RESEARCH UNIT FOR
INFORMATION SCIENCE AND
ARTIFICIAL INTELLIGENCE

Director: Prof. Dr. W. v. Hahn

University of Hamburg
Mittelweg 179
D-2000 Hamburg 13
Tel.: (040) 4123-4529

HAM
ANS

Memo ANS-22

ULM: EIN UCI-LISP - LISPMASCHINEN-LISP ÜBERSETZUNGS-SYSTEM

Bernhard Nebel

Juli 1984

ISSN 0722-950X

ULM: EIN UCI-LISP - LISPMASCHINEN-LISP ÜBERSETZUNGS-SYSTEM

Bernhard Nebel

Forschungsstelle für Informationswissenschaft
und Künstliche Intelligenz
Universität Hamburg

ABSTRACT

In this paper we describe the rule driven LISP inter-dialect translation system ULM, which translates from UCI-LISP to LM-Lisp (ZetaLisp). It was developed using the FRANZLATOR, a translator from INTERLISP to FranzLisp, but it has an improved rule interpreter and a more extensive rule set. The design considerations for the rule set and improvements in the interpreter are discussed in detail. Three UCI-lisp systems have been translated using ULM so far, the AI language interpreter micro-FIT, a set of benchmark functions and one of the components of the Natural Language system HAM-ANS, the module VERBALIZE. Some interesting results of the quantitative performance differences between UCI-LISP on the DEC-10(KI) and LM-Lisp on the Symbolics 3600 are presented at the end of the paper.

ULM: EIN UCI-LISP - LISPMASCHINEN-LISP ÜBERSETZUNGS-SYSTEM

Bernhard Nebel

Forschungsstelle für Informationswissenschaft
und Künstliche Intelligenz
Universität Hamburg

1. EINLEITUNG

Die Übersetzung von LISP-Programmen von einem LISP-Dialekt in den anderen ist ein Thema, das seine Aktualität nicht verliert (vgl. [TEITELMAN 78, Kapitel 24], [FININ 82], [LAMPING & KING 82] und [Memo ANS-17]). Dies liegt vor allem daran, daß es keinen Standard für LISP gibt. Und es ist nicht zu erwarten, daß Versuche in diese Richtung, wie z.B. Common LISP [STEELE 82] dazu führen werden, daß alle anderen LISP-Dialekte verschwinden. Dafür sind Dialekte wie z.B. INTERLISP [TEITELMAN 78] oder FranzLisp [FODERADO 82] viel zu verbreitet und gut eingeführt. Es ist im Gegenteil eher zu vermuten, daß der Bedarf an guten LISP-Interdialekt-Übersetzungssystemen zunehmen wird.

Das ULM-System dient zur Übersetzung von UCI-LISP [MEEHAN 79] nach Lispmaschinen-Lisp (kurz LM-Lisp) [WEINREB & MOON 81] und ist in LM-Lisp geschrieben. Es baut auf den FRANZLATOR auf, ein regelgesteuertes System zur Übersetzung von INTERLISP nach FranzLisp [FININ 82]. Mitglieder des Projektes HAM-ANS (HAMburger Anwendungsorientiertes Natürlichsprachliches System) brachten den FRANZLATOR von einer Vortrags- und Informationsreise aus den USA mit [Memo ANS-19].

ULM soll exemplarisch die Portierbarkeit des natürlichsprachlichen Systems HAM-ANS, das in UCI-LISP/FUZZY [MEEHAN 79] [LE FAIVRE 78] implementiert ist, belegen. Es ist damit als Fortführung der Portierungsstudie von Th. Christaller [Memo ANS-17] zu verstehen.

Dieses Übersetzugssystem hat wie alle LISP-Interdialekt-Übersetzer nicht den Anspruch, eine vollständige Übersetzung zu liefern. Das ist aufgrund von Eigenschaften der Sprache LISP, wie Daten-Programm-Äquivalenz, vom Benutzer modifizierbare Syntaxtabelle und der Verzahnung von Programmierumgebung und Benutzerprogramm, nicht möglich. Es wird stattdessen eine 95%-ige Übersetzung geboten und der nicht übersetzbare Rest, sowie kritische Konstrukte werden mit Warnungen versehen, so daß diese Stellen "per Hand" nachübersetzt werden können.

Im folgenden Abschnitt wird kurz der FRANZLATOR beschrieben. Dieser Abschnitt kann übergangen werden, falls dem Leser [FININ 82] bekannt ist. Im dritten Abschnitt werden die Adaption und die Erweiterung des Übersetzungsalgorithmus vorgestellt und Angaben über Umfang und Leistungsfähigkeit der Transformationsregeln gemacht.

Dieser Aufsatz entstand im Rahmen des Projekts HAM-ANS, das aus Mitteln des BMFT gefördert wird.

Erfahrungen bei der Übersetzung einiger Systeme werden in Abschnitt 4 geschildert. Dazu gehört auch die Übersetzung und Ausführung eines Satzes von Benchmark-Funktionen, die für einen Laufzeitvergleich von LISP- und PASCAL-Programmen auf einem DECSystem 1070 und einer VAX 11/780 benutzt worden sind [NEBEL 83]. Der Vergleich der auf der Lispmaschine - einer Symbolics 3600 - gewonnenen Testdaten mit den vorher erhaltenen Daten brachte interessante Resultate.

In den Anhängen wird die Benutzung des ULM-Systems beschrieben, auf kritische Sprachkonstrukte hingewiesen und die in deklarativer Form vorliegenden Übersetzungsregeln aufgelistet.

2. *DER FRANZLATOR: EIN INTERLISP-FRANZLISP TRANSLATOR*

Der FRANZLATOR wurde implementiert, um das KL-ONE System [SCHMOLZE & BRACHMAN 82] von INTERLISP nach FranzLisp zu übersetzen. Es wurde dabei der Übersetzungs- im Gegensatz zum Emulationsansatz verfolgt, d.h. es wurde versucht, möglichst alle Ausgangssprachkonstrukte in Zielsprachkonstrukte zu übersetzen und die direkte Nachbildung von INTERLISP-Funktionen in FranzLisp zu vermeiden. Tatsächlich handelt es sich um ein gemischtes System, das aber im Gegensatz zu einem Emulator die FranzLisp-Umgebung intakt läßt, d.h. die Zeichensyntax-Tabelle und die Systemfunktionen von FranzLisp werden nicht überschrieben. Damit ist die Kombination von existierenden FranzLisp-Programmen mit dem übersetzten KL-ONE System ohne weiteres möglich.

Die Übersetzung wird durch Transformationsregeln gesteuert, ähnlich den in Expertensystemen oft verwendeten Produktionsregeln, die ursprünglich von Post [POST 43] als Formalismus zur Formulierung von Algorithmen eingeführt wurden. Um das Schreiben der Transformationsregeln zu vereinfachen, bietet der FRANZLATOR die Möglichkeit, Regeln mit LISP-Funktionen zu annotieren, die die Aufgabe von zusätzlichen Tests, Berechnungen usw. übernehmen. Die regelgesteuerte Übersetzung hat gegenüber anderen Ansätzen, bei denen z.B. der Prozeß der Transformation mithilfe von Editorkommandos beschrieben wird, den Vorteil, daß die Transformationsvorschrift einfach zu formulieren und zu modifizieren ist.

Der FRANZLATOR besteht aus 5 Teilen (vgl. Abb. 1):

- dem Programm, bestehend aus Regelinterpreter und globaler Ablaufsteuerung,
- einer Definition der INTERLISP-Zeichensyntax,
- einer Datenbasis mit Informationen über alle INTERLISP-Funktionen,
- einer Datenbasis mit Transformationsregeln und
- einer Menge von in FranzLisp geschriebenen INTERLISP-Kompatibilitätsfunktionen.

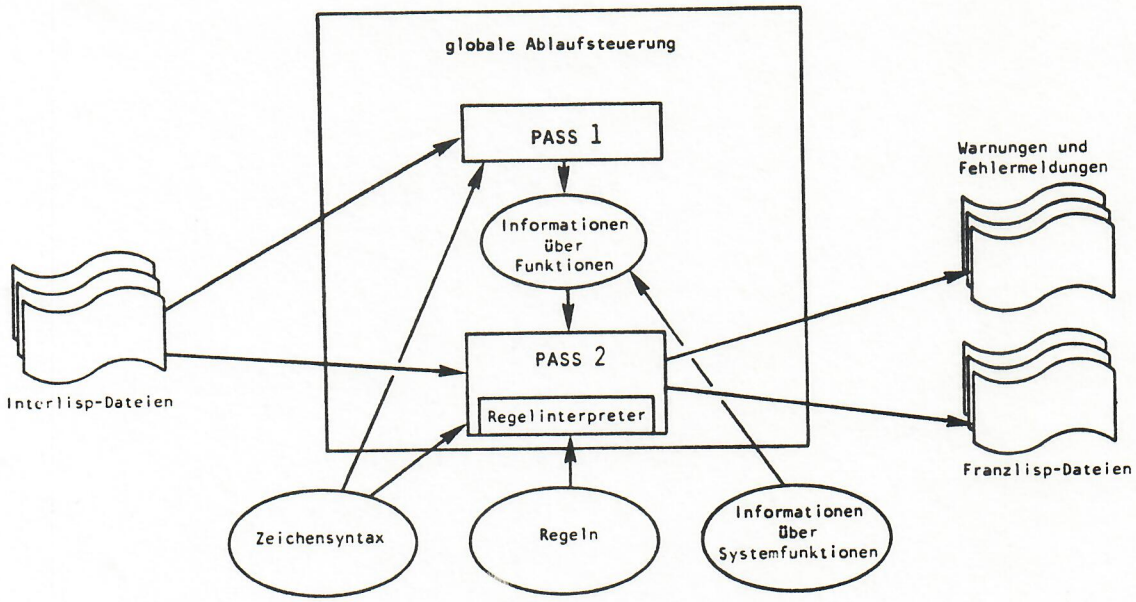

Abbildung 1: Struktur des FRANZLATORS

Es werden zwei Läufe über alle zu übersetzenden Dateien ausgeführt. Im ersten Lauf werden Informationen, wie Anzahl der Parameter und Art der Funktionen, über alle definierten Funktionen gesammelt. Diese Informationen werden während des zweiten Laufes benutzt, in dem die eigentliche Übersetzung stattfindet. Für jede zu übersetzende Datei wird eine Datei mit Bemerkungen, wie z. B. Warnungen bei nicht definierten Funktionen oder Fehlermeldungen bei fehlgeschlagenen Übersetzungen, erzeugt. Die Übersetzung, d.h. die Anwendung von Transformationsregeln, erfolgt sequentiell S-Ausdruck für S-Ausdruck, wobei zuerst rekursiv die Parameter jedes Funktionsaufrufes und dann der Ausdruck selbst übersetzt wird. Die Reihenfolge der Transformationsregelanwendungen ist also die gleiche wie bei der Evaluation der S-Ausdrücke durch den LISP-Interpreter.

Die Behandlung von Funktionen, die ihre Argumente nicht bei Aufruf evaluieren, ist naturgemäß schwierig. Falls nichts anderes spezifiziert wird, wird bei FEXPRs kein Parameter übersetzt, bei MACROS dagegen alle. Für die Systemfunktionen SELECTQ, COND, QUOTE usw. gibt es spezielle Parametertransformationsfunktionen.

Die Transformationsregeln bestehen aus zwei obligatorischen Teilen, dem Pattern- und dem Resultat-Teil. Daneben können optionale Attribute angegeben werden. Die Regeln haben folgende Syntax:

```
<Regel> ::= (<Pattern> <Resultat> . <Attributliste>)
<Attributliste> ::= () | (<Attribut> . <Attributliste>)
<Attribut> ::= <Attributname> <Attributwert>
<Attributname> ::= test | side-effect | priority | type | regiem
<Attributwert> ::= <S-Ausdruck>
```

Beispiele für Regeln sind:

```
[1]  (NIL nil)
[2]  ((NLISTP ,x) (not (dtpr ,x)))
[3]  ((MAPCAR ,list (FUNCTION ,f))
      (mapcar ',(makeMonadic f) ,list))
[4]  ((PLUS ,@a ,x ,@b ,y ,@c)
      (PLUS ,@a ,@b ,@c ,(+ x y))
      test (and (numberp x) (numberp y))
      regiem cyclic)
```

Regel [1] ist die einfachste Art einer Übesetzungsregel, die das Symbol "NIL" aus INTERLISP in das FranzLisp-Atom "nil" übersetzt. In der Regel [2] wird eine Pattern-Variable benutzt, die durch den ","-Prefix gekennzeichnet ist. Diese Notation entspricht der MacLisp Backquote-Konvention (vgl. [WEINREB & MOON 83]. Innerhalb des Pattern-Teils nimmt die Pattern-Variable einen Wert - einen S-Ausdruck - an, den sie im Resultatteil wieder abgibt. In der Regel [3] wird im Resultatteil das Ergebnis eines Funktionsaufrufs eingetragen. Das zweite Element im Resultatteil wird das gequotete Ergebnis des Funktionsaufrufes (makeMonadic f), wobei an f der Wert der Pattern-Variable ,f gebunden ist. Die Regel [4] führt die Verwendung von Pattern-Segment-Variablen mit dem ",@"-Prefix vor, die eine beliebige Anzahl (einschließlich 0) von S-Ausdrücken matchen können. Außerdem wird die Angabe von Attributen demonstriert.

Das SIDE-EFFECT Attribut führt einen LISP-Ausdruck ein, der bei erfolgreicher Anwendung evaluiert wird. Er wird normalerweise dazu benutzt, Meldungen auszugeben.

Das TEST-Attribut kann wie in [4] benutzt werden, um die Anwendbarkeit einer Regel zu steuern. Da der Test-Ausdruck erst nach einem erfolgreichen Match des Pattern-Teils erfolgt, kann beim Test von gebundenen Pattern-Variablen ausgegangen werden - in Regel [4] ,x und ,y.

Das TYPE-Attribut kontrolliert, ob das Resultat in dem umgebenden S-Ausdruck einfach eingesetzt wird, oder aber als Segment - durch Auslassung der beiden äußeren Klammern - eingetragen wird. Allerdings war dies Attribut in dem System, das wir erhalten haben, ebenso wie das PRIORITY- und das REGIEM-Attribut, die Kontrollaspekte betreffen, nicht implementiert.

Ein wesentlicher Aspekt bei der Implementation des FRANZLATORs ist die Repräsentation der Regeln. Diese werden per Makroexpansion in LISP-Funktionen übersetzt und nach dem ersten Element des Pattern-Teils indiziert, soweit es sich um Listen mit einem Atom als erstem Element handelt. Damit ist auch bei großen Regelmengen eine effiziente Abarbeitung der Regeln gewährleistet.

3. ADAPTION DES FRANZLATORS

Um den FRANZLATOR an unsere Bedürfnisse anzupassen, war es notwendig, das Programm von FranzLisp nach LM-Lisp zu übersetzen und die Datenbasen der Transformationsregeln und Informationen über Systemfunktionen so zu verändern, daß der Translator von UCI-LISP nach LM-Lisp übersetzt. Dies wird in der Abb. 2 mithilfe der T-Diagramm-Darstellung illustriert.

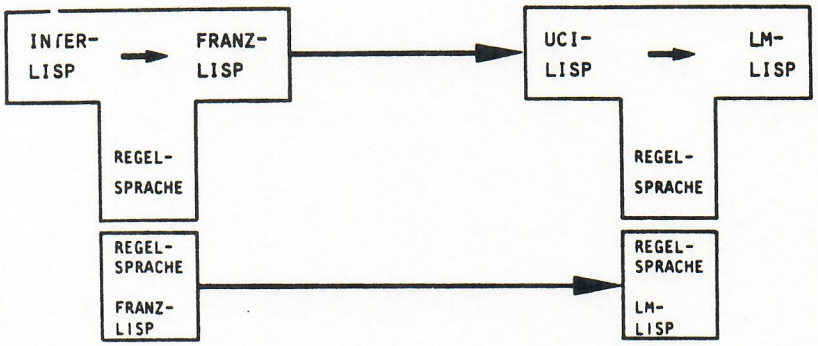

Abbildung 2: Anpassung des FRANZLATORS

Die Übersetzung von FranzLisp nach LM-Lisp war relativ einfach, da beide Dialekte aus MacLisp [MOON 74] hervorgegangen sind und deshalb viele gemeinsame Sprachelemente besitzen. Da jedoch der Regelinterpreter einige Schwächen aufwies, mußte er an vielen Stellen verändert und erweitert werden. Die Transformationsregeln und Informationen über Systemfunktionen mußten natürlich völlig neu geschrieben werden.

3.1. KRITIK AM FRANZLATOR

Leider waren einige Funktionen, die in [FININ 82] beschrieben waren, nicht implementiert. Dazu gehörte die Übersetzung von Atomen - die Regel [1] kann gar nicht angewandt werden - sowie die Interpretation der Attribute TYPE, REGIEM und PRIORITY.

Bei der Atomübersetzung wäre es zusätzlich wünschenswert, wenn diese auch bei der Übersetzung von S-Ausdrücken benutzt wird, die als erstes Element das Atom habe, das den Patternteil einer Regel ausmacht. Wenn man also statt der Regel

```
((foo ,∂x) (bar ,∂x))
```

nur noch

```
(foo bar)
```

schreiben muß. Dies hat den Vorteil der ökonomischen Notation und bewirkt außerdem, daß der Funktionsname auch in Konstrukten übersetzt wird, in der kein direkter Funktionsaufruf erfolgt, wie z.B.

(eval (cons 'foo arglist)).

Ein weiterer Kritikpunkt ist, daß der 1. Lauf, in dem Informationen über die vom Benutzer definierten Funktionen gesammelt werden, nicht durch Regeln formalisiert ist, sondern einfach durch das Programm geleistet wird.

Der schwächste Punkt innerhalb des Regelinterpreters ist die im folgenden beschriebene Kontrollstrategie:

(1) Finde unter den Regeln, die im ersten Element des Pattern-Teils eine Konstante haben, eine anwendbare Regel, wobei in der Reihenfolge der Regeleintragungen gesucht wird. Wende diese Regel an, markiere sie als für diesen S-Ausdruck nicht mehr anwendbar (um Totschleifen zu vermeiden) und wiederhole den Schritt. Gibt es keine anwendbare Regel, gehe zu (2).

(2) Finde unter den Regeln mit einer Pattern-Variablen als erstem Element, die ebenfalls nach der Reihenfolge des Eintragens geordnet sind, eine anwendbare Regel. Gibt es eine solche, verfahre wie unter (1) und wiederhole Schritt (2). Gibt es keine anwendbare Regel mehr, so ist der Übersetzungsvorgang für diesen S-Ausdruck abgeschlossen.

Diese Strategie hat folgende Schwächen:

(1) Aufgrund der realisierten Konfliktresolution ist es nicht möglich, eine allgemeine Regel vor allen speziellen Regeln auszuführen. Konkret existiert im FRANZLATOR eine allgemeine Regel, die fehlende Argumente ergänzt und überflüssige Argumente entfernt. Diese wird erst nach der Anwendung aller speziellen Regeln aktiviert. Zu diesem Zeitpunkt ist aber der Funktionsname schon geändert, so daß nicht mehr auf die Information zugegriffen werden kann, die unter dem ursprünglichen Funktionsnamen steht.

(2) Es wird nicht klar, ob die Übersetzung eines S-Ausdrucks erfolgreich war, da die Anwendung einer Regel kein hinreichendes Kriterium für den Erfolg ist und die Übersetzung auf jeden Fall mit der Nichtanwendbarkeit aller Regeln abbricht. Im FRANZLATOR kann diese Frage allerdings anhand der Groß-/Kleinschreibung entschieden werden. Da alle INTERLISP-Funktionen groß geschrieben werden und alle FranzLisp-Funktionen klein, war eine Übersetzung dann erfolgreich, wenn der neue Funktionsname kleingeschrieben ist.

(3) Eine ähnliche Problematik ergibt sich für den Abbruch von Übersetzungen nach einer erfolgreich angewendeten Regel. Mit der oben geschilderten Kontrollstrategie kann es passieren, daß ein erzeugtes Zielsprachkonstrukt fälschlicherweise für ein Ausgangssprachkonstrukt gehalten wird und noch einmal übersetzt wird. Dieses Problem tritt bei einer INTERLISP-FranzLisp Übersetzung aus den o.g. Gründen nicht auf, spielt aber z.B. bei einer UCI-LISP LM-Lisp Übersetzung eine Rolle.

Als letzter Kritikpunkt ist zu nennen, daß im Gegensatz zum Anspruch, den Übersetzungs- und nicht den Emulationsansatz zu verfolgen, es nur 38 Regeln aber 250 INTERLISP-Kompatibilitätsfunktionen gibt. Von den 38 Regeln sind 4 atomare Regeln (die also nicht benutzt werden), 2 allgemeine Regeln zum Ergänzen von Parametern und für Defaultübersetzungen, 5 zum Erzeugen von Warnungen und von dem Rest unterscheiden sich 2 Regeln nur in dem Pattern-Variablennamen, so daß nur 26 Regeln die Übersetzung von spezifischen Funktionen behandeln.

3.2. ÜBERSETZUNG AUF DER ZEICHENEBENE

Beim Einlesen von UCI-LISP Programmen müssen folgende Konventionen beachtet werden:

- zwischen {;; und } sowie zwischen {; und } stehen Kommentare. Ein Kommentar der Art

```
(...)
{;; bla bla
 blub blub}
(...)
```

  sollte in LM-Lisp folgendermaßen dargestellt werden

```
(...)
;;bla bla
;;blub blub
(...)
```

- "@" ist ein zusätzliches QUOTE-Zeichen.

- Statt runder Klammern können auch eckige Klammern verwendet werden.

- Das Zeichen "/", daß bewirkt, das das nächste Zeichen ohne weitere Interpretation gelesen wird, allerdings nicht innerhalb von Zeichenketten.

- Die Zeichen ":", "|" und "#" haben in UCI-LISP keine besondere Bedeutung, während es in LM-Lisp Readmacros sind.

- "~" ist das sogenannte Scanner-Kommentar Zeichen. Alles zwischen "~" und dem neuen Zeilenanfang wird überlesen, also auch das Zeilenende, so daß es möglich ist, Symbole beim Ausdrucken zu trennen.

