de Gruyter Lehrbuch
Hamann · Programmieren in LISP

Christian-Michael Hamann

Einführung in das

# Programmieren in LISP

2., bearbeitete und erweiterte Auflage

Mit einem Anhang
„LISP-Dialekte für Personal-Computer"

Walter de Gruyter · Berlin · New York 1985

*Dr.-Ing. Christian-Michael Hamann*
Klinikum Steglitz
Freie Universität Berlin
—Medizinische Informatik/GKZ—
Hindenburgdamm 30
D-1000 Berlin 45

Das Buch enthält 63 Abbildungen

*Für Bärbel, Sven und Niels*

CIP-Kurztitelaufnahme der Deutschen Bibliothek

**Hamann, Christian-Michael:**
Einführung in das Programmieren in LISP/Christian-Michael Hamann. Mit e. Anh. „LISP-Dialekte für Personal-Computer". — 2., bearb. u. erw. Aufl. — Berlin; New York: de Gruyter, 1985. (De-Gruyter-Lehrbuch) ISBN 3 11 010325 7 NE: Beigef. Werk


Printed in Germany. Druck: Karl Gerike, Berlin. — Bindearbeiten: Dieter Mikolai, Berlin.

## Geleitwort
zur 1. Auflage

In letzter Zeit ist eine deutliche Zunahme der Bedeutung von LISP festzustellen, verursacht durch das wachsende Interesse an Problemen der "Artificial Intelligence" und gefördert durch die Entwicklung der Computertechnik, die LISP-Dialekte für Mikrorechner und sogar LISP-Maschinen ermöglicht.

LISP kann als "höhere Assemblersprache" bezeichnet werden : Sie verfügt über die Mächtigkeit einer problemorientierten Sprache, erlaubt jedoch gleichzeitig die Freiheit der Manipulation von Daten und Programmen wie in einer Maschinensprache. Hiermit verbunden ist allerdings auch der Zwang zur Aufmerksamkeit im Detail sowie eine gewisse Sprödigkeit im Zugang.

"Eines der wirklichen Wunder von LISP besteht darin, daß es diese Sprache erlaubt, mit Ideen zu arbeiten" (J. Allen, BYTE, Aug. 1979). In diesem Sinne ist LISP nicht eine weitere Programmiersprache neben vielen anderen, und ein LISP-Lehrbuch kann nicht aus einer Aneinanderreihung von Vorschriften oder gar Rezepten bestehen. Vielmehr muß dieser Prozeß des Umsetzens von Ideen an ausführlich kommentierten Beispielen geübt werden ; der hier vorgelegte Text soll Begleiter sowie Ratgeber auch im Detail sein.

Hervorgegangen ist dieses erste deutsch-sprachige LISP-Lehrbuch aus Vorlesungen, die Herr Dr. Hamann an der Freien Universität Berlin hält. Ich hoffe, daß auch der Leser etwas von jenem Engagement des Autors verspürt, das die Hörer seiner Vorlesung so begeistert und mitgerissen hat.

Berlin - Steglitz , Mai 1982 Peter Koeppe

## Vorbemerkungen zur 2. Auflage

Die freundliche Aufnahme des Lehrbuches, die schon nach verhältnismäßig kurzer Zeit eine Neuauflage notwendig machte, zeigt, daß die gewählte Konzeption zur "Einführung in LISP" Anklang gefunden hat.

Der Kern des Lehrbuches, die Implementierung einer "Block-Welt" (Kap. 3 bis 7), ist nahezu unverändert übernommen worden: Einige Funktionen und Textstellen wurden verbessert, sowie das Merkblatt (2) zum 7. Kap. neu gestaltet.

Darüber hinaus enthält die Neuauflage, neben zahlreichen Verbesserungen und Ergänzungen im Detail, folgende Erweiterungen:

- Definition von Funktionen zur ein- und mehr-dimensionalen Feldverarbeitung (Abs. 9.2)
- Entwurf eines einfachen, modularen GO-Programms als Grundlage zur Entwicklung eines Experimentier-Systems (Kap. 8)
- Erweiterung der "Rekursiven Paraphrasen" (u.a. mit Funktionen für Property-Listen) (Abs. 9.4)
- LISP-Maschinen und LISP-Dialekte für Personal Computer (A-1)
- zusätzlich alphabetisches Verzeichnis der definierten Funktionen (in A-2)
- Aktualisiertes und erweitertes Literaturverzeichnis unter besonderer Berücksichtigung einführender deutsch-sprachiger Veröffentlichungen zu LISP und "Künstlicher Intelligenz" (A-4)

## Vorwort

### (1) Ziel

LISP (list-processing language) wurde bereits 1959 speziell für nicht-numerische Probleme, wie sie z.B. in der "Künstlichen Intelligenz" auftreten, geschaffen und ist die derzeit wichtigste Symbol-Manipulations-Sprache.

LISP unterscheidet sich von den meisten anderen höheren Programmiersprachen dadurch, daß drei wichtige Eigenschaften in einer Sprache vereinigt sind :

- Es können Datenstrukturen beliebiger Tiefe aufgebaut und verändert werden.
- Funktionen können rekursiv definiert werden.
- Es gibt in LISP keinen formalen Unterschied zwischen Daten und Programmen; daher ist es prinzipiell möglich, durch Programme neue Programme zu erzeugen und diese im gleichen Arbeitszyklus zur Ausführung zu bringen (z.B. Selbstoptimierung, Lernen).

Ebensowenig wie eine Fremdsprache durch Lesen einer Grammatik und eines Wörterbuchs zu lernen ist, kann die Programmiersprache LISP mit ihrer charakteristisch ausgeprägten Semantik und ihrem Nuancenreichtum durch das Studium der LISP-Funktionen aus einem Benutzerhandbuch beherrscht werden. Darum wird hier (im Kontext eines größeren und interessanten Problems) durch Entwurf, Implementierung und Test eines vollständigen, exemplarischen Programm-Pakets (der Simulation eines intelligenten Manipulators in einer "Bauklötzchen-Welt" als formalisiertes Befehls- und Frage / Antwort-System) die Programmierung in LISP systematisch geübt.

An Hand der vollständigen und ausführlichen Dialog-Protokolle ist der Leser auch ohne Rechner-Zugriff in der Lage, Zusammenhänge zu erkennen, Zweck und Anwendungsbreite der LISP-Funktionen zu verstehen und insbesondere rekursive Funktionsdefinitionen zu lernen.

