

Computational Linguistics Computerlinguistik

An International Handbook on Computer Oriented
Language Research and Applications

Ein internationales Handbuch zur computergestützten
Sprachforschung und ihrer Anwendungen

Edited by / Herausgegeben von
István S. Bátori · Winfried Lenders
Wolfgang Putschke

Offprint/Sonderdruck

*Höltz, Messeschrift und
Conussee*

Walter de Gruyter · Berlin · New York
1989

buch 166 · J. Ehlers 1983 · C. F. Goldfarb 1984 · S. Hockey/I. Mariott 1979 · E. Kohl 1977 · Standard for Electronic Manuscript Preparation and Ex-

change 1985 · S. Schindler et al. 1983 · J. M. Smith 1983 · W. Ott 1983.

Raimund Drewek, Stuttgart (Bundesrepublik Deutschland)/Zürich (Schweiz)

63. Eignung von Programmiersprachen zur Lösung linguistischer Problemstellungen: Entwicklung und Perspektiven

1. Übersicht
2. Eignungskriterien
- 2.1. Datentypen-bezogene Eignungskriterien
- 2.2. Kontrollstruktur-bezogene Eignungskriterien
- 2.3. Probleme der Datenmengen und Zugriffsarten
- 2.4. Benutzerfreundlichkeit, Universalität und technische Eignungskriterien
3. Historische Entwicklung
- 3.1. Überblick über klassische CL-Sprachen
- 3.2. Welche Programmiersprachen werden in der CL-Praxis verwendet?
4. Einflußnahme der CL auf die Programmiersprachenentwicklung
5. Literatur (in Auswahl)

1. Übersicht

In diesem Artikel sollen die in der Computerlinguistik (im folgenden CL genannt) benutzten Programmiersprachen auf ihre Eignung für diesen Zweck untersucht werden. Da es sehr viele Programmiersprachen — auch in diesem Bereich — gibt, muß zunächst eine Auswahl getroffen werden von besonders typischen oder besonders weit verbreiteten Vertretern. Dies sind folgende Sprachen (daneben ist das Jahr angegeben, an dem die jeweilige Programmiersprache veröffentlicht wurde):

— COMIT II	'57
— LISP	'58/'62
— FORTRAN	'58
— SNOBOL/SL5	'64/'76
— SmallTalk	'72/'80
— BASIC	'66
— Pascal	'71
— PL/1	'65
— PROLOG	'73
— Comskee	'76

Es soll hier nicht versucht werden, diese Sprachen alle im einzelnen vorzustellen, dazu wird auf die Literatur verwiesen, insbesondere auf (Sammet 1969, 382 ff., 540 ff.), wo auch auf weitere relevante Sprachen ein-

gegangen wird. Der folgende Artikel beschränkt sich darauf, die verschiedenen Beurteilungskriterien für eine Programmiersprache zu diskutieren und darauf hinzuweisen, wie sich die betrachteten Programmiersprachen zu den einzelnen Eignungskriterien verhalten (Kap. 2.). Im weiteren Verlauf wird in gleicher Art die historische Entwicklung beleuchtet (3.1.), insbesondere auch die Einflußnahme der CL auf die Entwicklung von Programmiersprachen (4.). In Abschnitt 3.2. wird eine Verbreitungsstatistik vorgestellt.

2. Eignungskriterien

2.1. Datentypen-bezogene Eignungskriterien

Da jede linguistische Datenverarbeitung auf *Textverarbeitung* aufbaut, ist es notwendig, in der benutzten Programmiersprache einen Datentyp zur Repräsentation von Zeichenketten (strings) zur Verfügung zu haben. Die komfortabelste Lösung dabei ist, daß man direkt einen Datentyp für strings hat, der möglichst keine Längenbeschränkung mit sich bringt (z. B. Comskee, SNOBOL mit Vorgänger COMIT und Nachfolger SL5, manche BASICs, mit Einschränkungen auch LISP (als Listen)). Die nächst weniger komfortable Lösung wird etwa durch den PL/1-Ansatz dargestellt, das variable, aber beschränkt lange Zeichenketten als direkten Datentyp zur Verfügung stellt. Am unteren Ende liegt der FORTRAN-Ansatz, der lediglich Zeichenketten fester Länge kennt, die dann mit Blanks (Leerzeichen) aufgefüllt werden müssen.

