Alte und neue Sprachkonzepte: Einstufung und Bewertung

Beitrag zur SVD-Tagung
"Ablösung der 3. Generationssprachen ?"

22. Juni 1990, Zürich

Jochen Ludewig
Institut für Informatik der Universität Stuttgart, BRD

Grundbegriffe der Programmiersprachen

Unter Sprache versteht man allgemein (vgl. Ludewig, 1989)

- das Gesprochene (also die Gesamtheit der Nachrichten, die via Mund und Ohr übertragen werden)
- das Gesprochene und alle (partiell) äquivalenten Darstellungsformen, also auch Schrift
- alle Konventionen zur Kommunikation, also beispielsweise auch Bild- und Formelsprachen

In der Informatik wird vor allem die letzte, allgemeinste Definition verwendet. Damit sind Programmiersprachen Sprachen im heute üblichen Sinne, also

Konventionen über Form und Bedeutung der Nachrichten,

mittels derer wir uns untereinander oder mit Maschinen verständigen. Die Konventionen werden insgesamt als **Grammatik** bezeichnet, die beiden Teile, die Form und Inhalt betreffen, als **Syntax** respektive **Semantik**.

Um die Kommunikation nutzbar zu machen, ist zusätzlich die **Pragmatik** notwendig, durch die Aussagen, hier also Programme, eine Verknüpfung mit realen Gegenständen erhalten. Dieser Aspekt ist bei den Programmiersprachen im allgemeinen schwach ausgeprägt, d.h. die Verknüpfung wird entweder durch die Menschen hergestellt oder "pragmatisch" realisiert, also ad hoc und auf niedrigem Abstraktionsniveau (man vergleiche die Behandlung der Ein- und Ausgabe in ALGOL60). Damit bleibt die Informatik nahe der Mathematik, die sich stets durch perfekte Trennung von der realen Welt die Pragmatik vom Leibe gehalten hat (man vergleiche Axiomensysteme, Tautologien usw.). Auch in den typischen Lehrbuchbeispielen der Informatik wird sie völlig unterschlagen. Bei speziellen Sprachen (sog. Kommandosprachen, Job-Control-Languages) und in der Ein-/Ausgabe tritt die Pragmatik aber in den Vordergrund.

Definitionen

Gegeben sei die Sprache S.

Syntax von S =
Festlegung der Menge aller in S zulässigen Aussagen.

Semantik von S =
Festlegung der den *zulässigen* Aussagen zugeordneten Bedeutungen. Syntaktisch falsche Sätze haben keine Semantik, auch für syntaktisch korrekte Sätze ist sie nicht immer definiert.

Pragmatik von S =
Verknüpfung der Semantik von S mit der realen Welt. (Aussagen über die *Wahrheit* eines Satzes gehören hierher!)

Beispiel: Dada-Gedicht (Kurt Schwitters: "Der Bahnhof", 1918)

Man hat eine Leiter zur Sonne gestellt	Sy Se ~~Pr~~
Die Sonne ist schwarz	Sy Se ~~Pr~~
Die Mühle blüht	Sy ~~Se Pr~~
...	
Du deiner dir dich	~~Sy Se Pr~~
Du deiner dich dir.	~~Sy Se Pr~~

Beispiel: die natürlichen Zahlen

Jede Zahl ist eine Sequenz von Ziffern (0..9), wobei die erste Ziffer nicht 0 ist. (Syntax)

Der Wert einer Zahl ist definiert als der Wert ihrer letzten Ziffer, vermehrt um den zehnfachen Wert der links davon stehenden Zahl, falls diese vorhanden ist. (Semantik)

Klassifikation von Programmiersprachen

Formale Klassifikationen

Wichtige Merkmale von Programmiersprachen sind,

a) wieviele Nachrichten möglich sind,

b) wie Form und Inhalt definiert sind (oder sein könnten),

c) welche Maschinen- und Lösungskonzepte der Sprache zugrundeliegen,

d) wie stark die Sprache durch die eingesetzten Rechner einerseits und durch die zu lösenden Probleme andererseits geprägt ist,

