Zeitschrift: Bulletin des Schweizerischen Elektrotechnischen Vereins, des Verbandes Schweizerischer Elektrizitätsunternehmen = Bulletin de l'Association suisse des électriciens, de l'Association des entreprises électriques suisses

Herausgeber: Schweizerischer Elektrotechnischer Verein ; Verband Schweizerischer Elektrizitätsunternehmen

Band: 80 (1989)

Heft: 3

Artikel: Spezialisierte Hard- und Softwaresysteme für KI-Anwendungen

Autor: Hänscheid, P.

DOI: https://doi.org/10.5169/seals-903636

Spezialisierte Hard- und Softwaresysteme für KI-Anwendungen

P. Hänscheid

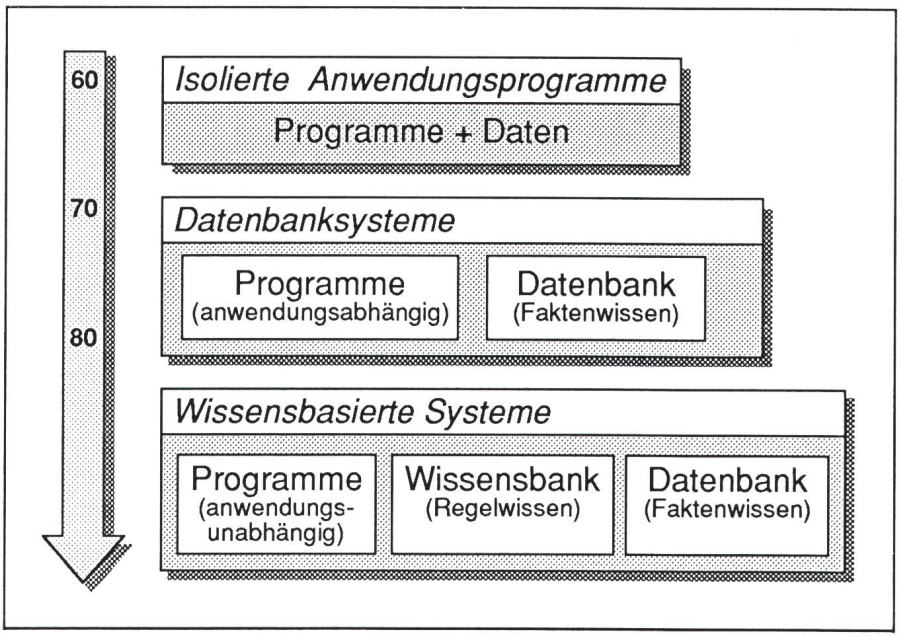
Der Weg von der Daten- zur Wissensverarbeitung führt von den konventionellen Sprachen wie etwa Fortran über die KI-Sprache Lisp und Prolog zur objektorientierten Programmierung. Zur Unterstützung des KI-Systementwicklers stehen heute spezialisierte Hard- und Softwaresysteme zur Verfügung, wie z.B. der Ivory-Chip und Joshua, die bei Symbolics Anwendung finden.

La voie allant du traitement des données à celui des connaissances conduit des langages classiques comme Fortran à la programmation orientée objet en passant par les langages découlant de l'intelligence artificielle Lisp et Prolog. En vue de soutenir le développeur dans le domaine de l'intelligence artificielle on dispose actuellement de systèmes matériels et logiciels spécialisés, comme le Ivory-Chip et Joshua, qui s'utilisent à Symbolics.

Adresse des Autors
Dr.-Ing. *Peter Hänscheid*, Fachbereichsleiter KI-Consult, Symbolics GmbH, Mergenthalerallee 77–81, D-6236 Eschborn/Ts.

Immer häufiger werden *wissensbasierte Systeme* (KI-Systeme) in den verschiedensten Branchen der Industrie, in Universitäten und in Forschungslabors als Intelligenz- und Effizienzverstärker bei der Lösung komplexer Aufgaben, der qualitativen Verbesserung von Entscheidungsfindungen und in der Aus- und Weiterbildung eingesetzt. Die Entwicklung von der Datenverarbeitung zur Wissensverarbeitung begann in den sechziger Jahren mit der Programmierung isolierter Anwendungsprogramme. Diese waren dadurch gekennzeichnet, dass alles Wissen über die zugrundeliegende Anwendung implizit in den Programmen und Daten eingebettet war (Fig. 1). In den siebziger Jahren wurde es nun möglich, durch den Einsatz von Datenbanksystemen Anwendungsprogramme datenunabhängig zu machen. Dabei wurde nur das Faktenwissen in Datenbanken abgelegt; die Regeln für die Bearbeitung blieben fest im Anwendungsprogramm eingebettet. Damit bleibt das Programm anwendungsabhängig. Zu Beginn der achtziger Jahre hatte die KI-Forschung Lösungen erarbeitet, welche die Entwicklung wissensbasierter Systeme möglich machten. Dabei wird, ähnlich wie bei der Auslagerung des Faktenwissens in eine Datenbank, das Regelwissen programmunabhängig in einer Wissensbank aufgebaut und gepflegt (Fig. 1). In einem wissensbasierten System ist demnach das Programm als *anwendungsunabhängiges Steuermodul* zu betrachten.

