Geschichte der Programmiersprachen

Horst Zuse

Bericht 1999-1

Technische Universität Berlin
Fachbereich Informatik
FR 5-3
Franklinstraße 28/29
10587 Berlin
Tel. +49-30-314-24788
Fax: +49-30-314-73489
E-mail: zuse@cs.tu-berlin.de
Internet: http://www.cs.tu-berlin.de/~zuse

ISSN 1436-9915

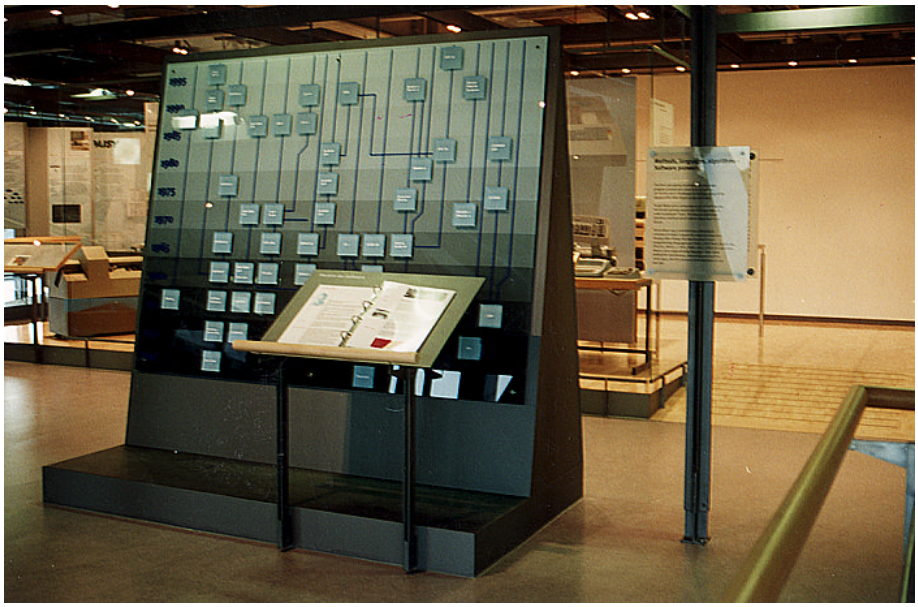
[#] [$] [+] [K] Inhaltsverzeichnis

[#] Inhaltsverzeichnis
[$] Inhaltsverzeichnis
[+] auto
[K] Inhaltsverzeichnis

[#] [$] [+] [K] Vorwort

Die Konstruktion der ersten programmgesteuerten Rechner (Prototypen des heutigen Computers) ist eng verbunden mit der Idee der Programmierung von Rechnern. Es ist heutzutage akzeptiert, daß die ersten Ideen der Programmierung von Babbage (1792-1871) /SWAD93/ im letzten Jahrhundert formuliert wurden. Aber es muß festgehalten werden, daß Konrad Zuse von 1942-1945/46 (Endfassung 1945/46) die erste Programmiersprache der Welt niedergeschrieben hat, welche er den Plankalkül /ZUSE45/ nannte.

Die vorliegende Ausarbeitung über die Entwicklung der Programmiersprachen wurde 1996 begonnen und immer wieder verfeinert. Ein Ergebnis dieser Ausarbeitung ist u.a. die Tafel im Heinz Nixdorf MuseumsForum in Paderborn auf dem folgendem Bild.



Ein Großbuch (vorne rechts) gibt eine stark verkürzte Fassung dieser Ausarbeitung wieder. Auch ist eine Vorversion dieser Ausarbeitung in /ZUSE98/ enthalten. Das obige Projekt wurde im Jahr 1996 mit Unterstützung von Herrn Nobert Ryska vom Heinz Nixdorf MuseumsForum realisiert.

Wir präsentieren hier eine Übersicht über die Entwicklung der Programmiersprachen und würdigen anschließend die wissenschaftlichen Leistungen von zwanzig Pionieren auf dem Gebiet der Programmiersprachen- und der Softwareentwicklung (Softwareengineering). Wir möchten betonen, daß selbstverständlich andere Pioniere auf dem Gebiet der Programmiersprachen und des Software Engineering große Verdienste erworben haben, wie z.B. F.L. Bauer und Heinz Zemaneck. Letztere und andere Pioniere werden z.B. in dem Büchern von J.A.N. Lee /Lee.95/ oder Dirk Siefkes /SIEF99/ gewürdigt. Eine Darstellung der Entwicklung der Programmiersprachen ist auch in /CERR98/ zu finden. Auf Seite 79 finden wir dort: *General Dwight D. Eisenhower, ca 1947: There will be no software in this man's army.*

[#] Vorwort
[$] Vorwort
[+] auto
[K] Vorwort

Es ist geplant, diese Ausarbeitung in Zukunft um weitere Pioniere der Programmiersprachen, aber auch der Softwareentwicklung, zu erweitern.

Horst Zuse, Berlin, Januar 2000

1 #+ $KEinleitung

Die ersten Ideen der Programmierung von Rechnern gehen zurück auf Charles Babbage. Charles Babbage (1792-1871) entwarf zwei Rechenmaschinen, die *Difference Machine* (1823) und die *Analytical Engine* (1834). Die Maschinen wurden niemals fertiggestellt. Die einzige arithmetische Operation, die ausgeführt werden sollte, war die Addition mit 27-stelligen Dezimalzahlen. Mit der Addition lassen sich eine Reihe von nützlichen Operationen ausführen. Die Technik dazu ist die *Methode der finiten Differenzen*. Babbage kannte noch keine Programmiersprache. Er dachte in einfachen Maschineninstruktionen. Konrad Zuse baute mit dem Rechner Z3 (1941) eine programmgesteuerte Rechenmaschine, basierend auf dem binären Zahlensystem (Gleitkommaarithmetik), die wirklich funktionierte. Die erste Programmiersprache der Welt für Rechner (Computer) wurde von 1942-1945 von Konrad Zuse /ZUSE45/ entwickelt. Er nannte sie den Plankalkül. Den Plankalkül wollte Konrad Zuse verwenden, um Probleme aus den Ingenieurwissenschaften, aber auch der Kombinatorik und Logistik zu formulieren /ZUSE98/. Der zweite Weltkrieg verhinderte eine Realisierung. dieser Ideen. Ab 1948 begann eine stürmische Entwicklung von Programmiersprachen für Computer.

