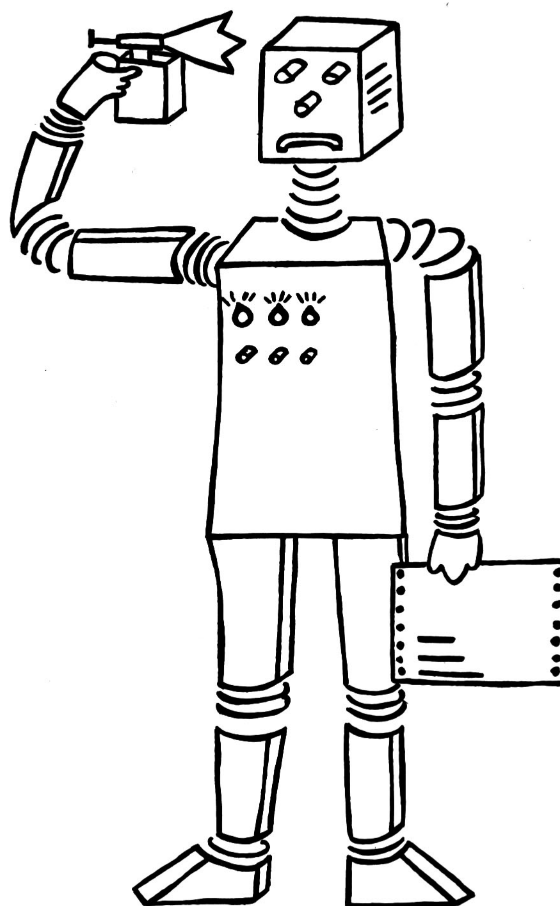
SEKI-Working Paper

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



Uebersetzung von LISPLOG-Programmen nach CPROLOG

Knut Hinkelmann

September 1986 SWP-86-05

UEBERSETZUNG VON LISPLOG-PROGRAMMEN NACH CPROLOG

Knut Hinkelmann

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

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

SWP-86-05

September 1986

ABSTRACT

A LISPLOG program is a set of horn clauses. Pure PROLOG's proof strategy is a kind of resolution. In LISPLOG, however, LISP predicates can be used like PROLOG goals. These will be proved by evaluating the LISP functions instead of using the resolution theorem prover. To translate a LISPLOG program into CPROLOG, these function applications have to be transformed into applications of PROLOG relations. In order to prove these relations by resolution, new horn clauses have to be added to the database. These new horn clauses are created from the definition of the corresponding LISP function.

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

Inhalt:
----

1. Einleitung.......................................3

2. Grundideen der Uebersetzung.......................4
   2.1 Klauseln......................................4
      2.1.1 Konklusion...............................4
      2.1.2 Praemissen...............................5
   2.2 Terme...5
      2.2.1 Konstanten...............................5
      2.2.2 Variable.................................6
      2.2.3 Strukturen...............................6
         2.2.3.1 Selbstdefinierte Praedikate.........7
         2.2.3.2 LISP-Praedikate.....................7
         2.2.3.3 LISPLOG-Primitive...................8
         2.2.3.4 Datenlisten........................10
   2.3 Sonderfaelle.................................11
      2.3.1 Die LISP-Funktion progn.................11
      2.3.2 Ausgabe.................................11
      2.3.3 Eingabe.................................13
      2.3.4 Funktionen zur Steuerung der Ausgabe....14
      2.3.5 Aufrufe der Funktionen and, or und cond als Praemissen..............................14
      2.3.6 Die Funktionen ass und rex zum Aendern der Datenbasis.............................15
   2.4 Umwandlung von LISP-Funktionsdefinitionen mit Seiteneffekten durch implizites und explizites progn....16
   2.5 Einschraenkungen.............................17

3. Benutzeroberflaeche..............................21

4. Implementierung..................................22
   4.1 Hauptfunktionen..............................22
   4.2 Hilfsfunktionen..............................24
   4.3 Variablen....................................26
   4.4 Transformation von LISP nach PROLOG..........27

5. Transformation von LISP-Funktionen und LISPLOG-Datenbasen nach CPROLOG mit allgemeinem Cut-Operator..........31

Literatur...32

Anhang A: Listing...................................33

Anhang B: Listing der LISP-PROLOG-Transformation....45

Anhang C: Listing der LISP-PROLOG-Transformation mit Cut.......71

1. Einleitung

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

Auch fuer die andere Uebersetzungsrichtung von CPROLOG nach LISPLOG existiert ein Translator [Dahmen 1985].

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

Fuer LISP-Funktionen, die als Goals von LISPLOG-Klauseln vorkommen, ist nur von Interesse, ob sie den Wert 'nil' haben oder nicht. Solche Funktionen werden im folgenden als LISP-Praedikate bezeichnet. Bei LISP-Funktionen, die in Aufrufe von LISP-Praedikaten eingebettet oder als Argumente des Primitivs 'is' aufgerufen werden, ist der Wert von Bedeutung. Diese Funktionen werden im folgenden algemeine LISP-Funktionen genannt.

Der hier vorgestellte Uebersetzer greift auf die Programme zur Abflachung von geschachtelten LISP-Funktionen und zur Umwandlung von LISP-Funktionsdefinitionen in LISPLOG-Klauseln aus [Hinkelmann & Morgenstern 1985] zurueck. Er ist eine Weiterentwicklung des in [Boley & Kammermeier et. al 1985] vorgestellten noch recht eingeschraenkten LISPLOG-CPROLOG-Translators.

2. Grundideen der Uebersetzung

Das Uebersetzungsprogramm wird nach dem Laden des LISPLOG-Systems aufgerufen. Eine zu uebersetzende LISPLOG-Datenbasis ist eine Datei von Klauseln. Diese Datei wird vom Programm geladen, das nun eine interne Darstellung der Datenbasis als Liste von Klauseln erzeugt. Die Klauseln werden nacheinander uebersetzt und auf eine Datei ausgegeben. Klauseln koennen Regeln oder Fakten sein. Ein Fakt ist in der internen Form eine Liste mit einer Struktur (der Konklusion) als einzigem Element. Eine Regel ist eine Liste mit einer Struktur (der Konklusion) als erstem Element gefolgt von einem oder mehreren Termen (den Praemissen). Terme sind entweder Konstanten, Variablen oder Strukturen.

Im folgenden sind mit der Bezeichnung 'Prozedur p' alle die Klauseln gemeint, deren Konklusion als Praedikatsname p haben.

2.1 Klauseln

Klauseln sind entweder Fakten oder Regeln. Regeln werden nach folgendem Muster uebersetzt: Zuerst erfolgt die Uebersetzung der Konklusion. Die Konklusion wird von den Praemissen durch ':-' getrennt. Anschliessend werden die Praemissen uebersetzt. Sie werden durch Kommata getrennt und die gesamte Klausel durch einen Punkt abgeschlossen.
Fakten haben keine Praemissen, so dass die Klausel nach der Uebersetzung der Konklusion direkt mit einem Punkt abgeschlossen wird.

Beispiel: Eine LISPLOG-Regel habe die Form

```
(<ll-konklusion> <ll-praemisse1> <ll-praemisse2>)
```

Diese wird zu

```
<cprolog-konklusion> :-
        <cprolog-praemisse1>,
        <cprolog-praemisse2>.
```

2.1.1 Konklusion

Als Konklusionen treten natuerlich nur Strukturen mit selbstdefinerten Praedikaten auf, die nach 2.2.3.1 uebersetzt werden. Ein Sonderfall bei der Uebersetzung der Konklusion ist die Syntax, die in LISPLOG fuer den Cut gewaehlt wurde. Der Cut, der in LISPLOG nur als initialer Cut vorkommt, wird nicht als

Praemisse, sondern als Teil der Konklusion aufgefasst. Die um den Cut erweiterte Konklusion hat folgende Syntax:

(cut <lisplog-konklusion>)

Das bedeutet, dass das Zeichen ':-', das CPROLOG-Konklusionen von den Praemissen trennt, in diesem Fall schon bei der Uebersetzung der Konklusion erzeugt werden muss. Man erhaelt also:

<cprolog-konklusion> :- !

2.1.2 Praemissen

Wie schon erwaehnt, sind die Praemissen einer Regel Terme. Die Praemissen werden nacheinander uebersetzt und auf die Zieldatei, jeweils 10 Stellen eingerueckt, ausgegeben. Durch diese Notation werden die Praemissen des CPROLOG-Programms sichtbar von der Konklusion abgehoben. Zur Uebersetzung der Terme siehe 2.2.

2.2 Terme

Terme werden verschieden uebersetzt, je nachdem, ob sie als Top-level-Element einer Klausel (Konklusion oder Praemisse) oder als Elemente von Strukturen auftreten.

2.2.1 Konstanten

Konstanten sind LISP-Atome. Tritt das Atom 'nil' als Komponente einer Struktur auf, so wird es als leere Liste interpretiert und somit in '[]' umgewandelt. Tritt eine andere Konstante als Komponente einer Struktur auf, so wird sie unveraendert uebernommen. Man sollte deshalb darauf achten, dass in dem zu uebersetzenden LISPLOG-Programm nur solche Symbole auftreten, die auch in CPROLOG erlaubt sind (vgl. [Clocksin & Mellish 1981/84]).

Das Goal 'nil' wird als 'fail' uebersetzt. Tritt eine Konstante ungleich 'nil' als Goal auf, so wird sie mit 'true' uebersetzt, da sie in LISPLOG direkt zum Erfolg fuehrt.

2.2.2 Variable

LISPLOG-Variable haben hier die interne Darstellung als Zweierlisten mit dem Fragezeichen als erstem und dem Variablennamen als zweitem Element. Bei der Uebersetzung wird der Name uebernommen, der an einen Unterstrich konkateniert wird.

Beispiel: (? var) --> _var

Die Moeglichkeit, den ersten Buchstaben durch den entsprechenden Grossbuchstaben zu ersetzen, wurde verworfen, da z.B. 3x, $res gueltige LISP-Atome sind, die dann nicht zu uebersetzen waeren. Auf diese Weise wurde zwar die Zahl der uebersetzbaren Variablennamen erweitert, aber der Anwender muss beachten, dass so auch Symbole uebernommen werden, die in CPROLOG nicht erlaubt sind (z.B. *res, #r, %x)!

Die anonyme Variable ID wird mit '_' uebersetzt.

Wird eine Variable als Goal aufgerufen, so muss sie vor ihrem Beweis an einen Term gebunden werden. Dazu muss sie vorher innerhalb einer Struktur auftreten. Ist der Term eine Datenliste, wie sie zum Beispiel bei der Uebersetzung einer eingebetteten Struktur erzeugt werden kann (siehe Beispiel 3 in 2.2.3.4), so muss diese vor dem Beweis in eine CPROLOG-Struktur umgewandelt werden. Wurde die Variable an eine CPROLOG-Struktur gebunden (z.B. durch eine Benutzeranfrage), so ist eine Umwandlung nicht noetig. Eine Variable als Praemisse wird deshalb wie folgt uebersetzt:

```
(? var) --> '=..-if-list'(_var,_resl), _resl
```

wobei die zur Umwandlung benoetigte CPROLOG-Prozedur an das Programm angefuegt wird:

```
'=..-if-list'([X|Y],Z) :- !, Z =.. [X|Y].

'=..-if-list'(X,X).
```

Vgl. dazu auch Abschnitt 4 unter 2.5.

2.2.3 Strukturen

LISPLOG-Strukturen sind Listen oder dotted-lists. Die Art der Umwandlung einer Struktur haengt ab vom dem Ort ihres Auftretens und von ihrem ersten Element.

```
-->  _res001 is (3 + (2 * _x)),
(2)  length(_y,_res002),
     _res001 > _res002
```

Dabei ist length eine CPROLOG-Prozedur, die aus der lambda-Definition der LISP-Funktion length erzeugt und an das resultierende CPROLOG-Programm angefuegt wird.

Zu dem Abflachungsalgorithmus ist noch folgendes zu sagen: Die eingebetteten Aufrufe von LISP-Funktionen werden zu Relationsaufrufen mit einem zusaetzlichen Argument (siehe im Beispiel length). Eine Ausnahme bilden die arithmetischen Funktionen. Sie werden nicht abgeflacht und in Relationsaufrufe umgewandelt, sondern stehen als Argumente des Primitivs is mit einer neuen Variable fuer den Wert (siehe im Beispiel plus).

