleineren Behaelter, bis dieser voll ist.
7. Entleere den kleineren Behaelter in den groesseren Behaelter.
8. Entleere den groesseren Behaelter in den kleineren Behaelter.

Man sieht sofort, dass die Regeln 1) und 3), 2) und 4) zu einander invers sind und auch die Regeln 5) und 6) bzw. 7) und 8) invers angewendet werden koennen.
Codiert man dies in LISPLOG, so ergibt sich folgendes Programm:

```
!(state 2 _y)                                    ; Zielzustand
(state _x _y)(lessp _x 4)                        ; Regel 1
             (state 4 _y)
(state _x _y)(lessp _y 3)                        ; Regel 2
             (state _x 3)
(state _x _y)(greaterp _x 0)                     ; Regel 3
             (state 0 _y)
(state _x _y)(greaterp _y 0)                     ; Regel 4
             (state _x 0)
(state _x _y)(greaterp (plus _x _y) 3)           ; Regel 5
             (greaterp _y 0)
             (state 4 (difference (plus _x _y)
                                  4))
(state _x _y)(greaterp (plus _x _y) 2)           ; Regel 6
             (greaterp _x 0)
             (state (diffrence (plus _x _y)
                               3)
                    3)
(state _x _y)(lessp (plus _x _y) 5)              ; Regel 7
             (greaterp _y 0)
             (state (plus _x _y) 0)
```

```
(state _x _y)(lessp (plus _x _y) 3)          ; Regel 8
             (greaterp _x 0)
             (state 0 (plus _x _y))
```

Drei kurze Anmerkungen sind zum Verstaendnis dieses Programmes notwendig:
1. Die Relation (state _x _y) bezeichnet einen Zustand, der durch die Fluessigkeitsmenge in den beiden Behaeltern charakterisiert wird. Hierbei steht das erste Argument (_x) fuer den Inhalt des groesseren Behaelters, das zweite Argument (_y) fuer den Inhalt des kleineren Behaelters.
2. Die Angabe von LISP-Ausdruecken in LISPLOG-Zielen ist nur in dieser speziellen Version von LISPLOG moeglich (vgl. Kapitel 3.1).
3. Dieses Programm arbeitet vorwaertsverkettet, d.h. man stellt eine Anfrage, z.B. (state 0 0) und das Programm sucht einen Loesungsweg von diesem Ausgangszustand zu dem Zielzustand (state 2 _y).
Mit einem Tiefensuchverfahren als Suchstrategie wuerde das Prgramm in einen Zyklus
(0,0) --> (4,0) --> (4,3) --> (0,3) --> (4,3)
geraten. Auch kann man hier Endlosschleifen nicht mehr durch eine einfache Umordnung der Klauseln vermeiden.

Sieht man einmal von diesen Beispielen ab, welche die augenscheinlichen Vorteile von Breitensuche deutlich machen, so gibt es aber durchaus auch Effizienzgruende, die fuer ein Breitensuchverfahren sprechen.
Das obige Programm hat einen ungefaehren Suchaufwand in der Groessenordnung o(3**n), wobei n die Anzahl der Beweisschritte vom Startzustand zum Zielzustand ist. Die Breitensuche bietet aber wesentlich bessere Ansaetze zum Einsatz von Heuristiken zur Steuerung des Suchprozesses als die Tiefensuche (vgl. [RICH , 1983] [NILSSON , 1982]) obwohl dies hier nicht weiter behandelt werden soll.

2.2 Vorueberlegungen zur Breitensuche

Ein Breitensuchverfahren ist eigentlich ein paralleles Verfahren (auch besonders gut fuer Pipelining geeignet), kann also auf konventionellen Maschinen nur mit entsprechenden Effizienzverlusten durch eine "abstrakte Maschine" realisiert werden.
Eine fruehe LISP/PROLOG-Breitensuche gab es in [RO/SI , 1982].
Zum leichteren Verstaendnis der folgenden Kapitel sollen hier nun erst einmal einige Begriffe festgelegt werden.

Prozedur P : Die Menge aller Klauseln deren Kopf das Praedikat p als Funktor enthaelt.

Knoten : Ein Knoten ist ein 3-Tupel (L, ENV, Z),
mit
- L ist eine Liste der noch zu beweisenden Ziele, die sequentiell abgearbeitet werden.
- ENV ist die in den bisherigen Beweisschritten brechnete Menge von Substitutionen (Bindungsumgebung, Environment).
- Z ist eine Zahl die, nur eine interne Bedeutung fuer die Umbenennung der Variablen hat.

Im folgenden gelte :
K := { Ki | Ki ist Knoten}
P sei eine LISPLOG-Prozedur
P := { cl .. cn }
ci := (hi pil .. pir) ist LISPLOG-Klausel
UI sei eine Funktion, die einen LISPLOG-Ausdruck durch Ersetzung aller Variablen durch ihren Kettenendwert instanziiert (vgl. ultimate-inst).

Resolve : Dies ist ein Operator der auf einen individuellen Knoten und eine Klausel angewendet wird und daraus einen neuen Knoten konstruiert.

Seien Ki = (Li, ENVi, Zi)
Li = (xl x2 .. xm)
Dann gilt:
1. Falls es ein ENVi' gibt, s.d.
(UI hj ENVi') = (UI xl ENVi') und
ENVi' erweitert ENVi konsistent
dann gilt
(Resolve Ki cj)
= (Resolve ((xl .. xm) ENVi Zi) cj)
:= ((pjl .. pjr x2 .. xm) ENVi' (Zi + 1))

2. Falls es kein solches ENVi' gibt, gilt
(Resolve Ki cj) := nil

Resolvents : Dies ist ein Operator der auf einen individuellen Knoten angewendet wird und daraus mittels des Operators Resolve und einer LISPLOG-Prozedur eine Menge von Knoten erzeugt.

1. fuer m > 0 gilt:
(Resolvents ((xl .. xm) ENVi Zi)) :=
{(Resolve Ki cl)} U .. U {(Resolve Ki cr)}
wobei {nil} als leere Menge betrachtet wird.

2. fuer m = 0 gilt:
(Resolvents Ki) := true

mit {cl .. cr} =: P
und P ist die Prozedur zum Funktor des Zieles

xl.

Or-parallel : Dieser Operator arbeitet auf der Menge aller Knoten und ist folgendermassen definiert:

1. (Or-parallel {Kl .. Kn}) :=
   (Or-parallel
   {Kl..Ki-l Ki+l..Kn} U (Resolvents Ki))
2. (Or-parallel {Kl .. Ki-l nil Ki+l .. Kn)
   := (Or-parallel {Kl .. Ki-l Ki+l .. Kn})
3. (Or-parallel {Kl .. Ki-l true Ki+l .. Kn})
   := true

Mit diesen Begriffen laesst sich ein Beweis als eine Anwendung der Operatoren Or-parallel, Resolvents und Resolve beschreiben, wobei der initiale Knoten (Anfrageknoten) aus der Liste der Anfrageziele, der leeren Bindungsumgebung und dem level (Z) 0 besteht.
Der Operator Or-parallel muss nun in geeigneter Weise einen Knoten zur Bearbeitung durch den Resolvents-Operator auswaehlen, der seinerseits den Resolve-Operator auf den Knoten und jede Klausel aus der durch den Funktor des ersten Zieles in der Zielliste bestimmten Prozedur anwendet. Das Verfahren terminiert, wenn
- entweder die Menge aller Knoten leer ist (fail),
- oder der Resolvents-Operator auf einen Knoten angewendet wird, dessen Liste der noch zu beweisenden Ziele leer ist (success).
Erfolgt die Auswahl der Knoten in geeigneter Weise (z.B. erst die Knoten mit minimalem Z), so ist sichergestellt, dass eine vorhandene Loesung in jedem Fall gefunden wird.

