H. Stoyan G. Görz

LISP

Eine Einführung in die Programmierung

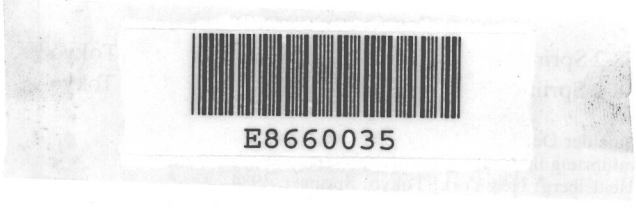
Springer-Verlag Berlin Heidelberg New York Tokyo

Herbert Stoyan Günter Görz

LISP

Eine Einführung in die Programmierung

Mit 29 Abbildungen



Springer-Verlag
Berlin Heidelberg New York Tokyo 1984

Dr. Ing. habil. HERBERT STOYAN
IMMD VI, Universität
Erlangen-Nürnberg
Martensstr. 3, 8520 Erlangen

Dipl.-Math. GÜNTER GÖRZ
RRZE,
Martensstr. 1, 8520 Erlangen

ISBN 3-540-13158-2 Springer-Verlag Berlin Heidelberg New York Tokyo
ISBN 0-387-13158-2 Springer-Verlag New York Heidelberg Berlin Tokyo

CIP-Kurztitelaufnahme der Deutschen Bibliothek
Stoyan, Herbert: Einführung in die Programmiersprache LISP / H. Stoyan;
G. Görz. – Berlin; Heidelberg; New York; Tokyo: Springer, 1984.
ISBN 3-540-13158-2 (Berlin ...)
ISBN 0-387-13158-2 (New York ...)
NE: Görz, Günther:

Datenkonversion: Daten- und Lichtsatz-Service, Würzburg
Druck und Einband: Graphischer Betrieb, Konrad Triltsch, Würzburg
2145/3140-543210

Symbolic Computation

Artificial Intelligence

Editors: **A.Bundy, N.J.Nilsson, J.Siekmann, A.Sloman**

N.J.Nilsson

Principles of Artificial Intelligence

1982. 139 figures. XV, 476 pages
ISBN 3-540-11340-1
(Available in North America through
William Kaufmann, Inc.)

In most of the previous treatments of artificial intelligence, it has been divided into its major areas of application including natural language processing, automatic programming, robotics, machine vision, automatic theorem proving, and intelligent data retrieval systems. The major difficulty with this approach is that these application areas are now so extensive that each could be only superficially treated in a book of this length.
The goal of this book is to describe the fundamental AI ideas that underly many of these applications. The organization of these ideas is not based on the application itself but based on general computational concepts. The book is designed as an introductory text on artificial intelligence. It is assumed that the reader has a background in the fundamentals of computer science; knowledge of a list processing language, such as LISP, would be helpful. At the end of each chapter the reader will find many exercise and citations which should provide interested students with adequate entry points to the most important literature in the field.

Springer-Verlag
Berlin
Heidelberg
New York
Tokyo

Symbolic Computation

Artificial Intelligence

Editors: **L. Bolç, A. Bundy, J. Siekmann, A. Sloman**

The Automation of Reasoning I

Classical Papers on Computational Logic 1957–1966
Editors: **J. Siekmann, G. Wrightson**
1983. XII, 525 pages. ISBN 3-540-12043-2

The Automation of Reasoning II

Classical Papers on Computational Logic 1967–1970
Editors: J. Siekmann, G. Wrightson
1983. XII, 637 pages. ISBN 3-540-12044-0

Logic has emerged as one of the fundamental disciplines of computer science. Computational logic, which continues the tradition of logic in a new technological setting, has led such diverse fields of application as automatic program verification, program synthesis, question answering systems, and deductive data bases as well as logic programming and the 5th generation computer system. These two volumes, the first covering the years 1957–1966 and the second, 1967–1970, contain those papers which shaped and influenced the field of computational logic. They make available the classical works in the field, which in many cases were difficult to obtain or had not previously appeared in English.