Sprachgrundlage ist INTERLISP, einer der wichtigsten LISP-"Dialekte". Das bedeutet jedoch keine Einschränkung, da man sich mit den hier erworbenen Grundlagen-Kenntnissen schnell in andere Dialekte einarbeiten kann.

Das Lehrbuch wendet sich nicht nur an Studenten der Informatik und Mathematik, sondern auch an diejenigen, für die Symbol-Manipulations-Sprachen wichtiges Handwerkszeug ihres Aufgabenkreises und Forschungsgebietes sind, wie z.B. Linguisten, Psychologen u.a.

(2) Inhalt

Das Lehrbuch ist aus der Vorlesungsreihe entstanden, die der Verfasser seit 1981 an der Freien Universität Berlin anbietet. Da sich der didaktische Ansatz bewährt hat, ist Darstellung und Einteilung des Lehrstoffs im wesentlichen beibehalten worden. Die neun Kapitel des Buches sind in drei Teile gegliedert :

| | | |
|---|---|---|
| I | Einführung | (Kap. 1 und 2) |
| II | Implementierung einer "Block-Welt" | (Kap. 3 bis 7) |
| III | Ergänzungen | (Kap. 8 und 9) |

Das Deckblatt zu jedem Kapitel enthält auf der Vorderseite ein ausführliches Inhaltsverzeichnis und auf der Rückseite ein "Merkblatt" zu einem wichtigen Thema dieses Kapitels.

Im ersten Teil wird im 1. Kapitel ein orientierender, hinreichend ausführlicher Überblick zur Sprache und den Sprachelementen von LISP gegeben. Übungsaufgaben mit Lösungen sollen den Lernprozeß unterstützen.

Im 2. Kapitel wird im ersten Abschnitt der Formalismus der Funktionsdefinitionen ausführlich geübt ; im zweiten (mit der Simulation des Puzzles "Die Türme von Hanoi") ein bereits komplexes, aber noch leicht überschaubares "Programm-System" als typische LISP-Lösung präsentiert.

Hier wie in allen folgenden Kapiteln ist jedem Abschnitt das originale Dialog-Protokoll einer Terminal-Sitzung zu-

geordnet : Der Leser schaut gleichsam dem Lehrer am Bildschirm "über die Schulter".

Die einzelnen Benutzereingaben werden vom LISP-System fortlaufend numeriert. Auf diese bezieht sich der zugehörige erklärende Text, der vor oder nach den (eingerahmten) Protokoll-Seiten zu finden ist. Ein Vorteil der Trennung von Protokoll und Text ist die Übersichtlichkeit beim Nachschlagen. Um Zusammenhänge leichter zu erkennen, wurde nach Möglichkeit Zusammengehöriges (z.B. Funktionsdefinition und -Test) auf einer Doppelseite angeordnet.

Da der Verfasser auf die Druckqualität der Protokolle keinen Einfluß hatte, außerdem ein "Zusammenschneiden" notwendig ist, wenn man aussagekräftige und von Eingabefehlern bereinigte Listen erstellen will, mußten nahezu alle per Hand "restauriert" werden. Dabei waren gelegentlich kleinere Mängel nicht zu vermeiden.

Der zweite Teil , die Implementierung einer "Block-Welt", ist das zentrale Thema dieses Lehrbuchs. Zunächst werden die von Winston in seinem Buch "Artificial Intelligence" /Wn77/ angegebenen Funktions-Module in INTERLISP transformiert. Anschließend werden alle notwendigen Strukturen initialisiert, Hilfsfunktionen definiert und mit Hilfe dieser die noch fehlenden Prozeduren für eine arbeitsfähige erste Version der "Block-Welt" formuliert. Schrittweise kommen wir von ganz einfachen zu immer komplexeren Strukturen, bis endlich ein "intelligentes" Frage/Antwort-System geschaffen ist.

Dabei lernt der Leser auf "natürliche" Weise die Sprache LISP kennen, indem folgende Fragen beantwortet werden :

- Welche Systemfunktionen stehen zur Verfügung ?
- In welchem Zusammenhang werden sie benutzt ?
- Was leisten sie ?
- Was benötigen wir darüber hinaus an Funktionen zur Lösung des gestellten Problems ?
- Wie definieren wir diese ?
- Welche Methoden sind anzuwenden ?

Die zu definierenden Funktionen (Namen und Variablen wurden entsprechend der angelsächsischen Systemumgebung gewählt) sind nach didaktischen Gesichtspunkten entworfen. Grundsätzlich wurde "Durchsichtigkeit" angestrebt ; der Verfasser hat dafür in Kauf genommen, daß insbesondere zu Beginn einige Funktionen (z.B. SPACE-LIMITS) wie FORTRAN-Programme in LISP-Schreibweise aussehen.

Der dritte Teil ist (bis auf die Frage-Funktion SAY) vom zweiten Teil unabhängig und kann zu jedem beliebigen Zeitpunkt durchgearbeitet werden. Kapitel 8 enthält (als Herausforderung) den Projekt-Vorschlag zum Entwurf eines GO-Spiel Programms. Im 9. Kapitel werden "destruktive" Systemfunktionen vorgestellt, Funktionen zur Feldverarbeitung definiert, sowie eine Funktion zur Muster-Erkennung (mit Anwendungsbeispielen) programmiert. Insbesondere der letzte Abschnitt dürfte für den Anfänger sofort von Interesse sein : Hier werden "Rekursive Paraphrasen" einiger Systemfunktionen formuliert. Als "Beispiel-Definitionen zum Nachschlagen" sind sie am Ende in alphabetischer Reihenfolge geordnet.

Da es inzwischen brauchbare "LISP-Dialekte für Personal Computer" gibt, werden im Anhang Testberichte und deren Ergebnisse zu diesem Thema vorgestellt. Des weiteren sind (als Index) systematische und alphabetische Übersichten zu den 143 definierten (eigenen) Funktionen und den 88 verwendeten Systemfunktionen angegeben. Das Literaturverzeichnis enthält nur die wichtigsten und aktuellen Quellen (in denen wiederum ausführliche Literaturhinweise zu finden sind) unter besonderer Berücksichtigung deutsch-sprachiger Veröffentlichungen zu LISP und "Künstlicher Intelligenz". Auf der letzten Seite ist die "Situation 0" der "Block-Welt" zur bequemen Referenz angefügt.