Doch nicht nur der Datentyp als solcher sollte vorhanden sein, auch die zugehörigen Operationen bestimmen die Benutzbarkeit eines Datentyps. Hierbei dürften Comskee und LISP die größte Auswahl bieten, bei Comskee als eingebaute Operationen, bei LISP als im Laufe der Zeit hinzugefügte

Um Struktur-Mustervergleiche bedeuen be-
schreiben zu können, ist es wichtig zu erläutern,
in der Verwendung von Programmsprachen
auch einen Regelformalismus für Grammati-
ken zu bestimmen. In einem solchen Formali-
smus kann man dann etwa Baumtransforma-
tionen beschriften, die gerade das in 2.1. an-
gesprochene Problem der Überführung von dis-
kontinuierlichen syntaktischen Strukturen in
zusammenhängende Strukturen lösen. Bei-
spiele für die entsprechenden Sprachmittel

2.2. Kontrollstruktur-bezogene Eingangsskriterien

Ganz eng verwandt mit der Problematik der Repräsentation von Graphen ist das Problem des *Struktur-Matchings* (*Struktur-Muster-Vergleich*). Diese Aufgabe stellt sich etwa bei der Text-Analyse, wenn bestimme Strukturen erkannnt und transormiert werden sollen. Eine solche Mustererkennung kann man nur sehr schlecht auf der String-Repräsentation des Textes vornehmen — auch wenn dies z. B. bei SNOROL (nämlich unter Benutzung eines Musters, das durch eine Kette von Zeichenwörtern definiert ist) der Fall ist. Die Ergebnisse dieser Werte sind die Struktur und die Abstimmungsabläufe wie klinisch als Baume darzustellen und dieser Struktur nach dem Schichtenprinzip muss auch Verteile haben werden, was gewisst man Effizienz, indem man kann: Oft gewinnt man durchaus auch Verteile, die ausprogrammiert sind, was den Suchalgorithmus dem Problem entsprechen lässt.

Funktionsnen. Auch bieten die meisten BA-
SICs eine ganze Reihe von Operatoren an,
wobei gegen SNOBOL mit seiner einzigen
Operatoren (Paternmatch und anschließender
Mustersetzung) weniger Flexibilität bietet.
Bei FORTAN ist der Antwortdruck an-
gebiesen, was er sich selbst an Funktionen
und Prozeduren schreibt — wenn man von
den vielfältigen Möglichkeiten der String-
verarbeitung im Zug der Ein/Ausgabe ab-
sieht.

Das Paternmatching (Musterkennung)
bildet die Basis einer jeden SNOBOL-Opera-
tion. In dieser Sprache wird diese Operation
in umassender Weise unterstellt: Man hat
alle Möglichkeiten, Pattern zu definieren, auf
anderen aufzubauen, dynamisch, und sogar
rekursiv, so daß die gesamte Machtskraft
kontextfrei Grammatiken zur Verfugung
steht. Schön der SNOBOL-Vorläufer CO-

MIT baute sehr stark auf dem Vorgang des
Pattern-Matching auf, allerdings ohne die
Möglichkeit, es zu ändern.

liefern wiederum SNOBOL — wenn auch nur auf String-Ebene — und PROLOG.

PROLOG besitzt darüber hinaus — als zentrales Sprachelement — eine Möglichkeit, *logische Inferenzen* (Schlußregeln) auszudrücken. Diese erlauben es, in eleganter Weise z. B. die gerade erwähnten Baumtransformationen noch von weiteren Bedingungen semantischer Art, wie z. B. von bestimmten Kasus- oder Valenzrahmenbedingungen, abhängig zu machen.