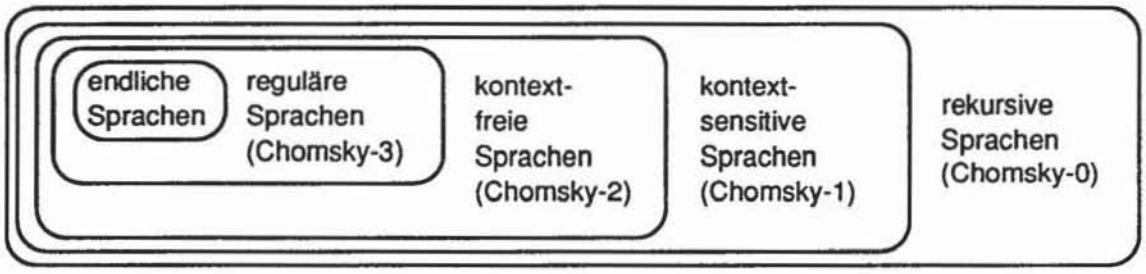
e) welches die markantesten Merkmale der Sprache sind,

f) welche Inhalte sich besonders einfach oder besonders exakt ausdrücken lassen (oder eben *nicht*)

g) wann die Sprache entstanden ist.

(a) führt zum Begriff der endlichen und der nicht-endlichen Sprachen; die üblichen Programmiersprachen sind sämtlich nicht-endlich, d.h. die Menge der verschiedenen möglichen Programme ist (im Prinzip) unbegrenzt.

(b) Formale Sprachen lassen sich durch Produktionssysteme definieren; je nach der Komplexität der zulässigen Produktionen unterscheidet man *reguläre, kontextfreie, kontextsensitive* und *allgemeine (rekursive)* Sprachen. Leider lassen sich Programmiersprachen nicht ohne weiteres in diese Klassifikation ordnen; vergröbernd läßt sich sagen, daß Programmiersprachen kontextsensitiv sind, daß sich aber weite Teile der Syntax als kontextfreie Sprache beschreiben lassen.



Sprachklassen von Chomsky (nach Ludewig, 1985)

(c) Bis heute werden nahezu sämtliche Programme auf Rechnern ausgeführt, die in der Tradition der von-Neumann-Maschine stehen. Es werden also die Befehle für einzelne Datenelemente sequentiell bearbeitet. Sprachen für Multiprozessoren und Vektorrechner spielen eine untergeordnete Rolle.

Viele neuere Sprach-Konzepte legen andere Lösungsmodelle zugrunde, obwohl die Programme letztlich doch auf von-Neumann-Rechnern bearbeitet werden. Hier sind vor allem die *funktionalen*, die *logischen* und die *objektorientierten* Programmiersprachen zu nennen.

(d) Intuitiv haben viele Menschen die Vorstellung, die Maschine sei primitiv und damit "niedrig", das Problem sei komplex und damit "hoch". Entsprechend entstanden auch die Charakterisierungen als *hohe* und *niedrige* Sprachen. Aber während man sicher sagen kann, daß Ada *über* Assembler liegt, ist die Reihung gerade in den interessanteren Fällen nicht möglich, da die exakten Kriterien für die "Höhe" fehlen. Daher gilt hier nach wie vor die Kritik von Parnas (1974).

(e) Die Merkmale sind zahlreich und natürlich überwiegend voneinander unabhängig. Beispiele sind *blockorientierte*, *formatfreie*, *standardisierte* Sprachen, Sprachen *mit strikter Typbindung* oder *mit Modulkonzept* usw.

(f) Auch bei Universalsprachen gibt es spezielle Möglichkeiten oder Beschränkungen. Beispielsweise ist Ada ein Sprache, die die Behandlung der Parallelität einschließt; die Programmierung verteilter Rechner ist dagegen in Ada nicht vorgesehen. Noch deutlicher ausgeprägt sind die Licht- und Schattenseiten der Sprachen natürlich bei den Spezialsprachen, beispielsweise COBOL als Sprache für den kommerziellen Bereich, PROLOG als Sprache für die logische Programmierung. Beide Sprachen eignen sich schlecht, um numerische Probleme zu bearbeiten, z.B. die numerische Integration einer Differentialgleichung.

(g) Wie jeder Mensch ist auch eine Programmiersprache stark geprägt durch ihre Entstehungszeit. Das bedeutet zunächst das Jahr, in dem die wichtigsten Konzepte der Sprache entstanden sind.

In der Terminologie des Marketings dominiert heute dagegen der - allerdings sehr unscharfe und oft irreführende - Generationen-Begriff, in dem Entstehungszeit und Sprachkonzept vermischt sind (siehe unten).