SIEMENS-INTERLISP ist ein sehr leistungsfähiges System, das neben dem Interpreter über Compiler, Editor, Unterbrechungs- und Fehlerbehandlungsmöglichkeiten verfügt. Zur

Systemumgebung gehören auch ein "Programmer's Assistant", der als "aktiver" Partner über die Befehle Buch führt und auf Wunsch Befehle und deren "Seiten-Effekte" (mittels UNDO) rückgängig machen kann, sowie die DWIM-("Do-What-I-Mean") Fähigkeit, mit der eine Vielzahl von Eingabe-Fehlern abgefangen und automatisch korrigiert werden kann.

Um den Text möglichst allgemeingültig zu halten, werden wir nicht auf diese Werkzeuge eingehen (allein die Besprechung des Editors belegt im Benutzerhandbuch /Ko81/ 58 Seiten). Von der Menge der verfügbaren Systemfunktionen werden wir nur eine Auswahl kennenlernen. Wir beschränken uns dabei aber nicht nur auf den (theoretisch interessanten) "funktionalen Kern" von LISP, sondern stellen auch die "iterativen" und "destruktiven" Funktionen vor, die zum Schreiben leistungsfähiger Programme für die Praxis benötigt werden.

Programmieren in LISP macht Spaß ! Trotzdem will dieses einführende Lehrbuch "erarbeitet" werden. Es soll den Leser auch an die englisch-sprachige und weiterführende LISP-Literatur heranführen, vor allem aber ihn in die Lage versetzen, eigene Probleme mit Hilfe des Benutzerhandbuches seines ihm zugänglichen LISP-Systems formulieren zu können.

Konstruktive Kritik, Hinweise auf Fehler und Verbesserungsvorschläge sind dem Verfasser stets willkommen !

(3) Danksagung

Das vom Verfasser benutzte LISP - SIEMENS-INTERLISP (V. 4.0) - wurde von der Fa. Siemens zu Testzwecken zur Verfügung gestellt. Dieses LISP ist Nachfolger der Version 3 ; es läuft auf dem Siemens-Rechner 7.541 des Dialogsystem Süd (DSS) der Freien Universität Berlin (FUB) unter dem Betriebssystem BS 2000 (V. 7.1)

Der Zugriff erfolgt mittels Display-Terminal mit Akustik-Koppler über Wählleitung und mittels 8160-Terminal über Standleitung.

Für die großzügige Unterstützung durch Bereitstellung von Betriebsmitteln sei dem DSS, Herrn J. Reker und seinen Mitarbeitern, sowie der Zentraleinrichtung für Datenverarbeitung (ZEDAT) an der FUB, Herrn A. Giedke und seinen Mitarbeitern Dank ausgesprochen.

Für wertvolle Hilfe und Anregungen ist den Herren D. Kolb und Dr. K. März von der Siemens AG München zu danken.

Zahlreiche Verbesserungsvorschläge kamen aus den Reihen der Studenten. Stellvertretend möchte ich hier Herrn F. Wunderlich nennen und Dank sagen.

Für die Durchsicht der ersten Version des Manuskripts und viele nützliche Hinweise sei den Herren Prof. Dr. W. Brecht von der Technischen Fachhochschule Berlin und H.B. v.Schweinichen von der FUB herzlich gedankt.

Besonderer Dank gilt Herrn Prof. Dr.-Ing. P. Koeppe : Dieser hat vom ersten Entwurf des Vorlesungsskripts bis zum fertigen Buch alle Phasen der Entwicklung miterlebt, als kritischer Gesprächspartner und geduldiger Leser wesentliche Anregungen gegeben und das Entstehen dieses Buches im Rahmen des Forschungsvorhabens "Maschinelle Intelligenz" (MAIN - Projekt) sehr gefördert.

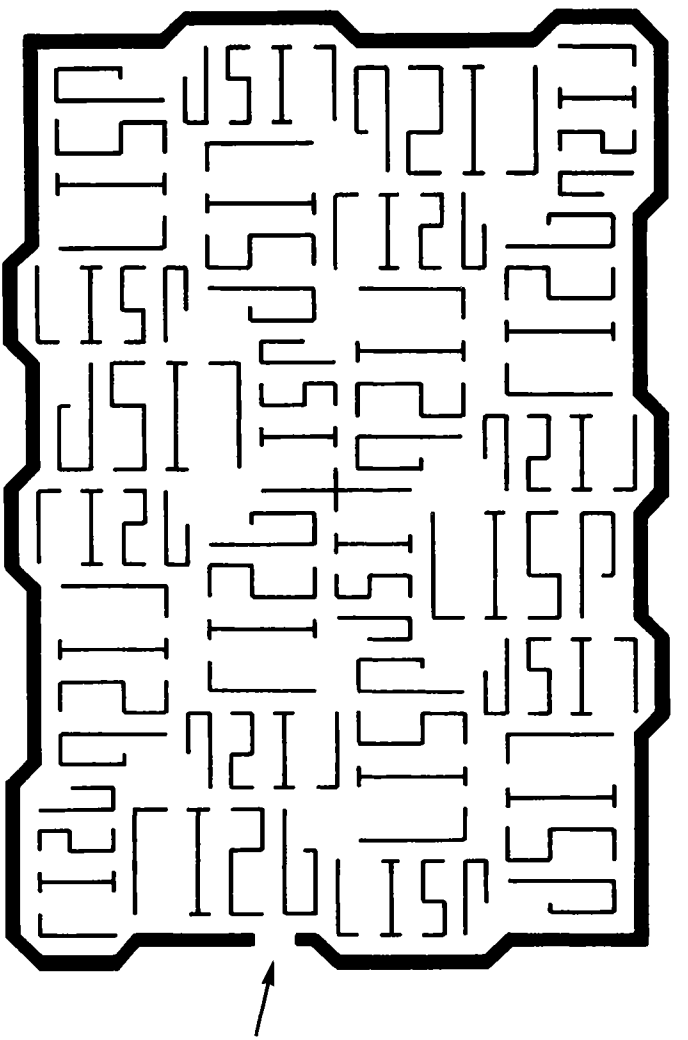
Nicht zuletzt muß die angenehme Zusammenarbeit mit dem Verlag de Gruyter erwähnt werden : Der Verfasser ist Herrn W. Schuder und seinen Mitarbeitern zu Dank verpflichtet für das ihm entgegengebrachte Vertrauen und die gewährte Freiheit zur Gestaltung des Buches.

