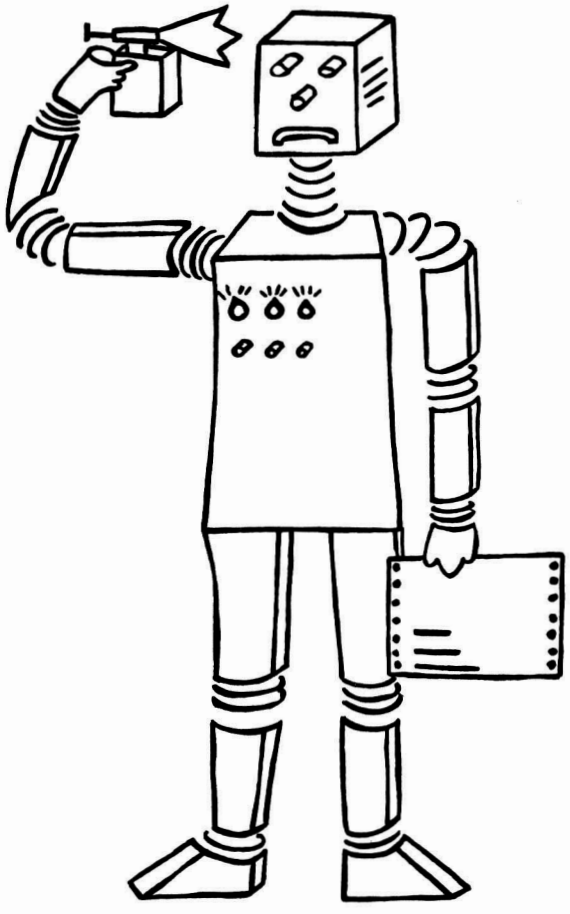
SEKI-Working Paper

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



Ein Ansatz für einen LISPLOG-Compiler mit LISP als Zielsprache

Jürgen Herr

Dezember 1986 SWP-86-06

Ein Ansatz fuer einen LISPLOG Compiler mit LISP als Zielsprache

Juergen Herr

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

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

Dezember 1986

SEKI Working Paper 86-06

A B S T R A C T

This paper describes my work on compilation of LISPLOGclauses into LISPfunctions and an improved runtimesystem for LISPLOG, in which these functions will be interpreted. The first part of this paper introduces and comments on the implemented functions, while the reasons for fundamental design decisions will be shown in the second part. Thus the first part is a program documentation, whereas the second part gives more general informations.
Furthermore the paper discusses in detail the question whether it makes sense to use (portable) LISP as the target language for a Hornclause compiler. This is the main point of the second part of this paper, and I hope it will become apparent by comparison with Warren's abstract machine, that (portable) LISP is not very suitable for this purpose.

I have to thank Dr. L. Wiesbaum and A. Poehlmann from the Siemens Corporation for papers and discussions which helped me to work on PLM (now WAM) and the descriptions of ECRC-PROLOG.

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

Gliederung:

...dass man immer wisse, was
zu sagen ist, doch vieles, was sich auch
noch sagen liesse, jetzt zurueckbehalte
und fuer den Platz, wo man's bedarf, verspare.

Horaz

0. Einleitung

Dieser Bericht dokumentiert meine Arbeit ueber Compilation von PROLOG-Klauseln und effizientere Interpretation derselben durch eine verbesserte Laufzeitumgebung. Im ersten Teil dieses Berichtes werden die implementierten Programme dokumentiert und kommentiert. Ausserdem werden moegliche Erweiterungen der gegenwaertigen Version angesprochen.

Im zweiten Teil wird eine Interpretation der bisherigen Arbeitsergebnisse durchgefuehrt. Der Schwerpunkt liegt dabei auf der Frage, ob LISP als Zielsprache fuer einen Hornklauselcompiler geeignet und sinnvoll ist. Unter diesem Gesichtspunkt wird auch die Interpretation der Vergleiche mit den anderen LISPLOG - Versionen und die Untersuchung der Perspektiven durchgefuehrt. Bei meiner Einarbeitung in Verfahren fuer intelligentes Backtracking stuetzte ich mich im wesentlichen auf [Bruynooghe 1978]. Fuer den Bereich Compilation von Hornklauseln und effiziente Implementierung von PROLOG sind die Arbeiten von D. H. D. Warren, Jacques Noye und D. S. Warren zu erwaehnen. Besonders [Noye 1984] gibt eine verstaendliche Einfuehrung in die abstrakte PROLOG-Maschine PLM.

Teil I: Programmdokumentationen

Die Listings der neu implementierten bzw. abgeaenderten Funktionen findet man in Anhang B. Diese Programme bauen auf den Hornklauselcompiler fuer eine breitensuchende LISPLOG-Version auf, der in [Herr 85] beschrieben ist. Dort findet man auch viele Einzelheiten, deren Wiederholung diese Arbeit unnoetig aufblaehen wuerde, da sie sich gar nicht oder nur unwesentlich von der hier beschriebenen Arbeit unterscheiden.

1.1 Ein Laufzeitsystem fuer compilierte Hornklauseln

Der in Teil 1.2 beschriebene Hornklauselcompiler uebersetzt PROLOG-Klauseln in LISP-Funktionen. Dabei ist zu beachten, dass diese generierten Funktionen keine prozeduralen Instruktionen fuer den Programmablauf enthalten. Diese Funktionen testen lediglich die Anwendbarkeit der Klausel und greifen auf die Laufzeitstacks zu. Fuer diese Beschraenkung gibt es mehrere Gruende, die in Teil II beschrieben sind (Dies steht im Zusammenhang mit dem Zwang zur iterativen Beweisfuehrung, um annehmbar "grosse" PROLOG-Programme zur Ausfuehrung bringen zu koennen). Zum leichteren Verstaendnis der folgenden Beschreibungen, werden nun erst einmal die Instruktionen fuer die durch den Compiler generierten Code aufgelistet:

'uniconst': Macro, welches fuer eine Konstante im "toplevel' des Klauselkopfes eingesetzt wird.
'unistruct': Macro, welches fuer eine Liste (in LISPLOG existieren keine Strukturen) auf dem "toplevel" eines Klauselkopfes eingesetzt wird.
'ustructvar': Macro fuer die Unifikation einer eingebetteten Variablen.
'ustructconst': Macro fuer die Unifikation einer eingebetteten Konstante.
'ustructstruct': Macro fuer die Unifikation einer eingebetteten Liste.

'action': Macro fuer das Updating der Stacks zur Steuerung des Beweises.
'det-action': Version von 'action' fuer deterministische Klauseln (Optimierung).

'put-binding': Macro, welches eine Variablenbindung in den Variablenstack eintraegt und im Trail vermerkt.
'address-of': Macro, welches die (relative) Adresse einer Variablen im Variablenstack ermittelt.

Diese Beschraenkung fuehrt aber dazu, dass immer noch ein Laufzeitsystem (Interpreter) fuer die Kontrolle der Beweisfuehrung benoetigt wird (PROLOG-Programme ohne dynamische Aenderungen an der Datenbasis koennten auch ohne eine solche Laufzeitumgebung auskommen). Im folgenden soll nun diese Laufzeitumgebung beschrieben werden. Sie besteht im wesentlichen aus den Komponenten der LISPLOG.1-Version, die jedoch folgendermassen abgeaendert wurden:

a) Die Korekursion von and-process und or-process wurde aufgeloest, die Stacks werden nun explizit gefuehrt.

b) Die PROLOG-Variablen werden in einem LISP-Vektor abgespeichert. Sie bestehen von nun an aus einer relativen Stackadresse, auf die zur Laufzeit eine Basisadresse aufaddiert wird (Umbenennen der Variablen).

c) Der or-process faellt weg. Statt einer allgemeinen Unifikation testet man die Anwendbarkeit einer Klausel durch Evaluation der aus ihr erzeugten LISP-Funktion.

d) Der automatische Quotierungsmechanismus von LISPLOG.1 wird bereits zur Compilezeit angewendet. Dadurch wird ein Fehler von LISPLOG.1 (erst Instanziierung und dann Quotierung) behoben.
(Anmerkung: Mittlerweile kann die automatische Quotierung auch ganz abgeschaltet werden !)

Diese Verbesserungen erzwangen aber auch die Benutzung von globalen Variablen, deren Verwendung nun kommentiert werden soll.

Zugriff auf die globalen Variablen

Der Interpreter dieser Version verwendet mehrere globale Variablen als Laufzeitstacks und zur Steuerung der do-Schleifen:

a) 'variablenstack' : Diese Variable ist ein Vektor (zur Zeit mit einer Laenge von 5000 Elementen) der die LISPLOG-Variablen speichert. Der Zugriff erfolgt ueber zwei Schnittstellenfunktionen:

(put-binding x y level)

Argumente:
- x — freie LISPLOG-Variable
- y — beliebiger LISPLOG-Term
- level — Nummer des Iterationsschrittes, in dem die Variable x gebunden wird.

Ergebnis: t

Seiteneffekte: Das Macro 'put-binding' greift auf die globale Variable 'variablenstack' und auf eine Variable 'trail' zu, welche die Adressen der in diesem Unifikationsversuch gebundenen Variablen enthaelt. In der durch die Adresse von x angegebenen Zelle wird (y level) abgelegt. 'level' wird zur Zeit noch nicht gebraucht sondern erst spaeter in Zusammenhang mit intelligentem Backtracking ('level' wird dort als Sprungmarke in den 'backtrackstack' verwendet [nicht implementiert]). Dieser 'trail' wird spaeter auf dem 'backtrackstack' abgelegt, bzw. bei einem Fehlschlag des Unifikationsversuches benutzt, um die Variablenbindungen wieder zurueckzusetzen.