Einleitung
+ auto
$ Einleitung
K Einleitung

2 #+$KEntwicklung der Programmiersprachen

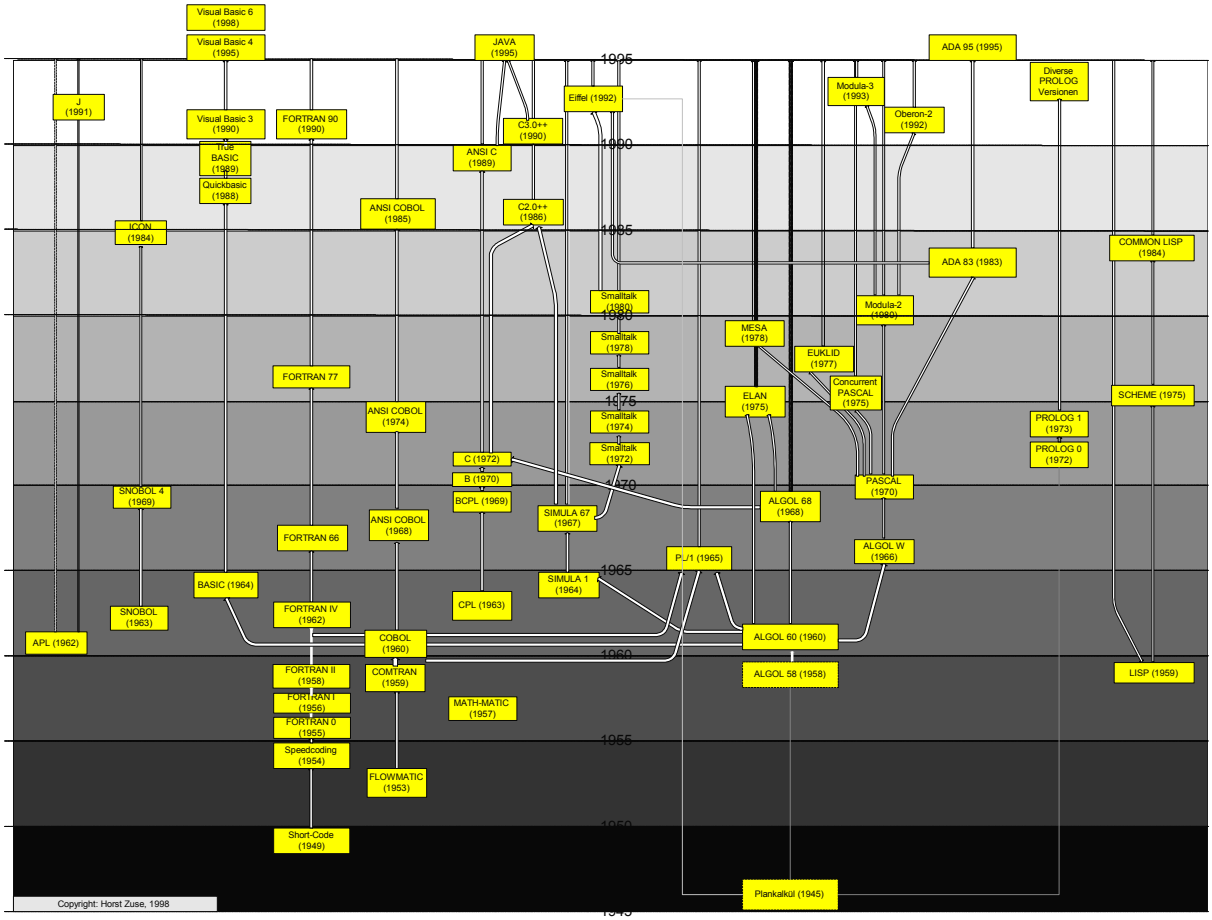
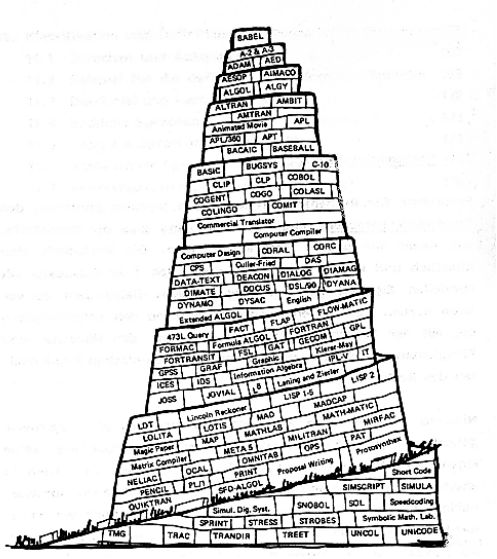


Abbildung 1: Evolution der Programmiersprachen.

Das obige Bild zeigt eine (grobe) Übersicht über die Entwicklung der Programmiersprachen. Gestrichelte Linien repräsentieren einen schwachen Einfluß auf andere Sprachen, wie z.B. bei ALGOL 68. Dünne Linien, wie vom Plankalkül zu ALGOL 58 und zu PROLOG dokumentieren Ähnlichkeiten zwischen den Sprachen. Der Plankalkül (1945) von Konrad Zuse hatte nur wenig bzw. gar keinen Einfluß auf ALGOL58 bzw. ALGOL 60, aber es wurde das Ergibtzeichen des Plankalkül diskutiert (Siehe dazu Peter Naur). Der Plankalkül hat Ähnlichkeiten mit PROLOG (Dünne Linien), aber auch eine Gemeinsamkeit mit EIFFEL (ASSERTION, siehe weiter unten). ALGOL68 wird wegen ihrer Kompliziertheit nur wenig genutzt, obwohl die Sprache einen starken Einfluß auf andere Sprachen hatte. Die gestrichelte Linie von ALGOL68 in Richtung Jahr 1995 zeigt, daß die Sprache nicht mehr aktuell ist. MATH-MATIC ist als eine separate Entwicklung zu sehen, die kaum einen Einfluß auf andere Sprachen hatte, obwohl eine grobe Ähnlichkeit zu FORTRAN 0 bestand.

[#] Entwicklung der Programmiersprachen
[+]auto
[$] Entwicklung der Programmiersprachen
[K] Entwicklung der Programmiersprachen

Der Plankalkül (1942-1945, Endfassung 1945/46) von Konrad Zuse, der schon 1941 mit der Z3 den ersten funktionierenden programmgesteuerten Rechner der Welt baute, gilt als die erste universelle Programmiersprache der Welt. Das Manuskript wurde von Konrad Zuse von 1942-1945/46 (Endfassung 1945/46) erstellt, aber der internationalen Öffentlichkeit erst 1972 bekannt. Der Plankalkül war aus heutiger Sicht eine bemerkenswert vollständige Programmiersprache, die u.a. Datenstrukturen enthielt, die sich erst 1968 bei der Entwicklung von ALGOL 68 wiederfanden. Der Plankalkül wurde niemals implementiert. Interessant ist auch eine Verbindung des Plankalkül mit der Sprache EIFFEL, und zwar das ASSERTION (*Assertion is a property of some of the values of program entities. For exanple, an assertion may express that a certain integer has a positive value or that a certain reference is void*).