Die wichtigsten Eigenschaften zwischen der konventionellen Datenver-



Figur 1 Vom Anwendungsprogramm zum wissensbasierten Programm
Von den 60er zu den 80er Jahren

FORMALE KRITERIEN	konventionelle Datenverarbeitung	WISSENS-verarbeitung
Verarbeitung von	numerischen Daten	symbolischen Ausdrücken
Datentypen	wenige Datentypen, aber viele Instanzen eines Typs	viele, auch komplizierte Datenstrukturen, häufig wenige Instanzen eines Typs
Typische höhere Programmiersprachen	Fortran, Pascal, C, Cobol	Lisp, Prolog
Programmier-methoden	konventionell strukturiert	explorativ objektorientiert
Verarbeitungsablauf	explizit festgelegt	implizit oder gar nicht vorgegeben
Verarbeitung unvollständiger Eingaben	werden zurückgewiesen	ist möglich
Modifikationen	selten	häufig
Verarbeitung auf konventioneller Rechnerarchitektur	selbstverständlich	nicht effizient möglich
INHALTLICHE KRITERIEN	**konventionelle Datenverarbeitung**	**WISSENS-verarbeitung**
Automatisierung von	monotonen, klar strukturierten wohldefinierten Informations-verarbeitungsprozessen	hochkomplexen Informationsstrukturen, inkl. Umgang mit diffusem Wissen
zu automatisierende Verarbeitungsabläufe	bekannt	kognitive Prozesse, nicht unmittelbar beobachtbar
hauptsächlich Verarbeitung von	homogen strukturierten Massendaten	heterogen strukturierten Wissenseinheiten
Komplexität durch	Umfang der Datenmenge	Vielfalt der Wissensstrukturen
Korrektheitsbeweis	prinzipiell möglich bei formaler Ein/Ausgabespezifikation	i.d.R. nicht möglich, da Verarbeitung durch Heuristiken und mit diffusem Wissen erfolgt

Tabelle I Unterscheidungsmerkmale wissensbasierter Systeme

arbeitung und der neuen Technologie der Wissensverarbeitung sind in Tabelle I stichwortartig gegenübergestellt. Ein wesentliches Merkmal der Wissensverarbeitung ist - wie man sieht - die Vielfalt und Komplexität der Datenstrukturen. Wissen besteht eben nicht aus streng strukturierten numerischen Daten, und Wissensverarbeitung ist dementsprechend nicht einfach Verarbeitung von numerischen Daten, sondern viel mehr Manipulation von Symbolen (Beispiel: mathematische Operationen +, -, Σ, ∫, Verkehrsschilder, Piktogramme usw.). Hinter einem *Symbol* können sich verbergen:

- Grundinformationen (Fakten),
- Abstraktion der Grundinformation zu Sachverhalten (Modelle),
- Informationen über Verknüpfungsmöglichkeiten zu Handlungsplänen (Regeln)

Wissensverarbeitung erfordert demnach nicht nur eine einfache Datenverarbeitungsanlage, sondern eine *Symbolverarbeitungsmaschine.*

Lisp - die Programmiersprache für Wissensverarbeitung

Sehr viele wissensbasierte Aktivitäten von Menschen können als Liste dargestellt werden. Eine Anzahl von Instruktionen, in bestimmter Reihenfolge organisiert, beinhalten Regeln und Fakten, d.h. Wissen, wie z.B. Kuchenrezepte und Möbelbauanleitungen (auch grafisch oder symbolhaft). Die Programmiersprache *Lisp* wurde speziell für die Belange der Wissensverarbeitung entwickelt und kann bereits auf einen Erfahrungszeitraum vergleichbar mit dem *Fortran* zurückblikken, formulierte doch *John McCarthy* 1957 schon die erste Lisp-Version. Seit 1960 wurde sie am MIT ständig weiterentwickelt. Der Industriestandard Common Lisp existiert seit 1984. Der Basisdatentyp von *Lisp* ist die Liste (Lisp steht für List-Processing). Darunter ist eine Struktur mit linearer Anordnung von Datenelementen zu verstehen, die selber wieder Listen sein können.

Lisp-Listen bestehen insbesondere aus den drei Objekttypen:

- Konstanten,
- Variablen,
- Funktionen,

wobei Variable und Funktionen im allgemeinen über Symbole angesprochen werden. Symbole haben einen Namen und können wiederum Listen, d.h. Wissen, repräsentieren.