(ultimate-assoc x)

Ergebnis: Kettenendwert der Bindungskette von x, bzw. nil falls keine Bindung existiert.

Argument: x eine beliebige LISPLOG-Variable

Seiteneffekte: keine

b) 'backtrackstack' : Der 'backtrackstack' ist ein Vektor mit zur Zeit 1000 Elementen. Die Schnittstellen zu dieser Variable sind leider nicht so sauber definiert. Der Zugriff auf diese Variable erfolgt an mehreren Stellen.

Ein Eintrag in den 'backtrackstack' sieht folgendermassen aus:

(l_trail l_goalstack l_anwendbareklauseln i_index)

Lesender Zugriff: Das Macro 'backtrack' greift auf de 'backtrackstack' zu um den letzten abgespeicherten Beweiszustand wieder herzustellen. Dabei wird der Trail benutzt, um Variablenbindungen zurueckzusetzen. Der 'goalstack' enthaelt die zu diesem Zeitpunkt noch zu beweisenden Ziele, in 'anwendbareklauseln' sind alle Funktionsidentifier gespeichert, welche noch nicht auf das goalpattern angewendet wurden. Eine Besonderheit ist, dass das goalpattern noch als erstes Argument auf dem 'goalstack' liegt.

Schreibender Zugriff: Schreibend wird auf den 'backtrackstack' an drei Stellen zugegriffen, naemlich ueberall dort, wo Variablenbindungen erzeugt werden, oder ein Backtracking moeglich ist.
1. 'prolog-primitives' ('execute-is') : Da durch ein is-statement eventuell eine Variablenbindung hergestellt wird, muss bei einem Backtracking vor dieses statement diese Bindung rueckgaengig gemacht werden. Hierzu wird in 'execute-is' mittels 'rplaca' die Adresse der neu gebundenen

Variablen auf den 'trail' des obersten Elementes im 'backtrackstack' gelegt (da kein eigener Choice-point erzeugt wird).
2. Unifikationsmacro 'action' : Dieses Macro wird benutzt, falls ein Backtracking moeglich ist. Falls noch anwendbare Klauseln vorhanden sind, wird der gesamte Beweisstatus auf den 'backtrackstack' gerettet und die Variable 'level' um 1 erhoeht. Sonst wird nur der 'trail' des obersten Elementes im 'backtrackstack' mittels 'rplaca' angepasst.
3. Unifikationsmacro 'det-action' : Dieses Macro wird bei Klauseln mit initialem Cut - Operator verwendet. Es passt lediglich den 'trail' des obersten Elementes im 'backtrackstack' an (s. Anmerkung zu (1), auch hier wird kein eigener Choice-point erzeugt).

c) 'goalstack' : Der Goalstack ist eine Liste von Goals, der wie in der LISPLOG.1 - Version verwaltet wird (--> 'list-of-goals').

d) 'abbruch' : Diese Variable wird zur Steuerung der do-Schleife im 'and-process' verwendet. Sie wird am Anfang mit 'nil' initialisiert und kann in 'y-or-n-p', 'backtrack' bzw. im 'and-process' ('intern-flag' = t und 'anzahl' = 1) auf t gesetzt werden, um die Iteration abzubrechen.

e) 'level' : Diese Variable wird am Anfang mit 0 initialisiert und ist ein Zeiger auf den Top-of-backtrackstack. Sie wird im Unifikationsmacro 'action' (eventuell) erhoeht, im Macro 'backtrack' (eventuell) um 1 erniedrigt.

f) 'anwendbareklauseln' : Diese Variable enthaelt eine Liste derjenigen Klauseln (bzw. Funktionsnamen) welche noch nicht auf das aktuelle Goal angewendet wurden. Der Zugriff auf diese Variable erfolgt in backtrack, execute-compiled-clauses und and-process.

g) 'index' : Diese Variable wird zur indirekten Adressierung der LISPLOG-Variablen verwendet. Jede LISPLOG-Variable enthaelt ihre lokale Adresse in der Klausel. Wird die Variable benutzt, so wird aus dieser lokalen Adresse und der Variablen index die Adresse der LISPLOG-Variablen im variablenstack bestimmt. Wird eine Klausel erfolgreich angewendet, so wird die Variable index um das lokale displacement dieser Klausel (= Maximum der lokalen Adressen) erhoeht (in den Unifikationsmacros action und det-action). Beim Backtracking wird sie wieder auf ihren urspruenglichen Wert zurueckgesetzt.

1.2 Ein Compiler zur Uebersetzung von Hornklauselkoepfen

1.2.1 Strategie der Codeerzeugung

Die Compilation nach LISP unterteilt sich in 4 Phasen:

a) Bestimmung der Aufrufmuster

In dieser Phase werden alle Klauseln einer Prozedur betrachtet, um die minimale bzw. maximale Anzahl von Parametern der zu generierenden Funktion zu bestimmen. Diese variable Parameterzahl wird durch '&optional'- bzw. '&rest'-Parameter der generierten Funktion ermoeglicht.

b) Variablentransformation

In dieser Phase werden nur noch einzelne Klauseln behandelt. Alle vorkommenden Variablen werden in folgende Form uebersetzt:

(? i_stackindex)

Der stackindex gibt eine relative Adresse auf dem 'variablenstack' an, der zur Laufzeit durch Aufaddieren einer Basisadresse ('index') in eine absolute Adresse umbenannt wird.

c) Musteruebersetzung Klauselkopf

In dieser Phase wird jedem Element des Klauselkopfes eine Instruktion des abstrakten Instruktionssatzes zugeordnet. Diese Zuordnung geschieht aehnlich wie in [Herr 85] beschrieben. Allerdings sind die Instruktionen der Zielsprache hier durch LISP-Macros realisiert. Ausserdem ist nun auch die Uebersetzung von LISPLOG-Prozeduren mit unterschiedlichen oder gar variablen Parameterzahlen moeglich.

d) Bestimmung des Aktionsteils (Stackverwaltung)

In dieser Phase werden die Stackverwaltungsoperationen fuer den Aktionsteil der generierten Funktion ermittelt. Dies besteht im wesentlichen aus der Unterscheidung zwischen deterministischen und nichtdeterministischen Klauseln, d.h. es wird eine CUT - Optimierung durchgefuehrt (im Gegensatz zu LISPLOG.1). Ausserdem wird hier auf einige globale Variablen der Laufzeitumgebung zugegriffen.

1.3 Moegliche Erweiterungen

Schnittstelle fuer interpretierte Klauseln (nicht realisiert)
Wird eine neue Klausel assertiert, so muessen folgende Aktionen durchgefuehrt werden:
a) Umbenennen der Klauselvariablen: Dies kann durch die Funktion rename-vars geschehen.

(rename-vars g_term)

Argument: g_term ist eine LISPLOG-Klausel

Ergebnis: g_term, wobei fuer jede Variable der Variablenname durch eine lokale Adresse ersetzt wurde.

Seiteneffekte: Das lokale displacement der Klausel wird in der zu rename-vars globalen variable displace gespeichert. Es muss auch fuer interpretierte Klauseln zugaenglich bleiben (vielleicht als erstes oder letztes Element der Liste, welche die Klausel darstellt.

b) Quotierung von LISP-Ausdruecken: Dies sollte unbedingt vor der Laufzeit erfolgen, um die Konsistenz mit compiliertem Code zu erhalten. Ein weiterer Vorteil der Quotierung vor der Instanziierung ist, dass Ausdruecke wie z.B. (atom _x) nicht von Hand quotiert werden muessen.

Beispiel: (von M. Lessel)
(eq _x (lpr -Plp))
mit _x = (lp hallo)

Bei Bearbeitung dieser Klauseln muessen folgende Bedingungen beachtet werden:
a) Beim Fehlschlag eines Unifikationsversuches muessen eventuelle Bindungen zurueckgesetzt werden ('trail').
b) Korrekte Adressenberechnung fuer die LISPLOG-Variablen durch Aufaddieren und Updaten von index. In diesem Zusammenhang ist auch wichtig, dass in den Praemissen einer erfolgreich angewendeten Klausel die lokalen Adressen durch absolute Adressen ersetzt werden muessen (Kein Goal - Sharing !).
c) Korrektes Updaten des 'backtrackstack'
d) Weiterhin ist die Benutzung des 'goalstack' etwas irrefuehrend. Das erste Element des 'goalstack' ist immer das aktuelle goal. Beim erfolgreichen Versuch einer Klausel wird zuerst der Beweiszustand auf den 'backtrackstack' gerettet, dann erst ein pop auf den 'goalstack' ausgefuehrt.

Allerdings waere auch noch sehr viel Arbeit zu leisten, um diese Version 1.5 zu einem brauchbaren und komfortablen Werkzeug zu machen.

a) Gemischte Abarbeitung von interpretierten und compilierten Prozeduren.
b) Integration des Datenbasisindexierungskonzeptes von A. Bernardi [Bernardi 86]
c) Anpassung der Interaktionspakete TRACE und BREAK
d) Optimierung des Compilers zur Verbesserung der Laufzeit- und Speicherplatzeffizienz (z. Zeit schwankt das Verhaeltnis LISPLOG-Code : VAX-Maschinencode je nach Komplexitaet der Klauselkoepfe um ca. 1 : 40).
Diese Optimierungen koennen z.B. durch mode-Deklarationen und durch differenziertere Pattern-Erkennung bei der Uebersetzung erfolgen.
e) Automatische Transformation von LISPLOG-Rekursionen in Iterationen (oder einfache Tail-Rekursionen) durch den Compiler. Viele rekursive LISPLOG-Prozeduren sind deterministisch (z.B. member, append, sumto, Fakultaet etc.) und koennen in eine Iteration umgewandelt werden.