Es gibt auch LISP-Funktionen, die als Praemissen vorkommen und die ohne Abflachung direkt in eine CPROLOG-Entsprechung umgewandelt werden koennen:

```
(atom (? x))     --> atomic(_x)
(litatom (? x)) --> atom(_x)
(numberp (? x)) --> number(_x)
```

Die Uebersetzung von Sonderfaellen wie Ein-/Ausgabefunktionen oder die Funktion progn werden in 2.3 behandelt.

2.2.3.3 LISPLOG-Primitive

a) Das is-Primitiv hat in LISPLOG die Syntax '(is <variable> <lisp-ausdruck>)', wobei <lisp-ausdruck> nicht wie in CPROLOG auf arithmetische Ausdruecke beschraenkt ist. Eine solche Struktur kann nur als Praemisse auftreten. Die Uebersetzung erfolgt in 2 Schritten.

   1. Die Struktur wird in eine Liste von Relationsaufrufen abgeflacht (siehe [Hinkelmann & Morgenstern 1985]).

   2. Uebersetzung dieser Relationsaufrufe als Konjunktion von Praemissen.

   Die nach dem Abflachen in dieser Liste auftretenden is-Primitive enthalten als <lisp-ausdruck> nur arithmetische Funktionsaufrufe. Diese koennen dann direkt uebersetzt werden, wobei die Aufrufe der arithmetischen LISP-Funktionen in vollstaendig geklammerte Ausdruecke mit den entsprechenden CPROLOG-Opteratoren uebersetzt werden. Sonstige in der Liste vorkommende Relationsaufrufe sind Aufrufe von selbstdefinierten Praedikaten und werden nach 2.2.3.1 uebersetzt.

Beispiele:

```
(is (? x) (plus 3 (times 4 (sin (? y)))))

    --> ((is (? x) (plus 3 (times 4 (sin (? y))))))
    (1)
    --> _x is (3 + (4 * (sin(_y))))
    (2)

(is (? x) (length (? y))) --> ((length (? y) (? x)))
                              (1)
                              --> length(_y,_x)
                              (2)

(is (? x) (plus (length (? y)) 5))

    --> ((length (? y) (? r001)) (is (? x) (plus (? r001) 5)))
    (1)
    --> length(_y,_r001), _x is (_r001 + 5)
    (2)
```

Fuer die LISP-Funktion length muss aus ihrer lambda-Defintion noch eine CPROLOG-Prozedur length erzeugt werden.

b) Die arithmetischen Primitive dienen zur Ausfuehrung der arithmetischen Grundoperationen. Sie koennen nur als Praemissen auftreten und haben die Form (<arith> <op1> <op2> <erg>), wobei <arith> eines der Praedikate addit, subit, multit oder divit ist. Bei der Uebersetzung werden sie in den Aufruf des CPROLOG-Praedikates is umgewandelt.

Beispiel: `(addit 4 (? x) (? y) --> _y is 4 + _x`

c) Die metalogischen Primitive var und nonvar koennen direkt uebernommen werden. Die Uebersetzung erfolgt analog zu 2.2.3.1.

d) Bei der Uebersetzung des Primitivs not gibt es zwei Moeglichkeiten:

1. Ist das Argument eine LISP-Funktion, so wird die gesamte Struktur in eine Liste von Relationsaufrufen abgeflacht (a). Im zweiten Schritt werden die Relationen uebersetzt (b).

2. Im anderen Fall wird das Argument des Primitivs uebersetzt und nach CPROLOG-Art negiert.

Beispiele:

1. Sei member nicht selbstdefiniert sondern eine LISP-Funktion:

```
(not (member (? a) (cdr (? l))))

--> ((cdr (? l) (? rl)) (not (member (? a) (? rl))))
(a)

--> cdr(_l,_rl), not(member(_a,_l))
(b)
```

2. Sei sohn ein selbstdefiniertes Praedikat:

```
(not (sohn john (? x)))  -->  not(sohn(john,_x))
```

2.2.3.4 Datenlisten

Datenlisten sind <u>alle</u> in selbstdefinierte Praedikate eingebettete Strukturen. Datenlisten sind ebenfalls alle in LISP-Funktionen eingebettete Strukturen, deren erstes Element keine LISP-Funktionsdefinition hat (d.h. der Teil, der nach der Abflachung noch eingeschachtelt uebrigbleibt). Datenlisten werden immer in CPROLOG-Listen mit eckigen Klammern uebersetzt.

Beispiele:

1. Sei member als LISP-Funktion definiert und a habe keine Funktionsdefinition:

   ```
   (member (? x) (a b c d)) --> member(_x,[a,b,c,d])
   ```

2. Sei sohn ein selbstdefiniertes Praedikat und sei member als LISP-Funktion definiert:

   ```
   (sohn (? x) (member (? y) (? l))) --> sohn(_x,[member,_y,_l])
   ```

3. Seien vater und sohn selbstdefinierte Praedikate:

   ```
   (vater (? x) (sohn (? y) (? z))) --> vater(_x,[sohn,_y,_z])
   ```

   Wuerde das zweite Argument mit einer Variable unifiziert werden, die danach als Praemisse auftritt, so muesste sie im CPROLOG-Programm zuerst in eine Struktur umgewandelt werden, wie es in 2.2.2 beschrieben ist.

2.3 Sonderfaelle

2.3.1 Die LISP-Funktion progn

Tritt die LISP-Funktion progn als LISPLOG-Praedikat auf, so werden einfach ihre Argumente als Konjunktion von Praemissen uebersetzt. Dabei darf es sich bei den n-1 ersten Argumenten nur um Funktionen handeln, deren Aequivalente in CPROLOG zu 'success' fuehren. Als Beispiel betrachte die Ausgabefunktion print, die in LISP zwar den Wert 'nil' hat, deren CPROLOG-Aequivalent write aber zu 'success' fuehrt. Fuer das n-te Argument gilt diese 'success'-Bedingung nicht, da ein dort entstehender nil-Funktionswert auch bei der LISPLOG-Interpretation zu 'failure' fuehren soll. Die Uebersetzung erfolgt analog zur Uebersetzung eines Aufrufs von progn als Rumpf einer Funktionsdefinition (vgl. Abschnitt 2.4).

2.3.2 Ausgabe

In CPROLOG gibt es 3 Ausgabefunktionen, naemlich put fuer Character sowie write und display fuer Terme. In LISPLOG stehen fuer die Ausgabe die entsprechenden LISP-Funktionen zur Verfuegung. Bei der Uebersetzung werden die Argumente, die den Ausgabeport angeben, nicht beachtet, da die Dateiorganisation von CPROLOG und LISPLOG bei der Ausgabe voneinander abweichen (siehe dazu auch Punkt 3 unter 2.5). Die LISP-Ausgabefunktionen werden alle mit 'write' uebersetzt. Da innerhalb der LISP-Ausgabefunktionen andere LISP-Funktionen eingebettet sein koennen, muss die LISP-Funktion im allgemeinen zuerst in eine Liste von Relationsaufrufen abgeflacht werden (vgl. 2.2.3.2), deren letztes Element der Aufruf der LISP-Ausgabefunktion ist. Die Uebersetzung dieses Funktionsaufrufs erfolgt analog zu der Uebersetzung selbstdefinierter Praedikate mit folgenden Einschraenkungen:

- Das Praedikat wird nicht einfach uebernommen, sondern durch 'write' ersetzt.

- Tritt als Argument des abgeflachten Ausgabefunktionsaufrufs ein String auf, so wird dieser in 'single quotes' eingeschlossen.

Das hat folgenden Grund: Durch write werden Strings als Listen mit den ASCII-Codes der Zeichen ausgegeben. Um eine lesbare Ausgabe zu erhalten, fasst man deshalb den String als Atom auf, indem man ihn in 'single quotes' einschliesst.

Spezialfaelle:

- Die LISP-Funktion patom liefert als Resultat den Wert des Argumentes, das ausgegeben wird. Wird der Wert nil ausgegeben, so fuehrt der Aufruf in LISPLOG zu einem 'failure', ausser wenn der Aufruf innerhalb eines progn erfolgt.

  a) Beim Aufruf von patom als Praemisse, wird der Aufruf abgeflacht. Anschliessend muss eine zusaetzliche Praemisse eingefuegt werden, die testet, ob das Argument den Wert 'nil' hat. Auf diese Weise wird gewaehrleistet, dass bei einer Ausgabe von 'nil' auch das CPROLOG-Programm einen 'failure' erzeugt. Diese zusaetzliche Praemisse wird nicht erzeugt, wenn der Aufruf als eines der n-1 ersten Argumente innerhalb von progn auftritt (vgl. 2.3.1).

     Beispiel: (patom (? x))

     --> ((patom (? x)) (not (null (? x))))

     --> write(_x), not(null(_x))

  b) Ist der Aufruf von patom in den Aufruf einer anderen LISP-Funktion eingebettet, so wird der gesamte Funktionsaufruf wie in 2.2.3.2 beschrieben abgeflacht und uebersetzt. Aus dem LISP-Funktionsaufruf patom wird nun allerdings ein Relationsaufruf patom mit einem zusaetzlichen Argument fuer den Wert des Aufrufs. Fuer den Beweis von patom wird eine zusaetzliche Klausel zu dem CPROLOG-Programm hinzugefuegt.

     Beispiel: (greaterp 3 (patom (? x)))

     --> ((patom (? x) (? rl)) (greaterp 3 (? rl)))

     --> patom(_x,_rl), 3 > _rl

     mit einer zusaetzlichen Klausel:

     patom(X,X) :- write(X).

     (Das LISPLOG-Goal '(greaterp 3 (patom (? x)))' ist erfuellt, wenn (? x) als Seiteneffekt ausgedruckt wurde und 3 groesser ist als das Argument (? x). Das CPROLOG-Goal 'patom(_x,_r)' wird durch die neue Klausel immer erfuellt, die die neue Variable _rl an den Wert von _x bindet und _x ausgibt.)

- Die LISP-Funktion print liefert unabhangig von ihrem Argument den Wert 'nil'. Da write in CPROLOG aber immer erfuellt wird, wird ein print-Aufruf als LISPLOG-Goal wie im folgenden Beispiel uebersetzt.

  Beispiel: (print (? x)) --> write(_x), fail

Tritt der Aufruf als eines der n-1 ersten Argumente in progn auf, so entfaellt das 'fail'. Das 'fail' entfaellt ebenfalls, wenn der Aufruf negiert ist, denn die Praemisse liefert 't', was als 'success' interpretiert wird.

2.3.3 Eingabe

LISPLOG greift fuer die Eingabe auf die LISP-Eingabefunktionen zurueck. Diese haben als Argument hoechstens die Angabe des Ports. Dieses wird allerdings bei der Uebersetzung nicht beachtet, da die Dateiorganisation bei CPROLOG der von LISPLOG nicht entspricht (siehe dazu auch Punkt 3 unter 2.5). Die LISP-Funktionen read und ratom werden mit 'read' uebersetzt. Der Anwender des resultierenden CPROLOG-Programms muss allerdings darauf achten, dass die Eingabe der CPROLOG-Konvention entspricht (z.B. '.' als Abschluss der Eingabe).

Die LISP-Funktion readc zur Eingabe einzelner Zeichen wird mit 'get0' uebersetzt. Da bei Erfuellung des CPROLOG-Goals 'get0(N)' N mit dem ASCII-Code des gelesenen Zeichens instanziiert ist, muss dieser Wert noch durch die Relation 'name(X,[N])' umgewandelt werden (siehe dazu [Pereira ohne Datum]).

Die Eingabefunktionen koennen in einem LISPLOG-Programm mit zwei verschiedenen Bedeutungen auftreten:

a) Im Normalfall ist ihr Aufruf eingebettet in den Aufruf anderer LISP-Funktionen oder in den Aufruf des Primitivs is, wo ihr Wert direkt verwendet wird. Diese Funktionsaufrufe werden wie in 2.2.3.2 bzw. 2.2.3.3 beschrieben in eine Liste von Relationsaufrufen abgeflacht. Dabei werden allerdings die Namen der Eingabefunktionen fuer die entsprechenden Relationsaufrufe beibehalten. Danach werden die Relationsaufrufe uebersetzt.

```
Beispiele:   (greaterp 3 (read))

                --> ((read (? rl)) (greaterp 3 (? rl)))
                (1)
                --> read(_rl), 3 > _rl
                (2)


             (is (? x) (readc))

                --> ((readc (? x)))
                (1)
                --> get0(_rl), name(_x,[_rl])
                (2)
```

b) Tritt ein Aufruf einer der Funktionen read oder ratom als Praemisse auf, kann mit einem nil-Wert als Eingabe das Praedikat von aussen zum Scheitern gebracht werden. (Fuer readc nicht sinvoll). Die Relation read hat in CPROLOG ein Argument. Bei der Uebersetzung wird daran die Argumentvariable als zusaetzliches Goal angefuegt, das bei Eingabe 'fail' dann ebenfalls zu einem 'failure' fuehrt.

