Anwendersoftwareklasse 2

LISP – Handbuch

Stand: 08.08.86

© 1986
Selbstverlag GMD Alle Rechte vorbehalten.
Insbesondere ist die Überführung in maschinenlesbare Form, sowie das Speichern in Informationssystemen, auch auszugsweise, nur mit schriftlicher Genehmigung der GMD gestattet.

Herausgeber:

Gesellschaft für Mathematik und Datenverarbeitung mbH

Postfach 1240, Schloß Birlinghoven
D-5205 Sankt Augustin 1
Telefon(02241) 14-1, Telex 8 89 469 gmd d
Telefax(02241) 14 28 89, BTX *43900#
Teletex 2627-224135=GMDVV

Softwareklasse 2 (Anwendersoftware)

Regelmäßige Wartung

Autoren:

John McCarthy
übersetzt und angepaßt von
J. Durcholz, P. Heyderhoff

Texterstellung:

Dieser Text wurde mit der EUMEL-Textverarbeitung erstellt und aufbereitet und mit dem Agfa Laserdrucksystem P400 gedruckt.

Umschlaggestaltung:

Hannelotte Wecken

Hinweis:

Diese Dokumentation wurde mit größtmöglicher Sorgfalt erstellt. Dennoch wird für die Korrektheit und Vollständigkeit der gemachten Angaben keine Gewähr übernommen. Bei vermuteten Fehlern der Software oder der Dokumentation bitten wir um baldige Meldung, damit eine Korrektur möglichst rasch erfolgen kann. Anregungen und Kritik sind jederzeit willkommen.

L I S P H a n d b u c h

Autor: John Mc.Carthy (M.I.T. 1962)
übersetzt und angepaßt von J.Durchholz, P.Heyderhoff
Gesellschaft für Mathematik und Datenverarbeitung Sankt Augustin

Inhaltsverzeichnis

1. Die Sprache LISP 2

1.1 Symbolische Ausdrücke 3
1.2 Elementare Funktionen 5
1.3 Listen Notation 8
1.4 Syntax und Semantik der Sprache 12

2. Das LISP - Interpreter - System 18

2.1 Die universelle LISP - Funktion "evalquote" 18
2.2 Anwendungsregeln und Beispiele 27
2.3 Variablen 30
2.4 Konstanten 31
2.5 Funktionen 32

3. Erweitertes LISP 33

3.1 Gequotete Parameter 34
3.2 Funktionen mit beliebig vielen Parametern 35
3.3 Funktionale Parameter 36
3.4 Prädikate und boolesche Konstanten 37
3.5 Unbenannte Atome 38
3.6 Aufruf von EUMEL aus 40

4. Detailbeschreibungen 42

4.1 Grundfunktionen 42
4.2 Weitere Funktionen sowie Eingabe und Ausgabe 48
4.3 Interpreter 50
4.4 Kommandoprozeduren 55

1. Die Sprache LISP

Die Sprache LISP ist primär für die Symbolmanipulation entworfen. Sie wurde für symbolische Berechnungen in verschiedenen Gebieten der künstlichen Intelligenz eingesetzt, u.a. für Differential– und Integralrechnung, Schaltkreistheorie, Mathematische Logik, Spiele, etc..

LISP ist eine formale mathematische Sprache. Daher ist es möglich, eine genaue und vollständige Beschreibung zu geben. Das ist der Sinn des ersten Abschnitts dieses Handbuchs. Andere Abschnitte werden Möglichkeiten zum vorteilhaften Einsatz von LISP und die Erweiterungen, die die Benutzung erleichtern, beschreiben.

LISP unterscheidet sich von den meisten Programmiersprachen in drei Punkten.

Der erste Punkt liegt in der Natur der Daten. In der Sprache LISP haben alle Daten die Form symbolischer Ausdrücke, die wir verkürzend LISP – Ausdrücke nennen werden. LISP – Ausdrücke haben keine Längenbegrenzung und eine verzweigte Baumstruktur, so daß Unterausdrücke leicht isoliert werden können. In LISP wird der meiste Speicherplatz für das Abspeichern der LISP – Ausdrücke in Form von Listenstrukturen gebraucht.

Der zweite wichtige Teil der Sprache LISP ist die Quellsprache, die festlegt, wie die LISP – Ausdrücke verarbeitet werden sollen.

Drittens kann LISP als LISP – Ausdrücke geschriebene Programme interpretieren und ausführen. Deshalb kann man die Sprache analog zu Assemblersprachen und im Gegensatz zu den meisten anderen höheren Programmiersprachen einsetzen, um Programme zu generieren, die gleich ausgeführt werden sollen.

1.1 Symbolische Ausdrücke

Ein elementarer Ausdruck ist ein Atom.

Definition: Ein Atom ist eine Zeichenkette bestehend aus Großbuchstaben und Ziffern.

```
Beispiele:  A
            APFEL
            TEIL2
            EXTRALANGEZEICHENKETTEAUSBUCHSTABEN
            A4B66XYZ2
```

Diese Symbole werden atomar genannt, weil sie als Ganzes aufgefaßt werden, das durch die LISP – Funktionen nicht weiter geteilt werden kann. A, B, und AB haben keinerlei Beziehung zueinander, außer der, daß sie alle verschiedene Atome sind.

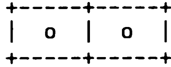
Alle LISP – Ausdrücke werden aus Atomen und den Satzzeichen "(", ")" und "." aufgebaut. Die grundlegende Operation zum Aufbau von LISP – Ausdrücken ist die, zwei LISP – Ausdrücke zusammenzufassen, um einen größeren herzustellen. Aus den zwei Atomen A und B kann man so den LISP – Ausdruck (A.B) bilden.

Definition: Ein LISP – Ausdruck ist entweder ein Atom, oder aus folgenden Elementen in dieser Reihenfolge aufgebaut: Eine öffnende Klammer, ein LISP – Ausdruck, ein Punkt, ein LISP – Ausdruck, eine schließende Klammer. Zwischen den Bestandteilen eines nichtatomaren LISP – Ausdruck können beliebig viele Leerzeichen eingestreut sein.

Diese Definition ist rekursiv.

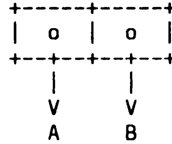
```
Beispiele:  ATOM
            (A . B)
            (A . (B . C))
            ((A1 . A2) . B)
            ((U . V) . (X . Y))
            ((U . V) . (X . (Y . Z)))
```

Um die Struktur solcher Ausdrücke zu verdeutlichen, wird in diesem Handbuch oft eine graphische Darstellung gewählt. In dieser Darstellung sind die Atome weiterhin Zeichenketten, statt der Paare steht jetzt aber ein Kasten

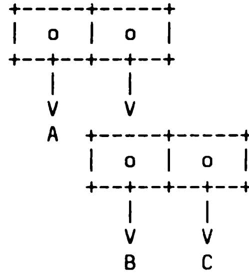


von dem zwei Zeiger ausgehen, die auf die graphische Darstellung des ersten bzw. zweiten Elements des Paars zeigen.

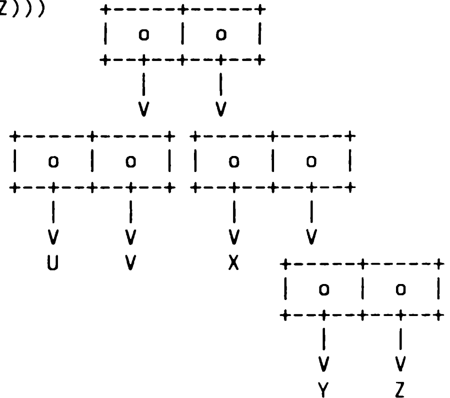
Beispiele: (A . B)



(A . (B . C))



((U . V) . (X . (Y . Z)))



1.2 Elementare Funktionen

Wir werden einige elementare Funktionen auf LISP – Ausdrücken einführen. Um die Funktionen von den LISP – Ausdrücken zu unterscheiden, werden wir Funktionsnamen mit Klein – statt Großbuchstaben schreiben. Außerdem steht der Funktionsname gefolgt von den Argumenten, auf die die Funktion angewendet werden soll, in Klammern eingeschlossen in einer Liste. Dabei sind die Argumente durch Blanks voneinander getrennt.

Die erste Funktion, die wir einführen, heißt "cons". Sie hat zwei Argumente und wird dafür benutzt, LISP – Ausdrücke aus kleineren LISP – Ausdrücken aufzubauen.

```
            Funktionsaufruf:         Ergebnis:

Beispiele:  (cons A B)             = (A . B)
            (cons (A . B) C)       = ((A . B) . C)
            (cons (cons A B) C)    = ((A . B) . C)
```

Die Beispiele zeigen Funktionsaufrufe. Ein Funktionsaufruf ist eine Liste beginnend mit einem Funktionsnamen, gefolgt von Argumenten. Alle Funktionsaufrufe haben ein Ergebnis, das im Fall von LISP – Funktionen immer ein LISP – Ausdruck ist.

In diesen Beispielen kommt nur die Funktion "cons" vor. Das letzte Beispiel ist ein Fall von Funktionsverkettung, das heißt, als Argument steht ein Funktionsaufruf. Um das Ergebnis eines Funktionsaufrufs zu berechnen, das Funktionsaufrufe als Argumente enthält, muß man statt dieser Argumente die Ergebnisse dieser Funktionsaufrufe einsetzen, so daß man den äußeren Funktionsaufruf in einen Aufruf ohne Funktionsaufrufe als Argumente umwandelt.

```
Beispiel: (cons (cons A B) C) = (cons (A . B) C) = ((A . B) . C)
```

Es ist möglich, durch Verkettung der Funktion "cons" jeden LISP – Ausdruck aus seinen atomaren Komponenten aufzubauen.

Die folgenden beiden Funktionen tun das genaue Gegenteil von "cons": sie liefern die Unterausdrücke eines gegebenen LISP – Ausdrucks.

Die Funktion "head" hat ein Argument. Ihr Wert ist der erste Unterausdruck des zusammengesetzen Arguments. Der "head" eines Atoms ist nicht definiert.

```
Beispiele: (head (A . B))           = A
           (head (A . (B1 . B2))) = A
           (head ((A1 . A2) . B)) = (A1 . A2)
           (head A)                 ist nicht definiert
```

Die Funktion "tail" hat ebenfalls ein Argument, und sie liefert das Argument bis auf dessen "head".

```
Beispiele: (tail (A . B))                 = B
           (tail (A . (B1 . B2)))         = (B1 . B2)
           (tail ((A1 . A2) . B))         = B
           (tail A)                         ist nicht definiert
           (head (tail (A . (B1 . B2)))) = B1
           (head (tail (A . B)))            ist nicht definiert
           (head (cons A B))              = A
```

Es ist bei jedem LISP – Ausdruck möglich, durch eine geeignete Verkettung von "head" und "tail" zu jedem Atom im Ausdruck zu gelangen.