Beispiel:
```
(ass ! (member _x (_x . ID)))
(ass (member _x (ID . _t))
     (member _x _t))
---
;semantisch nicht ganz aequivalente Zwischenversion
(ass (member _x (_h . ID))
     (equal _x _h))
(ass (member _x (ID . _t))
     (member _x _t))
---
(do (until (or (equal _x _h) (null _t))
           (return (equal _x _h)))
    (_h <--- (car _t))
    (_t <--- (cdr _t)))
```

Teil II: Ergebnisse und Perspektiven

Die bisher im Zusammenhang mit einer Effizienzverbesserung des LISPLOG.1-Systems geleistete Arbeit, geht davon aus, dass man LISPLOG-Beweise gleichsam auf zwei unterschiedlichen Abstraktionsebenen betrachten kann.
Auf der hoeheren Abstraktionsebene sieht man den Beweis als Suche in einem Suchbaum. Man geht dabei davon aus, dass ein Teil der Ineffizienz eines PROLOG-Systems ein Ergebnis der blinden Suche im Suchbaum ist.
Zur Verbesserung des Suchverhaltens ist ein "intelligentes Backtracking" erforderlich, welches Unifikationsinformationen, die im Verlaufe eines Beweises anfallen, sammelt, geeignet repraesentiert und ausnutzt, um beim Backtracking zu einem Zustand des Beweises (Knoten im Suchbaum) zurueckzusetzen, fuer den die Bedingungen, welche zum Fehlschlagen des Beweises eines Teilzieles fuehrten, nicht vorhanden sind.
Dies koennte man nun relativ leicht implementieren. Da bei allen Variablenbindungen auch der Bindungszeitpunkt mit abgespeichert wird, koennte man nach folgender Strategie verfahren:
Falls der Beweis eines Teilzieles fehlschlaegt, bestimme das Maximum der Bindungszeitpunkte der in diesem Teilziel enthaltenen Variablen und setze zu diesem Beweiszustand zurueck.

Wie man leicht nachvollziehen kann, gehen bei einer solchen Backtrackstrategie keine Loesungen verloren; wohl aber werden aussichtslose Teilbaeume moeglichst frueh abgeschnitten.

Eine Effizienzverbesserung durch intelligentes Backtracking ist jedoch nicht linear, sondern haengt in grossem Umfang von der zu beweisenden Anfrage und dem zugrundeliegenden LISPLOG-Programm ab.
In der Praxis sind jedoch viele LISPLOG- (PROLOG-) Prozeduren deterministisch, was z.T. durch die Einfuehrung des Cut-Operators bewirkt wird. Bei diesen Programmen erzielt man durch intelligentes Backtracking keine oder nur geringe Effizienzsteigerungen.
Auf einer tieferen Abstraktionsebene (der abstrakten PROLOG-Maschine) erkennt man jedoch weitere ineffiziente Features von LISPLOG.1, die durch die neue Laufzeitumgebung und den Hornklauselcompiler verbessert wurden.
Ungefaehr 90% der Laufzeit eines durchschnittlichen LISPLOG-Programmes entfaellt auf die Funktionen zur Unifikation, Umbenennung von Variablen und zum Retrieval von Variablenwerten. Hier bietet der bereits implementierte Klauselcompiler zusammen mit einem neuen iterativen Interpreter fuer die durch den Compiler generierten Funktionen einen guten Ansatz zur Verbesserung des Laufzeitverhaltens. Hierbei werden die Klauseln durch eine

Funktion ersetzt, welche im wesentlichen in zwei Teile zerfaellt (vgl. [Herr 85]).
a) Unifikationsteil: Durch den neuen Klauselcompiler werden fuer fast alle Konstrukte im Klauselkopf Unifikationsmakros zu einem Bedingungsteil zusammengesetzt.Dadurch wird eine beschleunigte Unifikation und eine verbesserte Repraesentation der LISPLOG - Variablen ermoeglicht.
b) Aktionsteil : Bei erfolgreicher Unifikation werden in diesem Teil globale Stacks und Steuerungsvariablen des Interpreters geaendert. Damit ist auch eine CUT - Optimierung ohne zusaetzlichen Laufzeitaufwand moeglich.

Die Variablen der Klauseln werden bereits zur Compilezeit durch eine Zahl ersetzt (lokales Displacement). Zur Laufzeit erhaelt man durch Aufaddieren eines Index die Adresse der Variablen im Stack.

Erste Laufzeitvergleiche mit der LISPLOG.1-Version ergaben recht unterschiedliche Ergebnisse. Der neue Interpreter (inklusive Compiler) war, je nach Anfrage, um einen Faktor zwischen 1 und 10 schneller. Dies kann man folgendermassen erklaeren:
Bei sehr kurzen Beweisen wirkte sich der hoehere Aufwand der neuen LISPLOG-Version 1.5 fuer Verwaltung der globalen Variablen aus, insbesondere ein groesserer Aufwand fuer Garbage Collection. Laengere Beweise koennen mit dem LISPLOG.1 Interpreter nicht gefuehrt werden, s.d. die Staerke der neuen Version (konstante Zugriffszeit fuer Variablen) nicht ausgenutzt werden kann. Im Vergleich mit der neuen Version von M. Dahmen, ergaben sich aehnliche Ergebnisse (langsamer bei Beweisen mittlerer Laenge, etwas schneller bei langen Beweisen).
Diese beiden neuen Interpreter haben jedoch gemeinsam, dass man mit ihnen mindestens 10 bis 20 mal laengere Beweise fuehren kann.

Allerdings weist diese Version auch einige handfeste Nachteile auf:

1. Der Speicheraufwand fuer den erzeugten Code ist sehr hoch. Wie bereits in 1.3 erwaehnt liegt das Verhaeltnis LISPLOG - Quellcode : VAX - Maschinencode bei etwa 1 : 40. Selbst bei einfachen Klauselkoepfen ist keine nennenswerte Verbesserung festzustellen. Dies hat mehrere Ursachen:

a) Die Verwendung von LISP - Macros als Instruktionssatz. Hier musste jedoch zwischen Laufzeiteffizienz (Macros) und Speicherplatzeffizienz (Funktionen) abgewogen werden.

b) Bisher sind noch keine Mode - Deklarationen implementiert (Dies wird wahrscheinlich auch nicht mehr geschehen, da diese

Version nicht mehr weiterentwickelt wird). Dies fuehrt dazu, dass haeufig redundanter Code erzeugt wird, der bei der beabsichtigten Verwendung der Klauseln nie benoetigt werden wird.

c) Da aus Zeitgruenden nur ein eingeschraenkter Instruktionssatz (mit wenigen komplexen Instruktionen) implementiert werden konnte, koennen natuerlich viele Spezialfaelle nur mit den allgemeinen Instruktionen uebersetzt werden (z.B. spezielle Konstanten wie NIL im Klauselkopf).

2) Es wird keine Unterscheidung zwischen den verschiedenen Sorten von Variablen (temporaer/permanent, safe/unsafe, void) durchgefuehrt. Damit ist aber auch keine Optimierung des Laufzeitspeicherbedarfs, abgesehen von einer eigentlich selbstverstaendlichen CUT - Optimierung moeglich (Diese erstreckt sich natuerlich auch nur auf den initialen CUT). Zusammen mit dem hohen Speicherbedarf fuer das uebersetzte LISPLOG - Programm ergibt sich eine wesentlich hoehere Belastung des Franz-LISP - Arbeitsspeichers (nicht des Namestacks) und damit ein erhoehter Aufwand fuer Garbage Collection.

3) Die Moeglichkeiten zur Steigerung der Laufzeiteffizienz sind groesstenteils ausgereizt. Es sind nur noch geringe Verbesserungen durch eine Erweiterung des Instruktionssatzes und durch verbesserte Behandlung von Klauseln mit einer einzigen Praemisse moeglich. Die wesentliche Ineffizienz liegt aber immer noch in der Behandlung der Variablen, dem Kopieren der Praemissen und der in LISP implementierten 'Symboltabelle' (LISPLOG - 'Symboltabelle' --> Franz-Lisp - Symboltabelle --> Adresse). Bei den Variablen ist allerdings noch eine Verbesserung in der Repraesentation moeglich. Statt Variablen als Listen mit einem '?' als erstem Element, muessten diese direkt durch einen Vektorindex dargestellt werden. Die dafuer benoetigten TAG - Felder und TAG - Operationen zur Unterscheidung von Integerkonstanten koennen jedoch in LISP nicht effizient genug implementiert werden.

4) Es wurden keine prozeduralen PLM - Instruktionen (z.B. call, execute, proceed, putxx, getxx ...) implementiert. Die Gruende dafuer liegen teilweise in dem dadurch bedingten zusaetzlichem Aufwand, aber auch in den Besonderheiten von LISP als Implementierungssprache des Instruktionssatzes. PLM besteht aus einer groesseren Anzahl von elementaren Operationen, die zum Teil sehr einfache Operationen darstellen (z.B. Laden der Argumentregister) und auf Maschinenelemente (z.B. Register oder simulierte Register) zugreifen. Dies ist in LISP nur durch die Verwendung von

globale Variablen zu simulieren. Demzufolge waeren die put - Anweisungen von PLM nicht effizient in LISP zu implementieren, so dass dies durch den Bindungsmechanismus des Franz-LISP - Systems durchgefuehrt wird. Andere prozedurale Anweisungen wie z.B. CALL oder EXECUTE als direkter Aufruf der generierten LISP - Funktion haetten dann aber wieder zu Problemen mit dem Franz-LISP - Namestack gefuehrt, insbesondere bei tiefen Rekursionen.