Berlin-Steglitz, August 1984 Christian-M. Hamann

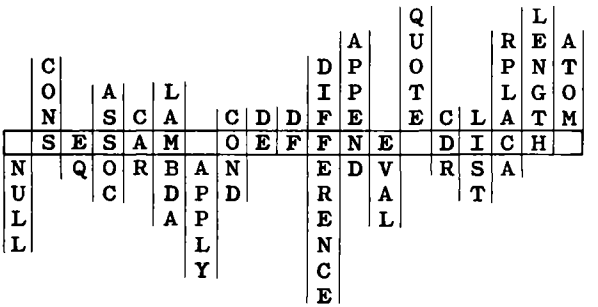
## Inhaltsübersicht

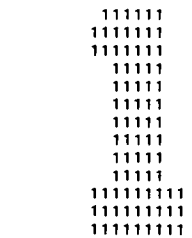
---

Jedes Deckblatt eines Kapitels enthält ausführliches Inhaltsverzeichnis



# EINFÜHRUNG


CONSEQ
ASSOC
CAR
LAMBDA
APPLY
COND
DEF
DIFFERENCE
APPEND
EVAL
QUOTE
CDR
LIST
REPLACA
LENGTH
ATOM
NULL



I N H A L T

M E R K B L A T T  zum 1. Kapitel

Für den Zugriff zum SIEMENS-INTERLISP System ist folgende Hierarchie zu durchlaufen:

Organisatorisches zum Rechenzentrum
(Zuteilung einer Benutzer-Nummer)

**Zugang zu einem Terminal**

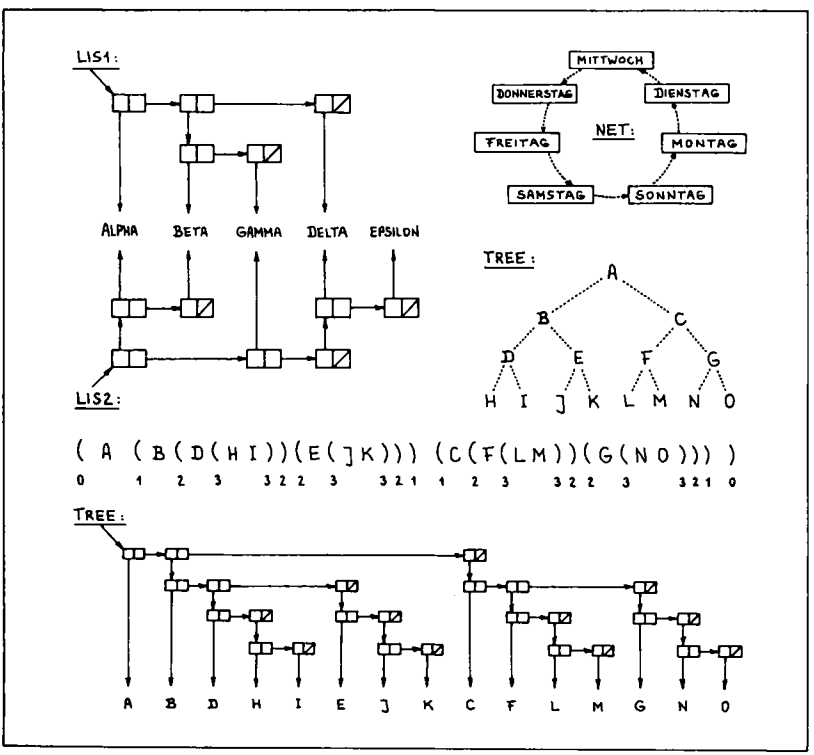
LOGON / LOGOFF - Prozedur
(Kommando-Ebene des BS 2000)

/DO $RZP.LISP

(Zugriff und Verlassen des SIEMENS-INTERLISP Systems)

(EXIT)

Strukturen zu den Übungsaufgaben und Lösungen



## 1.1 Einführende Übersicht

### 1.1.1 Einleitung

Allgemein bekannte Aufgaben für Computer sind Lösungen arithmetischer Problemstellungen. Höhere Programmiersprachen für diese technisch-wissenschaftlichen Zwecke sind z.B. FORTRAN und ALGOL.

Für kaufmännisch-verwaltungstechnische Belange ist die Textverarbeitung z.B. zum Erstellen von Lagerlisten, Rechnungen, Gehaltsnachweisen usw. von Bedeutung. Für diese Problem-Klasse wurde die höhere Programmiersprache COBOL entworfen.

Inzwischen gibt es für nahezu jeden Schwerpunkt des Computer-Einsatzes (z.B. Prozeßdatenverarbeitung in Echt-Zeit) adäquate höhere Programmiersprachen (z.B. PEARL). Moderne Sprach-Entwicklungen wie z.B. ADA sollen möglichst alle Anwendungsbereiche abdecken, um so eine Vereinheitlichung und Kosten-Reduzierung bei Programm-Entwicklung und -Wartung zu erreichen.

Im Gegensatz zu diesen üblichen Aufgaben für Computer, bei denen eindeutige, eine Lösung garantierende Algorithmen angebbar sind, gibt es Probleme meist nicht-numerischer Art, für deren Lösung kein Rezept bereit liegt. Hier helfen z.B. "heuristische" Verfahren, also Regeln, die sich beim Problemlösen bewähren, auch wenn keine strenge mathematische Begründung möglich ist oder eine Lösung (falls sie existiert) nicht immer garantiert werden kann.

Derartige Probleme lassen sich oft in Baum-Struktur (vgl. D.E. Knuth /Kn72/ ) darstellen, wie z.B. bei Spielen mit ihren

Zügen und Gegenzügen. Selbstverständlich müßte ein mathematisches Programm theoretisch den jeweils besten Zug berechnen können, jedoch würde es bei komplexen Spielen (wie z.B. Schach oder Dame) Jahrtausende dauern. Hier hilft nur eine Heuristik , die Züge und Gegenzüge bis in eine gewisse Tiefe verfolgt und dann (wie der Mensch) eine Auswahl trifft nach Kriterien der "Feldbeherrschung", "Bedrohung" usw. Ein erfolgreiches Beispiel dieser Methode ist das Dame-Spiel-Programm von A.L. Samuel (1959), das inzwischen (dank seiner Struktur) gelernt hat, besser zu spielen als ein menschlicher Gegner. (Samuels Aufsatz ist, zusammen mit anderen interessanten Beiträgen, in dem historischen Standardwerk /FF63/ zu finden.)