Wenn "x" und "y" irgendwelche LISP – Ausdrücke repräsentieren, gelten die folgenden Gleichungen immer:

```
(head (cons x y)) = x
(tail (cons x y)) = y
```

Außerdem gilt die folgende Gleichung für jeden nichtatomaren LISP – Ausdruck "z":

```
(cons (head z) (tail z)) = z
```

Die Symbole "x", "y" und "z", die wir in diesen Gleichungen benutzt haben, nennt man Variablen. In LISP werden Variable benutzt, um LISP – Ausdrücke zu repräsentieren, und zwar repräsentiert eine Variable in einer Gleichung immer denselben LISP – Ausdruck. Variablennamen werden wie Funktionsnamen gebildet, d.h. sie können Kleinbuchstaben und Ziffern enthalten.

Eine Funktion, deren Wert "wahr" oder "falsch" sein kann, wird Prädikat genannt. In LISP werden die Werte "wahr" und "falsch" durch die Atome "T" (true) und "F" (false) vertreten. Ein LISP – Prädikat ist also eine Funktion, deren Wert entweder "T" oder "F" ist.

Das Prädikat "eq" ist ein Gleichheitstest für Atome. Es ist bei nicht atomaren Argumenten nicht definiert.

```
Beispiele: (eq A A)              = T
           (eq A B)              = F
           (eq A (A . B))         ist nicht definiert
           (eq (A . B) B)         ist nicht definiert
           (eq (A . B) (A . B)) ist nicht definiert
```

Das Prädikat "atom" hat das Ergebnis ("liefert") "T", wenn sein Argument atomar ist, und "F", wenn sein Argument zusammengesetzt ist.

```
Beispiele: (atom EXTRALANGEZEICHENKETTE) = T
           (atom (U . V))                = F
           (atom (head (U . V)))         = T
```

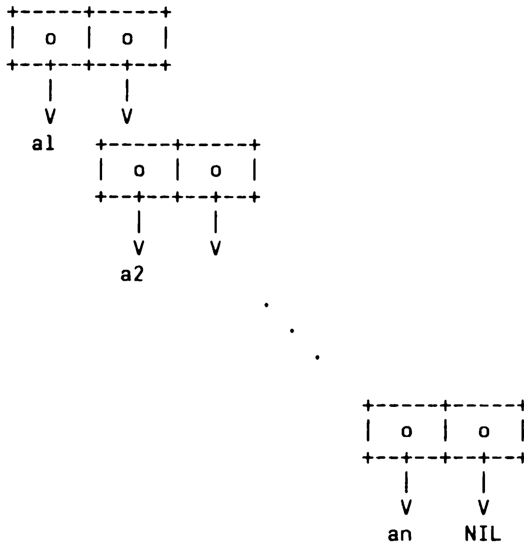
1.3 Listen – Notation

Alle LISP – Ausdrücke, die wir bisher gesehen haben, waren in Punkt – Notation geschrieben. Normalerweise ist es allerdings einfacher, statt der vielen Punkte und Klammern Listen von LISP – Ausdrücken zu schreiben, etwa in der Art (A B C XYZ).

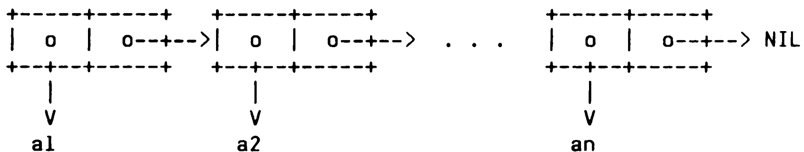
LISP bietet eine solche Alternative zur Punkt – Notation an:

Definition: Die Liste (a1 a2 ... an) ist äquivalent zum LISP – Ausdruck (a1 . (a2 . (... . (an . NIL) ...))).

Graphisch ausgedrückt heißt das:



Oft werden wir für Listen auch die graphische Form



benutzen.

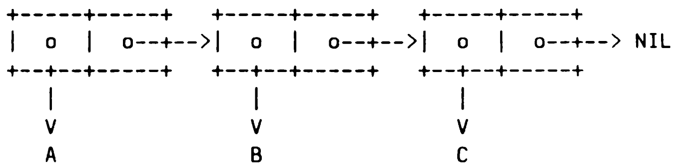
Aus der Graphik wird deutlich, daß NIL als eine Art Abschlußmarkierung für Listen dient.

Eine leere Liste wird durch das Atom NIL dargestellt. Das Prädikat "null" liefert "T", wenn sein Argument eine leere Liste ist, sonst "F".

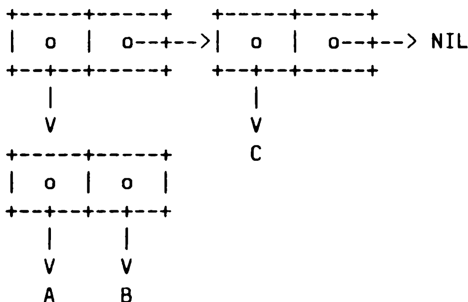
```
Beispiele: (null NIL)    = T
           (null () )    = T
           (null (A B)) = F
```

Die Listenelemente können selbst wieder Listen oder Paare in Punkt – Notation sein, so daß Listen – und Punkt – Notation beliebig kombinierbar sind.

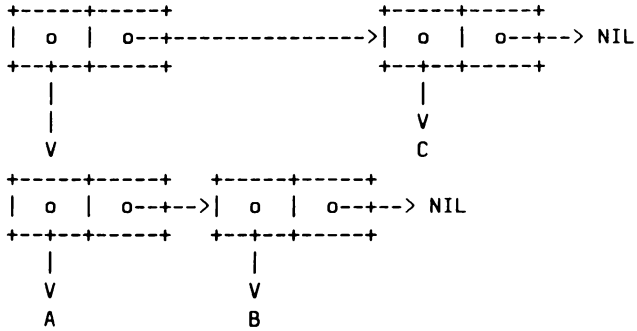
```
Beispiele:  (A B C) = (A . (B . (C . NIL)))
```



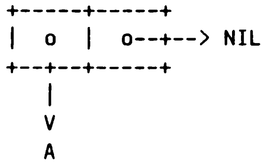

```
((A . B) C) = ((A . B) . (C . NIL))
```



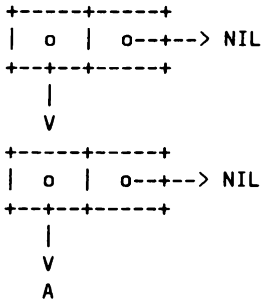
((A B) C) = ((A . (B . NIL)) . (C . NIL))



(A) = (A . NIL)



((A)) = ((A . NIL) . NIL)



Es ist sehr hilfreich, mit den Ergebnissen der elementaren Funktionen vertraut zu sein, wenn diese Listen als Argumente erhalten. Zwar können die Ergebnisse notfalls immer durch Übersetzung in Punkt – Notation bestimmt werden, aber ein direktes Verständnis ist einfacher.

```
Beispiele: (head (A B C)) = A
           (tail (A B C)) = (B C)
```

(Daher auch die Namen "head" und "tail"! Frei übersetzt heißen die beiden Funktionen "anfang" und "rest".)

```
           (cons A (B C)) = (A B C)
```

1.4 Syntax und Semantik der Sprache

Wir haben bisher einen Datentyp (LISP – Ausdrücke) und fünf elementare Funktionen eingeführt. Außerdem haben wir die folgenden Eigenschaften der Sprache beschrieben:

1. Funktions- und Variablennamen werden wie die Namen von Atomen geschrieben, außer, daß dafür Klein- statt Großbuchstaben verwendet werden.
2. Die Argumente einer Funktion folgen dieser in der Liste. Eine solche Liste von Funktion und folgenden Argumenten heißt Funktionsaufruf und hat einen LISP-Ausdruck als Ergebnis.
3. Funktionen können dadurch verkettet werden, daß ein Argument aus einem Funktionsaufruf selbst wieder ein Funktionsaufruf ist, dessen Argumente selbst wieder Funktionsaufrufe sein können, usw.

Diese Regeln erlauben es, Funktionsdefinitionen wie

```
(third x) = (head (tail (tail x)))
```

zu schreiben. "third" holt das dritte Element aus einer Liste.

Die Klasse der Funktionen, die man auf diese Weise bilden kann, ist ziemlich beschränkt und nicht sehr interessant. Eine viel größere Funktionenklasse kann man mit Hilfe des bedingten Ausdrucks schreiben; es handelt sich dabei um eine Möglichkeit, Verzweigungen in Funktionsdefinitionen einzubauen.

Ein bedingter Ausdruck hat die Form

```
(cond (p1 a1) (p2 a2) ... (pn an) )
```

Jedes pi ist ein Ausdruck, dessen Wert "T" oder "F" ist, also ein Prädikat. Die ai sind beliebige LISP – Ausdrücke.

Die Bedeutung eines bedingten Ausdrucks ist folgende: Wenn p1 wahr ist, ist a1 der Wert des ganzen Ausdrucks. Wenn p1 falsch ist, wird getestet, ob p2 wahr ist; wenn das der Fall ist, ist a2 der Wert des Ausdrucks. Die pi werden also von links nach rechts durchgegangen, bis ein wahrer Ausdruck gefunden ist; das zugehörige ai ist dann der Wert des bedingten Ausdrucks. Wenn kein wahres pi gefunden ist, ist der bedingte Ausdruck nicht definiert.
Jedes pi oder ai kann selbst wieder ein LISP – Ausdruck, ein Funktionsaufruf oder ein bedingter Ausdruck sein.

```
Beispiel: (cond ((eq (head x) A) (cons B (tail x))) (T x) )
```

Das Prädikat "T" ist immer wahr. Man liest es am besten als "SONST". Den Wert dieses Ausdruck erhält man, wenn man "head" von x durch B ersetzt, wenn der gerade gleich mit A ist, und sonst erhält man x.

Der Hauptzweck von bedingten Ausdrücken ist die rekursive Definition von Funktionen.

```
Beispiel: (firstatom x) = (cond ((atom x) x)
                                ( T  (firstatom (head x)))
                          )
```

Dies Beispiel definiert die Funktion "firstatom", die das erste Atom jedes LISP-Ausdrucks bestimmt. Diesen Ausdruck kann man so lesen: wenn "x" ein Atom ist, ist "x" selbst die Antwort; sonst muß "firstatom" auf "head" von "x" angewandt werden.

Wenn also "x" ein Atom ist, wird der erste Zweig gewählt, der "x" liefert; sonst wird der zweite Zweig "firstatom (head x)" gewählt, weil "T" immer wahr ist.

Die Definition von "firstatom" ist rekursiv, d.h. "firstatom" ist mit durch sich selbst definiert. Allerdings, wenn man immerzu den "head" von irgendeinem LISP-Ausdruck nimmt, errreicht man irgendwann ein Atom, so daß der Prozeß immer wohldefiniert ist.