2.3 Der initiale Cut und Breitensuche

Ein Nachteil der einfachen (wie oben beschriebenen) Breitensuche ist, dass nach dem Beweis von Teilzielen ueberfluessige Knoten, d.h. solche die nicht zu einer Loesung fuehren koennen, unnoetig weiter expandiert werden. Aehnlich wie bei der Tiefensuche, wo ein initialer Cut ueberfluessiges Backtracking vermeiden soll, sollte es auch hier dem Programmierer moeglich sein, diese weitere Expandierung explizit zu verhindern.
Dieses Cut-Konstrukt ist auch notwendig, um Endlosschleifen zu verhindern, die hier allerdings nur dann auftreten koennen, wenn keine weiteren Loesungen vorhanden sind.
Nun ist allerdings die Semantik des ueblichen initialen Cut nicht ohne weiteres uebertragbar, da diese wesentlich von der Reihenfolge der Klauseln in der Datenbasis abhaengt, waehrend wir hier die Datenbasis als echte Menge von Klauseln betrachten, die also keinerlei Ordnung impliziert.
Den initialen Cut fuer die Breitensuche in LISPLOG kann man daher eher mit dem EXCLUSIVE-Operator in FIT vergleichen

(vgl. [BOLEY , 1983]).
Dessen Wirkung wird am besten an einem Beispiel deutlich: Betrachten wir noch einmal das LISPLOG-Programm zur Berechnung der Summe von 1 bis n aus dem Kapitel 2.1. Der Cut-Operator vor dem Faktum (sumto 1 1) drueckt nun aus, dass diese Klausel, falls sie anwendbar ist, als einzige angewendet werden soll.
Man kann diese Definition auch leicht auf mehrere mit einem Cut versehene Klauseln verallgemeinern. In diesem Fall gilt:
Ist mindestens eine mit einem Cut versehene Klausel anwendbar, so sollen nur die mit einem Cut markierten Klauseln versucht werden.
Damit ist diese Definition des Cut wesentlich allgemeiner und auch nicht unbedingt mit der ueblichen Definition vertraeglich. Trotzdem soll aus Kompatibilitaetsgruenden die !-Notation beibehalten werden.
Durch den Cut muessen wir nun aber unsere Definition des Resolvents-Operators abaendern. Diese sieht nun folgendermassen aus:

1. Falls m > 0, gilt:
   Sei C := {(Resolve Ki cJ)| cj ist mit einem Cut markiert}
   N := {(Resolve Ki cj)| cj ist nicht mit einem Cut markiert}
   Dann gilt:
   Falls C <> nil
   (Resolvents Ki) := C
   sonst
   (Resolvents Ki) := N
2. wie oben

2.4 Implementierung

Die Implementierung weicht nur in einigen unwesentlichen Punkten von dem oben beschriebenen Konzept ab. Die kompletten Listings der LISP-Funktionen findet man in Anhang B, einige zusaetzliche bzw. gegenueber LISPLOG.1 (vgl. [BO/KA , 1985]) geaenderte Funktionen in Anhang D.
Der Operator Or-parallel wird durch eine gleichnamige LISP-Funktion realisiert, welche die Menge aller Knoten als einfache lineare Liste verwaltet. Diese Funktion Or-parallel besitzt folgendes Aufrufmuster:

```
(or-parallel l_list-of-and-nodes)
```

Sie tritt an die Stelle der in LISPLOG.1 vorhandenen Funktion and-process und verlangt als Parameter eine Liste von Knoten der Form:

```
(l_Knoten-1 .. l_Knoten-n)
```

wobei l_Knoten-i folgendermassen aussieht:

(l_list-of-goals l_environment i_level)

Or-parallel entfernt jeweils das erste Element aus der Liste list-of-and-nodes und uebergibt es zum Beweis an die Funktion Resolvents, die, mit einigen Abstrichen, den gleichnamigen Operator realisiert.
Die Rueckgabe der Ergebnisse erfolgt aus Effizienzgruenden jedoch nicht funktional, sondern neue Knoten werden durch eine spezielle Funktion put-node, unter Benutzung der destruktiven Funktion nconc, direkt an list-of-and-nodes appendiert. Damit ist auch eine einfache Schnittstelle zur spaeteren Implementierung von Heuristiken, welche die Suche ueber eine Ordnung der Liste list-of-and-nodes steuern, geschaffen.(*)

Die Funktion Resolvents besitzt folgendes Aufrufmuster:

(Resolvents l_goal l_goals-left l_environment i_level)

Ihre Aufgabe ist es im wesentlichen, fogende Faelle zu unterscheiden:

- compilierte, benutzerdefinierte Praedikate
- nicht compilierte, benutzerdefinierte Praedikate
- vordefinierte Praedikate (Primitive)
- LISP-Aufrufe

Im Gegensatz zu dem oben beschriebenen Konzept gibt es zwei Funktionen, die von Resolvents benutzt werden.

Dies sind die Funktionen execute-compiled-clauses und try-clauses, welche im wesentlichen den Operator Resolve realisieren. Execute-compiled-clauses werden wir im Zusammenhang mit dem Hornklauselcompiler noch eingehender besprechen (vgl. Kapitel 3.4). Die Funktion try-clauses durchlaeuft alle vorhandenen Klauseln sequentiell. Sie versucht zuerst, falls vorhanden, alle mit einem Cut markierten Klauseln mit dem Ziel zu unifizieren.(**)

* Eine Ausnahme bildet wiederum der Hornklauselcompiler. Hier werden, aus Effizienzgruenden, solche Knoten, welche durch Anwendung einer Regel erzeugt wurden, direkt an list-of-and-nodes appendiert.

** Aus Effizienzgruenden verlangt diese Funktion eine Ordnung der Klauseln, d.h. zuerst mit einem Cut markierte Klauseln, danach alle anderen. Dies wird durch eine geringfuegige Aenderung der Funktion ass-1 erreicht (vgl. Anhang D).

Ist keine dieser Klauseln anwendbar, so werden auch noch jene Klauseln versucht, die nicht mit einem Cut markiert sind. Dies wird durch eine einfache cond-Abfrage in der Bearbeitungsschleife erreicht. Zuerst erfolgt ein Test, ob eine mit einem Cut markierte Klausel vorliegt. Falls dies der Fall ist und diese Klausel auch anwendbar ist, so wird die Variable not-cut auf nil gesetzt. Ist die Klausel nicht mit einem Cut markiert, so wird mit Hilfe der Variablen not-cut ueberprueft, ob die Klausel angewendet werden darf.

Die Funktion put-node wird als Schnittstelle fuer den Zugriff auf die Variable list-of-and-nodes verwendet. Sie hat folgendes Aufrufmuster:

```
(put-node l_goals l_environment i_level g_front)
```

Sie testet, ob der Beweis erfolgreich abgeschlossen ist, d.h. ein Knoten mit leerer list-of-goals erzeugt wurde. In diesem Fall wird entweder der Benutzer gefragt oder aber die Bindungsumgebung als weitere Loesung fuer n-solutions in der Variablen all-environment abgespeichert (fuer eine detailliertere Beschreibung s. [BO/KA , 1985]). Falls es jedoch noch weitere Ziele zu beweisen gibt, wird mit Hilfe des Parameters front gesteuert, ob der neue Knoten an den Anfang oder an das Ende der Liste list-of-and-nodes angehaengt wird. Ein neuer Knoten wird genau dann an den Anfang von list-of-and-nodes angehaengt, wenn dieser Knoten durch die Auswertung eines LISP-Aufrufs, eines LISPLOG-Primitives oder eines compilierten LISPLOG-Faktums entstanden ist. Dies bedeutet, wie man sich leicht klarmachen kann, keinerlei Einschraenkung der Forderung, die Terminierung eines LISPLOG-Programmes fuer den Fall einer vorhandenen Loesung zu sichern, da in diesen Faellen die Liste list-of-goals des neuen Knoten echt kuerzer als diejenige des erzeugenden (Vater-) Knoten ist.

3. Klauselcompilation

3.1 Motivation

Waehrend der Implementierung der ersten Breitensuchversion von LISPLOG ergaben Laufzeitvergleiche mit der LISPLOG.1-Version groessenordnungsmaessig nur geringe Unterschiede im Laufzeitverhalten, d.h. die LISPLOG.1-Version war bei relativ kleinen, ausgeglichenen Suchbaeumen weniger als doppelt so schnell. Beim vollstaendigen Durchsuchen endlicher Suchbaeume ergab sich sogar ein annaehernd gleiches Zeitverhalten beider Verfahren. Trotzdem erschien es, vor allem im Hinblick auf die LISPLOG.1-Implementierung, notwendig, die Effizienz des Verfahrens zu verbessern. Dazu standen a priori zwei Loesungsmoeglichkeiten zur Auswahl:

- Steuerung der Suche durch Heuristiken, z.B. durch den A*-Algorithmus zur heuristischen Suche in ODER-Baeumen (*) (vgl. [RICH , 1983], [NILSSON , 1982]).