- Kleinbuchstaben sollen beim Einlesen nicht in Großbuchstaben umgewandelt werden, wie es in LM-LISP üblich ist.

- Die Backquote-Zeichen "`" und "," sind im Hamburger UCI-LISP, einer Erweiterung des UCI-LISP durch die HAM-ANS-Gruppe (vgl. [Online-Dokumentation]), ebenso wie in LM-Lisp definiert.

- Der Punkt kann in LM-Lisp Teil eines Bezeichners sein, falls er nicht nach einem Leerzeichen steht. In UCI-LISP ist dagegen der Punkt immer ein Seperator.

Eine Vorgehensweise, diese Konventionen zu behandeln, ist es, einen extra Lauf einzuführen, in dem eine zeichenweise Übersetzung durchgeführt wird. Dieser Lauf kann z.B. mit einem (möglichst programmierbaren) Texteditor wie z.B. ZMACS erledigt werden. Diese Lösung wurde bei dem INTERLISP LM-Lisp Translator IZZI gewählt [LAMPING & KING 82]. Eleganter ist es, die Zeichensyntaxtabelle (Readtable), so zu modifizieren, daß der Ausgangsdialekt ohne Schwierigkeiten gelesen oder der Zieldialekt direkt erzeugt werden kann, je nach dem ob der Translator im Ziel- oder Ausgangsdialekt geschrieben ist. Diese Modifikation der Readtable ist allerdings nicht in allen LISP-Dialekten möglich.

In unserem Fall konnte diese Möglichkeit ausgenutzt werden. Das Ausdrucken von Kommentaren in LM-Lisp ist durch die Definition entsprechender Printmacros möglich. Ein Problem ergibt sich bei der Behandlung von Readmacros, da in UCI-LISP diese Zeichen nur am Anfang eines Symbols erkannt werden (das Scanner-Kommentar-Zeichen ist eine Ausnahme) und in LM-Lisp Readmacros an jeder beliebigen Stelle stehen können (mit Ausnahme des #-Readmacros). Da jedoch die Definition der LM-Lisp-Readtable in deklarativer Form vorliegt, (allerdings ohne Dokumentation über ihren Aufbau) konnte eine neue Syntaxklasse eingeführt werden, die sog. "when-first" Readmacro-Zeichen, die der UCI-LISP Readmacro-Konvention entsprechen.

Die Behandlung des "/"-Zeichens erforderte zusätzlichen Aufwand. In LM-Lisp ist es möglich, mithilfe des "/" auch innerhalb von Zeichenketten einzelne Zeichen zu "quoten". Um ein "/"-Zeichen in eine Zeichenkette einzutragen, muß man also "//" schreiben. Diese Möglichkeit existiert in UCI-LISP nicht, so daß also die Stringbegrenzer nicht Teil des Strings sein können. Um UCI-LISP Zeichenketten korrekt einzulesen, wurde das Zeichen '"' als Readmacro definiert, daß das Einlesen der Zeichenkette mit einer modifizierten Readtable bewirkt, in der "/" ein normales Zeichen ist.

Der Einfachheit halber wurde von einer eingeschränkten Syntax ausgegangen, die den Konventionen des Ausgabesubsystems von UCI-LISP entspricht. Es wird davon ausgegangen, daß

- "." immer durch Leerzeichen abgegrenzt ist, wenn es sich um den "Listenpunkt" handelt,

- "]" nicht als "Superklammer" fungiert, d.h. zwischen [und] sind alle Klammern ausgewogen,

- "}" immer am Ende einer Zeile steht.

Diese Annahme ist zulässig, da i.a. die UCI-LISP Dateien nicht mit einem Texteditor sondern mit dem UCI-LISP System erstellt und bearbeitet werden.

Einziger Problemfall bei der Übersetzung auf der Zeichenebene ist das Scanner-Kommentarzeichen, das man in LM-Lisp nicht nachbilden kann. Es ist nicht möglich ein Symbol, das am Zeilenende mithilfe dieses Zeichens getrennt wurde, zusammenhängend einzulesen! Aus diesem Grunde sollte man beim Ausdrucken von UCI-LISP Dateien, die übersetzt werden sollen, eine möglichst grosse Ausgabezeilenlänge einstellen und einen Vorlauf mit dem ZMACS-Editor machen, in dem man nach der Sequenz "~<Zeilenende>" sucht.

Jedes zu übersetzende System kann zusätzlich noch eigene Readmacros definieren. Die vollständige Übersetzung von Readmacros wirft jedoch große Probleme auf und außerdem ist ein sofortiges Einlesen der übersetzten Funktionen nicht ratsam. Deshalb muß die Syntax für jedes System vorweg übersetzt werden und ULM als zusätzliche Datenbasis zugänglich gemacht werden. Da die Definitionen der Readmacros meist in einem Programmteil zusammengefaßt sind, scheint dies ein annehmbares Vorgehen zu sein.

3.3. ÜBERSETZUNG AUF DER EBENE DER S-AUSDRÜCKE

Bei der Übersetzung von einem Dialekt in einen anderen sind an sich nur Regeln notwendig, die für jedes Ausgangssprachkonstrukt ein Zielsprachkonstrukt angeben. Damit würde man jedoch Programme erzeugen, die ineffizient und vor allem unleserlich sind. Als Beispiel kann die Übersetzung des Prädikats CONSP von UCI-LISP nach LM-Lisp dienen. Während in UCI-LISP CONSP als Ergebnis NIL oder den Parameterwert liefert, gibt in LM-Lisp die entsprechende Funktion LISTP NIL oder T zurück. Eine korrekte Transformationsregel müßte also folgendermaßen aussehen:

```
((CONSP ,X) (LET ((X ,X)) (AND (LISTP X) X)))
```

Das LET ist notwendig, damit der CONSP-Parameter, der ja auch eine Funktion mit Seiteneffekten sein kann, nicht doppelt evaluiert wird. Wird CONSP allerdings in einer Umgebung aufgerufen, in der nur die Information benötigt wird, ob es sich um eine Liste handelt, also z.B. in einem NOT, so könnte CONSP direkt in LISTP übersetzt werden. Hat man jedoch nur die o.g. Transformationsregel, so lautet die Übersetzung für (NOT (CONSP L)):

```
(NOT (LET ((X L)) (AND (LISTP X) L),
```

was wohl eine etwas schwerfällige Formulierung für einen einfachen Test ist. Wünschenswert ist deshalb zusätzlich

(1) die Berücksichtigung des äußeren Kontextes, d.h. der Umgebung, in der die zu übersetzende Funktion aufgerufen wird,
(2) die Berücksichtigung des inneren Kontextes, d.h. der Parameter des aktuellen Aufrufs,
(3) die Optimierung nach der Übersetzung (ähnlich zu (2)),
(4) die Parametrisierbarkeit der Übersetzung,
(5) die Ausfaktorisierung von gemeinsamen Übersetzungsschritten verschiedener Regeln,
(6) das Erkennen und Markieren von nicht übersetzbaren Konstrukten (vgl. 3.1),
(7) die Ausgabe von Warnungen bei zweifelhaften Fällen,
(8) die Formalisierung des 1. Laufes (vgl. 3.1),
(9) die in 3.1 geschilderte verallgemeinerte Atom-Übersetzung.

Ein Beispiel für die Anwendung von (1) tritt neben dem eben gegebenen Beispiel bei der MAP-Funktion auf. Während in UCI-LISP die Funktion NIL als Ergebnis liefert, wird in LM-Lisp das zweite Argument zurückgegeben. Da MAP normalerweise nur wegen des Seiteneffekts aufgerufen wird, kann MAP in MAP übersetzt werden, außer in Kontexten, in denen das Ergebnis explizit benötigt wird. In diesem Fall muß das Übersetzungsergebnis (PROGN (MAP ...) NIL) lauten (vgl. Anhang D). Das Erkennen des Kontextes kann bei der implementierten Übersetzungsstrategie - "von innen nach außen" - durch eine TEST-Funktion geleistet werden, die den Keller der noch nicht transformierten S-Ausdrücke inspiziert.

Legt man die inverse Übersetzungsstrategie - "von außen nach innen" - zugrunde, so wäre es möglich die Kontextart, z.B. "nur Seiteneffekt", "nur boolsches Resultat", in die S-Ausdrücke jeweils einzutragen. Damit könnte man in diesem Fall ganz im Paradigma der Transformationsregeln bleiben und auf TEST-Funktionen verzichten.

Es spricht noch ein weiterer Punkt für die "von außen nach innen"-Strategie. Es gibt Funktionen, die sich in zwei ineinander geschachtelte Funktionsaufrufe übersetzen lassen, wobei für den inneren Aufruf schon Regeln existieren. Ein Beispiel ist (NOTANY...), das in

(NOT (SOME...)) übersetzt werden kann, wobei SOME je nach Argumentanzahl verschieden übersetzt werden muß. Bei einer "von außen nach innen"-Übersetzung würde sich die korrekte Behandlung dieses Falles von selber ergeben. Allerdings kann dieser Fall bei der "von innen nach außen"-Strategie auch durch explizite Übersetzung erledigt werden, wobei die Funktion allerdings als nicht Parameter-evaluierend gekennzeichnet werden muß, um eine doppelte Transformation der Parameter zu vermeiden. Die Regel lautet dann:

```
((NOTANY ,@body) (NOT ,(Translate-sexpression `(SOME ,@body))))
```

Gegen die "von außen nach innen"-Strategie spricht der Punkt (3), da nachträgliche Vereinfachungen damit nicht möglich sind. Daraus ergibt sich, daß eine Verbindung beider Strategien u.U. sinnvoll wäre. Da aber die beiden o.g. Punkte auch innerhalb der vorgegebenen Strategie befriedigend behandelt werden können, wurde darauf verzichtet, die Übersetzungsstrategie in diesem Aspekt zu modifizieren.

Ein Beispiel für die Forderung (2) ergibt sich bei der Übersetzung von GET. Diese wird normalerweise als kritisch gekennzeichnet, da in LM-Lisp der Wert und die Funktionsdefinition nicht Teil der Property-Liste eines Symbols sind. Wird bei GET als zweiter Parameter eine Konstante angegeben, so ist es möglich die jeweilige Zugriffsfunktion, nämlich

```
(FDEFINITION) und

(LOCF (SYMEVAL XX))
```

als Übersetzung zu erzeugen.

Die in (3) geforderte nachträgliche Optimierung ergibt sich aus dem Zusammenspiel mehrerer Übersetzungsregeln, z.B. bei der Übersetzung von

```
(COND ((OR (GET X 'EXPR)
           (GET X 'FEXPR)
           (GET X 'MACRO)
           (GET X 'SUBR)
           (GET X 'LSUBR)
           (GET X 'FSUBR))
       (PRIN X))
```

Diese Form wird durch die Regeln für GET in folgende Form übersetzt.

```
(COND ((OR (FPEFINEDP X)
           (FPEFINEDP X)
           .
           .
           (FPEFINEDP X))
       (PRIN X)))
```

Danach kann eine zyklische Vereinfachungsregel für OR angewandt werden, die alle identischen Funktionsaufrufe ohne Seiteneffekte, die direkt nacheinander aufgerufen werden, tilgt (vgl. Anhang D):

```
(COND ((FDEFINEDP X)
       (PRIN X)))
```

Da der erste Übersetzungsschritt keine exakte semantischen Äquivalenzen garantiert, wird der übersetzte Ausdruck mit einer Warnung markiert.

Die Bedeutung der Forderung (4) ergibt sich aus der Tatsache, daß über ein Programm Zusicherungen bekannt sein können, die sich nicht aus dem Programmtext schließen lassen, z.B. daß alle Mengenoperationen mit der EQ-Vergleichsoperation durchgeführt werden können, oder daß bei allen Indexoperationen nur positive Werte verwendet werden. Auch ist es möglich, daß eine möglichst schnelle Uebersetzung durchgeführt werden soll, bei der "Schönheitsaspekte" im Hintergrund stehen, daß z.B. ein Teil der Datei-I/O-Operationen emuliert werden soll. In ULM ist diese Parametrisierung durch globale Variable realisiert, die durch TEST-Attribute abgefragt werden.

Bei der Forderung (5) steht die Ökonomie der Notation im Vordergrund. Ein Beispiel ist die Behandlung der UCI-LISP-Prädikate, die statt T den Parameterwert zurückgeben, wie CONSP (s.o.), ZEROP, NUMBERP usw. Falls der äußere Kontext nur NIL oder non-NIL erwartet, z.B. in der ersten Form einer COND-Klausel, kann eine direkte Übersetzung erfolgen. Wird jedoch das Ergebnis benötigt, so muß in

```
(AND (<prädikat> <var>) <var>)
```

im Falle einer Variablen, oder in

```
(LET ((prdres <funktion>))
     (AND (<prädikat> prdres) prdres))
```

im Falle eines Ausdrucks übersetzt wurden. Da es sechs solcher Prädikate gibt, wären an sich achtzehn Regeln notwendig. Übersetzt man jedoch alle sechs Prädikate in eine gemeinsame Form, die von 3 Regeln behandelt werden kann, so benötigt man nur 9 Regeln.

Zu dem Punkt (5) gehört auch die in 3.1 geforderte Möglichkeit, eine allgemeine Regel vor allen anderen anzuwenden. Dieses Problem wurde dadurch gelöst, daß die in Abschnitt 2 geschilderte Kontrollstrategie so geändert wurde, daß bei der Suche nach einer anwendbaren Regel nur nach der Reihenfolge des Eintragens vorgegangen wird und nicht nach der inneren Struktur einer Regel. Damit wird auch die Einführung des PRIORITY-Attributes überflüssig.

Die Forderung (6) wurde schon in 3.1 als Kritikpunkt formuliert. Sie kann erfüllt werden, in dem man bei den Regeln zwischen terminalen Regeln, deren Anwendung eine erfolgreiche Übersetzung kennzeichnet und die Transformation beendet, und nicht-terminalen Regeln, die z.B. Zwischendarstellunger für eine Ausfaktorisierung von gemeinsamen Übersetzungsschritten erzeugen, unterscheidet. Es wurde deshalb ein neues Regelattribut RULE-TYPE eingeführt, das folgende Werte annehmen kann:

- TERMINAL, der Default-Wert,
- NON-TERMINAL und
- CYCLIC, für nicht-terminale Regeln, die mehrmals innerhalb eines Übersetzungszyklus angewendet werden können. (vgl. das REGIEM-Attribut in Abschnitt 2).

Die in (8) geforderte Formalisierung des ersten Laufes mithilfe von speziellen Regeln hat den Vorteil, daß die Übersetzung transparenter wird, und daß bei einem Übergang zu einem anderen Ausgangs-Dialekt nur Regeln und kein Programmcode geändert werden muß. Es wurde deshalb ein weiteres Regelattribut PASS eingeführt, daß als Wert die Zahlen 1, 2

oder die Liste (1 2) - für in beiden Läufen anwendbare Regeln - annehmen kann. Da beim 1. Lauf normalerweise nur die Strukturen auf der obersten Ebene interessant sind, die Funktionsdefinitionen, werden die Regeln nicht auf die Parameter sondern nur auf die S-Ausdrücke angewandt. Die Anwendung auf eingebettete S-Ausdrücke, z.B. in PROGNs, muß explizit durch Aufruf der Parameterübersetzungsfunktion "T-List" geleistet werden.

Zusammenfassend kann man feststellen, daß von den neun Forderungen vom Anfang des Abschnitts vier vollständig (1, 2, 4 und 7) und zwei zum Teil (3 und 5) schon durch den Übersetzungsalgorithmus des FRANZLATORs erfüllt werden. Allerdings werden diese Forderungen im Regelteil des FRANZLATORs kaum oder gar nicht berücksichtigt. Für die Forderungen 6, 8 und 9 und zum Teil 3 und 5 mußten zwei neue Attribute - RULE-TYPE und PASS - eingeführt und die Kontrollstrategie erheblich geändert werden.

3.4. LEISTUNGSFÄHIGKEIT UND UMFANG DER TRANSFORMATIONSREGELN

Der Ausgangsdialekt - das Hamburger UCI-LISP - umfaßt 480 Funktionen. Der größte Teil davon ist in [MEEHAN 79] beschrieben. Dazu kommen einige Erweiterungen, die aus MacLisp übernommen wurden [Online-Dokumentation] und einige Funktionen, die die Schnittstelle zu einer neuen Speicherverwaltung realisieren [Memo ANS-20]. In der Datenbasis mit den Informationen über alle UCI-LISP-Funktionen (Anhang C) sind neben den Informationen über Funktionswert und Parameteranzahl noch Kommentare eingetragen, die Angaben über die Übersetzbarkeit machen. Eine Zusammenfassung davon kann man der Tabelle 1 entnehmen.

Gruppe	Art der Funktionen	Anzahl	Prozent
A	gleich lautende, abbildbare Funktionen	115	24.0%
B	verschieden lautende, abbildbare Funktionen	66	13.8%
C	ignorierbare Funktionen	23	4.8%
D	übernommene Funktionen	22	4.8%
E	durch Regeln übersetzbare Funktionen	72	15.0%
F	durch Regeln übersetzbare Funktionen, bei denen Sonderfälle nicht behandelt werden können	16	3.3%
G	Funktionen, die durch Schalter gesteuert in eine Kompatibilitätsfunktion übersetzt werden können	28	5.6%
H	nicht übersetzbare Funktionen	138	28.7%
	Gesamt =	480	100.0%

Tabelle 1: Übersetzbarkeit der UCI-LISP Funktionen nach LM-Lisp

In der Gruppe A sind alle UCI-LISP-Funktionen zusammengefaßt, für die es gleichlautende LM-Lisp-Funktionen gibt, die mindestens die Funktionalität der UCI-LISP-Funktionen besitzen, wie z.B. EQ, CAR (zum Problem eines Parameters bei CAR und CDR s. Anhang B) und BOUNDP. Die Gruppe B umfaßt die Funktionen, für die anderslautende LM-Lisp-Funktionen existieren, die aber die Semantik der UCI-LISP-Funktionen realisieren können, z.B. für *APPEND APPEND, für -I 1- und für GT >. Es lassen sich also, wie man der Tabelle entnehmen kann, 181 aller Funktionen durch einfache Atom-Transformationsregeln übersetzen.

Bei der Gruppe C handelt es sich um Funktionen, die sich auf die Systemumgebung beziehen, deren Wirkung aber ignoriert werden kann. Beispiele für solche Funktionen sind GC, GCGAG, FREE, ERRCH. Der Funktionsaufruf wird bei der Übersetzung durch das Funktionsergebnis ersetzt, u.U. werden noch Parameter evaluiert, falls es sich dabei um Funktionsaufrufe handelt. Das Ignorieren der Funktionen wird durch den Schalter IGNORE-OPTION gesteuert.

In der Gruppe D sind die Funktionen zusammengefaßt, für deren Funktionalität es keine Entsprechungen in LM-Lisp gibt, und die auch nicht durch eine einfache Transformationsregel behandelt werden können, z.B. NEWSYM, TCONC, ATTACH und ENQUOTE. Diese Funktionen wurden deshalb in der LM-Lisp-Umgebung reimplementiert und werden durch einfache Atom-Regeln übersetzt.

Dazu im Gegensatz stehen die Funktionen der Gruppe G. Diese haben eine Entsprechung in LM-Lisp, aber ihre Parameterkonventionen oder Annahmen über interne Datenstrukturen weichen entschieden von UCI-LISP ab. Um eine schnelle und sichere Übersetzung zu gewährleisten, wurde ein Satz von Kompatibilitätsfunktionen geschrieben, zu deren Benutzung aber bei einer sorgfältigen Portierung nicht geraten wird. Aus diesem Grund wird die Übersetzung dieser Gruppe von Funktionen durch Schalter gesteuert. Zu diesen Funktionen gehören z.B. GET und PUT (da in LM-Lisp die Funktionsdefinition und der Wert eines Symbols nicht auf der Property-Liste sondern in speziellen Zellen stehen), INTERSECTION (da in LM-Lisp die Mengenzugehörigkeit mit EQ und nicht mit MEMBFN geprüft wird) und einige Platten-E/A-Funktionen wie INC, INCH, DSKIN.

Die Funktionen der Gruppen E, F und zum Teil auch G sind die, bei denen der Vorteil der regelgesteuerten Übersetzung voll zum Tragen kommt, da es bei diesen Funktionen möglich ist, mithilfe jeweils einer oder mehrerer Regeln ein entsprechendes Zielsprachkonstrukt zu erzeugen. Während die Funktionen der Gruppe E in jedem Fall übersetzt werden können, können in der Gruppe F einige Sonderfälle nicht behandelt werden, z.B. darf das 2. Argument bei EVAL kein BCP (binding context pointer) sein und die Funktion UNBOUND kann nur in einem SET/SETQ-Kontext übersetzt werden. Tritt ein nicht-übersetzbarer Sonderfall auf, wird der Funktionsaufruf mit einer Warnung markiert.

Überraschend ist, daß die Gruppe H, die nicht-übersetzbaren Funktionen, fast 1/3 aller UCI-LISP-Funktionen umfaßt. Damit scheint der in Abschnitt 1 erhobene Anspruch der 95%-igen Übersetzung nicht erfüllbar zu sein. Tatsächlich handelt es sich jedoch bei diesen Funktionen um in "normalen" LISP-Programmen sehr selten benutzte Sprachteile, die in größeren Systemen nicht mehr als 1-2% ausmachen dürften. Dies geht aus den Aufgabengebieten dieser Funktionen hervor, die sich folgendermaßen charakterisieren lassen:

- Toplevel-Funktionen, d.h. Funktionen, die normalerweise nur interaktiv und nicht aus einem Programm heraus aufgerufen werden, wie z.B. EDITF, TRACE und BREAK.

- Kelleroperationen - Funktionen, die den Keller der zu evaluierenden S-Ausdrücke und Variablenbindungen inspizieren. Diese Funktionen werden i.a. nur in Systempaketen wie z.B. TRACE benötigt.

- Maschinen-Interface-Funktionen, die einzelne Speicherstellen inspizieren und ändern können, sowie die Kommunikation mit Assemblerprogrammen ermöglichen.

- Meßfunktionen, die Aspekte wie Zeit- und Speicherplatzverbrauch messen.