Seit der Entwicklung des ersten Computers wurden ca. ein Tausend Programmiersprachen entwickelt, wie der *Turm von Babylon der Programmiersprachen* bezüglich der Programmiersprachen von Sammet /SAMM69/ illustriert.

Aus FLOW-MATIC, 1958, entstand unter entscheidender Mitwirkung von Grace Hopper die COBOL Sprachfamilie. Die Entwicklung von FORTRAN (Short-Code wurde von John Mauchly für die MARK I entwickelt), wurde wesentlich von John Backus initiiert. Allerdings entwickelte die Gruppe von Grace Hopper unter der Leitung von Charles Katz 1957 die Sprache MATH-MATIC, die ähnliche mathematische Sprachelemente aufwies. Ein wesentlicher Beitrag zu der Entwicklung und Struktur von Programmiersprachen fanden sich in ALGOL 60 (1960), das auf den Vorläufer ALGOL 58 von 1958 zurückgeht. Wissenschaftler, wie z.B. Peter Naur, Friedrich Bauer, Alan Perlis, Charles Hoare und Klaus Samuelsen haben wichtige Beiträge auf den ACM-GAMM (Association for Computing Machinery-Gesellschaft für Mathematik und Mechanik) dazu geleistet. Diese Entwicklung fand statt in Wechselbeziehung mit der Fortentwicklung der funktionellen Fähigkeiten der Rechenanlagen (symbolische Adresse von Wilkes 1953, indirekte Adressierung von Schecher 1955) und unter dem Eindruck der Probleme der maschinellen Übersetzung von algorithmischen Sprachen in Maschinensprachen (Rutishauser 1951, Bauer und Samuelson 1957). Rutishauser baute insbesondere auf dem *Plankalkül* von Konrad. Zuse (1948) auf. In diese Richtung war mit den Listen (Sprache LISP) von McCarthy (1959) und den Verbunden von Hoare (1966) der heutige Stand erreicht. Andere wesentliche Beiträge zu einem einheitlichen und vollständigen begrifflichen System der Programmierung oder, wie man es auch ausdrückt, zur Schaffung einer semantischen Basis, leisteten J.Green 1959 (Namenserzeugung), K.Samuelson 1959 (Blockstruktur), und N.Wirth 1965 (Referenzkonzept und die Sprachen ALGOLW, PASCAL, usw.). Allgemeine Fragen der Semantik behandelten McCarthy 1962, Landin 1965, Floyd 1967, und beginnend 1962, van Wijngaarden, dessen Untersuchungen über eine operative Semantik zu ALGOL 68 führten.

Aus der Sprache ALGOL entstanden viele andere imperative Sprachen, wie ALGOLW, PASCAL, SIMULA, ALGOL68, PL/1, C, und besonders ADA. ADA wurde als eine moderne und leistungsfähige Hochsprache von DoD (*Department of Defense der USA*) konzipiert. Es ist keine Frage, daß in ADA viele Zielvorstellungen des modernen Software Engineering im

Compiler enthalten sind, aber kritische Stimmen sind nicht ausgeblieben. Die Sprache EUCLID (1976-1977), entwickelt von der Universität Toronto, erweitert PASCAL u.a. mit abstrakten Datentypen. MESA (1976-1979) wurde vom Xerox Palo Alto Research Center entwickelt. Es erweitert PASCAL mit einer speziellen Modul-Einheit und Ausnahmebehandlungen, weches erlaubt, es in Betriebssystemen einzusetzen. Die Sprache ELAN wurde an der TU-Berlin im Rahmen des Forschungsvorhabens *Schulsprache* entwickelt. Das Projekt wurde bis 1977 von der DFG gefördert /HOMM79/. ELAN ist eine Sprache, die zur ALGOL60 Familie gehört, es sind aber mehr Ideen von ALGOL68 (nicht das Referenzkonzept) als von PASCAL eingegeangen. ELAN hatte Stärken in Hinsicht auf Lesbarkeit, Sicherheit und Ausdrucksmöglichkeit.

Bei Sprachen wie PROLOG (Kowalski) wird bewußt auf eine kontrollflußgesteuerte Auswertung verzichtet, wie dies bei prozeduralen Programmiersprachen (z.B. ALGOL, PL/1, usw.) der Fall ist. Ein PROLOG Programm besteht z.B. aus einer Folge von bedingten Regeln. Als die erste Sprache für Anwendungen in der künstlichen Intelligenz muß die Sprache LISP von McCarthy angesehen werden, aus der von 1975 bis 1985 die Sprachen SCHEME und COMMON-LISP entwickelt wurden. Es muß aber auch erwähnt werden, daß Konrad Zuse mit seinem Plankalkül typische Anwendungen der künstlichen Intelligenz betrachtete, nämlich das Schachspiel, und er dafür auch Programme schrieb.

Eine Art Außenseiter bei der Entwicklung von Programmiersprachen ist APL, welches schon vor 1960 von K. Iverson entwickelt wurde. Die Sprache basiert auf Matrizen, Symbolen und Skalaren. Eine erhebliche Erweiterung dieser Sprache wurde mit J 1991 von K. Iverson et. al. vorgestellt.

SNOBOL wurde in den frühen 60-ziger Jahren von Farber, Griswold und Plensky entwickelt. Idee war eine Sprache und einen Compiler für mächtige Zeichenkettenverarbeitung zu entwickeln. Die Sprache SNOBOL ist heutzutage nicht mehr weit verbreitet.

Die Sprache BASIC, entwickelt von Kemeny und Kurtz, hat eine erstaunliche Verbreitung gefunden. Die Einfachheit von BASIC kommt allen jenen Menschen entgegen, die den Computer zwar verwenden, aber nicht das Programmieren lernen möchten. BASIC hat durch QUICKBASIC, aus dem Visual BASIC3 (1990), Visual BASIC4 (1994), Visual BASIC5 (1997) und Visual BASIC6 (1998) von Microsoft folgten, eine bemerkenswerte Renaissance erlebt. Visual BASIC wurde u.a. für die Konzeption und Entwicklung von WINDOW Benutzer Schnittstellen (Interfaces) entwickelt.

Der Bereich interaktiver und kommunizierender Systeme, der im gewissen Sinne auch Sprachen für konkurrierende Systeme im einen Extrem und reine Abfragesprachen im anderen Extrem umfaßt, hat sicherlich seine stärkste Entwicklung noch vor sich. Wichtige Meilensteine in dieser Entwicklung stellen wohl Sprachen wie SIMULA67, und objektorientierte Sprachen wie SMALLTALK und Sprachen für die Büroautomation dar.