- Compilierung von LISPLOG-Klauseln nach LISP (Literatur zur Compilation von Hornklauseln: [WARREN , 1977])

Beide Ansaetze erschienen auch erfolgversprechend, wobei allerdings der Ansatz zur Compilation von Hornklauseln gegenueber der heuristischen Suche den Vorteil hat, dass er auf die, nach dem Tiefensuchverfahren arbeitende, LISPLOG.1-Version uebertragber ist.
Die ersten Tests mit dem Compiler ergaben dann auch gute Ergebnisse, es wurde zum Teil eine Geschwindigkeitssteigerung bei compilierten LISPLOG-Programmen um einen Faktor 5 bis 6 erreicht, in ganz wenigen, sehr extremen Faellen, wie etwa dem obigen Beispiel zur Loesung des Wasserbehaelterproblems, die durch fortwaehrende Anwendung von sehr einfachen Regeln (die Unifikation ist in diesem Fall geradezu trivial, eine einfache Wertzuweisung) charakterisiert sind, sogar eine enorme Steigerung um Faktoren zwischen 10 und 20. Ein weiterer Vorteil dieses Ansatzes ist die Moeglichkeit, mit sehr geringem zusaetzlichem Aufwand LISP-Ausdruecke in LISPLOG-Zielen auszuwerten und dadurch eine kompaktere Notation und verbesserte Effizienz zu erreichen (s. Beispiel

---

* In unserem Fall liegt nur ein ODER-Baum vor, da die UND-Verzweigungen als Bestandteile der graphentheoretisch atomaren Knoten aufgefasst werden. Es findet keine "and-parallele" Abarbeitung von Zielen statt, da hierdurch der Kommunikationsaufwand steigen und die gesamte Arbeit komplizierter wuerde, ohne zu dem wesentlichen Ziel, die Terminierung eines LISPLOG-Programmes zu sichern, beizutragen.

Kapitel 2.1).

3.2 Vorueberlegungen zur Klauselcompilation

Wie bereits in Kapitel 3.1 beschrieben, soll jede einzelne Klausel in eine LISP-Funktion uebersetzt werden. Dies erleichtert einmal die Uebertragung der Konzepte auf eine nach dem Tiefensuchverfahren arbeitende LISPLOG-Version, zum anderen erlaubt es wie in der interpretierten Version eine Indexierung der Klauseln (vor allem eine einfache Anpassung an ein erweitertes Indexierungskonzept).
Eine solche Funktion besitzt zwei Aufgaben:
a) Ueberpruefung der Anwendbarkeit, Unifikation
b) Erzeugung eines neuen Knotens
Sie besteht demzufolge aus einem Bedingungs- und einem Aktionsteil und hat folgenden Aufbau:

```
(lambda <Parameter-Liste>
  (and <Bedingung-1>
       <Bedingung-2>
       ...
       <Bedingung-n>
       <Aktion>))
```

Die Anzahl und der Inhalt der Bedingungen haengt dabei zur Zeit nur von den Konstrukten im Kopf der uebersetzten Klausel und dem Vorhandensein eines Cut innerhalb der Prozedur, zu der die Klausel gehoert, ab. Fuer die Zukunft ist aber geplant, alle LISP-Praedikate und LISPLOG-Primitive, die in den Praemissen einer Regel vor den Zielen mit benutzerdefinierten Praedikaten stehen, in den Bedingungsteil der erzeugten LISP-Funktion zu uebernehmen. Dies ist moeglich, da deren Variable ja schon bei Aufruf der LISP-Funktion gebunden sein muessen.

3.3 Diskussion der moeglichen Konstrukte in Klauselkoepfen

In diesem Kapitel sollen die moeglichen Konstrukte in Klauselkoepfen und ihre Bearbeitung durch den Hornklauselcompiler besprochen werden. Aus Gruenden der Vollstaendigkeit werden hier alle moeglichen Konstrukte aufgelistet, obwohl sie noch nicht alle implementiert sind. Dies soll jedoch bei der Anpassung des Compilers an die tiefensuchende LISPLOG-Version nachgeholt werden. Ein weiterer Grundgedanke bei dem Entwurf dieses Compilers war die moeglichst flexible Gestaltung, um ihn leicht an moegliche Weiterentwicklungen von LISPLOG anpassen zu koennen. Im folgenden sollen nun alle diejenigen Faelle aufgezaehlt werden, die der Compiler unterscheiden muss, um LISPLOG-Klauseln richtig uebersetzen zu koennen, ohne dass bestimmte Aspekte von LISPLOG verlorengehen, wie z.B. die variable Anzahl von Argumenten fuer Klauseln.

```
Bsp.:   (appimp nil)
        (appimp _res _l . _r)(append _l _x _res)
                              (appimp _x . _r)
```

Dies ist ein Konstrukt, wie es z.B. in CPROLOG nicht moeglich ist. Der Compiler sollte aber auch solche Klauseln uebersetzen koennen, z.B. in eine NLAMBDA-LISP-Funktion, um die Flexibilitaet und Vielseitigkeit des Interpreters moeglichst in vollem Umfang zu erhalten.
Aber beginnen wir erst einmal mit den einfachsten Faellen :

1. Eine Variable tritt in einem Klauselkopf genau einmal auf.

```
Bsp.:      (sumto _N _RES) ...
              **
```

In diesem Fall kann die Variable _N sofort mit dem korrespondierenden Wert des Zieles (oder umgekehrt) gebunden werden, da sie zum Zeitpunkt der Unifikation mit Sicherheit frei ist. Anstatt dies nun aber in der Bindungsumgebung durch Eintrag eines Bindungspaares zu vermerken, ersetzen wir zur Compilezeit jedes Vorkommen der Variablen _N in den eventuell vorhandenen Praemissen der Klausel durch den formalen Parameter der erzeugten LISP-Funktion, der als Platzhalter fuer den entsprechenden Wert des aktuellen Zieles steht. Dies hat mehrere Vorteile:

- Es kostet nur wenig Aufwand, zur Laufzeit LISP-Variablen in den durch die erzeugte Funktion zurueckgelieferten Listen zu evaluieren.

- Beim Abarbeiten der Praemissen dieser Klausel entfallen kostspielige Aufrufe der Funktion ultimate-assoc fuer diese Variable _N.

- Lange, sinnlose Bindungsketten von Variablen aneinander werden soweit wie moeglich vermieden.

Anmerkung: Dies funktioniert so nur bei der Breitensuch-version, da nur hier kein Backtracking, welches Bindungen zuruecknimmt, auftritt.

2. Sehr haeufig treten allerdings Faelle auf, in denen eine Variable mehrfach im Kopf einer Klausel auftritt. In diesem Fall koennen wir, um Mehrfachbindungen (Ueberschreiben eines Wertes) zu vermeiden, nicht einfach jedes Auftreten dieser Variablen ersetzen.

Bsp.: (append nil _LISTE _LISTE)
**** ****

Beim Bearbeiten dieser Klausel wird die Variable _LISTE beim ersten Auftreten gebunden. In diesem Fall behandeln wir beide Auftreten der Variablen _LISTE gleich und uebersetzen sie in folgendes LISP-Konstrukt, welches dann in dem Bedingungsteil der erzeugten LISP-Funktion erscheint.

```
((lambda (x y)
         (cond ((equal x y))
               ((variable-p x)
                (setq new-environment
                      (cons (list x y)
                            new-environment)))
               ((variable-p y)
                (setq new-environment
                      (cons (list y x)
                            new-environment)))))
  (ultimate-assoc <formaler Parameter> new-emvironment)
  (ultimate-assoc (quote <interne Notation von _LISTE>)
                  new-environment))
```

Wie man leicht sieht, erschoepft dieses Programmteil alle moeglichen Faelle:
a) Die ultimaten Werte des aktuellen Parameters und der Variablen _LISTE sind gleich.
b) Der ultimate Wert von zumindest einem der beiden ist eine Variable.
Gilt weder a) noch b), so schlaegt die Unifikation fehl.

3. Ein weiterer, einfacher Fall ist das Auftreten einer Konstanten im Klauselkopf.

```
Bsp.:       (sumto 1 1)
                  *
```

In diesem Fall muss zur Laufzeit verglichen werden, ob der korrespondierende aktuelle Parameter die gleiche Konstante (in diesem Beispiel 1) oder eine freie Variable ist. Dies wird durch die Uebersetzung in folgendes Programmkonstrukt erreicht:

```
((lambda (x y)
         (or (equal x y)
             (and (variable-p x)
                  (setq new-environment
                        (cons (list x y)
                              new-environment)))))
  (ultimate-assoc <formaler Parameter> new-environment)
  (quote <Konstante>))
```

Dies waren die einfachen Faelle, die in diesem Umfang auch bereits durch den Compiler verarbeitet werden. Im folgenden werden nun die komplizierteren Faelle, naemlich das Auftreten von Listenstrukturen in Klauselkoepfen, besprochen. Diese werden in der gegenwaertigen Version des Compilers noch durch einen einfachen Aufruf der unify-Funktion des LISPLOG-Interpreters bearbeitet (Fall 4) bzw. fuehren zu einem Abbruch der Compilation einer LISPLOG-Prozedur (Fall 5 und 6). Hier sollen nun einige Ideen zur Bearbeitung solcher Konstrukte entwickelt werden.

