LISP 64/128

Version 1.0

Der LISP-Interpreter für den C64/C128

Peter Feldtmann (LISP 64)
Roman Bartke (LISP 128)

Erschienen in INPUT 64 4/86, 5/86 und 6/86 (`NDAMEN.LSP` in 4/87 S. 30) beim Verlag Heinz Heise GmbH.

Vorwort zu LISP 64:
In einem nostalgischen Retro-Anfall wurde dieses Handbuch in der Osterzeit 2014 getippt, dabei auf die neue Rechtschreibung umgestellt, korrekturgelesen und geTEXt. Da mir bisher noch keine vollständige Kopie der Software und des Handbuchs im Internet untergekommen ist, und es sehr schade wäre, wenn dieses Stück 64'er-Geschichte im Datennirwana verschwunden bliebe, habe ich etwas Software-Archäologie betrieben. Das Handbuch ist mit LYX 2.1.0 gesetzt und dabei wird u.a. auf die freie TrueType-Schriftart „**Pet Me 64**“ (The Ultimate Commodore Font) zurückgegriffen. Ich hoffe, dass der eine oder andere etwas Spaß mit dem Programm hat.

Vorwort zu LISP 128:
Aufgrund meiner LISP 64–Aktivitäten hat mich Peter Feldtmann über das Forum 64 (`http://www.forum64.de`) kontaktiert und die Freigabe für die Veröffentlichung seiner Arbeiten gegeben. Nun genau 30 Jahre nach der Veröffentlichung von LISP 64 ist es an der Zeit für LISP 128! Für mich war es eine Übung in der C128–Programmierung. Endlich alle Dokus verfügbar zu haben, von denen ich vor 30 Jahren nur geträmt hätte! LISP 128 ist das Ergebnis des Experiments.

Handbuch-Version: 4.0
Stand: 17.09.2016

Inhaltsverzeichnis

1 Einführung

Wenn wir den Software-Markt recht überblicken, handelt es sich bei diesem Programm des Monats und Wettbewerbsgewinner um eine Weltneuheit. Wer vor 5 Jahren prophezeit hätte, dass eine Implementation von LISP auf einem Heim-Computer überhaupt möglich ist, wäre wahrscheinlich als Spinner abgetan worden. Peter Feldtmann (Jahrgang '63) hat das LISP-System während seines Informatik-Studiums an der Uni Hamburg entwickelt. „In Ergänzung zum nüchternen Studium begann ich bald – nachdem ich am uni-eigenen Großrechner die Bekanntschaft mit der Sprache LISP gemacht hatte und sofort fasziniert war – mit der Planung und Entwicklung eines LISP-Interpreters für meinen häuslichen Rechner, den C 64. Dabei musste ich feststellen, dass Implementations-Hinweise in der Literatur recht dünn gesät und für kleinere Rechner überhaupt nicht vorhanden waren. So war ich weitgehend auf mich selbst angewiesen, und nach und nach entstand etwas, was jetzt LISP 64 heißt“ beschreibt Peter Feldtmann die Entstehungsgeschichte des Systems.

Ehe Sie die ersten Gehversuche mit dieser Sprache beginnen, ein paar Anmerkungen zu den naheliegenden Fragen:

LISP — was ist das eigentlich?

LISP — was soll das überhaupt?

LISP — woher kommt das?

LISP steht als Abkürzung für „LISt Processor“ (frei übersetzt: Listen-Verarbeitungs-Sprache) und ist eine der ältesten existierenden Programmmier-Sprachen. Sie wurde bereits 1958 am MIT (Massachusetts Institute of Technology, die Hochburg der theoretischen Computer-Forschung schlechthin) entwickelt. LISP ist sozusagen die „Mutter“ verschiedener neuer Programmier-Sprachen (Modula, Small-Talk etc.), die sich die Struktur und Systematik zunutze gemacht haben. LOGO wurde sogar in LISP geschrieben. LISP ist die Sprache, ohne die die Entwicklung der „Künstlichen Intelligenz“ nicht möglich gewesen wäre und gewinnt nach einem zwanzigjährigen Außenseiter-Dasein zunehmend auch an Breitenbedeutung.

LISP ist (meistens) eine Interpreter-Sprache wie zum Beispiel BASIC. Das System befindet sich also ständig in einer Warteschleife und wartet auf Eingaben des Benutzers, wertet diese aus, um entweder eine Fehlermeldung auszuspucken oder zur Freude des Benutzers seine Anweisungen auszuführen, dann wieder zu warten und so fort. Damit hören die Parallelen zu BASIC auch schon auf. Denn LISP ist eine „ganz andere“ Sprache. Vor allem ganz anders als BASIC. Denn:

1.1 LISP im Allgemeinen ...

In LISP gibt es keine formalen Unterschiede zwischen Programmen und Daten. Daher können Programme andere Programme erzeugen oder sich selbst verändern. Dies ist eine der Voraussetzungen für eine bequeme Programmierung selbstlernender Systeme.

In LISP sind Datenstrukturen beliebiger Tiefe möglich, ohne dass diese Struktur vorher festgelegt sein muss, wie beispielsweise bei Arrays in BASIC.

In LISP lassen sich Befehle selbst definieren. Der Interpreter behandelt nach einer Benutzer-Definition den neuen Befehl, als wär's ein Teil von ihm.

In LISP ist rekursive Programmierung möglich.

Das heißt, aus einer Funktion heraus kann ebendiese Funktion aufgerufen werden.

In LISP herrscht die „Polnische Notation". An diesem Begriff soll auch gleich einiges des oben Angepriesenen konkreter entwickelt werden:

Normalerweise schreibt man eine Funktion bzw. den Operator zwischen die Argumente. Also in Infix-Notation: `3 + 4`. Wer schon einmal ein FORTH-Programm gesehen hat, kennt die „umgekehrte polnische Notation": `3 4 +`. Der Operator steht nach den Argumenten. (Sinngemäß also Postfix-Notation, dieser Ausdruck ist aber ungebräuchlich.) Diese polnische Notation – so benannt nach einer in Polen ansässigen Logiker-Schule – nicht umgekehrt heißt dann: `+ 3 4`; der Operator steht vor den Argumenten (Präfix-Notation).

1.2 ... und Besonderen

Der Ausdruck `+ 3 4` ist fast schon ein fertiges LISP-'Programm'. Es fehlen noch die Klammern, die in dieser Sprache jeden Ausdruck einschließen: `(+ 3 4)`. Und das System kennt nicht das Zeichen „+", sondern möchte es ausgeschrieben haben: `(PLUS 3 4)`. Ergebnis: 7. Falls Sie beim gewohnten „+" bleiben möchten, definieren Sie sich diesen Befehl selbst:

```
(DE + (X Y)
  (PLUS X Y)
)
```

DE heißt „Jetzt wird eine neue Funktion definiert", danach folgt, durch Leerzeichen getrennt der Name der neuen Funktion, in diesem Fall „+". In den Klammern stehen die Argumente, die unserer Funktion beim Aufruf übergeben werden, und in der nächsten Zeile, was mit ihnen geschehen soll. Die letzte, aus Gründen der Übersichtlichkeit so allein stehende, Klammer schließt die Definition ab. Der Interpreter liefert den Namen der neuen Funktion zurück: `+`. (Wenn Sie unvermutet viele Klammern eingeben müssen, um die Definition abzuschließen, ist meist irgend etwas an der Logik faul...) Wollen Sie nur zwei Zahlen zusammenzählen, tut's nach dieser Definition der Befehl `(+ 3 4)`. (Mit der Funktion `PLUS` können eigentlich beliebig viele Argumente

verarbeitet werden, es geht also auch: `(PLUS 3 4 6 128 34)`. Das Beispiel sollte aber zunächst möglichst einfach den `DE`-Befehl klarmachen.) Das Ergebnis 7 wird übrigens an das Terminal ausgegeben, ohne dass irgendwo eine PRINT-Anweisung steht! Diese 'automatische Ausgabe' ist LISP-typisch: nach der Eingabe von `(PRINT (PLUS 3 4))` steht die Ziffer 7 zweimal untereinander auf dem Bildschirm.

Nächstes Beispiel: Rekursion wird in allen Bekannten Lehrbüchern am Beispiel der Fakultätsfunktion abgehandelt. Dieser Tradition wollen wir und auch hier nicht entziehen. Also:

Definition 1 – Die Fakultät einer natürlichen Zahl n (geschrieben $n!$) ist das Produkt aller natürlichen Zahlen von 1 bis n. So ist $7! = 1 * 2 * 3 * 4 * 5 * 6 * 7 = 5040$

Definition 2 – kürzer und rekursiv: $n! = n * (n-1)!$

Im Klartext: „Die Fakultät einer Zahl n ist das Produkt von n und der Fakultät von $n-1$“. Damit sich die Katze nicht in den Schwanz beißt, muss dem ständigen Selbstaufruf ein klares Ende gesetzt werden. Falls $n = 0$, soll gelten $n! = 1$. Diese 'Abbruchbedingung' nennt sich im Informatiker-Jargon „Rekursions-Basis“. Die vollständige rekursive Definition heißt somit:

Wenn $n = 0$, dann $n! = 1$.

Sonst immer: $n! = n * (n-1)!$

Diese Definition lässt sich leicht in eine LISP-Funktion umsetzen:

```
(DE FAKU (N)
  [COND
    [(ZEROP N) 1]
    [T (TIMES N (FAKU (SUB1 N]
  ]
)
```

In der ersten Zeile wieder der `DE`-Befehl, gefolgt vom neuen Funktionsnamen und dem zu übergebenden Argument. `COND` ist, flapsig ausgedrückt, das `IF-THEN-ELSE` von LISP, und bezieht sich auf einer Reihe von Bedingungs-Aktions-Paaren, die alle LISP-gemäß in Klammern stehen. Die erste Bedingung ist `(ZEROP N)`, umgangssprachlich „Prüfe, ob N gleich Null ist!“. Ist dies der Fall, diese Bedingung also wahr, dann soll der Wert 1 zurückgegeben werden. Ansonsten geht es zur nächsten Bedingung (eine Zeile tiefer). Dort steht im Bedingungsteil ein schlichtes `T`. Das `T` steht in LISP für den Wahrheitswert „true“ (wahr). Durch das Einsetzen von `T` im Bedingungsteil dieser Zeile wird der zugehörige Ausführungsteil immer (!) bearbeitet, weil die Bedingung eben ausdrücklich auf „wahr“ gesetzt wurde. Man hätte auch schreiben können: `N=N`, `3=3` oder einen anderen Ausdruck, der in jedem Fall wahr ist. Die darauf hin auszuwertende Funktion ist `(TIMES N (FAKU (SUB1 N)))`. `TIMES` bildet das Produkt beliebig vieler Argumente. Die Argumente sind `N` und die Funktion `(FAKU (SUB1 N))`.

Eines der Argumente ist als eine neuer Aufruf der `FAKU`-Funktion, der als Argument `(SUB1 N)` übergeben wird. `(SUB1 N)` ist wiederum ein LISP-Ausdruck und bedeutet

„Ziehe von N Eins ab“. Die letzte eckige Klammer schließt alle noch geöffneten Klammern. Das System gibt FAKU zurück und „weiß“ künftig, dass beispielsweise (FAKU 6) gleich 720 ist.

Zum Schluss noch ein Wort zu den Begriffen *Atom* und *Liste*. Dies sind die unter LISP verfügbaren Daten-Formen. Ein Atom ist wie in der Physik unteilbar, es kann ein numerisches Atom sein (eine Zahl), eine nicht durch Leerzeichen getrennte Zeichenfolge oder ein String. Eine Liste beginnt mit der obligatorischen „(“ und enthält beliebig viele Elemente. Die Elemente sind entweder Atome oder wiederum Listen. Abgeschlossen wird jede Liste mit „)“. Zum Beispiel ist

```
'(BANANE APFEL BIRNE)
```

eine Liste mit drei Elementen, jedes dieser Elemente ist ein Atom. Diesen Atomen – und das ist einer der Clous von LISP – können Eigenschaften mit bestimmten Werten zugeordnet Werden. zum Beispiel:

```
(PUTPROP 'BANANE 'FORM 'KRUMM)
```

Dadurch wird dem Atom BANANE unter der Eigenschaft FORM der Wert KRUMM zugeordnet. Dies geht – um auf die Datenstrukturen beliebiger Tiefe zurückzukommen – beliebig oft:

```
(PUTPROP 'BANANE 'GESCHMACK 'GUT)
(PUTPROP 'BANANE 'KONSISTENZ 'WEICH)
```

und kann – ohne sich um Felder, Indizes oder ähnliches Kümmern zu müssen – genauso einfach wieder abgeholt werden:

```
(GETPROP 'BANANE 'GESCHMACK)
```

ergibt:

```
GUT
```

```
(GETPROP 'BANANE 'KONSISTENZ)
```

ergibt:

```
WEICH
```

und so fort. Die geht auch in die Tiefe:

```
(PUTPROP 'BANANE 'FARBE 'GELB)
(PUTPROP 'GELB 'HAEUFIGKEIT 'OFT)
```

und entsprechend:

```
(GETPROP (GETPROP 'BANANE 'FARBE) 'HAEUFIGKEIT)
```

ergibt

```
OFT
```

Überlegen Sie einmal, wie einfach man dadurch baumstrukturierte Datenbanken oder ein Wörterbuch anlegen kann!

Das Hochkomma vor „Banane“ und so weiter ist übrigens das Kürzel für „Quote“ und bedeutet, dass der Interpreter nicht nach dem Wert von „Banane“ suchen, sondern diesen Begriff als solchen nehmen soll.

1.3 Warum LISP?

„LISP ist eine Sprache, die den an prozedurale Sprachen wie FORTRAN oder PASCAL orientierten Denkweisen wenig entgegenkommt. Bei der ersten Konfrontation mit den Elementen von LISP wird sich mancher Leser fragen, wozu das alles einmal gut sein soll. Wozu z.B. soll man die komplizierte Schachtelung von Elementen in Kauf nehmen? Die 'konventionellen' Programmiersprachen kommen ohne solche Komplikationen aus!

LISP ist für die Verarbeitung *symbolischer* Daten gedacht. Betrachtet man es unter dem Blickwinkel gewohnter, vorwiegend numerischer Aufgabenstellungen, so erscheint es bizarr. Wer mit einer 'konventionellen' Programmiersprache vertraut ist wird z.B. frachen, wie sich eine Wertzuweisung $D := 3.14 * R * R$ in LISP darstellt.

Die Antwort auf diese Frage ist zugegebenermaßen nicht ermutigend. Später verlieren solche Fragen an Wichtigkeit, aber für LISP-Anfänger sind sie höchst beunruhigend. Man steht sie Anfangsschwierigkeiten leichter durch, wenn man [...] vor Augen hat, und ahnen kann, welche Motivation den Aufbau der Sprache bestimmt haben. [...]

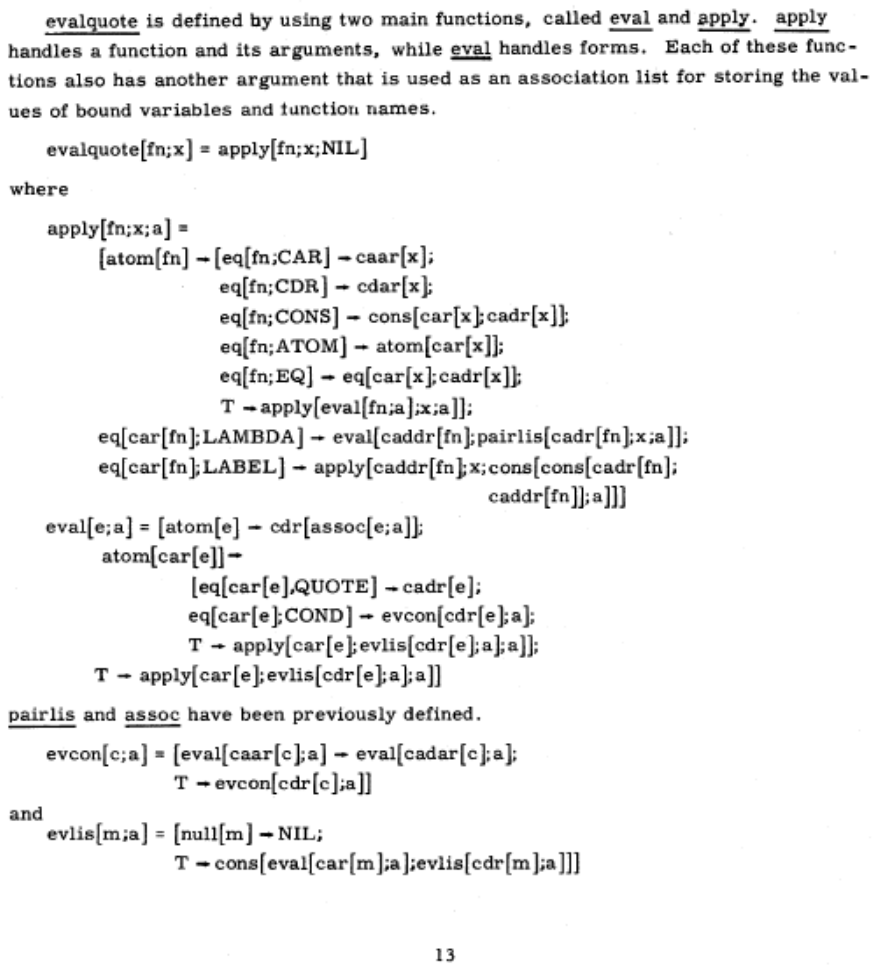
Man kann eine Programmiersprache als einen Code betrachten, in dem man Schritt für Schritt notiert, was der Rechner tun soll. LISP ist sicher etwas mehr. Es ist ein System von Ideen und einfachen Mechanismen, eine Art Mikrokosmos des Programmierens.“ [Mü85]

Und das haben auch andere festgestellt:

Alan Kay sagte in einem Interview[1][Kay04], es wäre für ihn eine große Offenbarung gewesen, als er im Studium endlich verstanden hatte, dass der Code auf der zweiten Hälfte von Seite 13 des LISP 1.5 Handbuchs [McC65] LISP in sich selbst war. Dies waren die „Maxwellschen Gleichungen“ der Software. Dies wäre die ganze Welt der Programmierung in einigen Zeilen, die man mit einer Hand verdecken könne.

[1] „Yes, that was the big revelation to me when I was in graduate school—when I finally understood that the half page of code on the bottom of page 13 of the Lisp 1.5 manual was Lisp in itself. These were “Maxwell's Equations of Software!” This is the whole world of programming in a few lines that I can put my hand over.

I realized that anytime I want to know what I'm doing, I can just write down the kernel of this thing in a half page and it's not going to lose any power. In fact, it's going to gain power by being able to reenter itself much more readily than most systems done the other way can possibly do.“



evalquote is defined by using two main functions, called eval and apply. apply handles a function and its arguments, while eval handles forms. Each of these functions also has another argument that is used as an association list for storing the values of bound variables and function names.

```
evalquote[fn;x] = apply[fn;x;NIL]

where

apply[fn;x;a] =
    [atom[fn] → [eq[fn;CAR] → caar[x];
                 eq[fn;CDR] → cdar[x];
                 eq[fn;CONS] → cons[car[x];cadr[x]];
                 eq[fn;ATOM] → atom[car[x]];
                 eq[fn;EQ] → eq[car[x];cadr[x]];
                 T → apply[eval[fn;a];x;a]];
     eq[car[fn];LAMBDA] → eval[caddr[fn];pairlis[cadr[fn];x;a]];
     eq[car[fn];LABEL] → apply[caddr[fn];x;cons[cons[cadr[fn];
                                                caddr[fn]];a]]]
eval[e;a] = [atom[e] → cdr[assoc[e;a]];
     atom[car[e]] →
          [eq[car[e],QUOTE] → cadr[e];
           eq[car[e];COND] → evcon[cdr[e];a];
           T → apply[car[e];evlis[cdr[e];a];a]];
     T → apply[car[e];evlis[cdr[e];a];a]]

pairlis and assoc have been previously defined.

evcon[c;a] = [eval[caar[c];a] → eval[cadar[c];a];
              T → evcon[cdr[c];a]]
and
evlis[m;a] = [null[m] → NIL;
              T → cons[eval[car[m];a];evlis[cdr[m];a]]]
```

13

Abbildung 1.1: Die „Maxwellschen Gleichungen“ von LISP [McC65]

Jemand greift dies wieder auf und vergleicht es mit einer Szene in der ersten Stunde seiner Physikvorlesung [Nie12].

Kann also die Beschäftigung mit LISP auf einem klassischen Computer, wie dem C64 oder dem C128, für die heutige Zeit überhaupt noch nützlich sein? Ist es nicht vertane Zeit?

Nun, LISP ist einfach zeitlos.

Dieter Müller bringt es im Vorwort zu [Mü85] auf den Punkt, wenn er sagt: „LISP knüpft an relativ alte mathematische Konzepte, insbesondere die Theorie der rekur-

siven Funktionen und den Lambda-Kalkül, an. Dies ist wohl einer der Gründe für seine Zeitlosigkeit."

Die gegenwärtige Diskussion über funktionale Programmiersprachen, man schaue z.B. auf den modernen LISP-Dialekt Clojure [WPC15], bekräftigt diese Aussage abermals.

1.4 Das System

Dies soll als Vorgeschmack auf die Beschäftigung mit LISP reichen. Wir können natürlich innerhalb des Magazins keinen LISP-Lehrgang anbieten! Falls Sie in die Sprache einsteigen wollen, müssen Sie sich (mindestens) eins der an anderer Stelle im Magazin besprochenen LISP-Bücher zu Gemüte führen.

Unsere Rolle 'beschränkt' sich darauf, Ihnen ein komplettes LISP-System zur Verfügung zu stellen; um LISP zu lernen oder – falls Sie es schon können – auf dem 64er anzuwenden. Da dieses System sehr umfangreich ist, wird LISP 64 in mehrerer Teilen veröffentlicht.

Im ersten Teil erhalten Sie den LISP-Interpreter, einige kleine Beispielprogramme, die den Einstieg erleichtern sollen, und eine richtige LISP-Anwendung: Ein Programm zum symbolischen Differenzieren. (Unmathematischer ausgedrückt: dieses Programm kann ableiten.) Das ist schon mehr als ein einsatzfähiges LISP-System.

Zu einer wirklich guten LISP-Implementation gehören außerdem noch ein TRACE-Modul (zur Fehlererkennung und Behandlung) und ein Listen-Editor, mit dem vorhandene Funktionen oder Listen komfortabel verbessert werden können. Tracer und Editor finden Sie in der nächsten Ausgabe, sowie drei Befehlserweiterungen: eine zur Mengen-, eine zur Array-Verarbeitung und `MACROS.LSP`. Letztere enthält so nützliche Dinge wie `REPEAT`, `FOR` und andere Kontrollstrukturen. Damit ist das LISP-System komplett. Als Zugabe veröffentlichen wir eine Ausgabe später zwei größere, LISP-typische Anwenderprogramme.

1.5 Abspeichern auf eigenen Datenträger

Den LISP-Interpreter selbst können Sie wie üblich auf Ihren eigenen Datenträger überspielen. Die mitgelieferten Programme (auch die Befehlserweiterung `MACROS.LSP`) sind sequentielle Files (auf Diskette als USR-Programme abgelegt). Diese werden aus dem Auswahl-Menü im Modul abgespeichert. Laden der Programme vom eigenen Datenträger (das heißt unter LISP auch: Einfügen in die `OBLIST`, also einbinden in den Interpreter) erfolgt unter LISP 64 durch:

```
(LOAD ga "name")
```

`ga` ist die Geräteadresse. Zum Beispiel:

```
(LOAD 8 "HANOI.LSP")
```

1.6 Beispielprogramme

Die „Befehlserweiterung“ `MACROS.LSP` ist unten beschrieben. Drei kleine Programme sollen den Einstieg in die LISP-Programmierung erleichtern:

1.6.1 `TOH.LSP`

`TOH.LSP` steht für das bekannte orientalische Spiel „Towers of Hanoi“. Es geht dabei darum, einen Turm aus Scheiben absteigender Größe (die kleinste Scheibe liegt oben) von einem Startfeld unter Benutzung eines Hilfsfeldes auf ein Endfeld zu schichten. Eine größere Scheibe darf niemals auf eine kleinere gelegt werden (Abb. 1.2).

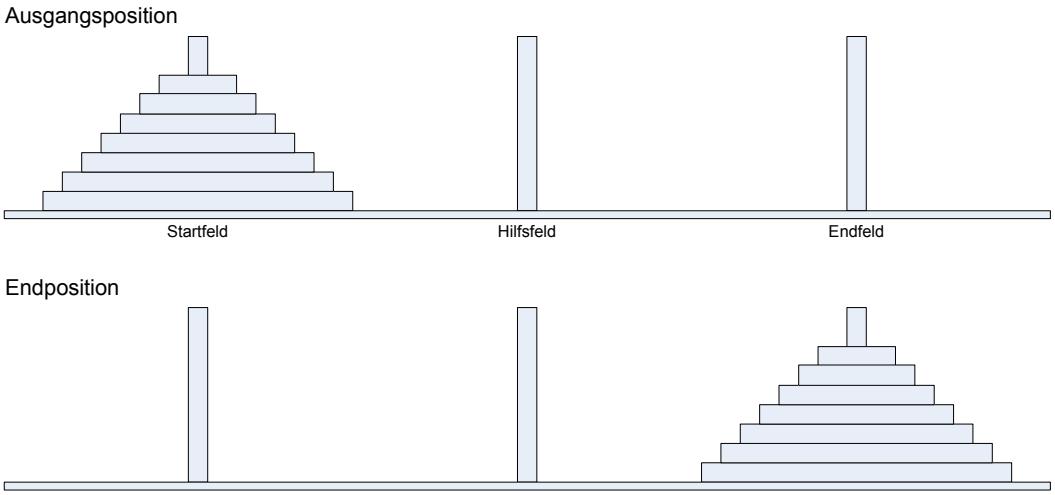


Abbildung 1.2: Türme von Hanoi

Der Theoretische Ansatz zur rekursiven Lösung dieses Problems ist in der einschlägigen Literatur schon häufig beschrieben worden, sehr allgemeinverständlich unter anderem im „c’t-Special 1 Software Knowhow“ (Heise-Verlag, 1985).

1.6.2 `HANOI.LSP`

`HANOI.LSP` behandelt das gleiche Problem, gibt aber gleichzeitig die Anzahl der Schritte aus. Interessanter ist aber – und deswegen aus zwei Programme zum selben Problem – die unterschiedliche Realisierung im Vergleich zu `TOH`. Dieses Programm wurde dem Buch „Einführung in das Programmieren in LISP“ von C.M. Hamann (de Gruyter Verlag) entnommen.

1.6.3 PERM.LSP

PERM.LSP permutiert eine Liste mit beliebig vielen Elementen. Eine Demonstration liefert die Eingabe von (DEMO). PERM selbst wird ausgeführt durch (PERM Liste), zum Beispiel (PERM '(HASE KATZE IGEL FUCHS)). PERM ruft PERM1, PERM2 und TAUSCH. PERM1 und PERM2 sind zwei sich gegenseitig aufrufende Funktionen. Dies ist zwar nicht gerade der einfachste Algorithmus, aber sehr instruktiv für die LISP-Programmierung.

1.6.4 SYMB-DIFF.LSP

SYMB-DIFF.LSP ist ein „richtiges“ Anwenderprogramm in LISP. Diese Funktion erstellt – wahlweise – die erste bis fünfte Ableitung einer beliebigen Funktion. Der Aufruf erfolgt mit (DO), dann wird hinter dem „F(X) =“ die zu differenzierende Funktion erwartet. Achten Sie bei der Eingabe auf die LISP-typische Notation und die trennenden Leerzeichen. Also nicht eingeben: **(X↑5)** sondern **(X ↑ 5)**. Außerdem gilt: INFIX-Notation (LISP untypisch), keine Punkt-vor-Strich-Regel. Zum Beispiel wird die Funktion $4X + 5X^2 + 8\sin(2X)$ eingegeben als:

```
((4 * X)+(5 * (X ↑ 2))+(8 * (SIN (2 * X]
```

Eine Erklärung der Arbeitsweise werden Sie hier vergeblich suchen. Denn wie oben schon gesagt, wir können Ihnen zwar ein LISP-System, aber keinen LISP-Lehrgang liefern.

1.7 Details der Implementation

Die Speicherplatzbelegung von LISP 64 ist in Abb. 1.3 dargestellt.

Die Speicherkonfiguration ist prinzipiell veränderbar durch folgende Adressen:

$0906 (dez. 2310) FSLTOP

$0908 (dez. 2312) DSTACK

$090A (dez. 2314) ASTACK

Da das System im RAM liegt, ist es natürlich gegen unbedachte POKE-Befehle nicht geschützt! Damit LISP 64 auch für Datasetten–Benutzer vernünftig handhabbar ist, wurde das schnelle Lade- und Speicherverfahren SuperTape im System installiert. Sie brauchen also SuperTape nicht vorher laden!

Es gelten folgende Festlegungen:

1.7.1 Zahlen, Strings und Literale

1.7.1.1 Zahlen:

32-Bit-Integer, vorzeichenbehaftet. Damit ist der Zahlenbereich von -2^{31} bis $2^{31} - 1$ darstellbar, das heißt: −2.147.483.648 bis +2.147.483.647

1.7.1.2 Strings:

in Hochkommata (") eingeschlossene Zeichenketten von maximal 128 Zeichen Länge.

1.7.1.3 Literale:

Zeichenketten bis maximal 80 Zeichen Länge, keine Leerzeichen oder syntaktische Zeichen (`;.()[]'"`) enthaltend. Literale sind zum schnellen Zugriff in einer hash-adressierten Liste organisiert – daher auch die „merkwürdige“ Anordnung bei `(OBLIST)`.

1.7.2 Systemfunktionen und -literale

Systemfunktionen und System-Literale (`VALUE`, `EXPR`, `NIL`, ...) können keine Eigenschaftslisten besitzen.

1.7.3 Garbage-Collection

Der von Zeit zur Zeit auftauchende weiße Stern in der rechten oberen Bildschirmecke ist das Zeichen des tätigen „Garbage Collectors“ `(GC)`.

1.7.4 EVAL-Lisp

LISP 64 ist ein EVAL-Lisp, das heißt: erstes Element = Funktion, Rest = Argumente.
Funktionen können sein:

- Systemfunktionen (`CAR`, `PLUS`, ...)
- `LAMBDA`-Ausdruck
- `NLAMBDA`-Ausdrücke (Argumente werden nicht ausgewertet)
- `LABEL`-Ausdrücke
- durch `DE`, `DF`, `DM` definierte Funktionen
- `VALUE` eines Literals

```
(SETQ FIRST 'CAR)
FIRST
(FIRST '(A B))
A
```

1.7.5 Definition neuer Funktionen

Beim Definieren neuer Funktionen gilt:
In der Form
`(DE Name Varlist ...)` bzw.

`(DF Name Varlist ...)`

bedeutet:

Varlist = `NIL` / Varlist = `()` – Funktion erwartet keine Argumente.