Betrachten wir ein weiteres Beispiel zur Illustration dieser sog. "symbolischen" Datenverarbeitung: Für einen Schrift-Erkennungs-Automaten, der auch ein schlecht gedrucktes "A" als "A" erkennen soll, ist die Programmierung einer (statistischen) Vergleichs-Methode mit der Schablone des "richtigen A" auch bei extremem Aufwand wenig brauchbar. Besser ist eine "semantische Analyse", bei der das Programm als "A" einen Buchstaben erkennt, der z.B. so beschrieben ist: "Zwei etwa parallele, etwa senkrechte Linien, die unten auseinander stehen und sich oben berühren und ungefähr in der Mitte eine waagerechte Verbindung haben". Ein solches Programm ist selbstadaptierbar und liefert auch dann brauchbare Ergebnisse, wenn sich die Eingabedaten stark geändert haben.

Das Spiel-Programm und das Schrift-Erkennungs-Programm sind Beispiele für ein wichtiges Anwendungsgebiet der nicht-numerischen Datenverarbeitung, das als "Maschinelle Intelligenz" oder "Künstliche Intelligenz" (Artificial Intelligence, AI) oder auch (nach einem Vorschlag von W. Bibel) als "Intellektik" bezeichnet wird.

AI beschäftigt sich mit Problemen, die - wenn sie von Menschen gelöst werden - "Intelligenz" erfordern. Wichtige Teilgebiete der AI-Forschung sind: Allgemeine Methoden zum "Problem-Lösen", Muster-Erkennung, mathematisch-logische Beweis-Verfahren, Roboter-Technologie, Natürliche Sprachverarbeitung, Expertensysteme.

Insbesondere "Expertensysteme" (das sind "Wissens-Basen", die mittels Inferenzkomponenten auch nicht-explizit gespeichertes Faktenwissen nach Regeln ableiten und verfügbar machen können) demonstrieren den erfolgreichen Einsatz von AI-Methoden (vgl. z.B. /Wn77/). Beispiele (in LISP) sind:

- Das Mathematik-Programm MACSYMA beherrscht (u.a.) symbolische Differentiation und Integration auf Expertenebene.
- Das Programm-System DENDRAL ist Experte zur Bestimmung der Molekül-Struktur aus den Daten des Massenspektrogramms.
- Das Programm MYCIN unterstützt den Arzt bei Diagnose und Therapie von Infektionskrankheiten.

Da die Systeme ihre Entscheidung begründen, werden sie auch als Tutoren zur Ausbildung eingesetzt. Darüber hinaus sind mit Meta-DENDRAL und EMYCIN generalisierte Versionen entwickelt worden, die aus "diffusem Wissen" Regeln ableiten können. Mit Hilfe dieser "Shells" sind Expertensysteme für andere Fachgebiete relativ schnell zu implementieren.

Methoden und Ziele der AI sind von allgemeinem Interesse und spielen in der heutigen Datenverarbeitung eine zunehmende Rolle: Die künftigen sehr komplexen Informations-Systeme können nicht mehr mit konventionellen Mitteln der Informatik beherrscht werden.

AI-Programme modellieren entsprechende Theorien. Einerseits sind Programme präzise Experimentier-Werkzeuge, womit Theorien gestützt oder verworfen oder zur Modifi-

kation gezwungen werden können. Andererseits haben AI-Programme noch weitgehend experimentellen Charakter: Von der Unsicherheit über Ziele und zu benutzende Methoden beim ersten Programm-Entwurf wird über erreichte Zwischenstadien die Programm-Entwicklung durch neu gewonnene Erkenntnisse gesteuert.

Aus dem zuletzt genannten Grund werden an AI-Sprachen andere Anforderungen gestellt als an die konventionellen höheren Programmiersprachen, die für "produktionsreife" Anwenderprobleme konzipiert wurden. (Darum ist z.B. ADA für AI-Applikationen wenig geeignet, wie der SRI-Report /SM80/ feststellt.)

Von den AI-Sprachen der "ersten Generation" (z.B. LISP, SAIL, POP-2, SNOBOL) ist LISP (list-processing language) die wichtigste und weitverbreitetste Symbol-Manipulations-Sprache und grundlegend zum Verständnis darauf aufbauender Sprachen der "zweiten Generation" (z.B. MICRO-PLANNER, CONNIVER, FUZZY).

Die erste Version von LISP wurde am Massachusetts Institute of Technology (MIT) unter J. McCarthy entwickelt: Das "LISP - Programmer's Manual" hat bereits 1959 als handschriftlicher Entwurf vorgelegen. Die Bedeutung von LISP unterstreichen folgende Zitate :

- P.C. Jackson /Ja74/ stellt fest: "Der Wert guter Programmiersprachen für die Entwicklung des AI-Forschungsgebietes kann gar nicht hoch genug eingeschätzt werden. Ohne LISP hätte AI nicht 'vom Boden abheben' können."
- P.H. Winston /Wn77/ formuliert das so: "LISP ist die Mathematik der AI. AI ohne LISP ist wie Physik für Poeten - laudabel und nützlich, aber nicht ganz ernst zu nehmen."
- L. Siklóssy /Si76/ zitiert J.E.Sammet (/Sa69/): "Programmiersprachen können in zwei Kategorien eingeteilt werden : In der einen Kategorie ist LISP und in der anderen sind alle anderen Programmiersprachen."

LISP unterscheidet sich von den meisten anderen höheren Programmiersprachen dadurch, daß drei wichtige Eigenschaften in einer Sprache vereinigt sind :

- Es können Daten-Strukturen beliebiger Tiefe (z.B. als binäre Bäume) aufgebaut und verändert werden.

  Infolge dynamischer Speicherverwaltung stößt ein eskalierendes Anwenderprogramm nicht an Grenzen vorgegebener Feldgrößen.

- Funktionen können rekursiv definiert werden.

  Bekanntes Beispiel einer rekursiven Definition ist die Fakultätsfunktion :

  $n! = n * (n-1)!$ für $n \geq 0$ mit $0! = 1$ per def.

- Es gibt in LISP keinen formalen Unterschied zwischen Daten und Programmen.

  Daher ist es möglich, durch Programme neue Programme zu erzeugen, oder Programme sich selbst verändern zu lassen und diese im gleichen Arbeitszyklus zur Ausführung zu bringen. Somit ist prinzipiell die Möglichkeit gegeben zur Strukturveränderung des Programms durch Selbstoptimierung, Selbstprogrammierung, Selbstlernen.

Eine weitere nützliche Eigenschaft :

- LISP ist eine typfreie Sprache

  Die Datentypen müssen nicht vom Programmierer explizit deklariert werden; welcher Datentyp jeweils vorliegt wird vom System selbständig erkannt.

Das Erlernen von LISP wird durch eine extrem einfache Syntax erleichtert. Kenntnisse anderer Programmiersprachen werden nicht benötigt ; im Gegenteil : "Vorkenntnisse sind oft ein Handicap" (Winston /Wn77/).