Lisp ermöglicht den Aufbau und die Verarbeitung hochstrukturierter symbolischer Daten und ist somit vorzüglich geeignet als Hochsprache für die Wissensverarbeitung. Sämtliche Datentypen werden als Listen repräsentiert und bedürfen keiner Vereinbarung. Beliebig komplexe Datenkonstrukte können erzeugt und selbst während der Laufzeit von Programmen verändert werden. Lisp bietet die für die Wissensverarbeitung erforderliche hohe Flexibilität.

Objektorientierte Programmierung

Im Gegensatz zu Lisp sind herkömmliche Programmiersprachen sehr stark auf die sogenannte *von-Neumann-Architektur* der konventionellen Universalrechner zugeschnitten. Die Wirkung eines Programms ist hier ein aus sequentiellen Speichertransformationen zusammengesetzter Prozess. Fortschrittliche KI-Programmiersprachen orientieren sich an etablierten mathematischen Konzepten, wie dem Funktionsbegriff (bei Lisp) oder dem des logischen Prädikats (bei Prolog). Viele Abläufe, die in Fortran oder Pascal als Aneinanderreihung von Speichertransformationen angesehen werden können, sind in Prolog oder Lisp durch die Konstruktion abstrakter, komplexer Datentypen zu bewerkstelligen.

Der übergeordnete Begriff zu diesen fortschrittlichen Programmierkonzepten ist *objektorientierte Programmiermethodik.* Jede Komponente innerhalb eines mit objektorientierter Programmierung erstellten Systems ist ein *Objekt,* sei es nun eine Variable, eine Konstante oder eine beliebig komplexe Datenstruktur. Ein Objekt modelliert die Eigenschaften eines Real-Welt-Objekts einschliesslich seiner

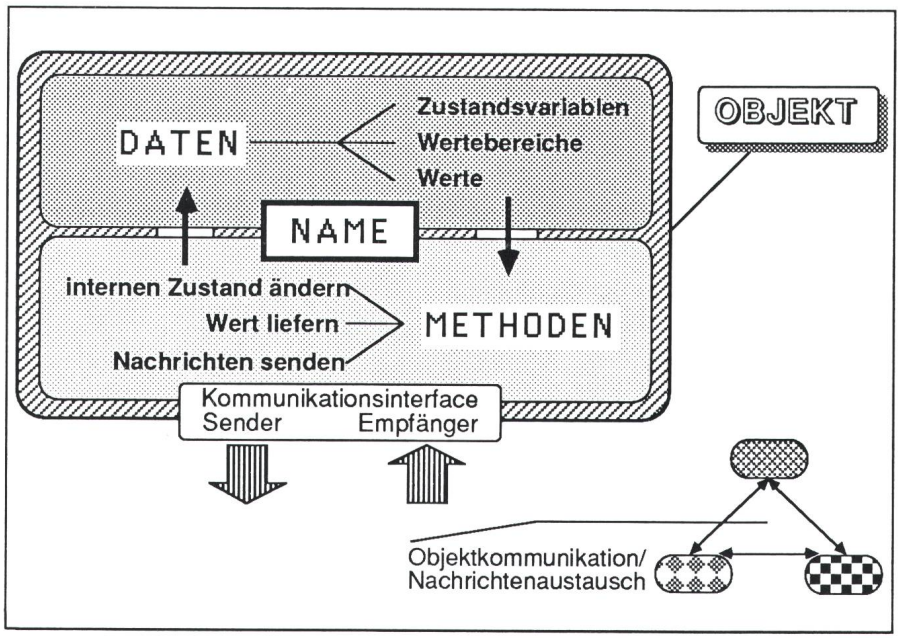
Funktionalität, d.h. seinen Kommunikationsmöglichkeiten und Verhaltensweisen zu anderen Objekten.

Im Computer werden Objekte nach bestimmten Aufbaumustern erzeugt. Derartige Rahmen von Erzeugungsregeln nennt man *Klassen* oder innerhalb Symbolics *Common Lisp* auch *Flavors*. Flavors sind also abstrakte Datentypen im New-Flavors-System, d.h. Objektschablonen, die diejenigen Eigenschaften und Verhaltensweisen definieren, die alle Individuen dieser Klasse gemeinsam haben. Komplexe Datentypen (z.B. Felder, Pointer) kannte man auch schon bei herkömmlichen Programmiersprachen. In der objektorientierten Programmiermethodik aber wurden diese Konzepte konsequent weiterverfolgt und in fortschrittlichen Programmiersprachen erheblich weiterentwickelt.