Der Generationenbegriff

Wie die meisten Wortschöpfungen des Marketings entzieht sich der Generationenbegriff einer sauberen Definition, selbst dicke Bücher, die sich mit diesem Thema befassen, bleiben in dieser Hinsicht stumm (vgl. Gutzwiller, Österle, 1988; Barth, 1987; Knolmayer, Disterer, 1987). Wir müssen daher den Begriff zum Wort entweder nach dem Sprachgebrauch formen (was zu Widersprüchen führt) oder neu schaffen (was notwendig mit dem Sprachgebrauch kollidiert). Hier soll das erste versucht werden.

Merkmale von 4GLs (nach Knolmayer, Disterer, 1987):

- 4GLs bieten mächtige Sprachkonstrukte
- 4GLs sind leicht erlernbar (für Laien)
- Editoren, Generatoren u.ä. sind Teil des Systems
- Die Programmierung erfolgt teilweise nichtprozedural
- Codierung und Ausführung können durch interaktives System gemischt werden (Interpreter)
- Datenbanksystem, Report-Generator im System enthalten
- Zugriffe auf DB, E/A und Kommunikationssystem mit einfachen Kommandos innerhalb der Sprache

Thesen zum Generationenbegriff

1. Der Generationenbegriff ist entstanden und wurde ganz überwiegendend angewendet auf Sprachen, die im sog. kommerziellen Bereich eingesetzt werden, noch schärfer: die mit COBOL konkurrieren; er wurde also eingeführt, um von "4th Generation Languages" (4GL) sprechen zu können. Damit ist eigentlich COBOL *per definitionem* die einzige Sprache der 3. Generation. Die vollständige Übersicht der gängigen Repräsentanten ist:

> 1. Generation: Maschinencode
> *Höhlenwohnung, aus dem vollen gegraben.*
>
> 2. Generation: Assembler
> *Hütte, aus Reisig und Lehm gebaut*
>
> 3. Generation: COBOL
> *Haus in konventioneller Bauweise ("Stein auf Stein")*
>
> 4. Generation: (Liste nach Knolmayer, Disterer, 1987)
> ADS-ONLINE, AS, DATAQUERY, DATATRIEVE, ES, FOCUS, GOGOL, IDEAL, MAPPER, MITROL, NATURAL, NOMAD 2, PINDAR, PISA/QL, POWERHOUSE, RAMIS II, SIRON, SPECTRA
> *Fertighaus, aus vorfabrizierten Komponenten*
>
> 5. Generation: PROLOG
> *Traumschloß, das sich automatisch entfaltet*