4. Ein haeufiger Fall von zusammengesetzten Strukturen in Klauselkoepfen sind dotted-pairs, die in LISPLOG an die Stelle der [HEAD | TAIL] - Konstrukte von CPROLOG treten.

```
Bsp.:    (append ( _K1 . _R1 ) _L ( _K1 . _R2 ))
                 *********
```

Hier muss zur Laufzeit unterschieden werden, ob der korrespondierende aktuelle Parameter eine freie Variable oder eine Liste, in der auch wieder Variablen auftreten koennen, ist. Verboten sind hier selbstverstaendlich Atome, die hiermit nicht unifiziert werden koennen. Die Idee fuer die Bearbeitung einer solchen Situation sieht folgende Fallunterscheidung vor:

a) Der entsprechende aktuelle Parameter ist eine Variable; dann kann die Bearbeitung mit einem einfachen Eintrag in die Bindungsumgebung beendet werden.

b) Der entsprechende aktuelle Parameter ist eine Liste; dann

muss eine Unifikation erfolgen, die die beiden CAR-Elemente wie in den oben beschriebenen einfachen Faellen bearbeitet oder einen Aufruf von unify veranlasst.
Fuer die Unifikation der beiden CDR-Elemente muss der Compiler rekursiv wieder in die verbleibende Liste hineingehen.

5. Ein spezieller Fall, der die Angabe von beliebig vielen Parametern fuer LISPLOG-Praedikate erlaubt, ist das oben beschriebene appimp - Beispiel. Die LISPLOG-Syntax erlaubt aber auch folgende Form:

Bsp.: (AND . _X)

Dies entspricht in etwa den aus LISP bekannten NLAMBDA-Ausdruecken und es ist naheliegend, bei der Uebersetzung solcher Klauseln darauf zurueckzugreifen. Abgesehen von dieser NLAMBDA-Eigenschaft kann diese Klausel aber aehnlich dem unter 4) beschriebenen Fall behandelt werden, wobei natuerlich wieder beachtet werden muss, dass anstelle der Variablen _X hier auch eine Liste von Variablen und/oder Konstanten auftreten kann.

6. Ein hoffentlich letzter Fall, der zu beachten ist, ist die Moeglichkeit, in LISPLOG-Prozeduren Klauseln einzufuegen, die eine unterschiedliche Anzahl von Parametern haben. Dies hat fuer den Compiler recht unschoene Nebenwirkungen, da die bisher verwendeten, als LAMBDA-Ausdruecke definierten, LISP-Funktionen auf eine "falsche" Parameterzahl mit einem Fehlerabbruch reagieren, s.d. die Compilation in einem solchen Fall bisher sicherheitshalber abgebrochen wurde. In Zukunft soll in diesem Fall aehnlich wie unter 5) beschrieben verfahren werden.

3.4 Der initiale Cut und Klauselcompilation

Wie schon in Kapitel 2.3 beschrieben, verlangt die Einfuehrung des initialen Cut eine Aenderung des Resolvents-Operators. Da dessen Aufgaben, der besseren Lesbarkeit und groesseren Uebersichtlichkeit des Programmes wegen, aus der Resolvents-Funktion herausgenommen wurden, muss die Bearbeitung des Cut in der Funktion execute-compiled-clauses und den erzeugten LISP-Funktionen stattfinden. Diese muessen

- signalisieren, ob die uebersetzte Klausel mit einem Cut markiert war
- ueberprufen, ob vorher eine mit einem Cut markierte Klausel anwendbar war, falls diese uebersetzte Klausel nicht mit einem Cut markiert war.

Falls in der zu uebersetzenden LISPLOG-Prozedur mit einem Cut markierte Klauseln enthalten sind, werden die oben erwaehnten Bedingungen und Aktionen durch folgende Erweiterungen der erzeugten LISP-Funktionen erreicht:

a) Falls die zu uebersetzende Klausel nicht mit einem Cut markiert ist, wird als erste Bedingung im Bedingungsteil der erzeugten LISP-Funktion die in execute-compiled-clauses mit true initialisierte Variable not-cut evaluiert.

b) Falls die zu uebersetzende Klausel mit einem Cut markiert ist, wird als zusaetzliche Aktion im Aktionsteil der erzeugten LISP-Funktion die Variable not-cut auf nil gesetzt. Durch diese Massnahmen wird erreicht, dass, wenn einmal eine mit einem Cut markierte Klausel anwendbar war, nur noch mit einem Cut markierte Klauseln anwendbar sind. Dies beruht natuerlich wieder auf einer Teilung der Prozedur in zwei Mengen von Klauseln, solche mit und solche ohne Cut.

3.5 Der Aufruf von compilierten Klauseln in LISPLOG

Wie in den vorhergehenden Kapiteln beschrieben, wird jede Klausel fuer sich in eine LISP-Funktion uebersetzt; deren Name ist folgendermassen aufgebaut :

<Praedikatname>-lisp_<Nummer>

Der LISPLOG-Interpreter muss nun, mittels einer Indexierung, Funktionen auswaehlen koennen. Also werden die Funktionsnamen, aehnlich wie nicht uebersetzte Praedikate, unter Property-Listen der Praedikate abgespeichert.
Die Funktion Resolvents testet dann, ob zu dem Praedikat des aktuellen Zieles eine solche Property-Liste existiert. Falls dies der Fall ist , werden die so ermittelten Funktionsnamen zusammen mit der Bindungsumgebung und der korrekt quotierten Liste der Argumente an die bereits erwaehnte Funktion execute-compiled-clauses uebergeben.
Diese initialisiert die Variablen not-cut und new-environment und wendet nun nacheinander die besagten Funktionen auf die Argumentliste an. Es muss dann nur noch nach jedem Schritt die durch Seiteneffekte geaenderte Variable new-environment (sie ist bzgl. der angewendeten Funktionen global) wieder den urspruenglichen Wert erhalten.

3.6 Aufruf und Anwendung des Compilers

Fuer die Anwendung des Compilers stehen dem Benutzer zwei neue Interaktionskommandos auf dem LISPLOG-Toplevel zur Verfuegung. Dies sind:

compile-db

Dieses Kommando bewirkt eine Uebersetzung der gesamten Datenbasis.

compile-p ‹Praedikat-1› ... ‹Praedikat-n›

Dieses Kommando kann mit beliebig vielen Argumenten (Praedikate) aufgerufen werden und bewirkt einer Uebersetzung dieser Praedikate.

Im Normalfall schreibt dann der Compiler die Praedikate der erfolgreich uebersetzten Prozeduren auf den Bildschirm, so dass man jederzeit erkennt, wie weit die Uebersetzung fortgeschritten ist.
Wegen der bereits erwaehnten Beschraenkungen des Compilers, kann in einigen oben erwaehnten Faellen auch eine Abbruchmeldung fuer einzelne Praedikate erscheinen.
Ausserdem erzeugt der Compiler zwei Dateien zum Sichern der generierten Funktionen und der notwendigen Informationen, um auf diese zugreifen zu koennen. Eine Separierung von Code und Zugriffsinformationen fuer den Interpreter hat den Vorteil, dass bei einer Aenderung der Indexierung der Code selbst beibehalten werden kann.
Dabei kann die Datei mit dem LISP-Code nach entsprechender Aufbereitung auch durch den LISP-Compiler uebersetzt werden. Fuer beide Dateien wird ein Name ‹name› vom Benutzer erfragt.

- ‹name›.load.file enthaelt die Informationen, die fuer den LISPLOG-Interpreter notwendig sind. Diese Datei kann in FRANZ-LISP mit dem 'dskin'-Kommando geladen werden.

- ‹name›.l enthaelt die erzeugten LISP-Funktionen. Diese Datei wird beim Laden der Datei ‹name›.load.file automatisch geladen.