Die Erzeugung eines individuellen Objekts geschieht durch Instanzierung aus einer Klasse (Objektschablone). Die Begriffe *Objekt* und *Instanz* werden somit synonym gebraucht. Eine Instanz erbt alle Zustandsvariablen und auch ihre Verhaltensweisen (Methoden) von ihrer Klasse. Die Belegung der Zustandsvariablen ist individuell, d.h. eine Instanz bzw. ein Objekt ist in der Objektwelt einmalig.

Nachrichten dienen der Kommunikation zwischen Objekten. Beim empfangenden Objekt löst eine Nachricht eine individuelle Reaktion aus (Methode). Eine Reaktion kann beispielsweise die Veränderung einer Zustandsvariablen, die Rückgabe eines Wertes oder das Abschicken einer Nachricht an ein drittes Objekt sein (Fig. 2). Die Möglichkeit der Mehrfachvererbung des Flavor-Konzeptes erlaubt den wirtschaftlichen und bequemen Aufbau von Objektklassen. Einerseits wird dabei das hierarchische *Vererbungskonzept* unterstützt, das für eine zu definierende Klasse die Eigenschaften von einer bereits existierenden, übergeordneten Klasse (Superklasse) automatisch übernimmt, so dass nur noch die zusätzlichen, weiterdifferenzierenden Eigenschaften spezifiziert werden müssen (Fig. 3), und anderseits erlaubt diese Technik auch das Mischen von Superklassen zu einer neuen Klasse (triviales Beispiel: *Schiffe*-Klasse + *Automobil*-Klasse = *Amphibienfahrzeug*-Klasse).

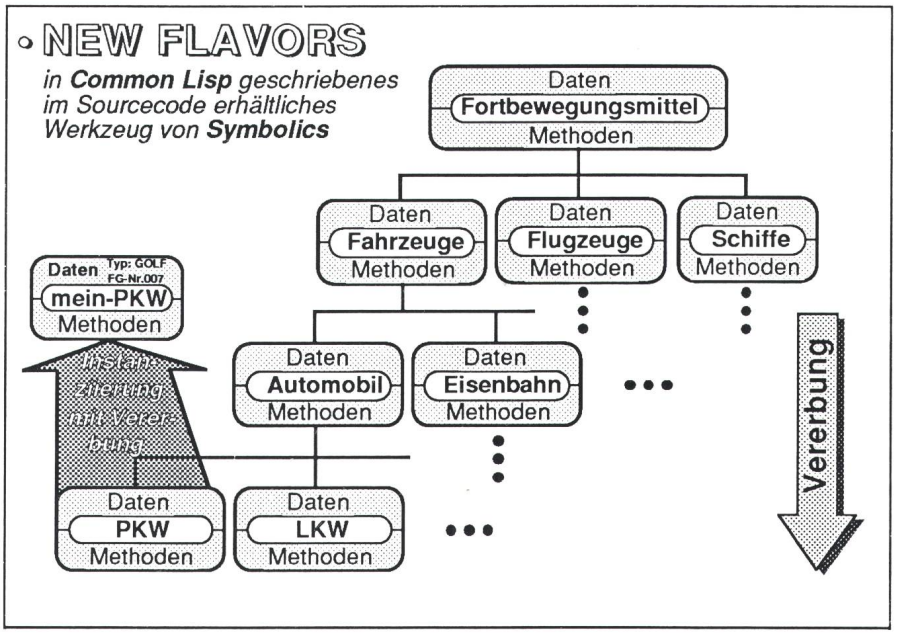
Dem modernen objektorientierten Programmierstil trägt Symbolics Rechnung mit seiner integrierten Entwicklungs- und Anwendungsumgebung *Genera*, seinen auf Common Lisp aufbauenden Softwarewerkzeugen *New Flavors* und *Joshua* (Fig. 4) und dem objektorientierten Datenbanksystem *Statice*. Die letztere erlaubt, Objekte persistent (nicht flüchtig) und mehrbenutzerfähig in einem Computernetzwerk zu verarbeiten.



Figur 2 Funktionalitäten eines Objektes beim objektorientierten Programmierstil

Abstraktionsfähigkeit der natürlichen Sprache als Ziel

Die gesamte Entwicklung der höheren Programmiersprachen von Cobol über Fortran, C, Pascal zu Ada hatte immer schon zum Ziel, Aufgabenstellungen möglichst problembezogen und maschinenunabhängig zu formulieren. Mit den Software-Technologien der KI und den Programmiersprachen Lisp und Prolog wird zielgerichtet das Abstraktionsniveau der natürlichen Sprache des Menschen zur Problemformulierung angestrebt (Fig. 5). Hier ist das objektorientierte Programmieren mit *New Flavors* ein ebenso konsequenter weiterer Schritt wie die in Ge-



Figur 3 Entwicklungswerkzeug für objektorientiertes Programmieren

nera integrierbare KI-Hochsprache Joshua. Dieses Instrument zum Bau von Expertensystemen und wissensbasierten Applikationen erweitert syntaxkonsistent die Funktionalität von Genera um Standard-KI-Komponenten (Fig. 4).