Es gibt rekursive Funktionen, die nur für bestimmte Argumente wohldefiniert sind, für bestimmte andere dagegen unendlich rekursiv. Wenn das EUMEL-LISP-System einen Funktionsionsaufruf mit einer solchen Funktion und "kritischen" Argumenten interpretiert, gerät es in unendliche Rekursion, bis entweder der zur Verfügung stehende Platz im LISP-Heap ausgeschöpft ist (im Heap werden die LISP-Ausdrücke gespeichert) oder bis der Laufzeitstack überläuft (der Laufzeitstack ist ein normalerweise unsichtbarer Bestandteil des ELAN-Systems).
Wir werden jetzt die Berechnung von "(firstatom ((A . B) . C))" durchführen. Zunächst ersetzen wir die Variable x in der Funktionsdefinition durch ((A . B) . C) und erhalten

```
(firstatom ((A . B) . C)) =
   (cond ( (atom ((A . B) . C))  ((A . B) . C) )
         ( T  (firstatom (head ((A . B) . C))) )
   )
```

```
((A . B) . C) ist kein Atom, deshalb wird daraus

                    = (cond ( T (firstatom (head ((A . B) . C)))) )
                    = (firstatom (head ((A . B) . C)) )
                    = (firstatom (A . B))
```

An diesem Punkt müssen wir wieder die Definition von "firstatom" benutzen, diesmal aber für "x" überall "(A . B)" einsetzen.

```
  (firstatom (A . B)) = (cond ( (atom (A . B)) (A . B) )
                                (T (firstatom (head (A . B))) )
                          )
                      = (cond (T (firstatom (head (A . B))) ) )
                      = (firstatom (head (A . B)) )
                      = (firstatom A)
                      = (cond ((atom A) A)
                                (T (firstatom (head A)) )
                          )
                      = A
```

Wenn in den bedingten Ausdrücken statt der LISP – Ausdrücke arithmetische Ausdrücke verwendet würden, könnte man damit auch numerische Rechenvorschriften definieren, wie z.B. den Betrag einer Zahl durch

```
  (abs x) = (cond ((x < 0) -x) (T x) )
```

oder die Fakultät durch

```
  (fak n) = (cond ((n = 0) 1)
                  (T (x * (fak (n - 1))))
            )
```

Die Fakultät terminiert bei negativen Argumenten nicht.

Es ist bei den meisten Mathematikern (außer den Logikern) üblich, das Wort "Funktion" nicht präzise zu verwenden und auf Ausdrücke wie "2x + y" anzuwenden. Da wir Ausdrücke benutzen werden, die Funktionen repräsentieren, benötigen wir eine Notation, die Funktionen und Ausdrücke unterscheidet. Dafür ist die Lambda – Notation von Alonzo Church geeignet.
"f" soll ein Ausdruck sein, der für eine Funktion zweier ganzzahliger Variablen steht.

Dann sollte es sinnvoll sein, den Funktionsaufruf

```
(f 3 4)
```

zu schreiben, so daß man dadurch den Wert dieses Funktionsaufrufs berechnen kann; z.B. könnte "(summe 3 4) = 7" gelten.

Wenn man "2x + y" als Funktion ansieht, kann man den Funktionsaufruf

```
((2x + y) 3 4)
```

schreiben. Der Wert dieses Ausdrucks ist aber nicht eindeutig bestimmt, denn es ist überhaupt nicht klar, ob nun "2*3+4" oder 2*4+3" gemeint ist. Eine Zeichenfolge wie "2x + y" werden wir deshalb Ausdruck und nicht Funktion nennen. Ein Ausdruck kann in eine Funktion umgewandelt werden, indem man die Zuordnung von Argumenten und Variablen festlegt. Bei "2x + y" könnte man beispielsweise festlegen, daß "x" immer das erste und "y" immer das zweite Argument sein soll.
Wenn "a" ein Ausdruck in den Variablen x1, ... xn ist, dann ist

```
(lambda (x1 ... xn) a)
```

eine Funktion mit n Argumenten. Den Wert der Funktionsaufrufe mit dieser Funktion (also der Ausdrücke der Form

```
((lambda (x1 ... xn) a) (b1 ... bn))
```

erhält man, indem man die Variablen x1 ... xn durch die n Argumente des Aufrufs ersetzt. Beispielsweise ist

```
((lambda (x y) (2*x + y)) (3 4)) = 2*3 + 4 = 10 ,
```

während

```
((lambda (y x) (2*x + y)) (3 4)) = 2*4 + 3 = 11
```

ist.

Die Variablen in einem Lambdaausdruck sind sogenannte Parameter (oder gebundene Variable). Interessant ist, daß eine Funktion sich nicht ändert, wenn man eine Variable systematisch durch eine andere Variable ersetzt, die nicht bereits im Lambdaausdruck vorkommt.

```
(lambda (x y) (2*y + x))
```

ist also dasselbe wie

```
(lambda (u v) (2*v + u)) .
```

Manchmal werden wir Ausdrücke benutzen, in denen eine Variable nicht durch das Lambda gebunden ist. Beispielsweise ist das n in

```
(lambda (x y) (x*n + y*n))
```

nicht gebunden. Eine solche nicht gebundene Variable nennt man frei.
Wenn für eine freie Variable vor der Benutzung kein Wert vereinbart wurde, ist der Wert des Funktionsaufrufs nicht definiert, falls der Wert der Variablen auf das Ergebnis einen Einfluß hat.

Die Lambdanotation reicht allein für die Definition rekursiver Funktionen nicht aus. Neben den Variablen muß auch der Name der Funktion gebunden werden, weil er innerhalb der Funktion für eine Zeichenfolge steht.

Wir hatten die Funktion "firstatom" durch die Gleichung

```
(firstatom x) = (cond ((atom x) x)
                      (T (firstatom (head x)))
                )
```

definiert. Mit der Lambda – Notation können wir schreiben:

```
firstatom = (lambda (x) (cond ((atom x) x)
                              (T (firstatom (head x)))
            )           )
```

Das Gleichheitszeichen ist in Wirklichkeit nicht Teil der LISP – Sprache, sondern eine Krücke, die wir nicht mehr brauchen, wenn wir die richtige Schreibweise eingeführt haben.

Die rechte Seite der obigen Gleichung ist als Funktion nicht vollständig, da dort nichts darauf hinweist, daß das "firstatom" im einem bedingten Ausdruck für eben die rechte Seite steht. Deshalb ist die rechte Seite als Definition für die linke Seite ("firstatom") noch nicht geeignet.

Damit wir Definitionen schreiben können, in denen der Name der gerade definierten Funktion auftaucht, führen wir die Label – Notation ein (engl. label = Marke, (Preis –) Schildchen). Wenn "a" eine Funktion ist, und "n" ihr Name, schreiben wir "(label n a)".

Nun können wir die Funktion "firstatom" ohne Gleichheitszeichen schreiben:

```
(label firstatom (lambda (x) (cond ((atom x) x)
                                   (T (firstatom (head x)))
)                )           )
```

In dieser Definition ist "x" eine gebundene Variable und "firstatom" ein gebundener Funktionsname.

2. Das LISP – Interpreter – System

2.1 Die universelle LISP – Funktion "evalquote"

Ein Interpreter oder eine allgemeine Funktion ist eine Funktion, die den Wert jedes gegebenen Ausdrucks berechnen kann, wenn der Ausdruck in einer geeigneten Form vorliegt. (Wenn der zu interpretierende Ausdruck einen Aufruf einer unendlich rekursiven Funktion enthält, wird der Interpreter natürlich ebenfalls unendlich rekursiv.)
Wir sind jetzt in der Lage, eine allgemeine LISP – Funktion

```
(evalquote function arguments)
```

zu definieren. "evalquote" muß als erstes Argument ein LISP – Ausdruck übergeben werden. Dieser wird als Funktion aufgefasst und auf die folgenden Argumente angewendet.

Im Folgenden sind einige nützliche Funktionen zur Manipulation von LISP – Ausdrücken angegeben. Einige von ihnen werden als Hilfsfunktionen für die Definition von "evalquote" gebraucht, die wir uns vorgenommen haben.

```
(equal x y)
```

ist ein Prädikat, das wahr ist, wenn seine Argumente gleiche LISP – Ausdrücke sind. (Das elementare Prädikat "eq" ist ja nur für Atome definiert.)

Die Definition von "equal" ist ein Beispiel für einen bedingten Ausdruck innerhalb eines bedingten Ausdrucks.

```
(label equal
  (lambda (x y)
    (cond
      ((atom x) (cond
                  ((atom y) (eq x y))
                  (T F)
                )
      )
      ((equal (head x) (head y)) (equal (tail x) (tail y)))
      (T F)
    )
  )
)
```

Folgende Funktion liefert einen LISP – Ausdruck, der gleich mit "destination" ist, außer daß darin überall statt "old" "new" steht.

```
(changeall (destination old new))

=   (cond ((equal destination old) new)
          ((atom destination) destination)
          (T (cons (changeall (head destination) old new)
                   (changeall (tail destination) old new)
             )
          )
    )
```

Beispielsweise gilt

```
(changeall ((A . B) . C) B (X . A)) = ((A . (X . A)) . C)
```

Die folgenden Funktionen sind nützlich, wenn Listen verarbeitet werden sollen.

1. (append x y)
   hängt an die Liste "x" den LISP – Ausdruck "y".

```
(append x y) =
  (cond ((null x) y)
        (T (cons (head x) (append (tail x) y) ))
  )
```

2. (member list pattern)
   Dies Prädikat testet, ob der LISP – Ausdruck "pattern" in der Liste "list" vorkommt.

```
(member list pattern) =
  (cond ((null list) F)
        ((equal (head list) pattern) T)
        (T (member (tail list) pattern))
  )
```

3. (pairlist list1 list2 oldpairlist)
   Diese Funktion liefert eine Liste von Paaren, die die sich entsprechenden Elemente der Listen "list1" und "list2" enthalten, und an der noch die Liste "oldpairlist" hängt.

```
(pairlist list1 list2 oldpairlist) =
  (cond ((null list1) oldpairlist)
        (T (cons (cons (head list1) (head list2))
                 (pairlist (tail list1) (tail list2) oldpairlist)
           )
        )
  )
```

```
Beispiel:
   (pairlist (A B C) (U V W) ((D . X) (E . Y)) ) =
            ((A . U) (B . V) (C . W) (D . X) (E . Y))
```

Eine solche Liste von Paaren wird auch Assoziationsliste genannt, wenn das erste Element jedes Paars ein Atom ist, das über diese Liste mit dem zweiten Element assoziiert ist.