M. M. Botvinnik

Computers in Chess

Solving Inexact Search Problems
Translated from the Russian by A. A. Brown
With contributions by numerous experts
1983. 48 figures. XIV, 158 pages. ISBN 3-540-90869-2

Contents: The General Statement. – Methods for Limiting the Search Tree. – The Search for a Solution and Historical Experience. – An Example of the Solution of an Inexact Problem (Chess). – Three Endgame Studies (An Experiment). – The Second World Championship. – Appendix 1: Fields of Play. – Appendix 2: The Positional Estimate and Assignment of Priorities. – Appendix 3: The Endgame Library in Pioneer (Using Historical Experience by the Handbook Method and the Outreach Method). – Appendix 4: An Associate Library of Fragments. – References. – Glossary of Terms. – Index of Notation. – Index.

The Design of Interpreters, Compilers, and Editors for Augmented Transition Networks

Editor: **L. Bolç**
1983. 72 figures. XI, 214 pages. ISBN 3-540-12789-5

Contents: The Planes Interpreter and Compiler for Augmented Transition Network Grammars. – An ATN Programming Environment. – Compiling Augmented Transition Networks into MacLisp. – Towards the Elastic ATN Implementation.

Springer-Verlag
Berlin
Heidelberg
New York
Tokyo

DM 30.38

Vorwort

Das vorliegende Buch ging aus einer Vorlesung hervor, die wir gemeinsam im Sommersemester 1982 an der Universität Erlangen-Nürnberg gehalten haben. Wesentliche Anregungen verdanken die Autoren den Manuskripten der Vorlesung „Structure und Interpretation of Computer Programs" von Abelson, Fano und Sussman (1981, 1982, 1983 als MIT-AI-TR-735 veröffentlicht) am Massachusetts Institute of Technology, sowie den Lehrbüchern „LISP" von Winston und Horn (1981) und „Artificial Intelligence Programming" von Charniak, Riesbeck und McDermott (1980) und dem Vorlesungsmanuskript „LISP — Programming and Proving" von McCarthy und Talcott (1980) von der Stanford University.

Im Mittelpunkt unserer Darstellung steht der Begriff des Programmierstils. Er gestattet eine sinnvolle Klassifikation der Ansätze und Leitlinien, nach denen in LISP programmiert wird und ermöglicht gleichzeitig die Einbeziehung neuer Grundgedanken wie des der Objektorientierung. Wir hoffen, damit eine neuartige Einführung in die Programmierung vorzulegen, die nur noch wenige Gemeinsamkeiten mit traditionellen Darstellungen teilt. So kommt es, daß in diesem Lehrbuch nichts über „interne Repräsentation" von LISP-Objekten (z.B. Speicherstrukturen, automatische Speicherverwaltung, usw.) und über interne Details der Implementation von LISP zu finden ist. Wir haben diese Entscheidung bewußt getroffen, da wir meinen, daß diese Themen Gegenstand eines weiterführenden Textes sein sollten. Der hieran interessierte Leser sei auf Allens „Anatomy of LISP" (1979) verwiesen.

Da die einschlägige Literatur über LISP fast ausschließlich in englischer Sprache vorliegt, haben wir uns bemüht, eine einheitliche deutsche Terminologie einzuführen, geben jedoch die englischen Originalbezeichnungen meist an. Nicht zuletzt, um den Leser auch an die Originalliteratur heranzuführen, tragen die Funktionen und Variablen in den Beispielen englische Namen. Dadurch wird auch eine größere Kohärenz mit den ohnehin dem Englischen entstammenden Namen der Grundfunktionen erreicht.

Eine große Schwierigkeit jeder einführenden Darstellung in LISP besteht darin, daß LISP keine standardisierte Programmiersprache ist, so daß wir heute eine Vielzahl heterogener und teilweise inkompatibler LISP-Dialekte vorfinden. In der vorliegenden Darstellung haben wir uns für TLC-LISP entschieden, einen modernen Dialekt, der weitgehend mit dem zukünftigen „Standard" CommonLISP kompatibel ist; er wurde von John Allen auf Mikrorechnern der Familien Z80/8080 und 8086 unter CP/M implementiert (Allen 1980), die weit verbreitet sind (Bezugsnachweise sind über die Autoren erhältlich).

Die Entstehung dieses Buches aus einer Vorlesung spiegelt sich in der Kapiteleinteilung wider: Jedes Kapitel entspricht in etwa einer Vorlesungsstunde, und — so war jedenfalls die ursprüngliche Idee — in jedem Kapitel sollte ein größeres Programmbeispiel schrittweise entwickelt werden. Für diese Beispiele haben wir zumeist Fragestellungen der Künstlichen Intelligenz als Ausgangspunkt gewählt.