3.7 Die Kernfunktionen des Compilers

In diesem Kapitel werden in einem Top-Down-Ansatz die wichtigsten Funktionen des Compilers und ihre Aufgabe beschrieben. Dieses Kapitel ist deshalb fuer den reinen Anwender nicht wichtig, sondern richtet sich mehr an diejenigen, die an der Implementierung interessiert sind. Fuer mehr Details sei auf Anhang C und E verwiesen.

```
(compile-db nil)
```

Die Funktion compile-db erfragt vom Benutzer die Datei, in der der Code gesichert werden soll, und wendet dann die Funktion compile-proc auf jedes Praedikat aus der Liste predicates, welche vom LISPLOG-Interpreter gefuehrt wird, an.

```
(compile-proc s_predicate)
```

Diese Funktion wird von der Funktion compile-db aufgerufen. Sie ruft die rekursive Funktion compile-proc-1 mit allen zu s_predicate vorhandenen Klauseln, einem Identifier zur Abspeicherung der zu erzeugenden Funktion, sowie deren Parameterliste auf. Danach testet sie, ob Gruende, die zum Abbruch der Compilation zwingen, vorliegen und leitet entsprechende Aktionen ein, d.h. macht bereits durchgefuehrte Compilationsschritte rueckgaengig, d.h. loescht bereits generierte Funktionen.

```
(compile-proc-1 l_clause-list s_identifier i_number
                l_param-list)
```

Die Funktion compile-proc-1 testet zuerst, ob die Compilation aus einem der oben erwaehnten Gruende abgebrochen werden soll und signalisiert einen Abbruch an compile-proc. Falls dies nicht der Fall ist, uebersetzt sie die erste Klausel aus l_clause-list durch einen Aufruf der Funktion compile-clause. Danach ruft sie sich mit den restlichen zu uebersetzenden Klauseln rekursiv wieder auf.

```
(compile-clause l_clause l_param-list s_identifier)
```

Die Funktion compile-clause erzeugt durch einen Aufruf der Funktion create-conditions den Bedingungsteil der aus l_clause zu erzeugenden LISP-Funktion und definiert diese dann nach dem in Kapitel 3.2 beschriebenen Muster.

```
(create-conditions l_clause-head-left
                   l_clause-head
                   l_param-list)
```

Diese Funktion ueberprueft die in Kapitel 3.3 beschriebenen Faelle und erzeugt daraus die ebenfalls bereits beschriebenen Bedingungsmuster bzw. einen Compilationsabbruch. Sie geht dabei sequentiell durch den Klauselkopf durch, testet welcher

Fall zutrifft, erzeugt den entsprechenden LISP-Code durch Ausfuellen eines vorgegebenen Patterns und ruft sich dann, falls keine Abbruchbedingung erfuellt ist, rekursiv mit dem Rest des Klauselkopfes wieder auf. Der Parameter l_clause-head wird benoetigt, um das eventuell mehrfache Auftreten einer Variablen im Klauselkopf abzutesten.

Die hier beschriebenen Funktionen benutzen zusaetzlich einige Hilfsfunktionen:

(rename-vars g_term)

Die Funktion rename-vars entspricht im Grunde der Funktion rename-variables des LISPLOG-Interpreters. Sie benennt zur Compilezeit die Variablen einer Klausel um. Dabei wird aus

(? <variablenname>) --> (? <variablenname> level).

Dieses Atom level evaluiert zur Laufzeit zu dem aktuellen Wert der dann vorhandenen LISP-Variablen level.

(smemb g_pattern l_liste i_anzahl)

Die Funktion smemb wird mit i_anzahl = 0 aufgerufen und liefert dann die Anzahl der Vorkommen von g_pattern in l_liste.

(quote-clause l_liste)

Die Funktion quote-clause wird auf die Praemissen einer Regel angewendet und erzeugt daraus einen Ausdruck, der exakt diese Praemissen zurueckgibt, s.d. die darin enthaltene Variable level und die formalen Parameter der durch Uebersetzung einer Klausel erzeugten LISP-Funktion zu ihrem aktuellen Wert evaluieren koennen.

(list-of-args i_number)

Die Funktion list-of-args erhaelt als Parameter die Anzahl der formalen Parameter der zu erzeugenden LISP-Funktion und konstruiert daraus eine Parameterliste. Um Namensgleichheiten, z.B. mit LISPLOG-Variablen, die fuer den Benutzer sehr undurchsichtige Nebeneffekte haetten, zu vermeiden, haben diese einzelnen Parameter folgendes Aussehen:

$argument<nummer>

Aus dem gleichen Grund wurden fuer die Identifier, unter denen die erzeugten LISP-Funktionen abgespeichert werden, laengere, aussagekraeftige Namen gewaehlt. Diese sind folgendermassen aufgebaut:

<predicate>-lisp-<nummer>

WARNUNG: Als Benutzer sollte man Namensueberschneidungen von LISPLOG-Praedikaten mit solchen Identifiern auf jeden Fall vermeiden, da sie einen NAMESTACK-OVERFLOW des FRANZ-LISP-Systems hervorrufen koennen, der zum RESET zwingt.

Trotzdem hat diese Namensgebung den Vorteil, dass auch fuer den Benutzer, der nicht mit der Implementierung vertraut ist, noch eine Zuordnung von LISPLOG-Klauseln zu ihren LISP-Funktionen moeglich ist. Ein Beispiel, fuer eine solche Namensueberschneidung:

Bsp.: Benutzung zweier LISPLOG-Praedikate
CRASH-LISP-0
CRASH

Dies fuehrt bei Compilation zum Auftreten einer Klausel und einer Funktion gleichen Namens.

Anhang A : Literatur

[BOLEY , 1983]
H. Boley
FIT-PROLOG
A Functional/Relational Language Comparison
Memo SEKI 83-14
Universitaet Kaiserslautern , 1983

[BO/KA , 1985]
H. Boley, F. Kammermeier et al
LISPLOG:
Momentaufnahmen einer LISP/PROLOG Vereinheitlichung
Memo SEKI 85-3
Universitaet Kaiserslautern , 1985

[NILSSON , 1982]
N.J. Nilsson
Principles of Artificial Intelligence
Tioga Publishing Co. 1982

[RICH , 1983]
E. Rich
Artificial Intelligence
McGraw-Hill Book Co. 1983

[RO/SI , 1982]
J. Robinson, E. Sibert
LOGLISP: An Alternative to PROLOG
Machine Intelligence 10 , 1982

[WARREN , 1977]
D.H.D. Warren
Implementing PROLOG
Compiling Predicate Logic Programs
Research Reports 39,40
Dept. of Artificial Intelligence
University of Edinburgh , 1977

Anhang B : Programmlistings Breitensuche

```
(setq or-parallel.lfns
      '(or-parallel breadth
                    try-clauses
                    create-node
                    create-cut-node
                    execute-compiled-clauses))

(def or-parallel
  (lambda (list-of-and-nodes)
    (let ((proved nil))
         (do ((current-node (first list-of-and-nodes)
                            (first list-of-and-nodes)))
             ((or proved (null list-of-and-nodes)))
             (setq list-of-and-nodes (rest list-of-and-nodes))
             (breadth (first (first current-node))
                      (rest (first current-node))
                      (second current-node)
                      (third current-node)))))))


(def breadth
  (lambda (goal goals-left environment level)
    (setq goal (ultimate-inst goal environment))
    (cond ((get (first goal) 'identifiers)
           (execute-compiled-clauses (quote-one (rest goal) 1)
                                     environment
                                     (get (first goal)
                                          'identifiers)))
          ((is-user-defined goal)
           (try-clauses goal goals-left environment level))
          ((is-primitive goal)
           (prolog-primitive goal goals-left environment level))
          ((is-lisp-defined goal)
           (lisp-predicates goal goals-left environment level))
          (t (princ "WARNING : UNDEFINED PREDICATE ")
             (print (first goal))
             (terpr)))))
```



```
(def try-clauses
  (lambda (goal goals-left environment level)
    (let ((database (get (first goal) 'clauses)))
         (do ((assertion (rename-variables (first database)
                                           (list level))
                         (rename-variables (first database)
                                           (list level)))
              (not-cut t))
             ((or (null database) proved))
             (cond ((cut-p assertion)
                    (create-cut-node (unify goal
                                            (s-conclusion
                                             (remove-cut
                                              assertion))
                                            environment)
                                     (append (rest assertion)
                                             goals-left)))
                   (not-cut
                    (create-node (unify goal
                                        (s-conclusion
                                         (remove-cut
                                          assertion))
                                        environment)
                                 (append (rest assertion)
                                         goals-left))))
             (setq database (rest database))))))


(def create-node
  (lambda (new-environment new-goals)
    (and new-environment
         (put-node new-goals new-environment (addl level) nil))))


(def create-cut-node
  (lambda (new-environment new-goals)
    (and new-environment
         (not (setq not-cut nil))
         (put-node new-goals new-environment (addl level) nil))))


(def execute-compiled-clauses
  (lambda (param-list environment identifiers)
    (do ((not-cut t) (new-environment environment environment))
        ((or (null identifiers) proved))
        (eval (cons (first identifiers) param-list))
        (setq identifiers (rest identifiers)))))
```