Varlist = Atom (non-NIL) – Funktion hat beliebig viele Argumente, Varlist enthält die Liste mit diesen Argumenten.

Varlist = Liste von Literalen: jedes Literal ist lokale Variable und nimmt beim Funktionsaufruf den Wert des zugehörigen Arguments an.

1.7.6 Funktionsergebnis

Funktionskörper und `COND`-Klauseln sind grundsätzlich vom `PROGN`-Typ, das heißt, Ergebnis ist der Wert der Ausführung des letzten von beliebig vielen Ausdrücken.

1.8 Fehlermeldungen

Die Fehlermeldungen werden nach dem Muster

`[Fehlermeldungs-Text][:Term]`

`[IN Name]`

erzeugt. Der Zusatz `[Term]` erscheint nur, wenn „Term“ ungleich `NIL` ist; der Zusatz `[Name]` nur, wenn der Fehler während der Ausführung einer Benutzerdefinierten Definition „`Name`“ eintritt.

Nach Ausgabe einer Fehlermeldung sowie der Erzeugung eines Programmabbruchs (durch RUN/STOP) befindet sich das LISP-System wieder in der Interpreterschleife. Lokale Variablen (von Funktionen, `LAMBDA`-Ausdrücken, `PROG`s) behalten ihre zum Zeitpunkt des Fehlerfalls gültigen Werte. Zur Fehleranalyse kann man jetzt mit der Funktion `(UNBIND)` jeweils die letztgültigen Variablenbindungen zurücknehmen, zum Beispiel eine rekursive Funktion Schritt für Schritt zurückverfolgen. Die Funktion `(RESET)` setzt alle benutzten Variablen wieder auf ihre globalen Werte zurück und sollte vor dem Neustart einer Funktion nach einem Fehlerfall ausgeführt werden.

Die Fehlermeldungen im einzelnen:

1.8.1 EMPTY STACK

Term = `NIL`. Es wird ein Zugriff auf den leeren Datenstack versucht; meist bei Aufruf einer Funktion mit fehlenden Argumenten, zum Beispiel `(CONS)`.

1.8.2 STACK OVERFLOW

Term = `NIL`. Der Datenstack ist voll, zum Beispiel bei einer nicht-terminierten Rekursion.

1.8.3 UNBOUND VARIABLE

Term = Variablen-Name. Es wird versucht, den Wert einer Variablen zu ermitteln, die weder global (durch `SETQ`) noch lokal (in Funktion, `LAMBDA`-Ausdruck, `PROG`) mit einem Wert belegt wurde.

1.8.4 UNDEFINED FUNCTION

Term = Funktionsname. Es wird versucht eine noch nicht mit `DE`, `DF` oder `DM` definierte Funktion aufzurufen.

1.8.5 NON-NUMERIC ARGUMENT

Term = Argument der numerischen Funktion. Es wird versucht, eine numerische Operation mit einem nicht-numerischen Wert (zum Beispiel `NIL`) durchzuführen, beispielsweise `(PLUS 3 'A)`.

1.8.6 READ-ERROR

Term = `NIL`. Beim Einlesen (`READ`, `READL`) eines LISP-Ausdrucks wird ein syntaktischer Fehler festgestellt, zum Beispiel `(A..B)`.

1.8.7 I/O ERROR #n

`+++ I/O ABORTED +++`

Beim Arbeiten mit den Eingabe-/Ausgabe-Funktionen (`OPEN`, `INPUT`, usw.) tritt ein Fehler auf. „`n`“ hat die Bedeutung wie im C64-Betriebssystem.

1	=	too many files open
2	=	file open
3	=	file not open
4	=	file not found
5	=	device not present
6	=	not input file
7	=	not output file
8	=	missing filename
9	=	illegal device number

Tabelle 1.1: I/O-Errors

1.8.8 MISSING PARAMETER

Term = `OPEN`. In einem `OPEN`-Befehl fehlen die nötigen Angaben, zum Beispiel Filenummer, Gerätenummer.

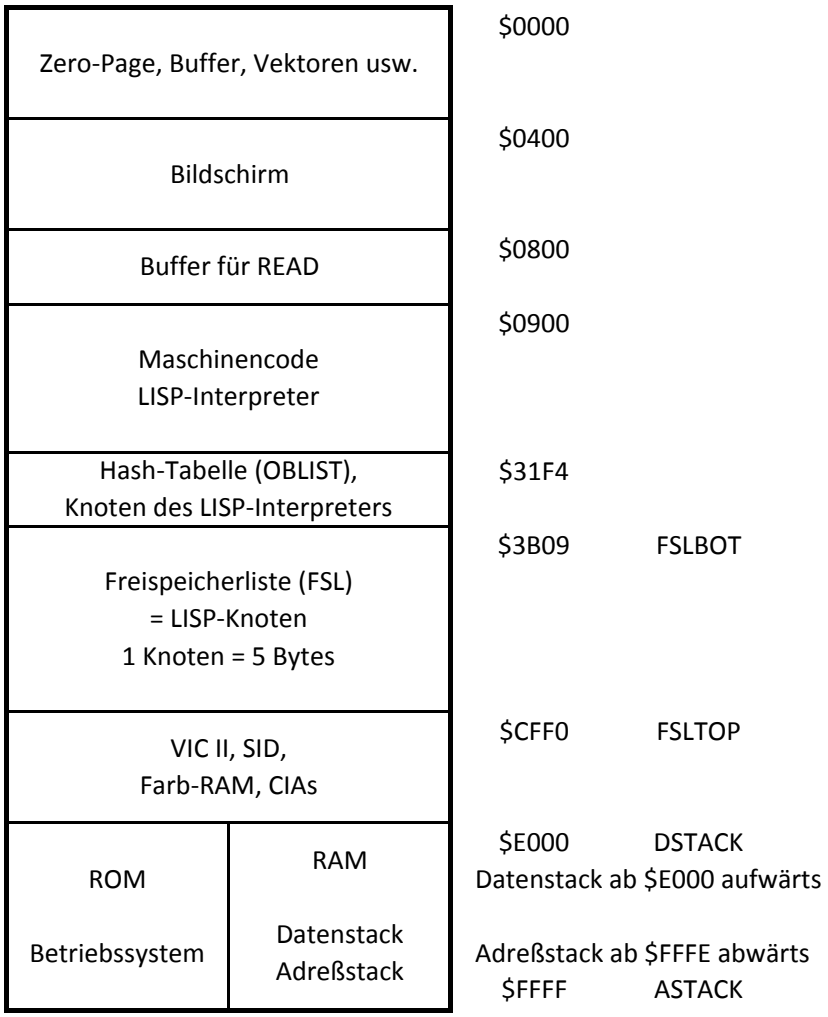


Abbildung 1.3: Speicherplatzbelegung

2 LISP 64 – Der Befehlssatz

Die Befehle selbst sind jeweils fett gedruckt, danach folgt die Erklärung und anschließend ein oder mehrere Beispiele.

*

Kommentarkennzeichnung
`(* BEISPIEL 1)`

ABS

Absolutbetrag
`(ABS -4) = 4`

ADD1

Inkrementieren (Zahl + 1)
`(ADD1 4) = 5`

AND

Und-Verknüpfung von Wahrheitswerten
`(AND NIL NIL T NIL) = NIL`
`(AND T T T) = T`

APPEND

Aneinanderhängen von Listen
`(APPEND '(A B) '(C D) '(E F)) = (A B C D E F)`

APPLY

Anwenden einer Funktion auf die Argumentliste

```
(APPLY 'PLUS '(3 4)) = 7
```

APPLY*

Anwenden einer Funktion auf die Argumente

```
(APPLY* 'PLUS 3 4) = 7
```

ASC

Ermittlung des ASCII-Codes

```
(ASC "A") = 65
```

ASSOC

Suchen (EQ-Vergleich) in Assoziationsliste

```
(ASSOC 'X '((X.A) (Y.B))) = (X.A)
```

ATOM

prüft auf atomar

```
(ATOM 'A) = T
```

CAAR

```
(CAR(CAR...))
(CAAR '((A B)C)) = A
```

CADR

```
(CAR(CDR...))
(CADR '(A B C D)) = B
```

CALL

Maschinenprogrammaufruf, A/X/Y/ST in Adressen 780–783
`(CALL 65487) = NIL`

CAR

Erstes Element einer Liste
`(CAR '(A B)) = A`

CDAR

`(CDR(CAR(...))`
`(CDAR '((A B) C)) = (B)`

CDDR

`(CDR(CDR(...))`
`(CDDR '(A B C D))=(C D)`

CDR

Liste ohne erstes Element
`(CDR '(A B C)) = (B C)`

CHAR

wandelt ASCII-Code nach Zeichen
`(CHAR 65) = "A"`

CLOSE

schließen einer mit OPEN eröffneten Datei
`(CLOSE 1) = T`

COMPL

Komplementbildung
`(COMPL 13) = -14`

COND

bedingter Ausdruck:

```
(COND
 (b1 a11 a12 ...)
 (b2 a21 a22 ...)
 ...
 (bn an1 an2 ...))
```

wenn Bedingung `bi`≠`NIL` dann `ai1 ai2`... auswerten

```
(COND ((EQ (READ) 'J) 'JA)
      (T              'NEIN))
```

CONS

konstruieren von S-Expr (steht für "symbolic expression") (Listen)
`(CONS 'A 'B) = (A.B)`
`(CONS 'A '(B C)) = (A B C)`

CONSP

prüft auf Liste = `(NOT(ATOM(...))`
`(CONSP '(A.B)) = T`

COPY

Erzeugen einer Kopie
`(COPY '(A B C)) = (A B C)`

DE

Definieren einer neuen Funktion vom EXPR-Typ

```
(DE STATE
  (X)
  (NOT(ZEROP(PEEK X))))
= STATE
```

DEFPROP

Ablegen eines Wertes (`1982`) unter einer Eigenschaft (`BAUJAHR`) auf der Eigenschaftsliste des Atoms (`AUTO`)

FEXPR! (s.a. `PUTPROP`)

```
(DEFPROP AUTO BAUJAHR 1982)
= 1982
```

DF

Definieren einer neuen Funktion vom FEXPR-Typ

```
(DF TEST
  (X Y)
  (CONS (LIST X)(LIST Y)))
= TEST
```

DIFFERENCE

Differenzbildung

```
(DIFFERENCE 10 3) = 7
```

DIR

Inhaltsverzeichnis der Diskette lesen

```
(DIR)
```

DISK

Befehl zur Floppy senden

```
(DISK "S:FILE1") = T
```

DM

Definieren einer MACRO-Funktion

```
(DM NCONS L
  (LIST 'CONS
    (CADR L) NIL))
= NCONS
```

EQ

prüft auf Atom- oder Zeigergleichheit

```
(EQ 'A 'A) = T
(EQ 4 5) = NIL
```

EQUAL

prüft auf Strukturgleichheit

```
(EQUAL '(A B C) '(A B C)) = T
```

ERROR

Fehlermeldungen erzeugen, Rücksprung auf top-level

```
(ERROR '(DIVISION DURCH 0))
```

EVAL

evaluieren eines S-Expr

```
(EVAL '(PLUS 3 4)) = 7
```

EXIT

Verlassen des LISP-Systems

```
(EXIT)
```

GC

Garbage Collector aufrufen, ermittelt Anzahl freier Knoten

`(GC) = `*`Anzahl`*

GETCHAR

ein Zeichen Lesen

`(GETCHAR)`

`= "`*`Zeichen`*`"`

GETDEF

ermitteln der Funktionsdefinition (property) eines Atoms

```
(GETDEF TEST)
= (DF TEST (X Y)(CONS ...))
```

GETPROP

Ermitteln eines Wertes (`1982`) einer Eigenschaft (`BAUJAHR`) auf der Eigenschaftsliste eines Atoms (`AUTO`)

```
(GETPROP 'AUTO 'BAUJAHR)
= 1982
```

GETPROPLIST

ermitteln der Eigenschaftsliste eines Atoms

```
(GETPROPLIST 'AUTO)
= (BAUJAHR 1982 ...)
```

GO

in `PROG`: Sprung auf Marke

```
(GO LOOP1)
```

GREATERP

Zahlenvergleich auf größer
`(GREATERP 5 3) = T`

HBYTE

höherwertiges Byte einer Adresse
`(HBYTE 1030) = 4`

INPUT

legt alle Eingaben (`READ`, `GETCHAR` usw.) auf das mit `(OPEN 1 ...)` eröffnete File
`(INPUT 1)`

LAST

letztes Element einer Liste
`(LAST '(A B C)) = C`

LBYTE

niederwertiges Byte einer Adresse
`(LBYTE 1030) = 6`

LENGTH

Anzahl der Elemente einer Liste
`(LENGTH '(A B C)) = 3`

LESSP

Zahlenvergleich auf kleiner
`(LESSP 3 5) = T`

LINE

Maximal-Zeilenlänge für Ausgaben

```
(LINE 80) = 80
```

LIST

bilden einer Liste

```
(LIST 'A 'B 'C) = (A B C)
```

LOAD

Laden einer mit `SAVE` abgespeicherten LISP-Datei von Atom-Eigenschaftslisten. SuperTape ist integriert!

1 = Laden von Kassette

7 = C64: SuperTape-Laden von Kassette oder RAM–Disk; C128: RAM–Disk

8 = Laden von Diskette

```
(LOAD 8 "FILE1") = (...)
(LOAD 7 "FIL*") = (...)
```

LOGAND

logisches (bitweises) Und

```
(LOGAND 12 4) = 4
```

LOGOR

logisches (bitweises) Oder

```
(LOGOR 8 4) = 12
```

LOGXOR

logisches (bitweises) Exklusiv-Oder

```
(LOGXOR 12 4) = 8
```

MAP

sukzessives Anwenden einer Funktion auf eine Liste, deren CDR, CDDR und so weiter...

```
(MAP 'PRINT '(A B C))
= (A B C)
(B C)
(C)
NIL
```

MAP2CAR

sukzessives Anwenden einer zweistelligen Funktion auf die jeweils stellengleichen Element-Paare zweier Listen

```
(MAP2CAR 'CONS '(A B) '(1 2))
= ((A.1) (B.2))
```

MAPC

sukzessives Anwenden einer Funktion auf die Elemente einer Liste

```
(MAPC 'PRINT '(A B C))
= A
B
C
NIL
```

MAPCAN

entspricht dem Ausdruck `(APPLY 'NCONC (MAPCAR ...))`

```
(MAPCAN '(LAMBDA (X) (LIST X X)) '(A B C))
= (A A B B C C)
```

MAPCAR

sukzessives Anwenden einer Funktion auf die Elemente einer Liste; Ergebnisse in einer Liste sammeln

```
(MAPCAR '(LAMBDA (X) (LIST X X)) '(A B C))
= ((A A) (B B) (C C))
```

MAPLIST

sukzessives Anwenden einer Funktion auf eine Liste, deren CDR, CDDR usw.; dabei Ergebnisse in einer Liste sammeln

```
(MAPLIST 'PRINT '(A B C)) =
(A B C)
(B C)
(C)
((A B C) (B C) (C))
```

MEMBER

Element in einer Liste suchen und wenn gefunden, Liste ab Element als Ergebnis

```
(MEMBER 'B '(A B C)) = (B C)
```

MINUS

Vorzeichenwechsel

```
(MINUS 4) = -4
```

MINUSP

Zahlenvergleich auf negativ

```
(MINUSP -4) = T
```

MSG

'Message' ausgeben: T gibt CR (Carriage Return, wörtlich „Wagenrücklauf“) aus, Zahlen geben Leerstellen (spaces) an

```
(MSG T "TEXT" 4 "TEXT2")
=<CR>
TEXT␣␣␣␣TEXT2NIL
```

NCONC

destruktives Aneinanderketten von Listen

```
(NCONC '(A B) '(C D) '(E F))
= (A B C D E F)
```

NCONC1

destruktives Verlängern von Listen
`(NCONC1 '(B C) 'A) = (B C A)`

NORMAL

Aufhebung eines `INPUT`/`OUTPUT`-Befehls: Eingabe von Tastatur, Ausgabe auf Bildschirm
`(NORMAL) = T`

NOT

Negation
`(NOT NIL) = T`

NTH

Liste ab N-tem Element
`(NTH '(A B C D) 3) = (C D)`

NULL

prüft auf leere Liste = NIL
`(NULL '()) = T`

NUMBERP

prüft auf numerisches Atom
`(NUMBERP 4) = T`

OBLIST

Ausgabe der Objektliste mit allen Atomnamen
`(OBLIST)`

OPEN

Eröffnen einer Eingabe- oder Ausgabedatei
`(OPEN 1 4)` für Drucker
`(OPEN 1 8 2 "TEST,S,W") = T`

OR

Oder-Verknüpfung von Wahrheitswerten
`(OR F F T F) = T`
`(OR NIL NIL F) = NIL`

OUTPUT

leitet alle Ausgaben auf ein mit `(OPEN 1 ...)` eröffnetes File um
`(OUTPUT 1)`

PACK

erzeugen eines neuen Atomnamens oder Strings durch Zeilenverkettung
`(PACK '(A B C D)) = ABCD`
`(PACK '("TE","ST",44)) = "TEST44"`

PDEF

übersichtliche Ausgabe einer Funktionsdefinition = `(PP (GETDEF ...))`

```
(PDEF TEST) =
(DF TEST
  (X Y)
  (CONS(LIST X)
    (LIST Y)))
NIL
```

PEEK

Inhalt einer Speicherzelle
`(PEEK 1) = 54`

PLUS

Addition beliebig vieler Werte

```
(PLUS 1 2 3 4) = 10
```

POKE

Adresse mit Wert besetzen

(POKE 650 128) = 128

PP

Übersichtliche Ausgabe einer Liste („pretty print“)

```
(PP '(A B(C D(E)))) =
(A B
  (C D
    (E)))
NIL
```

PRIN1

Ausgabe ohne CR, Strings ohne Anführungszeichen

```
(PRIN1 '(A "B" C)) =
(A B C)NIL
```

PRINC

Ausgabe ohne CR und ohne syntaktische Zeichen

```
(PRINC '(A ("B")((C)))) =
(A B C)NIL
```

PRINL

Ausgabe ohne syntaktische Zeichen, CR als Zeilenabschluss

```
(PRINL '(A(B C)(D))) =
A B C D
NIL
```

PRINT

Ausgabe eines S-Expr, CR als Zeilenabschluss

```
(PRINT '(A "B" (C))) =
(A "B" (C))
```

PROG

PROGram-Anweisung:
- lokale Variable (X)
- beliebig viele Anweisungen
= evaluierbare S-Expr
- Marken = atomare Namen

Sprung auf Marken mit GO, Verlassen des PROGs mit RETURN

```
(PROG(X) M1
  (SETQ X
    (READ))
  (COND((EQ X
       'QUIT)
      (RETURN 'OK)))
  (PRINT(EVAL X))
  (GO M1))
```

PROG1

beliebig viele S-Expr evaluieren, Wert des ersten zurückgeben

```
(PROG1(CAR '(A B))
  (SETQ X
    (CDR X))
  (CDR '(A B))) =
A
```

PROGN

beliebig viele S-Expr evaluieren, Wert des letzten zurückgeben

```
(PROGN(CAR '(A B))
```

```
  (SETQ X
    (CDR X))
  (CDR '(A B))) =
(B)
```

PUTPROP

Ablegen eines Wertes (81) unter einer Eigenschaft (BAUJAHR) auf der Eigenschaftsliste eines Atoms (AUTO); EXPR!

```
(PUTPROP 'AUTO 'BAUJAHR 81) =
81
```

QUOTE

Quotierung: Unterdrückung der Auswertung eines S-Expr

```
(QUOTE (A B C)) = (A B C)
(QUOTE A) = 'A = A
```

QUOTIENT

Quotientenbildung, siehe auch REMAINDER

```
(QUOTIENT 12 4) = 3
```

RANDOM

Zufallszahlenerzeugung im angegebenen Bereich

```
(RANDOM 10 20) = 14
```

READ

Einlesen eines S-Expr

```
(READ) = (A B C)
```

READCH

Einlesen eines Zeichens

```
(READCH) = "A"
```

READL

Einlesen beliebig vieler S-Expr und sammeln in einer Liste
`(READL) = ((A B)(C D E))`

REMAINDER

Rest bei Quotientenbildung
`(REMAINDER 13 3) = 1`

REMOB

Entfernen von Atomen aus der Objektliste
`(REMOB CASR TEST TEST1)`

REMOVE

Entfernen eines Elements aus einer Liste
`(REMOVE 'A '(A B A C A A D)) = (B C D)`

REMPROP

Entfernen einer Eigenschaft des Atoms
`(REMPROP 'AUTO 'BAUJAHR) = BAUJAHR`

RESET

nach Fehlerfall/Break: Zurücksetzen der Variablen auf ihre globale Werte
`(RESET) = NIL`

RETURN

Verlassen eines `PROG`s mit Wertrückgabe
`(PROG X (RETURN 'OK)) = OK`

REVERSE

Umkehrung der Reihenfolge der Listenelemente

```
(REVERSE '(A B C)) = (C B A)
```

RPLACA

destruktives Ersetzen des CARs einer Liste

```
(RPLACA '(A B) 'C) = (C B)
```

RPLACD

destruktives Ersetzen des CDRs einer Liste

```
(RPLACD '(A B) '(C)) = (A C)
```

SASSOC

suchen (EQUAL-Vergleich) in einer Assoziationsliste

```
(SASSOC '(A B)
   '(((A B)
     C)
    ((D E)
       F)))
= ((A B) C)
```

SAVE

Abspeichern der Eigenschaftslisten von Atomen in einer Liste

```
(SAVE 8 "FILE" '(FABFIB ...))
(SAVE 7 "FILE2" FUNCS)
```

SET

Wertzuweisung an Variable

```
(SET 'X '(A B)) = (A B)
```

SETQ

Wertzuweisung an Variable ohne deren Evaluierung =`(SET(QUOTE ...)...)`
`(SETQ X '(A B)) = (A B)`

SPACES

Ausgabe von Leerzeichen
`(SPACES 10)`

ST

Einlesen des Floppy-Status
`(ST) = (0 OK 0 0)`

STRINGP

prüft auf Zeichenkette
`(STRINGP "ABCD") = T`

SUB1

Dekrementieren (Zahl - 1)
`(SUB1 4) = 3`

TAB

Tabulatorfunktion
`(TAB 10)`

TERPRI

'terminal printing': CR ausgeben
`(TERPRI) =`
<CR>
`NIL`

TIMES

Produktbildung
`(TIMES 2 3 4) = 24`

UNBIND

bei Fehlerfall/Break: Aufhebung der letzten Variablen-Bindungen
`(UNBIND) = NIL`

UNPACK

Auflösung eines Atomnamens oder Strings in einzelne Zeichen
`(UNPACK 'ABCD) = (A B C D)`
`(UNPACK "A4") = ("A" "4")`

WAITCHAR

wartet, bis Zeichen eingegeben wird
`(WAITCHAR) = "A"`

ZEROP

Zahlenvergleich mit 0
`(ZEROP 0) = T`

EXPR

Der unter der Eigenschaft `EXPR` auf der Eigenschaftsliste eines Atoms abgelegte `LAMBDA`-Ausdruck gilt als Funktionskörper, siehe auch `DE`

F

`F` = False: Wahrheitswert 'falsch' = `NIL`

FEXPR

Der unter der Eigenschaft `FEXPR` auf der Eigenschaftsliste eines Atoms abgelegte `NLAMBDA`-Ausdruck gilt als Funktionskörper, siehe auch `DF`

LABEL

benennt eine Funktion, die rekursiv benutzt wird, temporär (nur für die Ausführung) mit einem Namen in der Form

`((LABEL` *`Name LAMBDA-Ausdruck`*`)` *`Argumente`*`)`

wobei im LAMBDA-Ausdruck

`(LAMBDA` *`Variablenliste Ausdruck1 Ausdruck2 ...`*`)`

unter den Ausdrücken die Funktionsaufrufform

`(`*`Name ...`*`)`

vorkommen darf (Rekursion).

LAMBDA

In der Form

`((LAMBDA` *`Variablenliste Ausdruck1 Ausdruck2 ...`*`)`

`Argument1 Argument2 ...``)`

werden zunächst die Argumente evaluiert und mit den Variablen der Variablenliste gebunden (wenn die Variablenliste ein Atom $\neq$ `NIL` ist, wird eine Liste mit allen evaluierten Argumenten an dieses Atom gebunden).

Danach werden die Ausdrücke ausgewertet; der Wert des letzten Ausdrucks ist der Wert des gesamten `LAMBDA`-Ausdrucks.

MACRO

Der unter der Eigenschaft `MACRO` auf der Eigenschaftsliste eines Atoms abgelegte `NLAMBDA`-Ausdruck gilt als Funktionskörper vom MACRO-Typ; sie auch `DM`.

NIL

bezeichnet sowohl ein Atom [`(ATOM NIL) = T`] als auch die leere Liste [`(NULL NIL) = T`] und den Wahrheitswert 'falsch'.

NLAMBDA

In der Form
`((NLAMBDA` *`Variablenliste Ausdruck1 Ausdruck2 ...`*`)`
`Argument1 Argument2 ...``)`
werden die Argumente **nicht** ausgewertet, alles weitere siehe `LAMBDA`.

T

Wahrheitswert 'true' = wahr

'

`'Ausdruck`
ist eine abkürzende Schreibweise für `(QUOTE` *`Ausdruck`*`)`

[]

Die 'Superklammern' erleichtern das Eingeben von verschachtelten Klammerausdrücken: wird ein Ausdruck mit einer Klammer '`[`' begonnen, so schließt eine spätere Klammer '`]`' alle noch offenen runden Klammen '`(`' bis zur '`[`' (einschließlich). Wird eine Klammer '`]`' ohne zugehörige '`[`' benutzt, dann werden alle noch offenen Klammern '`(`' geschlossen. Beispiele:

```
(COND[(ATOM X)(RETURN(LIST X(CAR X]
[T (PRINT(CDR X])
```

wird zu

```
(COND((ATOM X)(RETURN(LIST X(CAR X))))
(T (PRINT(CDR X))))
```

```
(A B (C D (E)(F (G H]
```

wird zu

```
(A B (C D (E)(F (G H))))
```

Taste F5

Druckvorgang anhalten

Taste F7

Druckvorgang fortsetzen

RUN/STOP-Taste

'BREAK': Evaluierungsprozess abbrechen und Rücksprung auf LISP-Top-Level.

3 LISP 128 – Erweiterungen

LISP 64 ist auf den C128 portiert worden. Dabei ist es um folgende eingebaute Funktionen erweitert worden, die in `config.i` jeweils für die Assemblierung aktiviert werden können.

BANK

Setzt die Speicherbank für `PEEK`, `POKE` und `CALL`.

`(BANK` ***n*** `) mit` $n \in \{0..15\}$

CFGAREA

Liefert die Adresse des Konfigurationsbereichs (Config–Area) (siehe auch Abschnitt 7.2).

`(CFGAREA) = 7208`

Ein Anwendungsbeispiel:

```
(DE COPYRIGHT ()
  (BANK 0)
  (POKE (DEC "03EE") 2)
  (CALL (PLUS (CFGAREA) 99))
)
```

COLD

Löst einen Kaltstart des Interpreters aus.

`(COLD)`

DEC

Umrechnung von HEX nach DEC.

`(DEC "FCE2") = 64738`

HEX

Umrechnung von DEC nach HEX.

```
(HEX 64738) = "FCE2"
```

ED

Aufruf des eingebauten Editors. Verlassen des Editors mit RUN/STOP. Der Editor ist eine angepasste Version von *MrED* (erschienen in [Mil87]).

```
(ED) = NIL
```

Editing Commands

1. The CRSR keys work pretty much the way you would expect. Use them to move (the cursor) or scroll (the window) up, down, left and right. You cannot CRSR out of the range of entered text.
2. The INST/DEL key is used to insert and delete one character at a time just the way it is in BASIC.
3. The RETURN key is non-destructive and does nothing but advance you to the next line.
4. F1 deletes whole lines to the right and can be used to join two lines as well.
5. F2 inserts a line, ie. a CHR$(13). It can be used to split a line in two as well as open up space.
6. F3 and F4 allow vertical page scrolling. F3 takes you down and F4 up one screen at a time.
7. F5 and F6 control horizontal paging. F5 moves you one screen to the right and F6 one screen to the left.
8. RUN/STOP exits to LISP.

Going up or down to a new line always places you at the beginning of it, ie. full window left.

Status Line

Das Format der Editor–Statuszeile ist wie folgt:

```
C000/L00000 M00000 T00000/00000/00000
|     |       |        |        |     |
Col   Line   MemFree Bottom/Current/Top
                     address pointers into the text buffer
```

Dabei ist *MemFree* = *Top* - *Current*

MONITOR

Aufruf des Betriebsystem-Monitors des C128. Mit X RETURN kann der Monitor wieder verlassen werden.

```
(MONITOR) = NIL
```

4 Tracer, Editor, Mengen-Verarbeitung, Array-Handling

Mit diesen Erweiterungen wird der LISP-Interpreter zu (fast) kompletten LISP-Paket. Alle Files sind LISP-Programme (natürlich nur unter dem LISP 64 Interpreter lauffähig), die man aus dem Magazin heraus als sequentielle Dateien auf Kassette oder Diskette abspeichern kann. Geladen – und das heißt unter LISP 64 auch: in den Interpreter eingebunden – werden die einzelnen Erweiterungen mit

```
(LOAD ga "fn")
```

`ga` ist die Geräte-Adresse (1, 7 oder 8), `fn` der Programmname (zum Beispiel `MACROS.LSP`).

4.1 MACROS.LSP

Durch diese Funktionen wird die weitgehend dem Standard 1.5 entsprechende Fassung des LISP 64 verschiedenen 'modernen' Versionen angenähert. Vor allem die Einführung weiterer Kontrollstrukturen (`FOR`, `IF`, `REPEAT`, `SELECTQ`, `WHILE`) erleichtert die Programmierung. Das 'klassische' LISP kennt nämlich nur `COND` und die Rekursion, und das ist, gelinde gesagt, ziemlich ungewohnt.

Makros sind Funktionen, die zunächst einen evaluierbaren S-Expr (für "symbolic expression") erzeugen, dessen Ergebnis dann den Funktionswert darstellt. Zum Definieren einer Funktion vom MACRO-Typ dient der Befehl `DM` (Define Macro).

Beim Aufruf eines Makros bildet der *gesamte* Makro-Aufruf einschließlich des Makro-Namens das Argument.

Beispiel:

```
(DM XCONS L
  (LIST 'CONS
    (CAR(CDDR L))
    (CADR L)))
```

Nach dem Aufruf `(XCONS 'A 'B)` wird der gesamte Ausdruck an die Variable `L` gebunden, das heißt `L = (XCONS 'A 'B)`. Danach ergibt sich als Zwischenergebnis der Ausdruck `(CONS 'B 'A)`, dessen Evaluierung schließlich das Ergebnis `(B.A)`.

Unter der *Expansion* eines Makros versteht man dessen *Ersetzen* durch den erzeugten S-Expr, zum Beispiel mit den Funktionen `RPLACA`, `RPLACD` usw.