Unser herzlicher Dank gilt allen, die unser Vorhaben kritisch und wohlwollend begleitet haben: W. Bibel, T. Christaller, M. Gehrke, P. Mertens, G. Nees, E. Noeth, F. di Primio, R. Schiedermeier, F. Simon, R. Woitok und W. Wahlster sowie den Gutachtern des Verlages. Für die Erstellung einer ersten Fassung der Reinschrift danken wir herzlich E. Kotal und P. Bächle.

Erlangen-Nürnberg, im Mai 1984 Die Autoren

Inhaltsverzeichnis

1 Einleitung

1.1 Warum LISP?

Es gibt zwei wesentliche Gründe, LISP zu lernen: Zum einen hat LISP interessante Anwendungsgebiete, die sich nur über die Kenntnis der Programmiersprache vollends erschließen lassen. Zum zweiten enthält LISP als Programmiersprache viele einzigartige Elemente und präsentiert viele Konzepte, die aus anderen konventionellen Programmiersprachen auch bekannt sind, in anderer Weise.
Anwendungsgebiete von LISP sind (nicht nach Bedeutung geordnet):

- Formelmanipulation (symbolischer Umgang mit mathematischen Formeln), Kombination solcher symbolischer mit numerischen Schritten in Verfahren der numerischen Mathematik.
- Programmanalyse, -verifikation, und -synthese.
- Programmierumgebungen, Editoren.
- Modellbildung und Simulation.
- Computergraphik.
- Betriebssysteme.
- Übersetzer für Programmiersprachen.
- Expertensysteme, d. h. Programmsysteme, die mit „Wissen" über einen bestimmten Gegenstandsbereich arbeiten, z. B. die medizinische Diagnose unterstützen.
- Problemlösende Systeme, d. h. Programmsysteme, die mit ihren Fähigkeiten zu logischem und inhaltlichen Schließen und zur heuristischen Suche Verstandesleistungen zeigen.
- Verstehen natürlicher Sprache.
- Automatisches Beweisen logischer und mathematischer Sätze, Deduktive Systeme.

Die vier abschließend genannten Anwendungsgebiete sind Teilgebiete des relativ jungen Wissenschaftszweiges „Künstliche Intelligenz" (engl. artificial intelligence), dessen Ziel es ist, durch Programmierung elektronischer Rechenanlagen Systeme zu schaffen, die Leistungen vollbringen, zu denen der Mensch seinen Verstand benötigt.
LISP gilt als **die** Programmiersprache zur Bearbeitung von Problemen der Künstlichen Intelligenz.
Besondere Eigenschaften von LISP sind:

- LISP ermöglicht die Verarbeitung hochstrukturierter symbolischer Daten. Durch die dynamische Speicherverwaltung wird der Programmierer von gewöhnlich unangenehmen (weil fehleranfälligen) Aufgaben entlastet.
- Programme werden durch Datenstrukturen repräsentiert und können als solche verarbeitet, erzeugt und aktiviert werden.

- Durch Beschränkung auf Teilsprachen von LISP bzw. bewußte Anwendung von inhärenten Ordnungsprinzipien kann der Programmierer moderne Programmierstile verwirklichen. Durch LISP ist insbesondere der funktionale Programmierstil eingeführt worden. Auch der objektorientierte Programmierstil wird durch LISP unterstützt.
- LISP-Implementationen sind nicht auf Compiler gestützt, mit denen Maschinenprogramme erzeugt werden, die zusammen mit einem Laufzeitsystem abgearbeitet werden. Vielmehr sind sie typischerweise als interpretierende Verarbeitungssysteme organisiert.
- In LISP-Systemen sind die Aufgaben von Editoren, Syntaxprüfern, Testsystemen, Programmverwaltungssystemen und Compilern integriert.
- Der Programmierer fügt seine Programme und Daten schrittweise dem LISP-System hinzu. Er kann auch vorhandene Systemkomponenten ändern.
- LISP gestattet es, sehr schnell zu Prototypen von Programmen zu kommen.
- In LISP wird eine spezielle Notation für Funktionen verwendet, die Lambda-Notation. Durch diese besteht eine enge Beziehung zum Lambda-Kalkül von A. Church, einem Funktionskalkül der mathematischen Logik. Damit ist LISP ein theoretisch sehr interessanter Forschungsgegenstand.