Ganz allgemein kann man feststellen, dass die wesentlichsten Vorteile der Standardtechniken zur Compilation von PROLOG kaum ausgenutzt werden konnten. PLM ist im wesentlichen eine Abbildung von PROLOG auf konkrete Maschinenelemente (Register und Stacks), die in (portablem) LISP nicht effizient abgebildet werden koennen und profitiert von einer moeglichst effizienten Repraesentation der PROLOG - Datenstrukturen (s. obiges Beispiel der Variablenrepraesentation). Bei einer Implementierung in LISP fuehrt dies zu einem im Verhaeltnis wesentlich hoeheren Aufwand gerade bei den haeufig verwendeten elementaren Operationen, wie z.B. Test auf freie Variablen, Kopieren von Strukturen, Zugriff auf Variablenbindungen etc., welches dann in der (mangelhaften) Effizienz des gesamten Systems evident wird.

LITERATUR

[Bailey 1985] D. Bailey: The University of Salford Lisp/Prolog System. Software - Practice and Experience, 15(6), Juni 1985, S. 595-609

[Bernardi 1986] Ansgar Bernardi: Ein Indexierungsverfahren fuer LISPLOG-Datenbasen. Interner Bericht an die LISPLOG-Gruppe. Universitaet Kaiserslautern, April 1986.

[Boley & Kammermeier et al. 1985] H. Boley, F. Kammermeier u. die LISPLOG-Gruppe: LISPLOG: Momentaufnahmen einer LISP/PROLOG-Vereinheitlichung. Universitaet Kaiserslautern, FB Informatik, MEMO SEKI-85-03, August 1985. Also in: Proc. Workshop Logisches Programmieren und Lisp, TU Berlin, 17/18. Juni 1985

[Bruynooghe 1978] M. Bruynooghe: Intelligent Backtracking fo an Interpreter of Horn Clause Logic Programs. Colloqu.um on Mathematical Logic in Computer Science, Salgotarjan (Hungary), 1978.

[Bruynooghe ?] M. Bruynooghe: Analysis of Dependencies to improve the Behaviour of Logic Programs. Katholieke Universiteit Leuven B-3030 Heverlee/Belgium

[Cohen 1985] Jacques Cohen: Describing PROLOG by its Interpretation and Compilation. CACM 12/1985 Vol 28.

[Colmerauer 1985] A. Colmerauer: PROLOG in 10 Figures. CACM 12/1985 Vol 28.

[Dilger & Janson 1981] A. Janson, W. Dilger: Unifikationsgraphen fuer intelligentes Backtracking in Deduktionssystemen. Proceedings GWAI 1981, Springer Verlag.

[ESPRIT 1984] ESPRIT Project 112, Knowledge Information Management System Task 14 FROG-S Description, Nov. 1984 Draft Laboratoires de Marcoussis C.R.C.G.E. Route de Nozay France-91460 Marcoussis.

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

[Herr 1985] J. Herr: Breitensuche und Klauselcompilation fuer LISPLOG. in: Erweiterungen zu LISPLOG Universitaet Kaiserslautern, FB Informatik, MEMO SEKI-85-10, November 1985.

[Noye 1985] Jacques Noye: Notes on PLM. ECRC Technical

Report CA-8, 1985.

[Poehlmann & Petersen 1985] A. Poehlmann, K. Petersen: Implementierung des abstrakten PROLOG-Befehlssatzes PLM auf Mikroprozessoren. Siemens AG, Muenchen 1985.

[Robinson 1985] J. A. Robinson: Beyond LOGLISP: Combining Functional and Relational Programming in a Reduction Setting. Syracuse University, April 1985.

[Van Roy 1984] Peter Van Roy: A Prolog Compiler for the PLM. Report No. UCB/CSD 84/203, Computer Science Division (EECS), University of California, November 1984.

[Warren 1977] D. H. D. Warren: Implementing PROLOG - Compiling Predicate Logic Programs. University of Edinburgh DAI - Reports 39/40, 1977.

[Warren 1983] D. H. D. Warren: An Abstract PROLOG Instruction Set. SRI Technical Note 309, October 1983.

[Warren 1984a] D. S. Warren: Generating Intermediate Code Programs for PROLOG. Computer Science Department, SUNY at Stony Brook, NY 11794, March 1984.