- Datei-E/A-Funktionen, die Dateien löschen, umbenennen, Directories lesen usw. Diese werden im Gegensatz zu den mehr elementaren E/A-Funktionen wie INPUT, OUTPUT usw. nicht durch Kompatibilitätsfunktionen unterstützt.

- Editorfunktionen zum programmgesteuerten Suchen, Substituieren usw.

- Prettyprint-Kommandos, die die Ausgabe von in PPL-Listen spezifizierten LISP-Objekten steuern (vor allem im Zusammenhang mit DSKOUT).

- Scanner-Funktionen, die die Zeichensyntaxtabelle ändern oder Auskünfte über Syntaxeigenschaften von Zeichen geben. Der Aufbau der Zeichensyntaxtabelle in UCI-LISP unterscheidet sich so erheblich von LM-Lisp, daß hier eine Handübersetzung notwendig ist (vgl. auch Abschnitt 3.2).

Nach dieser Charakterisierung der Leistungsfähigkeit nun noch ein paar Daten zum Umfang des Regelwerkes. Es existieren zur Zeit 415 Transformationsregeln, von denen 115+22 die Form x -> x, und 66 die Form x -> y besitzen (vgl. Tabelle 1). Zieht man außerdem die Regeln zur Behandlung der Funktionen aus Gruppe C, den ignorierbaren Funktionen ab, dann bleiben 183 Regeln zur Übersetzung der Funktionen aus den Gruppen E, F und G übrig. Dies entspricht einem Verhältnis von durchschnittlich 1.6 Regeln pro Funktion, was auf eine extensive Sonderfallbehandlung schließen läßt.

Bei einem Vergleich mit dem FRANZLATOR fällt auf (vgl. Tabelle 2), daß ULM in bedeutend höherem Maße den Ansprüchen von Vollständigkeit und möglichst geringer Emulation entspricht, wobei die Vollständigkeit bei INTERLISP aufgrund der großen Anzahl von Systemfunktionen natürlich auch nur schwer erreichbar ist. Die Absolutzahl der behandelten Funktionen ist jedoch in beiden Systemen ungefähr gleich (ca. 400).

Charakteristikum	FRANZLATOR	ULM
vorhandene Systemfunktionen im Ausgangsdialekt	1314	480
behandelte Systemfunktionen	ca. 400	342
Vollständigkeit	ca. 30%	71%
Regelanzahl	38	415
übernommene Funktionen und Kompatibilitätsfunktionen	ca. 250	50
Verhältnis von Regeln zu regelübersetzten Funktionen	1.07	1.6

Tabelle 2: Vergleich des FRANZLATORs mit ULM anhand einiger charakteristischer Größen

4. ERFAHRUNGEN MIT ULM

Bisher wurden mit ULM drei Programmsysteme übersetzt, das Micro-FIT-System, ein Satz von Benchmark-Funktionen und eine Komponente des HAM-ANS Systems, VERBALIZE. Die generelle Erfahrung dabei war, daß relativ wenige nachträgliche Änderungen nach der Übersetzung notwendig waren. Auch Erweiterungen der Transformationsregeln waren kaum erforderlich. Die wenigen notwendigen Erweiterungen konnten - aufgrund der guten Integration von ULM in die LM-Umgebung - interaktiv und inkrementell durchgeführt werden. Innerhalb des LM-Editors ZMACS trägt man die neue Regel textuell in die Regeldatei ein und bewirkt dann durch Betätigung zweier Tasten die Übersetzung der Regel in eine Match-Funktion und die Aufnahme in die Regeltabelle (vgl. Abschnitt 2). Ist die Regel reihenfolgesensitiv, soll eine bestehende Regel geändert oder gelöscht werden, so muß die gesamte Regeldatei neu eingelesen werden, dies dauert zwar 2-3 min., erfordert aber auch nur einen Editorbefehl.

4.1. ÜBERSETZUNG VON MICRO-FIT

Das einfachste und vom Umfang her kleinste (2 DIN A4 Seiten) System war das Micro-Fit System MUFITE, das eine Prinzip-Implementation der KI-Sprache FIT [BOLEY 83] darstellt. Die einzige Schwierigkeit war die Übersetzung eines EVAL-Aufrufs mit einem 2. Parameter, einer A-Liste. Da LM-Lisp EVAL nur einen Parameter hat, war keine einfache Übersetzung möglich. Vermutlich hätte man den Algorithmus mithilfe von Macros reimplementieren können, wobei dann der Aufbau der Variablenbindungen aus der A-Liste schon zur Expansionszeit vorbereitet wird und ein expliziter EVAL-Aufruf überflüssig wird. Solche Restrukturierungen sind jedoch regelgesteuert nicht zu leisten. Stattdessen wurde eine Funktion EVAL-ENVIRONMENT implementiert, die eine A-Liste als zweiten Parameter akzeptiert und dynamisch die zugehörigen Bindungen erzeugt. Die gesamte Übertragung dauerte etwa 1/2 Stunde und erforderte nicht mehr als die geschilderte Definition der Funktion EVAL-ENVIRONMENT.

4.2. ÜBERSETZUNG EINES SATZES VON BENCHMARK-FUNKTIONEN

Das zweite System, das mit ULM übersetzt wurde, war ein Satz von Benchmark-Funktionen, die für einen Zeitvergleich von LISP- und PASCAL-Programmen benutzt worden waren [NEBEL 83]. Dies System war etwas umfangreicher (9 DIN A4 Seiten), die verwendeten Sprachkonstrukte jedoch relativ einfach. Bis auf Änderungen in der eigentlichen Meßfunktion, in der einige E/A-Aufrufe und der Zeitfunktionsaufruf geändert werden mußten, und in der E/A-Benchmark-Funktion, waren keine zusätzlichen "Hand-Übersetzungen" erforderlich.

Nach erfolgreicher Übersetzung erbrachte die Ausführung der Benchmark-Funktionen die in Tabelle 3 aufgelisteten Ergebnisse. Zu beachten ist hierbei, daß es sich bei der DEC-10 und der VAX 11 um reine CPU-Zeiten handelt, während auf der Lispmaschine (einer Symbolics 3600) die Antwortzeiten gemessen wurden. Dies ist zulässig, da es einerseits keine Funktion für die CPU-Zeitmessung gibt und andererseits auf der LM normalerweise die CPU- gleich der Antwortzeit ist. Problematisch ist diese Annahme allerdings bei Plattenzugriffen, wie man deutlich bei den Zeiten für die Copy-Char-Funktion erkennt.

Als Ergebnis kann man der Tabelle 3 entnehmen, daß compilierte Funktionen auf der LM im Schnitt 2-3 mal schneller als auf der VAX 11/780 oder der DEC-10 sind! Dabei sind auf den Timesharing-Rechnern die Zeiten zugrunde gelegt, die beim Ausschöpfen aller Optimierungsoptionen zu erreichen sind, also NOCALL auf der DEC-10 und translink=t sowie die Verwendung spezieller FIXNUM-Operationen auf der VAX-11. Überraschend ist, daß die LM im interpretativen Modus etwa 1.5 mal langsamer ist als die Timesharing-Rechner. Dies spielt allerdings keine so große Rolle, denn man arbeitet auf der LM normalerweise mit compilierten Programmen, da die Benutzerschnittstelle zum Compiler sehr komfortabel ist und man schon vor der Ausführung des Programms auf kleinere Fehler hingewiesen wird (vgl. [Symbolics 83a], S. 62).

Test-Programme		UCI-LISP auf DECsystem-10		FranzLisp auf VAX-11/780		LM-Lisp auf Symbolics 3600	
		interp.	comp.	interp.	comp.	interp.	comp.
SORT(20)	nicht dest.	962.0	49.0	956.3	39.9	1450.0	46.7
	destruktiv	1063.0	36.0	909.6	30.0	1290.0	10.0
SORT(40)	nicht dest.	4015.0	219.0	3921.7	156.6	5726.7	103.3
	destruktiv	3891.0	163.0	3731.8	119.9	5336.7	50.0
SORT(60)	nicht dest.	6679.0	352.0	6674.0	263.2	9510.0	153.3
	destruktiv	6733.0	278.0	6274.2	203.3	8990.0	76.6
SORT(100)	nicht dest.	13472.0	642.0	12888.2	509.8	18593.3	296.7
	destruktiv	13345.0	573.0	12355.1	399.8	17616.7	160.0
REVERSE-A	nicht dest.	104.0	1.8	119.1	2.5	145.0	4.1
	destruktiv	113.4	1.8	126.6	0.8	145.0	0.1
REVERSE-B	nicht dest.	96.6	3.3	103.3	2.5	161.7	2.5
	destruktiv	90.6	1.8	92.5	0.8	151.7	0.8
REVERSE-C	nicht dest.	158.6	59.4	202.4	67.5	214.2	54.2
	destruktiv	107.0	18.7	115.8	10.8	145.0	12.5
ACKERMANN(3,3)		4122.6	121.1	3844.3	67.5	6555.8	25.0
ACKERMANN(3,4)		17125.8	498.8	16425.1	286.6	29038.3	200.0
QUICKSORT(20)		417	38	666.4	69.8	649.9	16.7
QUICKSORT(40)		868	80	1466.1	133.3	1350.0	33.3
QUICKSORT(60)		1261	103	2182.5	199.9	1983.3	66.7
QUICKSORT(100)		2272	162	4098.4	383.1	3849.9	133.3
CopyChar(10)		67.8	156.7	257.4	100.8	1325.8	1188.3
CopyChar(10000)		11791.0	2024.8	36442.1	3838.5	27045.0	9193.3
KNAPSACK		16578.2	665.8	25634.7	537.3	20505.8	234.2
WANG	de Morgan	262.5	15.0	903.8	17.5	879.2	11.7
	Composition	235.5	16.4	818.0	13.3	752.5	7.5
	de Mo.<=>not Comp.	194.2	14.9	716.4	14.2	630.8	9.2

Tabelle 3: Laufzeitergebnisse der Benchmark-Funktionen

4.3. ÜBERSETZUNG DER HAM-ANS-KOMPONENTE VERBALIZE

Die Übersetzung einer HAM-ANS Komponente stellte die Feuerprobe für das ULM-System dar, da im Gegensatz zu den beiden eben beschriebenen Systemen in HAM-ANS die Idiosynkrasien von UCI-LISP stark ausgenutzt werden. Wir haben als exemplarischen Testfall die Komponente VERBALIZE ausgewählt, die die Transformation von SURF-Ausdrücken in präterminale Ketten leistet (vgl. [BUSEMANN 84]).

4.3.1. GRUNDLAGEN FÜR DIE ÜBERTRAGUNG VON VERBALIZE

Als Voraussetzung für die Portierung von VERBALIZE war die Portierung des FUZZY-Systems und des EBNF-Interpreters notwendig, eines Systems zur Analyse und Synthese von Ausdrücken von Sprachen, die durch einen BNF-ähnlichen Formalismus definiert sind. Beide Systeme wurden vor der Entwicklung des ULM-Systems per Hand portiert, einmal um sich mit der Lispmaschine vertraut zu machen, und zum anderen um Erfahrungen mit der Übersetzung von UCI-LISP nach LM-Lisp zu sammeln.

Die Portierung des FUZZY-Interpreters[1] konnte auf die Erfahrungen aufbauen, die bei der Portierung von UCI-LISP nach FranzLisp gewonnen wurden. Neben dem Problem, das sich im Zusammenhang mit rückverfolgbaren LISP-Variablen ergab (vgl. [Memo ANS-17], S. 4) und befriedigend gelöst werden konnte, warf die Behandlung der CATCH-Funktion Schwierigkeiten auf. Die "SELECTQ-Variante" des CATCH in UCI-LISP wird in einen verschachtelten CATCH-CONTINUATION-Aufruf übersetzt, ein CATCH, das zwischen erfolgtem THROW und normaler Auswertung unterscheidet. Die "catch all"-Variante, bei der man NIL als Sprungmarke angibt, wird in LM-Lisp auf der Symbolics 3600 leider nicht unterstützt [Symbolics 83b]. Deshalb war eine Analyse aller CATCH- und THROW-Aufrufe notwendig, um die möglichen Sprungmarken für diesen Fall zu bestimmen.

Die Integration in die Programmierumgebung wurde im Gegensatz zur Franz-LISP Übertragung vollständig geleistet. Dazu gehören
- die Anpassung der Read- und Printmacros,
- Anschluß an den TRACER und den EDITOR und
- Anpassung der Fehlerbehandlung.

Um eine möglichst gute Adaption an LM-LISP zu erreichen, wurde versucht, das FUZZY-System in ein eigenes "Package" - einen privaten Namensraum - zu laden. Dabei wurden zwei Schwachstellen des "Package"-Mechanismus im LM-Lisp deutlich:

- Wenn man innerhalb eines Packages ein neues Symbol einführt, z.B. durch Definition einer Funktion, und dieses Symbol schon global bekannt ist, so wird das globale Symbol benutzt. D.h. Namenskonflikte, die man vermeiden wollte, treten doch auf. Man hat jedoch die Möglichkeit Package-lokale Namen mit einer speziellen Funktion einzuführen.

- Es gibt keine vernünftige Möglichkeit Symbole zu exportieren, d.h. sie global zugreifbar zu machen, so daß sie ohne Package-Präfix ansprechbar sind. Es gibt zwar eine Funktion GLOBALIZE, die das leistet, die aber zusätzlich alle Werte von gleichlautenden

---

[1] Die Portierung von FUZZY auf die LM wurde von Th. Christaller durchgeführt.

Symbolen aus allen anderen Packages versucht unter dem globalen Symbol abzulegen.

Bei der Übersetzung des EBNF-Interpreters ergaben sich keine Schwierigkeiten bei der Transformation einzelner Funktionen. Hier lagen die Probleme vielmehr darin, daß in UCI-LISP eine andere Strategie bei der Verwaltung von Datenbasen als in LM-Lisp verfolgt wird. Während man in UCI-LISP während einer Sitzung eine als Textdatei vorliegende Datenbasis einmal einliest, dann innerhalb des Arbeitsspeichers mithilfe des Strukturediters modifiziert und am Ende der Sitzung wieder ausliest, wird in LM-Lisp immer nur die Textdatei mithilfe eines Texteditors, der aber auch struktursensitiv ist, modifiziert und nach jeder Änderung neu eingelesen (vgl. Abb. 3). Da nun im UCI-LISP-EBNF-Interpreter alle Verwaltungsdaten bei der Ausgabe einer EBNF-Datenbasis erzeugt werden und dieses in LM-Lisp wegfällt, mußten diese Aufgaben von den Einlesefunktionen übernommen werden. D.h. es war einiges an Restrukturierung notwendig. Dies kann als Beispiel dafür dienen, was bei einer sorgfältigen Portierung an zusätzlicher Anpassung notwendig ist, auch wenn man ein Übersetzungsystem wie ULM benutzt.

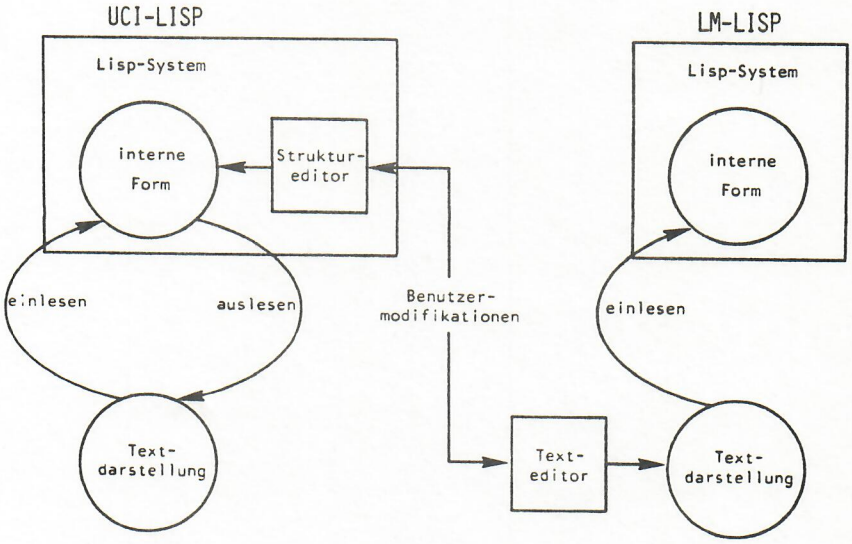

Abbildung 3: Modifikation von Datenbasen
in LM-Lisp und UCI-LISP

4.3.2. *ERGEBNISSE DER ÜBERSETZUNG VON VERBALIZE*

Die Übersetzung von VERBALIZE[2] bereitete keine großen Schwierigkeiten. Probleme traten lediglich im Zusammenhang mit "aufgetrennten" Bezeichnern (vgl. 3.2), drei fehlenden Funktionen der HAM-ANS-Funktionsbiblothek und fehlenden Deklarationen für die Compilation auf. Der letztgenannte Punkt hängt damit zusammen, daß aus Gründen der Flexibilität und wegen Benutzung des ATTACH-Pakets zum Nachladen von Funktionen [JAMESON 80] HAM-ANS zum größten Teil interpretiert ausgeführt wird.

---

[2] durchgeführt von S. Busemann und Th. Christaller

Eine Funktion der HAM-ANS-Biblothek war allerdings nicht portierbar. Sie mußte reimplementiert werden. Der geneigte Leser wird sofort erkennen, woran das liegt, wenn er die folgende Funktionsdefinition aufmerksam studiert:

```
(DF ZIEHEN (L)
  (PROG (THIS REST)
    (SETQ REST (OR (CDR L) (CDR (RPLACD L (CAR L)))))
    (SETQ THIS (POP REST))
    (RPLACD L REST)
    (RETURN THIS)))
```

Ein Aufruf dieser Funktion, der folgende Gestalt hat:

```
(ZIEHEN (A B C))
```

würde nacheinander A,B,C,A,B... als Ergebnis liefern. Die zugrundeliegende Annahme ist, daß der formale Parameter einer FEXPR direkt auf die aktuelle Parameterliste zeigt, und diese destruktiv modifizierbar ist. Diese Möglichkeit ist in UCI-LISP gegeben, wenn sie auch aus gutem Grund nicht dokumentiert ist. In compiliertem LM-Lisp wird die Parameterübergabe jedoch anders realisiert, so daß statt der Parameterliste eine globale Variable als "Gedächtnis" benutzt werden muß - eine meiner Meinung nach durchsichtigere Lösung.

Die nach der Übertragung durchgeführten Zeitvergleiche brachten überraschende Resultate. Die durch die Benchmark-Funktionen geweckten Erwartungen wurden nicht erfüllt. Zu erwarten war eine Steigerung um den Faktor 10 bis 30 beim Übergang vom interpretierten Programm auf der DEC-10[3] zum compilierten Programm auf der Symbolics.

Tatsächlich war das Verhältnis vom compilierten Programm auf der Symbolics zum interpretierten Programm auf der Symbolics zum interpretierten Programm auf der DEC-10 wie 1:6:3.5. Unter der Annahme, daß der Compilationsgewinn bei den interpretierten Komponenten ebenso hoch wie bei den Benchmark-Funktionen ist, also ca. 40, folgt, daß ein Großteil der CPU-Zeit vom FUZZY-System verbraucht wird.

Sei f[x] die CPU-Zeit, die FUZZY verbraucht,
v[x] die CPU-Zeit, die VERBALIZE und EBNF verbrauchen, und
g[x] die gesamte verbrauchte CPU-Zeit,

wobei
x=i interpretiert und
x=c compiliert bedeutet,

dann folgt aus

(1) f[c] + v[i] = g[i]
(2) f[c] + v[c] = g[c]
(3) v[i] = 40v[c]
(4) g[i] = 6g[c]

daß 7/8 g[c] = f[c] ist.

---

[3] bestehend aus dem compilierten FUZZY-System und den interpretierten Komponenten EBNF-Interpreter und VERBALIZE.

5. ZUSAMMENFASSUNG UND AUSBLICK

Das ULM-System, das aus dem FRANZLATOR hervorgegangen ist, ist ein effizientes und komfortables Werkzeug für die Portierung größerer LISP-Programme, wie die Erfahrung bei der Übersetzung dreier Programme gezeigt hat. 70% aller UCI-LISP-Funktionen könnnen automatisch übersetzt werden und bei den restlichen 30% handelt es sich hauptsächlich um Funktionen, die etwas mit der UCI-LISP-Systemumgebung zu tun haben, und deshalb in normalen Benutzerprogrammen selten vorkommen.

Gegenüber dem FRANZLATOR sind vor allem folgende Verbesserungen zu erwähnen: Die Modifikation des Regelinterpreters, die eine korrektere und komfortablere Formulierung der Regeln erlauben; die Ausnutzung der regelgesteuerten Übersetzung für einfache Programmtransformationen und der um eine Grössenordnung umfangreichere Regelsatz, der zu einer vollständigeren Abdeckung des Ausgangsdialekts beiträgt und die Definition von Kompatibilitätsfunktionen überflüssig macht.

Als interessantes Zusatzresultat fielen noch Laufzeitverleichszahlen zwischen UCI-LISP-Programmen auf der DEC-10, FranzLisp-Programmen auf der VAX 11/780 und LM-Lisp-Programmen auf der Symbolics 3600 ab: Während die Symbolics 3600 im compilierten Modus zwei bis dreimal schneller als die DEC-10 und die VAX 11 ist, ist die Symbolics 3600 im interpretativen Modus um den Faktor 1.5 langsamer. Dies liegt vermutlich daran, daß die Software im Gegensatz zu den Mikroprogrammen noch nicht optimiert wurde (vgl. [Symbolics 83b], S.1). Ein weiteres überraschendes Ergebnis trat bei den Laufzeitvergleichen zwischen der interpretierten VERBALIZE-Komponente auf der DEC-10 und der gleichen Komponente compiliert auf der Symbolics 3600 auf. Die Laufzeitgewinne lagen weit unterhalb dessen, was die Benchmark-Funktionen erwarten ließen. Dies hängt damit zusammen, daß der Großteil der Laufzeit auf das auf beiden Rechnern compiliert ausgeführte FUZZY-System entfielen und deshalb der Compilationsgewinn bedeutend kleiner ausfiel.