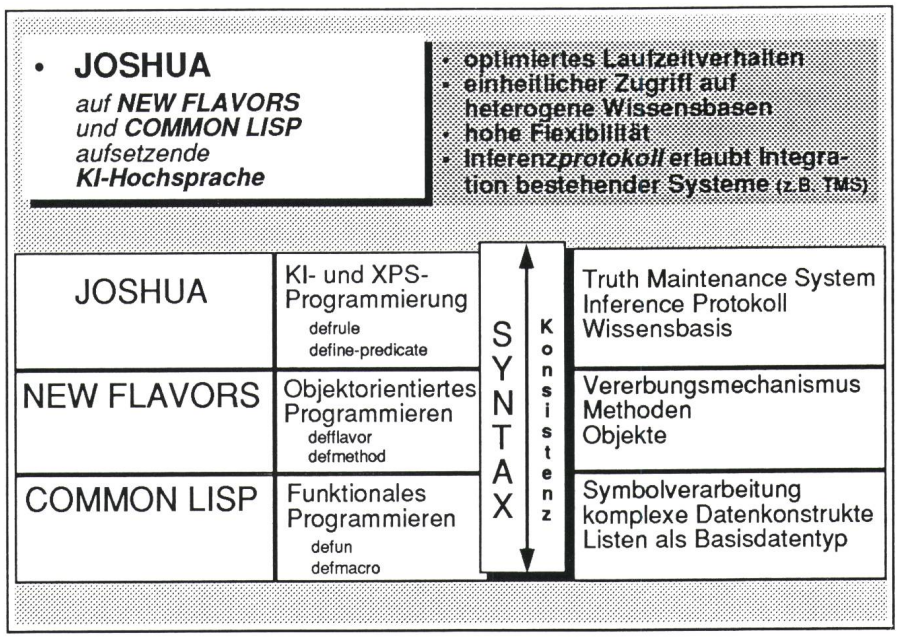
Produktivität in Entwicklung und Anwendung

Jeder erfahrene Systementwickler weiss, dass bei der Entwicklung von komplexen Programmen erhebliche Zeit für das Testen, Verbessern und Optimieren des Systems aufzuwenden ist. Der Begriff der Software-Gaps ist in aller Munde. Bei wissensbasierten Systemen richtet sich die zentrale Forderung für Betriebssysteme, Entwicklungsumgebungen und Applikationen auf eine wesentlich leistungsfähigere Mensch-Maschine-Schnittstelle.

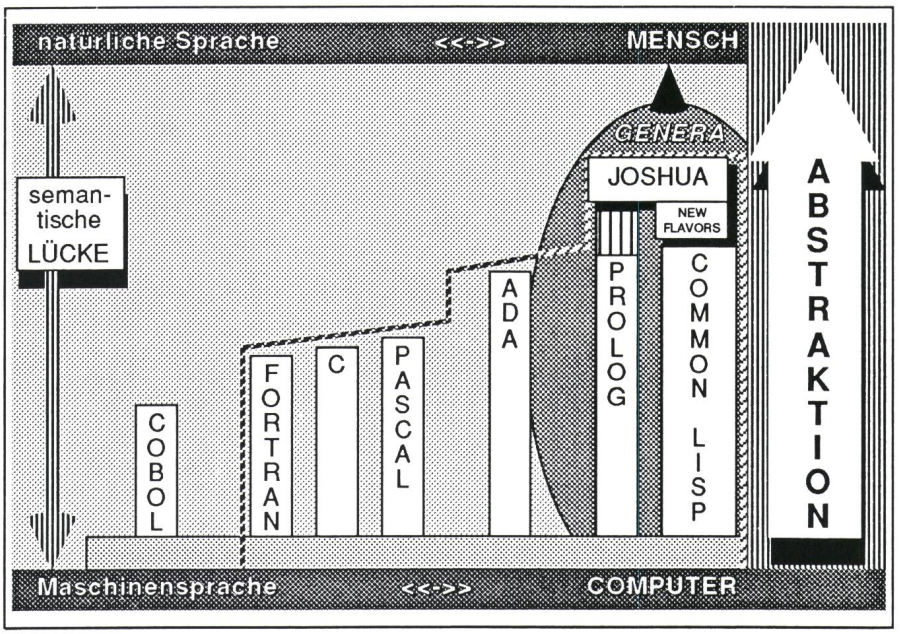
Symbolics Common Lisp wurde konzipiert, um ein interaktives Multiprozess-Umfeld vollständig zu unterstützen. Die Prozesssynchronisation ist durch gleichzeitige gemeinsame Benutzung des virtuellen Speichers durch alle Prozesse möglich. Die Kommunikation zwischen den Prozessen kann so erfolgen, dass gewisse Lisp-Objekte gemeinsam benutzt werden.