---

Die "Einführende Übersicht" ist die überarbeitete und erweiterte Fassung des Artikels :

Hamann, C.-M.
"Die Programmiersprache LISP - Eine einführende Übersicht"
Elektronik, 1(1982)99-102, 2(1982)60-64

(Mit freundlicher Genehmigung des Franzis Verlag, München)

## 1.1.2 Darstellung von Listenstrukturen

### 1.1.2.1 Syntaktische Darstellung

Daten und Programme in LISP werden als "S-Expression" (symbolischer Ausdruck) geschrieben. Eine S-Expression ist z.B. ein "Atom" oder eine "Liste".

Ein Atom ist entweder ein literales Atom oder eine Zahl, z.B.:

| | | |
|---|---|---|
| A | = | literales Atom |
| BANANE | = | literales Atom |
| 1ETWASSELTSAMESATOM | = | literales Atom, da keine Zahl |
| 2 | = | ganze Zahl (Integer) |
| 3.1416 | = | dezimale Zahl (Real) |

Als Liste kann beispielsweise der Inhalt einer Schüssel beschrieben werden:

(APFEL BANANE CITRONE)

oder "abstrakt":

(A B C)

Eine Liste ist entweder leer oder enthält Atome und/oder Sublisten, z.B.:

| | | |
|---|---|---|
| ( ) ≡ NIL | = | leere Liste, ist mit dem Atom NIL identisch ! |
| (D(E F)G) | = | Liste mit Atom D, der Subliste (E F) und dem Atom G |

Listenelemente sind in Klammern eingeschlossen. Eine hierarchische Ordnung wird durch die Einbettung von Sublisten ausgedrückt.

Durch Numerierung der korrespondierenden Klammer-Paare (per Hand !) ist die "Schachtel-Tiefe" leicht zu ersehen, z.B.:

```
( A ( X Y ) ( D ( E F ) G ) H )
0   1     1 1   2     2   1   0
```

Diejenigen Elemente einer Liste, die durch das 0-Klammer-Paar eingeschlossen sind, heißen "top-level" Elemente. In dieser Liste sind das:

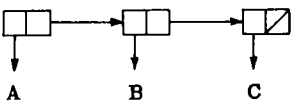
| | |
|---|---|
| A | das Atom A |
| (X Y) | als Subliste |
| (D(E F)G) | als Subliste |
| H | das Atom H |

Der Numerierungs-Modus wird im folgenden Abschnitt deutlich.

### 1.1.2.2 Interne Darstellung

Man kann sich ein Listenelement als aus einer Doppel-Zelle bestehend vorstellen, die zwei Zeiger (pointer) enthält. Der linke Zeiger verweist auf den "Inhalt", der rechte Zeiger verweist auf das nächste Listenelement. Das Listenende ist durch den Zeiger auf das Atom NIL gegeben.

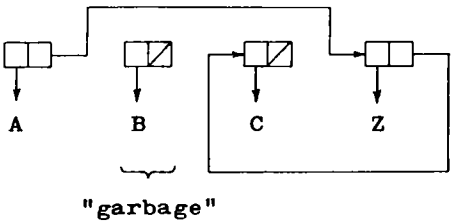
Beispiel einer linearen Liste, z.B. (A B C):




Statt des NIL-Zeigers zeichnet man der besseren Übersicht wegen in die letzte Zelle einen diagonalen Strich.

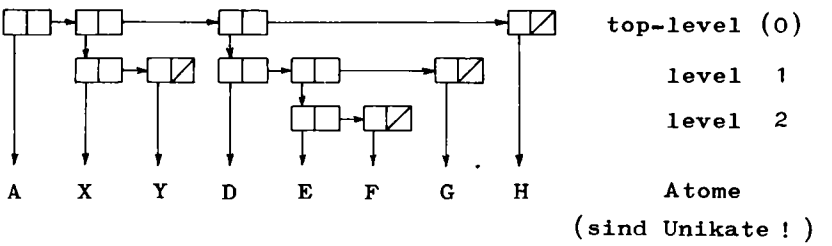
Wie leicht zu erkennen ist, können Änderungen in der Listenstruktur einfach durch Änderung der Zeiger vollzogen werden. Aus der Liste (A B C) wird z.B. die Liste

(A Z C), wobei die auf B zeigende Zelle wieder dem System zur freien Verfügung zurückgegeben werden kann ("garbage collection" = "Müll-Sammlung") :




Beispiel einer hierarchischen Liste (vgl. Beispiel des letzten Abschnitts)

( A ( X Y ) ( D ( E F ) G ) H ) :




## 1.1.3 Generelles zu LISP-Funktionen

Der hier vorgestellte moderne LISP-"Dialekt" (genaueres dazu in Abs. 1.1.15) heißt INTERLISP. Der Benutzer arbeitet interaktiv ( daher der Name ) mit dem LISP - Interpreter und initialisiert die normale Arbeitsschleife "Einlesen - Auswerten - Drucken" (read-eval-print).

In einer (zu evaluierenden) Liste wird das erste Atom als Funktionsname angesehen, die restlichen Elemente als deren Argumente. Beispiele für einige arithmetische Funktionen verdeutlichen das Prinzip :

```
(PLUS 5 3)                    →   8
(TIMES 5 3 2)                 →  30
(DIFFERENCE 5 3)              →   2
(MINUS 5)                     →  -5
(ADD1 5)                      →   6
(SUB1 5)                      →   4
```

Es handelt sich hier um Präfix-Notation, die vom polnischen Logiker J. Lukasiewicz (1925) begründet wurde. (Die Schul-Algebra benutzt Infix-, ein HP-Taschenrechner Postfix-Notation, auch "umgekehrte polnische Notation" = UPN genannt.) Hervorzuheben ist, daß es in LISP Funktionen gibt (wie z.B. PLUS und TIMES), die beliebig viele Argumente akzeptieren!