In Zukunft müßten insbesondere folgende Aufgabengebiete im Bereich der Interdialekt-Übersetzer bearbeitet werden:

- Die Erstellung eines Übersetzers in einem genügend kleinen Subset von LISP, so daß dieser leicht in jeden Dialekt übersetzt werden kann. Die Hauptschwierigkeit wird dabei vermutlich bei der Behandlung der Zeichensyntax liegen;

- die Berücksichtigung eines größeren Kontextes als dem des gerade zu übersetzenden S-Ausdrucks, um z.B. in unserem Fall die Print- und Readmacros automatisch zu übersetzen;

- Codeanalyse oder Metaevaluation zur Unterstützung der automatischen Übersetzung von Makros und FEXPRs;

- das Einbeziehen von speziellen Deklarationen für den Interdialektübersetzer, ähnlich denen, die für den LISP-Compiler notwendig sind, falls nacheinander mehrere Versionen des gleichen Systems übersetzt werden sollen;

- und schließlich die Unterstützung der Posttranslationsphase durch z.B. die Generierung von Testdaten.

LITERATUR

[BOLEY 83] Boley, H.: From Pattern-Directed to Adapter-Driven Computation via Function-Applying Matching; In: Kupka, I. (ed.), GI - 13. Jahrestagung 1983, Springer, Heidelberg 1983, 86-100.

[BUSEMANN 84] Busemann, S.: Topicalization and Pronominalization, Extending a Natural Language Generation System; To appear in: Proc. 6th ECAI, Pisa 1983.

[FININ 82] Finin, T.: Translating KL-ONE from Interlisp to FranzLisp; In: [SCHMOLZE & BRACHMANN 82].

[FODERADO 82] Foderado, J.K.: The FranzLisp Manual, Univ. of California, Berkley (CA), 1982.

[JAMESON 80] Jameson, A.: ATTACH: A Package for accessing LISP Programs and Data from Disk; Universität Hamburg, Projektgruppe Simulation von Sprachverstehen, Memo Nr. 7, Hamburg, März 1980.

[LAMPING & KING 82] Lamping, J., King, J.J.: IZZI - A Translator from INTERLISP to ZetaLisp; Hewlett Packard, Computer Science Lab., Technical Note CSL-82-10, Palo Alto (CA), März 1982.

[LE FAIVRE 78] Le Faivre, R.A.: FUZZY Reference Manual; Rutgers Univ., Computer Science Dep., New Brunswick (NJ), 1978.

[MEEHAN 79] Meehan, J.R.: The New UCI-LISP Manual; Lawrence Erlbaum, Hillsdale (NJ), 1979.

[Memo ...] Memos der Forschungsstelle für Informationswissenschaft und Künstliche Intelligenz (s. Literaturliste am Ende).

[MOON 74] Moon, D.A.: MacLisp Reference Manual, M.I.T., Project MAC, Cambridge (MA) 1974.

[NEBEL 83] Nebel, B.: Ist LISP eine "langsame" Sprache?; In: Neumann, B. (ed): GWAI-83. 7th German Workshop on Artificial Intelligence, Springer, Heidelberg 1983, 21-30.

[Online-Dokumentation] ist auf der Hamburger DEC-10 unter AIL:LISP?.NEW und AIL:NEWFUN.TXT verfügbar.

[POST 43] Post, E.: Formal Reductions of the General Combinatorial Problem; In: American Journal of Mathematics, Vol. 65, 1943, 197-268.

[SCHMOLZE & BRACHMAN 82] Schmolze, J.G., Brachman,R.J.(ed): Proceedings of the 1981 KL-ONE Workshop, Bolt Beranek and Newman Inc., Report No. 4842, Cambridge (MA), Juni 1982.

[STEELE 82] Steele, G.L.: An Overview of Common Lisp, In: Conf. Record of the 1982 ACM Symposium on LISP and Functional Programming, Pittsburgh (PA), 1982, 163-162.

[Symbolics 83a] Symbolics Inc., Program Development Tools and Techniques; Symbolics Inc., Cambridge (MA), August 1983.

[Symbolics 83b] Symbolics Inc., Notes on the 3600 for LM-2 Users; Symbolics Inc., Cambridge (MA), September 1983.

[TEITELMAN 78] Teitelman, W., INTERLISP Reference Manual; Xerox PARC, Palo Alto (CA), 1978.

[WEINREB & MOON 83] Weinreb, D., Moon, D., Lisp Machine Manual, Symbolics Inc., Cambridge (MA), Juli 1983.

ANHANG A: BENUTZUNG DES ULM-SYSTEMS

Alle notwendigen Dateien sind z.Zt. auf der Maschnine "Hans" (Hamburg application-oriented nice slave) unter der Directory ">berni>ulm" zusammengefaßt. Die Systemdefinition des Systems steht in der Datei ulm.lisp. Um eine Übersetzung durchzuführen, gehe man folgendermaßen vor:

1) Man transferiere die zu übersetzenden Dateien auf die LM. In Hamburg kann man das mit Hilfe des Modem-Link Programms und einem Telefonkoppler durchführen.

2) Die Systemdefinition muß geladen werden:
(load ">berni>ulm>ulm").

3) Das System muß geladen und eventuelle Änderungen müssen compiliert werden:
(make-system 'ulm ':compile).
Alternativ kann man auch ein eigenes System laden, in dem ULM als "component-system" aufgeführt ist.

4) Die eigentliche Übersetzung wird durch Aufruf der Funktion TRANSLATE-FILES initiiert. TANSLATE-FILES hat die Parameter 'file-list', 'source-directory' und 'target-directory'. Ein Aufruf, um FOO.LSP und BAR.LSP, die in der Directory >A stehen, zu übersetzen und das Ergebnis in der Directory >B abzulegen, müßte lauten: (TRANSLATE-FILES '("FOO.LSP" "BAR.LSP") ">A>" ">B>"). Die übersetzten Dateien erhalten die Extension "LISP", die zugehörigen Warnungen "NOTICE".

5) Für die manuelle Nachbearbeitung schaffe man sich am besten zwei Fenster in ZMACS, das obere etwas kleiner, und gehe dann parallel die LISP- und die NOTICE-Datei durch, ähnlich wie bei der Bearbeitung von Compiler-Meldungen.

6) Die Dateien BASICFNS mit den übernommenen Funktionen und ggfs. UCI-COMP müssen bei der Ausführung und Übersetzung von portierten Programmen dazu geladen werden.

ANHANG B: PROBLEMATISCHE FUNKTIONEN IN UCI-LISP

In diesem Anhang sind alle Funktionen aufgelistet, die Probleme bei der Übersetzung bereiten nebst den Ursachen dafür. Nicht mit aufgeführt sind die nicht-übersetzbaren Funktionen, die in Anhang C mit einem x gekennzeichnet sind.

ADDPROP, GET, GETL, PUT, PUTLIST, PUTROP, REMPROP, REMPROPS
Bei allen Funktionen, die auf Property-Listen operieren, besteht das Problem darin, daß in LM-Lisp für den Wert und die Funktionsdefinition eines Symbols extra Zellen bereitgestellt werden im Gegensatz zu UCI-LISP, das entsprechend der LISP 1.5-Tradition diese Informationen auf der Property-Liste ablegt. Wenn als Property Konstanten angegeben werden, ist eine zuverlässige Übersetzung möglich, sonst muß man davon ausgehen, daß weder Funktions- noch Werteeigenschaften über diese Funktionen angesprochen wurden oder aber in die UCI-Kompatibilitätsfunktionen übersetzen (gesteuert durch die Option PROPERTY-LIST-OK-OPTION).

ANTHCHAR, NTHCHAR, SUBSTRING
Bei diesen Funktionen gibt es zwei Probleme mit den zulässigen Datentypen für die Parameter. In UCI-LISP sind negative Indizes erlaubt, die Indizierung vom rechten Ende her bedeuten, während in LM-Lisp nur positive Indizes zulässig sind. Ist man sicher, daß nur positive Indizes benutzt werden, kann eine direkte Übersetzung erfolgen, sonst ist die Benutzung der entsprechenden UCI-Kompatibilitätsfunktionen erforderlich (gesteuert durch INDEX-OK-OPTION). Das zweite Problem betrifft die automatische Typkonversion des String-Parameters: In UCI-LISP werden Zahlen in den entsprechenden String bestehend aus den Ziffern, die die Zahl ausmacht, konvertiert, während in LM-Lisp ein Buchstabe erzeugt wird. Ist man sicher, daß nie Zahlen als String-Parameter auftauchen, kann der Schalter NO-NUMBER-STRING-OPTION auf T gesetzt werden.

APPLY, EVAL
Falls ein drittes bzw. zweites Argument angegeben wird, darf es sich dabei nur um eine A-Liste handeln, nicht um einen BCP, da dieses Konstrukt in LM-Lisp weder bekannt noch emulierbar ist.

ARRAY
Da das ARRAY-Konstrukt nur teilweise in LM-Lisp unterstützt wird, können die Angabe einer unteren Grenze und Angabe eines anderen Wertebereichs als der allgemeine LISP-Datentyp nicht behandelt werden. Für eine allgemeinere Behandlung dieses Falls könnte man die LM-Lisp Funktionen AREF, ASET und MAKE-ARRAY benutzen.

CAR, CDR und Verwandte:
Die Semantik von CAR, CDR und Verwandten unterscheidet sich zwischen UCI-LISP und LM-Lisp nur bei der Behandlung von Atomen. Während in diesem Fall in UCI-LISP bei CAR eine Atom-Kennung und bei CDR die Property-Liste zurückgegeben wird, wird in LM-Lisp ein Fehler ausgelöst. Da es sich bei Zugriffen mit CAR und CDR auf Atome zumeist um Fehler handelt, wurde darauf verzichtet, diesen Fall zu behandlen. Es ist natürlich möglich, in LM-Lisp eine Fehlerbehandlungsfunktion zu schreiben, die diese Fälle abfängt.

CHRCT, CHRPOS
Da die aktuelle Druckposition kein Attribut einer beliebigen Datei ist, sondern nur beim Terminal-E/A-Strom Buch über die Druckposition geführt wird, werden die o.g. Funktionen nur für diesen Fall übersetzt. Falls die Option DISK-IO-OPTION aktiviert ist, wird jedoch in Kompatibilitätsfunktionen übersetzt, die einen Wert liefern, wenn sie innerhalb von GRIND-TOP-LEVEL aufgerufen werden.

CSYM
In UCI-LISP kann man mit CSYM einen drei-buchstabigen Präfix für GENSYM-Symbole setzen. In LM-Lisp kann dagegen nur ein ein-buchstabiger Präfix mit Hilfe von GENSYM gesetzt werden.

DSKIN, DSKLOG, INC, INCH, INPUT, LOGOUT, OUTC, OUTCH, OUTPUT
Diese Funktionen werden nur dann in Kompatibilitätsfunktionen übersetzt, wenn der Schalter DISK-IO-OPTION gesetzt ist. Außerdem werden die Directory-Namen ignoriert, nur die Datei-Namen sind signifikant.

ENTER, INTERSECTION, UNION
In UCI-LISP hängt die Mengenzugehörigkeit von der Funktion ab, die der Wert von MEMBFN (default EQUAL) ist, während in LM-Lisp immer mit EQ getestet wird. Ist man sicher, daß der EQ-Test ausreicht, so kann man die Option EQ-SETS-OPTION ausnutzen, ansonsten wird in die UCI-LISP-Kompatibilitätsfunktionen übersetzt.

ERRSET, ERR
Wenn mit Hilfe von ERR oder anders ein Fehler ausgelöst wird, so gibt in LM-Lisp ERRSET immer NIL zurück, während in UCI-LISP das atomare Argument von ERR oder EOF bei end-of-file Fehlern zurückgegeben wird.

GO
In LM-Lisp sind im Gegensatz zu UCI-LISP keine numerischen Sprungmarken erlaubt.

NTH
Während in UCI-LISP bei einem Indexwert kleiner als 1 die Liste mit einem vorangestellten NIL zurückgegeben wird, löst die entsprechende LM-Lisp-Funktion NTHCDR bei einem Indexwert, der kleiner als 0 ist, einen Fehler aus. Ob in NTHCDR oder in die UCI-Kompatibilitätsfunktion übersetzt werden soll, kann mit INDEX-OK-OPTION gesteuert werden.

PRINA, PRINAC, PRINL, PRINLC, TYOA
Diese Funktionen nehmen ein optionales zweites Argument, das die Druckposition in der nächsten Zeile angibt, von der ab gedruckt werden soll, falls das zu druckende Atom oder Zeichen nicht mehr in die aktuelle Zeile paßt. Bei der Übersetzung wird diese Argument ignoriert.

PRINLEV, PRINTLEV
Diese beiden Funktionen rufen PRINA usw. auf und geben diesen Funktionen den globalen Wert der Variablen PRINLEV als zweiten Parameter.

SPRINT
wird in GRIND-TOP-LEVEL übersetzt, falls kein zweiter Parameter angegeben ist, und in SI:GRIND-FORM, falls ein zweiter Parameter vorhanden ist. Diese Annahme, daß es sich um keinen Toplevel-Aufruf handelt, kann u.U. fehlerhaft sein.

STR
Hier ergeben sich für den String-Parameter die gleichen Problem wie bei den Funktionen ANTHCHR, NTHCHAR und SUBSTRING.

TAB
In LM-Lisp ist es, wie oben schon gesagt, nicht möglich in Plattendateien an eine bestimmte Druckspalte zu springen. Bei Plattendateien wird deshalb in diesem Fall nichts gedruckt.