Das Benutzerumfeld - die sehr leistungsfähige Software-Entwicklungsumgebung *Genera* - von Symbolics macht konsequent von der objektorientierten Programmiermethodik Gebrauch und umfasst mehr als 1 Million Zeilen Lisp-Code. Es bildet die Grundlage für eine ausserordentlich leistungsfähige Mensch-Maschine-Schnittstelle. Die Aufteilung des Bildschirms in einzelne Fenster, die hochauflösende, schnelle Grafik sowie die durchgängig mausgesteuerte Menütechnik sind zur Selbstverständlichkeit geworden. Zur Unterstützung der Systemarbeit können - gerade bei komplexen Projekten und Anwendungen - nicht einzelne Werkzeuge, sondern nur eine konsistente Benutzerumgebung die notwendige Hilfe geben.

Um eine höhere Produktivität in der Entwicklung und Anwendung zu erreichen, muss der Benutzer von allem entlastet werden, was mit Bedienungsspezialitäten, Systemabläufen, Speicherverwaltung, Dokumentation und weiteren systemtechnisch bedingten Details zu tun hat. Die Symbolics-Workstations wurden für komplexe Aufgaben der Wissensverarbeitung entwickelt und leisten somit optimale Unterstützung für



Figur 4 Joshua-Entwicklungswerkzeug für KI-Softwareprogrammierung



Figur 5 Ziel: Abstraktionsfähigkeit der natürlichen Sprache

- traditionelle Softwareentwicklung,
- objektorientiertes Programmieren,
- exploratives Programmieren,
- Rapid Prototyping.

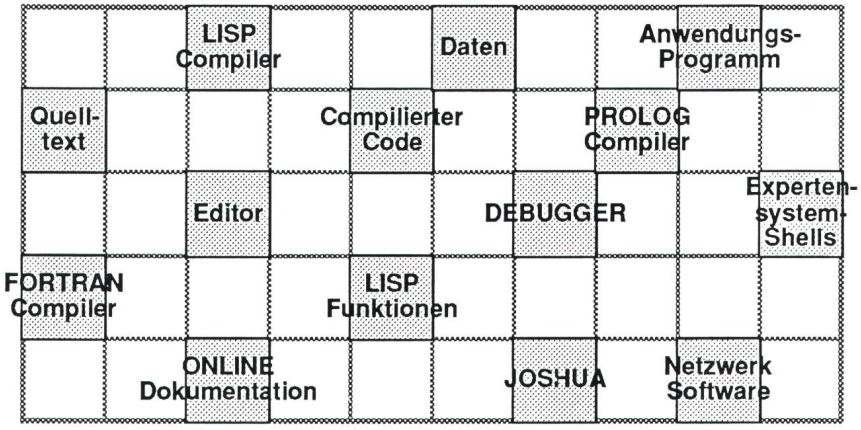
Bei Softwareprojekten, die in den Phasen Problemanalyse, Strukturierung oder Flussdiagrammerstellung bereits an der Komplexität scheitern - hierzu gehören insbesondere wissensbasierte Systeme - setzt sich das Konzept des *Rapid Prototyping* durch, das dem Endbenutzer schon in sehr frühem Stadium eine Vorstellung von seinem Programm gibt. Durch schrittweise Verfeinerung, orientiert an den Angaben des Auftraggebers, entsteht so mehr oder weniger interaktiv das Programmsystem.

Die in Genera eingebetteten Hilfsmittel und die Möglichkeit des inkrementalen Kompilierens sowie der Wegfall der Load-Link-Phase ermöglichen die effiziente Nutzung dieser Technik. Ebenso wird das explorative Programmieren gefördert, bei dem der Programmierer bewusst spielerisch mit der Maschine arbeitet, um seine Gedanken und Konzepte zu klären. Da-

durch wird ihm ein kreativitätsförderndes Element (zurück)gegeben, das bei der traditionellen Softwareentwicklung durch die langen Entwicklungszeiten auf der Strecke bleibt.

Hier sollen nicht alle Charakteristika von Genera diskutiert werden. Erwähnt werden muss jedoch, dass es eine Mehrsprachenumgebung ist. Auf Lisp basierend, unterstützt Genera eine Reihe weiterer Sprachen ebenso effizient. Dazu zählen Prolog, Fortran, Pascal, C und Ada. Hier wird Integration ebenfalls gross geschrieben; denn die Entwicklungsunterstützung dieser Sprachen ist gleichwertig zu Lisp in Genera integriert und die Kommunikation der Sprachen untereinander ist möglich.