Beispiel: Die Klausel

```
((abbruch) (read))
```

wird zu

```
abbruch :-
          read(R1), R1.
```

2.3.4 Funktionen zur Steuerung der Ausgabe

Aufrufe der LISP-Funktionen terpr und terpri werden mit 'nl,fail' uebersetzt, da diese als Wert nil haben. Treten sie allerdings als Argumente von progn auf, so werden sie mit 'nl' uebersetzt, das in CPROLOG immer erfuellt wird. Ein Aufruf der Funktion tab wird in einen Aufruf des CPROLOG Primitivs tab uebersetzt, das allerdings eine andere Semantik hat (siehe dazu auch Punkt 3 unter 2.5).
Auch hier werden die Argumente, die den Ausgabeport der LISP-Funktion angeben, nicht beachtet.

2.3.5 Aufrufe der Funktionen and, or und cond als Praemissen

Treten Aufrufe der LISP-Funktionen and, or oder cond als Praemissen auf, so koennte man das CPROLOG-Programm dem LISPLOG-Programm durch die Trennzeichen ';' und ',' angleichen. Das entsprechende CPROLOG-Programm wuerde dann allerdings schlecht lesbar. Ausserdem waere die klare Darstellung als Hornklauselmenge durchbrochen. Aus diesem Grund wird eine LISPLOG-Klausel, die Aufrufe dieser Funktionen enthaelt, so umgewandelt, dass daraus unter Umstaenden mehrere Hornklauseln entstehen, die allerdings die gleichen Ergebnisse liefern. Man erreicht dies, indem man die Praemissen in disjunktive Normalform umwandelt und aus jedem Disjunkt eine Klausel bildet (vgl. [Hinkelmann & Morgenstern 1985]). Man nimmt dabei allerdings in Kauf, dass das entstehende Programm ineffizienter ist.

Betrachte dazu folgendes Beispiel:

```
        ((test (? a) (? b))
         (greaterp (? a) (? b))
         (or (eq (? a) 3)
             (and (eq (? a) 5) (lessp (? b) 3))
             (greaterp (? a) 4))
         (bed (? a) (? b)))

    --> test(_a,_b) :- _a>_b, _a=3, bed(_a,_b).
        test(_a,_b) :- _a>_b, _a=5, _b<3, bed(_a,_b).
        test(_a,_b) :- _a>_b, _a>4, bed(_a,_b).
```

Sei _a=5, _b=2 und bed(_a,_b) scheitere. Dann wird in der LISPLOG-Klausel jede Praemisse einmal berechnet. In CPROLOG dagegen wird _a>_b und bed(_a,_b) dreimal zu erfuellen versucht. Sei es andererseits so, dass bei _a=5 und _b=2 bed(_a,_b) erfuellt werden kann, aber mit diesen Werten das nachfolgende Goal in der "aufrufenden" Klausel scheitert. Dann wird in CPROLOG die Prozedur test zweimal erfuellt und das nachfolgende Goal mit den gleichen Werten fuer _a und _b zum zweimal vergebens probiert.

Ein anderer Fall, der bei der Uebersetzung zu mehreren Horn-Klauseln fuehren kann, ist der, dass die Funktion cond als Argument der LISPLOG-Relation is aufgerufen wird.

Beispiel:

```
       ((test (? a) (? b) (? c))
        (is (? c) (cond ((eq (? b) 0) 1)
                        ((greaterp (? b) 0)
                         (divide (? a) (? b)))
                        (t (minus (divide (? a) (? b)))))))

   --> test(_a,_b,_c) :- _b =:= 0, _c = 1.     (* Unifikation *)
       test(_a,_b,_c) :- not(_b =:= 0),
                         _b>0,
                         _c is _a/_b.
       test(_a,_b,_c) :- not(_b =:= 0),
                         not(_b > 0),
                         true,
                         _c is -(_a/_b).
```

2.3.6 Die Funktionen ass und rex zum Aendern der Datenbasis

Die LISP-Funktionen ass und rex dienen zum dynamischen Aendern der Datenbasis. Sie werden mit 'assertz' und 'retract' uebersetzt. Ass und rex haben als Argumente eine variable Anzahl von Termen. Das erste Argument ist die Konklusion und der Rest

sind die Praemissen der zu assertierenden bzw. zu loeschenden Klausel. Die Uebersetzung einer Klausel (dem cdr des Funktionsaufrufs) erfolgt wie in 2.2 angegeben. Bei einer solchen Klausel kann die Konklusion allerdings eine Variable sein (vgl. dazu Punkt 4 in 2.5). Durch Aufrufe der Funktionen or und cond als Praemissen koennen aus einer Regel mehrere Relationsaufrufe von assertz und retract entstehen. Zu beachten ist, dass assertz und retract nur ein Argument fuer die zu behandelnde Klausel haben. CPROLOG-Regeln werden deshalb in zusaetzliche Klammern eingeschlossen, damit sie als ein Argument aufgefasst werden koennen.
Wichtig: Beachte dazu auch Abschnitt 4 unter 2.5.

Beispiele:

```
(rex (? x))  -->  retract(_x)

(ass (? x) (princ (? y)))  --> assertz((_x :- write(_y)))

(ass (testl a (? x)) (? v) (princ (? x)))

     --> assertz((testl(a,_x) :-
                  '=..-if-list'(_v,_x001), _x001,
                  write(_x)))

(rex (elternteil (? x) (? y))
     (or (vater (? x) (? y)) (mutter (? x) (? y))))

     --> retract((elternteil(_x,_y) :-
                  vater(_x,_y))),
         retract((elternteil(_x,_y) :-
                  mutter(_x,_y)))

(ass ((? y) a (? x)) (? v) (princ (? x)))

     --> "error!!!",
         da eine Variable (? y) als Praedikat vorkommt!
```

2.4 Umwandlung von LISP-Funktionsdefinitionen mit Seiteneffekten durch implizites und explizites progn

Das Programm zur Umwandlung von LISP-Funktionsdefinitionen in PROLOG-Relationen aus [Hinkelmann & Morgenstern 1985] hatte unter anderem die Einschraenkung, dass nur Funktionen umgewandelt werden konnten, deren Rumpf aus einem einzigen Funktionsaufruf bestand (kein implizites progn). Auch ein expliziter Aufruf der Funktion progn war nicht moeglich. Diese Einschraenkungen werden

durch dieses Programm beseitigt.

Es musste neben allgemeinen Funktionen und Praedikaten ([Hinkelmann & Morgenstern 1985]) eine dritte Art der LISP-Funktionen eingefuehrt werden. Besteht naemlich ein Funktionsrumpf aus einer Folge von n Funktionsaufrufen, so wird der Wert der Funktion nur durch den letzten Funktionsaufruf bestimmt, waehrend die ersten n-1 Aufrufe nur Seiteneffekte ausfuehren (vgl. progn in LISPLOG-CPROLOG-Translator). Werden diese nach PROLOG uebersetzt, so muessen die herauskommenden Relationen immer erfuellt werden. Diese Funktionen werden im folgenden prog-Funktionen genannt. Ihr Name wird durch Anhaengen von '$p' gekennzeichnet.

Es ist auch moeglich, Funktionen zu uebersetzen, deren Rumpf aus einem Aufruf der Funktion progn besteht. Auch die Uebersetzung geschachtelter Aufrufe von progn ist moeglich.

2.5 Einschraenkungen

In folgenden Faellen kann die Semantik der eingegebenen LISPLOG-Datenbasis von derjenigen der herauskommenden CPROLOG-Datenbasis abweichen:

1. In LISPLOG ist es moeglich, Strukturen als Punkt-Listen der Form (p a1 a2 ... aN . a) zu schreiben, so dass z.B. das Praedikat p durch eine einzige Klausel mit einer solchen Struktur als Konklusion fuer eine variierende Aritaet >= N (N >= 0) aktueller Argumente definiert werden kann. In CPROLOG sind dagegen nur Strukturen mit fester Aritaet erlaubt. Der Uebersetzter uebersetzt obige Struktur zwar in die Form p(a1, a2, ..., aN, a), gibt aber eine Warnung aus, dass beim Aufruf nun nur genau N+1 Argumente erlaubt sind.

   Hinweis: Es gibt zwei Moeglichkeiten, solche Strukturen unter Beibehaltung der Semantik zu uebersetzen:

   a) Man kann die Argumente von Strukturen mit variabler Argumentzahl als CPROLOG-Listen schreiben, z.B. 'p([a1,a2, ... ,aN|a])'. Weil dies aber fuer jedes Auftreten des Praedikates p notwendig waere, also auch dann, wenn die Argumentzahl fest ist, muesste man vor dem eigentlichen Uebersetzen einen Durchlauf durch die gesamte LISPLOG-Datenbasis machen, um zu erkennen, welche Praedikate mit variabler Argumentzahl vorkommen.

   b) Man kann <u>alle</u> Strukturen mit gelisteten Argumenten uebersetzen. Diese doppelte Klammerung mit '([' bzw. '])' aller Strukturen der CPROLOG-Datenbasis ist allerdings

unueblich und ungewoehnlich.

Da solche Strukturen aber nicht sehr oft vorkommen, lohnt der Aufwand nicht. Dem Anwender bleibt somit die Aufgabe, die notwendigen Aenderungen der herauskommenden CPROLOG-Datenbasis von Hand durchzufuehren.

2. Fuer gewisse, als LISPLOG-Primitive verwendete LISP-Funktionen (numberp, atom, litatom, equal, eq, greaterp, >, >&, lessp, <, <&, =, =&), werden entsprechende CPROLOG-Praedikate erzeugt. Zu einer grossen Zahl von LISP-Funktionen (z.B. listp, null, zerop usw. sowie benutzerdefinierte Funktionen) steht in CPROLOG dagegen kein einfaches Aequivalent zur Verfuegung; Wenn moeglich, generiert der Uebersetzer die notwendigen CPROLOG-Klauseln. Dies gilt allerdings nur fuer nicht-compilierte pure lambda-Funktionen, die zur Zeit der Uebersetzung geladen sein muessen (siehe [Hinkelmann & Morgenstern 1985]).

3. Zur Ein- und Ausgabe koennen in LISP und somit auch in LISPLOG mehrere Ports gleichzeitig goeffnet sein, so dass ohne Umschaltung auf mehrere Dateien geschrieben bzw. von verschiedenen Dateien gelesen werden kann. In CPROLOG kann allerdings zu jeder Zeit immer nur eine Datei zum Lesen oder Schreiben geoeffnet sein. Die Funktionsaufrufe zum Oeffenen und Schliessen von Dateien werden von dem Uebersetzer zwar in Relationsaufrufe abgeflacht aber dann nicht weiter behandelt. Dem Anwender des Uebersetzungsprogramms ist daher zu empfehlen, seine Dateiorganisation nach dem Uebersetzen das Programms an CPROLOG anzupasssen.

   In andere Funktionsaufrufe eingebettete Aufrufe der Ausgabefunktionen werden im Moment noch nicht richtig uebersetzt, da in der jetzigen Fassung des Programms nicht eindeutig festgestellt werden kann, ob das letzte Argument ein Port ist oder den Wert des Funktionsaufrufs annehmen soll. Dieser Fall kommt allerdings selten vor und kann durch Nacheditieren leicht korrigiert werden.

   Zu beachten ist auch, dass nach Aufruf der LISP-Funktion tab die Ausgabe an der als Argument angegebenen Stelle weitergeht. Im Gegensatz dazu bewirkt die CPROLOG-Relation tab, dass die Ausgabe um die als Argument angegebene Zahl von Stellen weiterrueckt. Es gibt allerdings keine CPROLOG-Relation, die zur LISP-Funktion tab aequivalent ist.

4. Wie in 2.3.6 erwaehnt, haben ass und rex eine variable Anzahl von Argumenten, waehrend deren CPROLOG-Entsprechungen assertz und retract genau ein Argument haben, naemlich die Klausel. Um eine LISPLOG-Regel zu assertieren, gibt es 3 Moeglichkeiten:

   - Man gibt jede Praemisse explizit an.