5. (association pattern associationlist)
   Wenn "association list" eine Assoziationsliste wie oben beschrieben ist, liefert "association" das Paar der Liste, dessen erstes Element "pattern" ist. Es ist also eine Funktion zum Durchsuchen von Tabellen.

```
(association pattern alist) =
  (cond ((eq (head (head alist)) pattern) (head alist))
        (T (association pattern (tail alist)))
  )
```

```
Beispiel:

(association B ( (A . (M N))
                 (B . (HEAD X))
                 (C . (QUOTE M))
                 (B . (TAIL X))
               )                  ) = (B . (HEAD X))
```

(replace expr alist)
   "alist" muß eine Assoziationsliste sein. "replace" produziert einen Ausdruck, der "expr" sehr ähnlich ist, nur sind alle Atome darin durch den LISP – Ausdruck ersetzt, mit dem sie in "alist" assoziiert sind.

```
(replace expr alist) =
  (cond ((atom expr) (association expr alist))
        (T (cons (replace (head expr) alist)
                 (replace (tail expr) alist)
           )
        )
  )
```

Beispiel:

```
(replace (X SCHRIEB Y)
         ((Y . (GOETZ VON BERLICHINGEN)) (X . GOETHE))
)

=        (GOETHE SCHRIEB (GOETZ VON BERLICHINGEN))
```

Die allgemeine Funktion "evalquote", die wir jetzt definieren wollen, gehorcht der folgendem Beispiel zugrundeliegenden Regel:

Beispiel:

```
                    (evalquote
Funktion:             (LAMBDA (X Y) (CONS (HEAD X) Y) )
Argumente:            (A B)  (C D)
                    )
=
                    (apply
Funktion:             (LAMBDA (X Y) (CONS (HEAD X) Y))
Argumentliste:        ((QUOTE (A B)) (QUOTE (C D)))
Bindung:              NIL
                    )
```

Die Argumente von "evalquote" werden also zu einer gequoteten Argumentliste von "apply". Die QUOTE – Funktion bewirkt, daß das Argument der QUOTE – Funktion wörtlich genommen, also nicht weiter evaluiert wird. Das dritte Argument von "apply", das NIL ist eine leere Bindeliste zur Bindung von Parametern und Argumenten im nächsten Schritt:

```
=
                         (eval
Argumente:                 (CONS (HEAD X) Y)
Bindung:                   ((X.(A B)) (Y . (C D)))
                         )
=
                         (cons (head (A B)) (C D))
=
                         (A C D)            =  Ergebnis von "evalquote" .
```

"evalquote" wird hauptsächlich durch die Hilfsfunktion "apply" definiert. "apply" berechnet Funktionsaufrufe, indem es die Argumente und die Parameter der Funktion bindet und den Funktionsrumpf berechnet. Die Bindungen werden in einer Assoziationsliste, der Bindeliste, gespeichert. Da bedingte Ausdrücke und Konstanten formal wie Funktionsaufrufe von Funktionen "cond" und "quote" aussehen, werden sie auch so behandelt.

Wir definieren also:

```
(evalquote fkt expr) = (apply fkt (quote expr) NIL) .
```

sowie :

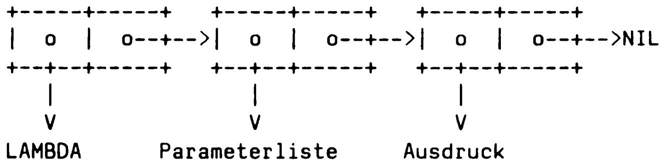
```
(eval expr binding) =
  (cond ((atom expr) (tail (association expr binding)))
        (T (apply (head expr) (tail expr) binding))
  )
```

"eval" stellt also erst fest, ob es sich um ein Atom oder um einen Funktionsaufruf handelt. Da es nur diese beiden Möglichkeiten gibt, ist diese Einteilung vollständig.

Atome sind immer Übersetzungen von Variablen, für die eine Bindung existieren muß, so daß ihr Wert aus der Bindeliste geholt wird.

Funktionsaufrufe sind immer Listen; im zweiten Zweig werden die Funktion und die Parameterliste getrennt und an "apply" übergeben.

Um sich die Aktionen in diesem zweiten Zweig von "eval" genauer vorstellen zu können, ist vielleicht die in Abschnitt 1.1 beschriebene graphische Darstellungsmethode hilfreich; beispielsweise würde sich ein Lambda – Ausdruck so ausnehmen:



"apply" bekommt nun von "eval" eine Funktion und eine Parameterliste sowie die Bindeliste übergeben. Mit diesen beiden macht es folgendes:

```
 (apply fn args binding) =
(cond
  ((atom fn)
     (cond ((eq fn HEAD)  (head (eval (head args) binding)))
           ((eq fn TAIL)  (tail (eval (head args) binding)))
           ((eq fn CONS)  (cons (eval (head args) binding)
                                (eval (head (tail args)) binding)
           )              )
           ((eq fn ATOM)  (atom (eval (head args) binding)))
           ((eq fn EQ)    (eq   (eval (head args) binding)
                                (eval (head (tail args)) binding)
           )              )
           ((eq fn QUOTE) (head args))
           ((eq fn COND) (evalcond args binding))
           (T (apply (tail (association fn binding)) args binding))
     )
  ((eq (head fn) LABEL)
     (apply (head (tail (tail fn)))
             args (cons (cons (head (tail fn))
                              (head (tail (tail fn)))
                        )
                        binding)
  ) )
  ((eq (head fn) LAMBDA) (eval (head (tail (tail fn)))
                               (pairlist (head (tail fn))
                                         args binding)
  )                      )
)
```

Das erste Argument von "apply" ist eine Funktion (unter der Voraussetzung, daß "quote" und "cond" als Funktionen anerkannt werden).

Wenn es eine der elementaren Funktionen "head", "tail", "cons", "atom" oder "eq" ist, wird die jweilige Funktion auf die Argumente angewandt, die vorher berechnet werden. Diese Berechnung erfolgt mit "eval", das ja für Variablen Werte aus der Bindeliste liefert und für Funktionsaufrufe das, was "apply" mit ihnen machen kann.

Wenn es sich um "quote" handelt, wird das erste Argument unverändert geliefert "quote" heißt ja "dies ist eine Konstante, die so, wie sie da steht, übernommen werden soll".

Wenn es sich um "cond" handelt, wird die Funktion "eval cond" aufgerufen, doch auch ihre Argumente werden nicht berechnet, außerdem gehört die Assoziationsliste zu den Argumenten:

```
eval (cond condlist, binding) =
    (cond ((eval (head (head condlist)) binding)
            (eval (head (tail (head condlist))) binding)
           )
           (T (cond (tail condlist) binding))
    )
```

Hier empfiehlt es sich, einen bedingten Ausdruck in graphischer Form hinzuschreiben und die Auswertung anhand der Zeichnung nachzuvollziehen.

Wenn die Funktion nichts von alledem ist, wird in der Bindeliste nachgesehen, ob dies Atom nicht an eine Funktion gebunden ist; falls ja, wird eine Auswertung dieser Funktion mit den gleichen Argumenten versucht.

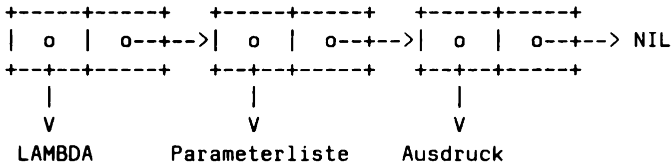
Wenn das erste Argument von "apply" kein Atom ist, muß es ein LABEL– oder ein LAMBDA–Ausdruck sein.

Ein LABEL–Ausdruck hat die Form

```
+-----+-----+    +-----+-----+    +-----+-----+
|  o  |  o--+-->|  o  |  o--+-->|  o  |  o--+--> NIL
+--+--+-----+    +--+--+-----+    +--+--+-----+
   |                |                |
   V                V                V
 LABEL             Name           Funktion
```

Funktionsname und Definition werden in einem funktionalen Eintrag in die Bindeliste eingefügt, so daß der Name an die Funktion gebunden ist.

Ein LAMBDA – Ausdruck hat die Form



Dabei ist die Parameterliste eine Liste von Atomen, den Parametern. Die Auswertung läuft so ab, daß die Parameter durch "pairlist" an die Argumente gebunden werden und mit dieser neuen Bindeliste der Ausdruck berechnet wird.

Das EUMEL – LISP bietet eine Reihe weiterer Möglichkeiten, die erst später beschrieben werden. Hier können wir allerdings schon die folgenden Punkte abhandeln:

1. Jede LISP – Eingabe ist ein LISP – Ausdruck. Der "head" dieses Ausdrucks wird als Funktion aufgefaßt und auf den gequoteten "tail" des Ausdrucks, nämlich die nicht zu evaluierenden Argumente angewandt. Die Übersetzung von Kleinbuchstaben in Großbuchstaben wird vom LISP – System übernommen.

2. In der Theorie des reinen LISP müssen alle Funktionen außer den fünf Basisfunktionen an allen Stellen wieder definiert werden, an denen sie aufgerufen werden. Das ist eine für die Praxis äußerst unhandliche Regelung; das EUMEL – LISP – System kennt weitere vordefinierte Funktionen und bietet die Möglichkeit, beliebig viele weitere Standardfunktionen einzuführen, auch solche Funktionen, deren Argumente nicht berechnet werden (wie "quote") oder solche, die beliebig viele Argumente haben dürfen (wie "cond").

3. Die Basisfunktion "eq" hat immer einen wohldefinierten Wert, dessen Bedeutung im Fall, daß Nicht – Atome verglichen werden, im Kapitel über Listenstrukturen erklärt wird.

4. Außer in sehr seltenen Fällen schreibt man nicht (quote T), (quote F) oder (quote NIL), sondern T, F und NIL.

5. Es besteht die Möglichkeit, mit Ganzzahlen zu rechen, die als weiterer Typ von Atomen gelten. Außerdem können TEXTe und Einzelzeichen (CHARACTERs) gespeichert werden.

6. Es besteht die Möglichkeit der Ein– und Ausgabe von LISP – Ausdrücken, Ganzzahlen, TEXTen und CHARACTERs.

WARNUNG: Die oben angegebenen Definitionen von "eval" und "apply" dienen nur pädagogischen Zwecken und sind nicht das, was wirklich im Interpreter abläuft.
Um zu entscheiden, was wirklich vor sich geht, wenn der Interpreter aufgerufen wird, sollte man sich an die ELAN – Quellprogramme halten.

2.2 Anwendungsregeln und Beispiele

Die Funktionsweise des LISP – Interpreteres kann bequem unter Verwendung der Funktion "trace" verfolgt werden. Der Aufruf:

```
(trace)
```

schaltet den Trace – Protokollmodus des Interpreters ein bzw. aus.

Das folgende Beispiel ist ein LISP – Programm, das die drei Funktionen "union", "intersection" und "member" als Standardfunktionen einführt Die Funktionen lauten folgendermaßen:

```
(member pattern list) = (cond ((null list) F)
                              ((eq (head list) pattern) T)
                              (T (member pattern (tail list)))
                        )

(union x y) = (cond ((null x) y)
                    ((member (head x) y) (union (tail x) y))
                    (T (cons (head x) (union (tail x) y)))
              )

(intersection x y) = (cond ((null x) NIL)
                           ((member (head x) y)
                               (cons  (head  x)  (intersection
                               (tail x) y))
                           )
                           (T (intersection (tail x) y))
                     )
```

Um die Funktionen als neue Standardfunktionen einzuführen, benutzen wir die Pseudofunktion "define":

```
(DEFINE
  (MEMBER . (LAMBDA (PATTERN LIST)
             (COND ((NULL LIST) F)
                   ((EQ (HEAD LIST) PATTERN) T)
                   (T (MEMBER PATTERN (TAIL LIST)))
  )          ) )
  (UNION . (LAMBDA (X Y)
            (COND ((NULL X) Y)
                  ((MEMBER (HEAD X) Y) (UNION (TAIL X) Y))
                  (T (CONS (HEAD X) (UNION (TAIL X) Y)))
  )         ) )
  (INTERSECTION . (LAMBDA (X Y)
                   (COND ((NULL X) NIL)
                         ((MEMBER (HEAD X) Y)
                           (CONS (HEAD X) (INTERSECTION (TAIL
                           X) Y))
                         )
                         (T (INTERSECTION (TAIL X) Y))
  )                ) )
)
```

Die Funktion DEFINE, liefert als Pseudofunktion nicht nur einen LISP – Ausdruck als Ergebnis, sondern hat auch einen bleibenden Effekt, nämlich eine Veränderung im LISP – Heap.