Es wird Aufgabe des vorliegenden Lehrbuches sein, dem Leser diese Eigenschaften von LISP, ihre Bedeutung und die relevanten Begriffe zu erklären.
LISP bietet auf Grund seiner einfachen und doch allgemeinen Konzeption grundsätzlich alle Möglichkeiten, die in anderen universellen Programmiersprachen verfügbar sind. So treten Zuweisungen, arithmetische Ausdrücke, Datentypen, Kontrollstrukturen in anderer Gestalt auf. Bedingte Ausdrücke sind spezielle Konstrukte in LISP (sie sind auch in den Sprachen ALGOL60 und ALGOL68 enthalten), die sowohl Aspekte von Kontrollstrukturen als auch von Ausdrücken in sich vereinigen.

1.2 Zur Entwicklungsgeschichte von LISP

LISP entstand aus dem Bedarf nach einer höheren Programmiersprache, die zur Entwicklung von Programmen für die Manipulation symbolischer Objekte geeignet sein sollte.
Die erste Formulierung der Eigenschaften von LISP leistete John McCarthy im September 1958. Starke Einflüsse auf den Entwurf gingen von folgenden Aufgabenstellungen aus, für die McCarthy sich interessierte:

- Symbolisches Differenzieren,
- Computerschach,
- Entwicklung von Programmen, die aus geeigneten Situationsbeschreibungen intelligente Schlüsse zu ziehen in der Lage sein sollten (Advice Taker).

Daneben hatte noch eine spezielle Anwendung starken Einfluß:

- ein Programmsystem zum automatischen Beweisen von Sätzen der Geometrie (Winkelsätze, kongruente Dreiecke etc.)

L I S P heißt „LISt Processor“

Listen als Darstellungs- und Strukturierungsmittel für symbolische Daten waren 1956 in einer Programmiersprache namens IPL eingeführt worden. Noch heute sind sie für diesen Zweck so gut wie konkurrenzlos. Ende der fünfziger Jahre war das Konzept der Listen ganz neu und hochaktuell, und allein ihre Verwendung konnte schon eine Programmiersprache charakterisieren. Listen sind bis heute die wichtigste Datenstruktur von LISP geblieben.
Die Flexibilität der Listen hat zu einem guten Teil zum Überleben von LISP beigetragen: Man sollte sich vor Augen halten, daß LISP nach FORTRAN die zweitälteste noch verwendete höhere Programmiersprache ist.
IPL war einer Assemblersprache sehr ähnlich. Es erlaubte nicht die Notation von Ausdrücken, die denen der Arithmetik vergleichbar sind. FORTRAN jedoch, das etwa in derselben Zeit entwickelt wurde, gestattete dies. Um die guten Seiten beider Sprachen zu vereinigen, versuchte McCarthy, die zentrale IPL-Datenstruktur, die Liste, in FORTRAN aufzunehmen. Dazu wählte er den Weg, das Spektrum der Standardfunktionen durch Grundfunktionen für die Listenverarbeitung zu erweitern. Während aber die FORTRAN-Standardfunktionen für numerische Argumente Zahlenwerte lieferten und damit den zugehörigen abstrakten mathematischen Funktionen (die die Menge der entsprechenden Zahlen in sich selbst abbilden) entsprachen, war das für die neuen Grundfunktionen für die Listenverarbeitung nicht der Fall. Man versuchte, auch sie als Funktionen von natürlichen Zahlen zu verstehen, die natürliche Zahlen als Werte haben. Damit gab es jedoch Probleme, weil diese Zahlen nur Adressen von Listenstrukturen darstellten, und die Funktionswerte von Speicherzuständen oder ähnlichen, für die mathematischen Funktionen als zufällig anzusehenden Nebenbedingungen abhingen.
Der Kern des theoretischen Problems bestand also darin, daß man Grundfunktionen mit dem Definitions- und Wertebereich der Listen als Funktionen (Operationen) von Zahlen ansehen wollte. Diese Auffassung stammte aus dem mit FORTRAN zusammenhängenden Denkmodell. Als man verstand, daß ein anderer Definitionsbereich vorlag — nämlich die Menge der „symbolischen Ausdrücke" — war das Problem gelöst. Dies war im März 1959 geschehen; die grundlegende Arbeit mit der theoretischen Begründung und Bezugnahme auf Resultate und Ansätze der Algorithmentheorie (Theorie der Berechenbarkeit) erschien 1960 (McCarthy (1960)).
Ein entscheidender Einfluß auf die Gestaltung von LISP ging von der durch einen Interpreter charakterisierten Implementierung aus. Im Laufe des Jahres 1959 wurde LISP so weit entwickelt, daß es bereits die Mehrzahl seiner wesentlichen heutigen Eigenschaften besaß.
Schon während dieser Entwicklung wurde LISP für die symbolische Berechnung elektrischer Netzwerke und für Entwurf, Programmierung und Test des ersten LISP-Compilers benutzt.
Im März 1960 war das LISP1-Programmiersystem vollendet. Es bestand aus Interpreter, Compiler und Speicherverwaltungsprogrammen.
Die Weiterentwicklung und Portierung auf einen neuen Rechner führte 1961 zum LISP1.5-System, das lange Zeit so etwas wie ein LISP-Standard war (McCarthy et al. (1962)).
Die hauptsächlichen Anwendungen der frühen sechziger Jahre bestanden in symbolischer Vereinfachung arithmetischer Ausdrücke, symbolischer Integration, Programmen zur Überprüfung logischer Beweise und der Verarbeitung natürli-