Anhang C : Programmlistings Hornklauselcompiler

```
(setq compiler.lfns
      '(condition1 condition2
                   condition3
                   pattern1
                   pattern2
                   create-conditions
                   rename-vars
                   smemb
                   compile-clause
                   compile-db
                   compile-proc
                   compile-proc-1
                   list-of-args
                   quote-clause))

(setq condition1
      '((lambda (x y)
                (cond ((equal x y))
                      ((variable-p x)
                       (setq new-environment
                             (cons (list x y)
                                   new-environment)))
                      ((variable-p y)
                       (setq new-environment
                             (cons (list y x)
                                   new-environment)))))

        (ultimate-assoc musterl new-environment)
        (ultimate-assoc 'muster2 new-environment)))

(setq condition2
      '((lambda (x y)
                (or (equal x y)
                    (and (variable-p x)
                         (setq new-environment
                               (cons (list x y)
                                     new-environment)))))

        (ultimate-assoc musterl new-environment)
        'muster2))

(setq condition3
      '(setq new-environment
             (unify musterl 'muster2 new-environment)))
```



```
(setq patternl
      '(cond ((null list-of-and-nodes)
              (setq list-of-and-nodes
                    (list
                     (list (append muster goals-left)
                           (subst level
                                  'level
                                  new-environment)
                           (addl level)))))
             (t
              (nconc list-of-and-nodes
                     (list
                      (list (append muster goals-left)
                            (subst level
                                   'level
                                   new-environment)
                            (addl level)))))))

(setq pattern2
      '(cond (not-cut (setq not-cut nil) t) (t)))

(def create-conditions
  (lambda (clause-head-left clause-head param-list)
    (let ((pattern nil))
         (cond ((null clause-head-left)
                (and param-list (setq lexpr-flag t))
                nil)
               ((null param-list) (setq lexpr-flag t) nil)
               ((variable-p clause-head-left)
                (setq lexpr-flag t)
                nil)
               ((variable-p (first clause-head-left))
                (cond ((greaterp (smemb (first
                                         clause-head-left)
                                        clause-head
                                        0)
                                 1)
                       (setq pattern
                             (subst (first param-list)
                                    'musterl
                                    condition1))
                       (setq pattern
                             (subst (first clause-head-left)
                                    'muster2
                                    pattern))
                       (append (list pattern)
                               (create-conditions
                                (rest clause-head-left)
                                clause-head
                                (rest param-list))))
                      (t (setq premisses
                               (subst (first param-list)
                                      (first
```

```
                                clause-head-left)
                               premisses))
                    (create-conditions (rest
                                        clause-head-left)
                                       clause-head
                                       (rest
                                        param-list)))))
          ((listp (first clause-head-left))
           (setq pattern
                 (subst (first param-list)
                        'muster1
                        condition3))
           (setq pattern
                 (subst (first clause-head-left)
                        'muster2
                        pattern))
           (append (list pattern)
                   (create-conditions (rest
                                       clause-head-left)
                                      clause-head
                                      (rest param-list))))
          ((atom (first clause-head-left))
           (setq pattern
                 (subst (first param-list)
                        'muster1
                        condition2))
           (setq pattern
                 (subst (first clause-head-left)
                        'muster2
                        pattern))
           (append (list pattern)
                   (create-conditions (rest
                                       clause-head-left)
                                      clause-head
                                      (rest
                                       param-list))))))))
```



```
(def rename-vars
  (lambda (term)
    (cond ((variable-p term) (append term '(level)))
          ((atom term) term)
          (t
           (cons (rename-vars (first term))
                 (rename-vars (rest term)))))))


(def smemb
  (lambda (pattern liste anzahl)
    (cond ((equal pattern liste) (addl anzahl))
          ((null liste) anzahl)
          ((equal pattern (first liste))
           (smemb pattern (rest liste) (addl anzahl)))
          ((listp (first liste))
           (smemb pattern
                  (rest liste)
                  (smemb pattern (first liste) anzahl)))
          (t (smemb pattern (rest liste) anzahl)))))


(def compile-clause
  (lambda (clause param-list identifier)
    (let ((cut-flag (cut-p clause))
          (head (rename-vars (rest (s-conclusion clause))))
          (premisses (rename-vars (s-premises clause))))
         (let ((conditions
                 (create-conditions head head param-list)))
              (cond ((not cut-flag)
                     (setq conditions
                           (cons 'not-cut conditions)))
                    (t
                     (setq conditions
                           (append conditions
                                   (list pattern2)))))
              (eval
               (list 'def
                     identifier
                     (list 'lambda
                           param-list
                           (cons 'and
                                 (append conditions
                                         (list
                                          (cond
                                           ((null premisses)
                                            '(put-node
                                              goals-left
                                              (subst
                                               level
                                               'level
                                               new-environment)
                                              (addl level)
```

```
                                          t))
                                        (t
                                         (subst
                                          (cons 'list
                                                (quote-clause
                                                 premisses))
                                          'muster
                                          pattern1))))))))))
        (putprop (first (s-conclusion clause))
                 (cond (cut-flag
                        (cons identifier
                              (get (first
                                    (s-conclusion clause))
                                   'identifiers)))
                       (t
                        (append (get (first
                                      (s-conclusion clause))
                                     'identifiers)
                                (list identifier))))
                 'identifiers))))


(def compile-db
  (lambda nil
    (patom "Enter Filename for LISP-Code: ")
    (let ((filename (read)))
         (let ((file1 (concat filename '.load.file))
               (file2 (concat filename '.l))
               (filevar nil))
              (let ((p-port (outfile file1)))
                   (mapc 'compile-proc predicates)
                   (set (concat file2 'fns) filevar)
                   (apply 'dskout (list file2))
                   (print (list 'dskin file2) p-port)
                   (terpr p-port)
                   (close p-port))))
    t))
```



```
(def compile-proc
  (lambda (predicate)
    (let ((lexpr-flag nil)
          (clause-list (get predicate 'clauses)))
         (remprop predicate 'identifiers)
         (compile-proc-1 clause-list
                         (concat predicate '-lisp-)
                         0
                         (list-of-args
                          (subl
                           (length
                            (s-conclusion
                             (first clause-list))))))
         (cond (lexpr-flag (patom "Compilation of ")
                           (princ predicate)
                           (patom " aborted! ")
                           (terpr)
                           (remprop predicate
                                    'identifiers))
               (t (print predicate)
                  (print (list 'putprop
                               (list 'quote predicate)
                               (list 'quote
                                     (get predicate
                                          'identifiers))
                               ''identifiers)
                         p-port)
                  (terpr p-port)
                  (setq filevar
                        (append filevar
                                (get predicate
                                     'identifiers)))
                  (terpr))))))


(def compile-proc-1
  (lambda (clause-list identifier number param-list)
    (cond ((null clause-list) nil)
          ((not lexpr-flag)
           (compile-clause (first clause-list)
                           param-list
                           (concat identifier number))
           (compile-proc-1 (rest clause-list)
                           identifier
                           (add1 number)
                           param-list)))))
```

```
(def list-of-args
  (lambda (number)
    (cond ((zerop number) nil)
          (t
           (append (list-of-args (subl number))
                   (list
                    (concat (concat '$
                                    'argument)
                            number)))))))


(def quote-clause
  (lambda (liste)
    (cond ((null liste) nil)
          ((or (member (first liste) param-list)
               (equal (first liste) 'level))
           (cons (first liste) (quote-clause (rest liste))))
          ((listp (first liste))
           (cons (cons 'list
                       (quote-clause (first liste)))
                 (quote-clause (rest liste))))
          (t
           (cons (list 'quote (first liste))
                 (quote-clause (rest liste)))))))
```