DEFINE hat beliebig viele Parameter der Form (Name . Funktion) und bewirkt, daß die Funktionen unter dem jeweiligen Namen im System verfügbar werden, also für die weitere Programmausführung definiert werden. Das Ergebnis von DEFINE ist eine Liste der neuen Funktionsnamen, also hier

```
(MEMBER UNION INTERSECTION)
```

Der Wert den der LISP – Interpreter bei Eingabe von

```
(intersection (a1 a2 a3) (a1 a3 a5))
```

liefert ist `(A1 A3)` ,

Die Funktion

```
(union (x y z) (u v w x))
```

liefert `(Y Z U V W X)` .

Es folgen einige elementare Regeln für LISP – Programme:

1. Ein LISP – Programm besteht aus einem Funktionsaufruf. Im Beispiel ist das die Funktion DEFINE, die ihre Parameter (beliebig viele) berechnet und ausgibt. Die Berechnung der Parameter erfolgt dabei in der Reihenfolge der Parameter (normale LISP – Funktionen mit mehreren Parametern berechnen standardmäßig alle Parameter, allerdings in irgendeiner Reihenfolge).

2. LISP ist formatfrei, d.h. jedes Symbol kann in jeder Spalte stehen. Für die Bedeutung des Programms ist nur die Reihenfolge der Symbole maßgeblich. Zeilenwechsel wird als Leerzeichen aufgefaßt.

3. Atome müssen mit einem Buchstaben anfangen, damit sie nicht mit Zahlen verwechselt werden.

4. Ein LISP – Ausdruck der Form (A B C . D) ist eine Abkürzung für (A.(B.(C.D))). Jede andere Plazierung des Punkts ist ein Fehler (falsch wäre z.B. (A . B C)).

5. Eine Anzahl von Basisfuntionen existiert von Anfang an, ohne daß sie durch DEFINE eingeführt wurden. Der Programmierer kann weitere Funktionen bleibend oder für die Dauer eines Programmlaufs einführen; dabei ist die Reihenfolge der neuen Funktionen gleichgültig.

2.3 Variablen

Eine Variable ist ein Symbol, das ein Argument einer Funktion repräsentiert. Man kann also schreiben: "a + b, wobei a = 3 und b = 4". In dieser Situation ist keine Verwechslung möglich, so daß klar ist, daß das Ergebnis 7 ist. Um zu diesem Ergebnis zu kommen, muß man die Zahlen anstelle der Variablen einsetzen und die Operation ausführen, d.h. die Zahlen addieren.

Ein Grund, weshalb das eindeutig ist, liegt darin, daß "a" und "b" nicht "direkt" addiert werden können, so daß etwa "ab" entsteht. In LISP kann die Situation viel komplizierter sein. Ein Atom kann eine Variable oder ein Atom sein.

Sollte der zukünftige LISP – Benutzer an dieser Stelle entmutigt sein, sei ihm gesagt, daß hier nichts Neues eingeführt wird. Dieser Abschnitt ist nur eine Wiederholung der Überlegungen aus Abschnitt 1.4. Alles, was wir in diesem Abschnitt sagen, kann man aus den Regeln für LISP – Ausdrücke oder aus der allgemeinen Funktion "evalquote" ableiten.

Der Formalismus, der in LISP die Variablen kennzeichnet, ist die Lambdanotation von Church. Der Teil des Interpreters, der die Variablen an Werte bindet, heißt "apply". Wenn "apply" auf eine Funktion stößt, die mit LAMBDA anfängt, wird die Variablenliste (Argumentliste) mit der Parameterliste gepaart und am Anfang der Bindeliste eingefügt.

Während der Berechnung des Funktionsrumpfs müssen manchmal Variablen durch ihre Werte ersetzt werden. Das geschieht dadurch, daß ihr Wert in der Bindeliste nachgesehen wird. Wenn eine Variable mehrmals gebunden wurde, wird die zuletzt etablierte Bindung verwendet. Der Teil des Interpreters, der diese "Berechnungen" und die Berechnung von Funktionsaufrufen durchführt, heißt "eval".

2.4 Konstanten

Manchmal heißt es, eine Konstante stehe für sich selbst, im Gegensatz zu einer Variablen, die für etwas anderes, nämlich ihren Wert, steht.
Dies Konzept funktioniert in LISP nicht so ohne weiteres; es ist hier sinnvoller, zu sagen, eine Variable ist konstanter als die andere, wenn sie in einer höheren Ebene gebunden ist und ihren Wert seltener ändert.
In LISP bleibt eine Variable im Bereich des LAMBDA konstant, von dem sie gebunden ist. Wenn eine Variable einen festen Wert hat, unabhängig davon, was in der Bindeliste steht, wird sie (echte) Konstante genannt. Dies wird mit Hilfe der Eigenschaftsliste (E – Liste) des Atoms erreicht.
Jedes Atom hat eine E – Liste, in der Paare von Atomen und beliebigen Strukturen gespeichert sind. Ein Atom hat die Eigenschaft A, wenn in seiner E – Liste ein Paar mit dem Atom A enthält; die dazugehörige "beliebige Struktur" heißt Wert dieser Eigenschaft.
Wenn ein Atom die Eigenschaft APVAL besitzt, ist es eine Konstante, deren Wert der Wert der Eigenschaft ist.
Konstanten können durch die Pseudofunktion

```
(set atom wert)
```

gesetzt werden; nach der Auswertung eines solchen Aufrufs hat das Atom "atom" immer den Wert "wert" – bis zum nächsten "set". Eine interessante Klasse von Konstanten sind solche Konstanten, die sich selbst als Wert haben. Ein Beispiel dafür ist NIL. Der Wert dieser Konstanten ist wieder NIL. Daher kann NIL nicht als Variable benutzt werden, da es ja eine Konstante ist. (T und F gehören ebenfalls zu dieser Klasse).

2.5 Funktionen

Wenn ein LISP – Ausdruck für eine Funktion steht, ist die Situation ähnlich der, in der ein Atom für einen Wert steht. Wenn die Funktion rekursiv ist, muß sie einen Namen bekommen. Das geht mit einem LABEL – Ausdruck, der den Namen mit der Funktionsdefinition in der Bindeliste paart. Dadurch wird der Name an die Funktionsdefinition gebunden, so wie eine Variable an ihren Wert gebunden wird. In der Praxis setzt man LABEL selten ein. Normalerweise ist es einfacher, Name und Definition wie bei den Konstanten zu verknüpfen. Dies geschieht mit der Pseudofunktion DEFINE, die wir am Anfang des Kapitels benutzt haben.
Diese Funktion kann beliebig viele Parameter der Form

```
(atom . funktion)
```

haben, wobei "atom" der Name der zu definierenden Funktion "funktion" werden soll. Sie bewirkt, daß die Definition unter der Eigenschaft FUNCTION in der E – Liste des Atoms abgelegt wird.

3. Erweitertes LISP

In diesem Kapitel werden wir einige Erweiterungen zum reinen LISP einführen. Zu diesen Erweiterungen gehören Möglichkeiten für Arithmetik, Zeichenkettenverarbeitung, Funktionen, die spezielle Argumente erwarten, und Ein- und Ausgabe.

In allen Fällen handelt es sich bei den Erweiterungen um zusätzliche Funktionen. So heißt das Kommando für die Ausgabe eines LISP-Ausdrucks PUT. Syntaktisch ist PUT nichts anderes als eine Funktion mit einem Argument. Sie kann mit anderen Funktionen verkettet werden, und diese Verkettung wird ganz auf die übliche Art behandelt, zuerst Berechnung der innern, dann der äußeren Funktionsaufrufe. Ein Ergebnis ist nur in dem trivialen Sinn vorhanden, daß PUT sein Argument wieder liefert, also die Identität ist.

Funktionen, die eine Aktion wie Ein- oder Ausgabe bewirken, oder die Langzeitwirkung (gesehen auf die Ausführungsdauer des Programms) haben, wie DEFINE und SET, heißen Pseudofunktionen. Es ist eine Besonderheit von LISP, daß alle Funktionen einschließlich den Pseudofunktionen ein Ergebnis haben müssen. In einigen Fällen ist das Ergebnis trivial und kann ignoriert werden.

In diesem Kapitel beschreiben wir verschiedene Erweiterungen der Sprache LISP, die im System fest enthalten sind.

3.1 Gequotete Parameter

Bevor ein Argument an eine Funktion übergeben wird, wird erst sein Wert in der Bindeliste nachgesehen, d.h. es wird nicht der Name der Variablen übergeben, sondern ihr Wert. Wenn das Argument als Konstante behandelt werden soll, muß es ge"quotet" werden, d.h. statt "argument" steht (quote argument). Wenn ein Argument einer Funktion immer als Konstante behandelt werden soll, ist es bequemer, das Argument nicht jedesmal zu quoten. Das EUMEL - LISP - System erlaubt, in diesem Fall den formalen Parameter in der Funktionsdefinition bereits zu quoten.

Dieser Mechanismus wurde auch benutzt, um QUOTE zu implementieren; die Funktion lautet

```
quote = (lambda ((QUOTE x)) x)
```

3.2 Funktionen mit beliebig vielen Argumenten

Ein Beispiel ist "list", das beliebig viele Argumente haben kann, die zu einer Liste zusammengefaßt werden. Da eine Funktion nur eine feste Anzahl von Parametern haben kann, eine Funktion mit beliebig vielen Argumenten aber gewiß keine feste Anzahl von Argumenten hat, werden die beliebig vielen Argumente zu einer Liste zusammengefaßt und ein einziger Parameter wird an diese Liste gebunden. Da "list" genau diese Liste liefern soll, wird diese Funktion ebenfalls zu einer "Identität":

```
list = (lambda ((INDEFINITE x)) x)
```

Solche Parameter werden durch INDEFINITE gekennzeichnet. Sie können auch gequotet werden, indem man (INDEFINITE QUOTE parameter) schreibt; das wirkt so, als wären alle Argumente, die diesem Parameter zugeordnet werden, einzeln gequotet worden.

```
evalquote = (lambda (fkt (INDEFINITE QUOTE expr))
                    (apply fkt expr NIL) )
```

3.3 Funktionale Parameter

In der Mathematik gibt es Funktionen, die andere Funktionen als Argument haben. In der Algebra könnte man die Funktion "(operation operator a b)" definieren, wobei "operator" ein funktionales Argument ist, das die Operation festlegt, die auf "a" und "b" ausgeführt werden soll. Beispielsweise gilt

```
operation (+ 3 4) = 7
operation (* 3 4) = 12
```

In LISP sind funktionale Argumente sehr nützlich. Eine wichtige Funktion mit einem Argument ist MAPLIST. Ihre Definition ist

```
(LAMBDA (LIST (FUNCTION FN))
  (COND ((NULL LIST) NIL)
        (T (CONS (FN (HEAD LIST)) (MAPLIST (TAIL LIST) FN)))
) )
```

Diese Funktion nimmt eine Liste und eine Funktion als Argument und wendet die Funktion auf die Listenelemente an.

3.4 Prädikate und boolesche Konstanten

Die booleschen Werte sind, wie in Kapitel 1 gesagt, T und F. Bei LISP – Ausdrücken müßte daraus (quote T) und (quote F) werden, aber da die APVALs dieser Atome wieder den Wert T und F haben, ist das quoten nicht nötig.