4.1.1 Die Makros des Files MACROS.LSP

MACRO-EXPANSION

globale Variable: wenn = `T`, dann Makros expandieren

(EXPAND)

setzt `MACRO-EXPANSION = T`

(NO-EXPAND)

setzt `MACRO-EXPANSION = NIL`, das heißt Makros werden nicht expandiert.

4.1.2 Makro, Erläuterung, Beispiel, Expansion

Ähnlich, wie bei den Interpreter-Befehlen, gilt die Reihenfolge: Befehl, Erläuterung, Beispiel und Expansion.

DECR

Dekrementieren einer Variablen

```
(DECR X)
= (SETQ X (SUB1 X))
```

INCR

Inkrementieren einer Variablen

```
(INCR X)
= (SETQ X (ADD1 X))
```

PUSH

Wert auf einer Liste ablegen

```
(PUSH X 'Y)
= (SETQ X (CONS 'Y X))
```

POP

Wert von einer Liste holen, Liste um dieses Element verkürzen

```
(POP X)
= (PROG1 (CAR X)
         (SETQ X (CDR X)))
```

NEQ

Vergleich auf non-eq

```
(NEQ X Y)
= (NOT(EQ X Y))
```

MCONS

'multiple CONS'

```
(MCONS 'A 'B 'C)
= (CONS 'A (CONS 'B 'C))
= (A B.C)
```

NCONS

CONS Wert mit NIL

```
(NCONS 'A)
= (CONS 'A NIL)
= (A)
```

XCONS

vor dem CONS Werte vertauschen

```
(XCONS 'A 'B)
= (CONS 'B 'A)
= (B.A)
```

FUNCTION

= QUOTE, aber lesbarer, zum Beispiel bei

```
(APPLY (FUNCTION CONS) '(A B))
= (A.B)
(FUNCTION CAR)
= (QUOTE CAR)
```

F:L

verkürzende Schreibweise für `(FUNCTION(LAMBDA(...))`

```
(F:L(X)(LIST X X))
= (FUNCTION(LAMBDA(X)(LIST X X)))
```

Q:L

verkürzende Schreibweise für `(QUOTE(LAMBDA(...))`

```
(Q:L(Y)(PRINT Y))
= '(LAMBDA (Y) (PRINT Y ))
```

LOCAL

Einführung lokaler Variablen, die mit `NIL` vorbesetzt werden

```
(LOCAL (A B)(PRINT(LIST A B)))
= ((LAMBDA (A B)(PRINT(LIST A B))) NIL NIL)
```

LET

Einführung lokaler Variablen, die mit den zugeordneten Werten besetzt werden

```
(LET((A 3)(B 4))(PLUS A B))
= ((LAMBDA (A B)(PLUS A B)) 3 4)
= 7
```

IF

IF-THEN-ELSE: erster S-Expr = Bedingung, zweiter S-Expr = THEN-Teil, restliche S-Expr = ELSE-Teil

```
(IF (LESSP X 4) (PRINT '<VIER) (PRINT '>=VIER))
= (COND((LESSP X 4)(PRINT '<VIER))
        (T         (PRINT '>=VIER)))
```

WHILE

WHILE-Schleife: solange die Bedingung (= erster S-Expr) $\neq$ `NIL` ist, restliche S-Expr evaluieren

```
(WHILE (LESSP X 20)
       (PRINT X)
       (INCR X))
```

```
= (PROG NIL LOOP
    (COND((LESSP X 20)
        (PRINT X)
        (INCR X))
      (T(RETURN NIL)))
    (GO LOOP))
```

REPEAT

Die S-Expr wiederholt so oft evaluieren, wie der Wert der ersten S-Expr (der nur einmal evaluiert wird) angibt

```
(REPEAT 4 (PRINT X)(INCR X))
= (PROG(N)
    (SETQ N 4) LOOP
    (COND((ZEROP N)
        (RETURN NIL))
      (T(PRINT X)
        (INCR X)))
    (SETQ N
      (SUB1 N))
    (GO LOOP))
```

FOR

FOR-NEXT-Schleife: Laufvariable (lokal), Startwert, Endwert, Schrittweite = 1 (oder -1, wenn Startwert > Endwert) beliebig viele S-Expr als Anweisungen

```
(FOR I 10 20
  (TAB I)(PRINT I))
= (PROG(I)
    (SETQ I 10) LOOP
    (COND((GREATERP I 20)
        (RETURN NIL))
      (T(TAB I)
        (PRINT I)))
    (SETQ I
      (ADD1 I))
    (GO LOOP))
```

SELECTQ

Fallunterscheidung ('CASE')

```
(SELECTQ X
  (4 (PRINT 'VIER))
  (5 (PRINT 'FUENF))
  ((6 7 8) (PRINT '>5&<9))
  (9 (PRINT 'NEUN))
  (PRINT '>9))
= (COND((EQ X 4)
      (PRINT 'VIER))
    ((EQ X 5)
      (PRINT 'FUENF))
    ((MEMBER X
       '(6 7 8))
      (PRINT '>5&<9))
    ((EQ X 9)
      (PRINT 'NEUN))
    (T(PRINT '>9)))
```

4.2 TRACER.LSP

Wie der Name schon andeutet, dient `TRACER.LSP` zum Testen von LISP-Programmen. Mit vier Befehlen werden verschiedene Trace-Modi ein- oder ausgeschaltet:

4.2.1 (TRACE Function) / (TRACE Function1 Function2 ...)

setzt den Tracer auf eine oder mehrere selbstdefinierte Funktionen an, zum Beispiel `(TRACE PERM1 PERM2)`. Bei jedem Aufruf einer „geTRACEten" Funktion erscheint die Meldung '`ENTERING` ***`Function`***' sowie in Klammern die der Funktion übergebenen Argumente. Beim Verlassen der Funktion gibt der Tracer die Meldung '`EXITING` ***`Function`*** `=` ' sowie das Funktionsergebnis aus.

Angewandt auf die Funktion TOH.LSP (Towers of Hanoi), stellt sich der Ablauf des Tracing folgendermaßen dar:

1. TRACE.LSP laden
2. TOH.LSP laden
3. Den Tracer auf beide Funktionen von TOH.LSP ansetzen:
   `(TRACE TOH BEWEGE)`

4. Funktion aufrufen, die Eingabe von `(TOH 3)` liefert das folgende Trace-Protokoll, das die rekursiven Aufrufe von BEWEGE mit den jeweiligen Argumenten deutlich macht. (Die Benutzereingaben sind am Zeilenanfang mit `>` kenntlich gemacht.)

```
>(LOAD 8 "TRACER*")
(TRACE UNTRACE EVTRACE PRINTENTRY PRINTENTRY1 PRINTEXIT SINGLE-STEP
NO-SINGLE-STEP SINGLE-STEP-V TRACFNS)
>(LOAD 8 "TOH*")
(TOH BEWEGE)
>(TOH 3)
(LEGE SCHEIBE VON LINKS NACH RECHTS)
(LEGE SCHEIBE VON LINKS NACH MITTE)
(LEGE SCHEIBE VON RECHTS NACH MITTE)
(LEGE SCHEIBE VON LINKS NACH RECHTS)
(LEGE SCHEIBE VON MITTE NACH LINKS)
(LEGE SCHEIBE VON MITTE NACH RECHTS)
(LEGE SCHEIBE VON LINKS NACH RECHTS)
T
>(TRACE TOH BEWEGE)
(TOH BEWEGE)
>(TOH 3)
 ENTERING TOH [3]
  ENTERING BEWEGE [LINKS,RECHTS,MITTE,3]
   ENTERING BEWEGE [LINKS,MITTE,RECHTS,2]
    ENTERING BEWEGE [LINKS,RECHTS,MITTE,1]
     ENTERING BEWEGE [LINKS,MITTE,RECHTS,0]
     EXITING  BEWEGE = T
(LEGE SCHEIBE VON LINKS NACH RECHTS)
     ENTERING BEWEGE [MITTE,RECHTS,LINKS,0]
     EXITING  BEWEGE = T
    EXITING  BEWEGE = T
(LEGE SCHEIBE VON LINKS NACH MITTE)
    ENTERING BEWEGE [RECHTS,MITTE,LINKS,1]
     ENTERING BEWEGE [RECHTS,LINKS,MITTE,0]
     EXITING  BEWEGE = T
(LEGE SCHEIBE VON RECHTS NACH MITTE)
     ENTERING BEWEGE [LINKS,MITTE,RECHTS,0]
     EXITING  BEWEGE = T
    EXITING  BEWEGE = T
   EXITING  BEWEGE = T
(LEGE SCHEIBE VON LINKS NACH RECHTS)
   ENTERING BEWEGE [MITTE,RECHTS,LINKS,2]
    ENTERING BEWEGE [MITTE,LINKS,RECHTS,1]
     ENTERING BEWEGE [MITTE,RECHTS,LINKS,0]
     EXITING  BEWEGE = T
(LEGE SCHEIBE VON MITTE NACH LINKS)
     ENTERING BEWEGE [RECHTS,LINKS,MITTE,0]
     EXITING  BEWEGE = T
    EXITING  BEWEGE = T
(LEGE SCHEIBE VON MITTE NACH RECHTS)
    ENTERING BEWEGE [LINKS,RECHTS,MITTE,1]
     ENTERING BEWEGE [LINKS,MITTE,RECHTS,0]
     EXITING  BEWEGE = T
(LEGE SCHEIBE VON LINKS NACH RECHTS)
     ENTERING BEWEGE [MITTE,RECHTS,LINKS,0]
     EXITING  BEWEGE = T
    EXITING  BEWEGE = T
   EXITING  BEWEGE = T
  EXITING  BEWEGE = T
 EXITING  TOH = T
T
```

4.2.2 (SINGLE-STEP)

schaltet den Einzelschritt-Modus ein, das heißt, nach jedem 'ENTERING...' wartet der Tracer auf das Drücken einer (beliebigen) Taste.

4.2.3 (NO-SINGLE-STEP)

macht (SINGLE-STEP) wieder rückgängig.

4.2.4 (UNTRACE Function) / (UNTRACE Function1 Function2 ...)

hebt den TRACE-Befehl für die jeweilige Funktion auf.

Der TRACE-Befehl verändert die angegebene Funktion. Deshalb muss natürlich vor dem eventuellen Abspeichern einer Funktion der TRACE-Modus durch (UNTRACE ...) aufgehoben werden.

4.3 ARRAYS.LSP

Bisweilen kann es auch in LISP nützlich sein, mit größeren Datenfeldern (Arrays) zu arbeiten – BASIC- oder PASCAL-ähnlich.

Die Darstellung von eindimensionalen Feldern ist sehr einfach: es sind Listen, deren geordnete Elemente die Feldkomponenten bilden.

Der Zugriff auf das n-te Element geschieht mit

```
(CAR (NTH L N))
```

das Schreiben mit

```
(RPLACA (NTH L N) E)
```

Mehrdimensionale Felder kann man in einer Weise darstellen, indem die Listenelemente wiederum Listen sind und so fort. Dabei wird jedoch der Zugriff auf ein einzelnes Feldelement sehr umständlich und unübersichtlich. Die folgenden Funktionen nehmen dem Anwender diesen Teil der Programmierarbeit ab.

4.3.1 (ARRAY N I Dim1 Dim2 ... Dimn)

ARRAY definiert ein n-dimensionales Feld mit dem Bezeichner N und initialisiert alle Feldelemente mit dem Wert I. Das Feld wird in der Eigenschaftsliste des Bezeichners unter der Eigenschaft ARRAY abgelegt.

```
(ARRAY A 0 3 2 2)
```

definiert das Array mit dem Bezeichner `A` als dreidimensional (*`Dim1`*) mit einer Tiefe von jeweils zwei in der zweiten und dritten Dimension. (Statt `A` könnte der Bezeichner natürlich auch ein sinnvoller Name sein, wie RAEUME oder dergleichen...)
Der Befehl

```
(GETPROP 'A 'ARRAY)
```

macht dies sichtbar, er gibt zurück:

```
(((0 0) (0 0)) ((0 0) (0 0)) ((0 0) (0 0)))
```

4.3.2 (LOD N Index1 Index2 ... Indexn)

`LOD` ermittelt das durch die Indizes bestimmte Feldelement des Feldes `N`. Werden mehr Indizes angegeben als vorhanden, liefert `LOD NIL`. Werden weniger Indizes angegeben als die Dimensionierung erfordert, wird auf eine größere Feldstruktur („Feldvektor“) zugegriffen:

```
(LOD A 1 1 1) = 0
```

aber

```
(LOD A 1 1) = (0 0)
```

und

```
(LOD A 1) = ((0 0)(0 0))
```

und folgerichtig

```
(LOD A) = (((0 0) (0 0)) ((0 0) (0 0)) ((0 0) (0 0)))
```

gesamtes Feld `A`.

4.3.3 (STO N V Index1 Index2 ... Indexn)

`STO` (store) schreibt den Wert `V` an die durch die Indizes bestimmte Stelle des Feldes `N`; der alte Wert des Feldelementes geht verloren. Auch hier gilt: werden weniger Indizes Angegeben als die Dimensionierung erfordert, so wird an eine größere Feldstruktur zugewiesen. Der Wert von `STO` ist `V`. Zum Beispiel:

```
(STO A 55 1 1 1) = (55 0)
```

ändert das Feld `A` folgendermaßen

```
(((55 0) (0 0)) ((0 0) (0 0)) ((0 0) (0 0)))
```

auch feststellbar durch

```
(LOD A 1 1) = (55 0)
```

aber

```
(STO A 55 1 1) = (55 (0 0))
```

für Feld `A`. Inhalt von `A` danach:

```
((55 (0 0)) ((0 0) (0 0)) ((0 0) (0 0)))
```

4.4 SETS.LSP

Mengen sind in LISP Listen, in denen jedes Element höchstens einmal vorkommt:
`(A B C D)` ist eine Menge mit vier Elementen,
`(A B A D)` ist keine Menge (siehe aber `MAKESET`).

4.4.1 Mengenprädikate

(MEM1 M1 M2)

`MEM1` prüft, ob mindestens ein Element der Menge `M1` in der Menge `M2` enthalten ist. Zum Beispiel:

```
(MEM1 '(A B C) '(X B Y)) = (B Y)
(MEM1 '(A B) '(C D E)) = NIL
```

(SETEQ M1 M2)

`SETEQ` prüft auf Mengengleichheit von `M1` und `M2`.

```
(SETEQ '(A B C) '(B A C)) = T
(SETEQ '(A B C) '(B C D)) = NIL
```

(SUBSETP M1 M2)

`SUBSETP` prüft, ob `M1` eine Untermenge von `M2` ist. (Das heißt, ob alle Elemente von `M1` auch in `M2` enthalten sind.)

```
(SUBSETP '(A B C) '(A B C D)) = T
(SUBSETP '(A B) '(D B C A X)) = T
(SUBSETP '(A B C) '(A B D E)) = NIL
```

4.4.2 Mengenfunktionen

(ATTACH X L)

ATTACH platziert X destruktiv (unter Verwendung von RPLACA/RPLACD!) an den Anfang von L.

```
(ATTACH 'D '(A B C)) = (D A B C)
(ATTACH 'A '()) = (A)
```

(DREMOVE X L)

DREMOVE entfernt alle Vorkommen von X in der Liste L destruktiv.

```
(DREMOVE 'A '(A B A C D)) = (B C D)
```

(ENTER X M)

Wenn X noch kein Element der Menge M ist, wird es hinzugefügt, ansonsten bleibt M unverändert.

```
(ENTER 'C '(A B D E)) = (C A B D E)
(ENTER 'C '(A B C D)) = (A B C D)
```

(INSERT X N L)

INSERT fügt das Element X vor dem N-ten Element der Liste (Menge) L ein.

```
(INSERT 'C 3 '(A B D E)) = (A B C D E)
(INSERT 'A 2 '(A B C)) = (A A B C)
```

(INTERSECTION M1 M2)

Bildung des Mengendurchschnitts.

```
(INTERSECTION '(A B C D) '(C D E F)) = (C D)
(INTERSECTION '(A X D C) '(C A Y D)) = (A D C)
```

(MAKESET L)

MAKESET macht aus einer Liste L eine Menge, indem mehrfach vorkommende Werte nur einmal aufgenommen werden.

```
(MAKESET '(A B B A C D)) = (B A C D)
(MAKESET '(A B A A)) = (B A)
```

(REMOVE X L)

REMOVE entfernt alle Vorkommen von X in der Liste L, indem eine neue Liste ohne das Element X aufgebaut wird (siehe auch DREMOVE).

```
(REMOVE 'A '(A B A C D A)) = (B C D)
(REMOVE '(A B) '(A (A B) C (A B) D)) = (A C D)
```

(SUBSET FN M)

SUBSET bildet eine Untermenge aus den Elementen von M, die der Prädikatsfunktion FN genügen.

```
(SUBSET 'NUMBERP '(A 1 B 2 44 C)) = (1 2 44)
(SUBSET 'ATOM '(A (A B) C (D E) D)) = (A C D)
```

(SYMM-DIFF M1 M2)

Bildung der Mengendifferenz zwischen M1 und M2. (Trotz des Namens ist es jedoch keine 'symmetric difference'!)

```
(SYMM-DIFF '(A B C D E) '(B D)) = (A C E)
(SYMM-DIFF '(A B C) '(A)) = (B C)
```

(UNION M1 M2)

Bildung der Vereinigungsmenge von M1 und M2.

```
(UNION '(A B C D) '(A D E B C)) = (A D E B C)
(UNION '(A B C) '(C D)) = (A B C D)
```

4.5 EDITOR.LSP

Dieser in LISP geschriebene Editor ist ein sogenannter Listen-Editor, das heißt, er dient dem Editieren und Modifizieren beliebiger S-Expr (Listen). Solche Listen sind dabei auch Funktions-Definitionen (die entweder nicht 'laufen' oder geändert werden sollen), Atom-Eigenschaften oder Variablen-Werte. Mit dem Aufruf des Editors wird ein besonderer EDIT-Modus betreten. Der Editor meldet sich mit dem Prompt *ED*, es gelten folgende Regeln:

- Es können mehrere Editier-Kommandos in einer Zeile eingegeben werden.
- Editier-Kommandos ohne Argumente brauchen nicht geklammert zu werden, es genügt die Angabe des Kommando-Namens.
- Nach Abarbeitung eine Zeile mit Editier-Befehlen erfolgt nur dann eine Ausgabe, wenn Ausgabebefehle (P und PP) darunter sind.

- Zahlen haben im EDIT-Modus eine besondere Funktion, da sie zur Anwendung bestimmter Editier-Kommandos führen (siehe unten).

Der EDIT-Modus wird durch die Eingabe von `OK` verlassen.

4.5.1 Befehle zum Aufruf des Editors

(EDIT X)

Allgemeine Editier-Funktion für beliebigen S-Expr. Zum Beispiel:
`(EDIT '(A B (C D) E))`

(EDITF FN)

Editieren der Funktion `FN`. `EDITF` sucht in der Eigenschaftsliste von `FN` nach der Eigenschaft `EXPR` bzw. `FEXPR` und editiert sie. Im Ergebnis wirkt `EDITF` wie `DE` bzw. `DF`.

`(EDITF FAK)`

`(EDITF EXPT)`

(EDITP X Y)

`EDITP` editiert die Eigenschaft (property) `Y` des Bezeichners `X` und wirkt insgesamt wie ein `PUTPROP`.

`(EDITP PUNKT XYZ-KOORD) = (3 4 1)`

(EDITV X)

`EDITV` editiert den Wert einer Variablen `X` und entspricht somit `(EDITP X VALUE)`.

(EDFNS)

`EDFNS` ist eine Variable, die alle Editier-Funktionen sowie deren Hilfsfunktionen enthält.

Im folgenden ist unter der Bezeichnung „aktuelle Liste“ bzw. „aktueller S-Expr“ der gerade bearbeitete S-Expr zu verstehen.

Es gibt drei Klassen von Editier-Kommandos:

Ausgabe-Kommandos: Kommandos, die den aktuellen S-Expr oder anderes ausgeben. Diese Kommandos sind ohne Klammern einzugeben; Ausnahme: `(E X)`

„Zeiger“-Kommandos: Kommandos, die nicht den aktuellen S-Expr verändern, sondern im S-Expr ab- bzw. aufsteigen und so die „Aufmerksamkeit“ des Editors auf bestimmte Teile des S-Expr richten. Ebenfalls ohne Klammern einzugeben; Ausnahme: `(FI X)`

Modifikationskommandos: Kommandos, die den aktuellen S-Expr modifizieren (unter Verwendung der LISP-Funktionen `RPLACA`, `RPLACD`, `NCONC`). Diese Kommandos werden *mit* Klammern eingegeben; Ausnahme: `UNDO`

Alle Änderungen während des Editierens lassen sich mit `UNDO` rückgängig machen.

Falls der Prompt des Editors nicht mehr erscheint, ist er wahrscheinlich aufgrund einer Fehlbedienung 'ausgestiegen'! Ein Interpreter-Warmstart mit RUN/STOP + RESTORE hilft dann weiter.

In den folgenden Beispielen wird als aktuelle Liste stets angenommen:

```
(A B (C D E) F G H)
```

```
(LOAD 8 "EDIT*")
(NX FIND FI B : LO LI R REPL RO BO BI
RI EXPT E <- N A CONC DEL UNDO OUT ADD
P@ BACK G P P& H EDFNS EDIT EDITF
EDITV EDITP)
(EDIT '(A B (C D E) F G H))
(A B & F G H)
*ED*: P
(A B & F G H)
*ED*:
```

4.5.2 Ausgabekommandos

P

`P` druckt den aktuellen S-Expr, wobei alle nicht-atomaren Elemente durch das Zeichen `&` ersetzt werden.

```
*ED*: P
(A B & F G H)
```

PP

`PP` erzeugt ein „pretty print“ des aktuellen S-Expr.

```
*ED*: PP
(A B
  (C D E) F G H)
```

P@

wie `PP`

(E X)

E evaluiert den S-Expr X und druckt das Ergebnis.

```
*ED*: (E (PLUS 1 2 3 4))
10
```

4.5.3 „Zeigerkommandos“

n

n steht für einen numerischen Wert, der bewirkt:

bei $n = 0$: Rücksprung zum Listenanfang, das heißt: die Liste, in der der gegenwärtige aktuelle S-Expr enthalten ist, wird zur neuen Liste.

bei $n > 0$: Das n-te Element der aktuellen Liste wird zum neuen aktuellen S-Expr.

bei $n < 0$: Das n-te Element der aktuellen Liste, vom Listen*ende* aus *rückwärts* gezählt, wird zum neuen aktuellen S-Expr.

Bezogen auf die oben angegebene Beispiel-Liste, kann dies folgendermaßen aussehen:

Erst die ganze Liste betrachten:

```
*ED*: P
(A B & F G H)
```

Auf den dritten S-Expr (= das dritte Listenelement) zeigen lassen und diesen ausgeben

```
*ED*: 3 P
(C D E)
```

Zurück zum Anfang

```
*ED*: 0 P
(A B & F G H)
```

Vorletzten S-Expr betrachten (wäre nicht zuvor das 0-Kommando gegeben worden, wäre der aktuelle S-Expr weiterhin (C D E))!

```
*ED*: -2 P
G
```

Erneut zurück zum Anfang, letztes Element auswählen und anzeigen

```
*ED*: 0 -1 P
H
```

NX

meint NEXT, das heißt, wenn ein *n*-Kommando gegeben wurde, wird nun das n+1-te Element zum aktuellen S-Expr.

```
*ED*: P
```

```
(A B & F G H)
*ED*: 4 P
F
*ED*: NX P
G
```

←

← führt so oft das 0-Kommando aus, bis der Anfang der ursprünglich zu editierenden Liste erreicht ist.

```
*ED*: P
(A B & F G H)
*ED*: 3 2 P
D
*ED*: ← P
(A B & F G H)
```

(FI X)

FI (Find) sucht den S-Expr X im aktuellen S-Expr. Wird er gefunden, so wird die Liste, in der X enthalten ist, zur neuen aktuellen Liste. Wird X nicht gefunden, bleibt das FI-Kommando ohne Wirkung.

```
*ED*: P
(A B & F G H)
*ED*: (FI D)
(C D E)
*ED*: 0 P
(A B & F G H)
*ED*: (FI Z)
(A B & F G H)
```

4.5.4 Modifikations-Kommandos

(Die Beispiele sind meist mit UNDO abgeschlossen, um beim Ausprobieren nicht jedes Mal die Test-Liste neu eingeben zu müssen. Normalerweise ist der beendende UNDO-Befehl natürlich unsinnig!)

(*n*)

n ist, wie bei den Zeigerkommandos, eine ganze Zahl. Es gilt:

bei $n = 0$: keine Wirkung

bei $n > 0$: Das n-te Element der aktuellen Liste wird gelöscht (aus der Liste entfernt).

bei $n < 0$: Das n-te Element der aktuellen Liste, vom Listenende aus rückwärts gezählt, wird gelöscht.

```
*ED*: P
(A B & F G H)
*ED*: (2) P
(A & F G H)
*ED*: (-2) P
(A & F H)
*ED*: (3) (3) P
(A &)
*ED*: UNDO
```

(*n* E1 E2 ... Ei)

Lösch- und Insert-Kommando, es gilt:

bei $n = 0$: Die S-Expr *E1* bis *Ei* werden am Anfang der aktuellen Liste eingefügt.

bei $n > 0$: Das n-te Element der aktuellen Liste wird gelöscht und dafür die S-Expr *E1* bis *Ei* eingesetzt.

bei $n < 0$: Das n-te Element der aktuellen Liste, vom Listenende aus rückwärts gezählt, wird gelöscht und durch die S-Expr *E1* bis *Ei* ersetzt.

```
*ED*: P
(A B & F G H)
*ED*: (2 X)(-2 Y)P
(A X & F Y H)
*ED*: (1 (CHAR X) Z (CDR X)) P
((CHAR X) Z & X & F Y H)
*ED*: (0 A B)P
(A B & Z & X & F Y H)
*ED*: UNDO
```

(A E1 E2 ... Ei)

`A` (after) fügt die S-Expr *E1* bis *Ei* hinter das aktuelle Listenelement ein.

```
*ED*: P
(A B & F G H)
*ED*: 4 (A X Y Z)P
(A B & F X Y Z G H)
*ED*: 2 (A (CAR X)) P
(A B & & F X Y Z G H)
*ED*: UNDO
```

(B E1 E2 ... Ei)

`B` (begin) fügt die S-Expr *E1* bis *Ei* vor dem ersten Element der aktuellen Liste ein.

```
*ED*: P
(A B & F G H)
*ED*: (B X Y)P
(X Y A B & F G H)
*ED*: UNDO
```

(N E1 E2 ... Ei)

`N` (NCONC) fügt die S-Expr *E1* bis *Ei* am Ende der aktuellen Liste ein (mit der Funktion `NCONC`).

```
*ED*: P
(A B & F G H)
*ED*: (N X Y)P
(A B & F G H X Y)
*ED*: UNDO
```

(: E) / (: (E1 E2 ... Ei))

: ersetzt den aktuellen S-Expr durch *E* bzw. die Liste der S-Expr *E1* bis *Ei*.

```
*ED*: P
(A B & F G H)
*ED*: 3 (: X) 0 PP
(A B X F G H)
*ED*: UNDO
*ED*: 3 (: (X Y)) 0 PP
(A B
  (X Y) F G H)
*ED*: UNDO
```

(R E1 E2)

R (replace) ersetzt *alle* Vorkommen von *E1* im aktuellen S-Expr durch *E2*.

```
*ED*: P
(A B & F G H)
*ED*: (R G X)P
(A B & F X H)
*ED*: (3 A X A A Y)P
(A B A X A A Y F X H)
*ED*: (R A Z)P
(Z B Z X Z Z Y F X H)
*ED*: UNDO
```

(BI *n1 n2*)

BI (both in) setzt das *n1*-te bis *n2*-te Element der aktuellen Liste in Klammern.

```
*ED*: P
(A B & F G H)
*ED*: (BI 1 3)P
((A B &) F G H)
*ED*: (BI 2 3)PP
((A B
  (C D E))
  (F G) H)
*ED*: UNDO
```

(BO *n*)

BO (both out) entfernt die Klammern um das *n*-te Element der aktuellen Liste, wenn es kein Atom ist.

```
*ED*: P
(A B & F G H)
*ED*: (BO 3)P
(A B C D E F G H)
*ED*: UNDO
```

(LI *n*)

LI (left in) setzt vor das *n*-te Element und nach dem letzten Element der aktuellen Liste eine Klammer.

```
*ED*: P
(A B & F G H)
*ED*: (LI 2)PP
(A(B(C D E) F G H))
*ED*: UNDO
```

(LO *n*)

LO (left out) entfernt die Klammer vor dem *n*-te Element der aktuellen Liste, wenn es nicht-atomar ist.

```
*ED*: PP
(A B
  (C D E) F G H)
*ED*: (LO 3)P
(A B C D E)
*ED*: UNDO
```

Achtung: Die Elemente *n*+1, *n*+2 usw. gehen verloren!

(RI *n1 n2*)

RI (right in) verschiebt, wenn das *n1*-te Element der aktuellen Liste nicht-atomar ist, die schließende Klammer des *n1*-ten Elements hinter dessen *n2*-tes Element. Die Beispiele mögen dies näher verdeutlichen.

```
*ED*: PP
(A B
  (C D E) F G H)
*ED*: (RI 3 2)PP
(A B
  (C D) E F G H)
*ED*: (RI 3 1)PP
(A B
  (C) D E F G H)
*ED*: UNDO
```

(RO *n1 n2*)

RO (right out) verschiebt, wenn das *n1*-te Element der aktuellen Liste nicht-atomar ist, dessen schließende Klammer hinter das *n2*-tes Element der aktuellen Liste.

```
*ED*: PP
```

```
(A B
  (C D E) F G H)
*ED*: (RO 3 5)PP
(A B
  (C D E F G) H)
*ED*: (RO 3 4)PP
(A B
  (C D E F G H))
```

UNDO

UNDO macht alle vorgenommenen Listen-Modifizierungen rückgängig und springt an den Anfang der ursprünglich zu editierenden Liste (genauer: einer Kopie der Liste zurück).

4.5.5 Beispiel-Sitzung mit dem Editor

Die bekannte Fakultäts-Funktion sei wie folgt fehlerhaft definiert:

```
(DE FAK
  (X Y Z)
  (COND(ZEROP X 0)
    (T(TIMES(X)
        (FAK(ADD1 Z X))))))
```

Richtig muss es heißen:

```
(DE FAK
  (X)
  (COND((ZEROP X)
        1)
    (T(TIMES X
        (FAK(SUB1 X))))))
```

Im folgenden soll gezeigt werden, wie dieses Ziel durch schrittweises Anwenden der Editier-Funktionen erreicht werden kann.