- Falls man die Anzahl der Praemissen kennt, aber einige Praemissen erst waehrend eines Beweises berechnet werden, werden an diese Stellen Variablen eingefuegt, an die die spaeter zu berechnende Praemisse gebunden wird.

- Falls man die Anzahl der Praemissen nicht kennt, kann man ass als dotted-pair aufrufen, wobei an die Variable nach dem Punkt eine Liste mit beliebig vielen Praemissen gebunden werden kann.
  Beispiel: (ass (test (? x)) . (? y))

Man beachte, dass ein Aufruf dieser dritten Art nicht uebersetzt werden kann und zu einem Abbruch des Uebersetzungsprogramms fuehrt!

Bei der Uebersetzung von ass und rex (2.3.6) kann als Konklusion eine Variable auftreten. Wird diese waehrend eines Beweises mit einer Datenliste unifiziert, so kann die Datenliste in LISPLOG als Konklusion interpretiert werden, was bei deren Uebersetzung nicht moeglich ist. Im Gegensatz zur Uebersetzung einer Goal-Variable kann diese aber nicht durch den CPROLOG-Operator '=..' in eine Struktur umgewandelt werden. Fuer den Benutzer ist deshalb Vorsicht geboten. Er sollte eine Datenliste, die waehrend eines Beweises als Konklusion an eine Variable gebunden werden soll, nach der Uebersetzung von Hand in eine CPROLOG-Struktur umwandeln.

Beispiel: Gegeben sei folgende Datenbasis:

```
((erweitern (? x)) (konklusion (? x) (? xl))
                   (praemisse (? x) (? yl))
                   (fakt (? x) (? zl))
                   (ass (? zl))
                   (ass (? xl) (? yl) (member (? a) (? l))))

 (fakt familie (vater joe (jim jack)))

 (fakt schule (lehrer mueller (hans helmut herbert)))

 (konklusion familie (sohn (? a) (? y)))

 (konklusion schule (schueler (? a) (? y)))

 (praemisse familie (vater (? y) (? l)))

 (praemisse schule (lehrer (? y) (? l))))
```

<u>Uebersetzung</u>:

```
erweitern(_x) :-
        konklusion(_x,_xl),
        praemisse(_x,_yl),
```

```
            fakt(_x,_zl),
            assertz(_zl),
            assertz((_xl :-
                    '=..-if-list'(_yl,_x001), _x001,
                    member(_a,_l))).

fakt(familie,[vater,joe,[jim,jack]]).

fakt(schule,[lehrer,mueller,[hans,heiner,herbert]]).

konklusion(familie,[sohn,_a,_l]).

konklusion(schule,[schueler,_a,_l]).

praemisse(familie,[vater,_y,_l]).

praemisse(schule,[lehrer,_y,_l]).
```

<u>Anderung von Hand</u>:

```
fakt(familie,vater(joe,[jim,jack])).

fakt(schule,lehrer(mueller,[hans,heiner,herbert])).

konklusion(familie,sohn(_a,_l)).

konklusion(schule,schueler(_a,_l)).
```

Alle anderen Klauseln bleiben unveraendert.

3. Benutzeroberflaeche

Normalerweise wird der Uebersetzer durch die Funktion translc in der Form "(translc <lisplogdatei> <cprologdatei>)" aufgerufen, wobei <lisplogdatei> die Quelldatei ist, auf der die zu uebersetzende LISPLOG-Datenbasis steht; <cprologdatei> ist die Zieldatei, auf die die herauskommende CPROLOG-Datenbasis geschrieben wird.

Der Uebersetzer kann auch durch die Funktion transcprolog aufgerufen werden. Um die Datenbasis in die interne Darstellung zu bringen, steht nach ihrem Erstellen oder Laden aus einer Quelldatei die Funktion get-db zur Verfuegung, so dass ein geschachtelter Aufruf der Form "(transcprolog (get-db) <cprologdatei>)" verwendet werden kann, wobei <cprologdatei> der Name der Zieldatei ist, auf die die uebersetzte CPROLOG-Datenbasis geschrieben wird.

(translc 's_lisplogdatei 's_cprologdatei)

WERT: t, falls die Uebersetzung durchgefuehrt wird; nil sonst.

SEITENEFFEKT: Uebersetzt die in der Datei s_lisplogdatei stehende LISPLOG-Datenbasis in CPROLOG-Syntax und schreibt sie auf die Datei s_cprologdatei.

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

(transcprolog 'l_dbase 's_cprologdatei)

WERT: t

SEITENEFFEKT: Uebersetzt die LISPLOG-Datenbasis l_dbase in CPROLOG-Syntax und schreibt sie auf die Datei s_cprologdatei.

BEACHTE: Die LISPLOG-Datenbasis hat folgende get-db-Form:

```
((<konklusion> [<praemisse> ...])
 (<konklusion> [<praemisse> ...])
  .
  .
  . )
```

(get-db)

WERT: LISPLOG-Datenbasis in der fuer die Eingabe in transcprolog benoetigten Form.

BEACHTE: Die Funktion wird nach dem interaktiven Erstellen oder Laden einer Datenbasis aufgerufen.

4. Implementierung

Das Programm arbeitet in zwei Stufen. Als erstes wird das LISPLOG-Programm wie oben beschrieben uebersetzt. In der zweiten Stufe werden zu den im LISPLOG-Programm aufgetretenen LISP-Funktionsaufrufen, die ja in Relationsaufrufe umgewandelt wurden, die entsprechenden CPROLOG-Prozeduren erzeugt. Dazu wird das in [Hinkelmann & Morgenstern 1985] beschriebene Programm aufgerufen.

4.1 Hauptfunktionen

(translc 's_lisplogdatei 's_cprologdatei)

hat als Argumente die Quelldatei s_lisplogdatei, auf der die zu uebersetzende LISPLOG-Datenbasis steht, und die Zieldatei s_cprologdatei, auf die die herauskommende CPROLOG-Datenbasis geschrieben wird.

Die Datenbasis wird aus der Datei s_lisplogdatei mit Hilfe der Funktion consult geladen, durch die Funktion get-db in die interne Form gebracht und durch Aufruf von transcprolog uebersetzt.

(transcprolog '1_dbase 's_file)

hat als Argumente die gesamte listifizierte LISPLOG-Datenbasis 1_dbase sowie einen Dateinamen s_file. Die Funktion erzeugt fuer s_file einen Port, der der Variablen prt zugewiesen wird. Durch transclause werden die Klauseln einzeln uebersetzt und ueber den Port prt ausgedruckt. Anschliessend werden durch transglob fuer die in der Datenbasis vorkommenden LISP-Funktionen CPROLOG-Prozeduren erzeugt (siehe dazu 4.4). Der Port prt ist fuer transclause und transglob global.

Falls eine Variable als Praemisse auftrat, ist goal-variable gleich 't', und es werden die in 2.2.2 angegebenen Klauseln zur Erzeugung von Strukturen an das Programm angefuegt.

Abschliessend wird der Port geschlossen.

(transclause '1_clause)

hat als Argument eine Klausel in der internen Notation. Als Seiteneffekt druckt sie die Klausel in CPROLOG-Syntax in einer Art pretty-print Schreibweise ueber den globalen Port prt aus. Ist das Argument ein Fakt, so wird das einzige Element (die Konklusion) durch transconclusion uebersetzt. Ist das Argument eine Regel, so wird aus den Praemissen eine DNF erzeugt (siehe 2.3.5). Jedes Disjunkt wird zusammen mit

der durch transruleconclusion uebersetzen Konklusion als eine Klausel uebersetzt.

(transconclusion 'l_structure)
hat als Wert die Uebersetzung der LISPLOG-Konklusion l_structure als FRANZ LISP String gemaess 2.1.1.

(transruleconclusion 'l_structure)
hat als Wert die von transconclusion erzeugte Uebersetzung der Konklusion l_structure. Daran wird ein ':-" konkateniert, das die Konklusion von den Praemissen trennen soll, bzw. ein Komma, falls das ':-' schon durch transconclusion erzeugt wurde (vgl. 2.1.1).

(transpremiselist 'l_premiselist)
ruft fuer jedes Element in l_premiselist die Funktion transpremise auf, die die Uebersetzung der Praemissen ueber den Port prt ausgibt.

(transpremise 'sl_term)
druckt als Seiteneffekt die Uebersetzung der LISPLOG-Konklusion sl_term ueber den globalen Port prt aus. Die Uebersetzung erfolgt in der in 2.2 angegebenen Weise. Ist sl_term ein Aufruf des Primitivs is, so wird er durch opt-flattenfunc (vgl. [Hinkelmann & Morgenstern 1985]) in eine Liste von Relationsaufrufen abgeflacht, die in einem 2. Schritt von lispfunctions weiterverarbeitet werden. Entsprechendes gilt fuer Aufrufe von LISP-Praedikaten, die allerdings durch opt-flattenrel abgeflacht werden. (Ausnahmen: progn, Eingabe-Funktionen, ass, rex, atom, litatom, number).

(lispfunctions 'l_premiselist)
hat als Argument eine Liste von Relationsaufrufen, die bei der Abflachung von LISP-Praedikatsaufrufen und Aufrufen des Primitivs is entstand. Die Elemente werden nacheinander zur Uebersetzung an lispfct uebergeben (Schritt 2 aus 2.2.3.2 und 2.2.3.3 a)).

(lispfct 'l_structure)
gibt als Seiteneffekt die Uebersetzung des Relationsaufrufs l_structure ueber den Port prt aus.

4.2 Hilfsfunktionen

(component 'l_complist)
hat als Argument die Liste der Argumente einer Struktur, die der Reihe nach an terms uebergeben werden.

(terms 'sl_object)
hat als Wert die CPROLOG-Syntax des LISPLOG-Terms sl_object als FRANZ LISP String. Als sl_object koennen nur Variable, Datenlisten und Konstanten auftreten.

(arith-expr 'g_expression)
uebersetzt einen arithmetischen LISP-Ausdruck. Die Argumente eines Ausdrucks koennen wiederum arithmetische LISP-Ausdruecke sein, weshalb arith-expr sich rekursiv aufruft.

(prologlist 'l_lisplist)
wandelt die LISPLOG-Liste l_lisplist in eine CPROLOG-Liste um. Prologlist wird von terms aufgerufen, wobei '[' schon in terms erzeugt wird. Die Elemente von l_lisplist sind wieder Terme, so dass prologlist zur Verarbeitung dieser Elemente wieder terms aufruft.

(variab 'l_variable)
bekommt als Argument eine LISPLOG-Variable der Form (? name). Ihr Wert ist der Name der Variable, der an einen Unterstrich konkateniert ist.
Beispiel: (? var) --> _var

(arith-comp 'l_list)
wandelt ein arithmetisches LISP-Praedikat in den entsprechenden Infix-Operator in CPROLOG um:
Beispiel: (greaterp 4 2) --> 4 > 2

(is-construct 'l_list)
erzeugt aus einem arithmetischen LISPLOG-Primitiv ein is-Konstrukt in CPROLOG.

(prologout 'l_out)
liefert als Wert die Uebersetzung einer abgeflachten LISP-Ausgabefunktion (vgl. 2.3.2).

(prologin 'l_structure)
liefert als Wert die Uebersetzung eines Aufrufs einer LISP-Eingabefunktion gemaess 2.3.3.

(progfct 'l_progargs)
gibt als Seiteneffekt die Uebersetzung der Argumente l_progargs eines Aufrufs der Funktion progn aus. Die ersten n-1 Argumente werden von transprogpremise, das letzte Argument wird von transpremise uebersetzt (vgl. 2.3.1).

(transprogpremise 'l_structure)

uebersetzt l_structure, das als Argument von progn auftrat, nach CPROLOG. Atome, ausser nil, werden nicht uebersetzt, da sie immer zu success fuehren. LISP-Funktionen werden durch opt-flattenprog abgeflacht und durch lispfunctions uebersetzt.

(assrex 'l_structure)

uebersetzt einen Aufruf der LISP-Funktionen ass und rex wie in 2.3.6 beschrieben. Die entsprechenden CPROLOG-Relationsnamen stehen in der Assoziationsliste assrex-alist. Ist das Argument ein Fakt, d.h. l_structure hat die Laenge 2, dann wird die Uebersetzung ausgedruckt.
Ist das Argument eine Klausel, so wird zuerst aus den Praemissen eine DNF erzeugt (vgl. transclause). Mit der CPROLOG-Entsprechung des Funktionsnamens, der uebersetzten Konklusion und dieser DNF als Argumenten wird assrex-rulelist aufgerufen.
Eine Variable als Konklusion ist hier erlaubt.