ANHANG C: INFORMATIONEN ÜBER ALLE UCI-LISP-FUNKTIONEN

```
;;; -*- Mode: LISP; Package: USER; Base: 10 -*-
;;; informations about all UCI-LISP system functions
;;; the character after the semicolon has the following meaning:
;;;   = - this function has a Zeta Lisp counter part with the same name and
;;;       equivalent semantic (e.g. CAR)
;;;   * - this function has a Zeta counter part with another name (e.g. PRINSTR -> PRINC)
;;;   + - this function is translated by a more or less complex rule (e.g. SOME)
;;;   f - this function is reimplemented in the file BASICFNS.LISP (e.g. TCONC)
;;;   u - we have written a compatibility function, but recommend the recoding of the program.
;;;       All these function are found in UCI-COMP.LISP and they have the prefix "UCI-".
;;;   i - this function may safely ignored.This will happen, if IGNORE-OPTION  is T.
;;;   x - this function is intentionally not handeled by the translator, because
;;;       it is an FILE-I/O, EDITOR, ENVIRONMENT, STACK or MACHINE-INTERFACE function
;;;   () means, that the translation does't cover some special cases, but a warning is issued

(definfo /#/# FSUBR (0))              ;x EDITOR call
(definfo %DEVP SUBR 1)                ;x tests for device or ppn
(definfo * SUBR 2)                    ;=
(definfo *APPEND SUBR 2)              ;* primitive APPEND (only 2 args)
(definfo *APPLY SUBR 2)               ;* primitive APPLY  (only 2 args)
(definfo *DIF SUBR 2)                 ;* = - (only 2 args)
(definfo *EVAL SUBR 1)                ;* = EVAL (only 1 args)
(definfo *EXPAND SUBR 2)              ;f macro expander (L FN)
(definfo *EXPAND1 SUBR 2)             ;f s.a.
(definfo *EXPAND2 SUBR 2)             ;f s.a.
(definfo *FUNCTION FSUBR 1 t-AllA)    ;x building functional args
(definfo *GETSYM SUBR 1)              ;x MACHINE INTERFACE: get value from LOADER-symboltable
(definfo *GREAT SUBR 2)               ;* = >
(definfo *LESS SUBR 2)                ;* = <
(definfo *MAX SUBR 2)                 ;* = MAX with 2 args
(definfo *MIN SUBR 2)                 ;* = MIN with 2 args
(definfo *NCONC SUBR 2)               ;* = NCONC with 2 args
(definfo *PG* SUBR 0)                 ;* I/O print FF
(definfo *PLUS SUBR 2)                ;* = + with 2 args
(definfo *PUTSYM SUBR 2)              ;x MACHINE INTERFACE: put value to LOADER-table
(definfo *QUO SUBR 2)                 ;* = //
(definfo *RENAME SUBR 2)              ;x FILE-I/O rename a1 to a2
(definfo *RGETSYM SUBR 1)             ;x MACHINE INTERFACE
(definfo *RPUTSYM SUBR 2)             ;x MACHINE INTERFACE
(definfo *RPUTSYMV SUBR 2)            ;x MACHINE INTERFACE
(definfo *RSET SUBR 1)                ;x ERROR-package
(definfo *TIMES SUBR 2)               ;* = *
(definfo + SUBR 2)                    ;= plus with 2 args
(definfo +I SUBR 1)                   ;* = ADD1
(definfo - SUBR 2)                    ;= difference with 2 args
(definfo -I SUBR 1)                   ;* = SUB1
(definfo // SUBR 2)                   ;= = divide with 2 args
(definfo /; FSUBR (0) t-NoA)          ;= = NILL
(definfo /;/; FSUBR (0) t-NoA)        ;= = NILL
(definfo < SUBR 2)                    ;=
(definfo = SUBR 2)                    ;= ?? not documented
(definfo =0 SUBR 1)                   ;* = (EQ x 0)
(definfo > SUBR 2)                    ;=

(definfo ABS SUBR 1)                  ;=
(definfo ADD1 SUBR 1)                 ;=
(definfo ADDPROP SUBR 3)              ;f adds a value under some prop
(definfo AEXPLODE SUBR 1)             ;+ like EXPLODE, but gives numerical values
(definfo AEXPLODEC SUBR 1)            ;* s.a.
(definfo ALLSYM FSUBR 1 t-NoA)        ;f SYMBOL-CREATION
(definfo AND FSUBR (0) t-AllA)        ;=
(definfo ANTHCHAR SUBR 2)             ;+u gives ASCII-value
(definfo APPEND LSUBR (0))            ;=
(definfo APPLY LSUBR (2 3))           ;+() 3rd arg may be BCP or A-list
(definfo APPLY# SUBR 2)               ;* allows FEXPRs and MACROS as arg
(definfo APPLYCOUNT SUBR 0)           ;x ENVIRONMENT: number of APPLies (HH-UCI)
(definfo AREF MACRO (2) t-AllA)       ;* HH-UCI like MacLisp
(definfo ARG SUBR 1)                  ;= to access args in lexprs
(definfo ARRAY FSUBR (3 7) t-AllA)    ;+() ARRAYs
```

```
(definfo ASCII SUBR 1)              ;+
(definfo ASET MACRO (3 t-AllA))     ;= HH-UCI: like MacLisp
(definfo ASSOC SUBR 2)              ;* uses EQ for comparision
(definfo ASSOC# SUBR 2)             ;* uses EQUAL
(definfo AT SUBR 1)                 ;+ atomize S-Expr
(definfo ATCAT LSUBR (0))           ;+ CONCAT + AT
(definfo ATOM SUBR 1)               ;=
(definfo ATTACH SUBR 2)             ;f attach destructively at front

(definfo BLKLST SUBR 2)             ;x ARRAYs
(definfo BOOLE LSUBR (2))           ;= bitwise boolen operations
(definfo BOUNDP SUBR 1)             ;= test for presence of value cell
(definfo BREAK FSUBR (0))           ;x TOPLEVEL-function like trace
(definfo BREAK0 SUBR 3)             ;x setup a break
(definfo BREAK1 SUBR 5)             ;x more comfortable version
(definfo BREAKIN FSUBR (0))         ;x TOPLEVEL, like BREAK

(definfo CAAAAR SUBR 1)             ;= for all C....R
(definfo CAAADR SUBR 1)
(definfo CAAAR SUBR 1)
(definfo CAADAR SUBR 1)
(definfo CAADDR SUBR 1)
(definfo CAADR SUBR 1)
(definfo CAAR SUBR 1)
(definfo CADAAR SUBR 1)
(definfo CADADR SUBR 1)
(definfo CADAR SUBR 1)
(definfo CADDAR SUBR 1)
(definfo CADDDR SUBR 1)
(definfo CADDR SUBR 1)
(definfo CADR SUBR 1)
(definfo CAR SUBR 1)
(definfo CATCH FSUBR (1) t-AllA)    ;+()
(definfo CDAAAR SUBR 1)
(definfo CDAADR SUBR 1)
(definfo CDAAR SUBR 1)
(definfo CDADAR SUBR 1)
(definfo CDADDR SUBR 1)
(definfo CDADR SUBR 1)
(definfo CDAR SUBR 1)
(definfo CDDAAR SUBR 1)
(definfo CDDADR SUBR 1)
(definfo CDDAR SUBR 1)
(definfo CDDDAR SUBR 1)
(definfo CDDDDR SUBR 1)
(definfo CDDDR SUBR 1)
(definfo CDDR SUBR 1)
(definfo CDR SUBR 1)
(definfo CHRCT SUBR 0)              ;+u I/O on LM not implementable for files
(definfo CHRPOS SUBR 0)             ;+u I/O s.a.
(definfo CHRVAL SUBR 1)             ;+ =  (ANTHCHR x 1)
(definfo CLRBFI SUBR 0)             ;+ I/O clear tty input
(definfo COMMENTBEGIN SUBR 1)       ;x SCANNER
(definfo COMMENTEND SUBR 1)         ;x SCANNER
(definfo CONCAT LSUBR (0))          ;+
(definfo COND FSUBR (0) t-CondA)    ;=
(definfo CONS SUBR 2)               ;=
(definfo CONSCOUNT SUBR 1)          ;x ENVIRONMENT, HH-UCI count of conses since start
(definfo CONSP SUBR 1)              ;+ (NOT (ATOM x))
(definfo COPY SUBR 1)               ;* copy whole structure
(definfo CORE FSUBR (0) t-AllA)     ;x ENVIRONMENT: gives amount of space, that is consumed
(definfo CSYM FSUBR 1 t-NoA)        ;+() SYMBOL-CREATION

(definfo DATE SUBR 0)               ;+ give date (month day year-1900)
(definfo DATEPRINT SUBR 0)          ;* nice printing of date
(definfo DDTIN SUBR 1)              ;i ENVIRONMENT, change TTY-I/O mode
(definfo DE FSUBR (3) t-DefA)       ;+
(definfo DECLARE FSUBR (0) t-AllA)  ;=
(definfo DECR MACRO 1 t-AllA)       ;*
```

```
(definfo DEFLIST FSUBR (2 3))       ;+ a multi DEFPROP
(definfo DEFMACRO MACRO (2)t-DefA) ;= HH-UCI: like in MacLisp
(definfo DEFP FSUBR 3 t-AllA)       ;+ set prop to another symbol
(definfo DEFPROP FSUBR 3 t-DfpA)    ;+

(definfo DEFUN MACRO (3) t-DefA)    ;= HH-UCI: like MacLisp
(definfo DEFV FSUBR 2 t-dvA)        ;+ = (setq x 'y)
(definfo DELETE FSUBR (0))          ;x FILE-I/O delete files
(definfo DELIM SUBR 1)              ;x SCANNER predicate
(definfo DELLIM SUBR 1)             ;x SCANNER
(definfo DELLIMP SUBR 1)            ;x SCANNER = DELIM
(definfo DEPOSIT SUBR 2)            ;x MACHINE INTERFACE (set memory location x to y)
(definfo DF FSUBR 3 t-DefA)         ;+
(definfo DIFFERENCE MACRO(0)t-AllA);* multiple -
(definfo DIR SUBR 1)                ;x FILE-I/O give directory as list for ppn
(definfo DIRF FSUBR 1 t-NoA)        ;x FILE-I/O prints all matching files of direc.
(definfo DIVIDE SUBR 2)             ;* = (CONS (QUOTIENT x y) (REMAINDER x y))
(definfo DM FSUBR 3 t-DefA)         ;+
(definfo DO MACRO (0) t-AllA)       ;+
(definfo DREMOVE SUBR 2)            ;* destructive remove
(definfo DREVERSE SUBR 2)           ;* destructive reverse
(definfo DRM FSUBR 2 t-AllA)        ;x readmacro definition
(definfo DSKIN FSUBR (0) t-NoA)     ;+u FILE-I/O load programs or data
(definfo DSKLOG FSUBR (0 1) t-NoA) ;+u ENVIRONMENT remeber session in file
(definfo DSKOUT FSUBR (1) t-NoA)    ;x FILE-I/O print programs and date
(definfo DSM FSUBR 2 t-AllA)        ;x splicing readmacro
(definfo DSUBST SUBR 3)             ;* destructive SUBST with EQ
(definfo DTIME SUBR 0)              ;x time of day im msec
(definfo DUMPATOMS FSUBR(0 1)t-NoA);i FILE-I/O dumps obsolet atoms (is obsolet)
(definfo DV FSUBR 2 t-DvA)          ;+ = DEFV (for us)

(definfo E/: FSUBR (0) t-AllA)      ;x PRETTYPRINT COMMAND
(definfo EDIT4E SUBR 2)             ;x EDITOR pattern matcher
(definfo EDITCH SUBR 1)             ;x I/O (TTY)
(definfo EDITE SUBR 3)              ;x EDITOR
(definfo EDITEXPR SUBR 1)           ;x EDITOR
(definfo EDITF FSUBR (1))           ;x EDITOR call (toplevel)
(definfo EDITFINDP SUBR 3)          ;x EDITOR
(definfo EDITFNS FSUBR (1))         ;x EDITOR call on many funs
(definfo EDITFPAT SUBR 2)           ;x EDITOR
(definfo EDITL SUBR 5)              ;x EDITOR
(definfo EDITP FSUBR (1))           ;x EDITOR for property lists
(definfo EDITV FSUBR (1))           ;x EDITOR for value
(definfo ENQUOTE SUBR 1)            ;f means: eval x and quote result
(definfo ENTER SUBR 2)              ;+u put into set
(definfo EQ SUBR 2)                 ;*
(definfo EQSTR SUBR 2)              ;* eq for strings
(definfo EQUAL SUBR 2)              ;=
(definfo ERR SUBR 1)                ;=
(definfo ERRCH SUBR 1)              ;i ENVIRONMENT, sets TTY-Error-Char
(definfo ERROR SUBR 1)              ;+
(definfo ERRORX SUBR 1)             ;x not documented
(definfo ERRSET FSUBR (1 2) t-AllA);=
(definfo EVAL LSUBR (1 2))          ;+()
(definfo EVAL-WHEN FSUBR(0)t-ProgA);= HH-UCI: like MacLisp
(definfo EVALCOUNT SUBR 0)          ;x ENVIRONMENT, HH-UCI: count of eval sincs start
(definfo EVALV SUBR 2)              ;x STACKOP secnd arg is pointer into stack
(definfo EVERY LSUBR (2))           ;+
(definfo EXAMINE SUBR 1)            ;x MACHINE INTERFACE get from memory location
(definfo EXARRAY FSUBR (3 8)t-AllA);x MACHINE INTERFACE use extern ARRAY
(definfo EXCISE SUBR 0)             ;* FILE-I/O, ENVIRONMENT: close all files etc.
(definfo EXIT SUBR 0)               ;x TOPLEVEL: leave LISP
(definfo EXPANDMACRO SUBR 1)        ;* expand in a sexpr all macro calls
(definfo EXPLODE SUBR 1)            ;=
(definfo EXPLODEC SUBR 1)           ;=

(definfo F/: FSUBR 1 t-AllA)        ;x PRETTYPRINT COMMAND
(definfo F/:L MACRO (1) t-ProgA)    ;+ (FUNCTION (LAMBDA (x) ...))
(definfo FILBAK SUBR 1)             ;x FILE-I/O backup file before writing
```

```
(definfo FIX SUBR 1)                ;=
(definfo FLATSIZE SUBR 1)           ;= size of sexpr when printing with PRIN1
(definfo FLATSIZEC SUBR 1)          ;* same for PRINC
(definfo FLATTEN SUBR 1)            ;f flat list
(definfo FNDBRKPT SUBR 1)           ;x STACKOP
(definfo FNTH SUBR 2)               ;+ fast NTH
(definfo FORMS/: FSUBR (0) t-AllA) ;x PRETTYPRINT COMMAND
(definfo FRECOR SUBR 0)             ;x ENVIRONMENT: gives amount of free core in a list
(definfo FREE SUBR 1)               ;i ENVIRONMENT: gives cons cells back to freelist
(definfo FREELIST SUBR 1)           ;i s.a.
(definfo FSCNT SUBR 0)              ;x ENVIRONMENT: gives amount of free FS cells
(definfo FUNCALL MACRO (1) t-AllA) ;= HH-UCI: like MacLisp
(definfo FUNCTION FSUBR 1 t-QuoteA);=
(definfo FWCNT SUBR 0)              ;x ENVIRONMENT gives amount of free FWS cells

(definfo GC SUBR 0)                 ;i ENVIRONMENT: calls GC
(definfo GCD SUBR 2)                ;= greatest common divisor
(definfo GCGAG SUBR 1)              ;i ENVIRONMENT: set/reset GC-verbose flag
(definfo GCTIME SUBR 0)             ;+ ENVIRONMENT: returns time spent in GC
(definfo GE SUBR 2)                 ;* = >=
(definfo GENSYM SUBR 1)             ;= SYMBOL-CREATION
(definfo GEQUAL SUBR 1)             ;* = GE
(definfo GET SUBR 3)                ;+u
(definfo GETDEF FSUBR (1) t-NoA)    ;x FILE-I/O loads specified functions from file
(definfo GETL SUBR 3)               ;+u get prop list
(definfo GETSYM FSUBR (2))          ;x MACHINE INTERFACE: *GETSYM for many args
(definfo GO FSUBR 1 t-NoA)          ;+()
(definfo GREATERP MACRO (0) t-AllA);* > for many args
(definfo GT SUBR 2)                 ;* = >
(definfo GTBLK SUBR 2)              ;x returns a block of memory
(definfo HGHEND SUBR 0)             ;x MACHINE INTERFACE,ENVIRONMENT
(definfo HGHIN FSUBR (0) t-NoA)     ;x FILE-I/O, MACHINE INTERFACE read prog into HIGH-seg
(definfo HGHORG SUBR 1)             ;x MACHINE INTERFACE,ENVIRONMENT
(definfo HGHPGS SUBR 1)             ;x HH-UCI, MACHINE INTERFACE, ENVIRONMENT replaces HGHCOR
(definfo HGHCOR SUBR 1)             ;x MACHINE INTERFACE, ENVIRONMENT now obsolet

(definfo IASCII SUBR 1)             ;* = (INTERN (ASCII x))
(definfo IGNORE SUBR 1)             ;x SCANNER
(definfo IGNOREP SUBR 1)            ;x SCANNER predicate
(definfo INC SUBR 2)                ;+u I/O set current input channel
(definfo INCH SUBR 0)               ;+u I/O get current input channel
(definfo INCR MACRO 1 t-AllA)       ;* incr var
(definfo INITFL FSUBR (0) t-NoA)    ;i ENVIRONMENT: sets init-file names
(definfo INITFN SUBR 1)             ;i ENVIRONMENT: sets init-function
(definfo INITPROMPT SUBR 1)         ;i ENVIRONMENT: set prompt char to use after re-init
(definfo INITSYM FSUBR 1 t-NoA)     ;f SYMBOL-CREATION
(definfo INP SUBR 2)                ;f check for inclusion
(definfo INPUT FSUBR (1) t-NoA)     ;+u FILE-I/O
(definfo INSERT SUBR  4)            ;f insert destructivly into a sorted list
(definfo INTERN SUBR 1)             ;=
(definfo INTERSECTION LSUBR 0)      ;+u
(definfo INUMP SUBR 1)              ;i on the LM there are no INMUs!!!!

(definfo LABEL FSUBR 2 t-dvA)       ;x defines a temporary function (very, very ancient)
(definfo LAMBDA FSUBR (2) t-ProgA) ;= looks strange, but does what we want
(definfo LAP FSUBR 2 t-AllA)        ;x MACHINE INTERFACE: start of a LAP function
(definfo LAST SUBR 1)               ;=
(definfo LAYOUT SUBR 0)             ;x ENVIRONMENT,HH-UCI
(definfo LCONC SUBR 2)              ;f
(definfo LCONC1 SUBR 2)             ;f not documented, maybe like NCONC1
(definfo LDIFF SUBR 2)              ;=
(definfo LE SUBR 2)                 ;* =  <=
(definfo LENGTH SUBR 1)             ;=
(definfo LEQUAL SUBR 2)             ;* =  <=
(definfo LESSP MACRO (1) t-AllA)    ;* =  < for many args
(definfo LET MACRO (1) t-ProgA)     ;+ HH-UCI: a special form of the LET
(definfo LET-CLOSED MACRO(1)t-ProgA);+ HH-UCI: like the Maclisp LET-CLOSED  with an odd syntax
(definfo LETTER SUBR 1)             ;x SCANNER
(definfo LETTERP SUBR 1)            ;x SCANNER predicate
```

```
(definfo LEXORDER SUBR 1)           ;f like alphalessp, but it looks into the cars
(definfo LINELENGTH SUBR 1)         ;+ I/O: sets and asks output linelength
(definfo LINEREAD SUBR 0)           ;f I/O: read one line as a list
(definfo LINES SUBR 1)              ;+ I/O: makes TERPRI's
(definfo LIST FSUBR (0) t-AllA)     ;=
(definfo LISTIFY1 SUBR 1)           ;* HH-UCI: like MacLisp LISTIFY
(definfo LITATOM SUBR 1)            ;* in MacLisp SYMBOLP
(definfo LOAD SUBR 1)               ;x ENVIRONMENT, MACHINE INTERFACE: load REL File
(definfo LOCAL-DECLARE MACRO (0) t-AllA);* HH-UCI: like MacLisp
(definfo LOGOUT SUBR 0)             ;+u ENVIRONMENT: ends a DSKLOG
(definfo LOOKUP SUBR 1)             ;x FILE-I/O
(definfo LSH SUBR 1)                ;= left shift bits
(definfo LSUBST SUBR 3)             ;f
(definfo LT SUBR 2)                 ;* = <

(definfo MACROEXPANSION FSUBR 2 t-AllA);x should not occur
(definfo MAKE-ARRAY MACRO(1)t-AllA);= HH-UCI: like MacLisp
(definfo MAKNAM SUBR 1)             ;= SYMBOL CREATION
(definfo MAKNUM SUBR 1)             ;x MACHINE INTERFACE
(definfo MAP LSUBR (1))             ;+
(definfo MAPATOMS SUBR 1)           ;+
(definfo MAPC LSUBR (1))            ;+
(definfo MAPCAN LSUBR (1))          ;=
(definfo MAPCAR LSUBR (1))          ;=
(definfo MAPCL LSUBR (1))           ;*
(definfo MAPCON LSUBR (1))          ;=
(definfo MAPCONC LSUBR (1))         ;*
(definfo MAPL LSUBR (1))            ;*
(definfo MAPLIST LSUBR (1))         ;=
(definfo MAX MACRO (1) t-AllA)      ;= max over many args
(definfo MBD/: FSUBR 1)             ;x PRETTYPRINT COMMAND
(definfo MCONS MACRO (0) t-A[illegible])  ;* cons for many args
(definfo MEMB SUBR 2)               ;* = MEMQ
(definfo MEMBER SUBR 2)             ;=
(definfo MEMBFN SUBR 2)             ;* normally a variable
(definfo MEMQ SUBR 2)               ;=
(definfo MERGE SUBR 4)              ;f merge 2 sorted lists
(definfo MIN MACRO (1) t-AllA)      ;= min for many args
(definfo MINUS SUBR 1)              ;* unary -
(definfo MINUSP SUBR 1)             ;=
(definfo MODCHR SUBR 1)             ;x SCANNER function
(definfo MORBPS SUBR 1)             ;i HH-UCI, ENVIRONMENT
(definfo MORFS SUBR 1)              ;i HH-UCI, ENVIRONMENT
(definfo MORFWS SUBR 1)             ;i HH-UCI, ENVIRONMENT
(definfo MORGAG SUBR 1)             ;i HH-UCI, ENVIRONMENT
(definfo MSG FSUBR (0) t-msgA)      ;+ I/O, similar to format
(definfo MSUBST SUBR 3)             ;f not doc., similar to SUBST?
(definfo MYPPN SUBR 0)              ;x FILE-I/O

(definfo NCONC LSUBR 2)             ;=
(definfo NCONC1 SUBR 2)             ;+
(definfo NCONS SUBR 1)              ;=
(definfo NEQ SUBR 2)                ;+
(definfo NEQUAL SUBR 2)             ;+
(definfo NEWPDL SUBR 0)             ;x HH-UCI, ENVIRONMENT
(definfo NEWSYM FSUBR 1 t-NoA)      ;f SYMBOL CREATION
(definfo NEXTEV SUBR 1)             ;x STACK OP
(definfo NILL FSUBR (0) t-NoA)      ;+ Comment
(definfo NOCALL FSUBR (0) t-NoA)    ;i ENVIRONMENT (used in DECLARE)
(definfo NOCOMPILE FSUBR (0) t-AllA);* means in Zeta (EVAL-WHEN (EVAL) ...)
(definfo NODUPLES SUBR 1)           ;* = (UNION x)
(definfo NOT SUBR 1)                ;=
(definfo NOTANY MACRO (1) t-NoA)    ;+ (NOT (SOME ..
(definfo NOTEVERY MACRO (1) t-NoA)  ;+ (NOT (EVERY ..
(definfo NOUUO SUBR 1)              ;i ENVIRONMENT
(definfo NTH SUBR 2)                ;+u
(definfo NTHCHAR SUBR 1)            ;+u
(definfo NULL SUBR 1)               ;=
(definfo NUMBERP SUBR 1)            ;=
```

```
(definfo NUMTYPE SUBR 1)            ;+ gives INUM, FIXNUM, FLONUM or NIL
(definfo NUMVAL SUBR 1)             ;x MACHINE INTERFACE

(definfo OLDSYM FSUBR 1 t-NoA)      ;f SYMBOL CREATION
(definfo OR FSUBR (0) t-AllA)       ;=
(definfo OUTC SUBR 2)               ;+u I/O: set new channel as current channel
(definfo OUTCH SUBR 1)              ;+u I/O: gives current channel back
(definfo OUTPUT FSUBR (1) t-NoA)    ;+u FILE-I/O
(definfo OUTVAL SUBR 2)             ;x STACK OP

(definfo P/: FSUBR (0) t-AllA)      ;x PRETTYPRINT COMMAND
(definfo PATH FSUBR (0) t-NoA)      ;x FILE-I/O, gives current default path
(definfo PATOM SUBR 1)              ;* if x is a atom or s.t., but not a list
(definfo PEEKC SUBR 0)              ;* I/O, in Zeta TYIPEEK
(definfo PLEV SUBR 0)               ;x I/O, has s.t. to do with BREAK
(definfo PLUS MACRO (1) t-AllA)     ;* + for many args
(definfo POP MACRO 1 t-AllA)        ;=
(definfo PP FSUBR (0) t-AllA)       ;+ I/O, in Zeta GRINDEF
(definfo PP-FORMAT SUBR 3)          ;x I/O, SPRINT-Macro
(definfo PP-LABELS SUBR 1)          ;x I/O, also
(definfo PP-MISER SUBR 1)           ;x I/O, also
(definfo PP-SPECIAL SUBR 1)         ;x I/O, also
(definfo PPL FSUBR (0) t-AllA)      ;x I/O, interpret PP-commands and sprint
(definfo PRELIST SUBR 2)            ;+
(definfo PREVEV SUBR 1)             ;x STACK OP
(definfo PRIN SUBR 1)               ;* I/O = PRIN1
(definfo PRIN1 SUBR 1)              ;= I/O
(definfo PRINA SUBR (1 2))          ;+() I/O, 2nd arg is startpos
(definfo PRINAC SUBR (1 2))         ;+() I/O, the same for PRINC
(definfo PRINC SUBR 1)              ;= I/O
(definfo PRINL SUBR (1 2))          ;+() I/O, print without outer ()
(definfo PRINLC SUBR (1 2))         ;+() I/O, same with PRINC
(definfo PRINLEV SUBR 2)            ;+() I/O, print to depth y
(definfo PRINT SUBR 1)              ;+ I/O, TERPRI, PRIN1 and SPACE
(definfo PRINTC SUBR 1)             ;+ I/O, same with PRINC
(definfo PRINTLEV SUBR 2)           ;+() I/O, TERPRI and PRINLEV
(definfo PRINTSTR SUBR 1)           ;* I/O, TERPRI and PRINC
(definfo PRINTTY SUBR 1)            ;+ I/O, prin1 to tty
(definfo PROG FSUBR (1) t-ProgA)    ;+
(definfo PROG1 SUBR (0 5))          ;+
(definfo PROG2 SUBR (0 5))          ;+
(definfo PROGN FSUBR (0) t-AllA)    ;+
(definfo PROMPT SUBR 1)             ;i ENVIRONMENT: sets prompt char
(definfo PUSH MACRO 2 t-AllA)       ;+ MacLisp takes the args in the other way
(definfo PUT SUBR 3)                ;+u
(definfo PUTLIST SUBR 3 )           ;+u puts props on many symbols
(definfo PUTPROP SUBR 3)            ;+u same as PUT
(definfo PUTSYM FSUBR (0))          ;x MACHINE INTERFACE

(definfo QUOTE FSUBR 1 t-QuoteA)    ;=
(definfo QUOTIENT MACRO (2) t-AllA);* divide for many args

(definfo RDFILE SUBR 0)             ;x FILE-I/O
(definfo RDFILENAM SUBR 0)          ;x FILE-I/O
(definfo RDNAM SUBR 0)              ;+ I/O, reads but doesn't intern
(definfo READ SUBR 0)               ;= I/O
(definfo READCH SUBR 0)             ;= I/O
(definfo READL SUBR 0)              ;+ I/O, = LINEREAD
(definfo READLIST SUBR 1)           ;=
(definfo READP SUBR 0)              ;+ I/O, checks, whether TTY input is available
(definfo READTTY SUBR 1)            ;+ I/O, read from tty
(definfo REALLOC SUBR (0 5))        ;x ENVIRONMENT, now obsolet
(definfo RELOCATE LSUBR 6)          ;x MH-UCI: ENVIRONMENT
(definfo REMAINDER SUBR 2)          ;=
(definfo REMLIST SUBR 2 t-NoA)      ;+u REMPROP for a list of IDs
(definfo REMOB FSUBR (0) t-AllA)    ;+
(definfo REMOVE SUBR 2)             ;=
(definfo REMPROP SUBR 2)            ;+u
(definfo REMPROPS SUBR 2 t-NoA)     ;+u REMPROP a list of props
```

```
(definfo REMSYM FSUBR 1 t-NoA)      ;f SYMBOL CREATION
(definfo RENAME FSUBR 2 t-NoA)      ;x FILE-I/O
(definfo REPEAT FSUBR (1) t-AllA)   ;+
(definfo REREADCH SUBR 1)           ;i ENVIRONMENT, set the reread char for TTY
(definfo RETFROM SUBR 2)            ;x STACK OP
(definfo RETURN SUBR 1)             ;=
(definfo REVERSE SUBR 1)            ;=
(definfo RGETSYM FSUBR (1))         ;x MACHINE INTERFACE
(definfo RPLACA SUBR 1)             ;=
(definfo RPLACD SUBR 1)             ;=
(definfo RPTQ MACRO (1) t-AllA)     ;+
(definfo RPUTSYM FSUBR (0))         ;x MACHINE INTERFACE
(definfo RUN FSUBR 3 t-NoA)         ;x MACHINE INTERFACE, start another program
(definfo RUNCCL FSUBR 3 t-NoA)      ;x MACHINE INTERFACE, s.a.

(definfo SASSOC SUBR 3)             ;* ASSOC, if nothing is found execute 3rd parameter
(definfo SAVE FSUBR 1 t-AllA)       ;x save function definition (see UNSAVE)
(definfo SCAN SUBR 1)               ;x FILE-I/O, sets/resets the SCAN-switch
(definfo SELECTQ FSUBR (2) t-SelectqA);+
(definfo SET SUBR 2)                ;=
(definfo SETARG SUBR 2)             ;= sets arg in lexpr
(definfo SETCHR SUBR 2)             ;x SCANNER
(definfo SETCOREMAX SUBR 1)         ;i ENVIRONMENT, HH-UCI
(definfo SETF MACRO 2 t-AllA)       ;= HH-UCI, like MacLisp
(definfo SETQ FSUBR 2 t-AllA)       ;=
(definfo SETSYS FSUBR 2 t-NoA)      ;i ENVIRONMENT
(definfo SLASHIFY SUBR 1)           ;x SCANNER
(definfo SOME LSUBR (1))            ;+ a mapper
(definfo SORT SUBR  3)              ;+
(definfo SPACES SUBR (1 2))         ;+ I/O
(definfo SPDLFT SUBR 1)             ;x STACK OP
(definfo SPDLPT SUBR 0)             ;x STACK OP
(definfo SPDLRT SUBR 1)             ;x STACK OP
(definfo SPEAK SUBR 1)              ;x = CONSCOUNT (ENVIRONMENT)
(definfo SPECIAL FSUBR 0 t-AllA)    ;= for DECLARE
(definfo SPREDO SUBR 1)             ;x STACK OP
(definfo SPREVAL SUBR 2)            ;x STACK OP
(definfo SPRINT SUBR (1 2))         ;+() I/O, the grinder
(definfo STKCOUNT SUBR 3)           ;x STACK OP
(definfo STKNAME SUBR 1)            ;x STACK OP
(definfo STKNTH SUBR 1)             ;x STACK OP
(definfo STKPTR SUBR 1)             ;x STACK OP
(definfo STKSRCH SUBR 3)            ;x STACK OP
(definfo STORE FSUBR 2 t-AllA)      ;= store into ARRAY (currently emulated in BASICFNS)
(definfo STR SUBR 1)                ;+ make string
(definfo STRINGBEGIN SUBR 1)        ;x SCANNER
(definfo STRINGEND SUBR 1)          ;x SCANNER
(definfo STRINGP SUBR 1)            ;+
(definfo STRLEN SUBR 1)             ;*
(definfo SUB1 SUBR 1)               ;=
(definfo SUBLIS SUBR 2)             ;=
(definfo SUBPAIR SUBR 3)            ;+
(definfo SUBSET SUBR 2)             ;=
(definfo SUBST SUBR 3)              ;=
(definfo SUBSTRING SUBR 3)          ;+u
(definfo SUFLIST SUBR 2)            ;+
(definfo SYMSTAT FSUBR 1 t-NoA)     ;f SYMBOL CREATION
(definfo SYSCLR SUBR 0)             ;+ ENVIRONMENT

(definfo TAB SUBR 1)                ;+() I/O, tab to position (on LM only on TTY poss.)
(definfo TAILP SUBR 2)              ;=
(definfo TALK SUBR 0)               ;i I/O, enables TTY-output after ^O again
(definfo TCONC SUBR 2)              ;f
(definfo TERPRI SUBR (0 1))         ;+ I/O, gives evaled arg as result (urgh!)
(definfo THROW FSUBR 2 t-AllA)      ;+
(definfo TIME SUBR 1)               ;x gives number of msecs, job has computed sincs login
(definfo TIMER FSUBR (0) t-AllA)    ;x TOPLEVEL measurement tool
(definfo TIMES MACRO (1))           ;* = * with many args
(definfo TRACE FSUBR (0))           ;x TOPLEVEL Trace Call
```

```
(definfo TRACEV FSUBR (0))          ;x same for vars
(definfo TTYECHO SUBR 0)            ;x ENVIRONMENT: complements tty echo flag
(definfo TTYIN FSUBR (0) t-AllA)    ;+ I/O, all input in this special form is read from TTY
(definfo TTYMSG FSUBR (0)t-AllA)    ;+ I/O, = (TTYOUT (MSG ...))
(definfo TTYOUT FSUBR (0)t-AllA)    ;+ I/O, similar to TTYIN
(definfo TYI SUBR 0)                ;= I/O
(definfo TYILIST FSUBR(0)t-AllA)    ;+ I/O, read input from list
(definfo TYO SUBR 1)                ;= I/O
(definfo TYOA SUBR (1 2))           ;+() I/O, 2nd parameter is start pos.
(definfo TYOLIST FSUBR(0)t-AllA)    ;+ I/I, output to list
(definfo TYPE FSUBR 1 t-NoA)        ;x FILE-I/O, types file on TTY

(definfo UFDINP SUBR 2)             ;x FILE-I/O, opens for dir. reading
(definfo UGETI SUBR 0)              ;x FILE-I/O, returns input pos in file
(definfo UGETO SUBR 0)              ;x FILE-I/O, return output pos in file
(definfo UNBOUND SUBR 0)            ;+() returns the UNBOUND value
(definfo UNBREAK FSUBR (0))         ;x UNBREAK functions
(definfo UNDUMPATOMS FSUBR(0 1)t-NoA);i ENVIRONMENT
(definfo UNION LSUBR (0))           ;+u
(definfo UNMACEXPAND SUBR 1)        ;x undo macroexpansions
(definfo UNPACKSTRING SUBR 1)       ;x ?? undoc.
(definfo UNSAVE FSUBR 1 t-AllA)     ;x ENVIRONMENT
(definfo UNTRACE FSUBR (0))         ;x UNTRACE functions
(definfo UNTRACEV FSUBR (0))        ;x same for values
(definfo UNTYI SUBR 1)              ;* I/O
(definfo USE SUBR 1)                ;x ?? undoc.
(definfo USETI SUBR 1)              ;x set pointer in file

(definfo V/: FSUBR (0))             ;x PRETTYPRINT COMMAND
(definfo VIRCOR SUBR 0)             ;x ENVIRONMENT,HH-UCI

(definfo XCONS SUBR 2)              ;= = (CONS y x)

(definfo ZEROP SUBR 1)              ;+

;;; special hack to cope with the backquote-readmacro
;;; the following functions are not part of the UCI-LISP system
;;; but they will be generated by the reader, when it encounters backquotes
(definfo si:xr-bq-cons subr 2)
(definfo si:xr-bq-list lsubr (0))
(definfo si:xr-bq-list* lsubr (0))
(definfo si:xr-bq-append lsubr (0))
(definfo si:xr-bq-nconc lsubr (0))
```