Betreten des EDIT-Modus:

```
(EDITF FAK)
((X Y Z) &)
*ED*:
```

Variablen-Liste korrigieren:

```
*ED*: (1 (X))P
((X) &)
*ED*: 2 P
(COND & &)
*ED*: 2 P
(ZEROP X 0)
```

Erstes und zweites Element klammern:

```
*ED*: (BI 1 2)P
((ZEROP X) 0)
```

Löschen der `0` und ersetzen durch `1`:

```
*ED*: (-1 1) P
((ZEROP X) 1)
```

Eine Stufe zurück:

```
*ED*: 0 P
(COND & &)
*ED*: -1 P
(T &)
*ED*: -1 P
(TIMES & &)
*ED*: 2 P
(X)
```

Löschen und Ersetzen durch `X`:

```
*ED*: (: X)P
X
*ED*: 0 P
(TIMES X &)
*ED*: 3 2 P
(ADD1 Z X)
```

Löschen des `Z`:

```
*ED*: (2)P
(ADD1 X)
```

Ersetzen des `ADD1` durch `SUB1`:

```
*ED*: (R ADD1 SUB1)P
(SUB1 X)
*ED*: 0 P
(FAK &)
```

Zurück zum Anfang:

```
*ED*: ← P
((X) &)
```

Fertig:

```
*ED*: OK
FAK
```

Das Protokoll könnte dann folgendermaßen weitergehen:

```
(PDEF FAK)
(DE FAK
  (X)
  (COND((ZEROP X)
        1)
    (T(TIMES X
        (FAK(SUB1 X))))))
NIL
(FAK 5)
120
```

(Natürlich würde jeder vernünftige Mensch ein derart kurzes Programm einfach neu eingeben. Aber es geht ja um die Handhabung des Editors.)

4.6 PRINT-FILE.LSP

PRINT-FILE.LSP ist ein Utility, das die Handhabung vor Ein-/Ausgabe-Operationen erleichtert. Außerdem ist es sehr instruktiv für die Benutzung der I/O-Befehle (OPEN, INPUT, OUTPUT, CLOSE usw.) von LISP 64.

(PRINT-FILE gn "name")

Das File "name" wird vom Gerät gn (1=Kassette; 7=C64: SuperTape oder RAM–Disk, C128: RAM–Disk; 8=Diskette) gelesen und auf den Drucker (Gerätenummer 4) ausgegeben.

(PRINT-ON-SCREEN gn "name")

Wie PRINT-FILE, aber die Ausgabe erfolgt auf den Bildschirm.

5 Die Lisp-Oldies: EXPERTE.LSP und ELIZA.LSP

Lästermäuler bezeichnen die beiden hier vorgestellten LISP-Programme als „LISP-Oldies“. Ganz falsch ist das nicht – das berühmte Dialogprogramm ELIZA und „selbstlernende“ Expertensysteme sind wirklich die klassischen Beispiele der Anwendung sogenannter KI-Prinzipien, der Grundregeln der „Künstlichen Intelligenz“ also.

5.1 EXPERTE.LSP

Sogenannte Expertensysteme dienen der Darstellung und Verarbeitung von „Wissen“ über bestimmte Sachgebiete. So gibt es große Systeme, die chemische Analysen auswerten können, Krankheiten diagnostizieren oder geologische Karten auf Erdölvorkommen hin untersuchen.

Unser LISP-Beispielprogramm ist natürlich bescheidener als seine „großen Schwestern“. Es soll nur die allgemeinen Grundlagen einer bestimmten Form von Wissensaufbereitung demonstrieren, nämlich die Darstellung von Wissen als *Fakten*, *Regeln* und *Hypothesen*.

Fakten sind einfache Tatsachen oder wahre Aussagen, zum Beispiel:

- Es regnet.
- Ein Vogel kann fliegen.
- Alle geraden Zahlen sind durch zwei teilbar.

Regeln bestehen aus einem „Wenn“-Teil (den Prämissen) und einem „Dann“-Teil (den Konklusionen), zum Beispiel:

- Wenn es regnet, dann muss ich einen Schirm mitnehmen.
- Wenn ein Tier ein Vogel ist, dann kann es fliegen, hat Federn und legt Eier.
- Wenn die Spannung an der Basis eines NPN-Transistors 5 Volt beträgt und der Transistor leitet, dann beträgt die Spannung am Emitter 0,5-0,6 Volt.

Die Funktion der Regeln bei einer Deduktion (Ableitung vom Allgemeinen zum Besonderen) oder Diagnose (siehe unten) ist folgende: sind alle Prämissen einer Regel als Fakten bekannt, also wahr, dann sollen auch die Konklusionen dieser Regel als wahr gelten und in die Liste der Fakten aufgenommen werden.

Hypothesen schließlich sind Aussagen, die sich erst noch aus dem Zusammenspiel der Fakten und Regeln als wahr oder falsch erweisen müssen.

Zum Programm

Nach dem Starten mit `(DO)` befindet man sich im Dialog-Modus: hier kann man – in (nahezu) natürlicher Sprache – dem System Aufträge erteilen („Mache eine Diagnose“), Fragen stellen („Hast Du Regel 3 benutzt?“), Daten ausgeben lassen („Drucke alle Regeln“) und nicht zuletzt die Regeln definieren.

Die Eingaben brauchen nicht geklammert zu werden (LISP-untypisch). Achtung: ein Punkt als Satzabschluss ist nicht erlaubt (da der Punkt ein spezielles Syntaxzeichen ist); schließt eine Eingabe mit einem Doppelpunkt ab, so wird eine weitere Eingabezeile zur Verfügung gestellt, um längere Texte zu ermöglichen.

Die Kommandos

Für die folgende Beschreibung der Kommandos von `EXPERTE.LSP` gelten folgende Vereinbarungen:

- die Reihenfolge ist Kommando, Erklärung, Beispiel
- alternative Eingabemöglichkeiten zum Befehl sind (in Klammern eingeschlossen) hinter dem Kommando angegeben; diese Klammern müssen natürlich nicht mit angegeben werden
- Leerzeichen und Doppelpunkte müssen, wenn sie angegeben sind, unbedingt mit eingegeben werden
- ein Sternchen (`*`) steht für einen beliebigen Text, statt `WIE *` kann zum Beispiel eingegeben werden `WIE HAST DU DAS GEMACHT`. (Für Tautologen: natürlich kann auch ein `*` statt des `*` eingegeben werden...)

WENN : erste Prämisse

Leitet die Definition einer Regel ein, indem die erste Prämisse angegeben wird.

```
WENN : DAS TIER KANN FLIEGEN
```

UND (UND WENN) : nächste Prämisse

Hat eine Regel mehr als eine Prämisse, werden die weiteren durch UND verbunden (ein ODER gibt es nicht, es ergibt sich aus dem Aufstellen mehrerer Regeln mit den gleichen Konklusionen/Prämissen).

```
UND : DAS TIER LEGT EIER
```

DANN : erste Konklusion

Angabe der (ersten) Konklusion einer Regel.

```
DANN : DAS TIER IST EIN VOGEL
```

UND DANN : nächste Konklusion

Angabe der weiteren Konklusion einer Regel.

```
UND DANN : DAS TIER HAT FEDERN
```

ALS HYPOTHESE (ALS ∗)

Die zuletzt eingegebene Konklusion soll als Hypothese für eine spätere Diagnose dienen (im vorigen Beispiel `DAS TIER HAT FEDERN`).

```
ALS HYPOTHESE
```

VERGISS (LOESCHE) ∗ REGEL *n* ∗

Entfernen der Regel Nummer ***n*** (die Regeln sind durchnummeriert) aus der Regelmenge.

```
VERGISS REGEL 3
```

VERGISS (LOESCHE) ∗ REGELN (R)

Löschen aller Regeln, zum Beispiel bevor eine neue Aufgabenstellung eingegeben werden soll.

```
LOESCHE ALLE REGELN
```

DRUCKE (ZEIGE) ∗ REGEL *n*

Ausgabe der Regel Nummer ***n***.

```
DRUCKE REGEL 5
```

DRUCKE (ZEIGE) ∗ REGELN (R)

Ausdrucken aller Regeln.

```
ZEIGE ALLE REGELN
```

∗ FAKTUM (LERNE, MERKE) ∗ : *Faktum*

Füge „Faktum“ in die Liste der Fakten ein.

```
FAKTUM IST : DAS TIER KANN FLIEGEN
MERKE DIR : EIN VOGEL HAT FEDERN
```

VERGISS (LOESCHE) ∗ FAKTUM : *Faktum*

Entferne „Faktum“ aus der Liste der Fakten.

```
VERGISS DAS FAKTUM : DAS TIER KANN FLIEGEN
```

VERGISS (LOESCHE) * FAKTEN (F)

Alle bisherigen Fakten werden gelöscht.
`VERGISS ALLE FAKTEN`

DRUCKE (ZEIGE) * FAKTEN

Ausdrucken aller Fakten.
`DRUCKE DIE FAKTEN`

* HYPOTHESE (HYP) * : *Hypothese*

Füge „Hypothese“ in die Liste der Hypothesen ein.
`EINE HYPOTHESE SEI : DAS TIER IST EIN ALBATROSS`

VERGISS (LOESCHE) * HYPOTHESE (HYP) : *Hypothese*

Entferne „Hypothese“ aus der Liste der Hypothesen.
`VERGISS DIE HYP : DAS TIER IST EINE GIRAFFE`

VERGISS (LOESCHE) * HYPOTHESEN (H)

Alle bisherigen Hypothesen werden gelöscht.
`VERGISS DIE HYPOTHESEN`

DRUCKE (ZEIGE) * HYPOTHESEN

Ausdrucken aller Hypothesen.
`DRUCKE ALLE HYPOTHESEN`

* PRAEMISSE * : *Prämisse*

Ausdrucken derjenigen Regeln, die „Prämisse“ im Wenn-Teil enthalten.
`WELCHE REGELN BENUTZEN DIE PRAEMISSE : DAS TIER IST EIN VOGEL`

* KONKLUSION * : *Konklusion*

Ausdrucken derjenigen Regeln, die „Konklusion“ im Dann-Teil enthalten.
`WELCHE REGELN HABEN DIE KONKLUSION : DAS TIER KANN FLIEGEN`

* DEDUKTION (DEDUZIERE, DEDUZIEREN) *

Bei einer Deduktion wird versucht, mit Hilfe der bisher bekannten Fakten durch Prüfung aller Regeln weitere Fakten zu erhalten.

```
MACHE EINE DEDUKTION
VERSUCHE ETWAS ZU DEDUZIEREN
```

* DIAGNOSE *

Bei einer Diagnose wird versucht, eine der verschiedenen Hypothesen mit Hilfe der Regeln und Fakten als wahr zu erweisen; hierbei wird die Wahr- oder Falschheit von Regel-Prämissen, die dem Programm noch nicht bekannt sind, vom Benutzer erfragt.

```
MACHE EINE DIAGNOSE
```

* ANGEWENDET (BENUTZT) * *n*

Frage, ob bei einer Deduktion oder Diagnose die Regel Nummer ***n*** angewandt wurde.

```
HAST DU REGEL 3 BENUTZT
```

WIE * : *Konklusion*

Frage, mit Hilfe welcher Fakten die „Konklusion“ erschlossen wurde.

```
WIE HAST DU DEDUZIERT : DAS TIER KANN FLIEGEN
```

WARUM * : *Prämisse*

Frage, welche Fakten (Konklusionen) aus der „Prämisse“ folgten.

```
WARUM HAST DU BENUTZT : DAS TIER IST EIN VOGEL
```

WELCHE *

Frage nach den erfolgreich benutzten Regeln:

```
WELCHE REGELN HAST DU BENUTZT
```

ZOOTIERE.DAT

Dieses File enthält einige Regeln und Hypothesen über Tiere in einem Zoo und dient zur Demonstration des LISP-“Experten“ in Form eines kleinen Ratespiels. Es muss nach `EXPERTE.LSP` geladen werden und dient diesem als „Datenbank“.

Nachdem man sich mit `DRUCKE HYPOTHESEN` die Hypothesen angesehen hat, wähle man eine Möglichkeit für sich aus (zum Beispiel „Das Tier ist eine Giraffe“). In Form

einer Diagnose lässt man den Rechner jetzt untersuchen, welches Tier gemeint ist: `MACHE EINE DIAGNOSE` und beantwortet dementsprechend die Fragen des „Experten“.

Vor einer erneuten Diagnose sollte mit `VERGISS ALLE FAKTEN` zunächst die Faktenliste, die bei einer Diagnose aufgestellt wurde, gelöscht werden.

Beispielsitzung

Daden Sie (unter LISP 64 natürlich) erst `EXPERTE.LSP` und dann `ZOOTIERE.DAT` und starten das Programm mit `(DO)`.

```
(LOAD 8 "EXPERTE*")
(LOAD 8 "ZOOTIERE*")
(DO)
```

Dann bringen Sie dem Experten eine neue Regel bei:

```
WENN : DAS TIER KANN FLIEGEN
UND WENN : DAS TIER IST KEIN VOGEL
DANN : DAS TIER IST EINE FLEDERMAUS
```

Jede Zeile muss mit RETURN abgeschlossen werden.

Die Fledermaus in Liste der Hypothesen einfügen:

```
EINE HYPOTHESE SEI : DAS TIER IST EINE FLEDERMAUS
```

Dann:

```
MACHE EINE DIAGNOSE
```

Beantworten Sie nun die folgenden Fragen des „Experten“, so dass er die Fledermaus diagnostizieren soll. Also „DAS TIER FRISST FLEISCH“ mit „Nein“ und so weiter.

```
:? WENN : DAS TIER KANN FLIEGEN
REGEL 16
:? UND WENN : DAS TIER IST KEIN VOGEL
:? DANN : DAS TIER IST EINE FLEDERMAUS
:? EINE HYPOTHESE SEI : DAS TIER IST EINE FLEDERMAUS
:? MACHE EINE DIAGNOSE

IST DIES WAHR (J/N/W) :
DAS TIER HAT HAARE
J

+++ REGEL 1 +++
WENN : DAS TIER HAT HAARE
DANN : DAS TIER IST EIN SAEUGETIER

DEDUZIERT
--> DAS TIER IST EIN SAEUGETIER
```

```
IST DIES WAHR (J/N/W) :
DAS TIER FRISST FLEISCH
N
```

... und so weiter bis ...

```
IST DIES WAHR (J/N/W) :
DAS TIER IST KEIN VOGEL
J

+++ REGEL 16 +++
WENN : DAS TIER KANN FLIEGEN
UND WENN : DAS TIER IST KEIN VOGEL
DANN : DAS TIER IST EINE FLEDERMAUS

DEDUZIERT
--> DAS TIER IST EINE FLEDERMAUS

DIE HYPOTHESE :
--> DAS TIER IST EINE FLEDERMAUS
IST WAHR
```

5.2 ELIZA.LSP

ELIZA ist ein einfaches, aber doch nicht zu triviales Standard-Beispiel für Simulationen mit „Künstlicher Intelligenz“: es imitiert den „Seelendoktor“ in einem psychotherapeutischen Gespräch, indem es den Patienten nach seinen Problemen befragt, auf dessen Intentionen es eingeht, selber Fragen beantwortet und so weiter.

Zur Bedienung

Nach dem Laden wird das Programm durch Eingabe von `(ELIZA)` gestartet. Die Aufforderung `(BITTE ERZAEHLE MIR VON DEINEN PROBLEMEN :)` leitet dann den Dialog zwischen dem Patienten (Mensch) und Psychotherapeuten (Computer!) ein. Die „Stimmhaftigkeit“ oder „Echtheit“ des Dialogs hängt dabei in hohem Maße davon ab, ob man sich auf ein ernsthaftes Gespräch einlässt. Wichtiger Hinweis: Bei der Eingabe eines Satzes sind alle syntaktischen Zeichen (`.,!?` und so weiter) fortzulassen.

Das Programm

Die Hauptfunktion `ELIZA` dient lediglich dazu, eine Begrüßungsformel zu drucken, den Dialog zu starten und nach dessen Beendigung eine Abschiedsmeldung auszugeben.

Die Funktion `DIALOG` ist für den Ablauf des Mensch-Maschine-Dialogs zuständig: Nach dem Einlesen `(SETQ SATZ (READL))` wird versucht, auf diese Eingabe eine passende Antwort zu finden (`FINDE-ANTWORT`, siehe unten). Wird sie gefunden, so wird sie ausgedruckt `(PRINC RESULTAT)`. Die LISP-Funktion `PRINC` unterdrückt beim Ausgeben die Klammern eines LISP-Ausdrucks. Wird keine direkte Antwort gefunden, dann wird eine Ersatz-Antwort gegeben. Ist die Liste der Ersatzantworten erschöpft, so gilt der Dialog als beendet: `(RETURN NIL)`. Ansonsten bewirkt (`GO LOOP1`), dass die nächste Eingabe eingelesen und so der Dialog fortgeführt wird.

Um eine möglichst „intelligente“, das heißt passende Antwort zu finden, wird die Eingabe mit Mustersätzen (englisch: pattern) verglichen, von denen die jeweiligen Antwortsätze abhängig sind. Die Variable `DIALOG-REGELN` enthält eine Liste mit solchen Muster-Antwort-Paaren:

```
DIALOG-REGELN =
((Muster1 Antwortsatz1)
 (Muster2 Antwortsatz2)
...)
```

Ein Muster ist hierbei eine Liste von Wörtern mit folgenden Besonderheiten:

- das Zeichen `*` steht für eine Folge von beliebig vielen Wörtern
- statt eines einzelnen Wortes kann eine Liste mit (alternativen) Wörtern stehen.

Muster sind zum Beispiel:

```
(DIES IST EIN MUSTER)
(* ICH *)
(GUTEN (TAG MORGEN ABEND))
(* ICH (MAG LIEBE HASSE) * MENSCHEN)
```

Das sogenannte „pattern-matching“, das heißt, den Vergleich eines einzelnen Satzes mit einem Muster, führt die Funktion `MATCH` (= übereinstimmen, zuordnen) durch. Deren Wert ist `T` (true), wenn alle Wörter des Satzes dem Muster zugeordnet werden können, sonst `NIL`. Zum Beispiel:

```
(MATCH '(GUTEN (TAG MORGEN ABEND)) '(GUTEN MORGEN)) = T
(MATCH '(* IST *) '(DIE SONNE IST HEUTE ABEND ROT)) = T
```

Die Funktion `FINDE-ANTWORT` wendet `MATCH` nacheinander auf die in der Liste `DIALOG-REGELN` enthaltenen Muster an, bis eine Übereinstimmung gefunden wird, oder die Liste erschöpft ist. Im ersten Falle wird das Muster-Antwort-Paar an das Ende der `DIALOG-REGELN` befördert (durch die Funktionen `NCONC1` bis `RPLACD`), damit bei einer späteren gleichen oder ähnlichen Eingabe zuerst mit anderen Mustern verglichen und so (eventuell) eine andere Antwort gefunden wird.

Listings der wichtigsten ELIZA-Funktionen:

```
(DE ELIZA ()
  (MSG(CHAR 147) T
    "HALLO, ICH BIN ELIZA !" T T
    "BITTE ERZAEHLE MIR VON" T T
    "DEINEN PROBLEMEN :" T T)
  (DIALOG ERSATZ-ANTWORTEN)
  (MSG T "WIR MUESSEN UNSERE SITZUNG" T
     "LEIDER BEENDEN." T T
     "TSCHUESS !" T T))

(DE DIALOG (ERSATZ-ANTWORTEN)
  (PROG(RESULTAT SATZ)
    LOOP1
    (SETQ SATZ (READL))
    (SETQ RESULTAT (FINDE-ANTWORT SATZ DIALOG-REGELN))
    (COND(RESULTAT(PRINC RESULTAT))
      (ERSATZ-ANTWORTEN
         (PRINC(CAR ERSATZ-ANTWORTEN))
         (SETQ ERSATZ-ANTWORTEN
            (CDR ERSATZ-ANTWORTEN)))
      (T(RETURN NIL)))
    (TERPRI)
    (GO LOOP1)))

(DE FINDE-ANTWORT (S R)
  (PROG(RESULTAT)
    LOOP
    (COND((NULL R) (RETURN NIL))
       ((MATCH(CAAR R) S)
         (SETQ RESULTAT (CAR(CDAR R)))
         (NCONC1 R (CAR R))
         (RPLACA R (CADR R))
         (RPLACD R (CDDR R))
         (RETURN RESULTAT)))
    (SETQ R (CDR R))
    (GO LOOP)))
```

```
(DE MATCH (P S)
  (COND((NULL P) (NULL S))
    ((EQ(CAR P) '*)
      (COND((NULL S) (NULL(CDR P)))
        ((MATCH(CDR P) S))
        ((MATCH P (CDR S)))))
    ((NULL S) NIL)
    ((EQ(CAR P) (CAR S))
      (MATCH(CDR P) (CDR S)))
    ((AND(CONSP(CAR P))
      (MEMBER(CAR S) (CAR P)))
        (MATCH(CDR P) (CDR S)))))
```

Angemerkt sei, dass es, nachdem der Aufbau der globalen Liste `DIALOG-REGELN` beschrieben ist, sehr leicht ist, das Verhalten von `ELIZA` zu ändern, indem man die Liste ergänzt oder Muster-Antwort-Paare löscht. Soll `ELIZA` zum Beispiel auf die Eingabe `ICH HABE ANGST` mit der Frage `WOVOR FUERCHTEST DU DICH?` reagieren, wäre die folgende Ergänzung denkbar:

```
(NCONC1 DIALOG-REGELN
   '(((ICH HABE (ANGST FURCHT) *)
      (WOVOR FUERCHTEST DU DICH?))))
```

6 Erweiterungen für LISP 128: KEYDEF.LSP, VED.LSP und Autostart

Mit den C128–spezifischen Erweiterungen aus Kapitel 3 lassen sich nun einige Komfortfunktionen definieren, wie z.B. die Funktionstastenbelegung oder die Nutzung eines visuellen Editors.

6.1 Funktionstastenbelegung mit KEYDEF.LSP

```
(DE KEYDEF
  (K S)
  (COND((AND(LESSP K 11)
       (STRINGP S))
       (PL(UNPACK S)
         (SUB1(DEC "0B02")))
       (BANK 0)
       (POKE(DEC "03EC")
         (DEC "00FA"))
       (POKE(DEC "03ED") K)
       (POKE(DEC "03EE")
         (LENGTH(UNPACK S)))
       (POKE(DEC "00FA")
         (DEC "02"))
       (POKE(DEC "00FB")
         (DEC "0B"))
       (POKE(DEC "00FC")
         (DEC "00"))
       (BANK 15)
       (CALL(DEC "FF65")) T)
    (T NIL)))

(DE PL
  (L I)
  (COND((NULL L)
        T)
    (T(BANK 0)
      (POKE(ADD1 I)
        (ASC(CAR L)))
      (PL(CDR L)
        (ADD1 I)))))
```

Die Funktion `KEYDEF` bekommt als ersten Parameter K die Nummer der zu belegenden Funktionstaste mitgeteilt und als zweiten Parameter S den String, mit dem diese Taste belegt werden soll. Zur Belegung der Funktionstaste soll die Kernel-Routine `JKEYSET` $FF65 benutzt werden. Diese wird in Bank 15 aufgerufen, wo u.a. der Kernel zur Verfügung steht. Zum Aufruf der Routine sind zunächst einige Vorbereitungen zu treffen. In der Zero–Page ist an drei aufeinanderfolgenden Adressen ein Zeiger auf den neuen, der Funktionstaste zuzuordnenden Text einzurichten. Anschließend ist in den Akkumulator die Adresse des ersten Bytes dieses Zeigers zu laden (einen Zeiger auf den Zeiger sozusagen), in das X–Register die Nummer der Funktionstaste laut

unten stehender Tabelle und in das Y–Register die Länge des Textes. Funktionstaste F3 soll z.B. mit dem Text „(dir)“ definiert werden, der zuvor im Speicher ab Adresse $0B02 im Kassettenpuffer abgelegt wird (dies übernimmt die Hilfsroutine PL (PokeList), die die übergebene Liste *L* ab Adresse *I* in den Speicher POKEt). (Die ersten beiden Bytes des Kassettenpuffers $0B00 und $0B01 sind bereits für MrEd reserviert, und zeigen auf das Ende des Editor–Textes.) Die Speicherstellen $FA–$FE in der Zeropage sind nicht belegt, wir können also $FA–$FB für den Zeiger auf den Text und $FC für die Banknummer in der der Text steht benutzen.
Belegt wird also

- AC ($03EC) mit $FA, der Zeigeradresse
- XR ($03ED) mit $03, der Nummer der Funktionstaste
- YR ($03EE) mit $05, der Länge des Textes
- $FA mit $02, dem Low–Byte der Adresse $0B02
- $FB mit $0B, dem High–Byte der Adresse $0B02
- $FC mit $00, der Banknummer 0

Nun kann mit `(BANK 15)` und nachfolgendem `(CALL (DEC "FF65"))` die Routine aufgerufen werden.
Mögliche Funktionstasten sind die Tasten F1 bis F8, die HELP –Taste und die SHIFT-RUN/STOP –Taste. Die Nummern sind in der nachfolgenden Tabelle angegeben:

Nummer	Taste	Standardwert in BASIC
1	F1	`GRAPHIC`
2	F2	`DLOAD"`
3	F3	`DIRECTORY + CHR$(13)`
4	F4	`SCNCLR + CHR$(13)`
5	F5	`DSAVE"`
6	F6	`RUN + CHR$(13)`
7	F7	`LIST + CHR$(13)`
8	F8	`MONITOR + CHR$(13)`
9	SHIFT-RUN/STOP	`DL"*" + CHR$(13) + RUN + CHR$13)`
10	HELP	`HELP + CHR$(13)`

Man sollte beachten, dass beim Belegen der Funktionstasten alle zehn Definitionen zusammen nicht mehr als 246 Zeichen umfassen dürfen. Siehe hierzu auch [Sch86] S. 63f.

Anwendungsbeispiel

Zur Vorbelegung von Funktionstasten könnte man sich damit z.B. folgende Funktion erstellen und diese etwa unter `KEYDEFS.LSP` speichern:

```
(DE KEYDEFS ()
  (KEYDEF 1 (ADDCR '"(PDEF KEYDEFS)"))
  (KEYDEF 3 (ADDCR '"(DIR)"))
  (KEYDEF 5 "")
  (KEYDEF 7 "")
)

(DE ADDCR (L)
  (PACK (APPEND (UNPACK L)
                (CONS (CHAR 13) NIL))))
```

Somit würde F1 die aktuelle Funktionstastenbelegung zeigen und F3 das aktuelle Directory auflisten.

6.2 Visueller Editor mit VED.LSP

In LISP 128 ist neben dem Kommando `(ED)` (siehe S. 40) auch eine RAM–Disk „`RaDi`“ eingebaut (siehe Abschnitt 6.3). Damit ergibt sich die Möglichkeit mit einigen Hilfsroutinen LISP-Funktionen in den Editor-Speicher zu transferieren, dort zu bearbeiten und dann aus dem Editor-Speicher wieder zurückzuladen. Die RAM–Disk ist so eingerichtet, dass sie mit dem Editor-Speicher arbeitet. Die Hilfsroutinen können aus der Datei `VED.LSP` geladen werden.

```
(DE TED
  (N)
  (OPEN 1 7 2 "")
  (OUTPUT 1)
  (PRINT-PROPS-L N)
  (NORMAL)
  (CLOSE 1))

(DE FRED NIL
  (PROG(E)
    (NORMAL)
    (CLOSE 1)
    (OPEN 1 7 0 "")
    (INPUT 1) LOOP
    (SETQ E
      (READ))
    (COND((ATOM E)
        (NORMAL)
        (CLOSE 1)
        (RETURN T)))
    (PRINT(EVAL E))
    (GO LOOP)))

(DE PRINT-PROPS-L
  (L)
  (COND((NULL L)
      NIL)
    (T(PRINT-PROPS(CAR L)
        (GETPROPLIST(CAR L)))
      (PRINT-PROPS-L(CDR L)))))

(DE PRINT-PROPS
  (A P)
  (COND((NULL P)
      NIL)
    (T(PP(COND((EQ(CAR P)
               'EXPR)
             (CONS 'DE
               (CONS A
                 (CDR(CADR P)))))
            ((EQ(CAR P)
               'FEXPR)
             (CONS 'DF
               (CONS A
                 (CDR(CADR P)))))
            ((EQ(CAR P)
               'MACRO)
             (CONS 'DM
               (CONS A
                 (CDR(CADR P)))))
            ((EQ(CAR P)
               'VALUE)
             (LIST 'SETQ A
               (LIST 'QUOTE
                 (CADR P))))
            (T(LIST 'DEFPROP A
                (CAR P)
                (CADR P)))))
      (TERPRI)
      (PRINT-PROPS A
        (CDDR P)))))

(SETQ VEDFNS
   '(TED FRED PRINT-PROPS-L
    PRINT-PROPS VEDFNS))
```

Um die Routinen nutzen zu können, müssen wir sie zunächst mit `(LOAD 8 "VED*")` laden.

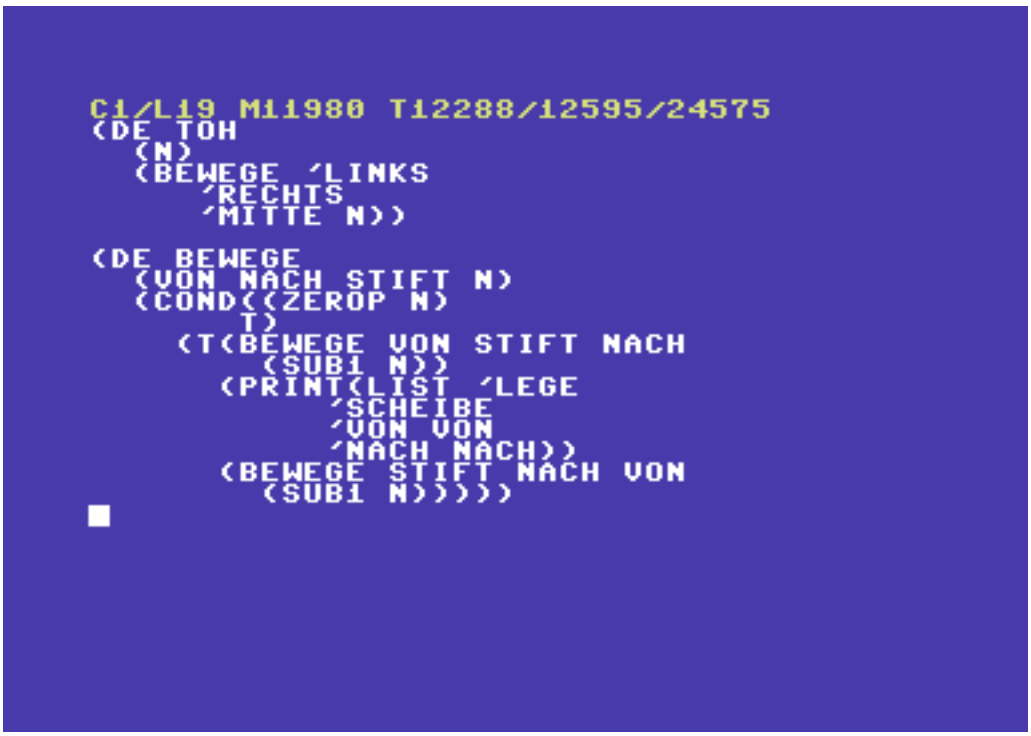


Abbildung 6.1: MrEd

6.2.1 TED („TO ED“)

Transferiert eine oder mehrere Funktionen in den Editor. Angenommen wir haben VED aktiviert und nun mit `(LOAD 8 "TOH*")` die beiden Routinen `TOH` und `BEWEGE` geladen und wollen diese nun im Editor bearbeiten. Dann geben wir ein `(TED '(TOK BEWEGE))` und anschließend `(ED)`. Dann können wir die Funktionen ansehen und bearbeiten, wie in Abb. 6.1 zu sehen. Der Editor wird, wie auf S. 40 beschrieben, mit RUN/STOP verlassen.