Logische Inferenzen und Grammatik-Regeln werden auch als *deklarative* Sprachmittel bezeichnet. Sie stehen im Gegensatz zu *imperativen* (prozeduralen) Sprachmitteln, mit denen man eher den Gang eines Algorithmus als seinen Netto-Effekt beschreibt. Es gibt seit längerer Zeit einen wissenschaftlichen Streit darüber, ob man sich nicht den größeren Komfort, den man durch deklarative Sprachmittel zur Verfügung hat („man überläßt das Programmieren dem Computer“), durch zu sehr herabgesetzte Effizienz — die sogar zur Nichtbenutzbarkeit eines Programms führen kann, erkauft. Unabhängig davon gibt es Anwendungen, die sich nur schlecht in ein vorgegebenes deklaratives Schema pressen lassen, so daß es hier die Verwendung prozeduraler Sprachmittel erlaubt, einen Algorithmus knapper und eleganter zu beschreiben. Vertreter des deklarativen Ansatzes sind SmallTalk (Goldberg 1984, 1 ff.) (wenngleich hier auch der Begriff der objektorientierten Programmierung ein bestimmendes Merkmal darstellt) und besonders PROLOG. Verwandte Konstrukte gibt es in SNOBOL, auch sind die funktionalen Programmiersprachen (LISP) nicht so weit vom deklarativen Ansatz entfernt wie die imparativ-prozeduralen Programmiersprachen.

2.3. Probleme der Datenmengen und Zugriffsarten

Einen ganz anderen Aspekt bieten die Daten an sich, die in typischen linguistischen Anwendungen auftreten. Gehen wir wieder von der Zeichenkettenverarbeitung aus, so tritt schon hier das Problem der *Dynamik der Daten* auf. Wenn man nur einen einzelnen Satz eines Textes betrachtet, so kann dieser Satz aus 20 oder auch aus 2000 Zeichen bestehen. Auch bei Wortmengen, Regelmengen etc. hat man das Problem, daß man im vorhinein schlecht angeben kann, wie groß diese Mengen werden, in dem Sinne, daß sich — in den meisten Fällen — nur viel zu große obere

Schranken für die maximale Elementanzahl angeben lassen. Hier ist es hilfreich, von der Programmiersprache Unterstützung zu finden. In LISP, PL/1 und Pascal wird man dynamische Strukturen auf selbst verwaltete Listen abbilden, was eine gewisse Komfort-Einbuße, z. B. verglichen zu Comskee, darstellt. Gerade am Beispiel von Mengen läßt sich das illustrieren: Pascal hat zwar ein Mengenkonzept, dies kann man aber nur in solchen Fällen verwenden, in denen die Elemente einer schon im Programm festgelegten Obermenge entstammen (z. B. die 4 Kasus). Ist diese Voraussetzung verletzt, so muß auf die komplizierter zu handhabende Listenverwaltung ausgewichen werden. Im Gegensatz dazu bietet Comskee einen Datentyp 'Menge', bei dem keine Obermenge festgelegt zu werden braucht, und auch die Elementanzahl a priori unbeschränkt ist.

Ein zu der Dynamik der Daten verwandtes Problem ist das der *großen Datenmengen*. Gehen wir wieder von der Analyse natürlicher Sprachen aus. Bei dieser Aufgabe begegnen uns verschiedene „große Daten“. Die „langen Sätze“ hatten wir gerade angesprochen, daneben die relativ großen und — durch die in natürlicher Sprache vorhandenen Mehrdeutigkeiten — vielen Ableitungsbäume, die es auch zu verwalten gilt, und die nicht immer in den (virtuellen) Hauptspeicher eines Rechners passen. Man rufe sich ins Gedächtnis, daß sich Mehrdeutigkeiten multiplizieren. Hat man etwa für einen Satz mit 10 Wörtern bei jedem Wort 2 Alternativen, so ergibt das schon $2^{10} = 1024$ Möglichkeiten (Rohmehrdeutigkeit). In Wirklichkeit ist der Durchschnittswert eher höher als 2 Alternativen pro Wort.