Anhang D : Weitere Listings

```
(setq listing.fkt.lfns
      '(commands put-node
                 ass-l
                 unify
                 execute-solutions
                 execute-solve-once
                 prolog-primitive
                 lisp-predicates
                 n-solutions))

(def commands
  (lambda (inline)
    ((lambda (x)
             (cond (x (car x))
                   (t
                    (twolist 'error--lisp:internal
                             '???))))

     (errset
      (let ((com (first inline)) (intern-flag nil))
           (cond ((is-command com)
                  (cond ((eq com '+)
                         (apply 'ass (rest inline)))
                        ((eq com '-)
                         (apply 'rex (rest inline)))
                        ((eq com 'l)
                         (apply 'listing
                                (rest inline)))
                        ((eq com 'lisp)
                         (setq leave-prolog t))
                        ((eq com 'compile-db)
                         (compile-db))
                        ((eq com 'compile-p)
                         ((lambda (predicates)
                                  (compile-db))

                          (rest inline)))
                        ((member com lisp-coms)
                         (apply com (rest inline)))))
                 (t
                  (catch
                   (or-parallel
                    (list
                     (list (rename-variables inline
                                             '(0))
                           '((bottom-of-environment))
                           1)))))))))))
```

```
(def put-node
  (lambda (goals environment level front)
    (cond (goals
           (cond (list-of-and-nodes
                  (cond (front
                         (setq list-of-and-nodes
                               (cons (list goals
                                           environment
                                           level)
                                     list-of-and-nodes)))
                        (t
                         (nconc list-of-and-nodes
                                (list
                                 (list goals
                                       environment
                                       level))))))
                 (t
                  (setq list-of-and-nodes
                        (list
                         (list goals environment level))))))
          (t
           (cond (intern-flag (setq all-environment
                                    (cons environment
                                          all-environment))
                              (cond ((eq anzahl 1)
                                     (setq proved t))
                                    (t
                                     (setq anzahl
                                           (subl anzahl)))))
                 (t (print-bindings (rest
                                     (reverse environment))
                                    environment)
                    (y-or-n-p "MORE? (y OR n) "))))))))


(def ass-1
  (lambda (assertion)
    (let ((pred-1 (first (s-conclusion assertion))))
      (cond ((cut-p assertion)
             (putprop pred-1
                      (cons assertion
                            (get pred-1 'clauses))
                      'clauses))
            (t
             (putprop pred-1
                      (append (get pred-1 'clauses)
                              (ncons assertion))
                      'clauses)))
      (setq predicates (union predicates
                              (list pred-1))))))
```



```
(def unify
  (lambda (x y environment)
    (let ((x (ultimate-assoc x environment))
          (y (ultimate-assoc y environment)))
         (cond ((equal x y) environment)
               ((variable-p x) (cons (list x y) environment))
               ((and (listp x)
                     (is-lisp-defined (first x))
                     (not (eq (first (first x)) '?)))
                (let ((new-environment
                       (unify (eval (quote-all (first x)))
                              (first y)
                              environment)))
                     (and new-environment
                          (unify (rest x)
                                 (rest y)
                                 new-environment))))
               ((variable-p y) (cons (list y x) environment))
               ((or (atom x) (atom y)) nil)
               (t
                (let ((new-environment
                       (unify (first x)
                              (first y)
                              environment)))
                     (and new-environment
                          (unify (rest x)
                                 (rest y)
                                 new-environment)))))))))


(def execute-solutions
  (lambda (goal-args environment level)
    (let ((all-environment nil)
          (anzahl (caddr goal-args))
          (intern-flag t))
         (or-parallel
          (list (cdddr goal-args)
                '((bottom-of-environment))
                (addl level)))
          (unify (cadr goal-args)
                 (mapcar '(lambda (env)
                                  (ultimate-inst
                                   (car goal-args)
                                   env))
                         all-environment)
                 environment))))
```

```
(def execute-solve-once
  (lambda (goals environment level)
    (let ((all-environment nil) (anzahl 1) (intern-flag t))
         (or-parallel
          (list (ultimate-inst goals environment)
                '((bottom-of-environment))
                (addl level)))
         (and all-environment
              (unify goals
                     (ultimate-inst goals
                                    (car all-environment))
                     environment)))))


(def prolog-primitive
  (lambda (goal goals-left 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)
                (put-node 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
                          t)))
          ((is-p goal)
           (let ((new-environment
                  (execute-is (second goal)
                              (third goal)
                              environment)))
                (and new-environment
                     (put-node goals-left
                               new-environment
                               level
                               t))))
          ((eq (car goal) 'not)
           (and (execute-not goal environment level)
                (put-node goals-left environment level t)))
          ((eq (car goal) 'solutions)
           (let ((new-environment
                  (execute-solutions (ultimate-inst (cdr
                                                     goal)
```



```
                                                     environment)
                                    environment
                                    level)))
                (and new-environment
                     (put-node goals-left
                               new-environment
                               level
                               t))))
          ((eq (car goal) 'solve-once)
           (let ((new-environment
                  (execute-solve-once (cdr goal)
                                      environment
                                      level)))
                (and new-environment
                     (put-node goals-left
                               new-environment
                               level
                               t))))
          ((eq (car goal) 'var)
           (and (var-p (first (rest goal)) environment)
                (put-node goals-left environment level t)))
          ((eq (car goal) 'nonvar)
           (and (non-var-p (first (rest goal)) environment)
                (put-node goals-left environment level t))))))


(def lisp-predicates
  (lambda (goal rest-of-goals environment level)
    (cond
     ((cond
       (((lambda (erg)
                 (cond ((null erg)
                        (throw
                         'undefined-function:lisp-predicates))
                       (t (car erg))))

         (errset (getd (car goal))))
        ((lambda (erg)
                 (cond ((null erg)
                        (throw
                         'Eval-not-possible-in-lisp-predicates))
                       (t (car erg))))

         (errset
          (eval
           (quote-all (ultimate-inst goal environment)))))))
      (put-node rest-of-goals environment level t)))))
```

```
(def n-solutions
  (lambda (goal anzahl)
    (let ((all-environment nil) (anzahl anzahl) (intern-flag t))
         (cond ((null goal) t)
               ((equal anzahl 0) nil)
               ((listp goal)
                (or-parallel
                 (list (rename-variables goal '(0))
                       '((bottom-of-environment))
                       1))
                (cond (all-environment
                       (cond ((equal (first all-environment)
                                     '((bottom-of-environment)))
                              all-environment)
                             (t
                              (cons-bindings-all
                               all-environment))))
                      (t nil)))
               (t nil)))))
```