6.2.2 FRED („FROM ED“)

`(FRED)` lädt die im Editor befindlichen Routinen wieder in den LISP-Interpreter. Die beiden Routinen `TED` und `FRED` demonstrieren auch den Umgang mit der RAM–Disk.

6.3 RaDi: Die RAM–Disk

Die SuperTape–Routinen aus LISP 64 wurden entfernt und durch entsprechende Routinen zur Bedienung einer RAM–Disk ersetzt. Die RAM–Disk wird so ähnlich wie eine Kassette angesprochen, jedoch unter der Geräternummer 7. Die folgenden Kernel–Routinen wurden dazu erweitert: `OPEN`, `CLOSE`, `CHKIN`, `CKOUT`, `CLRCH`, `BASIN`, `BSOUT`, `GETIN`, `LOAD` und `SAVE`.

6.4 SYSINFO.LSP

SYSINFO zeigt wie einige Werte aus der Config–Area ausgegeben werden können. COPYRIGHT demonstriert den Aufruf einer Maschinen–Routine.

```
(DE SYSINFO NIL
  (MSG "STACKS" T "------" T)
  (PRINC "DSTACK:    $")
  (PRIN1(HEEK(PLUS(CFGAREA) 8)))
  (MSG T)
  (PRINC "DSTACKMAX: $")
  (PRIN1(HEEK(PLUS(CFGAREA) 12)))
  (MSG T)
  (PRINC "ASTACKMIN: $")
  (PRIN1(HEEK(PLUS(CFGAREA) 14)))
  (MSG T)
  (PRINC "ASTACK:    $")
  (PRIN1(HEEK(PLUS(CFGAREA) 10)))
  (MSG T))

(DE HEEK
  (A)
  (HEX(DEEK A)))

(DE DEEK
  (A)
  (BANK 0)
  (PLUS(PEEK A)
    (TIMES(PEEK(ADD1 A)) 256)))

(DE COPYRIGHT NIL
  (BANK 0)
  (* NO CLRSCR: SET YR=2 *)
  (POKE(DEC "03EE") 2)
  (CALL(PLUS(CFGAREA) 99)))
```

6.5 Autostart mit INIT.LSP

Der LISP 128 – Interpreter sucht beim Kaltstart nach einer User-Datei (,U) namens "INIT␣␣␣␣␣␣␣␣.LSP". Ist diese vorhanden, wird sie geladen und ausgeführt. Wichtig ist, dass diese Datei nur eine Funktionsdefinition enthalten darf, die dann ausgeführt werden kann. Nachfolgend ein Beispiel für den Inhalt dieser Datei:

```
(DE DOINIT NIL
  (MSG "INITIALIZING..." T)
  (LOAD 8 "VED          .LSP")
  (LOAD 8 "KEYDEF       .LSP")
  (LOAD 8 "KEYDEFS      .LSP")
  (KEYDEFS)
  (MSG "INIT DONE." T))
```

In Abb. 6.2 ist zu sehen, wie die Startdatei geladen wird. Dies ist implementiert mit simulierten Tastendrücken, die in den Tastaturpuffer geschrieben werden.

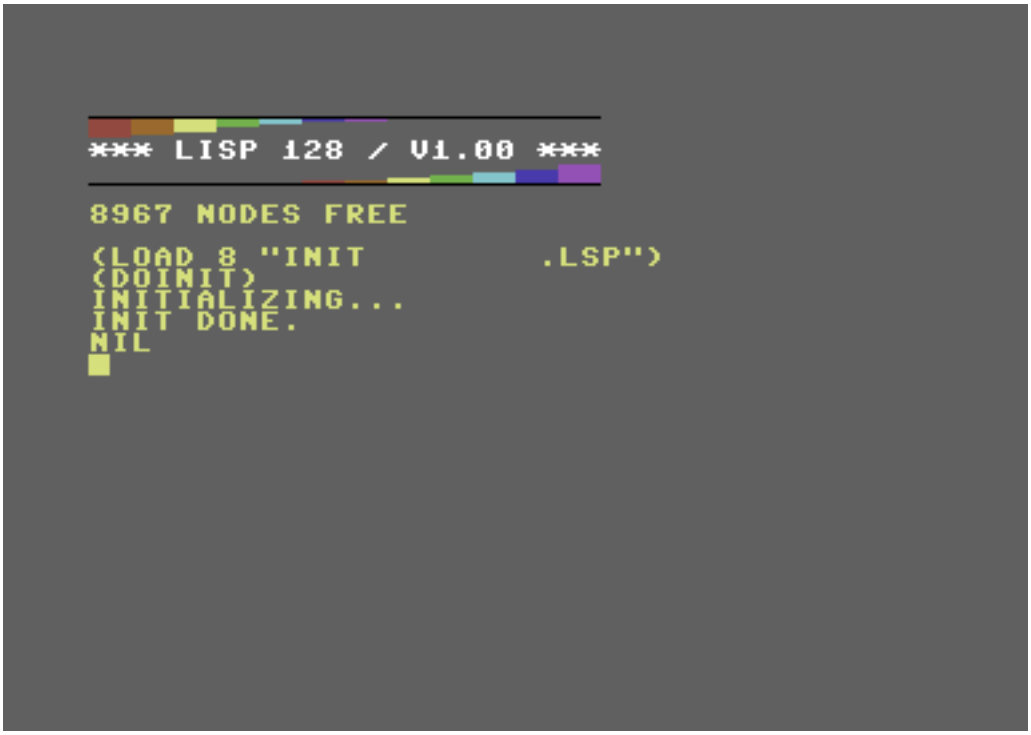


Abbildung 6.2: Ein Screenshot nach dem Interpreter–Kaltstart

7 LISP 64/128 - die Interna

7.1 Programmstart

LISP 64/128 wird wie ein Basic-Programm geladen (beim C64 an die Adresse $0801 und beim C128 an die Adresse $1C01).

Dort befindet sich eine einzige Basic-Zeile die mit SYS, bzw. BANK und SYS (beim C128) den Einsprung in die Startroutine `Main` durchführt.

`Main` überspringt mit „`JMP Init`“ die im folgenden Abschnitt (7.2) dokumentierte Config-Area. Beim C128 wird mit `JSR Bootstrap` noch eine Routine aufgerufen, die verschiedene Bestandteile des Systems (JSF (JSR-FAR), MrEd (Editor) und RaDi (RAM–Disk)) an den für sie vorgesehenen Stellen im System installiert.

7.2 Speicher/System-Konfigurationsbereich

Daran schließt die Config-Area (siehe Tabelle 7.1) an, die wichtige Systemparameter und einige Zeiger enthält. Eine Anpassung der Zeiger DSTACK und ASTACK erlaubt es z.B. die LISP-Stapel an eine andere Stelle zu verlegen. Mit der Anpassung von FSLTOP kann bestimmt werden, bis wohin im Speicher LISP seine Freispeicherliste aufbaut. DSTACKMAX und ASTACKMIN sind für Zwecke der Systemanalyse unter C128 gedacht und speichern die Stack-Pointer-Maxima, um zu sehen, wie die Stacks bei verschiedenen Anwendungen genutzt werden. Diese Werte sind in der C128 Version nur vorhanden, wenn in „`config.i`“ `Debug_Stacks = 1` gesetzt wurde.

Adr. C64	Adr. C128	Offset	Label	Typ	Value C64	Value C128
$0900	$1C28	$00	CfgArea/MaxLine	WORD	38	38
$0902	$1C2A	$02	HighWaterMark	WORD	128	128
$0904	$1C2C	$04	FSL	WORD	EndOfProg	EndOfProg
$0906	$1C2E	$06	FSLTOP	WORD	$CFF0	$FEF0
$0908	$1C30	$08	DSTACK	WORD	$E000	$E000
$090A	$1C32	$10	ASTACK	WORD	$FFFE	$FEFE
nicht vorh.	$1C34	$12	DSTACKMAX	WORD	-	$0000
nicht vorh.	$1C36	$14	ASTACKMIN	WORD	-	$0000
$090C	$1C38	$12/$16	OBJ	WORD	NIL	NIL
$090E	$1C3A	$14/$18	NODES	WORD	ATBOT	ATBOT
$0910	$1C3C	$16/$1A	TACK	WORD	NIL	NIL

Tabelle 7.1: Config Area

Direkt an die Config-Area schließt der unterste Knoten (HashBase = `NIL`) an und an diesen die im nächsten Abschnitt beschriebene Hash-Tabelle (32 Byte-Paare als Zeiger auf Hash-Bucket-Listen $00-$1F) (siehe Listing 7.1).

Listing 7.1 Die Hash-Tabelle

```
001C3E  34 4A        HashBase:     .WORD nNIL
001C40  3E 1C                      .WORD HashBase
001C42  41                         .BYTE $41      ; Type 1: LITERAL; Flag 4 set

001C43  48 4A        HashTable:    .WORD Chain00  ; HashTable[$00]
001C45  52 4A        HashTable_02: .WORD Chain01  ; HashTable[$02]
001C47  66 4A        HashTable_04: .WORD Chain02  ; HashTable[$04]
001C49  70 4A        HashTable_06: .WORD Chain03  ; ...
001C4B  7A 4A        HashTable_08: .WORD Chain04
001C4D  8E 4A        HashTable_0A: .WORD Chain05
001C4F  A7 4A        HashTable_0C: .WORD Chain06

...

001C75  69 4C        HashTable_32: .WORD Chain19
001C77  7D 4C        HashTable_34: .WORD Chain1A
001C79  A5 4C        HashTable_36: .WORD Chain1B
001C7B  C3 4C        HashTable_38: .WORD Chain1C
001C7D  D2 4C        HashTable_3A: .WORD Chain1D
001C7F  DC 4C        HashTable_3C: .WORD Chain1E
001C81  EB 4C        HashTable_3E: .WORD Chain1F  ; HashTable[3E]
                                                  ; (last HashTable entry)
001C83  3E 1C                      .WORD HashBase
001C85  3E 1C                      .WORD HashBase
```

7.3 Die Objektliste (OBLIST)

In Abschnitt 1.7.1.3 wurde kurz erwähnt, dass Literale zum schnellen Zugriff in einer hash-adressierten Liste organisiert sind, und diese deshalb eine seltsame Anordnung bei `(OBLIST)` zeigt. Schauen wir uns die Situation zunächst direkt nach dem Neustart des LISP-Interpreters an:

```
> (oblist)
(go gc)
(pp open output oblist)
(getprop)
(greaterp getdef)
(print prinl getchar poke)
(prin1 pdef prog1 hbyte waitchar)
(plus putprop getproplist prog progn pack peek and)
(princ)
(atom add1 input)
(append apply quote asc abs)
(*)
(read quotient assoc apply*)
(zerop reset readl)
(rplacd readch st)
(rplaca remprop remob remove set random)
(cond reverse return call)
(cdr car t setq copy sub1 consp compl save)
(close cadr cddr cdar caar char)
(cons de dm remainder stringp)
(df list last load)
(lessp tab sassoc spaces label)
(length lbyte logand)
(eq terpri disk logor defprop map lambda)
(eval times logxor exit)
(equal)
(minusp msg unbind expr)
(nil f not difference nth mapc maplist error)
(null mapcar mapcan member macro minus)
(map2car unpack)
(value normal)
(numberp nconc1 fexpr)
(or nconc nlambda)
t
```

Die Ausgabe von `(OBLIST)` gibt also genau 32 Unterlisten aus, den Hash-Tabellen-Buckets 0 bis 31 entsprechen. In welche dieser Listen ein Literal gehört, wird von der folgenden Hashfunktion entschieden, die zum Namen eine Zahl zwischen 0 und 31 liefert:

```
sub GetHashTableIndex
{
        my $string = shift;
        my $lstchpos = length($string)-1;
        my $frstchar = substr($string,0,1);
        my $lastchar = substr($string,$lstchpos,1);

        my $hti = (4 * (ord($frstchar) & 7) +
                        (ord($lastchar) & 3) +
                             $lstchpos) & 0x1F;

        return $hti;
}
```

Die Funktion ist hier in Perl angegeben, damit man schnell den Hash-Wert neuer (z.B. wenn eine neue interne Funktion eingebaut werden soll) und bestehender Literale (für Analyse-Zwecke) berechnen kann. Im 6502-Assembler-Original sieht die Funktion so aus:

```
001AA9  ;
001AA9  ; HashTableIndex = 2 * ((4*(Buf[1] & 7) + (Buf[LastChar] & 3) + Len(Buf)-1)
    & 0x1F)
001AA9  ; ====================================
001AA9                  MakeHash:
001AA9  AD F1 07                LDA READBUF+1
001AAC  29 07                   AND #7
001AAE  0A                      ASL A
001AAF  0A                      ASL A
001AB0  85 29                   STA HashTableIndex
001AB2  AE F0 07                LDX READBUF
001AB5  BD F0 07                LDA READBUF,X
001AB8  29 03                   AND #3
001ABA  18                      CLC
001ABB  65 29                   ADC HashTableIndex
001ABD  85 29                   STA HashTableIndex
001ABF  CA                      DEX
001AC0  8A                      TXA
001AC1  18                      CLC
001AC2  65 29                   ADC HashTableIndex
001AC4  29 1F                   AND #$1F
001AC6  0A                      ASL A
001AC7  85 29                   STA HashTableIndex
001AC9  60                      RTS
```

Die Hashtabelle liegt im Speicher ab $0917 und enthält 32 Byte-Paare (WORD) als Zeiger auf die Bucket-Listen. Nehmen wir die erste davon als Beispiel und lassen uns diese grafisch anzeigen (Abb. 7.1).

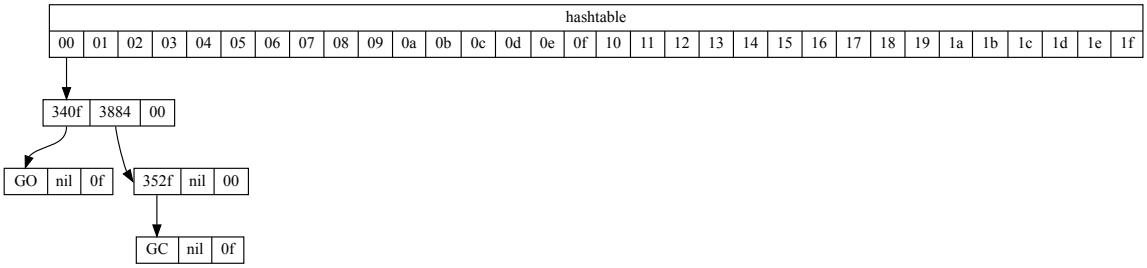


Abbildung 7.1: Bucket 00 direkt nach dem Neustart des Interpreters

Wir sehen oben die Hashtabellen-Einträge $00 bis $1F und ausgehend vom Eintrag $00 die erste Unterliste `(go gc)`. Eintrag $00 zeigt also auf die Cons-Zelle ($387f), und diese mit ihrem CAR-Slot auf GO und mit ihrem CDR-Slot auf eine weitere Cons-Zelle. Diese zeigt wiederum auf GC mit ihrem CAR-Slot und auf NIL mit ihrem CDR-Slot. Also die bereits in Abschnitt **??** erwähnte Listendarstellung durch einen binären Baum. Jede Cons-Zelle (auch LISP-Knoten, LISP-Node genannt) trägt

als 5. Byte noch eine Typ-Information, die Näheres zum Inhalt und Aufbau dieser Zelle festlegt (Tabelle 7.2).

Wert	Typ	Info
$00	Liste	Cons-Zelle
$01	Literal	Zeichenkettenknoten-Liste folgt
$41	Literal + Flag 4	benutzt bei `NIL`, `F`, `T`
$02	Zahl	32-Bit-Ganzzahl
$04	Zeichenkettenknoten	1 bis 4 Zeichen
$08	String	Zeichenkettenknoten-Liste folgt
$0F	Interner-Konten	mit Assembler-Routine als Handler

Tabelle 7.2: LISP 68/128-Knotentypen

7.4 Die Freispeicherliste (FSL)

Die Freispeicherliste beginnt bei FSL (`CfgArea`+$04) und geht bis `FSLTOP` (`CfgArea`+$06). Bei der C64-Version also ab `FSL`/`EndOfProg` ($3B09) bis `FSLTOP` ($CFF0).

$$(\texttt{FSLTOP} - \texttt{FSL} + 1) \,/\, 5 = 7624.$$

Dies ist die angezeigte Anzahl von freien Knoten in der Startmeldung.
Die FSL wird in `InitFSL` aufgebaut. Im Pseudocode sieht dies in in etwa so aus, wie im Listing 7.2.

Listing 7.2 Initialisierung der Freispeicherliste (FSL)

```
(ACC32) = 0 ; Node counter
AX = (EndOfProg) ; = $3b09
(FSLPtr) = AX
do {
  (NodePtr) = AX
  (NextPtr) = AX
  (NextPtr) += 5
  (NodePtr)[0..1] = #NIL ($0912)
  (NodePtr)[2..3] = (NextPtr)[0..1]
  (NodePtr)[4] = 0
  (ACC32)++
  AX = (NextPtr)
} while ((NextPtr) < (FSLTOP))

$3b09  12 09 0e 3b 00
       |   | |   | ||
       |   | |   | ++-- Flag
       |   | +---+----- NextPtr
       +---+----------- NIL
$3b0e  12 09 13 3b 00
$3b13  12 09 18 3b 00
$3b18  12 09 1d 3b 00
$3b1d  12 09 22 3b 00
$3b22  12 09 27 3b 00
$3b27  12 09 2c 3b 00

...

$cfe2  12 09 e7 cf 00
$cfe7  12 09 ec cf 00
$cfec  12 09 12 09 00   ; last node!
       |   | |   | ||
       |   | |   | ++-- Flag ($CFF0 = FSLTOP)
       |   | +---+----- NextPtr (=NIL!)
       +---+----------- NIL
$cff1  .. ..
```

Eine grafische Visualisierung findet sich in Abb. 7.2.

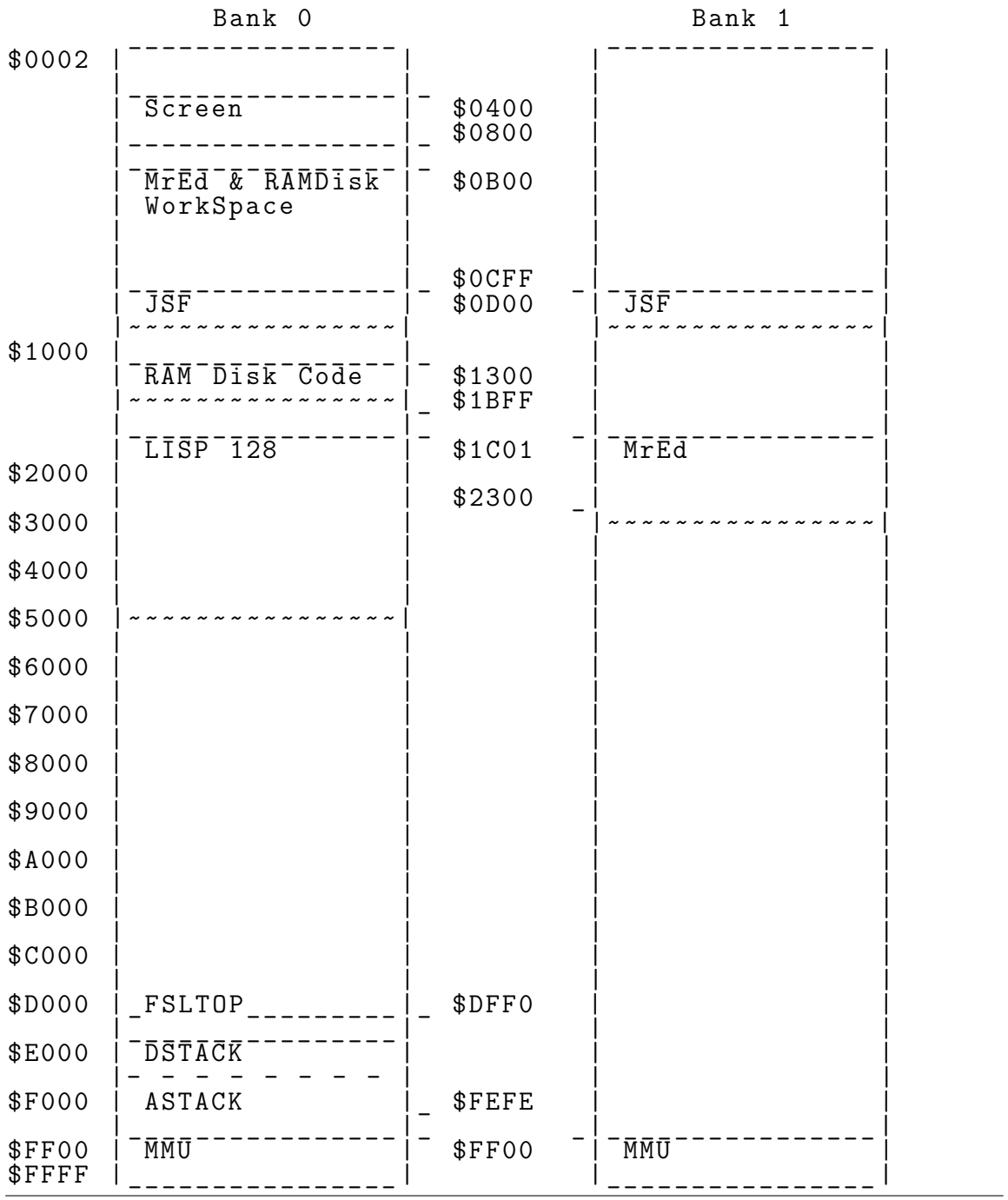
7.5 JSF der Speicherbankmanager

JSF (Jump to Subroutine Far) ist der C128-Speicherbankmanager von Sven Friedrichs (siehe [Fri96]). JSF enthält die folgenden Routinen, die das Arbeiten durch die Umschaltung zwischen verschiedenen Speicherbanken durch eine automatische Umschaltung erleichtern.

- JIMM (JSR IMMediate): JSR fuer Fremdbank
- SIMM (Stash IMMediate): STASH fuer Fremdbank
- FIMM (Fetch IMMediate): FETCH fuer Fremdbank

Der Code liegt in Bank 0 und Bank 1 an Adresse $0D00–$0D75.

7.6 Memory–Map



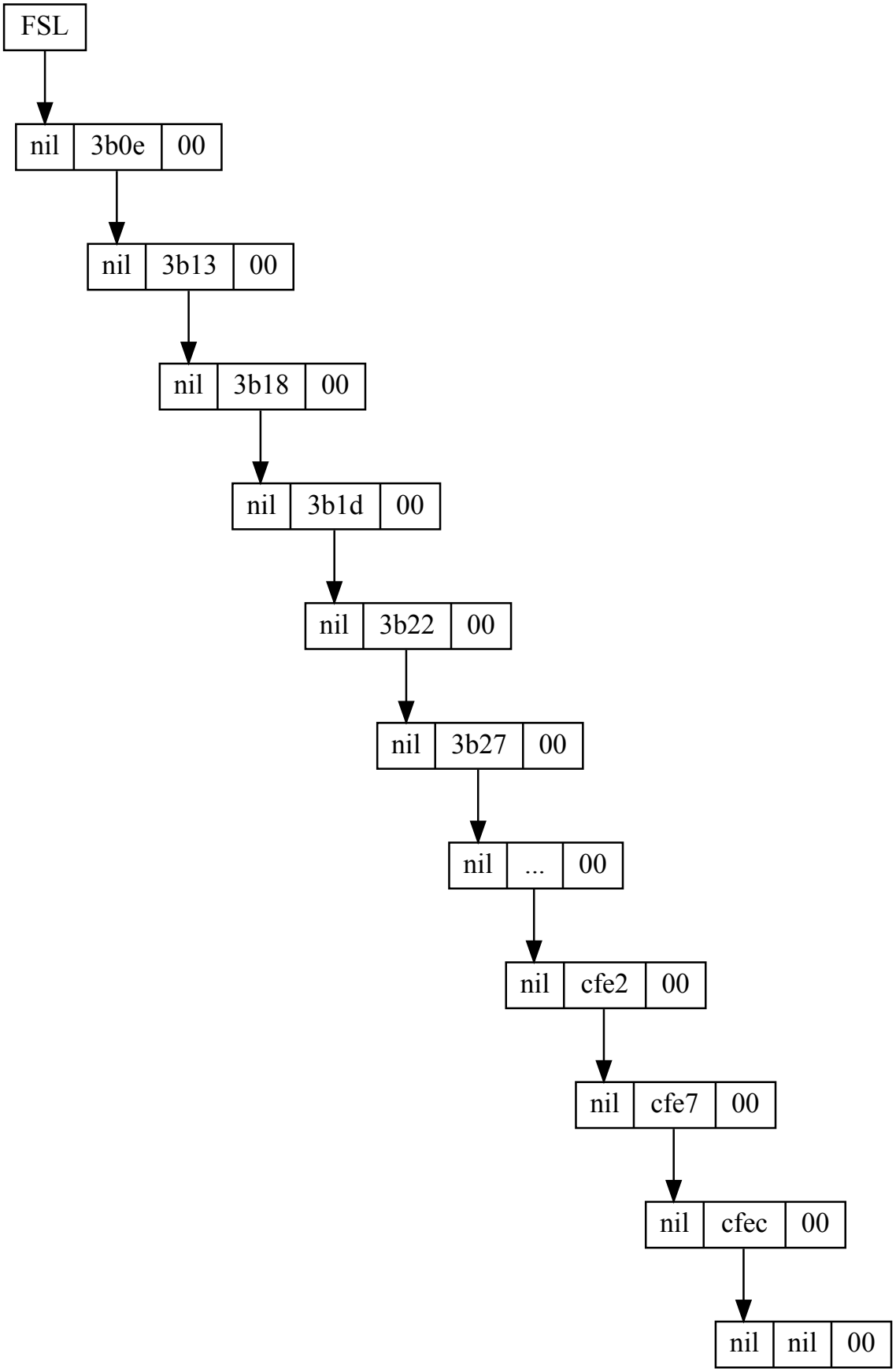


Abbildung 7.2: Freispeicherliste (FSL)

8 Beispielprogramme

8.1 TOH.LSP

```
(DE TOH
  (N)
  (BEWEGE 'LINKS
     'RECHTS
     'MITTE N))

(DE BEWEGE
  (VON NACH STIFT N)
  (COND((ZEROP N)
        T)
    (T(BEWEGE VON STIFT NACH
        (SUB1 N))
      (PRINT(LIST 'LEGE
           'SCHEIBE
           'VON VON
           'NACH NACH))
      (BEWEGE STIFT NACH VON
        (SUB1 N)))))
```

8.2 HANOI.LSP

```
(DE COUNTUP
  (N)
  (REVERSE(COUNTDOWN N)))

(DE COUNTDOWN
  (N)
  (COND((EQ N 0)
        NIL)
    (T(CONS N
        (COUNTDOWN(SUB1 N))))))

(DE RESET-TOH
  (N)
  (SETQ C NIL)
  (SETQ B NIL)
  (COND((ZEROP N)
      (SETQ A
        (COUNTUP 4)))
    (T(SETQ A
        (COUNTUP N)))))

(DE TRANSFER-TOH
  (FROM TO SPARE N)
  (COND((ZEROP N)
      (PRINT(LIST FROM
            'LEER)))
    ((EQ N 1)
      (MOVEDISK-TOH FROM TO))
    (T(APPEND(TRANSFER-TOH FROM SPARE TO
          (SUB1 N))
        (MOVEDISK-TOH FROM TO)
        (TRANSFER-TOH SPARE TO FROM
          (SUB1 N))))))

(DE MOVEDISK-TOH
  (FROM TO)
  (PRINT(LIST 'SCHRITT
      (SETQ STEP
        (ADD1 STEP))
       'BEWEGE
      (CAR(EVAL FROM))
       'VON FROM
       'NACH TO))
  (COND((NULL(EVAL FROM))
      (PRINT(LIST FROM
            'LEER)))
    ((OR(NULL(EVAL TO))
      (LESSP(CAR(EVAL FROM))
        (CAR(EVAL TO))))
      (SET TO
        (CONS(CAR(EVAL FROM))
          (EVAL TO)))
      (SET FROM
        (CDR(EVAL FROM))))
    (T(PRINT(LIST 'CANT
           'BEWEGE
          (CAR(EVAL FROM))
           'NACH
          (CAR(EVAL TO)))))))

(DE HANOI
  (N)
  (RESET-TOH N)
  (SETQ STEP 0)
  (TRANSFER-TOH 'A
     'C
     'B
    (LENGTH A)) T)
```

8.3 PERM.LSP

```
(DE PERM
  (L)
  (PRINT L)
  (PERM1 L
    (CDR L) L) T)

(DE PERM1
  (L L1 L2)
  (COND((ATOM L1)
        NIL)
    (T(PERM1 L
         (CDDR L2)
         (CDR L2))
      (PERM2 L L1 L2))))

(DE PERM2
  (L L1 L2)
  (COND((ATOM L1)
        NIL)
    (T(TAUSCH L1 L2)
      (PRINT L)
      (PERM1 L
        (CDDR L2)
        (CDR L2))
      (TAUSCH L1 L2)
      (PERM2 L
        (CDR L1) L2))))

(SETQ PERM2
   'NIL)

(DE TAUSCH
  (X Y Z)
  (SETQ Z
    (CAR X))
  (RPLACA X
    (CAR Y))
  (RPLACA Y Z))

(DE DEMO NIL
  (MSG T
    "ERZEUGUNG VON PERMUTATIONEN" T
    "--------------------------" T T)
  (MSG "(PERM '(1 2 3 4)) = " T T)
  (PERM '(1 2 3 4))
  (MSG T T
    "(PERM '(APFEL BANANE CITRONE)) = " T T)
  (PERM '(APFEL BANANE CITRONE)))
```