[Warren 1984b] D. S. Warren: The Runtime Environment for a PROLOG Compiler using a copy algorithm. Computer Science Department, SUNY at Stony Brook, NY 11794, March 1984.

```
>>>>> Absynt - Paket <<<<<


(declare (special environment))

(setq absynt.l.lfns
      '(first second
              third
              fourth
              fifth
              sixth
              seventh
              rest
              twolist
              consp
              compiled-predicate
              is-user-defined
              is-primitive
              is-lisp-defined
              cut-p
              s-conclusion
              s-premises
              remove-cut
              is-p
              not-p
              is-command
              make-environment
              make-environment-1
              variable-p))

(def first
  (macro (defmacroarg)
    ((lambda (x)
             (list 'car x))
      (cadr defmacroarg))))


(def second
  (macro (defmacroarg)
    ((lambda (x)
             (list 'cadr x))
      (cadr defmacroarg))))


(def third
  (macro (defmacroarg)
    ((lambda (x)
             (list 'caddr x))
      (cadr defmacroarg))))


(def fourth
  (macro (defmacroarg)
```



```
    ((lambda (x)
             (list 'cadddr x))
      (cadr defmacroarg))))


(def fifth
  (macro (defmacroarg)
    ((lambda (x)
             (list 'car (list 'cddddr x)))

     (cadr defmacroarg))))


(def sixth
  (macro (defmacroarg)
    ((lambda (x)
             (list 'cadr (list 'cddddr x)))

     (cadr defmacroarg))))


(def seventh
  (macro (defmacroarg)
    ((lambda (x)
             (list 'caddr (list 'cddddr x)))

     (cadr defmacroarg))))


(def rest
  (macro (defmacroarg)
    ((lambda (x)
             (list 'cdr x))
      (cadr defmacroarg))))


(def twolist
  (macro (defmacroarg)
    ((lambda (l1 l2)
             (list 'cons l1 (list 'cons l2 'nil)))

     (cadr defmacroarg)
     (caddr defmacroarg))))


(def consp
  (macro (defmacroarg)
    ((lambda (expr)
             (list 'dtpr expr))
      (cadr defmacroarg))))

(defmacro compiled-predicate (goal)
  `(get (first ,goal) 'identifiers))
```

```
(defmacro is-user-defined (goal)
  `(member (car ,goal) predicates))

(def is-primitive
  (macro (defmacroarg)
    ((lambda (goal)
             (list 'member (list 'first goal) 'primitives))

     (cadr defmacroarg))))


(def is-lisp-defined
  (macro (defmacroarg)
    ((lambda (goal)
             (list 'getd (list 'first goal)))

     (cadr defmacroarg))))


(def cut-p
  (macro (defmacroarg)
    ((lambda (l-clause)
             (list 'eq
                   (list 'first (list 'first l-clause))
                   'cut))

     (cadr defmacroarg))))


(def s-conclusion
  (lambda (l-clause)
    (cond ((cut-p l-clause) (second (first l-clause)))
          (t (first l-clause)))))


(def s-premises
  (macro (defmacroarg)
    ((lambda (l-clause)
             (list 'rest l-clause))
      (cadr defmacroarg))))


(def remove-cut
  (macro (defmacroarg)
    ((lambda (l-clause)
             (list 'cons
                   (list 's-conclusion l-clause)
                   (list 's-premises l-clause)))
     (cadr defmacroarg))))


(defmacro is-p (goal)
  (list 'and (list 'consp goal)
```



```
             (list 'eq (list 'first goal) 'is)))


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


(def is-command
  (macro (defmacroarg)
    ((lambda (x)
             (list 'member
                   x
                   (list 'append
                         (list 'quote '(+ - l lisp compile-db))
                         'lisp-coms)))
     (cadr defmacroarg))))

(defmacro make-environment (anfrage)
  `(let ((environment '((bottom-of-environment))))
        (make-environment-l ,anfrage)
        environment))

(defmacro variable-p (expr)
  (list 'and (list 'consp expr)
             (list 'eq (list 'first expr) '?)))

(defun make-environment-l (anfrage)
  (cond ((atom anfrage) nil)
        ((variable-p anfrage)
         (and (null (assoc anfrage environment))
              (setq environment
                    (cons (list anfrage (ultimate-inst anfrage))
                          environment))))
        ((listp anfrage)
         (make-environment-l (first anfrage))
         (make-environment-l (rest anfrage)))


>>>>> Assert - Paket <<<<<


(setq assert.l.lfns
      '(ass ass-l
            assertimp
            assertlst
            edit
            destroy
            get-clauses
            get-db
            get-predlst
            listing
            consult
```

```
            pp-clauses
            pp-predlst
            rex
            retr-1
            abolish
            retract-plst
            retractimp
            retractlst
            tell
            save-dblst
            sort-db))

(def ass
  (nlambda (assertion)
    (ass-1 assertion)
    'ok))


(def ass-1
  (lambda (assertion)
    (let ((pred-1 (first (s-conclusion assertion))))
         (putprop
             pred-1
             (append (get pred-1 'clauses) (ncons assertion))
             'clauses)
         (setq predicates (union predicates (list pred-1))))))


(def assertimp
  (nlambda (asslist)
    (assertlst asslist)))


(def assertlst
  (lambda (asslist)
    (cond ((null asslist) predicates)
          (t (ass-1 (first asslist))
             (assertlst (rest asslist))))))


(def edit
  (nlambda (predlist)
    (edite (get (first predlist) 'clauses)
           nil
           (first predlist))))


(def destroy
  (lambda nil
    (retract-plst predicates)
    (setq predicates nil)
    ))
```



```
(def get-clauses
  (lambda (clauseslst)
    (cond ((null clauseslst) nil)
          (t
           (cons (cons 'ass (first clauseslst))
                 (get-clauses (rest clauseslst)))))))


(def get-db
  (lambda nil
    (get-predlst predicates)))


(def get-predlst
  (lambda (predlist)
    (cond ((null predlist) nil)
          (t
           (append (get-clauses (get (first predlist) 'clauses))
                   (get-predlst (rest predlist)))))))


(def consult
  (nlambda (dat-nam-lst)
    (cond ((null dat-nam-lst)
           (apply (function dskin) '(system.wrk.db)))
          ( t (apply (function dskin) dat-nam-lst)))))


(def pp-clauses
  (lambda (clauseslst p-port)
    (cond ((null clauseslst) (terpr p-port))
          (t (pp-external-form
              (cons 'ass (first clauseslst)) p-port)
             (pp-clauses (rest clauseslst) p-port)))))

(def listing
  (nlambda (p-nam-list)
    (cond ((null p-nam-list) (pp-predlst predicates nil))
          (t (pp-predlst p-nam-list nil)))))

(def pp-predlst
  (lambda (predlist p-port)
    (cond ((null predlist) (terpr p-port))
          (t (pp-clauses (get (first predlist) 'clauses) p-port)
             (pp-predlst (rest predlist) p-port)))))


(def rex
  (nlambda (retr-clause)
    (retr-1 retr-clause)
    'ok))
```

```
(def retr-1
  (lambda (retr-clause)
    (let ((pred-1 (first (s-conclusion retr-clause))))
         (let ((clauses-left
                 (without (get pred-1 'clauses) retr-clause)))
              (cond ((null clauses-left)
                     (remprop pred-1 'clauses)
                     (setq predicates
                           (without predicates pred-1)))
                    (t (putprop pred-1 clauses-left 'clauses)
                       predicates)))))))


(def abolish
  (nlambda (predlist)
    (setq predicates (retract-plst predlist))
    'ok))


(def retract-plst
  (lambda (predlist)
    (cond ((null predlist) predicates)
          (t (remprop (first predlist) 'clauses)
             (without (retract-plst (rest predlist))
                      (first predlist))))))


(def retractimp
  (nlambda (retr-list)
    (retractlst retr-list)))


(def retractlst
  (lambda (retr-list)
    (cond ((null retr-list) predicates)
          (t (retr-1 (first retr-list))
             (retractlst (rest retr-list))))))


(def tell
  (nlambda (dat-nam-list)
     (cond ((null dat-nam-list) (save-dblst '(system.wrk.db) ))
           ( t (save-dblst dat-nam-list)))))


(def save-dblst
  (lambda (dat-nam-list)
    (cond ((null dat-nam-list) nil)
          (t (let ((p-port (outfile (first dat-nam-list))))
                  (pp-predlst predicates p-port)
                  (close p-port))
             (save-dblst (rest dat-nam-list))))))
```



```
(def sort-db
  (lambda nil
    (setq predicates (sort predicates nil))))


>>>>> Compiler - Paket <<<<<


(defmacro multiple-occurrence-p (pattern liste)
  `(greaterp (smemb ,pattern ,liste 0) 1))

(defun create-1 (clause-head-left clause-head param-list)
   (cond ((multiple-occurrence-p (first clause-head-left)
                                 clause-head)
          (append `((univar ,(first param-list)
                    ,(quote-clause (first clause-head-left))))
                  (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))))))

(defun create-2 (clause-head-left clause-head param-list)
   (append `((unistruct
               ,(first param-list)
               ,(quote-clause (first clause-head-left))
               ,(create-struct-condition
                   (first clause-head-left)
                   clause-head
                   'x)))
             ; X ist formaler parameter der lambda-expr.
             ; innerhalb von unistruct
           (create-conditions (rest clause-head-left)
                              clause-head
                              (rest param-list))))

(defun create-3 (clause-head-left clause-head param-list)
   (append `((uniconst
               ,(first param-list)
               ,(quote-clause (first clause-head-left))))
           (create-conditions (rest clause-head-left)
                              clause-head
                              (rest param-list))))

(def create-conditions
  (lambda (clause-head-left clause-head param-list)
      (cond ((member (first param-list) '(&optional &rest))
             (create-conditions clause-head-left
                                clause-head
```

```
                        (rest param-list)))
              ((null clause-head-left)
               nil)
              ((eq clause-head-left 'ID) nil)
              ((variable-p clause-head-left)
               `((univar ,(first param-list)
                         ,(quote-clause clause-head-left))))
              ((variable-p (first clause-head-left))
               (create-1 clause-head-left
                         clause-head
                         param-list))
              ((null (first clause-head-left))
               (create-3 clause-head-left
                         clause-head
                         param-list))
              ((listp (first clause-head-left))
               (create-2 clause-head-left
                         clause-head
                         param-list))
              ((eq (first clause-head-left) 'ID)
               (create-conditions (rest clause-head-left)
                                  clause-head
                                  (rest param-list)))
              ((atom (first clause-head-left))
               (create-3 clause-head-left
                         clause-head
                         param-list)))))


(def create-struct-condition
  (lambda (clause-head-left clause-head param)
       (cond ((null clause-head-left)
              nil)
             ((eq clause-head-left 'ID) nil)
             ((variable-p clause-head-left)
              `((univar ,param
                        ,(quote-clause clause-head-left))))
;dotted-pair-cdr matcht rest einer liste
             ((variable-p (first clause-head-left))
              (append
               (append
                `((ustructvar ,param
                              ,(quote-clause
                                (first clause-head-left))))
                       (and (rest clause-head-left)
                            (list
                             `(or (setq ,param (cdr ,param))
                             t))))
                (create-struct-condition
                   (rest clause-head-left)
                   clause-head
                   param)))
             ((eq (first clause-head-left) 'ID)
```



```
              (append
                 (and (rest clause-head-left)
                      `((or (setq ,param (cdr ,param)) t)))
                 (create-struct-condition
                    (rest clause-head-left)
                    clause-head
                    param)))
             ((atom (first clause-head-left))
                    (append
                      (append
                         `((ustructconst