Prädikate sind Funktionen, die T oder F als Ergebnis haben; es gibt also keine formalen Unterschiede zwischen anderen Funktionen und Prädikaten.

Daher ist es durchaus möglich, daß eine Funktion einen Wert liefert, der weder T noch F ist, daß aber durch einen bedingten Ausdruck an dieser Stelle ein boolescher Ausdruck verlangt wird. In diesem Fall ist die Wirkung des Ausdrucks nicht definiert.
Das Prädikat EQ hat folgendes Verhalten:

1. Wenn seine Argumente verschieden sind, ist das Ergebnis F.
2. Wenn die Argumente dasselbe Atom sind, ist das Ergebnis T.
3. Wenn die Argumente gleich, aber nicht atomar sind, ist das Ergebnis T oder F, je nachdem, ob sie ein und dasselbe Objekt im Heap sind oder nicht.

3.5 Unbenannte Atome

Die meisten Atome im EUMEL – LISP haben einen Namen, der sie bei Ein– und Ausgabeoperationen identifiziert.
Es gibt aber auch Atome, die keinen Namen haben und stattdessen durch ihre Werte repräsentiert werden. Momentan sind das Ganzzahlen und Zeichenketten (TEXTe); auch die booleschen Werte kann man in einem weiteren Sinn dazurechnen.

3.5.1 Ganzzahlen

Im EUMEL – LISP gibt es Funktionen, die Basisoperationen und Tests durchführen.

Ganzzahlen haben folgende Eigenschaften:

1. Eine Ganzzahl besteht aus einem optionalen Vorzeichen und einer Folge von Ziffern; zwischen Vorzeichen und Ziffern können Leerzeichen stehen.
2. Der Wert einer Ganzzahl liegt zwischen –32768 und 32767 (minint und maxint).
3. Eine Ganzzahl kann überall dort stehen, wo ein Atom stehen kann, außer als Parameter.
4. Ganzzahlen sind Konstanten; sie brauchen also nicht gequotet werden.

3.5.2 Arithmetische Funktionen und Prädikate

Es folgt eine Liste aller arithmetischen Funktionen.
Wenn ein Argument einer dieser Zahlen keine Ganzzahl ist, erfolgt eine Fehlermeldung.

(sum x1 ... xn)	liefert die Summe der xi; wenn keine Argumente gegeben werden, wird 0 geliefert.
(difference x y)	liefert die Differenz von x und y.
(product x1 ... xn)	liefert das Produkt seiner Argumente; wenn keine Argumente gegeben werden, wird 1 geliefert.
(quotient x y)	liefert den Quotienten von x und y, ohne den Rest zu berücksichtigen.
(remainder x y)	liefert den Rest der Division von x und y.
(getint)	liest eine Zahl vom Bildschirm ein und liefert sie.
(putint x)	gibt x auf den Bildschirm aus. Identitäts funktion.

3.5.3 Zeichenkettenverarbeitung

Im Moment ist nur Zeichenketten – Ein – und Ausgabe implementiert.
Die Ausgabe löst bei Argumenten, die keine Zeichenketten sind, eine Fehlermeldung aus.

(gettext)	liest eine Zeichenkette ein und liefert sie.
(puttext x)	gibt eine Zeichenkette aus.

3.5.4 Test auf Gleichheit

(equal x y)	testet, ob x und y vom gleichen Typ sind, und wenn ja, ob sie gleich sind.

3.6 Aufruf von EUMEL aus

Bevor man den LISP - Interpreter benutzen kann, muß er folgendermaßen implementiert werden:

archive ("lisp")
fetch all (archive)
release (archive)
check off
insert ("lisp.1")
insert ("lisp.2")
insert ("lisp.3")
insert ("lisp.4")
check on

Das LISP - System verfügt über einen Heap, in dem alle LISP - Ausdrücke gespeichert sind. Standardmäßig enthält der Heap eine Reihe von Funktionen, die nicht in den LISP - Programmen definiert werden müssen (Übersichten über die Standardfunktionen siehe Kapitel 3.5).

Mit

lisp

wird das LISP - System im EUMEL - Dialog gestartet. In einem Eingabefenster wird mit Hilfe des Paralleleditors eine LISP - EINGABE - Möglichkeit angeboten. Die Ausgabe erfolgt in dem LISP - AUSGABE - Fenster.
Das LISP - System kann folgendermaßen verlassen werden:
<ESC> <ESC> break lisp <RETURN>.

Statt dieser direkten Art der Benutzung der LISP - Maschine ist auch eine an ELAN angelehnte Art mit den Prozeduren "run lisp", insert lisp" usw. vorgesehen:

Mit

run lisp (TEXT CONST dateiname)

wird eine Kopie des Heaps angelegt, das Programm aus der Datei "dateiname" in die Kopie eingelesen und gestartet. Durch diesen Kopiermechanismus wird der Originalheap vor Zusammenbrüchen des LISP - Systems geschützt.

insert lisp (TEXT CONST dateiname)

bewirkt dasselbe wie "run lisp"; allerdings wird jetzt direkt auf dem Originalheap gearbeitet. Dadurch sind alle Änderungen im Heap, die das Programm verursacht (meist Definition von Funktionen durch DEFINE) bleibend, aber auch ein Zusammenbruch ist insoweit endgültig, als das LISP – System jetzt neu gestartet werden muß. Das geschieht mit

start lisp system (DATASPACE CONST dsname)

"dsname" gibt dabei den Datenraum an, der die zum Hochfahren notwendigen Daten enthält. Solche Daten im richtigen Format enthält der Datenraum "lisp.bootstrap".
Wenn der zuletzt benutzte Heap mit nicht mehr durch LISP – Programme erreichbare Strukturen vollgestopft ist, schafft die Prozedur

collect lisp heap garbage

Abhilfe; mit

lisp storage info

kann man den Erfolg kontrollieren.

4. Detailbeschreibungen

4.1 Grundfunktionen

Die Datei "lisp.1" enthält ein Paket, das die Grundlage des LISP – Systems bildet. Es implementiert

- die primitiven LISP – Funktionen wie "cons", "null", etc.,
- die Verwaltung des Heaps, in dem die LISP – Strukturen und die Objektliste (Oblist) gespeichert sind,
- einen Datentyp SYM, dessen Wertevorrat aus Zeigern auf die im Heap gespeicherten Strukturen besteht,
- Funktionen zur Konversion allgemeiner Daten in LISP – Strukturen (bisher realisiert: TEXT < – – > SYM und INT < – – > SYM).

Durch die Implementation der Basisoperationen als exportierte und damit allgemein verfügbare ELAN – Prozeduren ist es möglich, LISP – Strukturen durch ELAN – Programme zu manipulieren; insbesonders können ELAN – und LISP – Programme über diese Strukturen miteinander kommunizieren.

Anmerkung:
Wenn Eigenschaften von "SYM" – Objekten beschrieben werden, sind immer die Eigenschaften der Strukturen gemeint, auf die die Objekte zeigen, wenn nichts anderes angegeben wird.

Es werden folgende Prozeduren exportiert:

PROC initialize lisp system (DATASPACE CONST new heap):
"new heap" ist der neue Datenraum, in dem der LISP – Heap ab sofort geführt wird.
Vorsicht: Beim Wechsel zu einem neuen Datenraum sind die Werte der SYM – Variablen, die auf Strukturen im alten Heap zeigen, natürlich wertlos!

PROC dump lisp heap (FILE VAR f):
In "f" wird ein Dump des Heaps erstellt. Dieser Dump ist nur mit Kenntnis des Programmtextes aus "lisp 1" verständlich; er wird hier nicht beschrieben.

PROC lisp storage (INT VAR size, used):

Nach dem Aufruf gibt "size" die maximal verfügbare Anzahl von Knoten an, während "used" die Anzahl der tatsächlich von LISP – Strukturen belegten Knoten enthält. Zu diesen Strukturen können auch solche zählen, die nicht mehr durch "head" oder "tail" etc. erreichbar sind.

PROC collect lisp heap garbage:

Löscht die im LISP – Heap nicht mehr durch "atom (TEXT CONST)", "property", "head" und "tail" erreichbaren Strukturen. Es werden auch alle nur von ELAN – Programmen aus über SYM – Variable erreichbare Strukturen gelöscht, so daß die Werte dieser Variablen undefiniert werden.
Die Müllabfuhr wird von keiner Prozedur dieses Pakets aufgerufen, d.h. der Benutzer, der ELAN – Programme einsetzt, braucht nicht alle Strukturen in den Eigenschaftslisten von Atomen aufzubauen, um sie vor einer versehentlichen Löschung durch die Müllabfuhr zu schützen, vorausgesetzt, er ruft sie nicht selbst auf. Er muß allerdings darauf achten, daß im Heap noch genug Platz bleibt.

OP := (SYM VAR left, SYM CONST right):

Nach der Zuweisung zeigt "left" auf die gleiche Struktur wie vorher "right".

SYM CONST nil, pname;

Zwei Konstanten, die dem LISP – System ständig zur Verfügung stehen müssen. Ihre Drucknamen sind "NIL" bzw. "PNAME" (vgl. Schlußbemerkungen)

SYM PROC head (SYM CONST sym):

Entspricht der im Handbuch beschriebenen Funktion "head".

SYM PROC tail (SYM CONST sym):

Entspricht der im Handbuch beschriebenen Funktion "tail".

SYM PROC cons (SYM CONST head, tail):

Liefert einen SYM – Wert "zeiger" auf eine neue Struktur. Es gilt:
head ("zeiger") = "head" und tail ("zeiger") = "tail".

BOOL PROC eq (SYM CONST sym 1, sym 2):

Prüft, ob "sym 1" und "sym 2" auf dieselbe Struktur zeigen. Das ist genau dann der Fall, wenn sie durch Zuweisung auseinander hervorgegangen sind oder wenn sie auf das gleiche benannte Atom zeigen.

BOOL PROC equal (SYM CONST sym 1, sym 2):
Prüft, ob "sym 1" und "sym 2" dieselbe Struktur haben; "dieselbe Struktur" braucht aber nicht "Identität" zu bedeuten, wie "eq" das verlangt.
Umgewandelte TEXTe und INTs werden richtig verglichen (siehe "sym (INT CONST)" und "sym (TEXT CONST)").

BOOL PROC null (SYM CONST sym):
Prüft, ob "sym" gleich der Konstanten "NIL" ist (entspricht
eq ("sym", "NIL"), ist aber schneller).

BOOL PROC atom (SYM CONST sym):
Prüft, ob "sym" ein (benanntes oder unbenanntes) Atom ist.

BOOL PROC is named atom (SYM CONST sym):
Prüft, ob "sym" ein benanntes Atom ist.

PROC begin oblist dump:
Vorbereitung für "next atom".

SYM PROC next atom:
Liefert das nächste Atom aus der Objektliste. In der Objektliste sind alle benannten Atome, die der Heap enthält, aufgeführt (bis auf Ausnahmen; s."delete atom"). "NIL" wird immer als letzte Atom geliefert.

SYM PROC atom (TEXT CONST name):
Liefert einen Zeiger auf das Atom mit dem Namen "name". Wenn kein solches Atom in der Objektliste vorhanden ist, wird "NIL" geliefert.