8.4 SYMB-DIFF.LSP

```
(DE DO NIL
  (PROG(FUNC ABL VAR) LOOP1
    (MSG T T "F(X) = ")
    (SETQ FUNC
      (READ))
    (COND((EQ FUNC
          '-)
        (RETURN T))) LOOP2
    (MSG T "ANZ. D. ABLEITUNGEN: ")
    (SETQ ABL
      (READ))
    (COND((OR(NOT(NUMBERP ABL))
        (MINUSP ABL)
        (GREATERP ABL 5))
        (GO LOOP2)))
    (ABLFUNC(PREFIX FUNC) ABL)
    (GO LOOP1)))

(DE DIFF
  (Y X)
  (COND((CONSTP Y X) 0)
    ((EQ Y X) 1)
    (T(APPLY(DIFF-FUNCTION(CAR Y))
        (CDR Y)))))

(DE DIFF-FUNCTION
  (FN)
  (OR(CDR(ASSOC FN SYSTEM-FUNCTIONS))
    (GETPROP FN
      'DFUN)
    (ERROR '(NON-DIFFERENTIAL TERM))))

(SETQ SYSTEM-FUNCTIONS
   '((PLUS.DPLUS)
      (TIMES.DTIMES)
      (QUOTIENT.DQUOTIENT)
      (MINUS.DMINUS)))

(DE CONSTP
  (L X)
  (COND((ATOM L)
      (NOT(EQ L X)))
    ((CONSTP(CAR L) X)
      (CONSTP(CDR L) X))))

(DE SIMPLE
  (X)
  (COND((ATOM X)
       X)
    ((ASSOC(CAR X) SIASSOC)
      (APPLY(CDR(ASSOC(CAR X) SIASSOC))
        (MAPCAR 'SIMPLE
          (CDR X))))
    (T(CONS(CAR X)
        (MAPCAR 'SIMPLE
          (CDR X))))))

(SETQ SIASSOC
   '((PLUS.SPLUS)
      (TIMES.STIMES)
      (QUOTIENT.SQUOTIENT)
      (MINUS.SMINUS)
      (EXPT.SEXPT)))

(DE PLN
  (A B)
  (COND((ZEROP A)
       B)
    ((NUMBERP B)
      (PLUS A B))
    ((AND(EQ(CAR B)
          'MINUS)
      (NUMBERP(CADR B)))
      (DIFFERENCE A
        (CADR B)))
    ((EQ(CAR B)
       'PLUS)
      (COND((NUMBERP(CADR B))
          (LIST 'PLUS
            (PLUS A
              (CADR B))
            (CAR(CDDR B))))
```

```
        ((NUMBERP(CAR(CDDR B)))
          (LIST 'PLUS
            (PLUS A
              (CAR(CDDR B)))
            (CADR B)))
        (T(LIST 'PLUS A B))))
    (T(LIST 'PLUS A B))))

(DE TIN
  (A B)
  (COND((NUMBERP B)
      (TIMES A B))
    ((EQ A 0) 0)
    ((EQ A 1) B)
    ((MINUSP A)
      (SIMPLE(LIST 'MINUS
          (LIST 'TIMES
            (MINUS A) B))))
    ((EQ(CAR B)
       'TIMES)
      (COND((NUMBERP(CADR B))
          (LIST 'TIMES
            (TIMES A
              (CADR B))
            (CAR(CDDR B))))
        ((NUMBERP(CAR(CDDR B)))
          (LIST 'TIMES
            (TIMES A
              (CAR(CDDR B)))
            (CADR B)))
        (T(LIST 'TIMES A B))))
    (T(LIST 'TIMES A B))))

(DE EXPT
  (A B)
  (COND((CONSTP B X)
      (LIST 'TIMES B
        (LIST 'TIMES
          (DIFF A X)
          (LIST 'EXPT A
            (LIST 'PLUS B -1)))))
    (T(LIST 'TIMES
        (LIST 'EXPT A B)
        (LIST 'PLUS
          (LIST 'TIMES
            (DIFF B X)
            (LIST 'LN A))
          (LIST 'TIMES B
            (LIST 'QUOTIENT
              (DIFF A X) A)))))))

(DEFPROP EXPT DFUN EXPT)

(DE INFIX
  (L)
  (COND((ATOM L)
       L)
    ((ATOM(CDR L))
      (CAR L))
    ((EQ(CAR L)
       'MINUS)
      (LIST '-
        (INFIX(CADR L))))
    ((ATOM(CDDR L))
      (LIST(CAR L)
        (INFIX(CADR L))))
    ((ATOM(CDR(CDDR L)))
      (LIST(INFIX(CADR L))
        (CDR(ASSOC(CAR L) INFIXASSOC))
        (INFIX(CAR(CDDR L)))))
    (T(CONS(INFIX(CADR L))
        (CONS(CDR(ASSOC(CAR L)
              INFIXASSOC))
          (INFIX(CONS(CAR L)
              (CDDR L))))))))

(SETQ INFIXASSOC
   '((PLUS.+)
      (MINUS.-)
      (TIMES.*)
      (QUOTIENT./)
      (EXPT.↑)
      (LOG.LOG)))

(DE PREFIX
  (L)
  (COND((ATOM L)
```

```
      L)
    ((ATOM(CDR L))
      (CAR L))
    ((EQ(CAR L)
       '-)
      (LIST 'MINUS
        (PREFIX(CADR L))))
    ((ATOM(CDDR L))
      (LIST(CAR L)
        (PREFIX(CADR L))))
    ((ATOM(CDR(CDDR L)))
      (COND((EQ(CADR L)
              '-)
          (LIST 'PLUS
            (PREFIX(CAR L))
            (LIST 'MINUS
              (PREFIX(CAR(CDDR L))))))
        (T(LIST(CDR(ASSOC(CADR L)
                  PREFIXASSOC))
            (PREFIX(CAR L))
            (PREFIX(CAR(CDDR L)))))))
    ((EQ(CADR L)
       '*)
      (PREFIX(CONS(LIST(CAR L)
             '*
             (CAR(CDDR L)))
         (CDR(CDDR L)))))
    ((EQ(CADR L)
       '-)
      (LIST 'PLUS
        (PREFIX(CAR L))
        (PREFIX(CONS(LIST 'MINUS
              (CAR(CDDR L)))
            (CDR(CDDR L))))))
    (T(LIST(CDR(ASSOC(CADR L)
             PREFIXASSOC))
        (PREFIX(CAR L))
        (PREFIX(CDDR L))))))

(SETQ PREFIXASSOC
   '((+.PLUS)
     (*.TIMES)
     (/.QUOTIENT)
     (↑.EXPT)
     (-.MINUS)
     (LOG.LOG)))

(DE DFUNC
  (TM)
  (LIST 'TIMES
    (DIFF A X) TM))

(SETQ DFUNC
   'NIL)

(DE PRINFIX
  (L)
  (COND((ATOM L)
      (MSG L))
    ((EQ(CAR L)
       '-)
      (MSG '-)
      (PRINFIX(CADR L)))
    ((ATOM(CDDR L))
      (MSG "("
        (CAR L) " ")
      (PRINFIX(CADR L))
      (MSG ")"))
    ((EQ(CADR L)
       '+)
      (PRIN+ L))
    ((EQ(CADR L)
       '*)
      (MAPCAR 'PRINFIX L))
    (T(MSG "(")
      (MAPCAR 'PRINFIX L)
      (MSG ")"))))

(DE PRIN+
  (L)
  (PROG NIL
    (MSG "(")
    (PRINFIX(CAR L)) LOOP
    (SETQ L
      (CDR L))
    (COND((NOT(EQ(CAR(CADR L))
            '-))
```

```
        (MSG "+")))
    (SETQ L
      (CDR L))
    (PRINFIX(CAR L))
    (COND((ATOM(CDR L))
        (RETURN(MSG ")"))))
    (GO LOOP)))

(DE ABLFUNC
  (FUNC N)
  (PROG(DFUNC I)
    (SETQ I 1) LOOP
    (COND((GREATERP I N)
        (RETURN NIL)))
    (MSG T T "F"
      (CAR(NTH '("'" "''" "'''"
           "''''" "'''''") I))
      "(X) = ")
    (SETQ DFUNC
      (SIMPLE(DIFF FUNC
          'X)))
    (PRINFIX(INFIX(FLAT DFUNC)))
    (SETQ FUNC DFUNC)
    (SETQ I
      (ADD1 I))
    (GO LOOP)))

(DE FLAT
  (X)
  (COND((ATOM X)
      X)
    ((ATOM(CDR X))
      X)
    ((EQ(CAR X)
       'MINUS)
      (LIST 'MINUS
        (FLAT(CADR X))))
    ((ATOM(CDDR X))
      (LIST(CAR X)
        (FLAT(CADR X))))
    ((EQ(CAR X)
       'EXPT)
      (LIST 'EXPT
        (FLAT(CADR X))
        (FLAT(CAR(CDDR X)))))
    (T(SETQ Y
        (MAPCAR 'FLAT
          (CDR X)))
      (SETQ X
        (CAR X))
      (CONS X
        (MAPCAN '(LAMBDA(Z)
            (COND((ATOM Z)
                (LIST Z))
              ((EQ(CAR Z) X)
                (CDR Z))
              (T(LIST Z)))) Y)))))

(DE CADDR
  (X)
  (CAR(CDDR X)))

(DE DPLUS
  (A B)
  (LIST 'PLUS
    (DIFF A X)
    (DIFF B X)))

(DE SPLUS
  (X Y)
  (COND((NUMBERP X)
      (PLN X Y))
    ((NUMBERP Y)
      (PLN Y X))
    ((EQUAL X Y)
      (SIMPLE(LIST 'TIMES 2 X)))
    ((EQ(CAR Y)
       'MINUS)
      (COND((EQUAL X
          (CADR Y)) 0)
        (T(LIST 'PLUS X Y))))
    ((EQ(CAR Y)
       'PLUS)
      (COND((EQ X
          (CADR Y))
          (SIMPLE(LIST 'PLUS
              (LIST 'TIMES 2 X)
```

```
                (CAR(CDDR Y)))))
          ((EQ X
            (CAR(CDDR Y)))
            (SIMPLE(LIST 'PLUS
                (LIST 'TIMES 2 X)
                (CADR Y))))
          (T(LIST 'PLUS X Y))))
      (T(LIST 'PLUS X Y))))

(DE DMINUS
  (A)
  (LIST 'MINUS
    (DIFF A X)))

(DE SMINUS
  (X)
  (COND((NUMBERP X)
      (MINUS X))
    ((EQ(CAR X)
       'MINUS)
      (CADR X))
    ((EQ(CAR X)
       'PLUS)
      (SIMPLE(LIST 'PLUS
          (LIST 'MINUS
            (CADR X))
          (LIST 'MINUS
            (CADDR X)))))
    (T(LIST 'MINUS X))))

(DE DTIMES
  (A B)
  (LIST 'PLUS
    (LIST 'TIMES
      (DIFF A X) B)
    (LIST 'TIMES
      (DIFF B X) A)))

(DE STIMES
  (X Y)
  (COND((NUMBERP X)
      (TIN X Y))
    ((NUMBERP Y)
      (TIN Y X))
    ((EQUAL X Y)
      (LIST 'EXPT X 2))
    ((EQ(CAR Y)
       'TIMES)
      (COND((EQUAL X
          (CADR Y))
          (SIMPLE(LIST 'TIMES
              (CAR(CDDR Y))
              (LIST 'EXPT X 2))))
        ((EQUAL X
          (CAR(CDDR Y)))
          (SIMPLE(LIST 'TIMES
              (CADR Y)
              (LIST 'EXPT X 2))))
        (T(LIST 'TIMES X Y))))
    ((EQ(CAR Y)
       'MINUS)
      (SIMPLE(LIST 'MINUS
          (LIST 'TIMES X
            (CADR Y)))))
    ((EQ(CAR X)
       'MINUS)
      (SIMPLE(LIST 'MINUS
          (LIST 'TIMES Y
            (CADR X)))))
    ((AND(EQ(CAR Y)
         'EXPT)
      (EQUAL X
        (CADR Y)))
      (LIST 'EXPT X
        (ADD1(CAR(CDDR Y)))))
    ((EQ(CAR Y)
       'QUOTIENT)
      (COND((EQUAL X
          (CADR Y))
          (LIST 'QUOTIENT
            (LIST 'EXPT X 2)
            (CADDR Y)))
        ((EQUAL X
          (CADDR Y))
          (CADR Y))
        (T(LIST 'TIMES X Y))))
    (T(LIST 'TIMES X Y))))
```

```
(DE DQUOTIENT
  (A B)
  (LIST 'QUOTIENT
    (LIST 'PLUS
      (LIST 'TIMES
        (DIFF A X) B)
      (LIST 'MINUS
        (LIST 'TIMES
          (DIFF B X) A)))
    (LIST 'EXPT B 2)))

(DE SQUOTIENT
  (A B)
  (COND((EQUAL A B) 1)
    ((AND(NUMBERP A)
      (NUMBERP B))
      (COND((ZEROP(REMAINDER A B))
          (QUOTIENT A B))
        (T(LIST 'QUOTIENT A B))))
    ((EQ(CAR A)
       'TIMES)
      (COND((EQUAL(CADR A) B)
          (CADDR A))
        ((EQUAL(CADDR A) B)
          (CADR A))
        (T(LIST 'QUOTIENT A B))))
    ((EQ(CAR B)
       'TIMES)
      (COND((EQUAL(CADR B) A)
          (LIST 'QUOTIENT 1
            (CADDR B)))
        ((EQUAL(CADDR B) A)
          (LIST 'QUOTIENT 1
            (CADR B)))
        (T(LIST 'QUOTIENT A B))))
    (T(LIST 'QUOTIENT A B))))

(DE SEXPT
  (X Y)
  (COND((NUMBERP X)
      (EXPT X Y))
    ((EQ Y 0) 1)
    ((EQ Y 1) X)
    ((EQ(CAR X)
       'EXPT)
      (SIMPLE(LIST 'EXPT
          (CADR X)
          (LIST 'TIMES Y
            (CAR(CDDR X))))))
    (T(LIST 'EXPT X Y))))

(DE LOG
  (A B)
  (LIST 'TIMES
    (DIFF B X)
    (LIST 'QUOTIENT 1
      (LIST 'TIMES B
        (LIST 'LN A)))))

(DEFPROP LOG DFUN LOG)

(DE EXP
  (A)
  (DFUNC(LIST 'EXP A)))

(DEFPROP EXP DFUN EXP)

(DE SQRT
  (A)
  (DFUNC(LIST 'QUOTIENT 1
      (LIST 'TIMES 2
        (LIST 'SQRT A)))))

(DEFPROP SQRT DFUN SQRT)

(DE SIN
  (A)
  (DFUNC(LIST 'COS A)))

(DEFPROP SIN DFUN SIN)

(DE COS
  (A)
  (DFUNC(LIST 'MINUS
      (LIST 'SIN A))))
```

```
(DEFPROP COS DFUN COS)

(DE TAN
  (A)
  (DFUNC(LIST 'PLUS 1
       (LIST 'EXPT
         (LIST 'TAN A) 2))))

(DEFPROP TAN DFUN TAN)

(DE COT
  (A)
  (DFUNC(LIST 'MINUS
       (LIST 'PLUS 1
         (LIST 'EXPT
           (LIST 'COT A) 2)))))

(DEFPROP COT DFUN COT)

(DE LN
  (A)
  (LIST 'QUOTIENT
    (DIFF A X) A))

(DEFPROP LN DFUN LN)

(DE LG
  (A)
  (DFUNC(LIST 'QUOTIENT
       (LIST 'LG
          'E) A)))

(DEFPROP LG DFUN LG)

(DE ARCSIN
  (A)
  (DFUNC(LIST 'QUOTIENT 1
       (LIST 'EXPT
         (LIST 'PLUS 1
           (LIST 'MINUS
             (LIST 'EXPT A 2)))
          '(QUOTIENT 1 2)))))

(DEFPROP ARCSIN DFUN ARCSIN)

(DE ARCCOS
  (A)
  (DFUNC(LIST 'QUOTIENT -1
       (LIST 'EXPT
         (LIST 'PLUS 1
           (LIST 'MINUS
             (LIST 'EXPT A 2)))
          '(QUOTIENT 1 2)))))

(DEFPROP ARCCOS DFUN ARCCOS)

(DE ARCTAN
  (A)
  (DFUNC(LIST 'QUOTIENT 1
       (LIST 'PLUS 1
         (LIST 'EXPT A 2)))))

(DEFPROP ARCTAN DFUN ARCTAN)

(DE ARCCOT
  (A)
  (DFUNC(LIST 'QUOTIENT -1
       (LIST 'PLUS 1
         (LIST 'EXPT A 2)))))

(DEFPROP ARCCOT DFUN ARCCOT)

(DE SINH
  (A)
  (DFUNC(LIST 'COSH A)))

(DEFPROP SINH DFUN SINH)

(DE COSH
  (A)
  (DFUNC(LIST 'SINH A)))

(DEFPROP COSH DFUN COSH)

(DE TANH
  (A)
  (DFUNC(LIST 'PLUS 1
```

```
      (LIST 'MINUS
        (LIST 'EXPT
          (LIST 'TANH A) 2)))))

(DEFPROP TANH DFUN TANH)

(DE COTH
  (A)
  (DFUNC(LIST 'PLUS 1
       (LIST 'MINUS
         (LIST 'EXPT
           (LIST 'COTH A) 2)))))

(DEFPROP COTH DFUN COTH)

(DE ARSINH
  (A)
  (DFUNC(LIST 'QUOTIENT 1
       (LIST 'EXPT
         (LIST 'PLUS 1
           (LIST 'EXPT A 2))
         (LIST 'QUOTIENT 1 2)))))

(DEFPROP ARSINH DFUN ARSINH)

(DE ARCOSH
  (A)
  (DFUNC(LIST 'QUOTIENT 1
       (LIST 'EXPT
         (LIST 'PLUS
           (LIST 'EXPT A 2) -1)
         (LIST 'QUOTIENT 1 2)))))

(DEFPROP ARCOSH DFUN ARCOSH)

(DE ARTANH
  (A)
  (DFUNC(LIST 'QUOTIENT 1
       (LIST 'PLUS 1
         (LIST 'MINUS
           (LIST 'EXPT A 2))))))

(DEFPROP ARTANH DFUN ARTANH)

(DE ARCOTH
  (A)
  (DFUNC(LIST 'QUOTIENT 1
       (LIST 'PLUS 1
         (LIST 'MINUS
           (LIST 'EXPT A 2))))))

(DEFPROP ARCOTH DFUN ARCOTH)

(SETQ ABFNS
   '(DO DIFF DIFF-FUNCTION
    SYSTEM-FUNCTIONS CONSTP SIMPLE
    SIASSOC PLN TIN EXPT INFIX
    INFIXASSOC PREFIX PREFIXASSOC
    DFUNC PRINFIX PRIN+ ABLFUNC FLAT
    CADDR DPLUS SPLUS DMINUS SMINUS
    DTIMES STIMES DQUOTIENT SQUOTIENT
    SEXPT LOG EXP SQRT SIN COS TAN
    COT LN LG ARCSIN ARCCOS ARCTAN
    ARCCOT SINH COSH TANH COTH ARSINH
    ARCOSH ARTANH ARCOTH ABFNS))
```

8.5 MACROS.LSP

```
(DE EXPAND NIL
  (SETQ MACRO-EXPANSION T))

(DE NO-EXPAND NIL
  (SETQ MACRO-EXPANSION NIL))

(SETQ MACRO-EXPANSION
   'T)

(DM FOR L
  (REPLACE L
    (PROG(VAR VON NACH COUNT-FN TEST-FN)
      (SETQ VAR
        (CADR L))
      (SETQ VON
        (EVAL(CAR(CDDR L))))
      (SETQ NACH
        (EVAL(CADR(CDDR L))))
      (COND((GREATERP VON NACH)
          (SETQ TEST-FN
             'LESSP)
          (SETQ COUNT-FN
             'SUB1))
        (T(SETQ TEST-FN
             'GREATERP)
          (SETQ COUNT-FN
             'ADD1)))
      (RETURN(LIST 'PROG
          (LIST VAR)
          (LIST 'SETQ VAR VON)
           'LOOP
          (LIST 'COND
            (LIST(LIST TEST-FN VAR NACH)
                '(RETURN NIL))
            (CONS 'T
              (CDDR(CDDR L))))
          (LIST 'SETQ VAR
            (LIST COUNT-FN VAR))
           '(GO LOOP))))))

(DM SELECTQ L
  (REPLACE L
    (CONS 'COND
      ((LABEL SELECTQ1
        (LAMBDA(X L)
          (COND((ATOM(CDR L))
              (LIST(LIST 'T
                   (CAR L))))
            (T(CONS(CONS(LIST(COND((
                       ATOM(CAAR L))
                         'EQ)
                       (T 'MEMBER)) X
                     (LIST 'QUOTE
                       (CAAR L)))
                   (CDAR L))
                 (SELECTQ1 X
                   (CDR L)))))))
        (CADR L)
        (CDDR L)))))

(DE REPLACE
  (X Y)
  (COND(MACRO-EXPANSION(RPLACA X
         (CAR Y))
       (RPLACD X
         (CDR Y)))
    (T Y)))

(DM IF L
  (REPLACE L
    (LIST 'COND
      (LIST(CADR L)
        (CAR(CDDR L)))
      (CONS 'T
        (CDR(CDDR L))))))

(DM WHILE L
  (REPLACE L
    (LIST 'PROG NIL
       'LOOP
      (LIST 'COND
        (LIST(LIST 'NOT
```

```
            (CADR L))
          (LIST 'RETURN NIL))
        (CONS 'T
          (CDDR L)))
       '(GO LOOP))))

(DM REPEAT L
  (REPLACE L
    (LIST 'PROG
       '(N)
      (LIST 'SETQ
         'N
        (EVAL(CADR L)))
       'LOOP
      (LIST 'COND
        (LIST '(ZEROP N)
           '(RETURN NIL))
        (CONS 'T
          (CDDR L)))
       '(SETQ N
        (SUB1 N))
       '(GO LOOP))))

(DM LET L
  (REPLACE L
    (CONS(CONS 'LAMBDA
        (CONS(MAPCAR 'CAR
            (CADR L))
          (CDDR L)))
      (MAPCAR 'CADR
        (CADR L)))))

(DM LOCAL L
  (REPLACE L
    (CONS(CONS 'LAMBDA
        (CDR L)) NIL)))

(DM INCR L
  (REPLACE L
    (LIST 'SETQ
      (CADR L)
      (LIST 'ADD1
        (CADR L)))))

(DM DECR L
  (REPLACE L
    (LIST 'SETQ
      (CADR L)
      (LIST 'SUB1
        (CADR L)))))

(DM PUSH L
  (REPLACE L
    (LIST 'SETQ
      (CADR L)
      (LIST 'CONS
        (CAR(CDDR L))
        (CADR L)))))

(DM POP L
  (REPLACE L
    (LIST 'PROG1
      (LIST 'CAR
        (CADR L))
      (LIST 'SETQ
        (CADR L)
        (LIST 'CDR
          (CADR L))))))

(DM MCONS L
  (REPLACE L
    (COND((ATOM(CDDR L))
        (CADR L))
      (T(LIST 'CONS
          (CADR L)
          (CONS 'MCONS
            (CDDR L)))))))

(DM NCONS L
  (REPLACE L
    (LIST 'CONS
      (CADR L) NIL)))

(DM XCONS L
  (REPLACE L
    (LIST 'CONS
      (CAR(CDDR L))
```

```
      (CADR L))))

(DM FUNCTION L
  (REPLACE L
    (LIST 'QUOTE
      (CADR L))))

(DM F:L L
  (REPLACE L
    (LIST 'QUOTE
      (CONS 'LAMBDA
        (CDR L)))))

(DM Q:L L
  (REPLACE L
    (LIST 'QUOTE
      (CONS 'LAMBDA
        (CDR L)))))

(DM NEQ L
  (REPLACE L
    (LIST 'NOT
      (LIST 'EQ
        (CADR L)
        (CAR(CDDR L))))))

(SETQ MACROS
   '(EXPAND NO-EXPAND MACRO-EXPANSION
    FOR SELECTQ REPLACE IF WHILE
    REPEAT LET LOCAL INCR DECR PUSH
    POP MCONS NCONS XCONS FUNCTION
    F:L Q:L NEQ MACROS))
```

8.6 TRACER.LSP

```
(DF TRACE L
  (SETQ TRACE-SPACES 0)
  (NO-SINGLE-STEP)
  (MAPC '(LAMBDA(FUNC)
       (PROG(X)
         (SETQ X
           (OR(GETPROP FUNC
                  'EXPR)
              (GETPROP FUNC
                  'FEXPR)
              (GETPROP FUNC
                  'MACRO) NIL))
         (COND((NULL X)
             (RETURN NIL)))
         (RPLACD(CDR X)
           (LIST(LIST 'EVTRACE FUNC
                (CADR X)
                (CDDR X)))))) L) L)

(DF UNTRACE L
  (SETQ TRACE-SPACES 0)
  (MAPC '(LAMBDA(FUNC)
       (PROG(X)
         (SETQ X
           (OR(GETPROP FUNC
                  'EXPR)
              (GETPROP FUNC
                  'FEXPR)
              (GETPROP FUNC
                  'MACRO) NIL))
         (COND((NULL X)
             (RETURN NIL)))
         (RPLACD(CDR X)
           (LAST(LAST X))))) L) L)

(DF EVTRACE
  (TRFUN TRVARS TRBODY)
  (PROG(TRRESULT)
    (PRINTENTRY TRFUN TRVARS)
    (SETQ TRRESULT
      (APPLY 'PROGN TRBODY))
    (PRINTEXIT TRFUN TRRESULT)
    (RETURN TRRESULT)))

(DF TRACE L
  (SETQ TRACE-SPACES 0)
  (NO-SINGLE-STEP)
  (MAPC '(LAMBDA(FUNC)
       (PROG(X)
         (SETQ X
           (OR(GETPROP FUNC
                  'EXPR)
              (GETPROP FUNC
                  'FEXPR)
              (GETPROP FUNC
                  'MACRO) NIL))
         (COND((NULL X)
             (RETURN NIL)))
         (RPLACD(CDR X)
           (LIST(LIST 'EVTRACE FUNC
                (CADR X)
                (CDDR X)))))) L) L)

(DF UNTRACE L
  (SETQ TRACE-SPACES 0)
  (MAPC '(LAMBDA(FUNC)
       (PROG(X)
         (SETQ X
           (OR(GETPROP FUNC
                  'EXPR)
              (GETPROP FUNC
                  'FEXPR)
              (GETPROP FUNC
                  'MACRO) NIL))
         (COND((NULL X)
             (RETURN NIL)))
         (RPLACD(CDR X)
           (LAST(LAST X))))) L) L)

(DF EVTRACE
  (TRFUN TRVARS TRBODY)
  (PROG(TRRESULT)
```

```
    (PRINTENTRY TRFUN TRVARS)
    (SETQ TRRESULT
      (APPLY 'PROGN TRBODY))
    (PRINTEXIT TRFUN TRRESULT)
    (RETURN TRRESULT)))

(DE PRINTENTRY
  (TRFUN TRVARS)
  (SPACES(SETQ TRACE-SPACES
      (ADD1 TRACE-SPACES)))
  (MSG "ENTERING " TRFUN " [")
  (PRINTENTRY1 TRVARS)
  (MSG "]" T)
  (COND(SINGLE-STEP-V(WAITCHAR))))

(DE PRINTENTRY1
  (TRVARS)
  (COND((NULL TRVARS)
        NIL)
    ((ATOM TRVARS)
      (PRIN1(EVAL TRVARS)))
    ((ATOM(CDR TRVARS))
      (PRIN1(EVAL(CAR TRVARS))))
    (T(PRIN1(EVAL(CAR TRVARS)))
      (MSG ",")
      (PRINTENTRY1(CDR TRVARS)))))

(DE PRINTEXIT
  (TRFUN TRRESULT)
  (SPACES(SETQ TRACE-SPACES
      (SUB1 TRACE-SPACES)))
  (MSG " EXITING  " TRFUN " = ")
  (PRINT TRRESULT)
  (COND(SINGLE-STEP-V(WAITCHAR))))

(DE SINGLE-STEP NIL
  (SETQ SINGLE-STEP-V T))

(DE NO-SINGLE-STEP NIL
  (SETQ SINGLE-STEP-V NIL))

(SETQ SINGLE-STEP-V
   'NIL)

(SETQ TRACFNS
   '(TRACE UNTRACE EVTRACE PRINTENTRY
    PRINTENTRY1 PRINTEXIT SINGLE-STEP
    NO-SINGLE-STEP SINGLE-STEP-V
    TRACFNS))
```

8.7 ARRAYS.LSP

```
(DF ARRAY L
  (PUTPROP(CAR L)
     'ARRAY
    (DIM(MAPCAR 'EVAL
        (CDDR L))
      (EVAL(CADR L))))
  (CAR L))

(DF LOD L
  (LOD1(GETPROP(CAR L)
       'ARRAY)
    (MAPCAR 'EVAL
      (CDR L))))

(DF STO L
  (STO1(GETPROP(CAR L)
       'ARRAY)
    (EVAL(CADR L))
    (MAPCAR 'EVAL
      (CDDR L))))

(DE DIM
  (NLIS E)
  (COND((ATOM NLIS)
       (COPY E))
    (T(BUILD(CAR NLIS)
        (DIM(CDR NLIS) E)))))

(DE BUILD
  (N E)
  (COND((ZEROP N)
       NIL)
    (T(CONS(COPY E)
        (BUILD(SUB1 N) E)))))

(DE STO1
  (L E DIMS)
  (COND((ATOM DIMS)
       NIL)
    ((ATOM(CDR DIMS))
      (RPLACA(NTH L
          (CAR DIMS)) E))
    (T(STO1(CAR(NTH L
             (CAR DIMS))) E
        (CDR DIMS)))))

(DE LOD1
  (L DIMS)
  (COND((ATOM DIMS)
       L)
    (T(LOD1(CAR(NTH L
             (CAR DIMS)))
        (CDR DIMS)))))

(DF FOR L
  (PROG(VAR FST LST EXPRS TEST-FN
      COUNT-FN)
    (SETQ VAR
      (CAR L))
    (SETQ FST
      (EVAL(CADR L)))
    (SETQ LST
      (EVAL(CAR(CDDR L))))
    (COND((LESSP LST FST)
        (SETQ TEST-FN
           'LESSP)
        (SETQ COUNT-FN
           'SUB1))
      (T(SETQ TEST-FN
           'GREATERP)
        (SETQ COUNT-FN
           'ADD1)))
    (SETQ EXPRS
      (CDR(CDDR L))) LOOP
    (COND((TEST-FN FST LST)
        (RETURN NIL)))
    (SET VAR FST)
    (MAPC 'EVAL EXPRS)
    (SETQ FST
      (COUNT-FN FST))
    (GO LOOP)))
```

```
(DF WHILE L
  (PROG(CON EXPRS)
    (SETQ CON
      (CAR L))
    (SETQ EXPRS
      (CDR L)) LOOP
    (COND((EVAL CON)
        (MAPC 'EVAL EXPRS)
        (GO LOOP)))))

(DF IF L
  (COND((EVAL(CAR L))
      (EVAL(CADR L)))
    (T(LAST(MAPCAR 'EVAL
          (CDDR L))))))

(SETQ ARRFNS
   '(ARRAY LOD STO DIM BUILD STO1
    LOD1 FOR WHILE IF ARRFNS))
```