Die Geschichte der objektorientierten Programmierung begann schon in den Siebziger-Jahren mit der Programmiersprache SIMULA, einer Spezialsprache zur Simulation von Vorgängen in der realen Welt. Obwohl mit SIMULA bereits die Begriffe Klasse und Objekt eingeführt wurden, sprach damals noch kein Mensch von objektorientierter Programmierung. Das änderte sich mit SMALLTALK, der ersten rein objektorientierten Sprache. SMALLTALK gilt wegen der reinen Objektorientiertheit auch heute noch als die objektorientierte Programmiersprache an sich. Das ist unter anderem daran erkennbar, daß die heute übliche

Terminologie fast ausschließlich aus der SMALLTALK-Literatur stammt und daß sich fast alle neueren objektorientierten Sprachen in der einen oder anderen Weise an SMALLTALK anlehnen. Die meisten objektorientierten Sprachen sind blockstrukturiert und haben daher letztlich ihren Ursprung in der Sprache ALGOL. JAVA ist eine objekt-orientierte Sprache, welche sich aus C++ ableitete. JAVA kennt aber nicht das Pointerprinzip von C++ und verfügt eine standardisierte Beschreibung einer visuellen Schnittstelle für alle Plattformen von Betriebssystemen.

Auf PASCAL als die heute noch wichtigste strukturierte Programmiersprache folgte MODULA-2 als modulorientierte Sprache. Aus MODULA-2 ging wiederum OBERON hervor, das 1991 zu OBERON-2, einer echten objektorientierten Sprache ausgebaut wurde. Object-PASCAL ist eine objektorientierte Erweiterung von PASCAL. Von der Sprache C stammen zwei objektorientierte Sprachen ab: C++ und Objective C. Objective C lehnt sich stärker an SMALLTALK an als C++, das viele eigenständige Sprachelemente enthält, die man in anderen objektorientierten Sprachen nicht findet.

Noch heute werden u.a. LISP, FORTRAN und COBOL, die zu den ältesten Programmiersprachen zählen, im wissenschaftlichen und kaufmännischen Bereich verwendet. Es stellt sich die Frage: Was sind die Programmiersprachen der Zukunft? Die vor nicht allzulanger Zeit im Rahmen der Anstrengungen des US-Verteidigungsministeriums geäußerte forsche Einschätzung: *ADA ist wohl die letzte Programmiersprache*, ist sicherlich im Sinn der historisch letzten Programmiersprache heute als falsch erkennbar. Ob ADA aus PASCAL entstand, wird kontrovers diskutiert. ADA ist eine imperative Sprache, wie z.B. ALGOL60, enthält aber das Konzept der abstrakten Datentypen, die Vererbung und das Klassenkonzept. ADA erleichtert die modulare Erstellung von Modulen (Package Konzept).

Prinzipiell muß der Hoffnung auf die eine allen Anwendungen gerecht werdende Programmiersprache wohl mit großer Skepsis begegnet werden. Jüngste Forschungen zeigen die Vielfalt von programmiersprachlichen Konzepten. Eine Einbeziehung all dieser Konzepte in eine Programmiersprache scheint weder möglich noch wünschenswert. Was vielmehr benötigt wird, ist eine Familie von schlanken Sprachen mit klaren semantischen Konzepten für die unterschiedlichen Anforderungen.

Sprachen der Zukunft werden sich u.a. dadurch auszeichnen, daß der Programmierer oder Anwender formulieren kann, was er gelöst haben möchte. Er wird die Problemlösung nicht formulieren müssen. Dieser Ansatz wird vor allem von den Datenbank-Abfrage-Sprachen genutzt. Bekannte Vertreter dieser Sprachen der vierten Generation sind u.a. ADF, FOCUS, MANTIS und NATURAL. Diese Sprachen werden aber FORTRAN, COBOL und ADA nicht ersetzen, sondern ergänzen.

3 [#] [$] [+] [K]Der Begriff des Software Engineering

Was verbirgt sich hinter dem Begriff *Software Engineering?* Ist das nur ein neuer Name für Programmierung oder eine neue Technologie? Der Begriff Software Engineering war zu der Zeit, in der er geprägt wurde, wohl provokatorisch gemeint und sollte anzeigen, daß auch die Herstellung von Software den Charakter einer *Ingenieurdisziplin* aufweisen muß. Die Ursache für den Begriff Software Engineering ist wohl aus dem Begriff *Softwarekrise* abgeleitet. Mitte der sechziger Jahre führte die Herstellung von Software zu immer größeren Schwierigkeiten. Es fehlte an theoretischen Grundlagen und methodischen Hilfsmitteln. Für die Softwarekrise gibt es in der Hardware keine Entsprechung, weshalb man auch von einem Hardwarewunder sprechen kann. Das bedeutet nicht, daß die Hardware stets fehlerfrei ist (Der Fehler im PENTIUM 90 Prozessor im Jahre 1994 ist ein populäres Beispiel dafür), aber die Praxis zeigte, daß der Grund der Schwierigkeiten beim Einsatz von Computersystemen meist in der fehlerhaften Software zu suchen ist. Die durch neue Rechnergenerationen geschaffenen Möglichkeiten übertrafen bei weitem die bis dahin entwickelten Programmiertechniken. Aber die wachsende ökonomische Bedeutung der Softwareherstellung und die enorme Expansion der Datenverarbeitungsbranche rückten die Forderung nach einer verbesserten *Programmiertechnologie* immer stärker in den Mittelpunkt des Forschungsinteresses der Informatik.

Die Anstrengungen zur Entwicklung einer verbesserten Programmiertechnologie fanden in der Forschung ihren ersten Niederschlag in den beiden von der NATO veranstalteten Software Engineering-Konferenzen in Garmisch 1968 und in Rom 1969. Auf diesen Konferenzen wurde darauf hingewiesen, daß Programme Industrieprodukte sind, und deshalb wurde die Forderung erhoben: Abkehr von der *Kunst* des Programmierens und Hinwendung zu einer ingenieurmäßigen Software-Entwicklung. Seither wird versucht, die Softwareherstellung als zusammenhängenden Prozeß wissenschaftlich zu durchdringen und vor allem Fragen der Spezifikation, des methodischen Programmentwurfs, der Strukturierung von Softwaresystemen, der Wiederverwendbarkeit von Softwarebausteinen, der Projektorganisation, der Qualitätssicherung, der Dokumentation, der Anforderungen an Werkzeuge zur Unterstützung der Softwareentwicklung und der automatischen Softwareherstellung in den Mittelpunkt des Forschungsinteresses zu rücken. Obwohl die Prägung des Begriffes *Software Engineering* nun schon mehr als 25 Jahre zurückliegt, gibt es aber noch keine allgemein anerkannte, feststehende Definition dieses Begriffs. Wir geben hier einige Beispiele:

- Boehm definiert in /BOEH79/ Software Engineering folgendermaßen: *Software Engineering ist die praktische Anwendung wissenschaftlicher Erkenntnisse auf den Entwurf und die Konstruktion von Computerprogrammen, verbunden mit der Dokumentation, die zur Entwicklung, Benutzung und Wartung der Programme erforderlich ist.*

[#] Software Engineering
[$] Software Engineering
[+] auto
[K] Software Engineering

- Bei Dennis findet man in /DENN75/ die Definition: *Software Engineering ist die Anwendung von Prinzipien, Fähigkeiten und Kunstfertigkeiten auf den Entwurf und die Konstruktion von Programmsystemen.*

- Parnas schreibt in /PARN74/: *Software Engineering ist die Programmierung unter mindestens einer der zwei Bedingungen: Mehr als eine Person schreibt und benutzt das Programm. Mehr als eine Fassung des Programms wird erzeugt.*

- Bei Fairley findet man in /FAIR85/ die Definition: *Software Engineering ist die technische und organisatorische Disziplin zur systematischen Herstellung und Wartung von Softwareprodukten, die zeitgerecht und innerhalb vorgegebener Kostenschranken hergestellt und modifiziert werden.*

- Sommrville schreibt in /SOMM85/: *Software Engineering befaßt sich mit dem Bau von Softwaresystemen, die nicht von einem Entwickler alleine hergestellt werden können. Software Engineering beruht auf der Anwendung ingeniermäßiger Prinzipien und umfaßt sowohl technische als auch nicht-technische Aspekte.*

- Bauer schließlich schreibt in /BAUE75/: *Das Ziel der Softwaretechnik ist die wirtschaftliche Herstellung zuverlässiger und effizienter Software.*

- Bei Pomberger et al. /POMB93/ finden wir die folgende Definition: *Software Engineering ist die praktische Anwendung wissenschaftlicher Erkenntnisse für die wirtschaftliche Herstellung und den wirtschaftlichen Einsatz qualitativ hochwertiger Software.*

Trotz erheblich verbesserter Methoden der Softwareentwicklung haben sich die anteiligen Wartungskosten der Produkte an den Gesamtkosten stark erhöht, wie die nächste Abbildung zeigt.

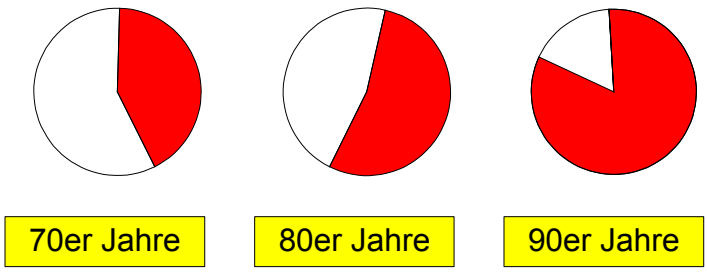
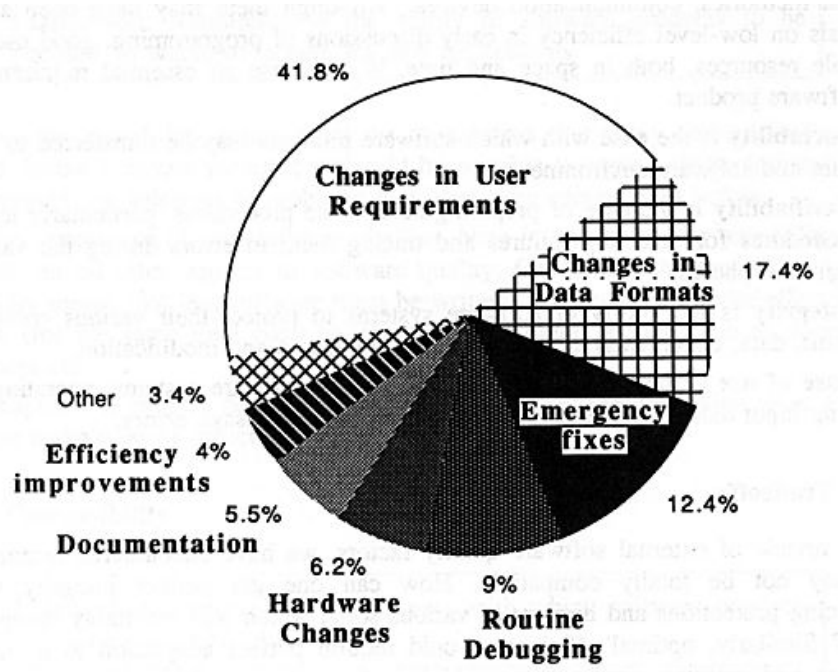


Abbildung 2: Anteil der Wartung an den Entwicklungskosten. In den 70er Jahren waren es ca. 35-40%, in den 80er Jahren ca. 4-60% und in den 90er Jahren sind es ca. 70-80%.

Der Term Softwarewartung

Der Begriff Wartung von Software muß näher erläutert werden. Die nähere Betrachtung zeigt, daß dieser Term eine Fehlbezeichnung ist. Wartung von Software ist nicht das Gleiche wie die Wartung eines Automobils nach z.B. 40000 km Fahrleistung. Software verschleißt nicht durch häufige Benutzung. Eine Untersuchung von 487 Installationen zeigt, wo die Ursachen der Wartungskosten (1980) liegen /MEYE88/.



Mehr als 2/3 der Wartungskosten beziehen sich auf Benutzeranforderungen, die Erweiterungen und Modifikationen des Softwareproduktes nach sich ziehen. Interessant sind die relative niedrigen Kosten für die Dokumentation (5%).

Das Ziel ist qualitative hochwertige Software zu erstellen. Dazu müssen wir interne und externe Qualitätsfaktoren betrachten. Die wichtigsten externen Qualitätsfaktoren sind nach /MEYE88/:

- ***Correctness*** *is the ability of software products to exactly perform their tasks, as defined by the requirements and specification.*
- ***Robustness*** *is the ability of software systems to function even in abnormal conditions.*
- ***Extendibility****: Extendibility is the ease with which software products may be adapted to changes of specification.*
- ***Reuseability*** *is the ability of software products to be reused, in whole or in part, for new application.*
- ***Compatibility*** *is the ease with which software products may be combined with others.*

Die Entwicklung neuer Paradigm, wie z.B. objektorientierte Programmierung oder quantitative Methoden des Software Engineering (Softwaremetrie) können u.U. helfen, den Anteil der Wartungskosten zu senken. Voraussetzung aller dieser Maßnahmen ist aber ein Verständnis darüber, was Programme für den Menschen schwer verständlich bzw. komplex macht.