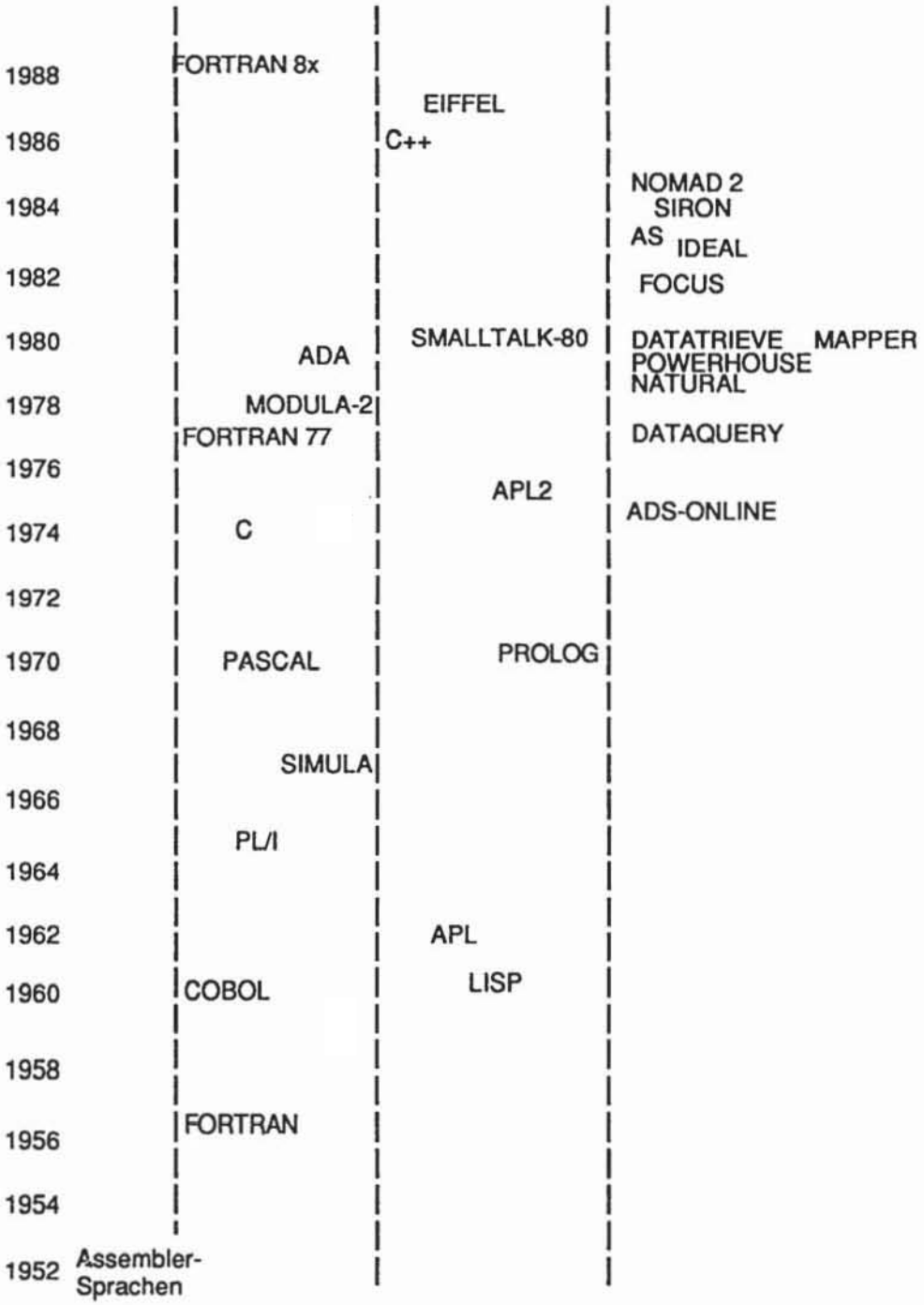
   Hiervon abweichende Definitionen haben sich nicht durchgesetzt; beispielsweise ist FORTH (verkürzt aus "FOURTH") nach üblichem Verständnis *keine*4GL.

2. Interpretiert man 4GL als Synonym zur imperativen Programmiersprache (FORTRAN, COBOL, PL/I, PASCAL, MODULA-2, ADA), so erweist sich der Generationen-Begriff als unsinnig.

   Das folgende Schema zeigt die Sprachen, geordnet nach Generationen.

V	PROLOG	LISP	SMALLTALK-80	SIMULA	C++	APL2	EIFFEL
IV	ADS-ONLINE	NOMAD 2	DATATRIEVE	IDEAL	FOCUS	AS	
	SIRON	POWERHOUSE	MAPPER	DATAQUERY	NATURAL		
III	FORTRAN	COBOL	PL/I	C	PASCAL	MODULA-2	ADA
II	Assembler-Sprachen						

Das nächste Bild stellt dagegen die Sprachen entsprechend ihrem Entstehungsdatum zusammen. Man sieht, daß von einer Generationenfolge keine Rede sein kann.



3. Die erste und fünfte Generation sind reine Dekoration, wirklich relevante Alternativen zu COBOL sind nur die Sprachen der 2., 3. und 4. Generation, also Assembler, Universalsprachen und 4GL.

4. Eine ernsthafte Berücksichtigung der modernen Hochsprachen (**"Universalsprachen"**) wie MODULA-2 und Ada (die meist als "weitere Sprachen der 3. Generation" beiseite geschoben werden) im Vergleich zu den 4GL (**"Generatorsprachen"**) hätte wesentlich andere Resultate beim Sprachvergleich zur Folge.

5. Eine 4GL ist tatsächlich keine Sprache, sondern ein System, geprägt durch ein Werkzeug, das Programme generiert, mit fertigen Komponenten verbindet und interpretiert. Ein Datenbanksystem gehört zu den wichtigsten Bestandteilen dieses Systems. Die Sprache ist meist dem Werkzeug pragmatisch angepaßt, nicht umgekehrt.

6. Mangelnde Standardisierung ist die große Schwäche der 4GL, und vermutlich ein wichtiger Grund für ihre Beliebtheit bei den Anbietern (4GL erzeugen Abhängigkeit).

7. Die angeblich höhere Erlernbarkeit der 4GL gilt offenbar nur für Laien. Informatikern erscheinen die Programmbeispiele mehr oder minder COBOL-ähnlich, also verbos und strukturarm. Von den Errungenschaften der modernen Universalsprachen (Block-, Prozedur- und Modul-Konzept, Strong Typing, Rekursion) wird kaum Gebrauch gemacht.