8.8 SETS.LSP

```
(SETQ SETFNS
   '(SETFNS MEM1 SUBSET SYMM-DIFF
    UNION INTERSECTION MAKESET SETEQ
    SETP SUBSETP ENTER ATTACH INSERT
    DREMOVE))

(DE MEM1
  (L1 L2)
  (COND((ATOM L1)
        NIL)
    ((MEMBER(CAR L1) L2))
    (T(MEM1(CDR L1) L2))))

(DE SUBSET
  (FUN L)
  (COND((ATOM L)
        NIL)
    ((APPLY* FUN
       (CAR L))
       (CONS(CAR L)
         (SUBSET FUN
           (CDR L))))
    (T(SUBSET FUN
        (CDR L)))))

(DE SYMM-DIFF
  (L1 L2)
  (COND((ATOM L1)
        NIL)
    ((MEMBER(CAR L1) L2)
      (SYMM-DIFF(CDR L1) L2))
    (T(CONS(CAR L1)
        (SYMM-DIFF(CDR L1) L2)))))

(DE UNION
  (L1 L2)
  (COND((ATOM L1)
        L2)
    ((MEMBER(CAR L1) L2)
      (UNION(CDR L1) L2))
    (T(CONS(CAR L1)
        (UNION(CDR L1) L2)))))

(DE INTERSECTION
  (L1 L2)
  (COND((ATOM L1)
        NIL)
    ((MEMBER(CAR L1) L2)
      (CONS(CAR L1)
        (INTERSECTION(CDR L1) L2)))
    (T(INTERSECTION(CDR L1) L2))))

(DE MAKESET
  (L1)
  (COND((ATOM L1)
        NIL)
    ((NOT(MEMBER(CAR L1)
           (CDR L1)))
      (CONS(CAR L1)
        (MAKESET(CDR L1))))
    (T(MAKESET(CDR L1)))))

(DE SETEQ
  (L1 L2)
  (COND((EQUAL L1 L2))
    ((ATOM L1)
      (ATOM L2))
    ((MEMBER(CAR L1) L2)
      (SETEQ(CDR L1)
        (REMOVE(CAR L1) L2)))))

(DE SETP
  (L1)
  (COND((NULL L1)
        T)
    ((MEMBER(CAR L1)
      (CDR L1))
       NIL)
    (T(SETP(CDR L1)))))

(DE SUBSETP
  (L1 L2)
```

```
  (COND((EQUAL L1 L2))
    ((ATOM L1))
    ((MEMBER(CAR L1) L2)
      (SUBSETP(CDR L1) L2))))

(DE ENTER
  (X L)
  (COND((MEMBER X L)
        L)
    (T(ATTACH X L))))

(DE ATTACH
  (X L)
  (COND((ATOM L)
        (CONS X NIL))
    (T(RPLACD L
        (CONS(CAR L)
          (CDR L)))
      (RPLACA L X))))

(DE INSERT
  (X Y L)
  (ATTACH X
    (NTH L Y)) L)

(DE DREMOVE
  (L1 L2)
  (PROG(L3 L4)
    (SETQ L4 L2) LOOP
    (COND((ATOM L4)
        (RETURN L2))
      ((SETQ L3
        (MEMBER L1 L4))
        (COND((ATOM(CDR L3))
            (RETURN L2))
          (T(RPLACA L3
              (CADR L3))
            (RPLACD L3
              (CDDR L3)))))
      (T(SETQ L4
          (CDR L4))))
    (GO LOOP)))
```

8.9 EDITOR.LSP

```
(DE NX NIL
  (COND((ATOM(CDR CLU)))
    (T(SETQ CLU
        (CDR CLU))
      (SETQ CL
        (CAR CLU)))))

(DE FIND
  (L I X W)
  (COND((ATOM L)
        NIL)
    ((MEMBER I
       (CAR L))
       (LIST X))
    ((SETQ W
       (FIND(CAR L) I 1))
       (CONS X W))
    (T(FIND(CDR L) I
        (ADD1 X)))))

(DF FI
  (I)
  (MAPC 'G
    (FIND CL I 1 NIL))
  (P))

(DF B L
  (RPLACA CLU
    (SETQ CL
      (CONC L
        (CAR CLU))))
  (BACK))

(SETQ B
   '(A B
    (C D E) F G))

(DF :
  (L)
  (RPLACA CLU
    (SETQ CL L)))

(DE LO
  (X Y Z)
  (SETQ Z
    (NTH CL
      (H X)))
  (SETQ Y
    (NTH CL
      (H(SUB1 X))))
  (COND((CONSP(CAR Z))
       (RPLACD Y
         (CAR Z)))))

(DE LI
  (X)
  (BI X -1))

(DF R
  (X Y)
  (REPL X Y CL))

(DE REPL
  (X Y L)
  (COND((ATOM L)
        NIL)
    ((EQUAL(CAR L) X)
      (RPLACA L Y)
      (REPL X Y
        (CDR L)))
    ((EQUAL(CDR L) X)
      (RPLACD L Y))
    (T(REPL X Y
        (CAR L))
      (REPL X Y
        (CDR L)))))

(DE RO
  (X Y)
  (COND((NULL Y)
       (SETQ Y
         (LENGTH CL))))
```

```
  (SETQ X
    (NTH CL
      (H X)))
  (SETQ Y
    (NTH CL
      (H Y)))
  (RPLACA X
    (CONC(CAR X)
      (CDR X)))
  (RPLACD X
    (CDR Y))
  (RPLACD Y NIL))

(DE BO
  (X)
  (SETQ X
    (NTH CL
      (H X)))
  (SETQ Y
    (CONC(CAR X)
      (CDR X)))
  (RPLACA X
    (CAR Y))
  (RPLACD X
    (CDR Y)))

(DE BI
  (X Y)
  (COND((NULL Y)
      (SETQ Y X)))
  (SETQ X
    (NTH CL
      (H X)))
  (SETQ Y
    (NTH CL
      (H Y)))
  (SETQ Z
    (CDR Y))
  (RPLACD Y NIL)
  (RPLACA X
    (CONS(CAR X)
      (CDR X)))
  (RPLACD X Z))

(DE RI
  (X Y)
  (SETQ X
    (NTH CL
      (H X)))
  (SETQ Y
    (NTH(CAR X)
      (H Y)))
  (SETQ Z
    (CDR X))
  (RPLACD X
    (CDR Y))
  (CONC(CDR Y) Z)
  (RPLACD Y NIL))

(DE EXPT
  (X Y)
  (COND((EQ Y 0) 1)
    (T(TIMES X
        (EXPT X
          (SUB1 Y))))))

(DF E
  (L)
  (PRINT(EVAL L)))

(SETQ E
   '(NLAMBDA(L)
    (PRINT(EVAL L))))

(DE ← NIL
  (SETQ CLU
    (LAST TR))
  (SETQ TR
    (LIST CLU))
  (SETQ CL
    (CAR CLU)))

(DF N L
  (CONC CL L))

(DF A L
  (RPLACD CLU
```

```
    (CONC L
      (CDR CLU)))
  (BACK))

(SETQ A
   '(NLAMBDA L
    (RPLACD CLU
      (CONC L
        (CDR CLU)))
    (BACK)))

(DE CONC
  (L1 L2)
  (COND((ATOM L1)
        L2)
    ((ATOM L2)
       L1)
    (T(NCONC L1 L2))))

(DE DEL
  (X L)
  (SETQ X
    (H X))
  (COND((ATOM CL)
        CL)
    ((ZEROP X)
      (RPLACA CLU
        (SETQ CL
          (CONC L CL))))
    ((EQ X 1)
      (RPLACA CLU
        (SETQ CL
          (CONC L
            (CDR CL)))))
    (T(RPLACD(NTH CL
           (SUB1 X))
        (CONC L
          (NTH CL
            (ADD1 X)))))))

(DE UNDO NIL
  (SETQ LIS
    (COPY OLD))
  (SETQ CLU
    (LIST LIS))
  (SETQ TR
    (LIST CLU))
  (SETQ CL
    (CAR CLU)))

(DE OUT NIL
  (SAVE 8 "@0:EDITOR.LSP" EDFNS))

(DF ADD L
  (COND((ATOM L)
        EDFNS)
    (T(SETQ EDFNS
        (CONS(CAR L) EDFNS))
      (APPLY 'ADD
        (CDR L)))))

(DE P@ NIL
  (PP CL))

(DE BACK NIL
  (COND((ATOM(CDR TR))
        CL)
    (T(SETQ CLU
        (CAR TR))
      (SETQ TR
        (CDR TR))
      (SETQ CL
        (CAR CLU)))))

(DE G
  (X)
  (SETQ X
    (H X))
  (COND((ZEROP X)
        (BACK))
    ((GREATERP X
      (LENGTH CL))
       CL)
    (T(SETQ TR
        (CONS CLU TR))
      (SETQ CLU
        (NTH CL X))
```

```
      (SETQ CL
        (CAR CLU)))))

(DE P NIL
  (PRINT(P& CL)))

(SETQ P
   '(LAMBDA NIL
    (PRINT(P& CL))))

(DE P&
  (L)
  (COND((ATOM L)
        L)
    (T(CONS(P&(CAR L))
        (MAPCAR '(LAMBDA(X)
            (COND((ATOM X)
                  X)
              (T '&)))
          (CDR L))))))

(DE H
  (X)
  (COND((MINUSP X)
      (SETQ X
        (ABS(PLUS 1 X
              (LENGTH CL))))))
  (COND((GREATERP X
      (LENGTH CL))
      (LENGTH CL))
    (T X)))

(SETQ EDFNS
   '(NX FIND FI B : LO LI R REPL RO
    BO BI RI EXPT E ← N A CONC DEL
    UNDO OUT ADD P@ BACK G P P& H
    EDFNS EDIT EDITF EDITV EDITP))

(DE EDIT
  (L)
  (PROG(OLD TR CL CLU E X LIS)
    (SETQ OLD L)
    (SETQ LIS
      (COPY L))
    (SETQ CLU
      (LIST LIS))
    (SETQ TR
      (LIST CLU))
    (SETQ CL
      (CAR CLU))
    (P) LOOP1
    (MSG "*ED*: ")
    (SETQ E
      (READL)) LOOP2
    (COND((ATOM E)
        (GO LOOP1)))
    (SETQ X
      (CAR E))
    (COND((NUMBERP X)
        (G X))
      ((EQ X
         'OK)
        (RETURN(CAR(LAST TR))))
      ((EQ X
         'PP)
        (PP CL))
      ((ATOM X)
        (EVAL(LIST X)))
      ((NUMBERP(CAR X))
        (DEL(CAR X)
          (CDR X)))
      (T(EVAL X)))
    (SETQ E
      (CDR E))
    (GO LOOP2)))

(DF EDITF
  (F L)
  (COND((SETQ L
      (APPLY 'GETDEF
        (LIST F)))
      (EVAL(CONS(CAR L)
          (CONS(CADR L)
            (EDIT(CDDR L)))))))))

(DF EDITV
  (F)
```

```
  (SET F
    (EDIT(EVAL F))))

(DF EDITP
  (A P)
  (PUTPROP A P
    (EDIT(GETPROP A P))))
```

8.10 PRINT-FILE.LSP

```
(DE PRINT-ON-SCREEN
  (GA FN)
  (PROG(E)
    (COND((EQ GA 8)
        (OPEN 1 GA 2
          (PACK(LIST FN ",U,R"))))
      (T(OPEN 1 GA 0 FN)))
    (INPUT 1) LOOP
    (SETQ E
      (READ))
    (COND((ATOM E)
        (CLOSE 1)
        (NORMAL)
        (RETURN T)))
    (PRINT-PROPS(CAR E)
      (CDR E))
    (GO LOOP)))

(DE PRINT-FILE
  (GA FN)
  (PROG(E)
    (COND((EQ GA 8)
        (OPEN 1 GA 2
          (PACK(LIST FN ",U,R"))))
      (T(OPEN 1 GA 0 FN)))
    (OPEN 2 4) LOOP
    (INPUT 1)
    (SETQ E
      (READ))
    (NORMAL)
    (COND((ATOM E)
        (CLOSE 1)
        (CLOSE 2)
        (RETURN T)))
    (OUTPUT 2)
    (PRINT-PROPS(CAR E)
      (CDR E))
    (NORMAL)
    (GO LOOP)))

(DE PRINT-PROPS
  (A P)
  (COND((NULL P)
       NIL)
    (T(PP(COND((EQ(CAR P)
               'EXPR)
             (CONS 'DE
               (CONS A
                 (CDR(CADR P)))))
          ((EQ(CAR P)
              'FEXPR)
            (CONS 'DF
              (CONS A
                (CDR(CADR P)))))
          ((EQ(CAR P)
              'MACRO)
            (CONS 'DM
              (CONS A
                (CDR(CADR P)))))
          ((EQ(CAR P)
              'VALUE)
            (LIST 'SETQ A
              (LIST 'QUOTE
                (CADR P))))
          (T(LIST 'DEFPROP A
              (CAR P)
              (CADR P)))))
      (TERPRI)
      (PRINT-PROPS A
        (CDDR P)))))
```

8.11 EXPERTE.LSP

```
(DE DO NIL
  (PROG(IN RULESUSED RULE) LOOP
    (MSG ":← ")
    (SETQ IN
      (READL))
    (COND((MEMBER(CAR IN)
          '(OK ENDE FERTIG))
        (RETURN T))
      ((EQ(LAST IN)
          ':)
        (MSG ":+ ")
        (SETQ IN
          (APPEND IN
            (READL)))))
    (EVAL(TO-DO IN))
    (GO LOOP)))

(DE REMEMBER
  (NEW)
  (COND((MEMBER NEW FACTS)
      NIL)
    (T(SETQ FACTS
        (CONS NEW FACTS)) NEW)))

(DE RECALL
  (FACT)
  (COND((MEMBER FACT FACTS)
       FACT)))

(DE TESTIF
  (RULE)
  (PROG(IFS)
    (SETQ IFS
      (CAR RULE)) LOOP
    (COND((NULL IFS)
        (RETURN T))
      ((RECALL(CAR IFS)))
      (T(RETURN NIL)))
    (SETQ IFS
      (CDR IFS))
    (GO LOOP)))

(DE USETHEN
  (RULE)
  (PROG(THENS SUCCESS)
    (SETQ THENS
      (CADR RULE)) LOOP
    (COND((NULL THENS)
        (RETURN SUCCESS))
      ((REMEMBER(CAR THENS))
        (SETQ SUCCESS T)
        (PRINT-RULE RULE)
        (MSG T "DEDUZIERT " T "--> ")
        (PRINC(CAR THENS))
        (TERPRI)))
    (SETQ THENS
      (CDR THENS))
    (GO LOOP)))

(DE TRYRULE
  (RULE)
  (COND((TESTIF RULE)
      (REMEMBERRULE RULE)
      (USETHEN RULE))))

(DE STEPFORWARD NIL
  (PROG(RULELIST)
    (SETQ RULELIST RULES) LOOP
    (COND((NULL RULELIST)
        (RETURN NIL))
      ((TRYRULE(CAR RULELIST))
        (RETURN T)))
    (SETQ RULELIST
      (CDR RULELIST))
    (GO LOOP)))

(DE DEDUCE NIL
  (SETQ RULESUSED NIL)
  (PROG(PROGRESS) LOOP
    (COND((STEPFORWARD)
        (SETQ PROGRESS T))
      (T(RETURN PROGRESS)))
```

```
    (GO LOOP)))

(SETQ RULES
   'NIL)

(SETQ FACTS
   'NIL)

(SETQ HYPS
   'NIL)

(DE VERIFY
  (FACT)
  (PROG(RELEVANT1 RELEVANT2)
    (COND((RECALL FACT)
        (RETURN T)))
    (SETQ RELEVANT1
      (INTHEN FACT RULES))
    (SETQ RELEVANT2 RELEVANT1)
    (COND((NULL RELEVANT1)
        (COND((MEMBER FACT ASKED)
            (RETURN NIL))
          ((ASK FACT)
            (REMEMBER FACT)
            (RETURN T))
          (T(SETQ ASKED
              (CONS FACT ASKED))
            (RETURN NIL)))))) LOOP1
    (COND((NULL RELEVANT1)
        (GO LOOP2))
      ((TRYRULE(CAR RELEVANT1))
        (RETURN T)))
    (SETQ RELEVANT1
      (CDR RELEVANT1))
    (GO LOOP1) LOOP2
    (COND((NULL RELEVANT2)
        (GO EXIT))
      ((TRYRULE+(CAR RELEVANT2))
        (RETURN T)))
    (SETQ RELEVANT2
      (CDR RELEVANT2))
    (GO LOOP2) EXIT
    (RETURN NIL)))

(DE TRYRULE+
  (RULE)
  (COND((TESTIF+ RULE)
      (REMEMBERRULE RULE)
      (USETHEN RULE))))

(DE TESTIF+
  (RULE)
  (PROG(IFS)
    (SETQ IFS
      (CAR RULE)) LOOP
    (COND((NULL IFS)
        (RETURN T))
      ((VERIFY(CAR IFS)))
      (T(RETURN NIL)))
    (SETQ IFS
      (CDR IFS))
    (GO LOOP)))

(DE INIF
  (FACT R)
  (MAPCAN '(LAMBDA(X)
      (COND((MEMBER FACT
          (CAR X))
          (LIST X)))) R))

(DE INTHEN
  (FACT R)
  (MAPCAN '(LAMBDA(X)
      (COND((MEMBER FACT
          (CADR X))
          (LIST X)))) R))

(DE DIAGNOSE NIL
  (SETQ RULESUSED NIL)
  (PROG(ASKED POS)
    (SETQ POS HYPS) LOOP
    (COND((NULL POS)
        (MSG T
          "KEINE HYPOTHESE KANN BEWIESEN WERDEN" T)
        (RETURN NIL))
      ((VERIFY(CAR POS))
        (MSG T "DIE HYPOTHESE :" T
```

```
          "--> ")
        (PRINC(CAR POS))
        (MSG T "IST WAHR" T)
        (RETURN(CAR POS))))
    (MSG T "DIE HYPOTHESE :" T "--> ")
    (PRINC(CAR POS))
    (MSG T
      "KANN NICHT BEWIESEN WERDEN" T)
    (SETQ POS
      (CDR POS))
    (GO LOOP)))

(SETQ DATA
   '(RULES FACTS HYPS))

(DE ASK
  (S)
  (MSG T "IST DIES WAHR (J/N/W) :" T)
  (PRINC S)
  (SETQ CH
    (WAITCHAR))
  (MSG T CH T)
  (COND((EQ CH "J")
        T)
    ((EQ CH "N")
       F)
    (T(TELLWHY)
      (ASK S))))

(DE TELLWHY NIL
  (MSG T "ICH VERSUCHE ZU BEWEISEN :"
    T)
  (PRINC(CAR POS))
  (MSG T T "ICH TESTE :" T)
  (PRINT-RULE RULE))

(DE PRINT-FACTS NIL
  (COND(FACTS(PRLIST FACTS))
    (T(MSG
        "ES SIND KEINE FAKTEN VORHANDEN!" T))))

(DE PRINT-HYPS NIL
  (COND(HYPS(PRLIST HYPS))
    (T(MSG
        "NOCH WURDE KEINE HYPOTHESE AUFGESTELLT!" T))))

(DE PRINT-RULES NIL
  (COND(RULES(MAPC 'PRINT-RULE RULES))
    (T(MSG "BITTE REGELN EINGEBEN!" T))))

(DE PRINT-RULE
  (R)
  (MSG T "+++ REGEL "
    (CAR(CDDR R)) " +++" T)
  (PRINT-IF(CAR R))
  (PRINT-THEN(CADR R)))

(DE PRLIST
  (L)
  (MAPC '(LAMBDA(X)
      (PRINC X)
      (TERPRI)) L))

(DE PRINT-RULE-N
  (N)
  (COND((SETQ X
      (GET-RULE-N N RULES))
      (PRINT-RULE X))
    (T(MSG
        "ES GIBT NOCH KEINE REGEL " N
        T))))

(DE GET-RULE-N
  (N R)
  (PROG NIL LOOP
    (COND((NULL R)
        (RETURN NIL))
      ((EQ N
        (CAR(CDDR(CAR R))))
        (RETURN(CAR R))))
    (SETQ R
      (CDR R))
    (GO LOOP)))

(DE PRINT-IF
  (IFS)
  (PRINC(CONS 'WENN
```

```
      (CONS ':
        (CAR IFS))))
  (TERPRI)
  (MAPC '(LAMBDA(X)
      (PRINC(APPEND '(UND WENN :) X))
      (TERPRI))
    (CDR IFS)))

(DE PRINT-THEN
  (THENS)
  (PRINC(CONS 'DANN
      (CONS ':
        (CAR THENS))))
  (TERPRI)
  (MAPC '(LAMBDA(X)
      (PRINC(APPEND '(UND DANN :) X))
      (TERPRI))
    (CDR THENS)))

(DF FORGET-FACT L
  (SETQ FACTS
    (REMOVE L FACTS)))

(DF FORGET-HYP L
  (SETQ HYPS
    (REMOVE L HYPS)))

(DE FORGET-FACTS NIL
  (SETQ FACTS NIL))

(DE FORGET-HYPS NIL
  (SETQ HYPS NIL))

(DE FORGET-RULES NIL
  (SETQ RULES NIL)
  (SETQ RULE NIL))

(DE FORGET-RULE
  (N)
  (SETQ RULES
    (REMOVE(GET-RULE-N N RULES) RULES)))

(DE WHAT? NIL
  (MSG "WAS SOLL ICH TUN ?" T))

(DF HOW FACT
  (COND((SETQ X
      (INTHEN FACT RULESUSED))
      (MSG "MIT DEN FAKTEN :" T)
      (MAPC '(LAMBDA(Y)
          (PRLIST(CAR Y))) X))
    ((MEMBER FACT FACTS)
      (MSG "DAS FAKTUM WAR GEGEBEN" T))
    (T(MSG
        "DAS HABE ICH NICHT DEDUZIERT" T))))

(DF WHY FACT
  (COND((MEMBER FACT HYPS)
      (MSG
        "ES WAR EINE DER HYPOTHESEN" T))
    ((SETQ X
      (INIF FACT RULESUSED))
      (MSG "ES FOLGT DARAUS :" T)
      (MAPC '(LAMBDA(Y)
          (PRLIST(CADR Y))) X))
    (T(MSG
        "DAS HABE ICH NICHT BENUTZT" T))))

(DF WHICH NIL
  (COND((NULL RULESUSED)
      (MSG "KEINE" T))
    (T(MSG "DIE REGELN ")
      (PRINC(MAPCAR 'LAST RULESUSED))
      (TERPRI))))

(DE DO NIL
  (PROG(IN RULESUSED RULE) LOOP
    (MSG ":← ")
    (SETQ IN
      (READL))
    (COND((MEMBER(CAR IN)
          '(OK ENDE FERTIG))
        (RETURN T))
      ((EQ(LAST IN)
          ':)
        (MSG ":+ ")
        (SETQ IN
```

```
          (APPEND IN
            (READL)))))
    (EVAL(TO-DO IN))
    (GO LOOP)))

(DE REMEMBER
  (NEW)
  (COND((MEMBER NEW FACTS)
        NIL)
    (T(SETQ FACTS
        (CONS NEW FACTS)) NEW)))

(DE RECALL
  (FACT)
  (COND((MEMBER FACT FACTS)
        FACT)))

(DE TESTIF
  (RULE)
  (PROG(IFS)
    (SETQ IFS
      (CAR RULE)) LOOP
    (COND((NULL IFS)
        (RETURN T))
      ((RECALL(CAR IFS)))
      (T(RETURN NIL)))
    (SETQ IFS
      (CDR IFS))
    (GO LOOP)))

(DE USETHEN
  (RULE)
  (PROG(THENS SUCCESS)
    (SETQ THENS
      (CADR RULE)) LOOP
    (COND((NULL THENS)
        (RETURN SUCCESS))
      ((REMEMBER(CAR THENS))
        (SETQ SUCCESS T)
        (PRINT-RULE RULE)
        (MSG T "DEDUZIERT " T "--> ")
        (PRINC(CAR THENS))
        (TERPRI)))
    (SETQ THENS
      (CDR THENS))
    (GO LOOP)))

(DE TRYRULE
  (RULE)
  (COND((TESTIF RULE)
        (REMEMBERRULE RULE)
        (USETHEN RULE))))

(DE STEPFORWARD NIL
  (PROG(RULELIST)
    (SETQ RULELIST RULES) LOOP
    (COND((NULL RULELIST)
        (RETURN NIL))
      ((TRYRULE(CAR RULELIST))
        (RETURN T)))
    (SETQ RULELIST
      (CDR RULELIST))
    (GO LOOP)))

(DE DEDUCE NIL
  (SETQ RULESUSED NIL)
  (PROG(PROGRESS) LOOP
    (COND((STEPFORWARD)
        (SETQ PROGRESS T))
      (T(RETURN PROGRESS)))
    (GO LOOP)))

(SETQ RULES
   'NIL)

(SETQ FACTS
   'NIL)

(SETQ HYPS
   'NIL)

(DE VERIFY
  (FACT)
  (PROG(RELEVANT1 RELEVANT2)
    (COND((RECALL FACT)
        (RETURN T)))
    (SETQ RELEVANT1
```

```
      (INTHEN FACT RULES))
    (SETQ RELEVANT2 RELEVANT1)
    (COND((NULL RELEVANT1)
        (COND((MEMBER FACT ASKED)
            (RETURN NIL))
          ((ASK FACT)
            (REMEMBER FACT)
            (RETURN T))
          (T(SETQ ASKED
              (CONS FACT ASKED))
            (RETURN NIL))))) LOOP1
    (COND((NULL RELEVANT1)
        (GO LOOP2))
      ((TRYRULE(CAR RELEVANT1))
        (RETURN T)))
    (SETQ RELEVANT1
      (CDR RELEVANT1))
    (GO LOOP1) LOOP2
    (COND((NULL RELEVANT2)
        (GO EXIT))
      ((TRYRULE+(CAR RELEVANT2))
        (RETURN T)))
    (SETQ RELEVANT2
      (CDR RELEVANT2))
    (GO LOOP2) EXIT
    (RETURN NIL)))

(DE TRYRULE+
  (RULE)
  (COND((TESTIF+ RULE)
      (REMEMBERRULE RULE)
      (USETHEN RULE))))

(DE TESTIF+
  (RULE)
  (PROG(IFS)
    (SETQ IFS
      (CAR RULE)) LOOP
    (COND((NULL IFS)
        (RETURN T))
      ((VERIFY(CAR IFS)))
      (T(RETURN NIL)))
    (SETQ IFS
      (CDR IFS))
    (GO LOOP)))

(DE INIF
  (FACT R)
  (MAPCAN '(LAMBDA(X)
      (COND((MEMBER FACT
          (CAR X))
          (LIST X)))) R))

(DE INTHEN
  (FACT R)
  (MAPCAN '(LAMBDA(X)
      (COND((MEMBER FACT
          (CADR X))
          (LIST X)))) R))

(DE DIAGNOSE NIL
  (SETQ RULESUSED NIL)
  (PROG(ASKED POS)
    (SETQ POS HYPS) LOOP
    (COND((NULL POS)
        (MSG T
          "KEINE HYPOTHESE KANN BEWIESEN WERDEN" T)
        (RETURN NIL))
      ((VERIFY(CAR POS))
        (MSG T "DIE HYPOTHESE :" T
          "--> ")
        (PRINC(CAR POS))
        (MSG T "IST WAHR" T)
        (RETURN(CAR POS))))
    (MSG T "DIE HYPOTHESE :" T "--> ")
    (PRINC(CAR POS))
    (MSG T
      "KANN NICHT BEWIESEN WERDEN" T)
    (SETQ POS
      (CDR POS))
    (GO LOOP)))

(SETQ DATA
   '(RULES FACTS HYPS))

(DE ASK
  (S)
```

```
  (MSG T "IST DIES WAHR (J/N/W) :" T)
  (PRINC S)
  (SETQ CH
    (WAITCHAR))
  (MSG T CH T)
  (COND((EQ CH "J")
        T)
    ((EQ CH "N")
       F)
    (T(TELLWHY)
      (ASK S))))

(DE TELLWHY NIL
  (MSG T "ICH VERSUCHE ZU BEWEISEN :" T)
  (PRINC(CAR POS))
  (MSG T T "ICH TESTE :" T)
  (PRINT-RULE RULE))

(DE PRINT-FACTS NIL
  (COND(FACTS(PRLIST FACTS))
    (T(MSG
        "ES SIND KEINE FAKTEN VORHANDEN!" T))))

(DE PRINT-HYPS NIL
  (COND(HYPS(PRLIST HYPS))
    (T(MSG
        "NOCH WURDE KEINE HYPOTHESE AUFGESTELLT!" T))))

(DE PRINT-RULES NIL
  (COND(RULES(MAPC 'PRINT-RULE RULES))
    (T(MSG "BITTE REGELN EINGEBEN!" T))))

(DE PRINT-RULE
  (R)
  (MSG T "+++ REGEL "
    (CAR(CDDR R)) " +++" T)
  (PRINT-IF(CAR R))
  (PRINT-THEN(CADR R)))

(DE PRLIST
  (L)
  (MAPC '(LAMBDA(X)
      (PRINC X)
      (TERPRI)) L))

(DE PRINT-RULE-N
  (N)
  (COND((SETQ X
      (GET-RULE-N N RULES))
      (PRINT-RULE X))
    (T(MSG
        "ES GIBT NOCH KEINE REGEL " N T))))

(DE GET-RULE-N
  (N R)
  (PROG NIL LOOP
    (COND((NULL R)
        (RETURN NIL))
      ((EQ N
        (CAR(CDDR(CAR R))))
        (RETURN(CAR R))))
    (SETQ R
      (CDR R))
    (GO LOOP)))

(DE PRINT-IF
  (IFS)
  (PRINC(CONS 'WENN
      (CONS ':
        (CAR IFS))))
  (TERPRI)
  (MAPC '(LAMBDA(X)
      (PRINC(APPEND '(UND WENN :) X))
      (TERPRI))
    (CDR IFS)))

(DE PRINT-THEN
  (THENS)
  (PRINC(CONS 'DANN
      (CONS ':
        (CAR THENS))))
  (TERPRI)
  (MAPC '(LAMBDA(X)
      (PRINC(APPEND '(UND DANN :) X))
      (TERPRI))
    (CDR THENS)))
```

```
(DF FORGET-FACT L
  (SETQ FACTS
    (REMOVE L FACTS)))

(DF FORGET-HYP L
  (SETQ HYPS
    (REMOVE L HYPS)))

(DE FORGET-FACTS NIL
  (SETQ FACTS NIL))

(DE FORGET-HYPS NIL
  (SETQ HYPS NIL))

(DE FORGET-RULES NIL
  (SETQ RULES NIL)
  (SETQ RULE NIL))

(DE FORGET-RULE
  (N)
  (SETQ RULES
    (REMOVE(GET-RULE-N N RULES) RULES)))

(DE WHAT? NIL
  (MSG "WAS SOLL ICH TUN ?" T))

(DF HOW FACT
  (COND((SETQ X
      (INTHEN FACT RULESUSED))
      (MSG "MIT DEN FAKTEN :" T)
      (MAPC '(LAMBDA(Y)
          (PRLIST(CAR Y))) X))
    ((MEMBER FACT FACTS)
      (MSG "DAS FAKTUM WAR GEGEBEN" T))
    (T(MSG
        "DAS HABE ICH NICHT DEDUZIERT" T))))

(DF WHY FACT
  (COND((MEMBER FACT HYPS)
      (MSG
        "ES WAR EINE DER HYPOTHESEN" T))
    ((SETQ X
      (INIF FACT RULESUSED))
      (MSG "ES FOLGT DARAUS :" T)
      (MAPC '(LAMBDA(Y)
          (PRLIST(CADR Y))) X))
    (T(MSG
        "DAS HABE ICH NICHT BENUTZT" T))))

(DF WHICH NIL
  (COND((NULL RULESUSED)
      (MSG "KEINE" T))
    (T(MSG "DIE REGELN ")
      (PRINC(MAPCAR 'LAST RULESUSED))
      (TERPRI))))

(DE REMEMBERRULE
  (RULE)
  (COND((NOT(MEMBER RULE RULESUSED))
      (SETQ RULESUSED
        (CONS RULE RULESUSED)))))

(DF RULES-WITH-IF L
  (MAPC 'PRINT-RULE
    (INIF L RULES)))

(DF RULES-WITH-THEN L
  (MAPC 'PRINT-RULE
    (INTHEN L RULES)))

(DE USED-RULE
  (N)
  (COND((SETQ X
      (GET-RULE-N N RULESUSED))
      (MSG "JA:" T)
      (PRINT-RULE X))
    (T(MSG "NEIN" T))))

(DF ADD-FACT L
  (COND((NULL FACTS)
      (SETQ FACTS
        (LIST L)))
    ((MEMBER L FACTS))
    (T(NCONC1 FACTS L))))

(DF ADD-HYP L
  (COND((NULL HYPS)
```

```
      (SETQ HYPS
        (LIST L)))
    ((MEMBER L HYPS))
    (T(NCONC1 HYPS L))))

(DF IF L
  (MSG "REGEL "
    (ADD1(LENGTH RULES)) T)
  (SETQ RULE
    (LIST(LIST L) NIL
      (ADD1(LENGTH RULES))))
  (COND(RULES(NCONC1 RULES RULE))
    (T(SETQ RULES
        (LIST RULE)))))

(DF ANDIF L
  (NCONC1(CAR RULE) L))

(DF THEN L
  (SETQ THEN-FACT L)
  (RPLACA(CDR RULE)
    (LIST L)))

(DE IS-HYP NIL
  (COND(THEN-FACT(APPLY 'ADD-HYP
        THEN-FACT))))

(DF ANDTHEN L
  (SETQ THEN-FACT L)
  (NCONC1(CADR RULE) L))

(DE TO-DO
  (S)
  (PROG(SENT)
    (SETQ SENT DIAREG) LOOP
    (COND((NULL SENT)
        (RETURN(LIST 'WHAT?)))
      ((MATCH(CAAR SENT) S)
        (RETURN(DO-FUNC(CDAR SENT) S))))
    (SETQ SENT
      (CDR SENT))
    (GO LOOP)))

(DE DO-FUNC
  (FUNC S)
  (CONS(CAR FUNC)
    (COND((CDR(MEMBER ': S)))
      ((IN-EXPR S)))))

(DF DO-LISP
  (L)
  (PRINT(EVAL L)))

(DE IN-EXPR
  (L)
  (COND((ATOM L)
       NIL)
    ((OR(NUMBERP(CAR L))
      (CONSP(CAR L)))
      (LIST(CAR L)))
    (T(IN-EXPR(CDR L)))))

(DE MATCH
  (P S)
  (COND((NULL P)
      (OR(NULL S)
        (EQ(CAR S)
           ':)))
    ((EQ(CAR P)
       '*)
      (COND((OR(NULL S)
          (EQ(CAR S)
             ':))
          (NULL(CDR P)))
        ((MATCH(CDR P) S))
        ((MATCH P
          (CDR S)))))
    ((NULL S)
       NIL)
    ((EQ(CAR P)
      (CAR S))
      (MATCH(CDR P)
        (CDR S)))
    ((AND(CONSP(CAR P))
      (MEMBER(CAR S)
        (CAR P)))
      (MATCH(CDR P)
```

```
        (CDR S)))))

(SETQ DIAREG
   '(((WENN).IF)
      ((UND DANN).ANDTHEN)
      ((DANN).THEN)
      ((UND *).ANDIF)
      ((ALS *).IS-HYP)
      (((DRUCKE ZEIGE D)
         *
        (REGELN R)).PRINT-RULES)
      (((DRUCKE ZEIGE D)
         *
        (FAKTEN F)).PRINT-FACTS)
      (((DRUCKE ZEIGE D)
         *
        (HYPOTHESEN H)).PRINT-HYPS)
      (((DRUCKE ZEIGE D)
         * REGEL *).PRINT-RULE-N)
      (((VERGISS LOESCHE V)
         *
        (FAKTEN F)).FORGET-FACTS)
      (((VERGISS LOESCHE V)
         *
        (HYPOTHESEN H)).FORGET-HYPS)
      (((VERGISS LOESCHE V)
         *
        (REGELN R)).FORGET-RULES)
      (((VERGISS LOESCHE V)
         * FAKTUM).FORGET-FACT)
      (((VERGISS LOESCHE V)
         *
        (HYPOTHESE HYP)).FORGET-HYP)
      (((VERGISS LOESCHE V)
         * REGEL *).FORGET-RULE)
      ((WIE *).HOW)
      ((WELCHE *).WHICH)
      ((WARUM *).WHY)
      ((*(FAKTUM LERNE MERKE L M) *).
        ADD-FACT)
      ((*(HYPOTHESE HYP) *).ADD-HYP)
      ((* DIAGNOSE *).DIAGNOSE)
      ((*(DEDUZIERE DEDUZIEREN
          DEDUKTION) *).DEDUCE)
      ((*(KONKLUSION K) *).
        RULES-WITH-THEN)
      ((*(PRAEMISSE P) *).
        RULES-WITH-IF)
      ((*(ANGEWENDET BENUTZT A) *).
        USED-RULE)
      ((* LISP *).DO-LISP)))

(SETQ EXPFNS
   '(DO REMEMBER RECALL TESTIF
    USETHEN TRYRULE STEPFORWARD
    DEDUCE RULES FACTS HYPS VERIFY
    TRYRULE+ TESTIF+ INIF INTHEN
    DIAGNOSE DATA ASK TELLWHY
    PRINT-FACTS PRINT-HYPS
    PRINT-RULES PRINT-RULE PRLIST
    PRINT-RULE-N GET-RULE-N PRINT-IF
    PRINT-THEN FORGET-FACT FORGET-HYP
    FORGET-FACTS FORGET-HYPS
    FORGET-RULES FORGET-RULE
    CHANGE-RULE WHAT? HOW WHY WHICH
    REMEMBERRULE RULES-WITH-IF
    RULES-WITH-THEN USED-RULE
    ADD-FACT ADD-HYP IF ANDIF THEN
    IS-HYP ANDTHEN TO-DO DO-FUNC
    DO-LISP IN-EXPR MATCH DIAREG
    EXPFNS))
```