                                ,param
                                ,(quote-clause
                                     (first clause-head-left))))
                           (and (rest clause-head-left)
                                (list `(or (setq ,param (cdr ,param))
                                       t))))
                       (create-struct-condition
                          (rest clause-head-left)
                          clause-head
                          param)))
             ((listp (first clause-head-left))
              (append (append
                       `((ustructstruct
                          ,param
                          ,(quote-clause (first clause-head-left))
                          ,(create-struct-condition
                              (first clause-head-left)
                              clause-head
                              param)))
                         (and (rest clause-head-left)
                              (list `(or (setq ,param (cdr ,param))
                                     t))))
                       (create-struct-condition
                          (rest clause-head-left)
                          clause-head
                          param))))))


(defun rename (term)
  (cond ((assoc term displace-list)
         (list '?
               (let ((disp (second (assoc term displace-list))))
                    (cond ((zerop disp)
                           'index)
                          (t (list 'add 'index disp))))))
        (t (setq displace-list (cons (list term displace)
                                     displace-list))
           (setq displace (add1 displace))
           (list '? (list 'add 'index (sub1 displace))))))

(defun rename-vars (term)
  (let ((displace-list nil))
       (rename-vars-1 term)))
```

```
(defun rename-vars-1 (term)
  (cond ((variable-p term)
         (cond ((assoc term displace-list)
                (list '? (second (assoc term displace-list))))
               (t (setq displace-list (cons (list term displace)
                                            displace-list))
                  (setq displace (add1 displace))
                  (list '? (sub1 displace)))))
        ((atom term) term)
        (t (cons (rename-vars-1 (first term))
                 (rename-vars-1 (rest term)))))

(defun rename-comp-vars (term)
  (let ((displace-list nil)
        (clause-head (rest (s-conclusion term))))
       (let ((clause-head
              (rename-clause-head clause-head clause-head 0)))
            (rename-comp-vars-1 term clause-head))))

(defun rename-clause-head (clause-head-left clause-head level)
  (cond ((variable-p clause-head-left)
         (cond ((lessp level 2)
                (cond ((multiple-occurrence-p clause-head-left
                                              clause-head)
                       (rename clause-head-left))
                      (t clause-head-left)))
               (t (rename clause-head-left))))
        ((atom clause-head-left)
         clause-head-left)
        (t (cons (rename-clause-head (first clause-head-left)
                                     clause-head (add1 level))
                 (rename-clause-head (rest clause-head-left)
                                     clause-head level)))))

(defun rename-comp-vars-1 (term clause-head)
  (cond ((and (variable-p term)
              (member term clause-head))
         term)
        ((variable-p term)
         (rename term))
        ((atom term) term)
        (t (cons (rename-comp-vars-1 (first term) clause-head)
                 (rename-comp-vars-1 (rest term) clause-head)))))

(def smemb
     (lambda (pattern liste anzahl)
             (cond ((equal pattern liste)(add1 anzahl))
                   ((atom liste) anzahl)
                   ((equal pattern (first liste))
                    (smemb pattern (rest liste) (add1 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 displace)
          (let ((cut-flag (cut-p clause))
                (head (rest (s-conclusion clause)))
                (premisses (s-premises clause)))
               (let ((conditions
                      (cond ((and (null head) param-list)
                             '((uniconst $argument1 nil)))
                            (t (create-conditions head
                                                  head
                                                  param-list)))))
                    (deffunc identifier
                             param-list
                             conditions
                             (and premisses
                                  (cons 'list
                                        (quote-premisses
                                              premisses)))
                             (cond (cut-flag 'det-action)
                                   (t 'action))
                             displace))
                    (putprop (first (s-conclusion clause))
                             (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 '(declare
                                  (special backtrackstack
                                           variablenstack
                                           anwendbareklausein
                                           trail
                                           level
                                           index
                                           success
                                           goalstack)
```

```
                                (macros t))
                               p-port)
                        (terpri p-port)
                        (print '(include lisplog-runtimes)
                               p-port)
                        (terpri p-port)
                        (print (list 'include file2)
                               p-port)
                        (terpr p-port)
                        (close p-port))))
      t))

(def compile-proc
  (lambda (predicate)
     (let ((clause-list (get predicate 'clauses)))
          (remprop predicate 'identifiers)
          (let ((info (parse-all-clause-heads clause-list)))
               (compile-proc-1 clause-list
                               (concat predicate '-lisp-)
                               0
                               (list-of-args (first info)
                                             (second info)
                                             (third info)
                                             (second info))
                               0)
               (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 displace)
          (cond ((null clause-list) nil)
                (t
                 (let ((clause
                        (rename-comp-vars (first clause-list))))
                      (compile-clause clause
                                      param-list
                                      (concat identifier number)
                                      displace))
                 (compile-proc-1 (rest clause-list)
                                 identifier
                                 (addl number)
                                 param-list
                                 0)))))
```



```
(def list-of-args
  (lambda (minimumarg maximumarg lexpr-flag number)
          (cond ((zerop number)
                 (and (zerop minimumarg)
                      (list '&optional)))
                (t (append (list-of-args minimumarg
                                         maximumarg
                                         lexpr-flag
                                         (subl number))
                           (cond ((and (eq number maximumarg)
                                       lexpr-flag)
                                  (list '&rest
                                        (concat
                                         (concat '$ 'argument)
                                         number)))
                                 ((and
                                   (greaterp maximumarg
                                             minimumarg)
                                   (eq number minimumarg))
                                  (list  '&optional
                                         (concat
                                          (concat '$ 'argument)
                                          number)))
                                 (t (list (concat
                                           (concat '$ 'argument)
                                           number)))))))))

(defun quote-premisses (term)
  (cond ((null term) nil)
        ((atom term) term)
        ((is-user-defined term)
         (quote-clause term))
        ((is-primitive term)
          (cond ((is-p term)
                 (quote-clause (list 'is
                                     (second term)
                                     (quote-lambda-arg
                                         (third term)))))
                (t (quote-clause term))))
        ((and (symbolp (first term))
              (is-lisp-defined term))
         (quote-clause (quote-lambda-arg term)))
        (t (cons (quote-premisses (first term))
                 (quote-premisses (rest term]

(defun quote-clause (clause)
  (cond ((null clause) nil)
        ((member clause param-list)
         clause)
        ((atom clause)
         (list 'quote clause))
        ((variable-p clause)
         (list 'list '(quote ?) (second clause)))
```

```
        (t (cons 'list (quote-sublist clause)))))

(defun quote-sublist (clause)
  (cond ((atom clause) nil)
        ((variable-p clause)
         (list '? (second clause)))
        (t (cons (quote-clause (first clause))
                 (quote-sublist (rest clause]

(defun deffunc
         (identifier param-list conditions
          premisses action-macro displace)
  (cond ((null conditions)
         (eval `(defun ,identifier ,param-list
                       (,action-macro ,displace ,premisses))))
        ;Keine Zuruecknahme von Variablenbindungen, da
        ;keine Bedingungen, die solche Bindungen erzeugen
        ;koennten vorhanden sind
        (t
         (eval `(defun ,identifier ,param-list
                       (cond ((and ,@conditions)
                              (,action-macro
                               ,displace
                               ,premisses))
                             (t (mapcar
                                 (function
                                   (lambda (x)
                                     (vset variablenstack
                                           x
                                           nil)))
                                 trail]


>>>>> Extern - Paket <<<<<


(setq extern.1.lfns
      '(external-form external-form-of-var
                      pp-external-form
                      pp-form-without-escape
                      print-external
                      print-neu
                      print-without-escape
                      printimp-without-escape))

(defmacro external-form-of-var (var)
  `(uconcat '"_" (second ,var)))

(def external-form
  (lambda (term)
    (cond ((atom term) term)
          ((variable-p term) (external-form-of-var term))
```



```
          (t (cons (external-form (first term))
                   (external-form (rest term)))))))


(def pp-external-form
  (lexpr (numargs)
    (let ((expr (arg 1))
          (port (cond ((eq numargs 2) (arg 2)) (t nil))))
      (pp-form-without-escape (external-form expr) port))))


(def pp-form-without-escape
  (lexpr (numargs)
    (let ((expr (arg 1))
          (port (cond ((eq numargs 2) (arg 2)) (t nil))))
      (putd 'print-alt (getd 'print))
      (putd 'print (getd 'print-neu))
      (let ((result (pp-form expr port)))
           (putd 'print (getd 'print-alt))
           result)))))


(def print-external
  (lexpr (numargs)
    (let ((expr (arg 1))
          (port (cond ((eq numargs 2) (arg 2)) (t nil))))
      (print-without-escape (external-form expr) port))))


(def print-neu
  (lexpr (numargs)
    (let ((expr (arg 1))
          (port (cond ((eq numargs 2) (arg 2)) (t nil))))
      (cond ((atom expr) (patom expr port))
            ((null (rest expr))
             (patom '"(" port)
             (cond ((atom (first expr))
                    (patom (first expr) port))
                   (t (print-alt (first expr) port)))
             (patom '")" port))
            ((atom (rest expr))
             (patom '"(" port)
             (cond ((atom (first expr)) (patom (first expr)
                                               port))