ANHANG D: TRANSFORMATIONSREGELN

```
;;; -*- Mode: LISP; Package: USER; Base: 10 -*-
;;; some rules take (runtime) options for the translation process:
;;; DISK-IO-OPTION - translate some of the common disk-io functions, namely
;;;   INC, INCH, OUTC, OUTCH, INPUT, OUTPUT, DSKIN, CHRCT, CHRPOS, DSKLOG and LOGOUT,
;;;   into the UCI compatibility functions UCI-INC etc., which are defined in the
;;;   file UCI-COMP.LISP
;;; INDEX-OK-OPTION - translate NTHCHAR, ANTHCHAR, NTH etc. with the assumption
;;;   that index range is correct, instead of using UCI-NTHCHAR etc.
;;; NO-NUMBER-STRING-OPTION - assume, that no numbers will be coerced into strings
;;;   in the functions: STR ATCAT CONCAT SUBSTRING ANTHCHAR NTHCHAR CHRVAL.
;;; EQ-SETS-OPTION - translate UNION, INTERSECTION etc. to the Zeta counterparts,
;;;   which use  EQ, otherwise the special functions with "UCI-" prefix are used.
;;; IGNORE-OPTION - translate GC, GCTIME etc. into nothing (if possible) or nil, 0
;;;   or whatever is adequat. If IGNORE-OPTION=NIL then the calls will be flaged as
;;;   untranslatable.
;;; PROPERTY-LIST-OK-OPTION - use only the Zeta functions.
;;;   That means that GET, GETL and PUT will be correct for functions, values etc.
;;;   only when they are constant properties. If this Option is NIL, the compatability
;;;   functions with the UCI-prefix are used.

(eval-when (compile eval) (setq **comma-flag** t))

(delete-all-rules)

(defvar disk-io-option nil)
(defvar no-number-string-option t)
(defvar index-ok-option t)
(defvar eq-sets-option t)
(defvar ignore-option t)
(defvar property-list-ok-option t)

;; some PASS1 rules first
(-> (defprop ,name (lambda ,args ,@body) ,prop) -nothing-
    pass 1
    test (memq prop '(expr fexpr macro))
    side-effect (note-function 'UCI-LISP name prop
                               (if (and (eq prop 'expr)
                                        (listp args))
                                   (length args)
                                   '(0))))

(-> (dm ,name ,par ,@body) (defprop ,name (lambda ,par ,@body) macro)
    pass (1 2)
    rule-type non-terminal)
(-> (de ,name ,par ,@body) (defprop ,name (lambda ,par ,@body) expr)
    pass (1 2)
    rule-type non-terminal)
(-> (df ,name ,par ,@body) (defprop ,name (lambda ,par ,@body) fexpr)
    pass (1 2)
    rule-type non-terminal)
(-> (defun ,name ,lambda-list ,@rest) -nothing-
    pass 1
    side-effect (note-function 'UCI-LISP name 'expr
                               (if (intersection '(/:REST /:OPTIONAL) lambda-list)
                                   '(0)
                                   (length lambda-list))))
(-> (defmacro ,name ,args ,@rest) -nothing-
    pass 1
    side-effect (note-function 'UCI-LISP name 'macro '(0)))
(-> (progn ,@body) -nothing-
    pass 1
    side-effect (t-List body))
(-> (prog ,vars ,@body) -nothing-
    pass 1
    side-effect (t-List body))

;; now all the pass 2 rules (alphabeticly ordered):
(-> (,f ,@arg-list) ,(correct-number-of-args-if-wrong f arg-list)
    rule-type non-terminal)                    ;this rule supplies missing arguments
```

```
(-> /:optional &optional)
(-> /:rest &rest)
(-> *append append)
(-> *apply apply)
(-> *dif -)
(-> *expand *expand)
(-> *expand1 *expand1)
(-> *expand2 *expand2)
(-> *eval eval)
(-> *great >)
(-> *less <)
(-> *max max)
(-> *min min)
(-> *nconc nconc)
(-> (*pg*) (format t "~:|"))
(-> *plus +)
(-> *quo //)
(-> *times *)

(-> + +)
(-> +I 1+)

(-> - -)
(-> -I 1-)

(-> // //)

(-> |;| |;|)                                    ;only for comments
(-> |;;| |;;|)
(-> |;;;| |;;;|)

(-> < <)

(-> = =)
(-> =0 zerop
    rule-type non-terminal)

(-> > >)

(-> abs abs)
(-> add1 add1)
(-> (addprop ,a ,v ',i) ($putproping$ putprop ,a ,v ',i)
    rule-type non-terminal
    test (memq i '(pname value expr fexpr macro subr lsubr fsubr)))
(-> addprop addprop
    test property-list-ok-option)
(-> addprop uci-addprop)
(-> (aexplode ,x)  (exploden (format nil "~S" ,x)))
(-> aexplodec exploden)
(-> allsym allsym)
(-> (and ,@x (and ,@y) ,@z) (and ,@x ,@y ,@z)
     rule-type cyclic)
(-> (and ,x) ,x)
(-> and and)
(-> anthchar getcharn
    test (and index-ok-option no-number-string-option))
(-> (anthchar ,s ,n) (getcharn (format nil "~A" ,s) ,n)
    test index-ok-option)
(-> anthchar uci-anthchar)
(-> append append)
(-> apply# apply)
(-> (apply ,f ,l) (apply ,f ,l))
(-> (apply ,f ,l ,env) (apply-enviroment ,f ,l ,env)
   side-effect (warn-on-pos "2nd argument to APPLY may only a A-list, not a BCP!"))
(-> (array ,id ,type ,@dims) (array ,id ,type ,@dims)
     side-effect (if (or (neq type 'T)
                         (some dims #'(lambda (cl)
                                        (not (numberp cl))))))
```

```
                    (warn-on-pos "WARNING: ARRAY-type and -offset are not supported")))
(-> aref aref)
(-> arg arg)
(-> (ascii ,x) (make-symbol (string ,x)))
(-> aset aset)
(-> assoc assq)
(-> assoc# assoc)
(-> (at ,x) (intern (format nil "~A" ,x)))
(-> (atcat ,@as) (intern (string-append ,@as))
    test no-number-string-option)
(-> (atcat ,@as) (intern (format nil "~{~A~}" (list ,@as))))
(-> atom atom)                                  ;this isn't 100% correct, but I think
                                                ;you can't see the difference
(-> attach attach)

(-> boole boole)
(-> boundp boundp)

(-> caaaar caaaar)
(-> caaadr caaadr)
(-> caaar  caaar)
(-> caadar caadar)
(-> caaddr caaddr)
(-> caadr  caadr)
(-> caar   caar)
(-> cadaar cadaar)
(-> cadadr cadadr)
(-> cadar  cadar)
(-> caddar caddar)
(-> cadddr cadddr)
(-> caddr  caddr)
(-> cadr   cadr)
(-> (car (nthcdr ,n ,x)) (nth ,n ,x))
(-> car    car)
(-> (catch ,f ,tg) (*catch ',tg ,f))
(-> (catch ,f) (catch ,f)
    side-effect "The CATCH-ALL doesn't work, don't know why.")
(-> catch catch
    side-effect "The SELECTQ kind of CATCH isn't implemented")
(-> cdaaar cdaaar)
(-> cdaadr cdaadr)
(-> cdaar  cdaar)
(-> cdadar cdadar)
(-> cdaddr cdaddr)
(-> cdadr  cdadr)
(-> cdar   cdar)
(-> cddaar cddaar)
(-> cddadr cddadr)
(-> cddar  cddar)
(-> cdddar cdddar)
(-> cddddr cddddr)
(-> cdddr  cdddr)
(-> cddr   cddr)
(-> cdr    cdr)
(-> chrct uci-chrct
    test disk-io-option)
(-> (chrct) (- (send standard-output ':size-in-characters)
               (send standard-output ':read-cursorpos ':character)))
(-> chrpos uci-chrpos
    test disk-io-option)
(-> (chrpos) (send standard-output ':read-cursorpos ':character))
(-> chrval character
    test no-number-string-option)
(-> (chrval ,x) (character (format nil "~A" ,x)))
(-> (clrbfi) (send terminal-io ':clear-input))
(-> concat string-append
    test no-number-string-option)
(-> (concat ,@args) (format nil "~{~A~}" (list ,@args)))
(-> cond cond)
```

```
(-> cons cons)
(-> (consp ,x) ($check-predicate-result-needed$ listp ,x)
    rule-type non-terminal)
(-> copy copytree)
(-> (csym ,id) (gensym ',id)
    side-effect
    (warn-on-pos "WARNING: Only the 1st char is used as a prefix char in GENSYM"))

(-> (date) (multiple-value-bind (ignore ignore ignore date month year ignore ignore)
               (time:get-time) (list month date year)))
(-> dateprint time:print-current-time)
(-> (ddtin ,x) ($ignore-if-possible$ (ddtin ,x) nil)
    rule-type non-terminal)
(-> declare declare)
(-> decr decf)
(-> (deflist ,l ,p) (deflist ,l T ,p)
    rule-type non-terminal)
(-> (deflist ,l ,d ,p) (translate-sexpression
                         `(progn ,@(if (cdr *translate-stack*)
                                       nil
                                       '('compile))
                                 ,@(mapcar #'(lambda (el)
                                               (if (atom el)
                                                   `(defprop ,el ,d ,p)
                                                   `(defprop ,(car el) ,(cadr el) ,p))) l))))
(-> (defmacro ,name ,args ,@body)
    ($progn-check$ defmacro ,name ,args $progn-check$ ,@body)
             side-effect (test-system-function-redefine name)
             rule-type non-terminal)

(-> (defp ,new ,old ,prop) (deff ,new (function ,old))
    test (memq prop '(expr fexpr macro subr fsubr lsubr)))
(-> (defp ,new ,old 'pname) nil
    side-effect (warn-on-pos "you can't change a pname!"))
(-> (defp ,new ,old 'value) (setq ,new ,old)
    side-effect (warn-on-pos "you only got the same value, but not the same special-cell"))
(-> (defp ,new ,old ,prop) (put ',new (get ',old ',prop) ',prop))
(-> (defprop ,name (lambda ,args ,@body) ,prop)
    ($progn-check$ defun ,name ,@(if (neq 'expr prop) (list prop) nil)
                   ,args $progn-check$ ,@body)
    test (memq prop '(expr fexpr macro))
    side-effect (test-system-function-redefine name)
    rule-type non-terminal)
(-> (defprop ,name ,val printmacro) (defprop ,name ,val si:grind-macro)
    side-effect (note-on-pos "GRIND-MACROS are a little bit different from PRINTMACROS"))
(-> defprop defprop)
;; df, de and dm are handeled by the very general non-terminal translation rules under pass1
(-> (defun ,name ,lambda-list ,@body)
    ($progn-check$ defun ,name ,lambda-list $progn-check$ ,@body)
    side-effect (test-system-function-redefine name)
    rule-type non-terminal)
(-> (defv ,var ,qval) (dv ,var ,qval)
    rule-type non-terminal)
(-> difference -)
(-> (divide ,x ,y) (cons (// ,x ,y) (\ ,x ,y))
    test (and (atom x) (atom y)))
(-> (divide ,x ,y) (let ((x ,x) (y ,y)) (cons (// x y) (\ x y))))
(-> (do while (not ,p) ,@body) (do until ,p ,@body)
    rule-type non-terminal)
(-> (do while ,p ,@body) (do until (not ,p) ,@body)
    rule-type non-terminal)
(-> (do until ,p ,@body) ($do-check$ do () (,p) ,@body)
    rule-type non-terminal)
(-> (do for ,v in ,l ,@body) ($progn-check$ dolist (,v ,l) $progn-check$ ,@body)
    test (only-side-effect)
    rule-type non-terminal)
(-> (do for ,v in ,l ,@body) ($do-check$ do ((*do-list* ,l (cdr *do-list*))
                                             (,v))
                                            ((null *do-list*))
```

```
                                        ,@body)
     rule-type non-terminal)
(-> (do for ,v on ,l ,@body) ($do-check$ do ((,v ,l (cdr ,v)))
                                        ((null ,v))
                                        ,@body)
     rule-type non-terminal)
(-> (do for ,v rpt ,n ,@body) ($do-check$ do ((,v 1 (1+ ,v))
                                             (*do-end-value* ,n))
                                        ((> ,v *do-end-value*))
                                        ,@body)
     rule-type non-terminal)
(-> ($do-check$ do ,vars ,test ,@body) ($progn-check$ do ,vars ,test $progn-check$ ,@body)
     test (or (only-side-effect) (null body))
     rule-type non-terminal)
(-> ($do-check$ do (,@vs) (,p) ,@body ,last)
    ($progn-check$ do (,@vs (*do-loop-result*)) (,p *do-loop-result*)
      $progn-check$ ,@body (setq *do-loop-result* ,last))
    rule-type non-terminal)
(-> do progn)                                   ;the primitive form of DO
(-> dremove delq)
(-> dreverse nreverse)
(-> dskin uci-dskin
     test disk-io-option)
(-> dsklog uci-dsklog
     test disk-io-option)
(-> dsubst nsubst)                              ;not 100% because NSUBST uses EQUAL
(-> (dumpatoms ,@x) ($ignore-if-possible$ (dumpatoms ,@x) nil)
    rule-type non-terminal)
(-> (@v ,var ,qval) (**ignore-form**)
     test (useless-file-variable var))
(-> (dv ,var ,qval) (defvar ,var ',qval))

(-> enquote enquote)
(-> (enter ,el ,set) (if (not (memq ,el ,set)) (cons ,el ,set))
     test eq-sets-option)
(-> enter uci-enter)
(-> eq eq)
(-> eqstr string-equal
    side-effect (warn-on-pos "STRING-EQUAL is not case sensitive!"))
(-> equal equal)
(-> err err)
(-> (errch ,x) ($ignore-if-possible$ (errch ,x) nil)
    rule-type non-terminal)
(-> (error nil) (ferror nil ""))
(-> (error ,x) (ferror nil "~A" ,x)
     test (or (stringp x) (and (listp x) (eq (car x) 'QUOTE) (atom (cadr x)))))
(-> (error ,x) (ferror nil "~{~A ~}" ,x)
     test (and (listp x) (or (eq (car x) 'list)
                             (and (eq (car x) 'QUOTE) (listp (cadr x))))))
(-> (error ,x)
     (ferror nil "~{~A ~}" (if (let ((errm ,x)) (if (atom errm) (ncons errm) errm)))))
(-> errset errset)
(-> (eval ,x) (eval ,x))
(-> (eval ,x nil) (eval ,x))
(-> (eval ,x ,env) (eval-enviroment ,x ,env)
    side-effect (warn-on-pos "2nd argument to EVAL may only a A-list, not a BCP!"))
(-> eval-when eval-when)
(-> (every ,p ,l) (every ,l ,p))
(-> (every ,p ,@ls)
    (*catch '*every-tag* (progn (mapc #'(lambda every-arg
                                          (cond ((not (apply ,fn (listify every-arg)))
                                                 (*throw '*every-tag* nil))))
                                      ,@ls)
                                t)))
(-> excise fs:close-all-files)
(-> expandmacro macroexpansion)
(-> explode explode)
(-> explodec explodec)
```

```
(-> (F/:L ,args ,@body) (function (lambda ,args ,@body)))
(-> fix fix)
(-> flatsize flatsize)
(-> flatsizec flatc)
(-> flatten flatten)
(-> fnth nth
    rule-type non-terminal)
(-> (free ,x) ($ignore-if-possible$ (free ,x) nil)
     rule-type non-terminal)
(-> (freelist ,x) ($ignore-if-possible$ (freelist ,x) nil)
     rule-type non-terminal)
(-> funcall funcall)
(-> function function)

(-> (gc) ($ignore-if-possible$ (gc) nil)
     rule-type non-terminal)
(-> gcd gcd)
(-> (gcgag ,x) ($ignore-if-possible$ (gcgag ,x) nil)
     rule-type non-terminal)
(-> (gctime) 0)
(-> ge >=)
(-> gensym gensym)
(-> gequal >=)
(-> (get ,v 'pname) (get-pname ,v)
    side-effect (warn-on-pos "You will get the print-name instead of the list of..."))
(-> (get ,v 'value) (locf ,v)
    side-effect (warn-on-pos "You will get a pointer to the value-cell (but never nil)"))
(-> (get ,v ,p) (uci-get ,v ,p)
    test (and (not property-list-ok-option) (or (nlistp p) (neq (car p) 'QUOTE))))
(-> (get ,v ,p) (get ,v ,p)
    test (or (nlistp p)
             (neq (car  p) 'QUOTE)
             (not (memq (cadr p) '(expr fexpr macro subr fsubr lsubr)))))
(-> (get ,v ,p) (fdefinedp ,v)
    test (no-predicate-result-needed))
(-> (get ,v ,p) (car (errset (fdefinition ,v) nil))
    side-effect (warn-on-pos "You will get the contents of the function cell"))
(-> (getl ,v ,pl) (uci-getl ,v ,pl)
    test (not property-list-ok-option))
(-> (getl ,v ,pl) (getl ,v ,pl)
    test
    (or (nlistp p)
        (neq (car  p) 'QUOTE)
        (null (intersection (cadr pl) '(expr fexpr macro subr fsubr lsubr value pname)))))
(-> (getl ,v `(,@a value ,@b)) (ncons (locf ,v))
    side-effect (warn-on-pos "You will get the NCONSED pointer to a value cell"))
(-> (getl ,v `(,@a pname ,@b)) (ncons (get-pname ,v))
    side-effect (warn-on-pos "You know, what you are doing????"))
(-> (getl ,v ,pl) (fdefinedp ,v)
    test (no-predicate-result-needed)
    side-effect (warn-on-pos "We check only, whether a function is defined"))
(-> (getl ,v ,pl) (errset (fdefintion ,v) nil)
    side-effect (warn-on-pos "You will only get the NCONSED contents of the function cell"))
(-> (go ,tag) (go ,tag)
     side-effect (if (numberp tag) (note-on-pos "in Zeta go-tags must not be numbers")))
(-> greaterp >)
(-> gt >)

(-> iascii ascii)
(-> inc uci-inc
    test disk-io-option)
(-> inch uci-inch
    test disk-io-option)
(-> incr incf)
(-> (initfl ,@rest) ($ignore-if-possible$ (initfl ,@rest) nil)
    rule-type non-terminal)
(-> (initfn ,fn) ($ignore-if-possible$ (initfn ,fn) nil)
    rule-type non-terminal)
(-> (initprompt ,x) ($ignore-if-possible$ (initprompt ,x) #/*)
```

```
    rule-type non-terminal)
(-> initsym initsym)
(-> inp inp)
(-> input uci-input
    test disk-io-option)
(-> (insert ,@rest) ($sort-check$ insert ,@rest)
    rule-type non-terminal)
(-> intern intern)
(-> (inump ,x) ($ignore-if-possible$ (inump ,x) nil)    ;in Zeta there are no INUMs
    rule-type non-terminal)

(-> (lambda ,args ,@body) ($progn-check$ lambda ,args $progn-check$ ,@body)
    rule-type non-terminal)
(-> lambda lambda)
(-> ((lambda ,fargs ,@body) ,@aargs)
    ($progn-check$ let ,(pair-pars fargs aargs) $progn-check$ ,@body)
    rule-type non-terminal)
(-> last last)
(-> lconc lconc)
(-> ldiff ldiff)
(-> le <=)
(-> length length)
(-> lessp <)
(-> (let ,par ,@body) ($progn-check$ let ,(make-let-parameter par) $progn-check$ ,@body)
    rule-type non-terminal)
(-> (let-closed ,par ,@body)
    ($progn-check$ let-closed ,(make-let-parameter par) $progn-check$ ,@body)
    rule-type non-terminal)
(-> lequal <)
(-> lexorder lexorder)
(-> (linelength ,x) (si:grind-width-of-stream standard-output)
    side-effect (warn-on-pos "we can't set the linelength but ask"))
(-> (lines ,n) (format t "~V&" ,(add1 n))
    test (numberp n))
(-> (lines ,n) (format t "~V&" (add1 ,n)))
(-> list list)
(-> listify1 listify)
(-> litatom symbolp)
(-> local-declare local-declare)
(-> logout uci-logout
    test disk-io-option)
(-> lsh lsh)
(-> lsubst lsubst)
(-> lt <)

(-> make-array make-array)
(-> maknam maknam)
(-> mapatoms mapatoms
    side-effect (warn-on-pos "to get really all atoms use MAPATOMS-ALL"))
(-> (map ,@rest) ($check-nil-needed$ map ,@rest)
     rule-type non-terminal)
(-> (mapc ,@rest) ($check-nil-needed$ mapc ,@rest)
     rule-type non-terminal)
(-> maplist maplist)
(-> mapl maplist)
(-> mapcar mapcar)
(-> mapcl mapcar)
(-> mapcon mapcon)
(-> mapcan mapcan)
(-> mapconc mapcan)
(-> max max)
(-> mcons list*)
(-> memb memq)
(-> member member)
(-> membfn memq
    test eq-sets-option)
(-> (membfn ,@as) (funcall uci-membfn ,@as))
(-> membfn uci-membfn)
(-> memq memq)
```

```
(-> merge merge)
(-> min min)
(-> minus -)
(-> (minusp ,x) ($check-predicate-result-needed$ minusp ,x)
     rule-type non-terminal)
(-> (morbps ,x) ($ignore-if-possible$ (morbps ,x) ,x)
    rule-type non-terminal)
(-> (morfs ,x) ($ignore-if-possible$ (morfs ,x) ,x)
    rule-type non-terminal)
(-> (morfws ,x) ($ignore-if-possible$ (morfws ,x) ,x)
    rule-type non-terminal)
(-> (morgag ,x) ($ignore-if-possible$ (morgag ,x) nil)
    rule-type non-terminal)
(-> (msg ,@args) (format t ,@(make-format-parameter args)))

(-> nconc nconc)
(-> (nconc1 ,x ,y) (nconc ,x (ncons ,y)))
(-> ncons ncons)
(-> neq neq)
(-> (nequal ,x ,y) (not (equal ,x ,y)))
(-> newsym newsym)
(-> (nill ,@a) ($check-nil-needed$ (comment ,@a)))
(-> (nocall ,@x) (**ignore-form**))
(-> (nocompile (**ignore-form**)) (**ignore-form**))
(-> (nocompile ,@x) (eval-when (eval) ,@x))
(-> noduples union
    rule-type non-terminal)
(-> not not)
(-> (notany ,@rest) (not ,(translate-sepression `(some ,@rest)))
    rule-type non-terminal)
(-> (notevery ,@rest) (not ,(translate-sexpression `(every ,@rest)))
    rule-type non-terminal)
(-> (nouuo ,flag) ($ignore-if-possible$ (nouuo ,flag) nil)
    rule-type non-terminal)
(-> nth uci-nth
    test (not index-ok-option))
(-> (nth ,x ,n) (nthcdr ,(sub1 n) ,x)
    test (and (numberp n) (> n 0)))
(-> (nth ,x ,n) (nthcdr (sub1 ,n) ,x))
(-> nthchar getchar
     test (and index-ok-option no-number-string-option))
(-> (nthchar ,x ,n) (getchar (format nil "~A" ,x) n)
    test index-ok-option)
(-> nthchar uci-nthchar)
(-> null null)
(-> (numberp ,x) ($check-predicate-result-needed$ numberp ,x)
     rule-type non-terminal)
(-> (numtype ,x) (car (memq (numtype ,x) '(fixnum bignum flonum small-flonum))))

(-> (or ,@x (or ,@y) ,@z) (or ,@x ,@y ,@z)
     rule-type cyclic)
(-> (or ,@x ,expr ,expr ,@y) (or ,@x ,expr ,@y)
    rule-type cyclic
    test (no-side-effect expr))
(-> (or ,x) ,x)
(-> or or)
(-> outc uci-outc
    test disk-io-option)
(-> outch uci-outch
    test disk-io-option)
(-> output uci-output
    test disk-io-option)

(-> patom atom)                                  ;that's 100%
(-> peekc tyipeek)
(-> plus +)
(-> pop pop)
(-> (pp ,x) (grindef ,x))
(-> (pp ,@x) (dolist (one-def ,x) (apply #'grindef one-def)))
```

```
(-> (prelist ,l ,n) (if (> ,n (length ,l)) (copylist ,l) (firstn ,n ,l))
    test (and (or (atom l) (eq (car l) 'QUOTE)) (atom n)))
(-> (prelist ,l ,n) (let ((prelist ,l )(elnum ,n) (if (> elnum (length prelist))
                                        (copylist prelist)
                                        (firstn elnum prelist)))))
(-> prin prin1)
(-> prin1 prin1)
(-> (prina ,x ,@rest) (prin1 ,x))              ;we ignore the conditional break-line
(-> (prinac ,x ,@rest) (princ ,x))             ;same thing
(-> princ princ)
(-> (prinl ,l) ($check-print$ format t "~{~S ~}" ,l)
    rule-type non-terminal)
(-> (prinlc ,l) ($check-print$ format t "~{~A ~}" ,l)
    rule-type non-terminal)
(-> (prinlev ,x ,depth) (let ((prinlevel ,depth)) (prin1 ,x)))
(-> print print)
(-> (printc ,x) ($check-print$ format t "~%~A " ,x)
    rule-type non-terminal)
(-> (printlev ,x) ((let ((prinlevel ,depth)) (print ,x))))