Eine Facette der Problematik der großen Daten ist die Frage, wie in einer Programmiersprache ein *Wörterbuchzugriff* bzw. allgemeiner ein *assoziativer Zugriff* oder *Datenbankfunktionen* zu realisieren sind. Die in die jeweiligen Programmiersprachen eingebauten Möglichkeiten — wenn überhaupt vorhanden — scheitern in der Regel an der Größe der Daten (z. B. in LISP und in SNOBOL gibt es für diesen Zweck Sprachmittel, sie sind aber auf eine Speicherung im Hauptspeicher beschränkt), allein in Comskee gibt es einen dedizierten Datentyp 'Wörterbuch'. Die Lösung des Problems erfolgt in anderen Programmiersprachen üblicherweise über den Mechanismus des Dateizugriffes auf ISAM-Dateien (PL/1) oder über Aufrufe (call-Schnittstelle) von Anschlußroutinen ei-

In der Anfangszeit der Computer gab es nur Maschinen-Code (Eingabe der Befehle als Bit-Muster). Einige groben Schritte nach vorne stellten dann die Assembler dar, die eine Ver-

3. Historische Entwicklung
3.1. Überblick über CL-Sprache

Bei der Auswahl der Programmiersprache spielen neben den sachlichen Gründen auch technische Aspekte eine besondere Rolle. Was nutzt eine herorrägend geingnete Programmiersprache, wenn sie auf der vergangenen Maschine nicht implementiert ist? Wenn man Programme entwickelt, die nicht auf dem Einwicklungsrechner laufen sollen, stellt sich auch das Problem der Portabilität, denn viele Programmiersprachen sind so unterschiedlich auf verschiedene Maschinen implementiert (BASIC ist hier ein Beispiel), daß eine Portierung außer Klassess Beispield), außer einer entsprechenden Umstellung auf die andere Maschine keine anderen Kosten verursachen. So kann man seine Programme leicht auf andere Computer übertragen.

ould not foresee precisely the type of pro-
grams we might want to write, an effort was
made to keep the language [COMITI] general
purpose. (YNGVE 1963, 83). Ob die Progra-
miersprache darüber hinaus eine klassische
Universalsprache (wie etwa PL/I) oder eher
eine auf den Problemkreis beschränkte Spe-
zialsprache sein sollte, hängt noch von ande-
ren Faktoren ab, wie z. B. der Einmachern Be-
triebzarkteit, dem Wartungsauwand, der Ef-
fizienz, der Verträglichkeit mit anderen Pro-
grammen etc.

Geraade bei Problemen der CL Spielet —
wegen der oft vorhandenen inharanten Kom-
plexität der Algorithmen — die Effizienz des
Werkszeuges, d. h. der Programmiersprache,
eine entscheidende Rolle. Neben der Ausstuh-
lungseffizienz ist aber auch die Effizienz der
Programmstellenlängung (Anzahl der Progra-
mmzeichen zur Realisierung eines bestimmen Al-
goritmus, Programmierarbeitswand in Arbeitss-
zeit etc.) zu berücksichtigen, die eng mit der
Benutzererfahrung korrelierter ist.

Eine andere Frage ist die, wie universal
die verwendete Programmiersprache sein
sollte. Obne Frage sollte sie die zu bearbei-
tende Problemkasse gut abdecken. Ein frit-
hes Zitat zu diesem Bereich stammt vom Ent-
wickler von COMIT: „However, since we
were dealing with unsolved problems and

2.4. Benutzerfreundlichkeit, Universalität

nes Datenbankpakete. Letztere Moglichkeiten bieten kaum Integration in die Programme - sprache, die Daten des Programs und die des Datenbanksystems sind konzeptuell getrennt, andererseits hat der Benutzer auf diese Weise die volle Umsetzung durch ein Datenbanksystem mit allen seinen Zusatzfunktionen Moglichkeit (Wiederauflistungsbearbeitung, mehrfach-Schreibzugriff etc.).