Integrierte Software-Entwicklungsumgebung bedeutet, dass alle Werkzeuge, Programmsysteme, Daten, Anwendungsprogramme usw. ständig im virtuellen Speicherbereich vorhanden sind. In der modellhaften Darstellung des virtuellen Speichers in Figur 6 sind beispielhaft einige Pages mit Objekten belegt. Auf einen einzigen Tastendruck hin erscheint in Sekundenschnelle das Fenster der angewählten Anwendung auf dem hochauflösenden Grafikbildschirm.



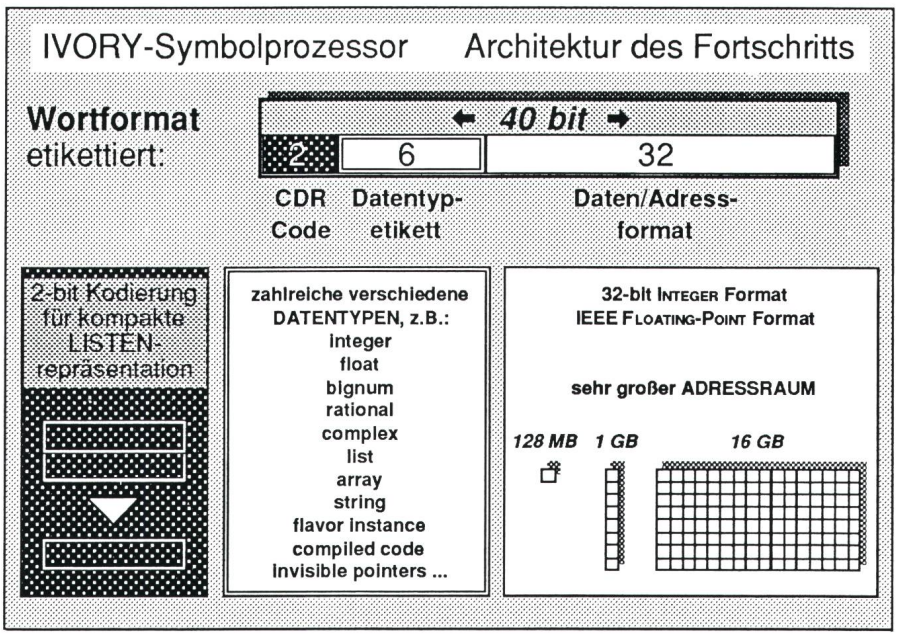
Figur 6 Integrierte Genera-Softwareentwicklungsumgebung
Alle Werkzeuge, Programme und Daten liegen zugriffsbereit im virtuellen Speicher

Softwareanforderungen bestimmen Hardwarearchitektur

Aus dem Prinzip des Software first, aus den speziellen Eigenschaften von Lisp und aus dem hohen Komfort der vorgestellten Hochleistungsentwicklungsumgebung ergeben sich spezielle Anforderungen an die Hardware, um den Leistungsbedarf für die Entwicklung komplexer wissensbasierter Anwendungen abzudecken. Symbolics verwendet bei Ivory nicht nur eine Wortbreite von 40 Bit, sondern auch die heute wohl fortschrittlichste Symbolprozessorarchitektur. Diese ergibt sich im wesentlichen aus den Softwareanforderungen.

Für viele arithmetische Operationen werden vom IEEE-Standard 754-1985 32-Bit-Daten verlangt, in denen ganze Zahlen (Fixnums) oder Gleitkommazahlen von einfacher Genauigkeit (Flonums) oder auch ganze Zahlen (Fixnums)untergebracht werden können. Zusätzlich zur strikten Einhaltung dieser 32-Bit-Datenformate und der damit sichergestellten Kompatibilität zu numerischen Koprozessoren, Standard-Workstations und Standard-PCs werden 8-Bit paralleles Tag[1]-Processing und damit eine Datentypprüfung für 64 verschiedene Datentypen (während der Laufzeit in der Hardware) ermöglicht sowie eine kompaktierte Listenrepräsentation (Fig. 7) und eine hardwaregestützte Speicherbereinigung (Garbage Collection). Die Wortbreite von 40 Bit erlaubt mit dem hier verwendeten Zeigerformat eine Adressierung von 16 GByte virtuellen Speichers, die nach dem Demand-Paging-Verfahren verwaltet werden. Die Hardwareunterstützung für Objekte und optimierte Funktionsaufrufe bewirken eine erhebliche Effizienzsteigerung bei der Lisp-Verarbeitung.