Ein anderer Aspekt zur Bewertung von Software ist die Analyse der Verstehbarkeit der Konzepte und Konstrukte von Programmiersprachen und der geschriebenen Programme. Hier können Methoden des quantitativen und empirischen Software Engineering hilfreich sein. Dies ist z.B. die Analyse der Programmkomplexität bzw. Programmlesbar- und Verstehbarkeit mit Softwaremaßen /ZUSE97/.

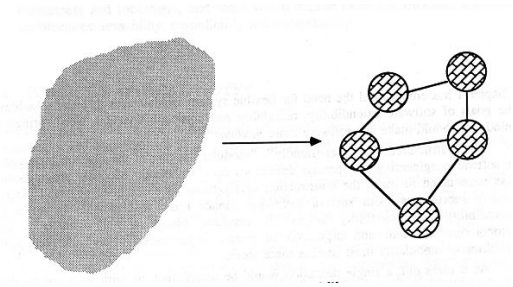
4 [#] [$+] [K] Modularität

Wir benötigen flexible Softwarearchitekturen, um z.B. Ziele wie *extendbility, reuseability* oder *compatablity* erreichen zu können. Das Schlagwort Modularisierung wird hier gerne verwendet, um die Problemlösung aufzuzeigen. Eine präzise Definition des Wortes Modul existiert nicht. Wir müssen die Modularisierung von Software aus verschiedenen Sichten betrachten. Oft wird darunter die Zerlegung von großen Programmen in kleinere Einheiten verstanden, die Module genannt werden. Der Term Modul ist aber abhängig von der verwendeten Programmiersprache bzw. dem Konzept.

Es ist daher notwendig, den Begriff modular näher zu erläutern. Uns erscheint der Vorschlag von Meyer /MEYE88/ sehr sinnvoll. Meyer schlägt fünf Kriterien für Modularität vor, die wir hier kurz diskutieren wollen.

Modulare Zerlegung (Modular Decomposability)

Dies ist die Reduzierung der Komplexität eines Softwaresystems in kleine Einheiten.



Modular Composeability (Modulare Zusammensetzung)

Dies bezeichnet die Möglichkeit Softwareelemente zu erstellen, die frei aus bestehenden Softwareelementen konstruiert werden können.

Modulare Verstehbarkeit (Modular Understandability)

Dies ist die Konstruktion von Modulen, welche leicht durch den Leser verstanden werden können. Dieses Kriterium ist besonders wichtig für die Softwarewartung. Hier werden auch Softwaremetriken eingesetzt, um die Verstehbarkeit von Modulen in Zahlen abzubilden.

Modulare Kontinuität (Modular Continuity)

Diese Eigenschaft beinhaltet, daß eine kleine Änderungen in einer Problemspezifikation nur eine kleine Modifikationen in einem oder sehr wenigen Modulen zur Folge. Solche Änderungen sollten nicht die Architektur des Systems beeinflussen (Beziehung zwischen den Modulen). Dieser Begriff ist aus der Mathematik abgeleitet, wo wir von stetigen Funktionen

---

[#] **Modularität**
[$] **Modularität**
[+] auto
[K] **Modularität**

sprechen. Grob gesprochen: eine Funktion ist stetig wenn eine kleine Änderung im Argument eine kleine Änderung im Ergebnis verursacht.

Modularer Schutz (Modular Protection)

Diese Methode unterstützt das Prinzip, daß eine abnormale Bedingung in einem Module, die in der Ausführungszeit entsteht, in dem betreffenden Module behandelt wird. In PL/1 kann dies durch die Ausnahmebehandlungen zu Problemen führen. In ADA und CLU ist dies mehr restriktiv gelöst.

Seit ca. 1970 werden die obigen Konzepte an verstehbare Programme auch bei Programmiersprachen berücksichtigt. Das objekt-orientirte Konzept ist ein solcher Weg. Die funktionale Programmierung ist ein anderes solches Konzept.

5 [#] [$] [+] [K] Das Objekt-Orientierte Paradigma

Das prinzipielle Ziel des Software Engineering ist qualitative hochwertige Software zu erstellen. Bis Anfang der achtziger Jahre wurde die funktionelle Zerlegung von großen Programmen in kleine Einheiten (Module) propagiert, um kleine unabhängige Programme zu erhalten. Dies wurde mit Sprachen wie FORTRAN, PASCAL, PL/1 usw. realisiert. Wir nennen dies die funktionale Zerlegung (decomposition) von Programmen. Das objekt-orientierte Paradigma verfolgt einen völlig anderen Weg. Wir wollen hier einige Grundprinzipien des objekt-orientierten Paradigmas erläutern.

Objekt-Orientierter Entwurf (Object-Oriented Design)

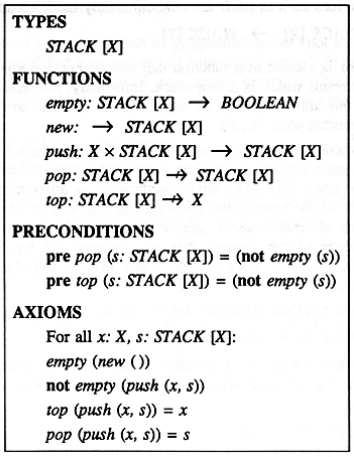
Während die funktionsorientierten Programmiersprachen (FORTRAN, PL/1) primär den Algorithmus betrachteten und die Daten (Variablen) um den Algrithmus plazierten, verfolgt der objekt-orientierte Ansatz vordergründig die Daten, deren Strukturen, die Einbettung der Daten und die Operationen mit diesen Daten. Eine Definition des objekt-orientierten Entwurfs ist nach /MEYE88/:

Definition: *Object-Oriented Design is the method which leads to software architectures based on objects every system or subsystem manipulates (rather than the function it is meant to ensure).*

Wir betrachten im objekt-orientierten Ansatz nicht nur individuelle Datenstrukturen, wie diese auch schon z.B. in COBOL und PL/1 existierten. Wir betrachten *Klassen* von Datenstrukturen. Der Term Klasse ist in der Tat der Begriff, welcher in dem objekt-orientierten Paradigma eingeführt wird, welcher Datenstrukturen durch (gemeinsame) Eigenschaften beschreibt. Datenstrukturen (hierarchisch) sind auch in Sprachen wie PL/1 oder sogar Visual Basic (TYPE) schon bekannt. Mit diesen Datenstrukturen konnte z.B. die Programmierung eines STACKS mit den Operationen *push, pull*, usw. realisiert werden.