Anhang E : Beispiele zur Compilation von LISPLOG-Klauseln

Franz Lisp, Opus 38.89 mit Cmu-, Ktu- und Flavors-Erweiterungen
08.02.1985

Patch Nr. 38.3 geladen
Patch Nr. 38.4 geladen

News sind mit der Funktion (news) zu erhalten Stand : 01-05-85

```
3.(dskin quickload.7)
[load quickload.7]
[load rpmacro.1.1]
[load andor.7.1]
[load primitives.3.1]
[load lispkonzept.2.1]
[load interface.4.1]
[load unifult.3.1]
[load variable.1.1]
[load assert.2.1]
[load intern.2.1]
[load absynt.4.1]
[load compiler.7.1]
(quickload.7)
4.(lisplog)
*consult doku.beispiel.db
[load doku.beispiel.db]
(doku.beispiel.db)
*listing
(ass (father jack ken))
(ass (father jack karen))
(ass (grandparent _grandparent _grandchild)
     (parent _grandparent _parent)
     (parent _parent _grandchild))
(ass (mother el ken))
(ass (mother cele jack))
(ass (parent _parent _child) (mother _parent _child))
(ass (parent _parent _child) (father _parent _child))
(ass !(sumto 1 1))
(ass (sumto _N _RES)
     (is _N1 (sub1 _N))
     (sumto _N1 _RES1)
     (is _RES (add _RES1 _N)))
(ass !(append nil _liste _liste))
(ass (append (_kopf . _rest1) _liste (_kopf . _rest2))
     (append _rest1 _liste _rest2))
(ass (and))
(ass (and _x . _y) _x (and . _y))
(ass (or _x . _y) _x)
(ass (or _x . _y) (or . _y))

nil
```



```
*(sumto 5 _x)
(_x = 15)
MORE? (y OR n) y
nil
*compile-p sumto append mother
Enter Filename for LISP-Code: d.lisp.code
sumto
append
mother
creating d.lisp.code.1

t
*compile-db
Enter Filename for LISP-Code:  bsp
father
grandparent
mother
parent
sumto
append
Compilation of and aborted!
Compilation of or aborted!
1: 2: old version moved to bsp.1.back

t
*lisp
t
nil
5.(trace append-lisp-0 append-lisp-1 sumto-lisp-0 sumto-lisp-1)
[autoload /usr/users/rau003/ktu/lib/lisp/trace]
[fasl /usr/users/rau003/ktu/lib/lisp/trace.o]
(append-lisp-0 append-lisp-1 sumto-lisp-0 sumto-lisp-1)
6.(lisplog)
*(append (a b c)(d e f) _liste)
1 <Enter> append-lisp-0 ((a b c) (d e f) (? liste 0))
1 <EXIT>  append-lisp-0  nil
1 <Enter> append-lisp-1 ((a b c) (d e f) (? liste 0))
1 <EXIT>  append-lisp-1  ((((append & & &)) ((& &) (& &) (& a)
(bottom-of-environment)) 2))
1 <Enter> append-lisp-0 ((b c) (d e f) (? rest2 1))
1 <EXIT>  append-lisp-0  nil
1 <Enter> append-lisp-1 ((b c) (d e f) (? rest2 1))
1 <EXIT>  append-lisp-1  ((((append & & &)) ((& &) (& &) (& b)
(& &) (& &) ...) 3))
1 <Enter> append-lisp-0 ((c) (d e f) (? rest2 2))
1 <EXIT>  append-lisp-0  nil
1 <Enter> append-lisp-1 ((c) (d e f) (? rest2 2))
1 <EXIT>  append-lisp-1  ((((append & & &)) ((& &) (& nil) (& c)
(& &) (& &) ...) 4))
1 <Enter> append-lisp-0 (nil (d e f) (? rest2 3))
(_liste = (a b c d e f))
MORE? (y OR n) y
1 <EXIT>  append-lisp-0  nil
```

```
1 <Enter> append-lisp-1 (nil (d e f) (? rest2 3))
1 <EXIT>  append-lisp-1  nil
nil
*(sumto 5 _x)
1 <Enter> sumto-lisp-0 (5 (? x 0))
1 <EXIT>  sumto-lisp-0  nil
1 <Enter> sumto-lisp-1 (5 (? x 0))
1 <EXIT>  sumto-lisp-1  ((((is & &) (sumto & &) (is & &))
((bottom-of-environment)) 2))
1 <Enter> sumto-lisp-0 (4 (? RES1 1))
1 <EXIT>  sumto-lisp-0  nil
1 <Enter> sumto-lisp-1 (4 (? RES1 1))
1 <EXIT>  sumto-lisp-1  ((((is & &) (sumto & &) (is & &)
(is & &)) ((& 4) (bottom-of-environment)) 3))
1 <Enter> sumto-lisp-0 (3 (? RES1 2))
1 <EXIT>  sumto-lisp-0  nil
1 <Enter> sumto-lisp-1 (3 (? RES1 2))
1 <EXIT>  sumto-lisp-1  ((((is & &) (sumto & &) (is & &)
(is & &) (is & &)) ((& 3) (& 4) (bottom-of-environment)) 4))
1 <Enter> sumto-lisp-0 (2 (? RES1 3))
1 <EXIT>  sumto-lisp-0  nil
1 <Enter> sumto-lisp-1 (2 (? RES1 3))
1 <EXIT>  sumto-lisp-1  ((((is & &) (sumto & &) (is & &)
(is & &) (is & &) ...) ((& 2) (& 3) (& 4)
(bottom-of-environment)) 5))
1 <Enter> sumto-lisp-0 (1 (? RES1 4))
1 <EXIT>  sumto-lisp-0  ((((is & &) (is & &) (is & &)
(is & &)) ((& 1) (& 1) (& 2) (& 3) (& 4) ...) 6))
1 <Enter> sumto-lisp-1 (1 (? RES1 4))
1 <EXIT>  sumto-lisp-1  nil
(_x  =  15)
MORE? (y OR n) y
nil
*lisp
t
nil
7.exit

 Auf Wiedersehen !!!
```



```
(setq d.lisp.code.lfns
      '(sumto-lisp-0 sumto-lisp-1
                     append-lisp-0
                     append-lisp-1
                     mother-lisp-0
                     mother-lisp-1))

(def sumto-lisp-0
  (lambda ($argument1 $argument2)
    (and ((lambda (x y)
                  (or (equal x y)
                      (and (variable-p x)
                           (setq new-environment
                                 (cons (list x y)
                                       new-environment)))))

          (ultimate-assoc $argument1 new-environment)
          '1)
         ((lambda (x y)
                  (or (equal x y)
                      (and (variable-p x)
                           (setq new-environment
                                 (cons (list x y)
                                       new-environment)))))

          (ultimate-assoc $argument2 new-environment)
          '1)
         (cond (not-cut (setq not-cut nil) t) (t))
         (put-node goals-left
                   (subst level 'level new-environment)
                   (add1 level)
                   t))))


(def sumto-lisp-1
  (lambda ($argument1 $argument2)
    (and not-cut
         (cond ((null list-of-and-nodes)
                (setq list-of-and-nodes
                      (list
                       (list (append (list (list 'is
                                                 (list '?
                                                       'N1
                                                       level)
                                                 (list
                                                  'subl
                                                  $argument1))
                                           (list 'sumto
                                                 (list '?
                                                       'N1
                                                       level)
                                                 (list '?
                                                       'RES1
```

```
                                                level))
                                        (list 'is
                                              $argument2
                                              (list
                                               'add
                                               (list '?
                                                     'RES1
                                                     level)
                                               $argument1)))
                                 goals-left)
                           (subst level
                                  'level
                                  new-environment)
                           (addl level)))))
                  (t
                   (nconc list-of-and-nodes
                          (list
                           (list (append (list (list 'is
                                                     (list '?
                                                           'N1
                                                           level)
                                                     (list
                                                      'subl
                                                      $argument1))
                                               (list 'sumto
                                                     (list '?
                                                           'N1
                                                           level)
                                                     (list '?
                                                           'RES1
                                                           level))
                                               (list 'is
                                                     $argument2
                                                     (list
                                                      'add
                                                      (list '?
                                                            'RES1
                                                            level)
                                                      $argument1)))
                                         goals-left)
                                   (subst level
                                          'level
                                          new-environment)
                                   (addl level)))))))))


(def append-lisp-0
  (lambda ($argument1 $argument2 $argument3)
    (and (setq new-environment
               (unify $argument1 'nil new-environment))
         ((lambda (x y)
                  (cond ((equal x y))
                        ((variable-p x)
```



```
                         (setq new-environment
                               (cons (list x y)
                                     new-environment)))
                        ((variable-p y)
                         (setq new-environment
                               (cons (list y x)
                                     new-environment)))))

                 (ultimate-assoc $argument2 new-environment)
                 (ultimate-assoc '(? liste level)
                                 new-environment))
                ((lambda (x y)
                         (cond ((equal x y))
                               ((variable-p x)
                                (setq new-environment
                                      (cons (list x y)
                                            new-environment)))
                               ((variable-p y)
                                (setq new-environment
                                      (cons (list y x)
                                            new-environment)))))

                 (ultimate-assoc $argument3 new-environment)
                 (ultimate-assoc '(? liste level)
                                 new-environment))
                (cond (not-cut (setq not-cut nil) t) (t))
                (put-node goals-left
                          (subst level 'level new-environment)
                          (addl level)
                          t))))


(def append-lisp-1
  (lambda ($argument1 $argument2 $argument3)
    (and not-cut
         (setq new-environment
               (unify $argument1
                      '((? kopf level) ? rest1 level)
                      new-environment))
         (setq new-environment
               (unify $argument3
                      '((? kopf level) ? rest2 level)
                      new-environment))
         (cond ((null list-of-and-nodes)
                (setq list-of-and-nodes
                      (list
                       (list (append (list
                                      (list 'append
                                            (list '?
                                                  'rest1
                                                  level)
                                            $argument2
                                            (list '?
```

```
                                                'rest2
                                                level)))
                                    goals-left)
                              (subst level
                                     'level
                                     new-environment)
                              (addl level)))))
                    (t
                     (nconc list-of-and-nodes
                            (list
                             (list (append (list
                                            (list 'append
                                                  (list '?
                                                        'rest1
                                                        level)
                                                  $argument2
                                                  (list '?
                                                        'rest2
                                                        level)))
                                           goals-left)
                                   (subst level
                                          'level
                                          new-environment)
                                   (addl level)))))))))
```

```
(def mother-lisp-0
  (lambda ($argument1 $argument2)
    (and not-cut
         ((lambda (x y)
                  (or (equal x y)
                      (and (variable-p x)
                           (setq new-environment
                                 (cons (list x y)
                                       new-environment)))))

          (ultimate-assoc $argument1 new-environment)
          'el)
         ((lambda (x y)
                  (or (equal x y)
                      (and (variable-p x)
                           (setq new-environment
                                 (cons (list x y)
                                       new-environment)))))

          (ultimate-assoc $argument2 new-environment)
          'ken)
         (put-node goals-left
                   (subst level 'level new-environment)
                   (addl level)
                   t))))
```



```
(def mother-lisp-1
  (lambda ($argument1 $argument2)
    (and not-cut
         ((lambda (x y)
                  (or (equal x y)
                      (and (variable-p x)
                           (setq new-environment
                                 (cons (list x y)
                                       new-environment)))))

          (ultimate-assoc $argument1 new-environment)
          'cele)
         ((lambda (x y)
                  (or (equal x y)
                      (and (variable-p x)
                           (setq new-environment
                                 (cons (list x y)
                                       new-environment)))))

          (ultimate-assoc $argument2 new-environment)
          'jack)
         (put-node goals-left
                   (subst level 'level new-environment)
                   (addl level)
                   t))))
```