SYM PROC new atom (TEXT CONST name):
Liefert einen Zeiger auf das Atom mit dem Namen "name". Wenn kein solches Atom in der Objektliste vorhanden ist, wird ein neues mit diesem Namen in sie eingefügt.

PROC create atom (TEXT CONST name):
Fügt ein Atom mit dem Namen "name" in die Objektliste ein. Wenn ein solches Atom bereits existiert, wird stattdessen eine Fehlermeldung ausgegeben.

PROC delete atom (SYM CONST atom):
Streicht das Atom "atom" aus der Objektliste.

PROC begin property list dump (SYM CONST atom):
Vorbereitung für "next property".

PROC next property (SYM VAR property id, property):
Liefert die nächste Eigenschaft aus der Eigenschaftsliste des zuletzt durch "begin property list dump" vorbereiteten Atoms. Wenn es sich bei der Eigenschaft um eine Flagge handelt, wird "property" auf "NIL" gesetzt; wenn es keine nächste Eigenschaft mehr gibt, werden "property" und "property id" auf "NIL" gesetzt.
Der Dump der Eigenschaftsliste beeinträchtigt die "Verwendbarkeit" des Atoms in keiner Weise; es ist während des Dumps sogar möglich, Eigenschaften und Flaggen zu lesen. Wenn während des Dumps Eigenschaften oder Flaggen geändert oder geschrieben werden, ist mit fehlerhaften Dumpergebnissen zu rechnen.

PROC add property (SYM CONST atom, property id, property):
"property id" muß ein benanntes Atom sein. Führt eine neue Eigenschaft mit der Bezeichnung "property id" und dem Wert "property" ein. Wenn bereits eine Eigenschaft mit der gleichen Bezeichnung existiert, wird die alte Version überdeckt, ist aber weiter vorhanden.

PROC alter property (SYM CONST atom, property id, property):
Bringt die Eigenschaft mit der Bezeichnung "property id" auf den neuen Wert "property". Wenn eine Eigenschaft mit dieser Bezeichnung noch nicht existiert, wird eine Fehlermeldung ausgegeben.

BOOL PROC property exists (SYM CONST atom, property id):
Prüft, ob das Atom eine Eigenschaft mit der Bezeichnung "property id" besitzt.

SYM PROC property (SYM CONST atom, property id):
Liefert den Wert der gerade sichtbaren Eigenschaft des Atoms, die die Bezeichnung "property id" hat. Falls die Eigenschaft nicht existiert, wird "NIL" geliefert.

PROC delete property (SYM CONST atom, property id):
Löscht den gerade sichtbaren Wert der Eigenschaft des Atoms, die die Bezeichnung "property id" hat. Wenn eine ältere Version dieser Eigenschaft durch "add property" überdeckt wurde, wird diese jetzt wieder sichtbar. Jede Eigenschaft bildet also für jedes Atom einen Stapel (Stack).

PROC add flag (SYM CONST atom, flag id):
Das Atom "atom" erhält die Flagge "flag id". Ein Atom kann dieselbe Flagge durchaus mehrmals haben.

BOOL PROC flag (SYM CONST atom, flag id):
Prüft, ob "atom" mindestens eine Flagge "flag id" hat.

PROC delete flag (SYM CONST atom, flag id):
Löscht eine Flagge "flag id" von "atom". Wenn keine Flagge existiert, wird nichts getan.

SYM PROC sym (TEXT CONST text):
Konvertiert "text" in ein unbenanntes Atom und liefert einen Zeiger auf dies Atom.

TEXT PROC text (SYM CONST sym):
Konvertiert "sym" in einen TEXT zurück, wenn es sich um einen konvertierten TEXT handelt; wenn nicht, wird eine Fehlermeldung ausgegeben.

BOOL PROC is text (SYM CONST sym):
Prüft, ob "sym" ein konvertierter TEXT ist.

SYM PROC sym character (TEXT CONST text):
"text" muß genau ein Zeichen enthalten. Das Zeichen wird in ein CHARACTER-Objekt im Heap konvertiert und ein Zeiger auf dies Objekt geliefert.

INT PROC character (SYM CONST sym):
"sym" muß auf ein CHARACTER-Objekt zeigen. Geliefert wird der Code des dort gespeicherten Zeichens.

SYM PROC sym (INT CONST i 1, i 2):
Konvertiert "i 1" und "i 2" in ein unbenanntes Atom und liefert einen Zeiger darauf.

INT PROC int 1 (SYM CONST sym):
INT PROC int 2 (SYM CONST sym):
Holt die Werte der ersten bzw. zweiten Ganzzahl aus "sym", wenn es sich um ein konvertiertes INT-Paar handelt; wenn nicht, wird eine Fehlermeldung ausgegeben.

BOOL PROC is int pair (SYM CONST sym):
Prüft, ob "sym" ein konvertiertes INT-Paar ist.

Prozedurübergreifende Aussagen über das Paket "lisp.1":

- Es gibt benannte und unbenannte Atome.

- Die unbenannten Atome sind Konversionsprodukte.

- Vor dem ersten Aufruf von "delete atom" sind alle benannten Atome in der Objektliste enthalten; d.h. sie können alle durch "begin oblist dump" und wiederholten Aufruf von "next atom" erreicht werden.

- Jedes benannte Atom hat genau einen Namen, der immer gleich bleibt. Der Name ist als Eigenschaft mit der Bezeichnung "pname" in der Eigenschaftsliste gespeichert. "add property", "alter property" und "delete property" geben deshalb eine Fehlermeldung aus, statt ihre normalen Aktionen durchzuführen, wenn ihnen als Eigenschaftsbezeichnung "pname" übergeben wird.

- Es gibt keine zwei Atome, die denselben Namen haben; dadurch reduziert sich die bei "eq" angegebene Fallunterscheidung auf einen Fall.

- Es kann durchaus zwei unbenannte Atome mit gleichen Werten geben, die von "eq" nicht als gleich anerkannt werden, weil sie in verschiedenen Strukturen gespeichert sind. "equal" achtet nicht auf die Position, sondern auf die Werte der zu vergleichenden Strukturen.

- Mehrfache Zugriffe auf die gleiche Eigenschaft desselben Atoms werden so optimiert, daß die Eigenschaftsliste nur beim ersten Zugriff (meist durch "property exists") durchsucht werden muß.

4.2 Weitere Funktionen sowie Eingabe und Ausgabe

Die Datei "lisp.2" enthält diverse Pakete, die die Verbindung vom LISP – System zur normalen EUMEL – Umgebung bilden. Momentan sind das Ein – und Ausgabe und (exemplarisch) die fünf Grundrechenarten für Ganzzahlen.

Die Ein – und Ausgabe von LISP – Strukturen wird durch das Paket namens "lisp io" mit den folgenden Prozeduren ermöglicht:

PROC get (FILE VAR f, SYM VAR sym):
Nach dem Aufruf zeigt "sym" auf eine neue aus "f" eingelesene Struktur.
In der ersten und hinter der letzten Zeile des S – Ausdrucks dürfen keine weiteren Daten stehen.

PROC get all (FILE VAR f, SYM VAR sym):
Wie "get (FILE V, SYM V)", nur daß die Datei nichts als den S – Ausdruck enthalten darf.

PROC get (SYM VAR sym):
Es wird mit "get all" ein S – Audruck von einer Scratch – Datei eingelesen, die dem Benutzer vorher zum Editieren angeboten wird. Bei Einlesefehlern wird die Datei zu Korrigieren angeboten, bis keine Fehler mehr auftreten.

PROC put (FILE VAR f, SYM CONST sym):
Wenn "sym" ein Ganzzahlpaar ist, wird die erste Zahl ausgegeben; wenn es ein konvertierter TEXT ist, wird der ursprüngliche TEXT wieder ausgegeben; bei einem benannten Atom oder einer allgemeinen LISP – Struktur wird ein S – Ausdruck ausgegeben.

PROC put (SYM CONST sym):
Wie "put (FILE V, SYM CONST), außer daß die Augabe direkt auf den Bildschirm erfolgt.

Das Paket "lisp int" enthält die Prozeduren

SYM PROC sum (SYM CONST summandenliste);
Erwartet eine Liste von "int pair" – Summanden und liefert deren Summe.

SYM PROC difference (SYM CONST minuend, subtrahend):
Liefert die Differenz der Parameter.

SYM PROC product (SYM CONST faktorenliste):
Liefert das Produkt der Listenelemente.

SYM PROC quotient (SYM CONST dividend, divisor):
Liefert den Quotienten der Parameter.

SYM PROC remainder (SYM CONST dividend, divisor):
Liefert den Rest.

4.3 Interpreter

Die Datei "lisp.3" enthält das Paket "lisp interpreter", das die Prozedur

SYM PROC evalquote (SYM CONST expression)

exportiert. Es handelt sich dabei um den im EUMEL – LISP – Handbuch beschriebenen Interpreter.

Wenn "expression" ein LISP – Ausdruck ist, liefert die Prozedur den Wert des Ausdrucks (vorausgesetzt, der LISP – Heap ist vorbereitet, siehe lisp.1).

Wirkungsweise:
"evalquote" ruft im Wesentlichen die Prozedur "eval" auf.
"eval" erwartet als Argumente einen solchen LISP – Ausdruck wie "evalquote", benötigt aber zusätzlich eine sog. Bindeliste. In einer Bindeliste sind durch LAMBDA – und LABEL – Ausdrücke bereits gebundene Variable und ihre Werte gespeichert. Die Manipulation der Bindeliste ist durch eine Reihe von Refinements, die am Schluß des Pakets stehen, realisiert.

Da bisher noch keine LAMBDA – oder LABEL – Ausdrücke verarbeitet wurden, übergibt "evalquote" die leere Bindeliste.

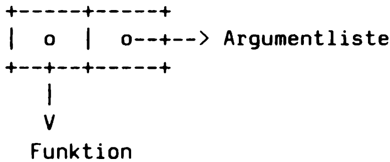
Wirkungsweise von

SYM PROC eval (SYM CONST expression, association list):

"eval" kann als erstes Argument ein Atom oder eine zusammengesetzte Struktur erhalten.

Atome werden als Variable aufgefaßt, deren Wert in der Bindeliste aufzusuchen ist. Vor der Konsultation der Bindeliste wird allerdings noch nach der Eigenschaft APVAL des Atoms gesehen; wenn sie vorhanden ist, handelt es sich um eine Konstante wie NIL, T oder F, die einen festen Wert hat, nämlich den Wert dieser Eigenschaft. Da diese Konstanten sich selbst als Wert haben, gilt "eval (NIL, Bindeliste) = NIL" (entsprechend für "T" und "F").

Wenn das erste Arugment von "eval" zusammengesetzt ist, wird angenommen, daß es sich um einen Funktionsaufruf der Form

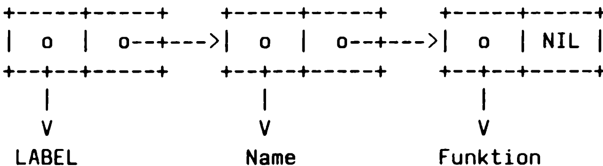


handelt. Die Bestandteile "Funktion" und "Argumentliste" werden mit der Bindeliste übergeben an:

SYM PROC apply (SYM CONST function, arguments, association list):

"apply" hat die Aufgabe, die Argumente durch "eval" berechnen zu lassen (das unterbleibt allerdings unter bestimmten Umständen) und die Berechnungergebnisse an die Parameter der Funktion zu binden; zum Schluß muß der Wert des Funktionsrumpfs in Abhängigkeit von diesen neuen Bindungen als Ergebnis der gesamten Prozedur "apply" berechnet werden; diese Berechnung geschieht wieder durch "eval".