Was wir aber wollen, ist nicht eine spezielle Implementierung eines Stacks, sondern eine abstrakte Beschreibung von Datenstrukturen und deren Operationen. Dies führt uns zu den abstrakten Datentypen. Die Theorie der abstrakten Datentypen beschreibt eine Klasse von Datenstrukturen nicht durch die Implementierung, sondern durch eine externe Sicht. Wir demonstrieren dies mit dem Beispiel des STACK. Die Spezifikation besteht aus vier Teilen: *Types, Functions, Preconditions* und *Axioms*.

[#] Das Objekt-Orientierte Paradigma
[$] Das Objekt-Orientierte Paradigma
[+] auto
[K] Das Objekt-Orientierte Paradigma



TYPES
STACK [X]

FUNCTIONS
empty: STACK [X] → BOOLEAN
new: → STACK [X]
push: X × STACK [X] → STACK [X]
pop: STACK [X] ↛ STACK [X]
top: STACK [X] ↛ X

PRECONDITIONS
pre *pop* (*s: STACK* [*X*]) = (**not** *empty* (*s*))
pre *top* (*s: STACK* [*X*]) = (**not** *empty* (*s*))

AXIOMS
For all *x: X, s: STACK* [*X*]:
empty (*new* ())
not *empty* (*push* (*x, s*))
top (*push* (*x, s*)) = *x*
pop (*push* (*x, s*)) = *s*

Eine Spezifikation für einen abstrakten Datentyp besteht aus vier Teilen: den Typen (types), den Funktionen (functions), den Vorbedingungen (preconditions) und den Axiomen (axioms).

Der Typeparagraph (Types Paragraph)

Der Typeparagraph listet die Typen der Spezifikation auf. In diesem Beispiel ist nur einer aufgelistet, und zwar STACK(X). Dieser spezielle Typ ist als STACK(X) eingeführt. Es ist ein generischer abstrakter Datentyp mit dem Dummyparameter X. Hier wird die Generizität benutzt (genericity). Der Sinn ist, daß nicht für verschiedene Datentypen wie *integer, float*, lexikalische Ausdrücke, usw. viele ähnliche Spezifikationen geschrieben werden müssen.

Funktionen (Functions)

Die Funktionen listen die Dienste (services) auf. Diese sind im obigen Fall als mathematische Funktionen spezifiziert: *empty, new, push, pop* und *top*. Jede Zeile in diesem Paragraphen ist die Beschreibung einer Funktion mit ihren Argumenten und Ergebnissen:

A1 x A2 x ... x Am -> B1 x B2 x ... x Bn.

Z.B. ist die Funktion *push* spezifiziert als:

Push: X x STACK(X) -> STACK(X).

Vorbedingungen (Preconditions)

Z.B. bedeutet die folgende Vorbedingung

pre pop (S: STACK(X)) = (not empty (s))

folgendes: Die Vorbedingung für *pop*, die erfüllt sein muß, steht auf der rechten Seite. Es bedeutet, daß s nicht-leer sein muß.

Axiome (Axioms)

Die ersten beiden Axiome beschreiben die Eigenschaften der empty-Funktion. Ein neuer STACK is leer und irgendein STACK, in den ein Element hineingetan wird, ist nicht-leer. Die letzten beiden Axiome beschreiben die last-in-first-out Eigenschaften eines Stacks.

Objekt-Oriented Design

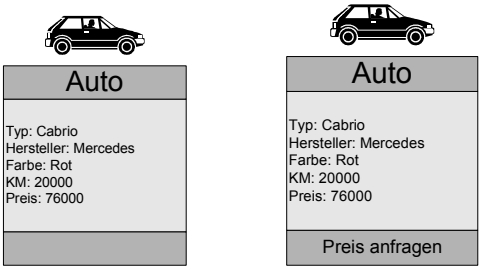
Wir können jetzt das objekt-orientierte Paradigma genauer fassen /MEYE88/.

Definition (MEYE88/:
Object-oriented design is the construktion of software system as structured collections of abstract data type implementation.
*
Der wichtige Aspekt bei der objekt-orientierten Programmierung sind die Zusammenfassung der Daten zu logischen Einheiten und die Operationen auf diesen. Dies führt zu den Begriffen wie Verkapselung (Encapsulation), Polymorhismus und Generizität (Genericity). Ein weiterer sehr wichtiger Aspekt ist die Vererbung (Inheritance). Es bedeutet, daß neue Datenstrukturen, die in Klassen zusammengefaßt sind, erweitert oder reduziert werden können, ohne die Ursprungsklasse zu modifizieren.

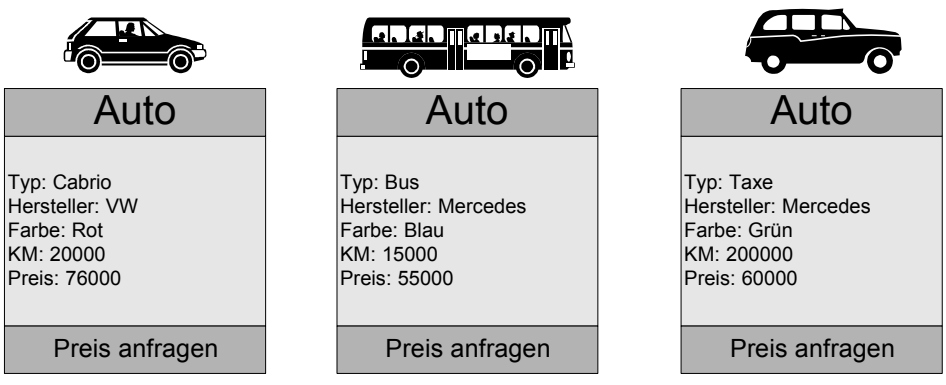
Objekte, Klassen, Vererbung...(Objects, Classes, Inheritance ..)

Wir wollen nun die Begriffe Objekt, Klasse, Botschaft, Vererbung, Polymorphismus und Assoziation im Detail erklären.

Objekt, Attributwerte und Operation



Ein *Auto* ist ein Objekt. Objekte sind Bestandteil der Realität. Betrachten wir das Objekt *Cabrio* genauer. Das Objekt *Cabrio* ist charakterisiert durch *den Typ, Hersteller, Farbe, KM,* und den *Preis*. Wir bezeichnen dies als die Attributwerte des Objektes *Cabrio*. Kommt ein Interessent für das *Cabrio*, so müssen wir den Preis abfragen. Dazu benötigen wir eine Funktion, die auf das Objekt *Cabrio* angewendet wird. Bei dem objekt-orientierten Paradigma sprechen wir von Operation statt Funktion. Ein Objekt enthält Attributwerte, auf die nur Mittels der Operation zugegriffen werden kann. Wir sagen auch, die Daten des Objekts sind verkapselt.