wendung von logischen Namen als Operanden der Maschinenbefehle und mnemonische Namen für die Maschinenbefehle ermöglichten. Aber auch Assemblersprachen haben als großen Nachteil ihre starke Maschinenabhängigkeit und die relative Ferne von der Problemstellung. Ende der 50er Jahre tauchten dann die ersten höheren Programmiersprachen auf (die damals aber noch weit von unseren heutigen Vorstellungen von höheren Programmiersprachen entfernt waren); besonders bekannt sind aus dieser Zeit FORTRAN und ALGOL. Anfang der 60er Jahre beginnt eine 'Explosion' auf dem Gebiet der Programmiersprachen, davon ist auch der Bereich der Computerlinguistik betroffen. Die ersten einschlägigen Programmiersprachen (COMIT, COMIT II, SNOBOL) entstehen. Diese bieten eine auf die Bedürfnisse der CL zugeschnittene Benutzerschnittstelle, wenngleich sie noch vergleichsweise problemfern und maschinennah waren („COMIT is not a language, however, in which one 'programs in English'.“ (Yngve 1963, 83)). In Allgemein-Sprachen (im Gegensatz zu Spezialsprachen) wird die Bedeutung der Textverarbeitung zu dieser Zeit noch nicht gesehen. Die erste Einbeziehung von z. B. Datentypen für Zeichenkettenverarbeitung fällt in die Zeit der späten 60er, Anfang der 70er Jahre. Die bekanntesten Vertreter sind ALGOL 68 und PL/1. Weitere Allgemeinsprachen, die zumindest einen Datentyp 'String' kennen sind die Programmiersprachen BASIC (je nach Dialekt, nicht die Ur-Version von 1965) und UCSD-Pascal (eine an der Universität San Diego entwickelte Implementation von Pascal mit verschiedenen Erweiterungen gegenüber dem Jensen/Wirth-Standard). Auch diese beiden Sprachen kamen in der ersten Hälfte der 70er Jahre auf. Eine häufig zu beobachtende Technik, in der Spezialsprachen für die CL geschaffen werden, ist die, eine vorhandene Programmiersprache — z. B. ALGOL 60 — um spezielle Datentypen, Operationen und syntaktische Konstrukte zu erweitern und sich spezieller Präkompile zu bedienen, die von dem erweiterten Sprachstandard in die zugrundeliegende Basis-Sprache übersetzen. Mit Hilfe eines Satzes von Laufzeitprozeduren werden dabei die Operationen auf den mit der Erweiterung eingeführten Datentypen unterstützt. Ein typischer Vertreter für diesen Ansatz ist die Programmiersprache LINGUOL (Keil 1978, 1 ff.), eine ALGOL-Erweiterung. Auch FORTRAN diente häufig

als Ausgangspunkt für derartige Spracherweiterungen.

Eine von der in den prozeduralen Sprachen (wie ALGOL, COBOL, FORTRAN, PL/1, Pascal etc., vgl. 2.2.) verwendeten völlig abweichende Philosophie wird in der Programmiersprache LISP verfolgt, deren Anfänge in die späten 50er Jahre reichen. Diese Sprache lässt sich nicht in das Schema Spezial/Universal-Sprache einordnen, sie ist in gewisser Weise beides zugleich. Zum einen ermöglicht sie eine vergleichsweise komfortable Textverarbeitung. LISP hat sich besonders im Bereich der künstlichen Intelligenz etabliert und gilt dort noch heute als die am weitesten verbreitete Programmiersprachenfamilie.

Mit dem verstärkten Interesse an der künstlichen Intelligenz wurde auch dem Problem der Programmiersprachen seit Ende der 70er Jahre wieder zusätzliche Beachtung geschenkt. Wegen der nahen Verwandtschaft von Problemen der Computerlinguistik und denen der künstlichen Intelligenz wurde bei diesen Entwicklungen dem Gesichtspunkt der Textverarbeitung und der Verarbeitung allgemein dynamischer Strukturen besondere Aufmerksamkeit gewidmet.