(-> printstr princ)
(-> (printty ,x) (prin1 ,x terminal-io)
    test disk-io-option)
(-> printty prin1)
(-> (prog ,v ,@b) ($progn-check$ prog ,v $progn-check$ ,@b)
    rule-type non-terminal)
(-> prog prog)
(-> (prog1 ,p1 ,@rest) ($progn-check$ prog1 ,p1 $progn-check$ ,@rest)
    rule-type non-terminal)
(-> prog1 prog1)
(-> (prog2 ,p1 ,p2 ,@rest) ($progn-check$ ,p1 ,p2 $progn-check$ ,@rest)
    rule-type non-terminal)
(-> prog2 prog2)
(-> (progn ,x) ,x)
(-> (progn ,@body) ($progn-check$ progn $progn-check$ ,@body)
    rule-type non-terminal)
(-> progn progn)
(-> (prompt ,x) ($ignore-if-possible$ (prompt ,x) #/*)
    rule-type non-terminal)
(-> (push ,var ,val) (push ,val ,var))
(-> (put ,@rest) ($putproping$ putprop ,@rest)
    rule-type non-terminal)
(-> (putlist ,l ,v ,p) ($putproping$ (putlist ,l) *putlistarg* ,v ,p )
    rule-type non-terminal)
(-> (putprop ,@rest) ($putproping$ putprop ,@rest)
    rule-type non-terminal)

(-> ($putproping$ ,what ,a ,v 'value ) ($do-putproping$ ,what set ,a (cdr ,v))
    side-effect (warn-on-pos "we can't put an identical value-cell, but only the value")
    rule-type non-terminal)
(-> ($putproping$ ,what ,a ,v 'pname) nil
    side-effect (warn-on-pos "can't change a pname (you fool!)"))
(-> ($putproping$ ,what ,a ,v ',fp) ($do-putproping$ ,what apply #'defun (list* ,a ,fp (cdr ,v)))
    test (memq fp '(expr fexpr macro))
    rule-type non-terminal)
(-> ($putproping$ ,what ,a ,v ',fp) ($do-putproping$ ,what fdefine ,a ,v)
    rule-type non-terminal
    test (memq fp '(subr lsubr macro fsubr)))
(-> ($putproping$ ,what ,@rest) ($do-putproping$ ,what putprop ,@rest)
    rule-type non-terminal
    test property-list-ok-option)
(-> ($putproping$ ,what ,@rest) ($do-putproping$ ,what  uci-putprop ,@rest)
    rule-type non-terminal)
(-> ($do-putproping$ putprop ,@rest) ,rest)
(-> ($do-putproping$ (putlist ,expr) ,@rest) (dolist (*putlistarg* ,expr) (,@rest)))

(-> quotient //)
(-> quote quote)

(-> rdnam read
```

```
    rule-type non-terminal
    side-effect (warn-on-pos "non-interning reading isn't possible"))
(-> readlist readlist)
(-> read read)
(-> readch readch)
(-> readl lineread
    rule-type non-terminal)
(-> (readp) (send terminal-io ':listen))
(-> (readtty) (read terminal-io)
    test disk-io-option)
(-> readtty read
    rule-type non-terminal)
(-> remainder /\)
(-> (remlist ,l ,prop) ($remproping$ (remlist ,l) *remarg* ,prop)
    rule-type non-terminal)
(-> (remob ,a) ($check-nil-needed$ remob ',a)
    rule-type non-terminal)
(-> (remob ,@a) (dolist (*remobarg* ',a) (remob *remobarg*)))
(-> remove remove)
(-> (remprop ,@rest) ($remproping$ remprop ,@rest)
    rule-type non-terminal)

(-> ($remproping$ ,what ,var 'value) ($do-remproping$ ,what makunbound ,var)
    rule-type non-terminal)
(-> ($remproping$ ,what ,var 'pname) (**ignore-form**)
    side-effect (warn-on-pos "it's not very kind to steal the PNAME, i ignore this"))
(-> ($remproping$ ,what ,fun ',funprop) ($do-remproping$ ,what fmakunbound ,fun)
    test (memq funprop '(expr fexpr macro subr lsubr fsubr))
    rule-type non-terminal)
(-> ($remproping$ ,what ,v ,p) ($do-remproping$ ,what remprop ,v ,p)
    test property-list-ok-option
    rule-type non-terminal)
(-> ($remproping$ ,what ,v ,p) ($do-remproping$ ,what uci-remprop ,v ,p)
    rule-type non-terminal)
(-> ($do-remproping$ (,op ,l) ,@rest) (dolist (*remarg* ,l) ,@rest))
(-> ($do-remproping$ remprop ,@rest) ,rest
    test (only-side-effect))
(-> ($do-remproping$ remprop ,@rest) (not (null (,@rest))))

(-> (remprops ,at ',plist ,@aux) (progn ,@aux
                                        (dolist (*remarg* ',plist) (remprop ,at *remarg*)))
    test (progn (if (memq 'value plist) (push `(makunbound ,at) aux))
                (if (intersection '(expr fexpr macro subr fsubr lsubr) plist)
                    (push `(fmakunbound ,at) aux))
                t))
(-> (remprops ,at ,ps) ($remproping$ (remprops ,ps) ,at *remarg*)
    rule-type non-terminal)
(-> remsym remsym)
(-> (repeat ,n ,@body) ($progn-check$ dotimes (*rptcnt* ,n) $progn-check$ ,@body)
    rule-type non-terminal)
(-> (rereadch ,x) ($ignore-if-possible$ (rereadch ,x) ,x)
    rule-type non-terminal)
(-> return return)
(-> reverse reverse)
(-> rplaca rplaca)
(-> rplacd rplacd)
(-> (rptq ,n ,@body) (do for rptn rpt ,n ,@body)
    rule-type non-terminal)

(-> sassoc sassq)

(-> (selectq ,sel-expr ,@cases ,otherwise)
    (selectq ,sel-expr ,@cases (otherwise ,otherwise)))
(-> (set ,x (unbound)) (makunbound ,x)
    side-effect (if (not (only-side-effect))
                    (note-on-pos "You will get the symbol as the result, not (unbound)!")))
(-> set set)
(-> setarg setarg)
(-> setf setf)
```

```
(-> (setcoremax ,x) ($ignore-if-possible$ (setcoremax ,x) 512)
    rule-type non-terminal)
(-> (setq ,x (unbound)) (makunbound ',x)
    side-effect (if (not (only-side-effect))
                    (note-on-pos "You will get the symbol as the result, not (unbound)!")))
(-> setq setq)
(-> (setsys ,@rest) ($ignore-if-possible$ (setsys ,@rest) nil)
    rule-type non-terminal)
(-> (some ,p ,l) (some ,l ,p))
(-> (some ,p ,@ls ,last) (*catch '*some-tag*
                           (prog (*some-list*)
                                 (setq *some-list* ,last)
                                 (mapc #'(lambda every-arg
                                           (cond ((apply ,fn (listify every-arg))
                                                  (*throw '*some-tag* *some-list*))
                                                 (t (pop *some-list*))))
                                       ,@ls ,last))))

(-> (sort ,l ,p ,n) ($sort-check$ sort (copylist ,l) ,p ,n)
    side-effect (warn-on-pos "The COPYLIST in the translated SORT may be not necessary")
    rule-type non-terminal)
(-> (spaces ,n ,@pos) (format t "~VX" ,n))      ;we ignore the pos argument
(-> special special)
(-> (sprint ,x) (grind-top-level ,x))
(-> (sprint ,x ,p) (si:grind-form ,x (ncons ,x))
    side-effect (warn-on-pos "we assume, that this is not a toplevel call to sprint!"))
(-> store store) ;currently this must be emulated by BASICFNS
(-> str string
    test no-number-string-option)
(-> (str ,x) (format nil "~A" ,x))              ;Zeta STRING doesn't work for numbers!
(-> (stringp ,x) ($check-predicate-result-needed$ stringp ,x)
    rule-type non-terminal)
(-> strlen string-length)
(-> sub1 sub1)
(-> sublis sublis)
(-> subset subset)
(-> subst subst)
(-> (substring ,s ,low ,high) (substring ,s ,(sub1 low) ,high)
    test (and (numberp low) index-ok-option no-number-string-option))
(-> (substring ,s ,low ,high) (substring ,s (sub1 ,low) ,high)
    test (and  index-ok-option no-number-string-option))
(-> (substring ,s ,low ,high) (substring (format nil "~A" ,s) (sub1 ,low) ,high)
    test index-ok-option)
(-> substring uci-substring)
(-> (subpair ,old ,new ,exp) (sublis (pairlis ,old ,new) ,exp))
(-> (suflist ,l ,n) (nthcdr ,n ,l)
    test (or index-ok-option (and (numberp n) (> n 0))))
(-> (suflist ,l ,n) (if (minusp ,n) nil (nthcdr ,n ,l)))
(-> symstat symstat)
(-> sysclr excise
    rule-type non-terminal)

(-> (tab ,n) (format:tab ,n ':terpri)
    side-effect (if disk-io-option (warn-on-pos "TAB doesn't work for files")))
(-> (tailp ,x ,y) ($check-predicate-result-needed$ tailp ,x ,y)
    rule-type non-terminal)
(-> (talk) ($ignore-if-possible$ (talk) nil)
    rule-type non-terminal)
(-> tconc tconc)
(-> (terpri) ($check-nil-needed$ terpri)
    rule-type non-terminal)
(-> (terpri ,x) (terpri)
    test (and (or (atom x) (and (listp x) (eq (car x) 'QUOTE))) (only-side-effect)))
(-> (terpri ,x) (let ((*terpri-arg* ,x) (terpri) *terpri-arg*)))
(-> (throw ,form ,tag) (*throw ',tag ,form))
(-> times *)
(-> (ttyin ,@body) ($progn-check$ let ((standard-input terminal-io)) $progn-check$ ,@body)
    rule-type non-terminal
    test disk-io-option)
```

```
(-> (ttyin ,@body) (progn ,@body)
    side-effect (warn-on-pos "we substituted the TTYIN by a PROGN")
    rule-type non-terminal)
(-> (ttymsg ,@args) (format terminal-io ,@(make-format-parameter args))
    test disk-io-option)
(-> ttymsg msg
    rule-type non-terminal
    side-effect (warn-on-pos "We substituted the TTYMSG by MSG"))
(-> (ttyout ,@body) ($progn-check$ let ((standard-output terminal-io)) $progn-check$ ,@body)
    rule type non-terminal
    test disk-io-option)
(-> (ttyout ,@body) (progn ,@body)
    rule-type non-terminal
    side-effect (warn-on-pos "we substituted the TTYOUT by a PROGN"))
( > tyi tyi)
(-> (tyilist ,list ,@body) ($progn-check$
                            with-input-from-string (standard-input (string (implode ,list)))
                            $progn-check$ ,@body)
    test (only-side-effect)
    rule-type non-terminal)
(-> (tyilist ,list ,@body) (let ((*tyilist-index* 0)
                                 (*tylist-string* (string (implode ,list))))
                             (with-input-from-string (standard-input *tyilist-string*
                                                                     *tyilist-index*)
                             ,@body)
                             (explode (substring *tyilist-string* *tyilist-index*))))
(-> tyo tyo)
(-> (tyoa ,x ,@pos) (tyo ,x))                    ;we ignore the 2nd parameter
(-> (tyolist ,@body) (exploden (with-output-to-string (standard-output) ,@body)))

( > (unbound) (unbound)
    test (memq (cadr *translate-stack*) '(set setq)))
(-> unbound unbound
    side-effect (note-on-pos "You must not assign the UNBOUND-value, use MAKUNBOUND instead"))
(-> (undumpatoms ,@x) ($ignore-if-possible$ (undumpatoms ,@x) nil))
(-> union union
    test eq-sets-option)
(-> union uci-union)
(-> untyi untyi)

(-> xcons xcons)

(-> (zerop ,x) ($check-predicate-result-needed$ zerop ,x)
     rule-type non-terminal)

;; some rules, which do the checking for ignore etc.
(-> ($ignore-if-possible$ ,old ,new) ,old
     test (not ignore-option))
(-> ($ignore-if-possible$ (,f ,arg) ,new) ,arg
    test (and (neq (get-function-info 'UCI-LISP f 'type) 'fsubr)
              (or (only-side-effect) (equal new arg))
              (listp arg) (neq (car arg) 'QUOTE)))
(-> ($ignore-if-possible$ (,f ,@args ,lastarg) ,new) (progn ,@args ,lastarg)
    test (and (neq (get-function-info 'UCI-LISP f 'type) 'fsubr)
              (some args #'(lambda (a) (and (listp a) (neq (car a) 'QUOTE))))
              (equal lastarg new)))
(-> ($ignore-if-possible$ (,f ,@args) ,new) (progn ,@args ,new)
    test (and (neq (get-function-info 'UCI-LISP f 'type) 'fsubr)
              (some args #'(lambda (a) (and (listp a) (neq (car a) 'QUOTE))))))
(-> ($ignore-if-possible$ ,old ,new) (**ignore-form**)
     test (ignore-possible))
(-> ($ignore-if-possible$ ,old ,new) ,new)

(-> ($check-predicate-result-needed$ ,f ,@as) (,f ,@as)
     test (no-predicate-result-needed))
(-> ($check-predicate-result-needed$ zerop ,x) (and (zerop ,x) 0))
(-> ($check-predicate-result-needed$ ,f ,a ,@rest) (and (,f ,a ,@rest) ,a)
     test (or (atom a) (no-side-effect `(,a ,@rest))))
(-> ($check-predicate-result-needed$ ,f ,a ,@rest)
```

```
    (let ((pred-arg ,a)) (and (,f pred-arg ,@rest) pred-arg)))

(-> ($check-print$ ,@whole-format-call) (,@whole-format-call)
    test (only-side-effect))
(-> ($check-print$ ,@format-call ,var) (progn (,@format-call ,var) ,var)
    test (or (atom var) (eq (car var) 'QUOTE)))
(-> ($check-print$ ,@format-call ,expr) (let ((*format-parameter* ,expr))
                                          (,@format-call *format-parameter*)
                                          *format-parameter*))

(-> ($check-nil-needed$  ,@rest) ,rest
    test (only-side-effect))
(-> ($check-nil-needed$ ,@rest) (progn (,@rest) nil))

(-> ($sort-check$  ,@rest nil ,n) ($sort-check$ ,@rest #'lexorder ,n)
    rule-type non-terminal)
(-> ($sort-check$ ,f ,@rest ,p nil) (,f ,@rest ,p)
    side-effect (if (or (atom p) (not (memq (car p) '(QUOTE FUNCTION))))
                    (warn-on-pos "The case of a NIL as the predicate isn't handeled")))
(-> ($sort-check$ ,@rest ,p ,n) ($sort-check$ ,@rest ,p t)
    rule-type non-terminal
    side-effect (if
                  (not (or (eq n 'T)
                           (and (listp n) (eq (car n) 'QUOTE))
                           (and (atom n) (not (symbolp n)))))
                  (warn-on-pos "The variable at NODUP argument place was substituted by T")))
(-> ($sort-check$ sort (copylist ,l) ,p t) (sort (union ,l) ,p))
(-> ($sort-check$ insert ,x ,l ,p t) (if (not (memq ,x ,l)) (insert ,x ,l ,p) ,l))
(-> ($sort-check$ merge ,l1 ,l2 ,p t) (union (merge ,l1 ,l2 ,p)))

(-> ($progn-check$ ,@a $progn-check$ ,@x (progn ,@y) ,@z)
    ($progn-check$ ,@a $progn-check$ ,@x ,@y ,@z)
    rule-type cyclic)
(-> ($progn-check$ ,@a $progn-check$ ,@b) (,@a ,@b))
(-> ($progn-check$ ,@rest) (,@rest)
    side-effect (note-on-pos "internal confusion with $progn-check$"))

;;now some rules, which must be present to handle the backquote stuff
(-> si:xr-bq-cons si:xr-bq-cons)
(-> si:xr-bq-list si:xr-bq-list)
(-> si:xr-bq-list* si:xr-bq-list*)
(-> si:xr-bq-append si:xr-bq-append)
(-> si:xr-bq-nconc si:xr-bq-nconc)

(eval-when (compile eval) (setq **comma-flag** nil))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; These are some grind-macros                                              ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun print-ignore-form (exp ignore)
  ())

(defprop **ignore-form** print-ignore-form si:grind-macro)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; These are some TEST or SIDE-EFFECT functions for the Translating Rules   ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun test-system-function-redefine (name)
  ;; tests whether a Zeta-Lisp system function will be redefined
  (if (fdefinedp name) (addset *system-functions-redefined* name)))

(defun correct-number-of-args-if-wrong (function arguments)
       (let ((required (get-function-info *Source-Dialect*
                                          function
                                          'numberOfArgs))
             (actual (length arguments))
             (minimum)
```

```
            (maximum)
            (call (cons function arguments)))

            ; decode the minimum and maximum number of args.
            (cond ((null required) (setq minimum 0 maximum 999999))
                  ((numberp required)(setq minimum required
                                           maximum required))
                  (t (setq minimum (car required)
                           maximum (cond ((cdr required))
                                         (t 999999)))))

           ; massage the call
           (cond ((< actual minimum)
                  (note-msg (N 1) "Too few arguments to function call")
                  (note-sexpression call)
                  (note-msg "Supplying default arguments as NIL")
                  (append call (listOfNils (- minimum actual))))
                 ((> actual maximum)
                  (sif nil (note-msg (N 1) "Too many arguments to function call")
                           (note-sexpression call)
                       (note-msg "Removing extra arguments"))
                  (firstNElements maximum call))
                 (t call))))

(defun listOfNils (n)
       ; makes a list of N nil's
       (do ((i n (1- i))
            (list nil (cons nil list)))
           ((< i 1) list)))

(defun firstNElements (n list)
       ; returns a new list with just the first N elements of the old.
       (and (plusp (1+ n))
            (cons (car list) (firstNElements (1- n) (cdr list)))))



(defun no-predicate-result-needed (&optional (stack *translate-stack*) (curr *current-sexpr*))
  ;; most UCI-LISP predicates return  not only truth values, but their argument
  ;; for this case we test, whether this result is really needed, e.g.
  ;; if we have a form like (AND (CONSP X) (..)) or (COND .. ((CONSP X) (..)) ..)
  ;; or (NOT ..), or an AND,COND,NOT or OR in one of these structures, we doesn't need
  ;; the result.
  (let ((lse (cadr stack)))
    (or
      (null stack)
      (and (eq (car lse) 'and)
           (neq (car (last lse)) curr))
      (and (eq (car lse) 'cond)
           (every (cdr lse) #'(lambda (clause) (neq (car (last clause)) curr))))
      (eq  (car lse) 'not)
      (and (memq (car lse) '(or and cond))
           (no-predicate-result-needed (cdr stack) lse)))))

(defun only-side-effect (&optional (stack *translate-stack*) (curr *current-sexpr*))
  ;; checks whether the current sexpr exists only for side effects
  (let ((lse (cadr stack))
        (lcar (caadr stack)))
    (or (null lse)
        (selectq lcar
          (progn (neq (car (last lse)) curr))
          (prog t)
          (prog1 (neq (cadr lse) curr))
          (prog2 (neq (caddr lse) curr))
          ((df de dm lambda F/:L defun defmacro let let-closed)
           (and (not (memq curr (cadr lse)))
                (neq (car (last lse)) curr)))

          (cond (not (some (cdr lse)
```

```
                        #'(lambda (cl)
                            (or (eq (car cl) curr)
                                (eq (car (last cl)) curr))))))
          (selectq (not (or (eq (car (last lse)) curr)
                            (eq (cadr lse) curr)
                            (some (cddr lse)
                                  #'(lambda (cl) (eq (car (last cl)) curr)))))))
        (and (memq lcar '(F/:L quote function))
             (let ((llse (caddr stack))
                   (llcar (caaddr stack)))
               (and (memq llcar '(map mapc))
                    (eq (cadr llse) lse))))
        (and
          (or (memq lcar '(prog1 prog2 progn lambda de df dm defun defmacro let let-closed))
              (and (memq lcar '(or and not selectq))
                   (eq (car (last lse)) curr)))
          (only-side-effect (cdr stack) lse)))))

(defun no-side-effect (expr)
  ;;checks whether the expr is a pure function!
  ;;but tests only a very small range of possiblities
  (or (atom expr)
      (and (pure-function (car expr))
           (every (cdr expr) #'no-side-effect))))

(defun pure-function (x)
  (or (get x 'pure-function)                    ;explicit defined as pure
      (get x 'locf)                             ;an access function (is always pure)
      (and (eq #/P (getcharn x (string-length x)))
           (fdefinedp x))                       ;a system defined predicate
      (memq x '(errset progn and or not quote function prog1 prog2 atom eq equal neq null
                comment arg list cons ncons list* append))))

(defun ignore-possible(&optional (stack *translate-stack*) (curr *current-sexpr*))
  (let ((lse (cadr stack))
        (lcar (caadr stack)))
    (and (or (neq lcar 'prog2)
             (neq (cadr lse) curr))
         (only-side-effect stack curr))))


(defun make-format-parameter (args)
  (loop
    for arg in args
    when (and (symbolp arg) (neq arg 'T))
     collect "~S" into str and
     collect arg into exprs
    else when (stringp arg)
          collect arg into str
         else when (numberp arg)
               when (plusp arg)
                nconc `("~" ,arg "X") into str
               else nconc `("~" ,(- arg) "%") into str
              else when (eq arg 'T)
                   collect "~%" into str
                  else when (listp arg)
                       when (eq (car arg) 'T)
                        when (numberp (cadr arg))
                         nconc `("~" ,(cadr arg) ",T") into str
                        else collect "~V,T" into str and
                             collect (cadr arg) into exprs
                       e se when (eq (car arg) 'E)
                             collect "~Q" into str and
                             collect `(function (lambda () ,@(cdr arg))) into exprs
                            else collect "~S" into str and
                                 collect arg into exprs
    finally (return (cons (format nil "~{~A~}" str) exprs))))

(defun useless-file-variable (var-name)
```

```
  (let ((ppos (string-search-char #/. *infile-name*))
        (fstart (string-reverse-search-char #/> *infile-name*)))
    (if (not fstart) (setq fstart -1))
    (if (not ppos) (setq ppos (string-length *infile-name*)))
    (let ((file-name (string-upcase(substring *infile-name*  (add1 fstart) ppos))))
      (or (string-equal var-name (string-append file-name "LSP"))
          (string-equal var-name (string-append file-name "FNS"))))))

(defun make-let-parameter (let-list)
  (loop
    for (var val) on let-list by 'cddr
    collect (list var val)))

(defun pair-pars (fas aas)
  (loop
    for fa in fas
    for aa in aas
    collect (list fa aa)))
```