Die Datentyp-Bit dienen der Unterscheidung zwischen verschiedenen Datentypen. Diese Bit werden von der Hardware parallel zu den eigentlichen Operationen ausgewertet. Ausser den 6 Datentyp-Bit stehen zwei weitere sogenannte CDR-Bit zur Verfügung, die eine kompakte Abspeicherung von Listen erlauben. Bei der Listenspeicherung auf Universalrechnern wird je Li-



Figur 7 Symbolics Ivory – Tagged Architecture für Symbolverarbeitung

[1] Tag bezeichnet eine Datentypetikette

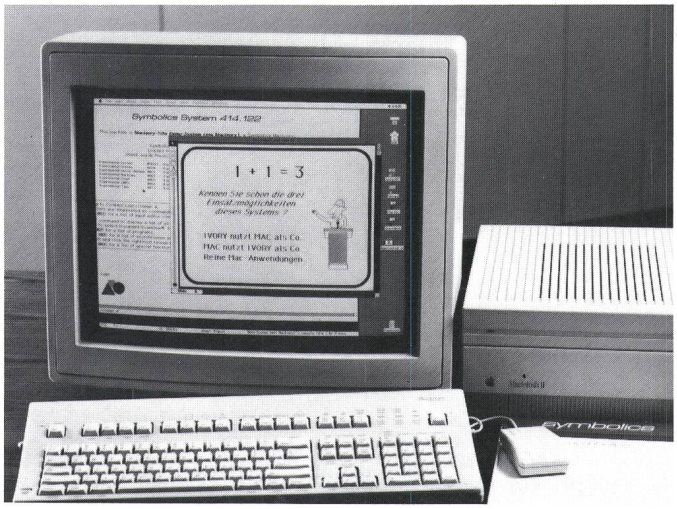
stenelement noch je ein weiterer Speicherplatz für einen Zeiger zum nächsten Listenelement verwendet. Bei Symbolics werden Listen meistens in kompakter Form abgelegt, die Listenelemente stehen also in konsekutiven Speicherzellen und die zwei CDR-Bit zeigen diese kompakte Darstellung an. Natürlich ist die interne Repräsentation der Listen für die Benutzer transparent. Die Vorteile dieses CDR-Coding sind Speicherersparnis und erheblich schnellere Zugriffszeiten, da das Pagingverhalten sehr günstig beeinflusst wird.

Symbolics hat mit einem eigenen wissensbasierten VLSI-Design-Werkzeug in einer Rekordzeit Ivory, den 40-Bit-Symbolprozessor auf einem Chip, fertiggestellt. Der Ivory-Chip wird als Grundbaustein der vierten Symbolprozessor-Generation von Symbolics angesehen, die für verschiedene Aufgaben zum Einsatz kommt:

- als kompakte hochleistungsfähige Entwicklungsmaschine,
- als KI-Koprozessor für Standard-Workstations,
- als leistungsfähiger KI-Koprozessor für 32-Bit-PCs.

Der Ivory-Prozessor ist zusammen mit einer Grundausstattung von Cachespeicher- und Interface-Bausteinen auf einer einzigen Steckkarte untergebracht. Zusammen mit einer Memorykarte kann die Ivory-Einschubkarte die komplette Funktionalität von Genera und der darauf entwickelten Applikationen in verschiedenen Host-Umgebungen zur Verfügung stellen.

Figur 8
Symbolics MacIvory, der kompakte KI-Computer



Die erste Implementation des 40-Bit-Symbolprozessors als KI-Koprozessor in einem 32-Bit-PC mit einer eleganten Software-Integration wurde im August 1988 mit dem Namen MacIvory vorgestellt (Fig. 8). Der Benutzer dieses Systems kann neben wissensbasierten Anwendungen unter Genera die Vielfalt der Apple-Macintosh-Applikationen benutzen.

Eine VME-Bus-basierte Ivory-Einschubkarte wird – neben dem heutigen Einsatz in der Symbolics-Workstation XL400 – auch in die Unix-Betriebssystemumgebungen anderer VME-Bus-basierter Workstations integriert. Ebenso wird die 80 386-basierte PC-Welt für Ivory erschlossen werden. Komplexe KI-Systeme halten damit Einzug in die Praxis.

Die Firma Symbolics hat die KI-Forschungsergebnisse der letzten 30 Jahre in die Praxis umgesetzt. Sie bietet heute bereits die 4. Hardware-Generation sowie die 7. Generation der Symbolics-Software-Entwicklungsumgebung *Genera* an. Das wesentliche Merkmal der dedizierten Symbolics-Hardware ist ihre Fähigkeit, mit höchster Effizienz Symbolverarbeitung zu ermöglichen. 1987 stellte Symbolics mit *Ivory* den ersten 40-Bit-Symbolprozessor auf einem Chip vor. Die zukunftsweisende Architektur und die zugrundeliegende Strategie zur Einbettung von Ivory in herkömmliche Standard-Computer brachten dieses Jahr die KI-Entwicklung einen wesentlichen Schritt weiter zum Ziel der Realisierung praxisreifer KI-Applikationen von beinahe beliebiger Komplexität.