Nur in einem LAMBDA – Ausdruck ist direkt bekannt, wo die Parameterliste steht.Solange das nicht der Fall ist, muß entweder ein LABEL – Ausdruck oder ein Atom vorliegen.
Ein LABEL – Ausdruck hat die Form



Da der Name für die Dauer der Auswertung des Funktionsrumpfs an die Funktion gebunden sein muß, wird dis Paar als funktionaler Bindelisteneintrag gespeichert. Funktionale und nichtfunktionale Bindelisteneinträge sind eindeutig unterschieden.

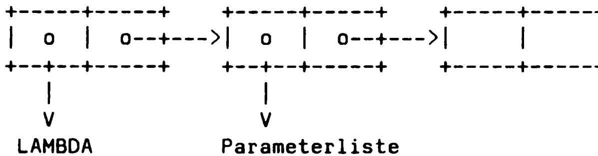
Nach dem Abspeichern wird wieder getestet, ob die Funktion diesmal ein LAMBDA – Ausdruck ist; wenn nicht, wird ein weiterer Schritt zum "Ablättern" von LABELs und Atomen versucht, usw.

Wenn die Funktion ein Atom ist, werden analog zu den Vorgängen in "eval" erst die Eigenschaftsliste und dann die Bindeliste durchsucht.

Ist die Eigenschaft FUNCTION in der Eigenschaftsliste vorhanden, ist der Wert der Eigenschaft die (evtl. weiter "abzublätternde") Funktion; ist die Eigenschaft nicht vorhanden, muß das Atom an eine Funktion gebunden sein, die dann aus der Bindeliste geholt werden kann.

Da alle Funktionen (auch die Standardfunktionen) letztendlich als LAMBDA-Ausdrücke definiert sind, kommt "apply" auf diese Weise zuletzt zu einem LAMBDA-Ausdruck.

Ein LAMBDA-Ausdruck hat die Form



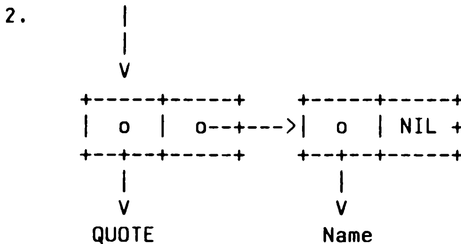
Als nächster Schritt werden die Argumente für die zu berechnende Funktion an die Parameter der Parameterliste gebunden, d.h. es werden Parameter-Argument-Paare in die Bindeliste eingetragen.

Die Methode des Eintrags ist je nach Art des Parameters unterschiedlich. Es gibt die folgenden Arten von Parametern:

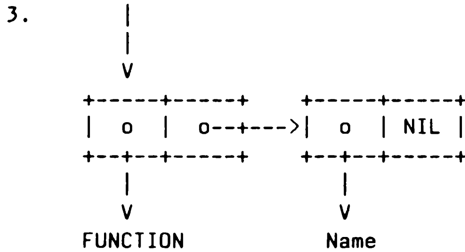
1.

"Name" ist hier – wie bei den restlichen Fällen – der Name des Parameters. Diese Art von Parametern ist der Normalfall; die Argumente, die einem solchen Parameter entsprechen, werden durch "eval" berechnet und zusammen mit dem Parameter in einem Bindelisteneintrag gespeichert.

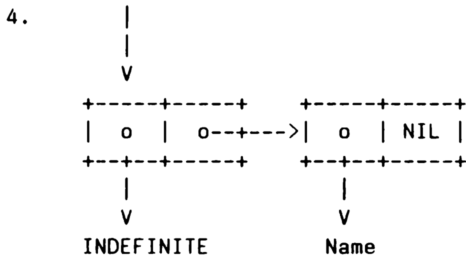
2.



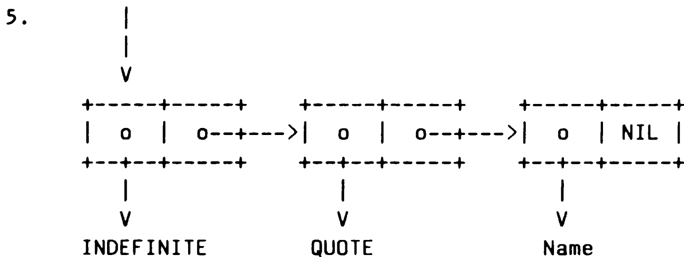
In diesem Fall wird das Argument ohne weitere Verarbeitung in die Bindeliste übernommen. Die Wirkung ist die gleiche, als wäre das Argument durch "(QUOTE ...)" eingeschlossen.



Hier wird ein funktionaler Bindelisteneintrag erzeugt, so daß "Name" im Funktionsrumpf als Name einer Funktion auftreten kann.



Dies ist ein Parameter, der beliebig viele berechnete Argumente aufnehmen kann. Der Einfachheit halber werden die Ergebnisse zu einer Liste zusammengefaßt und mit "Name" in einen Bindelisteneintrag gesteckt.



Dieser Parameter kann wie der in Fall 4. aufgeführte beliebig viele Argumente aufnehmen, die zu einer Liste zusammengefaßt werden. Im Gegensatz zu 4. wird aber wie bei 2. nichts durch "eval" berechnet, sondern die Argumente so wie sie vorkommen übernommen.

Auf einen Parameter der Form 4. oder 5. darf kein weiterer Parameter folgen, weil solch ein Parameter alle restlichen Argumente verbraucht. Solchen Parametern darf – als Ausnahme – auch kein Argument entsprechen; dann werden sie an die leere Liste (d.h. NIL) gebunden.

Der letzte Kasten in der Beschreibung des LAMBDA – Ausdrucks ist mit Absicht leer geblieben; er kann eine der Formen

```
+-----+-----+                +----------+----------+
|  o  | NIL |       oder     | Ganzzahl | XXXXXXXX |
+--+--+-----+                +----------+----------+
   |
   V
 Funktionsrumpf
```

annehmen.

Die erste Form heißt, daß die Funktion durch Berechnung des Funktionsrumpfs mittels "eval" berechnet werden soll; die zweite Form bewirkt den Aufruf einer der Standardfunktionen, je nachdem, welche Funktionsnummer bei "Ganzzahl" steht. In diesem zweiten Fall werden die Argumente aber nicht durch den Namen des Parameters identifiziert, sondern durch die Position des Eintrags in der Bindeliste. Dieser Programmteil hängt also wesentlich von der Reihenfolge ab, in der die Bindelisteneinträge, die bei der Parameter – Argument – Zuordnung entstehen, in die Bindeliste eingefügt werden. Zur Zeit ist das die Umkehrung der Reihenfolge der Parameter.

Die Namen der Refinements "arg 1", "arg 2", "arg 3" beziehen sich auch nicht auf die Position des Arguments in der Argumentsliste, sondern auf die Position des Eintrags in der Bindeliste.

4.4 Kommandoprozeduren

Die Datei "lisp.4" enthält eine Reihe von Prozeduren, mit denen der LISP – Interpreter ähnlich wie der ELAN – Compiler aufgerufen werden kann.

Die Prozedur

start lisp system

ermöglicht das erneute Starten des LISP – Systems, oder wenn "übersetzte" Programme, die in einem Heap einer anderen Task liegen, in dieser Task verarbeitet werden sollen.

Die Prozedur

lisp

stellt die LISP – Maschine in einem Doppelfenster im Bildschirmdialog zur Verfügung. Bei der erstmaligen Benutzung muß die Datei "lisp.bootstrap" vorhanden sein.

Die Prozedur

break lisp

koppelt die LISP – Task vom Benutzer – Terminal ab und baut das Doppelfenster für den Bildschirmdialog neu auf.

Die Prozedur

run lisp

bewirkt, daß ein LISP – Programm eingelesen und ausgeführt wird; nach der Ausführung wird das Ergebnis der Berechnung ausgegeben. Diese Operationen werden auf einer Kopie des Heaps ausgeführt, so daß Änderungen keine Dauerwirkung haben. Mit

run lisp again

wird das zuletzt eingelesene Programm noch einmal gestartet; da dafür die gleiche

Kopie des Heaps wie bei "run" benutzt wird, kann das Ergebnis diesmal anders sein.

insert lisp

wirkt wie "run lisp", außer daß diesmal alle Änderungen, die durch das Einlesen und Ausführen im Heap entstehen, dauerhaft sind.

PROC start lisp system (DATASPACE CONST heap):
Eine Kopie von "heap" wird der neue LISP – Heap. Wenn es sich um "nilspace" handelt, werden einige organisatorische Strukturen im Heap aufgebaut und die Atome "NIL" und "PNAME" erzeugt.

PROC start lisp system (DATASPACE CONST heap, FILE VAR f):
Zunächst wird "start lisp system (heap)" gegeben.
Danach werden die Eigenschaftsbeschreibungen aus "f" in Strukturen im Heap umgesetzt.

Jede Beschreibung in "f" muß mit dem Zeilenanfang beginnen und kann sich über mehrere Zeilen erstrecken. Jede Beschreibung besteht aus den Elementen
<Atom> <Eigenschaft> <Wert>
wobei <Eigenschaft> der Name einer Eigenschaft (i.a. APVAL oder FUNCTION) und <Wert> ein beliebiger S – Ausdruck sein müssen. Die drei Elemente müssen jeweils durch mindestens ein Leerzeichen getrennt sein.

Wenn das Atom <Atom> nicht existiert, wird es erzeugt; danach wird <Wert> unter <Eigenschaft> in der Eigenschaftsliste eingetragen.

Wenn <Eigenschaft> NIL ist, muß <Wert> wegfallen; dann wird nichts in die Eigenschaftsliste eingetragen.

DATASPACE PROC lisp heap:
Liefert den LISP – Heap. Das ist manchmal für Sicherheitskopien etc. nützlich. Die durch "run lisp" erzeugten Kopien sind nicht zugänglich.

PROC run lisp:
Ruft "run lisp (last param)" auf.

PROC run lisp (TEXT CONST file name):
Das in der Datei "file name" stehende LISP – Programm (d.h. der dort stehende in einen S – Ausdruck übersetzte M – Ausdruck) wird in eine neue Kopie des LISP – Heaps eingelesen und ausgeführt. Evtl. vorher durch "run lisp" erzeugte Kopien des Heaps werden vorher gelöscht.

Wenn das Programm syntaktisch nicht korrekt ist, wird es im Paralleleditor zur Korrektur angeboten.

PROC run lisp again:
Führt das zuletzt eingelesene Programm noch einmal im gleichen Heap aus.

PROC insert lisp:
Ruft "insert lisp (last param)" auf.

PROC insert lisp (TEXT CONST file name):
Wirkt wie "run lisp (file name)", nur daß alle Operationen auf dem Originalheap ausgeführt werden. Auch "run lisp again" wirkt nun nicht mehr auf der Kopie.