(assrex-rulelist 's_name 's_c-conclusion 'l_premiselists)

erzeugt fuer jedes Element von l_premiselists zusammen mit der CPROLOG-Konklusion s_c-conclusion eine CPROLOG-Klausel, die als Argument des CPROLOG-Relationsaufrufs s_name (assertz oder retract) ausgegeben wird. Dazu ruft sie fuer jedes Element von l_premiselists zusammen mit s_name und s_c-conclusion die Funktion assrex-rule auf.

(assrex-rule 's_name s_c-conclusion 'l_premiselist)

uebersetzt die LISPLOG-Praemissen in l_premiselist nach CPROLOG und generiert daraus zusammen mit s_c-conclusion eine CPROLOG-Klausel als Argument des Relationsaufrufs s_name.

4.3 Variablen

goal-variable
: globale Variable, die zu 't' gesetzt wird, falls im zu uebersetzenden Programm eine Variable als Goal auftrat.

tabulator
: hat als Wert eine natuerliche Zahl, die angibt, um wieviele Stellen die Ausgabe eingerueckt wird, um eine uebersichtliche Form des CPROLOG-Programms zu erhalten.

compares
: Liste der LISP-Praedikate zum Vergleichen numerischer Werte.

compares-alist
: Assoziationsliste mit den LISP-Praedikaten aus compares (s.o.) als Schluessel und den entsprechenden Infix-Operatoren aus CPROLOG als Werten.
Beispiel: (greaterp " > ")

infix-ops
: Assoziationsliste mit arithmetischen LISP-Funktionen als Schluessel und den entsprechenden CPROLOG-Operatoren als Werten.
Beispiel: (plus " + ")

newli
: Liste mit den Elementen 'terpr' und 'terpri', die in CPROLOG mit 'nl' uebersetzt werden.

newli-prog
: Liste mit den Elementen 'terpr$p' und 'terpri$p'. Diese Liste ist notwendig, da 'terpr' und 'terpri' nie mit 'nl,fail' uebersetzt werden duerfen, wenn sie innerhalb eines progn auftreten.

unary-ops
: Assoziationsliste mit einstelligen LISP-Funktionen als Schluessel und den CPROLOG-Entsprechungen als Werten.
Beispiel: (add1 " + 1")

outfunctions
: Liste der Ausgabefunktionen von LISP, die in CPROLOG mit 'write' uebersetzt werden.

outfunctions-prog
: Liste der Ausgabefunktionen von LISP, an die ein '$p' konkateniert wurde. Die Liste wird benoetigt, um Ausgabefunktionen, die innerhalb von progn vorkamen, von anderen zu unterscheiden.

infunctions
Liste der LISP-Eingabefunktionen.

prefix-ops
Liste von arithmetischen LISP-Funktionen, die in CPROLOG den gleichen Namen haben.
Beispiele: sin, cos, sqrt

primitiv-alist
Assoziationsliste mit LISPLOG-Primitiven als Schluessel und CPROLOG-Operatoren als Werten.
Beispiel: (addit " + ")

meta-logical
Assoziationliste mit den LISP-Funktionen 'numberp', 'atom' und 'litatom' sowie den meta-logischen Primitiven 'var' und 'nonvar' als Schluesseln und deren CPROLOG-Entsprechungen 'number', 'atomic', 'atom', 'var' und 'nonvar' als Werten.
(Hinweis: Diese Assoziationsliste kann erweitert werden, falls ein Anwender weitere LISP-Funktionen benoetigt, deren CPROLOG-Entsprechungen andere Namen haben.)

Zur Bedeutung der Variablen mind-func, mind-rel, still-func, still-rel und relation siehe [Hinkelmann & Morgenstern 1985].

4.4 Transformation von LISP nach PROLOG

Das Programm zur Transformation von LISP-Funktionen in PROLOG-Relationen aus [Hinkelmann & Morgenstern 1985] wird in transcprolog aufgerufen. Es musste fuer diese Anwendung allerdings in einigen Punkten angepasst werden.

- Die Zielsprache des Transformationsprogramms ist LISPLOG. Da hier CPROLOG die Zielsprache ist, wird das Ergebnis des Transformationsprogramms vor der Ausgabe nach CPROLOG uebersetzt. Dazu wird in der Funktion ausgabe fuer jede erzeugte LISPLOG-Klausel die Funktion printclause aufgerufen.

  (printclause 'l_clause)
  hat als Argument eine durch das Transformationspogramm erzeugte LISPLOG-Klausel. Die Konklusion wird durch transconclusion uebersetzt und ueber den Port prt ausgegeben, waehrend die Praemissen von der Funktion premiselist bearbeitet werden (vgl. transclause).

  (printpremiselist 'l_premiselist)
  ruft fuer jedes Element aus l_premiselist die Funktion printpremise auf, die die Uebersetzung der Praemissen ueber den Port prt ausgibt.

(printpremise 'sl_term)
druckt als Seiteneffekt die Uebersetzung der LISPLOG-Konklusion sl_term ueber den globalen Port prt aus (vgl. die Funktionen transpremise und lispfct). Es werden nur diejenigen Arten von Termen uebersetzt, die durch das Transformationsprogramm erzeugt werden koennen.

- In den Funktionen transfunction und transrelation wurden fuer einige LISP-Systemfunktionen die entsprechenden CPROLOG-Prozeduren angegeben.

- Eine Aenderung der Funktionsnamen durch Anhaengen von *r bzw. *p ist nicht notwendig, da keine Verwechslung mit den entsprechenden LISP-Funktionen moeglich ist. Wird eine LISP-Funktion einmal als allgemeine Funktion und einmal als Praedikat uebersetzt, ist eine Verwechslung durch die unterschiedliche Argumentzahl ausgeschlossen. An die Namen von prog-Funktionen wird allerdings ein '$p' konkateniert, um eine Verwechslung mit LISP-Praedikaten zu vermeiden.

- Zur Erzeugung disjunktiver Normalformen aus Klauselruempfen (vgl 2.3.5) werden die Funktionen cond-treat und is-treat hinzugefuegt.

(cond-treat 'l_neg 'l_condcascade)
hat als Argumente die Liste der cond-Klauseln condcascade sowie die Liste l_neg der negierten Bedingungen vorheriger Klauseln. Diese werden benoetigt, da cond-treat cdr-rekursiv ist. Das Resultat ist die DNF der Klauseln l_condcascade in der in [Hinkelmann & Morgenstern 1985] beschriebenen Notation.

(is-treat 'l_prefix 'l_condcascade 'l_neg)
hat als Argumente eine Liste von cond-Klauseln l_condcascade, die Liste l_neg der negierten Bedingungen vorheriger Klauseln sowie die die Liste l_prefix mit den beiden ersten Elementen des Relationsaufrufs, also das Symbol is und eine Variable. Das Resultat ist die DNF des Relationsaufrufs von is.

- Um eine Funktion mit implizitem bzw. explizitem progn zu uebersetzen, wurde vor allem die Funktion transbody geaendert.

(transbody 'l_action)
WERT: Liste der disjunktiven Normalformen der Funktionsaufrufe in der Liste l_action.

Die Funktion ist cdr-rekursiv, wobei die ersten n-1 Funktionsaufrufe als Aufrufe von prog-Funktionen angesehen werden. Der letzte Funktionsaufruf wird je nach dem Wert der globalen Variable relation als allgemeine Funktion oder Praedikat behandelt. Diese disjunktiven Normalformen muessen spaeter (in der aufrufenden Funktion) durch die

Funktion and-connect miteinander verknuepft werden, so dass man dann die disjunktive Normalform von l_action erhaelt.

- Zur Uebersetzung der prog-Funktionsdefinitionen wurden neue Funktionen eingefuehrt:

  (transbody-prog 'l_action)
  WERT: Liste der disjunktiven Normalformen der Funktionsaufrufe in der Liste l_action.

  Die Funktion arbeitet analog zu transbody. Sie wird nur fuer Funktionsruempfe von prog-Funktionen und fuer die Aktionen in cond-Klauseln aufgerufen, die als eines der n-1 ersten Argumente eines impliziten oder expliziten progn vorkommen.

  (transcond-prog 'l_neglist 'l_condcascade)
  WERT: Disjunktive Normalform eines Aufrufs der Funktions cond mit abgeflachten Funktionsaufrufen.

  Die Funktion ist gegenueber der Funktion transcond nur dahingehend geaendert, dass statt trclause nun trclause-prog aufgerufen wird. Dies ist notwendig da die zu uebersetzende cond-Funktion als Argument eines impliziten oder expliziten progn oder als Rumpf einer prog-Funktion aufgetreten war.

  (trclause-prog 'l_neglist 'l_condclause)
  WERT: Disjunktive Normalform der cond-Klausel l_condclause.

  Analog zur Funktion trclause. Einziger Unterschied ist, dass statt der Funktion transbody die Funktion transbody-prog aufgerufen wird.

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

  Die Funktion transprog arbeitet analog zu transfunction und transrelation. Zur Uebersetzung des Funktionsrumpfes wird allerdings nicht transbody, sondern transbody-prog aufgerufen. Ausserdem wird durch Anhaengen von '$p' an den Namen kenntlich gemacht, dass es sich um die Uebersetzung einer prog-Funktion handelt.

  (transaction-prog 'sl_term)
  WERT: Liste mit der abgeflachten Form von sl_term.

  Ist sl_term ein Atom, so muss auch bei einem nonnil-Wert das herauskommende PROLOG-Praedikat erfuellt werden. Es braucht also nicht uebersetzt zu werden. Ist sl_term ein Funktionsaufruf, so wird er durch opt-flattenprog abgeflacht.

(opt-flattenprog 'l_list)
WERT: optimierte abgeflachte Liste von l_list.

Der Funktionsaufruf l_list wird mit Hilfe der Funktion flattenprog abgeflacht. Auf das Ergebnis wird die Optimierungsfunktion opt angewendet.

(flattenprog 'l_list)
WERT: abgeflachte Form von l_list.

Der Funktionsname wird in den globalen Variablen mind-prog und still-prog gespeichert. Die Funktion ist sonst analog zu flattenrel.

- Es wurden auch neue globale Variable eingefuehrt naemlich mind-prog und still-prog, die den Variablen mind-rel und still-rel entsprechen.

5. Transformation von LISP-Funktionen und LISPLOG-Datenbasen nach CPROLOG mit allgemeinem Cut-Operator

Das Programm zur Transformation von LISP-Funktionsdefinitionen in PROLOG-Relationen aus [Hinkelmann & Morgenstern 1985] uebersetzt Aufrufe der Funktion cond sehr ineffizient. Das haengt damit zusammen, dass in der Zielsprache LISPLOG kein allgemeiner Cut-Operator implementiert ist. Durch die Uebersetzung der LISP-Funktionen nach CPROLOG, wo ein allgemeiner Cut-Operator existiert, kann das herauskommende Programm viel effizienter sein. Natuerlich koennen auch Aufrufe der Funktion cond als Praemissen eines LISPLOG-Programms durch den Cut-Operator effizienter uebersetzt werden.

In [Hinkelmann & Morgenstern 1985] ist die Uebersetzung eines Aufrufs der Funktion cond beschrieben. Zur Vermeidung des Cut nach jeder Bedingung einer cond-Klausel, werden die Bedingungen der vorher aufgetretenen Klauseln negiert und in disjunktive Normalform gebracht. Wenn der Cut nun nicht mehr vermieden werden muss, kann dieses aufwendige Verfahren entfallen.

Der Cut wird intern realisiert durch das nullstellige Praedikat '(!)'. Bei der Uebersetzung von nullstelligen Praedikaten nach CPROLOG wird der Praedikatsname einfach uebernommen, so dass fuer die Uebersetzung der entstehenden Klausel in printclause bzw. transclause keine neue Fallunterscheidung notwendig wird.

Die Realisierung geschieht folgendermassen:

- Die Funktionen, die die Negation der Bedingungen der cond-Klauseln berechneten, werden weggelassen. Dadurch werden auch die Funktionen transcond und trclause sehr viel einfacher.