3.3. Welche Programmiersprachen werden in der CL-Praxis verwendet?

Aus einer — für den deutschsprachigen Raum sicherlich repräsentativen — Aufstellung von 74 verschiedenen CL-Projekten (Petersen/Frackenpohl/Dvovak et al. 1982, 1 ff.) lässt sich entnehmen, daß in den beschriebenen Systemen die in der folgenden Statistik aufgeführten Programmiersprachen als Basis-Implementations-Sprache zum Einsatz kamen. Teilweise haben die dargestellten Systeme wiederum Programmiersprachen-Charakter, wie z. B. allgemeine Parser, die mit einer Grammatik als spezieller Form eines deklarativen Programmes versorgt werden können. Dieses höhere Niveau in der Programmiersprachenhierarchie bleibt in der folgenden Statistik unberücksichtigt.

Einsatz von Programmiersprachen in Projekten der Computerlinguistik:

Prozentuale Verteilung des Einsatzes der häufigsten Programmiersprachen in kombinierten (mehrere Programmiersprachen kamen zum Einsatz) und homogenen Systemen (nur eine verwendete Programmiersprache):
In Kombination:

FORTRAN	39 %
PL/1	34 %

Assembler	23 %	Bedeutung. Auf manchen UNIX-Rechnern ist sie als einzige hohere – sogenannte C-Programmiersprache vorhanden.
LISP	9 %	C-Programmiersprache zuerst – Programmiersprache von Alan Kay kann man davon aussehen, daß mitunter viele Micro-Computer-Anwendungen der Computer bestimmt die Wahl der Programmiersprache.
COBOL	4 %	Auch kann man davon aussehen, daß mitunter viele Micro-Computer-Anwendungen der Computer bestimmt die Wahl der Programmiersprache.
Sonstige	15 %	PL/I
FORTTRAN	27 %	PL/I
Assembler	26 %	Homogene Systeme:
LISP	7 %	PL/I
Sonstige	7 %	FORTTRAN
Assembler	7 %	Assembler besitzt bestimmt zu beeinflussen. In der LISP-Sprache, wenn eine der Sprachen bestimmt die Wahl der Programmiersprache.
LISP	7 %	LISP
Sonstige	7 %	(1) Die Frage der Portabilität und Verfügbilität scheint die Wahl der Programmiersprache bestimmt zu beeinflussen. In der LISP-Sprache, wenn eine der Sprachen bestimmt die Wahl der Programmiersprache.
Homogene Systeme:	15 %	(2) Domänant ist immer noch (Stand 1982) FORTTRAN, wenn auch nicht gezeigt von PL/I.
Assembler	27 %	(3) In 27 % aller Fälle wurden mindestens zwei verschiedene Programmiersprachen benötigt, um Textverarbeitung einzuführen. Wenn man darunter PL/I als Allroundsprache hat den Problem angegrapt.
LISP	7 %	(4) PL/I als Allroundsprache hat den Hauptsimplimentierungsprache mit herangezogen. In 16 % der Fälle wurde Assembler als hochstein Anteil an den homogenen Systemen. Die Übersicht war schließlich daran, daß in der PL/I das Allroundsprache mit vertrieben.
Sonstige	7 %	(5) Die ansonten recht weit verbreitete Programmiersprache BASIC ist nicht vertreten. Dies liegt wahrscheinlich daran, daß in der Übersicht aufgetragen wurde zuerst BASIC im Bereich der Computer-Großrechner und dann mit dem Vordringen von Datenbanken im heutigen Stammarten eines Beschäftigten auf die Ziffern Ziffern brachte, das in jeder Anwendung, die über eine einfache Konkordanz hinausgeht, — explizit oder implizit — Verwendunge gefunden. Auch vernezte Strukturen (ganz deutlich bei semantischen Netzzen oder Baumstrukturen zur Repräsentation und Graphenkonstrukten) werden durch das explizit untersetzt wurden (durch das Explizit-Pascal).
Assembler	26 %	(6) Über raschenderweise tritt Pascal gar einmitt. Nicht auf, obwohl es zum Zeitpunkt der Zwei-
LISP	7 %	sammenstellung dieser Statistik auch der Zwei-
Sonstige	7 %	betrachteten Maschinen relativ weit verbreitet war, insbesondere im Universitären Bereich, aus dem der größte Teil der dieser Statistik, aus dem der größte Teil der dieser Statistik, aus dem der größte Teil der dieser Statistik,
Homogene Systeme:	15 %	(7) Mit dem seit einiger Zeit zu beobachtenden Aufkommen des Betriebssystems UNIX gewinnt auch die damit eng verbundene Programmiersprache C zunehmend an Bedeutung.