8. Generatorsprachen haben ihre Domäne bei der Lösung isolierter Probleme in bestimmten Bereichen durch Nicht-Fachleute, Universalsprachen beim Aufbau komplexer, langlebiger Systeme durch Experten.

9. Vergleiche zwischen Universalsprachen und Generatorsprachen können je nach Problemstellung jedes beliebige Resultat haben (vgl. Misra, Jalics, 1988; Verner, Tate, 1988). Ist die Aufgabe auf die 4GL zugeschnitten, so siegt diese klar; sind dagegen präzise Spezifikationen vorgegeben, so kann das Resultat auch umgekehrt ausfallen; der Einsatz einer 4GL entspricht dann dem Versuch, sich im Warenhaus einen Maßanzug zu kaufen.

Man kann diese Eigenschaften mit folgendem Bild anschaulich machen: Wer in einer Universalsprache programmiert, hat die Freiheit des Architekten am Zeichenbrett; im Rahmen der Möglichkeiten herkömmlicher Baustoffe ist alles erlaubt.

Die 4GL entspricht demgegenüber dem Katalog einer Fertighaus-Fabrik. Lösungen können aus vorhandenen Komponenten schnell und billig zusammengesetzt werden - oder sie sind nicht möglich. Die Stärken liegen also in Geschwindigkeit und Kosten von Standardlösungen, die Schwächen in der mangelnden Flexibilität bei speziellen Problemen, insbesondere auch bei der Erweiterung vorhandener Software-Systeme. (Auch ein vorhandenes Gebäude erweitert man in der Regel nicht mit Fertigteilen.)

Aspekte der Sprachbeurteilung

Firmen und Organisationen kommen nur selten in die Lage, Sprachen wirklich frei wählen zu können. In den meisten Fällen werden diese Situationen dann nicht einmal bemerkt, sondern durch Kontinuität (Festhalten am bekannten schlechten anstelle eines Wechsels zum unbekannten, möglicherweise auch schlechten) zugedeckt. Falls einmal eine Entscheidung zu fällen ist, dann kommen die folgenden Aspekte in Betracht:

- Produktivität
- Gebrauchsqualität der Programme (d.h. Qualität aus Kundensicht)
- Wartungsqualität der Programme (d.h. Qualität aus Sicht derjenigen, die am Produkt weiterarbeiten)
- Prozeß-Qualität (d.h. Qualität des Projekts, z.B. Einhaltung der Termin- und Kostenvorgaben, Transparenz)
- Investitionsschutz (Weiter- und Wiederverwendbarkeit, damit auch Kompatibilität und Standardisierung)
- Firmenkultur und Attraktivität der Arbeitsplätze

Wie wechselt man eine Programmiersprache ?

Kommt es tatsächlich zu einem Sprach-Wechsel, so sollte dieser nicht planlos (euphemistisch auch "organisch" genannt) erfolgen, sondern nach entsprechenden Vorbereitungen. Wie also wechselt man eine Programmiersprache?

1. so selten wie möglich!
2. nach einer rationalen Auseinandersetzung mit allen Beteiligten (möglichst auch über die irrationalen Aspekte, z.B. Ängste)
3. mit definierten Zielen, die meßbar sind, und in Erwartung der Umstellungsprobleme
4. mit einer Strategie auch für die vorhandene Software
5. nach vorhergehender Schulung aller Mitarbeiter, nicht nur auf der Ebene der Syntax (das kann man tatsächlich auch durch das "Beregnen" mit Handbüchern erreichen), sondern durch Schulung der neuen Konzepte.

Thesen

1. Die Einteilung der Programmiersprachen in Generationen ist rational nicht zu begründen und damit nutzlos; indem nicht zwischen Sprache, Übersetzer und Entwicklungsumgebung unterschieden wird, wächst vielmehr die Begriffsverwirrung.

Sinnvoller erscheinen stattdessen die Begriffe
- Universalsprache/Spezialsprache
- Programmbibliothek/Programm-Generator
- Compiler-Sprachen/Generator-Sprachen
  (anstelle von Sprachen der 3./4. Generation)

2. Da auch die traditionellen Compiler-Sprachen wie FORTRAN teilweise interpretativ bearbeitet wurden und umgekehrt bei den Generatorsprachen neben der interpretativen Verarbeitung auch eine gewisses Maß an Übersetzung notwendig ist, lassen sich diese Sprachtypen ebensowenig scharf trennen wie die prozeduralen von den nicht-prozeduralen Sprachen.