- Nachdem die Bedingungen der cond-Klauseln in disjunktive Normalform umgewandelt und abgeflacht sind, wird zu jedem Disjunkt das Praedikat '(!)' hinzugefuegt. Dies geschieht in den Funktionen transcondition (bei Aufrufen der Funktion cond in LISP-Funktionsdefintionen) und cond-treat (bei Aufrufen der Funktion cond als Praemisse eines LISPLOG-Programms).

LITERATUR

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

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

[Dahmen 1985] M. Dahmen: Ein Translator von CPROLOG nach LISPLOG. In: [Dahmen, Herr, Hinkelmann, Morgenstern 1985].

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

[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

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

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

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

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

[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

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

ıhang A: Listing

'unktionsnamen und Variablennamen:

```
setq translate.progfns
      '(translc transcprolog
                get-db
                db-predlst
                transclause
                transconclusion
                transruleconclusion
                transpremiselist
                transpremise
                lispfunctions
                lispfct
                component
                terms
                arith-expr
                prologlist
                variab
                comp
                is-construct
                prologout
                prologin
                progfct
                transprogpremise
                assrex
                assrex-rule
                assrex-rulelist
                compares
                compares-alist
                infix-ops
                newli
                newli-prog
                unary-ops
                outfunctions
                outfunctions-prog
                infunctions
                prefix-ops
                primitiv-alist
                meta-logical
                assrex-alist))
```



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


(def translc
  (lambda (lisplogdatei cprologdatei)
    (cond (predicates (princ
                        "Es ist schon eine Datenbasis geladen!")
                       (terpri)
                       (princ
                        (uconcat
                         "Falls diese mit uebersetzt werden soll,"
                         " bitte die Quelldatei von Hand laden"))
                       (terpri)
                       (princ
                        (uconcat
                         "und den Uebersetzer aufrufen mit "
                         "'(transcprolog (get-db) <cprologdatei>)'."))
                       (terpri)
                       (princ
                        "Ansonsten bitte die Datenbasis loeschen!")
                       (terpri))
          (t (apply (function consult) (list lisplogdatei))
             (transcprolog (get-db) cprologdatei)))))

(def transcprolog
  (lambda (base file)
    (let ((prt (outfile file))
          (still-prog nil)
          (still-func '(quote))
          (still-rel '(null))
          (mind-func '(quote read ratom readc))
          (mind-rel
           (append compares
                   '(null print
                          princ
                          patom
                          pp
                          tab
                          terpr
                          terpri
                          numberp
                          atom
                          litatom)))
          (mind-prog '(print princ patom pp tab terpr terpri))
          (tabulator 10)
          (relation t)
          (goal-variable nil))
         (terpri)
         (princ "LISPLOG-CPROLOG-Uebersetzung laeuft!")
         (terpri)
         (mapcar (function transclause) base)
```

```
        (princ "LISP-CPROLOG-Umwandlung laeuft!")
        (terpri)
        (setq mind-func
              (append '(car cdr cons) mind-func))
        (transglob)
        (cond
         (goal-variable
          (princ '"'=..-if-list'([X|Y],Z) :- Z =.. [X|Y]." prt)
          (terpri prt)
          (terpri prt)
          (princ '"'=..-if-list'(X,X)." prt)
          (terpri prt)))
        (close prt))))

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

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

(def transclause
  (lambda (rule)
    (cond ((null (rest rule))
           (princ (uconcat (transconclusion (first rule)) ".")
                  prt)
           (terpri prt)
           (terpri prt)
           t)
          (t (mapcar (function
                      (lambda (x)
                              (princ (transruleconclusion
                                      (first rule))
                                     prt)
                              (transpremiselist x)
                              (princ '"." prt)
                              (terpri prt)
                              (terpri prt))
                      )
                     (and-connect
                      (mapcar (function transform-dnf)
                              (rest rule))))
             t))))

(def transconclusion
  (lambda (structure)
    (cond ((null (rest structure)) (first structure))
          ((eq (first structure) 'cut)
           (uconcat (transconclusion (second structure))
```



```
                    '":- !"))
          (t
           (uconcat (first structure)
                    '"("
                    (component (rest structure))
                    '")")))))

(def transruleconclusion
  (lambda (ll-conclusion)
    (cond ((eq (first ll-conclusion) 'cut)
           (uconcat (transconclusion ll-conclusion) '","))
          (t
           (uconcat (transconclusion ll-conclusion)
                    '" :-")))))

(def transpremiselist
  (lambda (premiselist)
    (cond ((null (rest premiselist))
           (tab tabulator prt)
           (transpremise (first premiselist)))
          (t (tab tabulator prt)
             (transpremise (first premiselist))
             (princ '"," prt)
             (transpremiselist (rest premiselist))))))

(def transpremise
  (lambda (structure)
    (cond ((null structure) (princ 'fail prt))
          ((atom structure) (princ 'true prt))
          ((memq (first structure) predicates)
           (cond ((null (rest structure))
                  (princ (first structure) prt))
                 (t
                  (princ (uconcat (first structure)
                                  '"("
                                  (component
                                   (rest structure))
                                  '")")
                         prt))))
          ((eq (first structure) 'is)
           (lispfunctions (opt-flattenrel structure)))
          ((assq (first structure) primitiv-alist)
           (princ (is-construct structure) prt))
          ((assq (first structure) meta-logical)
           (princ (uconcat (second
                            (assq (first structure)
                                  meta-logical))
                           '"("
                           (terms (second structure))
                           '")")
                  prt))
          ((eq (first structure) 'not)
           (cond ((and (not
```

```
                    (member (first (second structure))
                            predicates))
                   (getd (first (second structure))))
              (lispfunctions (opt-flattenrel structure)))
             (t (princ '"not(" prt)
                (transpremise (second structure))
                (princ '")" prt))))
       ((eq (first structure) 'progn)
        (progfct (rest structure)))
       ((memq (first structure) '(ass rex))
        (setq tabulator (add tabulator 9))
        (assrex structure)
        (setq tabulator (diff tabulator 9)))
       ((memq (first structure) '(read ratom))
        ((lambda (x)
               (princ (uconcat '"read("
                               x
                               '"),"
                               x)
                      prt))
         (gensym 'R)))
       ((variable-p structure)
        (setq goal-variable t)
        ((lambda (x)
               (princ (uconcat '"'=..-if-list'("
                               (variab structure)
                               '","
                               x
                               '"),"
                               x)
                      prt))
         (gensym 'X)))
       ((getd (first structure))
        (lispfunctions (opt-flattenrel structure)))
       ((null (rest structure))
        (princ (first structure) prt))
       (t
        (princ (uconcat (first structure)
                        '"("
                        (component (rest structure))
                        '")")
            prt)))))

def lispfunctions
 (lambda (structlist)
   (cond ((null (rest structlist))
          (lispfct (first structlist)))
         (t (lispfct (first structlist))
            (princ '"," prt)
            (tab tabulator prt)
            (lispfunctions (rest structlist))))))
```



```
(def lispfct
  (lambda (structure)
    (cond ((eq (first structure) 'is)
           (princ (uconcat (terms (second structure))
                           '" is "
                           (arith-expr (third structure)))
                  prt))
          ((eq (first structure) 'not)
           (cond ((memq (first (second structure)) newli)
                  (princ 'nl prt))
                 ((eq (first (second structure)) 'print)
                  (princ (prologout
                          (second (second structure)))
                         prt))
                 (t (princ '"not(" prt)
                    (lispfct (second structure))
                    (princ '")" prt))))
          ((eq (first structure) 'print)
           (princ (uconcat (prologout (second structure))
                           '", fail")
                  prt))
          ((eq (first structure) 'patom)
           (princ (uconcat (prologout (second structure))
                           '",")
                  prt)
           (tab tabulator prt)
           (princ (uconcat '"not(null("
                           (terms (second structure))
                           '"))")
                  prt))
          ((memq (first structure) outfunctions)
           (princ (prologout (second structure)) prt))
          ((memq (first structure) infunctions)
           (princ (prologin structure) prt))
          ((memq (first structure) newli)
           (princ '"nl, fail" prt))
          ((memq (first structure) compares)
           (princ (comp structure) prt))
          ((eq (first structure) 'unify)
           (princ (uconcat (terms (second structure))
                           '" = "
                           (terms (third structure)))
                  prt))
          ((memq (first structure) outfunctions-prog)
           (princ (prologout (second structure)) prt))
          ((memq (first structure) newli-prog) (princ 'nl prt))
          ((null (rest structure)) (princ (first structure) prt))
          (t
           (princ (uconcat (first structure)
                           '"("
                           (component (rest structure))
                           '")")
                  prt)))))
```

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


(def component
  (lambda (complist)
    (cond ((null (rest complist)) (terms (first complist)))
          ((variable-p (rest complist))
           (princ
            '"Variable Argumentzahl in CPROLOG nicht erlaubt!")
           (terpri)
           (princ
            (uconcat '"'"
                     (first structure)
                     '"' bitte von Hand anpassen!"))
           (terpri)
           (terpri)
           (uconcat (terms (first complist))
                    '","
                    (variab (rest complist))))
          ((eq (rest complist) 'ID)
           (princ
            '"Variable Argumentzahl in CPROLOG nicht erlaubt!")
           (terpri)
           (princ
            (uconcat '"'"
                     (first structure)
                     '"' bitte von Hand anpassen!"))
           (terpri)
           (terpri)
           (uconcat (terms (first complist)) '",_"))
          (t
           (uconcat (terms (first complist))
                    '","
                    (component (rest complist)))))))


(def terms
  (lambda (object)
    (cond ((variable-p object) (variab object))
          ((eq object 'ID) '"_")
          ((listp object)
           (uconcat '"[" (prologlist object)))
          ((stringp object)
           (uconcat '"'" object '"'"))
          (t object))))


(def arith-expr
  (lambda (expression)
    (cond ((numberp expression) expression)
          ((variable-p expression) (variab expression))
```



```
          ((eq (first expression) 'minus)
           (uconcat '"-" (arith-expr (second expression))))
          ((memq (first expression) prefix-ops)
           (uconcat (first expression)
                    '"("
                    (arith-expr (second expression))
                    '")"))
          ((assq (first expression) unary-ops)
           (uconcat '"("
                    (arith-expr (second expression))
                    (second
                     (assq (first expression) unary-ops))
                    '")"))
          ((assq (first expression) infix-ops)
           (uconcat '"("
                    (arith-expr (second expression))
                    (second
                     (assq (first expression) infix-ops))
                    (arith-expr (third expression))
                    '")"))
          (t (princ "arithmetischer Ausdruck erwartet!!!!!")
             (terpri)
             "********arithmetischer Ausdruck?!?!?!?!?!********"))))

(def prologlist
  (lambda (lisplist)
    (cond ((null lisplist) '"]")
          ((variable-p (rest lisplist))
           (uconcat (terms (first lisplist))
                    '"|"
                    (variab (rest lisplist))
                    '"]"))
          ((eq (rest lisplist) 'ID)
           (uconcat (terms (first lisplist)) '"|_]"))
          ((null (rest lisplist))
           (uconcat (terms (first lisplist)) '"]"))
          ((listp (rest lisplist))
           (uconcat (terms (first lisplist))
                    '","
                    (prologlist (rest lisplist))))
          (t
           (uconcat (terms (first lisplist))
                    '"|"
                    (terms (rest lisplist))
                    '"]")))))

(def variab
  (lambda (x)
    (uconcat '"_" (second x))))

(def comp
  (lambda (list)
    (uconcat (terms (second list))
```

```
           (second (assq (first list) compares-alist))
           (terms (third list)))))

lef is-construct
 (lambda (list)
   (uconcat (variab (fourth list))
           " is "
           (terms (second list))
           (second (assq (first list) primitiv-alist))
           (terms (third list)))))

lef prologout
 (lambda (out)
   (cond ((variable-p out)
          (uconcat '"write(" (variab out) '")"))
         ((listp out)
          (uconcat '"write([" (prologlist out) '")"))
         (t (uconcat '"write('" out '"')")))))

lef prologin
 (lambda (structure)
   (cond ((memq (first structure) '(read ratom))
          (uconcat '"read("
                   (terms (second structure))
                   '")"))
         ((eq (first structure) 'readc)
          ((lambda (x)
                   (princ (uconcat '"get0("
                                   x
                                   '"),")
                          prt)
                   (uconcat '"name("
                            (terms (second structure))
                            '",["
                            x
                            '"])"))
          (gensym 'Z))))))

lef progfct
 (lambda (rule)
   (cond ((null (rest rule)) (transpremise (first rule)))
         (t (transprogpremise (first rule))
            (princ '"," prt)
            (tab tabulator prt)
            (progfct (rest rule))))))

lef transprogpremise
 (lambda (structure)
   (cond ((atom structure)
          (cond (structure)
                (t (princ "nil als Argument von progn!"))))
         (t (lispfunctions (opt-flattenprog structure))))))
```



```
(def assrex
  (lambda (structure)
    (cond ((variable-p (second structure))
           (cond ((eq (length structure) 2)
                  (princ (uconcat (second
                                   (assq (first structure)
                                         assrex-alist))
                                  '"("
                                  (variab (second structure))
                                  '")")
                         prt))
                 (t
                  (assrex-rulelist (second
                                    (assq (first structure)
                                          assrex-alist))
                                   (uconcat (variab
                                             (second
                                              structure))
                                            '" :-")
                                   (and-connect
                                    (mapcar (function
                                             transform-dnf)
                                            (rest
                                             (rest
                                              structure)))))))))
          ((eq (length structure) 2)
           (princ (uconcat (second
                            (assq (first structure)
                                  assrex-alist))
                           '"(("
                           (transconclusion (second structure))
                           '"))")
                  prt))
          (t
           (assrex-rulelist (second
                             (assq (first structure)
                                   assrex-alist))
                            (transruleconclusion
                             (second structure))
                            (and-connect
                             (mapcar (function
                                      transform-dnf)
                                     (rest
                                      (rest structure)))))))))

(def assrex-rule
  (lambda (name c-conclusion premiselist)
    (princ (uconcat name '"((" c-conclusion) prt)
    (transpremiselist premiselist)
    (princ '"))" prt)))


(def assrex-rulelist
```

```
  (lambda (name c-conclusion premiselists)
    (cond ((null (rest premiselists))
           (assrex-rule name c-conclusion (first premiselists)))
          (t (assrex-rule name
                          c-conclusion
                          (first premiselists))
             (princ '"," prt)
             (tab (diff tabulator 9) prt)
             (assrex-rulelist name
                              c-conclusion
                              (rest premiselists)))))
```

```
;Variablen:
;----------

(setq compares '(eq equal greaterp > >& lessp < <& = =&))

(setq compares-alist
      '((eq " =:= ")
        (equal " =:= ")
        (greaterp " > ")
        (> " > ")
        (>& " > ")
        (lessp " < ")
        (< " < ")
        (<& " < ")
        (= " =:= ")
        (=& " =:= ")))

(setq infix-ops
      '((add " + ")
        (+ " + ")
        (plus " + ")
        (sum " + ")
        (diff " - ")
        (difference " - ")
        (- " - ")
        (times " * ")
        (product " * ")
        (* " * ")
        (divide " / ")
        (quotient " / ")
```



```
        (/ " / ")
        (mod " mod ")
        (remainder " mod ")
        (expt " ^ ")))


(setq newli '(terpr terpri))

(setq newli-prog '(terpr$p terpri$p))

(setq unary-ops
      '((add1 " + 1") (|1+| " + 1") (sub1 " - 1") (|1-| " - 1")))

(setq outfunctions '(patom print princ pp))

(setq outfunctions-prog '(patom$p print$p princ$p pp$p))

(setq infunctions '(ratom read readc))

(setq prefix-ops
      '(acos asin atan cos exp log sin sqrt tan))

(setq primitiv-alist
      '((addit " + ") (subit " - ") (multit " * ") (divit " / ")))

(setq meta-logical
      '((numberp number)
        (atom atomic)
        (litatom atom)
        (var var)
        (nonvar nonvar)))

(setq assrex-alist '((ass assertz) (rex retract)))
```