Schachtelungen werden "von innen nach außen" abgearbeitet (evaluiert), z.B.:

```
(TIMES(PLUS 5(ADD1 3))(MINUS 2))      →  -18
```

Mit der Funktion SETQ kann eine Wertbindung an eine Variable bewirkt werden, z.B. A=5 , B=3 , C=2 :

```
(SETQ A 5)
(SETQ B 3)
(SETQ C 2)
```

Wird jetzt A oder B eingegeben, so wird der gebundene Wert zurückgegeben. Man kann in weiteren Anweisungen darauf zurückgreifen, z.B.:

```
A                             →   5
(PLUS A B)                    →   8
```

Soll eine Variable als Wert eine Liste zugewiesen bekommen, so muß der Evaluierungs-Prozeß ("erstes Atom ist Funktionsname") außer Kraft gesetzt werden. Das geschieht mittels Hochkomma (quote-mark), z.B.:

```
(SETQ LISTE1 '(A B C) )
```

Wird jetzt der Name des Atoms LISTE1 eingegeben, so wird als der daran gebundene Wert, die Liste zurückgegeben:

```
LISTE1                                  →   (A B C)
```

Das Hochkomma ist eine vom Rechner akzeptierte Kurz-Schreibweise der Funktion QUOTE, die ihr Argument nicht evaluiert. Die oben angegebene Zuweisung wäre in ausführlicher Schreibweise:

```
(SETQ LISTE1 (QUOTE (A B C)))
```

Die hier offensichtlich lästige und fehleranfällige Klammerzählung kann stets durch "Superklammern" vereinfacht werden. Dem obigen Beispiel äquivalent ist:

```
(SETQ LISTE1 '(A B C>
```

Anmerkung: Zahlen definieren sich selbst, evaluieren darum zum gleichen Wert. Weiterhin folgt wegen des quote-mark, daß das zweite Argument von SETQ auch eine beliebige S-Expression sein kann, deren Wert (nach Auswertung) an das erste Argument gebunden wird. Mit dem Beispiel von oben A=5 , B=3 ergibt sich:

```
(SETQ SUM (PLUS A B))
SUM                                     →   8
```

Man beachte also folgenden Unterschied:

```
(SETQ SUM '(PLUS A B))
SUM                                     →   (PLUS A B)
```

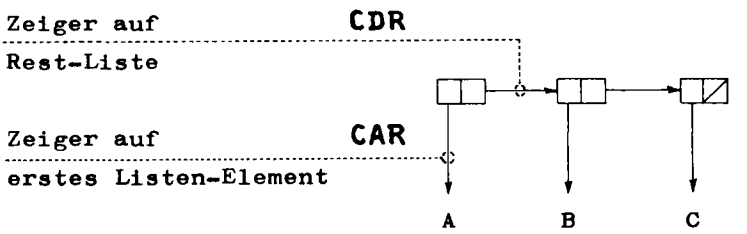
Es ist bemerkenswert, daß die hier als "Datum" an SUM gebundene Liste auch als "Programm" interpretierbar ist ! Mit der Funktion EVAL kann dieses zur Ausführung gebracht werden :

```
(EVAL SUM)                              →   8
```

Dieses einfache Beispiel zeigt die in der Einleitung erwähnte Ambivalenz von Daten und Programmen in LISP .

### 1.1.4 Elementarfunktionen zur Listenverarbeitung

Betrachten wir unsere bekannte LISTE1 als Zeiger-Diagramm:




Die Funktion CAR liefert das erste Element einer Liste, die Funktion CDR die Restliste, z.B.:

```
(CAR '(A B C))          →  A
(CDR '(A B C))          →  (B C)
```

oder:

```
(CAR LISTE1)            →  A
(CDR LISTE1)            →  (B C)
```

Die merkwürdigen, nicht-mnemotechnischen Funktionsnamen CAR (gesprochen: kahr) und CDR (gesprochen: kuder) haben ihre historische Ursache in der Wortstruktur der IBM 704 : "contents of address part of register" und "contents of decrement part of register" .

Bei wiederholter Verwendung der Funktionsnamen ("von innen nach außen" anzuwenden !)

```
(CAR(CDR '(A B C>       →  B
(CAR(CDR(CDR '(A B C>   →  C
```

sind die zusammengesetzten Funktionsnamen praktisch, wobei zwischen C ... R ein A für CAR, ein D für CDR steht:

```
(CADR '(A B C))                          →   B
(CADDR '(A B C))                         →   C
(CAADDR '(A(X Y)(D(E F)G)H) )            →   D
```

Mit den Funktionen CAR und CDR können Listen durchsucht, mit der Funktion CONS (construct) kann eine Liste gebildet oder erweitert werden, z.B.:

```
(CONS 'A NIL)                            →   (A)
(CONS 'A '(B C))                         →   (A B C)
```

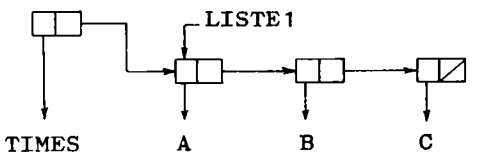
Man beachte die Äquivalenz :

```
(CONS(CAR '(A B C))(CDR '(A B C)))       →   (A B C)
```

Wir können auch hier die Ambivalenz von "Daten" und "Programm" demonstrieren. Im folgenden Beispiel sei wieder LISTE1 = (A B C) mit A=5 , B=3 , C=2 :

```
(CONS 'TIMES LISTE1)                     →   (TIMES A B C)
(EVAL(CONS 'TIMES LISTE1))               →    30
```

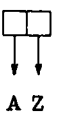
Mit der CONS-Funktion wird eine neue LISP-Zelle erzeugt. Das obere Beispiel als Zeiger-Diagramm dargestellt:




Ist das zweite Argument von CONS keine Liste, sondern ein Atom, so wird das entstehende Gebilde "dotted-pair" (Punkt-Paar) genannt, z.B.:

```
(CONS 'A 'Z)                             →   (A . Z)
```

als Zeiger-Diagramm:




wobei:
```
(CAR '(A . Z))   →   A
(CDR '(A . Z))   →   Z
```

Listen können auch in "dot"-Schreibweise dargestellt werden, z.B.:

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

Es gibt auch höhere Funktionen zur Listenverarbeitung *) , z.B.:

```
(REVERSE '(A B C))                    → (C B A)
(SORT '(5 2 4 1 3))                   → (1 2 3 4 5)
(REMOVE 'B '(A B C))                  → (A C)
(SUBST 'A 'B '(A B C))                → (A A C)
(UNION '(A B C) '(B C D))             → (A B C D)
(INTERSECTION '(A B C) '(B C D))      → (B C)
(NTH '(A B C) 2)                      → (B C)
(LAST '(A B C))                       → (C)
(PACK '(A B C))                       → ABC
(UNPACK 'ABC)                         → (A B C)
```

### 1.1.5 Property-Listen

In LISP können bestimmte Eigenschaften eines Atoms auf einer "Property-Liste" (Eigenschafts-Liste) gespeichert und bei Bedarf abgerufen werden.