                   (t (print-alt (first expr) port)))
             (patom '" . " port)
             (patom (rest expr) port)
             (patom '")" port))
            (t (print-alt expr port))))))


(def print-without-escape
  (lexpr (numargs
```

```
      (let ((expr (arg 1))
            (port (cond ((eq numargs 2) (arg 2)) (t nil))))
           (cond ((atom expr) (patom expr port) nil)
                 (t (patom '"(" port)
                    (printimp-without-escape expr port)
                    (patom '")" port)
                    nil)))))


(def printimp-without-escape
  (lexpr (numargs)
    (let ((exprlst (arg 1))
          (port (cond ((eq numargs 2) (arg 2)) (t nil))))
         (cond ((null (rest exprlst))
                (print-without-escape (first exprlst) port))
               ((atom (rest exprlst))
                (print-without-escape (first exprlst) port)
                (patom '" . " port)
                (patom (rest exprlst) port))
               (t (print-without-escape (first exprlst) port)
                  (patom '" " port)
                  (printimp-without-escape (rest exprlst) port]



>>>>> Interface - Paket <<<<<


(setq interface.l.lfns
      '(lisplog commands
                lisp-coms
                y-or-n-p))

(def commands
  (lambda (inline)
       (let ((com (first inline))
             (intern-flag nil)
             (displace 0))
            (cond ((is-command com)
                   (cond ((eq com '+)
                          (apply (function ass) (rest inline)))
                         ((eq com '-)
                          (apply (function rex) (rest inline)))
                         ((eq com 'l)
                          (apply (function listing)
                                 (rest inline)))
                         ((eq com 'lisp)
                          (setq leave-prolog t))
                         ((eq com 'compile-db)
                          (compile-db))
                         ((member com lisp-coms)
                          (apply com (rest inline)))))))
```



```
                  (t (setq inline (rename-vars inline))
                     (catch (and-process inline displace]

(setq lisp-coms
      '(ass assertimp
            rex
            retractimp
            abolish
            destroy
            listing
            edit
            tell
            consult))

(def y-or-n-p
  (lambda (message)
    (patom message)
    (let ((response (read)))
         (cond ((eq response 'y) (backtrack))
               ((eq response 'n) (setq abbruch t))
               (t (y-or-n-p message))))))


(def lisplog
  (lambda nil
    (do ((leave-prolog nil))
        (leave-prolog)
        (print '*)
        (pp-external-form (commands (lineread nil)))
        (terpr))))


>>>>> Lispeval - Paket <<<<<


(setq lispeval.l.lfns
      '(lisp-predicates quoted-p
                        lambda-appl-p
                        nlambda-appl-p
                        Macro-appl-p
                        application-p
                        discipline
                        contains-freevars
                        quote-lambda-arg
                        quote-nlambda-arg
                        quote-application))

(defmacro quoted-p (expr)
  (list 'and (list 'consp expr)
             (list 'eq (list 'first expr) 'quote)))
```

```
(def lambda-appl-p
  (macro (defmacroarg)
    ((lambda (expr)
             (list 'cond
                   (list (list 'atom expr) nil)
                   (list t
                         (list 'or
                               (list 'eq
                                     (list 'discipline
                                           (list 'first expr))
                                     'lambda)
                               (list 'eq
                                     (list 'discipline
                                           (list 'first expr))
                                     'lexpr)))))

     (cadr defmacroarg))))


(def nlambda-appl-p
  (macro (defmacroarg)
    ((lambda (expr)
             (list 'cond
                   (list (list 'atom expr) nil)
                   (list t
                         (list 'eq
                               (list 'discipline
                                     (list 'first expr))
                               'nlambda))))

     (cadr defmacroarg))))


(def Macro-appl-p
  (macro (defmacroarg)
    ((lambda (expr)
             (list 'cond
                   (list (list 'atom expr) nil)
                   (list t
                         (list 'eq
                               (list 'discipline
                                     (list 'first expr))
                               'macro))))

     (cadr defmacroarg))))


(def application-p
  (lambda (expr)
    (or (lambda-appl-p expr) (nlambda-appl-p expr))))


(def discipline
```



```
  (lambda (funct)
    (cond ((numberp funct) nil)
          ((atom funct)
           (let ((def (getd funct)))
                (cond ((null def) nil)
                      ((atom def) (getdisc def))
                      (t (first def)))))
          (t (first funct)))))


(def contains-freevars
  (lambda (expr)
    (cond ((or (atom expr) (quoted-p expr) (nlambda-appl-p expr))
               nil)
          ((variable-p expr) t)
          (( ambda-appl-p expr)
           (some (function contains-freevars) (rest expr))))))

(defun lisp-predicates (goal)
  (setq goal (ultimate-inst goal))
  (cond ((contains-freevars goal) nil)
        (t (let ((res (errset (eval goal))))
                (cond ((null res)
                       (throw (twolist
                        'error--lisp-predicates:eval-not-possible
                                goal)))
                      ((first res)
                       (setq goalstack (rest goalstack))
                       t)
                      (t nil]

(def quote-lambda-arg
  (lambda (expr)
    (let ((expr (expand-macro expr)))
         (cond ((or (null expr) (numberp expr)) expr)
               ((application-p expr) (quote-application expr))
               (t (twolist 'quote expr)))))))


(def quote-nlambda-arg
  (lambda (expr)
    (let ((expr (expand-macro expr)))
         (cond ((atom expr) expr)
               ((application-p expr) (quote-application expr))
               (t (mapcardot
                   (function quote-nlambda-arg) expr)))))))


(def quote-application
  (lambda (expr)
    (cond ((or (variable-p expr) (quoted-p expr)) expr)
          ((lambda-appl-p expr)
           (cons (first expr)
```

```
                (mapcar (function quote-lambda-arg)
                        (rest expr))))
          ((nlambda-appl-p expr)
           (cons (first expr)
                 (mapcar (function quote-nlambda-arg) (rest expr]

(defun expand-macro (expr)
  (cond ((macro-appl-p expr) (macroexpand expr))
        (t expr)))


>>>>> N-solutions - Paket <<<<<



(def n-solutions
  (lambda (goal anzahl)
    (let ((all-environment nil)
          (intern-flag t)
          (displace 0))
         (cond ((null goal) t)
               ((equal anzahl 0) nil)
               ((listp goal)
                (setq goal (rename-vars goal))
                (and-process (list goal) displace)
                all-environment]


>>>>> Parser - Paket fuer den Compiler <<<<<



(declare (special lexpr-flag))

(def parse-all-clause-heads
  (lambda (clause-list)
          (let ((minimumarg
                 (length (s-conclusion (first clause-list))))
                (maximumarg
                 (length (s-conclusion (first clause-list))))
                (lexpr-flag nil))
               (parse-clause-heads clause-list
                                   minimumarg
                                   maximumarg]

(def parse-clause-heads
  (lambda (clauses-left
           minimumarg
           maximumarg)
          (cond ((null clauses-left)
                 (list (subl minimumarg)
                       (subl maximumarg)
                       lexpr-flag))
```



```
                (t
                 (let ((anzahl
                        (parse-head
                         (s-conclusion (first clauses-left)))))
                      (cond ((lessp anzahl minimumarg)
                             (setq minimumarg anzahl))
                            ((greaterp anzahl maximumarg)
                             (setq maximumarg anzahl)))
                      (parse-clause-heads (rest clauses-left)
                                          minimumarg
                                          maximumarg]

(def parse-head
  (lambda (clause-head &optional (anzahl 0))
          (cond ((null clause-head)
                 anzahl)
                ((or (atom clause-head)
          ;Test auf dotted-pair
                     (variable-p clause-head))
          ;Keine anderen Listen
                 (setq lexpr-flag t)
                 (addl anzahl))
                (t (parse-head (rest clause-head)
                               (addl anzahl))))))



>>>>> Primitives - Paket <<<<<



(setq primitives.1.lfns
      '(prolog-primitive execute-not-intern
                         execute-is
                         primitives))

(def prolog-primitive
  (lambda (goal)
    (setq goalstack (rest goalstack))
    (cond ((is-p goal)
           (execute-is (second goal)
                       (third goal)))
          ((not-p goal)
           (execute-not-intern goal))
          ((eq (first goal) 'var)
           (var-p (first (rest goal))))
          ((eq (first goal) 'nonvar)
           (non-var-p (first (rest goal)))))))


(def execute-not-intern
  (lambda (goal)
    (let ((all-environment nil) (anzahl 1) (intern-flag t)
```

```
          (displace 0))
         (setq goal (rename-vars
                     (ultimate-inst (cadr goal))))
         (and-process (list goal) displace)
         (cond (all-environment nil) (t t)))))


(def execute-is
  (lambda (variable lispexpr)
       (setq lispexpr (ultimate-inst lispexpr))
       (cond ((contains-freevars lispexpr) nil)
             (t
              (let ((res (errset (eval lispexpr)))
                    (trail nil))
                   (cond ((null res)
                          (throw
                           (twolist
                            'error--execute-is:eval-not-possible
                                  lispexpr)))
                         ((univar variable (first res))
                          (and (greaterp level 0)
                               (nconc trail
                                      (car (vref backtrackstack
                                                 (sub1 level))))
                               (rplaca (vref backtrackstack
                                             (sub1 level))
                                       trail))
                          (setq index (add1 index]

(setq primitives
      '(is not var nonvar))


>>>>> Andor - Paket <<<<<


(defmacro backtrack nil
  `(cond ((zerop level) (setq abbruch t))
         (t (setq level (sub1 level))
            (let ((liste (vref backtrackstack level)))
                 (do ((todo (first liste)
                            (rest todo)))
                     ((null todo))
                     (vset variablenstack (first todo) nil))
                 (setq goalstack (second liste))
                 (setq anwendbareklauseln (third liste))
                 (setq index (fourth liste)))))))

(defmacro execute-compiled-clauses
          (goalargpattern anwendbareklauseln)
  `(prog (success failure)
         (return
```



```
          (do ((trail nil nil)
               (fun (first ,anwendbareklauseln)
                    (first ,anwendbareklauseln))
               (,anwendbareklauseln (rest ,anwendbareklauseln)
                                    (rest ,anwendbareklauseln)))
              ((or success failure) success)
              (or anwendbareklauseln
                  (setq failure t))
              (apply fun ,goalargpattern)))))

(defun and-process (anfrage index)
  (let ((variablenstack (new-vector 5000))
        (backtrackstack (new-vector 1000))
        (goalstack anfrage)
        (abbruch nil)
        (level 0)
        (anwendbareklauseln (get (first (first anfrage))
                                 'identifiers)))
       (do ((goal (ultimate-partinst (first goalstack))
                  (ultimate-partinst (first goalstack))))
           (abbruch)
           (cond ((null goalstack)
                  (cond (intern-flag
                         (setq all-environment
                               (cons (make-environment anfrage)