Anhang B: Listing der LISP-PROLOG-Transformation

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

(setq transform.progfns
      '(translate transglob
                  transfunction
                  transrelation
                  transprog
                  transbody
                  transcond
                  trclause
                  transbody-prog
                  transcond-prog
                  trclause-prog
                  transcondition
                  transaction
                  transaction-r
                  transaction-prog
                  ausgabe
                  printclause
                  printpremiselist
                  printpremise))

(def translate
  (lambda (func rel prog file)
    (let ((prt (outfile file))
          (still-func func)
          (still-rel (cons 'null rel))
          (still-prog prog)
          (mind-func (append '(car cdr cons) func))
          (mind-rel (append '(null equal eq) rel))
          (mind-prog
           (append '(print princ patom tab terpr terpri)
                   prog)))
         (transglob))))

(def transglob
  (lambda nil
    (cond ((not (null still-func))
           (transfunction (first still-func))
           (terpri prt)
           (terpri prt)
           (setq still-func (rest still-func))
           (transglob))
          ((not (null still-rel))
           (transrelation (first still-rel))
```



```
           (terpr prt)
           (terpr prt)
           (setq still-rel (rest still-rel))
           (transglob))
          ((not (null still-prog))
           (transprog (first still-prog))
           (terpr prt)
           (terpr prt)
           (setq still-prog (rest still-prog))
           (transglob))
          (t))))

(def transfunction
  (lambda (oldname)
    (let ((definition (getd oldname)))
         (setq relation nil)
         (cond ((eq oldname 'quote)
                (princ '"quote(X,X)." prt))
               ((eq oldname 'patom)
                (princ '"patom(X,X) :- write(X)." prt))
               ((or (not (dtpr definition))
                    (member (first definition)
                            '(nlambda macro
                                      lexpr)
                            ))
                (terpr)
                (princ
                 (uconcat '"Uebersetzung der Funktion '"
                          oldname
                          '"' nicht moeglich"))
                nil)
               (t
                (ausgabe (cons oldname
                               (appendl (mapcar (function
                                                  create-variables)
                                                 (second
                                                  definition))
                                        '(? $res)))
                         (and-connect
                          (transbody
                           (rest (rest definition)))))))))

(def transrelation
  (lambda (oldname)
    (let ((definition (getd oldname)))
         (setq relation t)
         (cond ((eq oldname 'null)
                (princ '"null([])." prt))
               ((or (not (dtpr definition))
                    (member (first definition)
                            '(nlambda macro
                                      lexpr)
                            ))
```

```
              (terpr)
              (princ
               (uconcat '"Uebersetzung des Praedikates '"
                        oldname
                        '"' nicht moeglich"))
              nil)
             (t
              (ausgabe (cons oldname
                             (mapcar (function
                                      create-variables)
                                     (second definition)))
                       (and-connect
                        (transbody
                         (rest (rest definition)))))))))

(def transprog
  (lambda (oldname)
    (let ((definition (getd oldname)))
         (cond ((or (not (dtpr definition))
                    (member (first definition)
                            '(nlambda macro
                                      lexpr)
                            ))
              (terpr)
              (princ
               (uconcat '"Uebersetzung der prog-Funktion '"
                        oldname
                        '"' nicht moeglich"))
              nil)
             (t
              (ausgabe (cons (concat oldname '"$p")
                             (mapcar (function
                                      create-variables)
                                     (second definition)))
                       (and-connect
                        (transbody-prog
                         (rest (rest definition)))))))))))


; Die Funktion transbody  hat als Wert die Liste der disjunktiven
; Normalformen der einzelnen Funktionsaufrufe des Rumpfes action.
; action ist eine Liste  von n Funktionsaufrufen, deren erste n-1
; Aufrufe lediglich Seiteneffekte ausfuehren, waehrend der letzte
; Funktionsaufruf den Wert angibt (vgl. progn).
; Die Art der Uebersetzung des  n-ten  Funktionsaufrufs  wird be-
; stimmt durch die Werte von progfct und relation.
; Die einzelnen DNFs muessen noch  durch  and-connect  verknuepft
; werden  (siehe  die  Aufrufe  von  transbody  in transfunction,
; transrelation usw.).

(def transbody
  (lambda (action)
    (cond ((null (rest action))
```



```
           (cond ((and (listp (first action))
                       (eq (first (first action))
                           'cond))
                  (list (transcond nil (rest (first action)))))
                 (relation
                  (list (transaction-r (first action))))
                 (t (list (transaction (first action))))))
          ((and (listp (first action))
                (eq (first (first action)) 'cond))
           (cons (transcond-prog nil (rest (first action)))
                 (transbody (rest action))))
          (t
           (cons (transaction-prog (first action))
                 (transbody (rest 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
           (and-connect (transbody (rest condclause)))))
         (list (and-connect
                (list neglist
                      (mapcar (function transcondition)
                              (newcondlist condlist))
                      actionlist))
               (and-connect
                (list neglist
                      (mapcar (function transcondition)
                              (negate condlist)))))))))

(def transbody-prog
  (lambda (action)
    (cond ((null (rest action))
           (cond ((and (listp (first action))
                       (eq (first (first action))
                           'cond))
                  (list (transcond-prog nil
                                        (rest (first action)))))
                 (t (list (transaction-prog (first action))))))
          ((and (listp (first action))
                (eq (first (first action)) 'cond))
           (cons (transcond-prog nil (rest (first action)))
```

```
          (transbody-prog (rest action))))
     (t
      (cons (transaction-prog (first action))
            (transbody-prog (rest action)))))))


ef transcond-prog
(lambda (neglist condcascade)
  (cond ((null condcascade) nil)
        (t
         ((lambda (x)
                  (append (first x)
                          (transcond-prog (second x)
                                          (rest condcascade))))
          (trclause-prog neglist (first condcascade)))))))

lef trclause-prog
(lambda (neglist condclause)
  (let ((condlist (transform-dnf (first condclause)))
        (actionlist
         (and-connect (transbody-prog (rest 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 '(? $res))))
         ((atom term)
          (list
           (cons '=
                 (list '(? $res)
                       (create-variables term)))))
```



```
         ((eq (first term) 'progn)
          (and-connect (transbody (rest term))))
         ((and (symbolp (first term)) (getd (first term)))
          (list
           ((lambda (action)
                    (appendl action
                             (cons '=
                                   (cons '(? $res)
                                         (last
                                          (first
                                           (last
                                            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 (create-variables term)))))
          ((eq (first term) 'progn)
           (and-connect (transbody (rest term))))
          ((and (symbolp (first term)) (getd (first term)))
           (mapcar (function flat-rel)
                   (transform-dnf (create-variables term)))))))

(def transaction-prog
  (lambda (term)
    (cond ((atom term) (list nil))
          ((eq (first term) 'progn)
           (and-connect (transbody-prog (rest term))))
          ((and (symbolp (first term)) (getd (first term)))
           (list (opt-flattenprog (create-variables term)))))))

(def ausgabe
  (lambda (conclusion premiselists)
    (cond ((null premiselists) t)
          (t
           (mapcar (function
                    (lambda (x)
                            (printclause
                             (optimize conclusion x)))
                    )
                   premiselists)))))

(def printclause
  (lambda (clause)
    (cond ((null (rest clause))
           (princ (transconclusion (first clause)) prt)
           (princ '"." prt)
```

```
          (terpri prt)
          (terpri prt)
          t)
         (t (princ (transconclusion (first clause)) prt)
            (princ '" :-" prt)
            (printpremiselist (rest clause))
            (princ '"." prt)
            (terpri prt)
            (terpri prt)
            t))))

(def printpremiselist
  (lambda (premiselist)
    (cond ((null (rest premiselist))
          (tab tabulator prt)
          (printpremise (first premiselist)))
         (t (tab tabulator prt)
            (printpremise (first premiselist))
            (princ '"," prt)
            (printpremiselist (rest premiselist))))))


(def printpremise
  (lambda (structure)
    (cond ((null structure) (princ 'fail prt))
         ((atom structure) (princ 'true prt))
         ((eq (first structure) 'is)
          (princ (uconcat (terms (second structure))
                          '" is "
                          (arith-expr (third structure)))
                 prt))
         ((eq (first structure) 'not)
          (princ '"not(" prt)
          (printpremise (second structure))
          (princ '")" prt))
         ((variable-p structure)
; kann nur vorkommen, falls der Wert von structure ein Atom ist
; (siehe transcondition)
          (princ (variab structure) prt))
         ((eq (first structure) 'print)
; nur richtig, falls print als Praedikat vorkommt
          (princ (uconcat (prologout (second structure))
                          '", fail")
                 prt))
         ((eq (first structure) 'patom)
; nur richtig, falls patom als Praedikat vorkommt
          (princ (uconcat (prologout (second structure))
                          '",")
                 prt)
          (tab tabulator prt)
          (princ (uconcat '"not(null("
                          (terms (second structure))
                          '"))")
```