Beispielsweise haben Bauklötze eine Farbe (rot, blau, ...), sind von einem bestimmten Typ (Block, Pyramide, Kugel ...), von bestimmter Größe, stehen auf einem bestimmten Platz im dreidimensionalen Koordinatensystem, usw.

Eigenschaften werden abgespeichert mit Hilfe der Funktion PUTPROP. Z.B. seien für den Baustein Block-D die folgenden Eigenschaften zu definieren:

```
(PUTPROP 'BLOCK-D 'FARBE 'ROT)
(PUTPROP 'BLOCK-D 'GROESSE '(2 2 2))
```

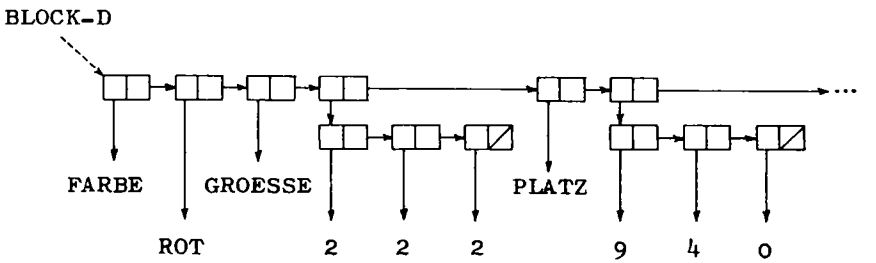
---

*) siehe ggf. Kurzbeschreibungen der Systemfunktionen im Anhang A-3

Die auf der Property-Liste abgelegten Eigenschaften können abgerufen werden mit der Funktion GETPROP . Z.B.:

```
(GETPROP 'BLOCK-D 'FARBE)      → ROT
(GETPROP 'BLOCK-D 'PLATZ)      → (9 4 0)
```

Die vom LISP-System organisierte und verwaltete Eigenschafts-Liste besteht aus einer Reihe von Attribut-Wert-Paaren und hat z.B. folgendes Aussehen:




Mit der Funktion REMPROP können Eigenschaften auch wieder gelöscht werden, z.B.:

```
(REMPROP 'BLOCK-D 'WERKSTOFF)
```

H. Stoyan /St80/ macht darauf aufmerksam, daß in Property-Listen als Eigenschaften auch Programme abgelegt werden können, die etwa die Verarbeitungsweise anderer Eigenschaften beschreiben.

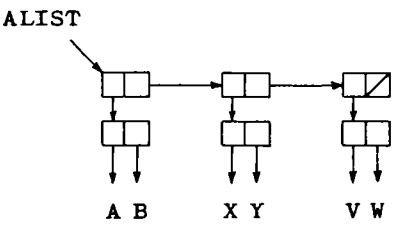
### 1.1.6 Assoziations-Listen

Assoziativ-Speicher sind mächtige organisatorische Werkzeuge. Sie werden in LISP durch "Assoziations-Listen" realisiert, die aus Punkt-Paaren folgender Form bestehen:

```
( (key1 . wert1)(key2 . wert2) ... (keyn . wertn) )
```

wobei key1 ... keyn (z.B. atomare) "Schlüssel", wert1 ... wertn beliebige S-Expressions sind. Es sei z.B. folgende Assoziations-Liste als Wert an das Atom ALIST gebunden:

```
(SETQ ALIST '( (A . B) (X . Y) (V . W) ) )
```




Die Funktion ASSOC liefert das zum Schlüssel keyx gehörige dotted-pair (keyx . wertx) der Assoziations-Liste. Es kann wertx dann z.B. so erhalten werden:

```
(CDR(ASSOC 'X ALIST))                    →    Y
```

Als Wert kann auch hier z.B. ein Programm stehen. D. Kolb /Ko81/ betont, daß neben komfortablen Zugriffsmöglichkeiten der Vorteil der Assoziations-Liste gegenüber der Property-Liste darin besteht, daß die Assoziations-Liste lokal an eine Variable (ein Atom) gebunden werden kann, Property-Listen dagegen stets global sind. Indikatoren von Property-Listen müssen literale Atome sein, während für die Schlüssel der Assoziations-Listen beliebige S-Expressions zugelassen sind.

### 1.1.7 Prädikatfunktionen

In LISP gibt es Test-Funktionen, z.B. die Funktion NULL, die feststellt, ob eine Liste leer ist. Das "Prädikat" ist ein Ergebnis, das entweder den Wahrheitswert "wahr" oder "falsch" hat. "Falsch" wird durch das Atom NIL, "wahr" durch das Atom T (true) ausgedrückt. NIL und T sind spezielle Atome, deren Werte feststehen : Sie evaluieren (wie Zahlen) auf sich selbst.

Einige Beispiele von Prädikatfunktionen im Test mit unserer LISTE1 = (A B C) :

| | | |
|---|---|---|
| (NULL () ) | → | T |
| (NULL LISTE1) | → | NIL |
| (ZEROP 0) | → | T |
| (ATOM(CAR LISTE1)) | → | T |
| (EQUAL 'A(CAR LISTE1)) | → | T |
| (MEMBER 'D LISTE1) | → | NIL |
| (MEMBER 'D '(A B C D E) ) | → | (D E) |

Die Funktion MEMBER im letzten Beispiel liefert statt T die Restliste, deren erstes Element das gesuchte top-level Element ist. Aus Gründen einer effizienten Weiterverarbeitung gilt verallgemeinernd in LISP, daß jeder Wert ungleich NIL den Wahrheitswert "wahr" hat.

### 1.1.8 Die LAMBDA-Funktion

Der $\lambda$-Kalkül von A. Church (1941) wird, wie Turing-Maschine, $\mu$-rekursive Funktionen und ähnlichem, zur Präzisierung des Algorithmen-Begriffs verwendet.

Zur Bearbeitung einer Formel, z.B. $y^x$ mit den Argumenten (2 , 3) ist es notwendig, eine Konvention bez. der Parameter-Bindung zu treffen: Gilt entweder $2^3 = 8$ oder $3^2 = 9$ ? In LISP wird in Anlehnung an die Schreibweise des $\lambda$-Kalküls, womit die zuordnende Paarung der Funktionsparameter mit den Werten der Argumentliste ausgedrückt wird

$$\lambda( (x, y); y^x ) (2 , 3) \quad = 9$$

die Funktion LAMBDA benutzt, um eine (nicht explizit benannte) Funktionsdefinition zu formulieren.

Beispielsweise sei das Volumen $V = \pi r^2 h$ eines Zylinders zu berechnen mit r = 1 , h = 2 :