                                     all-environment))
                         (cond ((eq anzahl 1)
                                (setq abbruch t))
                               (t (setq anzahl (sub1 anzahl)))))
                        (t (princ "success")
                           (terpri)
                           (print-bindings
                              (make-environment anfrage))
                           (y-or-n-p "More? (y or n) "))))
                 ((null
                   (cond ((atom goal) goal)
                         ((compiled-predicate goal)
                          (execute-compiled-clauses
                           (rest goal)
                           anwendbareklauseln))
                         ((is-primitive goal)
                          (prolog-primitive goal))
                         ((is-lisp-defined goal)
                          (lisp-predicates goal))))
                  (backtrack))
                 (t (setq anwendbareklauseln
                          (get (first (first goalstack))
                               'identifiers]


>>>>> Unifikationsmacros <<<<<
```

```
(defun ultimate-assoc (x)
  (cond ((variable-p x)
         (let ((binding
                (vref variablenstack (address-of x))))
              (cond ((null binding) x)
                    (t (ultimate-assoc (first binding))))))
        (t x)))

(defun ultimate-inst (x)
  (cond ((variable-p x)
         (let ((binding (vref variablenstack
                              (address-of x))))
              (cond ((null binding) x)
                    (t (ultimate-inst (first binding))))))
        ((atom x) x)
        (t (cons (ultimate-inst (first x))
                 (ultimate-inst (rest x))))))

(defun ultimate-partinst (x)
  (let ((x (ultimate-assoc x)))
       (cond ((atom x) x)
             (t (cons (ultimate-assoc (first x))
                      (rest x))))))
(defmacro univar (goalpattern clausepattern)
  `((lambda (x y)
       (cond ((equal x y))
             ((variable-p y)
              (put-binding (address-of y) x level))
             ((variable-p x)
              (put-binding (address-of x) y level))))
    (ultimate-assoc ,goalpattern)
    (ultimate-assoc ,clausepattern)))

(defmacro uniconst (goalpattern clausepattern)
   `((lambda (x)
        (or (eq x ,clausepattern)
            ;uniconst nur bei Atom in clausepattern
            (and (variable-p x)
                 (put-binding (address-of x)
                              ,clausepattern
                              level))))
     (ultimate-assoc ,goalpattern)))

(defmacro unistruct (goalpattern clausepattern conditions)
  `((lambda (x)
            (cond ((variable-p x)
                   (put-binding (address-of x)
                                ,clausepattern
                                level))
                  ((and (not (atom x))
                        ,@conditions))))
    (ultimate-assoc ,goalpattern)))
```



```
(defmacro ustructvar (goalpattern clausepattern)
  `(univar (car ,goalpattern) ,clausepattern))

(defmacro ustructconst (goalpattern clausepattern)
  `(uniconst (car ,goalpattern) ,clausepattern))

(defmacro ustructstruct (goalpattern clausepattern conditions)
  `(unistruct (car ,goalpattern) ,clausepattern ,conditions))

(defmacro action (displace premisses)
  `(prog nil
         (cond (anwendbareklauseln
                (vset backtrackstack level
                      (list trail
                            goalstack
                            anwendbareklauseln
                            index))
                (setq level (add1 level)))
               ((greaterp level 0)
                (nconc trail
                       (car (vref backtrackstack (sub1 level))))
                (rplaca (vref backtrackstack (sub1 level))
                        trail)))
         (setq goalstack (append ,premisses (cdr goalstack)))
         (setq index (add index ,displace))
         (setq success t)))

(defmacro det-action (displace premisses)
  `(prog nil
         (and (greaterp level 0)
              (nconc trail (car (vref backtrackstack
                                      (sub1 level))))
              (rplaca (vref backtrackstack (sub1 level))
                 trail))
         (setq goalstack (append ,premisses (cdr goalstack)))
         (setq index (add index ,displace))
         (setq success t)))

(defmacro put-binding (x y level)
  `(prog nil
         (vset variablenstack ,x
               (list ,y ,level))
         (setq trail (cons ,x trail))
         (return t)))

(defmacro address-of (x)
  `(eval (cadr ,x)))


>>>>> Utilities - Paket <<<<<


(setq utilities.l.lfns
```

```
        '(mapcardot some
                    remlist
                    setcons
                    union
                    without
                    expand-macro))

(def mapcardot
  (lambda (funct dotlst)
    (cond ((atom dotlst) dotlst)
          (t (cons (funcall funct (first dotlst))
                   (mapcardot funct (rest dotlst]


(def some
  (lambda (funct lst)
    (cond ((null lst) nil)
          ((funcall funct (first lst)) lst)
          (t (some funct (rest lst))))))


(def remlist
  (lambda (l1 l2)
    (cond ((null l1) l2)
          (t (remlist (rest l1) (remove (first l1) l2))))))


(def setcons
  (lambda (elem set)
    (cond ((member elem set) set) (t (cons elem set)))))


(def union
  (lambda (s1 s2)
    (cond ((null s1) s2)
          ((member (first s1) s2) (union (rest s1) s2))
          (t (cons (first s1) (union (rest s1) s2))))))


(def without
  (lambda (x y)
    (cond ((null x) nil)
          ((equal (first x) y) (rest x))
          (t (cons (first x) (without (rest x) y))))))


(def expand-macro
  (lambda (expr)
    (cond ((macro-appl-p expr) (macroexpand expr))
          (t expr))))
```



```
>>>>> Variablen - Paket <<<<<


(setq variable.l.lfns
      '(var-p non-var-p print-bindings))

(def non-var-p
  (lambda (x)
    (not (var-p x))))

(def var-p
  (lambda (x)
    (variable-p (ultimate-assoc x))))

(defun print-bindings (environment)
  (cond ((equal (first environment) '(bottom-of-environment))
         nil)
        (t
         (pp-external-form
          (list (caar environment) " = " (cadar environment)))
         (terpri)
         (print-bindings (rest environment)))))

(def consp
  (macro (defmacroarg)
    ((lambda (expr)
             (list 'dtpr expr))
     (cadr defmacroarg))))

(defmacro variable-p (expr)
  (list 'and (list 'consp expr)
             (list 'eq (list 'car expr) '?)))


>>>> Lisplog - runtimes Paket <<<<<


(defmacro put-binding (x y level)
  `(prog nil
         (vset variablenstack ,x
               (list ,y ,level))
         (setq trail (cons ,x trail))
         (return t)))

(defmacro address-of (x)
  `(eval (cadr ,x)))

(defun ultimate-assoc (x)
  (cond ((variable-p x)
         (let ((binding (vref variablenstack (address-of x))))
```

```
                (cond ((null binding) x)
                      (t (ultimate-assoc (car binding))))))
          (t x)))

(include unimac.3.1)
```



```
(setq bsp.lfns
      '(append-lisp-0 append-lisp-1
                      sumto-lisp-0
                      sumto-lisp-1
                      beispiel1-lisp-0
                      beispiel1-lisp-1
                      beispiel2-lisp-0))

(! (append nil _l _l))

-->

(def append-lisp-0
  (lambda ($argument1 $argument2 $argument3)
    (cond ((and (uniconst $argument1 nil)
                (univar $argument2 (list '? index))
                (univar $argument3 (list '? index)))
           (det-action 1 nil))
          (t
           (mapcar (function
                    (lambda (x)
                            (vset variablenstack x nil))
                    )
                   trail)))))

((append (_head . _tail1) _l (_head . _tail2)) (append _tail1 _l _tail2

-->

(def append-lisp-1
  (lambda ($argument1 $argument2 $argument3)
    (cond ((and (unistruct $argument1
                           (list (list '? index)
                                 '?
                                 (add index 1))
                           ((ustructvar x
                                        (list '? index))
                            (or (setq x (cdr x)) t)
                            (univar x
                                    (list '?
                                          (add index 1)))))
                (unistruct $argument3
                           (list (list '? index)
                                 '?
                                 (add index 2))
                           ((ustructvar x
                                        (list '?
                                              index))
                            (or (setq x (cdr x)) t)
                            (univar x
                                    (list '?
                                          (add index 2))))))
           (action 3
```

```
                        (list
                         (list 'append
                               (list '? (add index 1))
                               $argument2
                               (list '? (add index 2))))))
                (t
                 (mapcar (function
                          (lambda (x)
                                  (vset variablenstack x nil))
                          )
                         trail)))))

(! (sumto 1 1))

-->

(def sumto-lisp-0
  (lambda ($argument1 $argument2)
    (cond ((and (uniconst $argument1 '1)
                (uniconst $argument2 '1))
           (det-action 0 nil))
          (t
           (mapcar (function
                    (lambda (x)
                            (vset variablenstack x nil))
                    )
                   trail)))))

((sumto _n _res)(is _n1 (sub1 _n))(sumto _n1 _res1)(is _res (plus _n _re

-->

(def sumto-lisp-1
  (lambda ($argument1 $argument2)
    (action 2
            (list (list 'is
                        (list '? (add index 0))
                        (list 'sub1
                              (list 'quote $argument1)))
                  (list 'sumto
                        (list '? index)
                        (list '? (add index 1)))
                  (list 'is
                        $argument2
                        (list 'plus
                              (list '? (add index 1))
                              (list 'quote
                                    $argument1)))))))

>>>>>>>>>> Beispiel fuer verschiedene Stelligkeiten <<<<<<<<<<<

((beispiel1 _x _y))
((beispiel1 _x _y _z))
```



```
-->

(def beispiel1-lisp-0
  (lexpr (g00008)
    (declare (*args 1 3))
    ((lambda ($argument1 $argument2 $argument3)
             (cond ((> 2 g00008) (setq $argument2 nil))
                   (t (setq $argument2 (arg 2)) nil))
             (cond ((> 3 g00008) (setq $argument3 nil))
                   (t (setq $argument3 (arg 3)) nil))
             (action 0 nil))

     (arg 1)
     nil
     nil)))


(def beispiel1-lisp-1
  (lexpr (g00009)
    (declare (*args 1 3))
    ((lambda ($argument1 $argument2 $argument3)
             (cond ((> 2 g00009) (setq $argument2 nil))
                   (t (setq $argument2 (arg 2)) nil))
             (cond ((> 3 g00009) (setq $argument3 nil))
                   (t (setq $argument3 (arg 3)) nil))
             (action 0 nil))

     (arg 1)
     nil
     nil)))

>>>>>>>>>> Beispiel fuer variable Stelligkeit <<<<<<<<<<

((beispiel2  x . _y))

-->

(def beispiel2-lisp-0
  (lexpr (g00010)
    (declare (*args 1 nil))
    ((lambda ($argument1 $argument2)
             (cond ((> 2 g00010) (setq $argument2 nil))
                   (t (setq $argument2 (arg 2)) nil))
             (let* (($argument3
                     (do ((g00011 g00010 (1- g00011))
                          (g00012 nil
                                  (cons (arg g00011) g00012)))
                         ((< g00011 3) g00012))))
                   (cond ((and
                           (univar $argument2
                                   (list '?
                                         (add index 0))))
                          (action 1 nil))
```

```
                                (t
                                 (mapcar (function
                                          (lambda (x)
                                                  (vset variablenstack
                                                        x
                                                        nil)))
                                         trail)))))

        (arg 1)
        nil)))
```