RESEARCH UNIT FOR INFORMATION SCIENCE AND ARTIFICIAL INTELLIGENCE

REPORTS

ISSN 0722-9496

(# ::= Report out of print)

REP. GEN-1 WAHLSTER, Wolfgang: KI-Verfahren zur Unterstutzung der arztlichen Urteilsbildung. In: BRAUER, W. (ed.): GI-11. Jahrestagung. Berlin: Springer 1981, 568-579

REP. ANS-2 JAMESON, Anthony, WAHLSTER, Wolfgang: User Modelling in Anaphora Generation: Ellipsis and Definite Descriptions. In: Proceedings of the First European Conference on Artificial Intelligence, Orsay 1982, 222-227 #

REP. ANS-3 HOEPPNER, Wolfgang: A Multilayered Approach to the Handling of Word Formation. In: HORECKY, J. (ed.): COLING-82. Proceedings of the Ninth International Conference on Computational Linguistics, Prague. Amsterdam: North Holland 1982, 133-138 #

REP. GEN-4 CHRISTALLER, Thomas: An ATN Programming Environment. January 1982. In: BOLC, L. (ed.): The Design of Interpreters, Compilers and Editors for ATNs. Heidelberg: Springer 1983, 71-148 #

REP. GEN-5 WAHLSTER, Wolfgang: Aufgaben, Standards und Perspektiven sprachorientierter KI-Forschung. Einige Uberlegungen aus informatischer Sicht. In: BATORI, I., KRAUSE, J., LUTZ, H. D. (eds.): Linguistische Datenverarbeitung. Versuch einer Standortbestimmung im Umfeld von Informationslinguistik und Künstlicher Intelligenz. Tubingen: Niemeyer 1982, 13-24 #

REP. ANS-6 MORIK, Katharina: Differenzstudie zu fruheren sprachverarbeitenden Systemen der Bundesrepublik Deutschland. April 1982.

REP. ANS-7 NEBEL, Bernhard, MARBURGER, Heinz: Das naturlichsprachliche System HAM-ANS: Intelligenter Zugriff auf heterogene Wissens- und Datenbasen. In: NEHMER, J. (ed.): GI-12. Jahrestagung. Heidelberg: Springer 1982, 392-402 #

REP. GEN-8 CHRISTALLER, Thomas: Neuere LISP-basierte Softwaretechniken für die KI-Forschung: Dialekte, System Software und eingebettete Sprachen. In: NEHMER, J. (ed.): GI-12. Jahrestagung. Heidelberg: Springer 1982, 403-422 #

REP. ANS-9 HOEPPNER, Wolfgang: ATN-Steuerung durch Kasusrahmen. September 1982. In: WAHLSTER, W. (ed.): GWAI-82. 6th German Workshop on Artificial Intelligence. Bad Honnef, September 1982. Heidelberg: Springer 1982, 213-226 #

REP. GEN-10 WAHLSTER, Wolfgang: Naturlichsprachliche Systeme. Eine Einführung in die sprachorientierte KI- Forschung. Oktober 1982. In: BIBEL, W., SIEKMANN, J. (eds.): Kunstliche Intelligenz. Heidelberg: Springer 1982, 203-283 #

REP. GEN-11 CHRISTALLER, Thomas, METZING, Dieter: Parsing Interactions and a Multi-Level Parser Formalism Based on Cascaded ATNs. November 1982. In: SPARCK JONES, K., WILKS, Y. (eds.): Automatic Natural Language Parsing. Chichester: Horwood 1983, 46-60 #

REP. ANS-12 CHRISTALLER, Thomas, v.HAHN, Walther, HOEPPNER, Wolfgang, MARBURGER, Heinz, MORIK, Katharina, NEBEL, Bernhard, WAHLSTER, Wolfgang: Wissensbasierter naturlichsprachlicher Zugang zu unterschiedlichen Diskursbereichen mit dem KI-System HAM-ANS. In: SLAMA, R. (ed.): Workshop

Sprachverarbeitung, 8. Dezember 1982. GMD Bonn 1982, 100-135 #

REP. GEN-13 CHRISTALLER, Thomas: Ein objekt-orientierter Ansatz fur die Realisierung komplexer Kontrollstrukturen In: STOYAN, H., WEDEKIND, H. (eds.): Objektorientierte Software- und Hardwarearchitekturen Stuttgart: Teubner 1983, 300-318

REP. ANS-14 MORIK, Katharina: Marktstudie für natürlichsprachliche Zugangssysteme. Februar 1983

REP. ANS-15 WAHLSTER, Wolfgang, MARBURGER, Heinz, JAMESON, Anthony, BUSEMANN, Stephan: Over-answering Yes- No Questions: Extended Responses in a NL Interface to a Vision System. In: Proceedings of the 8th IJCAI, Karlsruhe 1983, 643-646

REP. ANS-16 HOEPPNER, Wolfgang, CHRISTALLER, Thomas, MARBURGER, Heinz, MORIK, Katharina, NEBEL, Bernhard, O'LEARY, Mike, WAHLSTER, Wolfgang: Beyond Domain-Independence: Experience with the Development of a German Language Access System to Highly Diverse Background Systems. In: Proceedings of the 8th IJCAI, Karlsruhe 1983, 588-594 #

REP. ANS-17 WAHLSTER, Wolfgang: Erklärungskomponenten als Dialogwerkzeuge. In: Office Management. Sonderheft 1983. 45-48 #

REP. GEN-18 v.HAHN, Walther: The Contribution of Artificial Intelligence to the Human Factors of Application Software. In: BLASER, A., ZOEPPRITZ, M. (eds.): Enduser Systems and their Human Factors. Heidelberg: Springer 1983. 128-138 #

REP. ANS-19 MORIK, Katharina: Demand and Requirements for Natural Language Systems. Results of an Enquiry. Extended version of a paper appeared in: Proceedings of the 8th IJCAI, Karlsruhe 1983, 647-649

REP. ANS-20 HOEPPNER, Wolfgang, MORIK, Katharina: Was haben Hotels, Straßenkreuzungen und Fische gemeinsam? - Mit HAM-ANS spricht man darüber. Extended Version of: Das Dialogsystem HAM-ANS: worauf basiert es, wie funktioniert es und wem antwortet es? In: Linguistische Berichte 88, Dezember 1983 (Themenheft "KI und Linguistik"), 3-36 #

REP. ANS-21 MARBURGER, Heinz, WAHLSTER, Wolfgang: Case role filling as a side effect of visual search. In: Proceedings of the first EACL meeting, Pisa 1983, 188-195

REP. ANS-22 MARBURGER, Heinz, NEBEL, Bernhard,: Naturlichsprachlicher Datenbankzugang mit HAM-ANS: Syntaktische Korrespondenz, naturlichsprachliche Quantifizierung und semantisches Modell des Diskursbereichs. In SCHMIDT, J.W. (ed.): Sprachen fur Datenbanken. Berlin: Springer 1983, 26-41.

REP. ANS-23 BUSEMANN, Stephan: Oberflachentransformationen bei der Generierung geschriebener deutscher Sprache. In: Neumann, B. (ed.): GWAI-83. 7th German Workshop on Artificial Intelligence, Dassel, September 1983, 90-99

REP. ANS-24 MORIK, Katharina, ROLLINGER, Claus-Rainer: The Real Estate Agent-Modelling Users by Uncertain Reasoning. English Version of: Partnermodellierung im Evidenzraum. In: Neumann, B. (ed.): GWAI-83. 7th German Workshop on Artificial Intelligence, Dassel, September 1983, 158-168

REP. ANS-25 MORIK, Katharina: Partnermodellierung und Interessenprofile bei Dialogsystemen der Kunstlichen Intelligenz. To appear in: Rollinger, C.-R. (ed.): Probleme des (Text-) Verstehens - Ansatze der Kunstlichen Intelligenz. (Sprache und Information). Tubingen 1984.

REP. ANS-26 HOEPPNER, Wolfgang, MORIK, Katharina, MARBURGER, Heinz: Talking it over: The Natural Language Dialog System HAM-ANS. To appear in: Bolc, L. (ed.): Cooperative Interactive Systems. Berlin: Springer 1984.

REP. ANS-27 BUSEMANN, Stephan: Surface Transformation during the Generation of Written German Sentences. To appear in: Bolc, L. (ed.): Natural Language Generation Systems. Berlin: Springer 1984.

REP. ANS-28 BUSEMANN, Stephan: Topicalization and Pronominalization. Extending a Natural Language Generation System. To appear in: Proceedings of the 6th ECAI, Pisa 1984.

RESEARCH UNIT FOR INFORMATION SCIENCE AND ARTIFICIAL INTELLIGENCE

MEMOS

ISSN 0722-950X

(# ::= Memo out of print)

MEMO ANS-1 HOEPPNER, Wolfgang: Eine Übersicht zur FUZZY- Programmierumgebung und den Dateizugriffen in HAM-ANS. September 1981. #

MEMO GEN-2 WAHLSTER, Wolfgang, v.HAHN, Walther: Mensch-Maschine- Kommunikation auf der Basis naturlicher Sprache. Positionspapier zur Arbeitstagung 12.-13.10.81. Oktober 1981. #

MEMO ANS-3 GNEFKOW, Wilhelm: Studien zu einer Programmierumgebung fur Augmented Transition Networks (ATN). Januar 1982.

MEMO ANS-4 JAMESON, Anthony: Documentation for three HAM-ANS Components: Ellipsis, NORMALIZE and NORMALIZE-1. November 1981. #

MEMO GEN-5 HUSSMANN, Michael, GENZMANN, Heinz: Performanzorientiertes Parsing - Ansätze zur robusten Analyse naturlicher Sprache. Februar 1982. #

MEMO ANS-6 WENDER, Herbert: Dokumentationsprinzipien im Projekt HAM-ANS. März 1982. #

MEMO ANS-7 NEBEL, Bernhard: Der Systemrahmen von HAM-ANS. März 1982. Überarbeitete Version Juni 1982.

MEMO ANS-8 BUSEMANN, Stephan: Probleme der automatischen Generierung deutscher Sprache. April 1982. #

MEMO ANS-9 MARBURGER, Heinz: Überlegungen zur Entwicklung eines deutschsprachigen Zugangssystems zu formatierten Massendaten. Mai 1982. #

MEMO ANS-10 BERGMANN, Henning: Lemmatisierung in HAM-ANS. Juni 1982. #

MEMO ANS-11 CHRISTALLER, Thomas: Konsistenzüberprufungen beim Aufbau von Wissensbasen. Juni 1982. # Überarbeitete Version erschienen in: Wahlster, W. (ed.): GWAI-82. 6th German Workshop on Artificial Intelligence. Bad Honnef, September 1982. Heidelberg: Springer 1982, 63-71. #

MEMO ANS-12 JAMESON, Anthony: The Semantics and Pragmatics of Answers: An Annotated Bibliography. Oktober 1982.

MEMO ANS-13 NEBEL, Bernhard: Ist LISP eine 'langsame' Sprache? Februar 1983. # Eine überarbeitete Version ist erschienen in: Neumann, B. (ed.): GWAI-83. 7th German Workshop on Artificial Intelligence, Dassel, September 1983, 21-30. #

MEMO ANS-14 MORIK, Katharina: Wertäußerungen und Repräsentationen von Bewertungen. März 1983.

MEMO ANS-15 O'LEARY, Mike: PEVAL: Towards an Interface between the HAM-ANS Core System and PASCAL/R Databases. April 1983. #

MEMO ANS-16 HOEPPNER, Wolfgang, MARBURGER, Heinz: Dialogsequenzen mit dem System HAM-ANS: Kommentierte Performanzbeispiele. Mai 1983. #

MEMO ANS-17 CHRISTALLER, Thomas: Eine Studie zur Portierung von LISP-Programmen. Juli 1983.

MEMO GEN-18 FLIEGNER, Michael: Überlegungen zur automatischen Schreibfehlerkorrektur fur ein KI-System. September 1983.

MEMO ANS-19 CHRISTALLER, Th.; HOEPPNER, Wolfgang: Bericht über eine Vortrags- und Informationsreise in den USA im Spätsommer 1983. Oktober 1983.

MEMO ANS-20 DANNENBERG, Rolf; NEBEL, Bernhard: Eine dynamische Speicherallokations- Strategie für UCI-LISP. November 1983.

MEMO ANS-21 BERGMANN, Henning; PAESELER, Annedore: Ansätze fur eine semantische Repräsentation zum naturlichsprachlichen Zugriff auf eine relationale Datenbank. Dezember 1983.

NEBEL, B.: ULM: EIN UCI-LISP - LISPMASCHINEN-LISP ÜBERSETZUNGS-SYSTEM

MEMO ANS-22

HAM
ANS

RESEARCH UNIT FOR
INFORMATION SCIENCE AND
ARTIFICIAL INTELLIGENCE

UNIVERSITÄT HAMBURG
Mittelweg 179
2000 HAMBURG 13

ISSN 0722-950X.