8.12 ZOOTIERE.DAT

```
(SETQ RULES
   '((((DAS TIER HAT HAARE))
    ((DAS TIER IST EIN SAEUGETIER)) 1)
    (((DAS TIER GIBT MILCH))
      ((DAS TIER IST EIN SAEUGETIER)) 2)
    (((DAS TIER HAT FEDERN))
      ((DAS TIER IST EIN VOGEL)) 3)
    (((DAS TIER KANN FLIEGEN)
      (DAS TIER LEGT EIER))
      ((DAS TIER IST EIN VOGEL)) 4)
    (((DAS TIER FRISST FLEISCH))
      ((DAS TIER IST EIN FLEISCHFRESSER)) 5)
    (((DAS TIER HAT SCHARFE ZAEHNE)
      (DAS TIER HAT KLAUEN)
      (DAS TIER HAT NACH VORN GERICHTETE AUGEN))
      ((DAS TIER IST EIN FLEISCHFRESSER)) 6)
    (((DAS TIER IST EIN SAEUGETIER)
      (DAS TIER HAT HUFE))
      ((DAS TIER IST EIN ZEHENFUESSER)) 7)
    (((DAS TIER IST EIN SAEUGETIER)
      (DAS TIER IST EIN WIDERKAEUER))
      ((DAS TIER IST EIN ZEHENFUESSER)) 8)
    (((DAS TIER IST EIN SAEUGETIER)
      (DAS TIER IST EIN FLEISCHFRESSER)
      (DAS TIER HAT EIN TARNFARBENFELL)
      (DAS TIER HAT DUNKLE PUNKTE))
      ((DAS TIER IST CHEETAH)) 9)
    (((DAS TIER IST EIN SAEUGETIER)
      (DAS TIER IST EIN FLEISCHFRESSER)
      (DAS TIER HAT EIN TARNFARBENFELL)
      (DAS TIER IST SCHWARZ-GESTREIFT))
      ((DAS TIER IST EIN TIGER)) 10)
    (((DAS TIER IST EIN ZEHENFUESSER)
      (DAS TIER HAT EINEN LANGEN HALS)
      (DAS TIER HAT LANGE BEINE)
      (DAS TIER HAT DUNKLE PUNKTE))
      ((DAS TIER IST EINE GIRAFFE)) 11)
    (((DAS TIER IST EIN ZEHENFUESSER)
      (DAS TIER HAT SCHWARZE STREIFEN))
      ((DAS TIER IST EIN ZEBRA)) 12)
    (((DAS TIER IST EIN VOGEL)
      (DAS TIER KANN NICHT FLIEGEN)
      (DAS TIER HAT EINEN LANGEN HALS)
      (DAS TIER HAT LANGE BEINE)
      (DAS TIER IST SCHWARZ UND WEISS))
      ((DAS TIER IST EIN STRAUSS)) 13)
    (((DAS TIER IST EIN VOGEL)
      (DAS TIER KANN NICHT FLIEGEN)
      (DAS TIER KANN SCHWIMMEN)
      (DAS TIER IST SCHWARZ UND WEISS))
      ((DAS TIER IST EIN PINGUIN)) 14)
    (((DAS TIER IST EIN VOGEL)
      (DAS TIER KANN SEHR GUT FLIEGEN))
      ((DAS TIER IST EIN ALBATROSS)) 15)))

(SETQ FACTS
   'NIL)

(SETQ HYPS
   '((DAS TIER IST CHEETAH)
    (DAS TIER IST EIN TIGER)
    (DAS TIER IST EINE GIRAFFE)
    (DAS TIER IST EIN ZEBRA)
    (DAS TIER IST EIN STRAUSS)
    (DAS TIER IST EIN PINGUIN)
    (DAS TIER IST EIN ALBATROSS)))
```

8.13 ELIZA.LSP

```
(DE ELIZA NIL
  (MSG(CHAR 147) T
    "■HALLO, ICH BIN ELIZA !■" T T
    "BITTE ERZAEHLE MIR VON" T T
    "DEINEN PROBLEMEN :" T T)
  (DIALOG ERSATZ-ANTWORTEN)
  (MSG T "WIR MUESSEN UNSERE SITZUNG"
    T "LEIDER BEENDEN." T T
    "TSCHUESS !" T T))

(DE DIALOG
  (ERSATZ-ANTWORTEN)
  (PROG(RESULTAT SATZ) LOOP1
    (SETQ SATZ
      (READL))
    (SETQ RESULTAT
      (FINDE-ANTWORT SATZ DIALOG-REGELN))
    (COND(RESULTAT(PRINC RESULTAT))
      (ERSATZ-ANTWORTEN(PRINC(CAR ERSATZ-ANTWORTEN))
        (SETQ ERSATZ-ANTWORTEN
          (CDR ERSATZ-ANTWORTEN)))
      (T(RETURN NIL)))
    (TERPRI)
    (GO LOOP1)))

(DE FINDE-ANTWORT
  (S R)
  (PROG(RESULTAT) LOOP
    (COND((NULL R)
        (RETURN NIL))
      ((MATCH(CAAR R) S)
        (SETQ RESULTAT
          (CAR(CDAR R)))
        (NCONC1 R
          (CAR R))
        (RPLACA R
          (CADR R))
        (RPLACD R
          (CDDR R))
        (RETURN RESULTAT)))
    (SETQ R
      (CDR R))
    (GO LOOP)))

(DE MATCH
  (P S)
  (COND((NULL P)
      (NULL S))
    ((EQ(CAR P)
        '*)
      (COND((NULL S)
          (NULL(CDR P)))
        ((MATCH(CDR P) S))
        ((MATCH P
          (CDR S)))))
    ((NULL S)
       NIL)
    ((EQ(CAR P)
      (CAR S))
      (MATCH(CDR P)
        (CDR S)))
    ((AND(CONSP(CAR P))
      (MEMBER(CAR S)
        (CAR P)))
      (MATCH(CDR P)
        (CDR S)))))

(DE LERNE NIL
  (PROG(EING EING2 NR)
    (SETQ NR
      (ADD1(LENGTH DIALOG-REGELN)))
    LOOP
    (MSG T NR 2)
    (SETQ EING
      (READL))
    (COND((EQUAL EING
        '(-))
      (RETURN NR))
      ((EQUAL EING
         '(PP))
        (PP DIALOG-REGELN)
        (GO LOOP)))
```

```
    (MSG NR 2)
    (SETQ EING2
      (READL))
    (NCONC1 DIALOG-REGELN
      (PRINT(LIST EING EING2)))
    (SETQ NR
      (ADD1 NR))
    (GO LOOP)))

(SETQ ERSATZ-ANTWORTEN
   '((WAS MEINST DU DAMIT?)
    (KOMM BITTE ZUR SACHE!)
    (TATSAECHLICH?)
    (SO SO)
    (UNSERE ZEIT IST BALD UM)
    (NA UND?)))

(SETQ DIALOG-REGELN
   '(((JA)
    (ICH MOECHTE ES GERNE GENAUER WISSEN))
    ((JA *)
      (KANNST DU MIR DAS NAEHER ERLAEUTERN?))
    ((* JA *)
      (DAS IST ABER INTERESSANT!))
    ((NEIN)
      (WIESO NICHT?))
    ((NEIN * NICHT *)
      (WIRKLICH NICHT? WIESO?))
    ((ABER *)
      (WAR DIR MEINE FRAGE UNANGENEHM?))
    ((NEIN *)
      (WAS WUERDEST DU DENN DAZU SAGEN?))
    ((* BIST WIE *)
      (IN WELCHER HINSICHT?))
    (((BIN BIST IST WAERE WAERST) *)
      (WIESO IST DAS FUER DICH WICHTIG?))
    ((DU BIST *)
      (WIE KOMMST DU DARAUF?))
    ((WEIL *)
      (WAS MEINST DU DAMIT?))
    ((ICH(LIEBE HASSE) *)
      (LASSEN WIR DIE GEFUEHLE BEISEITE!))
    ((ICH(MOECHTE WILL) *)
      (UND WARUM TUST DU ES NICHT?))
    ((ICH WEISS NICHT *)
      (MUSS MAN DAS DENN WISSEN?))
    ((ICH WEISS *)
      (BIST DU DIR SICHER?))
    ((ICH HEISSE *)
      (ERZAEHLE MIR MEHR VON DIR))
    ((WIE HEISST DU)
      (ELIZA - DAS WEISST DU DOCH!))
    ((* BIN *
      (TRAURIG UNGLUECKLICH UNZUFRIEDEN))
      (WAS MACHT DICH DENN TRAURIG?))
    ((* BIN *
      (GLUECKLICH ZUFRIEDEN))
      (ES GEHT DIR ANSCHEINEND GUT?))
    ((DU BIST *)
      (MACHT ES DIR ETWAS AUS?))
    ((* KEINE PROBLEME)
      (DAS IST JA SCHOEN!))
    ((DU HAST GESAGT *)
      (HABE ICH DAS WIRKLICH GESAGT?))
    ((ICH MUSS *)
      (DAS GLAUBE ICH AUCH))
    ((DU MUSST *)
      (JA - DAS TUE ICH))
    ((ICH HABE *)
      (WAS BEDEUTET DAS FUER DICH?))
    ((DU HAST *)
      (MEINST DU WIRKLICH?))
    ((HALLO *)
      (WIE GEHTS?))
    ((GUTEN(MORGEN ABEND TAG) *)
      (HALLO - WIE GEHTS?))
    ((WARUM *)
      (ICH WEISS NICHT - WAS MEINST DU?))
    ((WIE *)
      (VERSUCHE ES HERAUSZUFINDEN!))
    ((KANNST DU *)
      (ICH HOFFE ES))
    ((ICH KANN NICHT *)
      (SAGE MIR WARUM DU ES NICHT KANNST!))
    ((ICH KANN *)
      (KANNST DU DAS WIRKLICH?))
    (((WELCHE WELCHER WELCHES)
```

```
       *)
      (WAS MEINST DU?))
    ((WAS *)
      (WAS WUERDEST DU VORSCHLAGEN?))
    ((WER *)
      (AN WEN DENKST DU JETZT?))
    ((WO *)
      (DU WEISST ES DOCH SELBST))
    ((WOHER(WEISST WUSSTEST) DU *)
      (OH - ICH HATTE NUR SO EINE AHNUNG!))
    ((WIESO *)
      (DAS KANN ICH DIR AUCH NICHT SAGEN))
    ((ICH BIN *)
      (DAS GLAUBE ICH DIR NICHT))
    ((DU BIST *)
      (SPRECHEN WIR LIEBER UEBER DICH!))
    ((* ELIZA *)
      (WAS HAELST DU VON MIR?))
    ((*(FRAUEN FREUNDIN MAEDCHEN) *)
      (HAST DU PROBLEME MIT FRAUEN?))
    ((* SEX *)
      (HAST DU SEXUELLE PROBLEME?))
    ((* GOTT *)
      (WAS HAT RELIGION DAMIT ZU TUN?))
    ((* TSCHUESS *)
      (DU WILLST SCHON AUFHOEREN?))
    ((*(RECHNER COMPUTER C64 FLOPPY PROGRAMM) *)
      (ICH VERSTEHE DAVON LEIDER NICHTS))
    ((*(JEDE JEDER JEDES) *)
      (DENKST DU AN JEMAND BESTIMMTEN?))
    ((*(BLOND ROTHAARIG BRUENETT) *)
      (WAS MAGST DU AN IHR?))
    ((*(GELD TASCHENGELD DM KOHLE) *)
      (HAST DU EIN FINANZIELLES PROBLEM?))
    ((*(COMPUTER RECHNER) *)
      (WAS BEDEUTEN COMPUTER FUER DICH?))
    ((*(IDIOT SCHEISSE BLOEDE GEIL) *)
      (BITTE NICHT IN DIESEM TON!))
    ((* VIELLEICHT *)
      (DU BIST DIR NICHT SICHER?))
    ((* MANCHMAL *)
      (UND SONST?))
    ((* WEIL *)
      (IST DAS DER EINZIGE GRUND?))
    ((* NICHT *)
      (WARUM NICHT?))
    ((* NIE *)
      (MAN SOLL NIE NIE SAGEN!))
    ((*(WIRKLICH SEHR TOTAL) *)
      (WARUM BETONST DU DAS SO?))
    ((* MEIN *)
      (UND WAS WEITER?))
    ((* ICH *)
      (REDEST DU GERN UEBER DICH?))
    ((* DU *)
      (REDEN WIR LIEBER UEBER DICH!))
    ((* DICH *)
      (DAS HOERE ICH GERNE!))))

(SETQ ELIFNS
   '(ELIZA DIALOG FINDE-ANTWORT MATCH
    LERNE ERSATZ-ANTWORTEN
    DIALOG-REGELN ELIFNS))
```

8.14 NDAMEN.LSP

INPUT 64 4/87 S. 30

Selbstlernendes Lisp-Programm

Diese Aufgabenstellung war ungleich schwerer, da der Programmierer hier auch den Ein- und Ausgabeteil selber erstellen musste. (Das Rahmenprogramm war bekanntlich nur in BASIC geschrieben und von daher für ein Lisp-Programm nicht zu gebrauchen.) Die einzige Lösung erhielten wir von Frank Fetthauer, Essen.

Das Besondere an diesem Programm ist das eingebaute Gedächtnis und die Fähigkeit, sehr einfach von n Damen auf $n-1$ Damen schließen zu können.

Das Programm wird aufgerufen mit:

`(dame n)`

```
(DF DAME
  (M)
  (MEMCHECK M)
  (POS NIL NIL
    (REVERSE((LABEL DLIST
        (LAMBDA(Q W)
          (COND((ZEROP Q)
                W)
            (T(DLIST(SUB1 Q)
                (CONS Q W))))))
         M NIL))))

(DE POS
  (DIF SUM DEF)
  (COND((NULL DEF)
      (GRAFIK DIF SUM))
    (T(FOR I 1
        (LENGTH DEF)
        (COND((ELEMENT(DIFFERENCE(CAR(
                NTH DEF I))
              (LENGTH DEF)) DIF)
            T)
          ((ELEMENT(PLUS(CAR(NTH DEF
                  I))
              (LENGTH DEF)) SUM)
            T)
          (T(POS(CONS(DIFFERENCE(CAR(
                    NTH DEF I))
                (LENGTH DEF)) DIF)
              (CONS(PLUS(CAR(NTH DEF
                    I))
                (LENGTH DEF)) SUM)
              (REMOVE(CAR(NTH DEF I))
                DEF))))))))

(DE ELEMENT
  (EL LI)
  (NOT(EQUAL LI
      (REMOVE EL LI))))

(DE GRAFIK
  (D S)
  (FOR I 1
    (LENGTH D)
    (SETQ W
      (QUOTIENT(PLUS(CAR(NTH D I))
          (CAR(NTH S I))) 2))
    (MSG "■")
    (SPACES(SUB1 W))
    (MSG "D")
    (SPACES(DIFFERENCE(LENGTH D) W))
    (MSG "■" T))
  (MSG T)
  (MEMSET D S)
  (CLR)
  (ERROR '(EINE LOESUNG)))
```

```
(DE CLR NIL
  (POKE 631 145)
  (POKE 632 32)
  (POKE 633 32)
  (POKE 634 32)
  (POKE 635 32)
  (POKE 636 32)
  (POKE 637 32)
  (POKE 638 32)
  (POKE 639 32)
  (POKE 640 141)
  (POKE 198 10))

(DE MEMSET
  (DI SU)
  (SETQ Z
    (PACK(CONS 'M
        (LIST(LENGTH DI)))))
  (COND((NOT(NULL(GETPROP 'MEMD Z)))
       T)
    (T(PUTPROP 'MEMD Z DI)
      (PUTPROP 'MEMS Z SU)
      (COND((EQ(TIMES 2
            (LENGTH DI))
          (PLUS(CAR(REVERSE DI))
            (CAR(REVERSE SU))))
          (MEMSET(CDR(REVERSE DI))
            (CDR(REVERSE SU)))))))))

(DE MEMCHECK
  (R)
  (SETQ Z
    (PACK(CONS 'M
        (LIST R))))
  (COND((NOT(NULL(GETPROP 'MEMD Z)))
      (GRAFIK(GETPROP 'MEMD Z)
        (GETPROP 'MEMS Z)))))

(DM FOR L
  (PROG(VAR VON NACH COUNT-FN TEST-FN)
    (SETQ VAR
      (CADR L))
    (SETQ VON
      (EVAL(CAR(CDDR L))))
    (SETQ NACH
      (EVAL(CADR(CDDR L))))
    (COND((GREATERP VON NACH)
        (SETQ TEST-FN
           'LESSP)
        (SETQ COUNT-FN
           'SUB1))
      (T(SETQ TEST-FN
           'GREATERP)
        (SETQ COUNT-FN
           'ADD1)))
    (RETURN(LIST 'PROG
        (LIST VAR)
        (LIST 'SETQ VAR VON)
         'LOOP
        (LIST 'COND
          (LIST(LIST TEST-FN VAR NACH)
             '(RETURN NIL))
          (CONS 'T
            (CDDR(CDDR L))))
        (LIST 'SETQ VAR
          (LIST COUNT-FN VAR))
         '(GO LOOP)))))
```

Die Funktion `CLR` schreibt 10 Tastendrücke in den Tastaturpuffer, der im C64–Speicher bei Adresse 631 beginnt und bei Adresse 640 endet: Zunächst den Steuercode für CRSR-UP, dann 8 mal SPACE und schließlich den Steuercode für SHIFT+RETURN.
`(POKE 198 10)` meldet dem Kernal 10 gedrückte Tasten im Tastaturpuffer. Dies löscht wieder das ungewollte `NIL` nach `MEMSET`.

```
(DE CLR NIL
  (POKE 631 145)
  (POKE 632 32)  (POKE 633 32)  (POKE 634 32)  (POKE 635 32)
  (POKE 636 32)  (POKE 637 32)  (POKE 638 32)  (POKE 639 32)
```

```
  (POKE 640 141)
  (POKE 198 10))
```

Für den C128 sieht die Funktion `CLR` folgendermaßen aus:

```
(DE CLR NIL
  (BANK 15)
  (POKE 842 145)
  (POKE 843 32)
  (POKE 844 32)
  (POKE 845 32)
  (POKE 846 32)
  (POKE 847 32)
  (POKE 848 32)
  (POKE 849 32)
  (POKE 850 32)
  (POKE 851 141)
  (POKE 208 10))
```

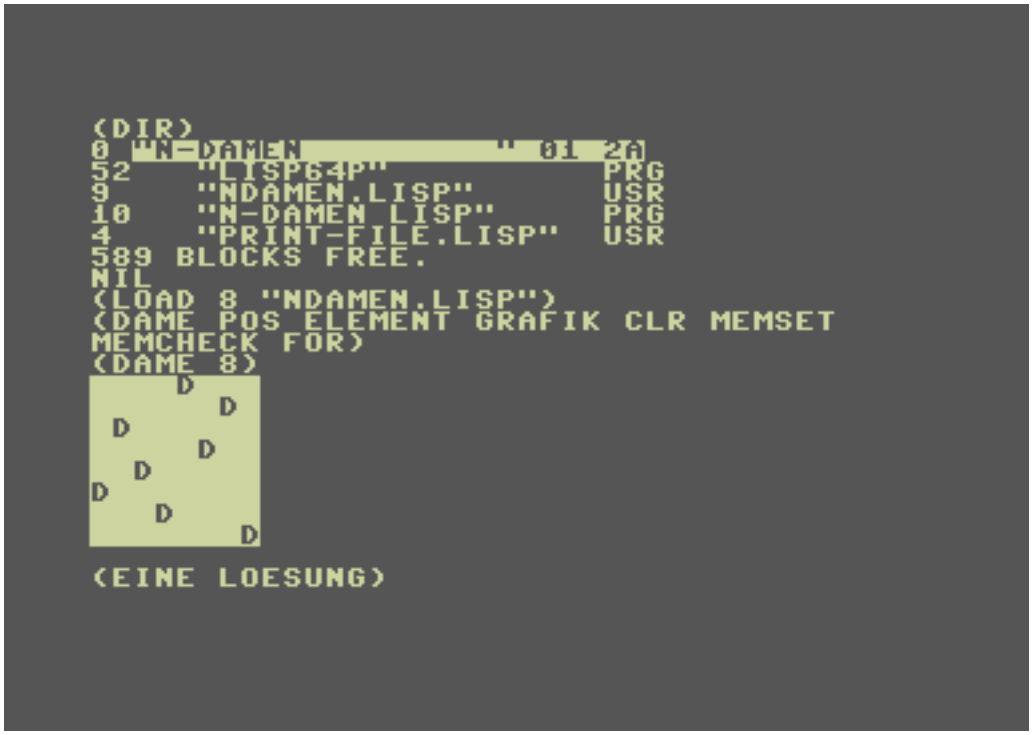


Abbildung 8.1: Beispiel für acht Damen

8.15 Der Lisassembler

INPUT 3/87, S. 2

Ein kurzer Tipp zu LISP 64 (der LISP-Interpreter aus INPUT 4/86): Der Strukturediteor aus der Ausgabe 5/86 ist zwar abgesehen von ein paar Abstürzen recht nett, wenn man jedoch mehr visuell orientiert ist, dann bietet er halt doch nicht den gewohnten Komfort. Hierzu wäre ein Texteditor das geeignete Element, doch woher nehmen... Ausgabe 6/86!

Der INPUT-ASS hätte es sich zwar nicht träumen lassen, dass er einmal kein Assemblerprogramm bearbeitet, er hat aber auch nichts dagegen. Doch wie sag' ich's meinem LISP?
Die beiden folgenden kurzen Funktionen erledigen das Nötige:

8.15.1 Lesen von Diskette aus dem sequentiellen Editor-Format

```
(DE GETDISK NIL
  (OPEN 1 8 0 "FROMEDIT,S,R")
  (INPUT 1)
  (READL))
```

Diese Funktion liest ein Editor-File ein und definiert sofort die Funktionen für LISP, wenn ein, zwei Dinge bei der Erstellung beachtet werden:

Am Beginn des Files muss NIL oder `()` (leere Klammer) stehen. Am Ende sollte zur Umschaltung auf Tastatureingaben `(NORMAL)` stehen. Nach dem Aufruf der Funktion mit `(GETDISK)` muss man sich leider noch der Mühe eines `(CLOSE 1)` unterziehen.

8.15.2 Übergabe von LISP-Programmen an den Editor

```
(DE PUTDISK (N)
  (DISK "S:FROMLISP")
  (OPEN 1 8 2 "FROMLISP,S,W")
  (OUTPUT 1)
  (PP NIL) (* hier wird Punkt 1 beachtet, das NIL)
  (MAPC 'PDEF N)
  (PP '(NORMAL)) (* hier wird das Ende geschrieben)
  (NORMAL)
  (CLOSE 1))
```

Das hiermit erzeugte File "FROMLISP" kann jetzt vom Editor gelesen werden.
(K.P. Meyer, Heilbronn)

Literaturverzeichnis

[Fri96] Friedrichs, Sven: *Komfortables Bankswitching mit dem C128*. `http://www.commodore128.de/128swit.htm`. Version: Februar 1996

[Kay04] *A Conversation with Alan Kay*. Internet content. `http://queue.acm.org/detail.cfm?id=1039523`. Version: 12 2004. – Inteview in ACMQueue Volume 2, issue 9

[Mü85] Müller, Dieter: *LISP: Eine elementare Einführung in die Programmierung nichtnumerischer Aufgaben*. Bibliograph. Inst., BI-Wissenschaftsverl., 1985

[McC65] McCarthy, John: *LISP 1.5 programmer's manual*. MIT press, 1965

[Mil87] Miller, Chris: Mr. Ed: A Modular Text Editor For The Commodore 64. In: *The Transactor* 8 (1987), September, Nr. 02, S. 62–67

[Nie12] Nielsen, Michael: *Lisp as the Maxwell's equations of software*. Internet content. `http://www.michaelnielsen.org/ddi/lisp-as-the-maxwells-equations-of-software/`. Version: April 2012

[Sch86] Schineis, O. M.; Demgensky N. Rudolf; Braun B. Rudolf; Braun: *C128 ROM–Listing: Operating System*. Markt & Technik, 1986. – ISBN 3–89090–221–9

[WPC15] *Clojure*. Wikipedia-Artikel. `http://de.wikipedia.org/wiki/Clojure`. Version: 03 2015