```
                 prt))
         ((memq (first structure) outfunctions)
          (princ (prologout (second structure)) prt))
         ((memq (first structure) infunctions)
          (princ (prologin structure) prt))
         ((memq (first structure) newli)
          (princ '"nl, fail" prt))
         ((memq (first structure) compares)
          (princ (comp structure) prt))
         ((assq (first structure) meta-logical)
          (princ (uconcat (second
                           (assq (first structure)
                                 meta-logical))
                          '"("
                          (terms (second structure))
                          '")")
                 prt))
         ((eq (first structure) 'unify)
          (princ (uconcat (terms (second structure))
                          '" = "
                          (terms (third structure)))
                 prt))
         ((memq (first structure) outfunctions-prog)
          (princ (prologout (second structure)) prt))
         ((memq (first structure) newli-prog)
          (princ 'nl prt))
         ((null (rest structure))
          (princ (first structure) prt))
         (t
          (princ (uconcat (first structure)
                          '"("
                          (component (rest structure))
                          '")")
                 prt)))))
```

Optimierung:

```
setq optimizefns
     '(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)
                                (member (second premise)
                                        conclusion))
                           (shorten
                            (subst-any-level
                             nil
                             (second premise)
                             (list conclusion
                                   premiselist-left
                                   (rest premiselist-right)))))
                          ((and (eq functor 'zerop)
                                (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)
                           (car-treat (second premise)
                                      (third premise)))
                          ((eq functor 'cdr)
                           (cdr-treat (second premise)
                                      (third premise)))
                          ((eq functor 'cons)
                           (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)
                     (null (second premise)))
                (dele (rest premiselist)))
               ((and (eq (first premise) 'atom)
                     (atom (second premise)))
                (dele (rest premiselist)))
               ((and (eq (first premise) 'zerop)
                     (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)
                            (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)
                            (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 (argl arg2)
    (cond ((variable-p argl)
           (shorten
            (subst-any-level (car-construct arg2)
                             argl
                             (list conclusion
                                   premiselist-left
                                   (rest premiselist-right)))))
          ((listp argl)
           (shorten
            (subst-any-level (first argl)
                             arg2
                             (list conclusion
                                   premiselist-left
                                   (rest premiselist-right))))))))

(def cdr-treat
  (lambda (argl arg2)
    (cond ((variable-p argl)
           (shorten
            (subst-any-level (cdr-construct arg2)
                             argl
                             (list conclusion
                                   premiselist-left
                                   (rest premiselist-right)))))
          ((listp argl)
           (shorten
            (subst-any-level (rest argl)
                             arg2
                             (list conclusion
                                   premiselist-left
                                   (rest premiselist-right))))))))
```

```
;Transformation in disjunktive Normalform:
;----------------------------------------

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

(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))
          ((eq (first condition) 'cond)
           (cond-treat nil (rest condition)))
          ((and (eq (first condition) 'is)
                (listp (third condition))
                (eq (first (third condition)) 'cond))
           (is-treat (list (first condition)
                           (second condition))
```



```
                       (rest (first (rest (rest condition))))
                       nil))
          (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 ((null (second neg)) (list (list t)))
          ((eq (second neg) t) (list (list nil)))
          ((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)))))))


(def cond-treat
  (lambda (neg condcascade)
    (cond ((null condcascade) nil)
          (t
           (append (and-connect
                    (list neg
                          (newcondlist
```

```
                      (transform-dnf
                       (first (first condcascade))))
                      (transform-dnf
                       (second (first condcascade)))))
                (cond-treat (and-connect
                              (list neg
                                    (not-treatm
                                     (list 'not
                                           (first
                                            (first
                                             condcascade))))))
                             (rest condcascade)))))))

(def is-treat
  (lambda (prefix condcascade neg)
    (cond ((null condcascade) nil)
          (t
           (append (and-connect
                     (list neg
                           (newcondlist
                            (transform-dnf
                             (first (first condcascade))))
                           (transform-dnf
                            (appendl prefix
                                     (second
                                      (first condcascade)))))))
                   (is-treat prefix
                             (rest condcascade)
                             (and-connect
                              (list
                               neg
                               (not-treatm
                                (list 'not
                                      (first
                                       (first
                                        condcascade)))))))))))
```



```
;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))))))))

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

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

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

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

ef eqsets
(lambda (ml m2)
  (cond ((and (null ml) (null m2)) t)
        ((member (first ml) m2)
         (eqsets (rest ml) (remove (first ml) m2)))
        (t nil))))

ef 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 ((eq term 't) t)
          ((null term) nil)
          ((symbolp term) (list '? term))
          ((atom term) term)
          ((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))))

(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)))

(def flat-prog
  (lambda (liste)
    (mapcan (function opt-flattenprog) liste)))
```

```
;Abflachung geschachtelter Funktionsaufrufe
;------------------------------------------

(setq flatten.progfns
      '(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
              opt-flattenprog
              flattenfunc
              flattenrel
              flattenprog
              mind-func
              mind-rel
              still-func
              still-rel
              laast-1
              last-1
              make-uniform
              member-1
              delete-member
              delete-notarith-member
              delete-arith-member
              not-arith-func-sym
              opt
              quote-sym
              replace-1
              search))

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

(def insert-arg
  (lambda (elem list)
    (append (begin list) (list (cons elem (last-1 list))))))
```



```
(def replace-arith-resvar
  (lambda (arith-resvar list)
    (append (begin list)
            (list
             (append (begin (last-1 list))
                     (list arith-resvar))))))

(def append-and-insert-resvar-1
  (lambda (list1 list2)
    (append list1
            (append (begin list2)
                    (list
                     (cons (laast-1 list1) (last-1 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 (list1 list2)
    (cond ((equal (length list1) 1)
           (cond ((equal (length list2) 1)
                  (list (cons (car list1) (car list2))))
                 (t
                  (cons (car list2)
                        (list
                         (cons (car list1) (cadr list2)))))))
          (t
           (cond ((equal (length list2) 1)
                  (cons (car list1)
                        (list
                         (cons (cadr list1) (car list2)))))
                 (t
                  (cons (append (car list1) (car list2))
                        (list
                         (cons (cadr list1) (cadr list2))))))))))

(def append-and-insert-resvar-2
  (lambda (list1 list2)
    (cond ((equal (length list2) 1)
           (cons list1
                 (list (cons (laast-1 list1) (car list2)))))
          (t
           (cons (append list1 (car list2))
                 (list (cons (laast-1 list1) (cadr list2))))))))
```

```
ief 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)))))))


ief append-and-insert-arith-resvar
 (lambda (list1 list2)
   (append list1
           (append (begin list2)
                   (list
                    (cons (cadr (last-1 list1))
                          (last-1 list2)))))))


ief 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-1 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
             add1
             |1+|
             sub1
             |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)
          (opt (flattenfunc list))))


(def opt-flattenrel
  (lambda (list)
    (cond ((equal (car list) 'is)
           (cond ((or (atom (caddr list))
                      (variable-p (caddr list)))
                  (list
                   (list 'unify
                         (cadr list)
                         (caddr list))))
                 ((member (car (last-l list)) arith-functions)
                  (replace-resvar (cadr list)
                                  (opt
                                   (flattenfunc
                                    (caddr list)))))
                 ((and (symbolp (car (last-l list)))
                       (getd (car (last-l list))))
                  (replace-arith-resvar (cadr list)
                                        (opt
                                         (flattenfunc
                                          (caddr list)))))
                 (t
                  (list
                   (list 'unify
                         (cadr list)
                         (caddr list))))))
          (t (opt (flattenrel list))))))


(def opt-flattenprog
  (lambda (list)
    (opt (flattenprog 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))
           (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 (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 (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)))))))

(def flattenprog
  (lambda (list)
    (cond ((null list) (list nil))
          ((variable-p (car list))
           (insert-arg (car list) (flattenprog (cdr list))))
          ((eq (car list) 'not)
           (insert-not 'not (flattenprog (cadr list))))
          ((and (symbolp (car list)) (getd (car list)))
           (cond
            ((not (member (car list) mind-prog))
             (setq mind-prog (cons (car list) mind-prog))
             (setq still-prog (appendl still-prog (car list)))))
           (insert-arg (concat (car list) '"$p")
                       (flattenprog (cdr list))))
          ((and (listp (car list)) (arith-func-sym (caar list)))
```

```
          (append-and-insert-arith-resvar (arith-func
                                           (car list))
                                          (flattenprog
                                           (cdr list))))
         ((and (listp (car list))
               (not-arith-func-sym (caar list)))
          (append-and-insert-resvar-1 (flattenfunc (car list))
                                      (flattenprog (cdr list))))
         (t (insert-arg (car list) (flattenprog (cdr list)))))))


setq mind-func 'nil)

setq mind-rel 'nil)

setq mind-prog nil)

setq still-func 'nil)

setq still-rel 'nil)

setq still-prog nil)

def laast-1
 (lambda (list)
   (last-1 (last-1 list))))


def last-1
 (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-1
```



```
  (lambda (x list)
    (cond ((null list) nil)
          ((and (listp (car list))
                (not (variable-p (car list))))
           (or (member-1 x (car list)) (member-1 x (cdr list))))
          (t (or (equal x (car list)) (member-1 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-1
                                            (car list2))
                                           (last-1 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 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-l
  (lambda (x y list)
    (cond ((null list) nil)
          ((and (listp (car list))
                (not (variable-p (car list))))
           (cons (replace-l x y (car list))
                 (replace-l x y (cdr list))))
          ((equal x (car list)) (cons y (cdr list)))
          (t (cons (car list) (replace-l x y (cdr list)))))))


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


Anhang C: Listing der LISP-PROLOG-Transformation mit Cut

Um LISP-Funktionsdefinitionen mit allgemeinem Cut-Operator zu uebersetzen (siehe Abschnitt 5), wurden gegenueber den in Anhang B angegebenen Listings einige Funktionen geaendert, andere Funktionen wurden weggelassen.

Folgende Funktionen wurden gestrichen:

```
newcondlist
negate-and-append
negate
cnf-to-dnf
neg-elements
without-equals
remequals
remequalsets
eqsets
element
delete-all
```

Folgende Funktionen wurden geaendert:

```
transbody
transcond
trclause
transbody-prog
transcond-prog
trclause-prog
transcondition
transform-dnf
cond-treat
is-treat
```

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


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


(def transcond
  (lambda (condcascade)
    (mapcan (function trclause) condcascade)))


(def trclause
  (lambda (condclause)
    (and-connect (cons (mapcar (function transcondition)
                               (transform-dnf (first condclause)))
                       (transbody (rest condclause))))))


(def transbody-prog
  (lambda (action)
    (cond ((null (rest action))
           (cond ((and (listp (first action))
                       (eq (first (first action))
                           'cond))
                  (list (transcond-prog (rest (first action)))))
                 (t (list (transaction-prog (first action))))))
          ((and (listp (first action))
                (eq (first (first action)) 'cond))
           (cons (transcond-prog (rest (first action)))
                 (transbody-prog (rest action))))
          (t
           (cons (transaction-prog (first action))
                 (transbody-prog (rest action)))))))
```



```
(def transcond-prog
  (lambda (condcascade)
    (mapcan (function trclause-prog) condcascade)))


(def trclause-prog
  (lambda (condclause)
    (and-connect (cons (mapcar (function transcondition)
                               (transform-dnf (first condclause)))
                       (transbody-prog (rest condclause))))))


(def transcondition
  (lambda (conjunction)
    (cond ((null conjunction) '(("!")))
          ((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)))))))
```

```
;Transformation in disjunktive Normalform:
;-----------------------------------------


(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))
          ((eq (first condition) 'cond)
           (cond-treat (rest condition)))
          ((and (eq (first condition) 'is)
                (listp (third condition))
                (eq (first (third condition)) 'cond))
           (is-treat (list (first condition)
                           (second condition))
                     (rest (first (rest (rest condition))))))
          (t (list (list condition))))))


(def cond-treat
  (lambda (condcascade)
    (mapcan (function
             (lambda (x) (and-connect
                          (list (transform-dnf (first x))
                                '((("!")))
                                (transform-dnf (second x))))))
            condcascade)))


(def is-treat
  (lambda (prefix condcascade)
    (mapcan (function
             (lambda (x) (and-connect
                          (list (transform-dnf (first x))
                               '((("!")))
                               (transform-dnf
                                (append1 prefix (second x)))))))
            condcascade)))
```