Datenbank-Funktionen zu erzielen sind (dies gilt aber auch — sofern überhaupt möglich — für die meisten CL-Spezialsprachen, eine Ausnahme bildet Comskee mit seinem integrierten Datentyp 'Wörterbuch').

Die Bedürfnisse — wie auch der Gegenstand — der CL überlappen sich in weitem Maße mit denen der künstlichen Intelligenz; so ist es nicht verwunderlich, daß vielfach die gleichen Programmiersprachen und -systeme benutzt werden. In beiden Bereichen finden — neben den zuvor erwähnten Voraussetzungen — besonders Grammatiken und deklara-

tives Wissen Verwendung. Typische Vertreter aus dem einen und dem anderen Lager sind SNOBOL und PROLOG, beide enthalten die Möglichkeit, durch eine Grammatik gesteuerte Mustererkennungen ablaufen zu lassen.

5. Literatur (in Auswahl)

A. Colmerauer 1979 · A. Goldberg 1984 · K. Jensen/N. Wirth [1972] 1975 · G. Keil 1978 · J. Messerschmidt 1981 · J. E. Sammet 1969 · V. Yngve 1963.

Jan Messerschmidt/Günter Hotz
Saarbrücken (Bundesrepublik Deutschland)

64. Special Hardware and Future Development in Computer Architecture in View of Computational Linguistics

1. Introduction
2. Traditional Computer Systems
3. The LISP Machines
4. The LISP Machine Software
5. Symbolic Input/Output Requirements
6. New Programming Paradigms
7. New Machine Concepts
8. Literature (selected)

1. Introduction

1.1. Symbolic vs. Numerical Computing

Computing with symbols and complex symbolic structures rather than with numbers, arrays, strings, etc. entails special requirements not only to programming languages, but also to computer architecture and organization. Although *symbolic computing* is nearly as old as numerical computing it has always been overshadowed by the latter. Until recently symbolic computing has had little influence on computer design. With a wider recognition of the importance of natural language and knowledge processing the situation has started to change.

The aim of this paper is to explicate the *special requirements* that symbolic computing presents to computer design and programming language development and to review the various machine and programming concepts that can support these requirements. We will start by discussing the nature of symbolic computing and the gaps and limitations that inflict current technology and then proceed to review the major advances and directions of research in the field.

1.2. The Nature of Language and Knowledge

The special nature of natural languages and knowledge as symbolic systems require corresponding programming and machine concepts. The theories of language and knowledge are currently developing toward formalisms that consist of *complex systems* of features, rules, relations, objects, class hierarchies, etc. Powerful formal systems and abstraction mechanisms are developed and applied to representation and processing of language and knowledge structures.

From the point of view of systems theory languages and knowledge are *very large symbolic systems*. One needs only consider the number of features or categories used to describe various levels of language. If in the basic phonology a few dozen features suffice, the levels of morphology and syntax require hundreds of features. The number required to cope with the logico-semantic, pragmatic, and discourse components of language and with various blocks of special domain and world knowledge are not known, but seem to run into thousands.

From the computational point of view individual words, sentence structures, and thoughts can be considered as *abstract data types* each of which is unique. Thus natural language and knowledge involve potentially infinitely large polymorphic symbolic type systems. Moreover, on higher conceptual levels linguistic and domain or world knowledge cannot be clearly separated from each