cher Sprache. Bemerkenswert ist auch die Tatsache, daß der LISP-Compiler sich selbst compilieren konnte. Dies wurde in mehreren Fällen ausgenutzt, in denen LISP-Systeme für neue Rechner erzeugt werden mußten.
Aus einer Implementation von LISP auf der PDP-6 entwickelte sich im Laufe der Zeit MacLISP, das heute auf DEC-10, DEC-20 und MULTICS-Rechnern verfügbar ist. Eine bemerkenswerte Eigenschaft dieser Implementation ist die Effizienz, mit der numerische Programme ausgeführt werden. Der MacLISP-Compiler erzeugt Code von der Qualität eines sehr guten FORTRAN-Compilers.
Aus einer Implementation von LISP auf der PDP-1, die auf eine SDS940 und von da auf eine DEC-10 übertragen und jeweils wesentlich erweitert wurde, entstand InterLISP, das heute auf DEC-10, DEC-20, VAX und Rechnern der Serie Xerox 1100 verfügbar ist. Das InterLISP-Programmiersystem ist sehr umfangreich und zeichnet sich durch großen Komfort an Hilfsmitteln zum Erstellen, Testen, Ändern und Verwalten von Programmen aus.
Wichtige Impulse hat die am Massachusetts Institute of Technology um 1977 begonnene Entwicklung von LISP-Maschinen ausgelöst, Maschinen, deren Sprache LISP ist. Seit 1981 werden derartige Maschinen auf dem Markt angeboten.
Bedauerlicherweise sind diese seit etwa 1965 zu beobachtenden Weiterentwicklungen von LISP1.5 unkoordiniert vollzogen worden, ohne alle Konsequenzen recht zu bedenken. Für den Anfänger unangenehm ist die große Anzahl verschiedener LISP-Dialekte, die sich nicht nur in den Namen wichtiger Grundfunktionen, sondern in ganz wesentlichen Eigenschaften unterscheiden. Eine Standardisierung von LISP1.5 ist versäumt worden.
Jede Einführung von LISP steht daher vor dem Problem, sich auf einen bestimmten Dialekt einlassen zu müssen. Dabei kommt es nicht nur darauf an, daß er dem modernen Stand der Sprachentwicklung entspricht. Weil praktische Übungen erforderlich sind, muß ein LISP-System verfügbar sein. Besonders günstig erscheinen hier Implementationen auf Mikrorechnern. Schließlich sollte der Dialekt mit einem der allgemein verwendeten Dialekte eng verwandt sein. Dies gilt weitgehend für das von uns verwendete TLC-LISP.
Auch heute noch wird LISP weiterentwickelt: Manche Aspekte werden anders als früher gesehen, die Bedeutung wichtiger Elemente wird besser verstanden, und dies schlägt sich in neuen Funktionen und manchmal auch Erweiterungen der Sprache nieder.