3. Generator-Sprachen sind solche Programmiersprachen, bei denen der Code überwiegend schon *vor* der Übersetzung eines speziellen Programms vorhanden ist, durch das Programm also weniger *definiert* als *parametrisiert* wird.

4. Generator-Sprachen haben aufgrund ihrer Geschichte typisch (aber nicht notwendig) die folgenden Merkmale:
   - Hintergrund nicht in Universitäten oder Forschungszentren, sondern in Software-Firmen
   - kaum Ähnlichkeiten mit blockorientierten Sprachen wie PASCALoder ADA, häufig Ähnlichkeit mit COBOL
   - Pragmatik voll in die Sprache eingebunden, dadurch weitgehende Integration traditioneller Problembereiche (z.B. E/A, Datenbanken usw.), aber auch starke Abhängigkeit von der System- und Software-Umgebung.

5. Generator-Sprachen stellen eine interessante Alternative zu den Compiler-Sprachen dar, wo
   - Standard-Probleme zu lösen sind (z.B. Entwicklung eines einfachen Informationssystems)
   - Datenbank-Inhalte ausgewertet und in Form von Tabellen, Graphik o.ä. dargestellt werden sollen
   - Unverbundene Anwendungen entwickelt werden (stand alone-Lösungen)
   - Ungelernte Programmierer (vgl. Ludewig, 1990) arbeiten

   Die Compiler-Sprachen werden dagegen kaum verdrängt werden, wenn
   - sehr große Systeme oder Systeme mit speziellen, sehr präzisen Anforderungen zu entwickeln sind,
   - bestehende Systeme zu erweitern sind,
   - Gelernte Informatiker programmieren,
   - Entwicklungsumgebungen für moderne Hochsprachen zur Verfügung stehen

Literatur

Barth, G. (1987): Zielsetzung, Definition und Klassifikation der 3. bis 5. Softwaregeneration. **Handbuch der modernen Datenverarbeitung Nr. 137**, Forkel-Verlag, p. 3-14.

Bolkart, W. (1989): **Programmiersprachen der 4. und 5. Generation**. McGraw-Hill, Hamburg (angekündigt).

Chorafas, D.N. (1987): **Forth and Fifth Generation Languages**. McGraw-Hill Book Company, New York usw.

Gutzwiller, Th., H. Österle (Hrsg.) (1988): **Anleitung zu einer praxisorientierten Software-Enwicklungsumgebung**. Band 2: Entwicklungssysteme und 4.-Generation-Sprachen. Angewandte InformationsTechnik Verlags GmbH, Halbergmoos.

Knolmayer, G., G. Disterer (1987): 4GL-Vergleich an einem Beispiel aus dem Berichtswesen. **Computer Magazin** 7/8/87, 41-47.

Ludewig, J. (1985): **Sprachen für die Programmierung - eine Übersicht.** BI-Hochschultaschenbuch Nr. 622, Bibliographisches Institut Mannheim.

Ludewig, J. (1989): **Skriptum Informatik**. vdf Zürich, 2. Aufl.

Martin, J. (1985): **Fourth-Generation Languages**. (Vol. I: Principles) Prentice Hall, Englewood Cliffs, NJ.

Martin, J., J. Leben (1986): **Fourth-Generation Languages**. (Vol. II: Representative 4GLs) Prentice Hall, Englewood Cliffs, NJ.

Misra, S.K., P.J. Jalics (1988): Third-generation versus fourth-generation software development. **IEEE Software**, July 1988, 8-14.
Briefe zu diesem Artikel und Antworten der Autoren in **IEEE Software**, November 1988, 6-11.

Parnas, D.L. (1974): On a 'buzzword': hierachical structure. **IFIP 74**, North-Holland Publishing Company, pp. 336-339.

Sebesta, R.W. (1989): **Concepts of Programming Languages**. Benjamin/Cummings, Redwood City, California, etc.

Sobell, M.G. (1988): **Programmiersprachen der vierten Generation, 4 GL: von der Datenbank zur Anwendung; am Beispiel IN-FORMIX 4GL**. Hanser-Verlag, München.

Verner, J.M., G. Tate (1988): Estimating size and effort in fourth-generation development. **IEEE Software**, July 1988, 15-22.