1.3 Was sind die Grundbausteine der Programmierung?

Wir wollen Maschinenprozesse der Informationsverarbeitung vorschreiben, d. h. planen oder programmieren. Diese Maschinen arbeiten — wenigstens heute noch —, indem sie nacheinander einzelne Verarbeitungsschritte ausführen. Selbst durch den Einsatz von Mehrfachprozessoren hat sich das noch nicht wesentlich geändert.
Für den Programmierstil ist nun entscheidend, welche Vorstellung der Programmierer von der Programmabarbeitung durch die Maschine hat. Es hat sich gezeigt, daß diese Vorstellung nicht unbedingt die reale Maschinenarchitektur widerspiegeln muß. Die Ansicht, daß man nur effiziente Programme schreiben

kann, wenn man sich dem derzeit vorherrschenden Maschinenmodell (auch v.Neumannsche Rechnerarchitektur) anpaßt, wird heute nicht mehr allgemein geteilt. Auch wird inzwischen großer Wert auf die Sicherheit und Wartbarkeit der Programme gelegt. Programmiersprachen, die vom konventionellen Maschinenmodell ausgehen, schneiden hier keineswegs gut ab.

LISP ist eine der wenigen Sprachen, in denen Programmierung unter Verfolgung von Leitideen verschiedener Programmierstile möglich ist. Die Ursache dafür ist die Unabhängigkeit der Sprache von einem speziellen Maschinenmodell. Wir wollen in diesem Buch einen der möglichen Programmierstile besonders herausheben. Um die Perspektive auf die anderen Wahlmöglichkeiten zu eröffnen, beschreiben wir drei bei der Programmierung in LISP bisher verwendete Maschinenmodelle:

Das **konventionelle Maschinenmodell** geht von schrittweise ablaufenden Folgen von Aktionen (Operationen) aus, die über Daten ausgeführt werden. Demgemäß werden Programme als sequentielle Pläne für solche Aktionenfolgen aufgefaßt. Die Wiederholung von Stücken dieser Folgen wird durch zyklische Ausführung erreicht. Dazu muß die Programmiersprache Ausdrucksmittel bereitstellen. Häufig wiederkehrende Muster von Aktionenfolgen werden zu sog. Prozeduren (Unterprogrammen) zusammengefaßt. Durch spezielle Operationen werden Daten eingelesen und Ergebnisse ausgegeben.

Der sich auf dieses Maschinenmodell beziehende Programmierer benötigt in seiner Programmiersprache

- Sprachelemente für die Grundoperationen, die seine Maschine ausführen kann,
- Mittel zur Beschreibung von Datenstrukturen,
- Mittel zur **Kombination** der Operationen, d.h. zur Ablaufplanung (Hintereinanderausführung, Zyklen),
- Mittel zur Einführung von Unterprogrammen.

Das **funktionale Maschinenmodell** geht vom mathematischen Funktionsbegriff aus. Funktionen in diesem Sinne sind Abbildungen von einem Definitionsbereich (in dem, als Menge von Tupeln, unterschiedliche Mengen kombiniert sein können) in einen Wertebereich. Sowohl Definitions- als auch Wertebereich existieren abstrakt — unabhängig davon, in welcher Reihenfolge Beispielpaare, die die funktionale Beziehung erfüllen, aus ihnen ausgewählt werden. Der Mathematiker sieht auch die Funktion extensional, d.h. als Menge aller dieser Paare, abstrakt gegeben an.

Diese Sicht kann der Programmierer nicht völlig nachvollziehen, weil er gerade Funktionsvorschriften angeben will: Er geht von gewissen Grundfunktionen aus und kann aus ihnen neue Funktionen aufbauen. Das gesamte Programm repräsentiert so eine Funktion vom Definitionsbereich der Eingangsdaten in den Wertebereich der Ausgangsdaten. Wenn der Wert der Funktion für ein bestimmtes Element des Definitionsbereichs ermittelt werden soll, so muß man die Funktion auf dieses Element „anwenden". Die konkrete Realisierung dieser Funktionsanwendung interessiert den Programmierer, der dem funktionalen Modell folgt, nicht: Im Idealfall ist der Wert unmittelbar gegeben. Die durch die Programmiersprache bestimmte Notation für solche Funktionsanwendungen, oft „Ausdruck" genannt, repräsentiert daher für den im funktionalen Modell denkenden Pro-