Attribute (Attributes)

Außer dem Objekt *Cabrio* gibt es noch andere Autos. Die obigen drei Objekte (Autos) besitzen zwar unterschiedliche Attributwerte, die aber alle von der gleichen Art sind. Wir sprechen von den Attributen *Hersteller, Farbe, KM,* und *Preis*.

Klasse (Classes)

Alle Objekte Auto besitzen die gleiche Operation *Preis abfragen*. Wir fassen daher diese Objekte zur Klasse *Auto* zusammen. Eine Klasse definiert die Attribute und Operationen ihrer Objekte.

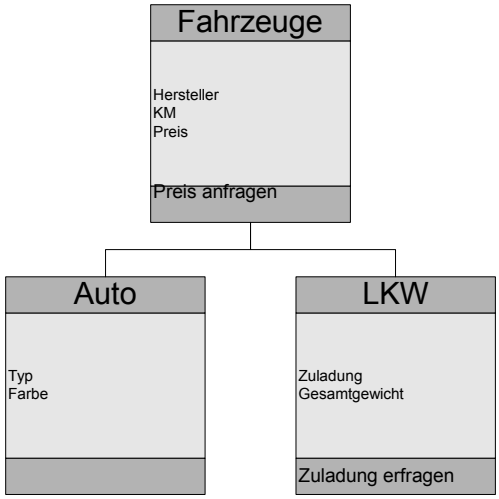
Botschaft (Message)

Wie kann die Operation Preis abfragen für das Objekt *Cabrio* ausgeführt werden. Wir senden dazu dem Objekt eine Botschaft. Sie aktiviert die Operation gleichen Namens. Die gewünschten Ausgabedaten werden an den Sender der Botschaft zurückgegeben.

Vererbung (Inheritance)

Wir führen nun eine weitere Klasse ein, nämlich die Klasse *LKW*.

Die Klassen *Auto* und *PKW* besitzen die *Attribute: Hersteller, KM, Preis* und die Operation *Preis anfragen* gemeinsam. Diese Attribute tragen wir in die neue Klasse *Fahrzeuge* ein.



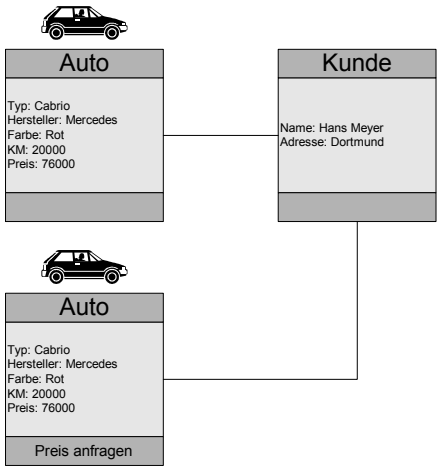
Die Klasse *Fahrzeuge* vererbt ihre Attribute und Operationen an die Klassen *Auto* und *LKW*. Das bedeutet, die Klassen *Auto* und *LKW* besitzen neben ihren eigenen Attributen und Operationen zusätzlich die Attribute und Operationen der Klasse *Fahrzeuge*. Dies ist das grundlegende Prinzip der Vererbung.

Polymorhismus (Polymorhism)

Polymorphismus bedeutet, daß dieselbe Botschaft an Objekte verschiedener Klassen gesendet werden kann und diese Objekte die Botschaft ganz unterschiedlich interpretieren.

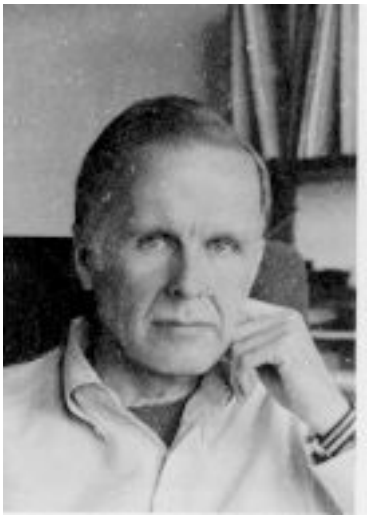
Assoziationen zwischen Objekten (Assoziation between Objects)

Nehmen wir an, es gibt eine weitere Klasse Kunden. Z.B. hat der Kunde Hans Meyer die beiden *Auto* Objekte *Cabrio* und *Taxe* gekauft. Wir müssen nun eine Beziehung zwischen den Klassen PKW und Kunde modellieren. Wir sprechen hier von einer Assoziation.



Dies sind die grundlegenden Ideen von Objekten, Klassen usw. Selbstverständlich gibt es weitere Definitionen über Zusammenhänge zwischen Klassen und Operationen. Eine ausführliche Behandlung würde den Rahmen dieses Papiers übersteigen.

6 [#$+K] John Warner Backus



1924
John Backus wurde am 3. Dezember 1924 in *Philadephia / Pennsylvenia* geboren.

1942
John Backus beginnt die Ausbildung an der *University of Virginia,* wechselte dann zur Armee und studierte an der *University of Pittsburg* von 1943-1944.

1950
Backus beginnt eine Tätigkeit bei der Firma IBM als Programmierer. Er arbeitete dort am *IBM Selective Sequence Electronic Calculator (SSEC*).

1954-1957
Erste Idden zu FORTRAN und Fertigstellung des ersten FORTRAN Übersetzers.

1959
Vorschlages einer Notation (später bekannt als Backus-Naur Form) zur Beschreibung der Syntax der Sprache ALGOL 60.

1970-1978
Backus kritisiert die damalige Programmiertechnik, die immer noch auf der Struktur der Rechnern in den 40-ziger Jahren basiert. Er führt den Begriff *Function-Level Languages* ein. Dafür erhielt er 1978 den *Turing Award*[1].

Auszeichnungen

[#] John Warner Backus
[$] John Warner Backus
[+] auto
[K] John Warner Backus
[1] Alan Mathison Turing, geboren am 23. Juni 1912, in London, gestorben am 7. Juni 1954, entwickelte 1936 das mathematische Modell einer idealisierten Rechenmaschine (Universelle Turing-Maschine).

IBM fellow, 1963; W.W. McDowell Award, IEEE, 1957; National Medal of Science, 1975; ACM Turing Award, 1977; IEEE Computer Society Pioneer Award, 1980; Member, National Academy of Sciences; Member, National Academy of Engineers; Charles Stark Draper Award, National Academy of Engineering (NAE), 1993.

Bedeutende Publikationen

Backus, John W., and Harlan Herrick, "IBM 701 Speedcoding and other Automatic-Programming Systems," ONR Symp. Automatic Programming for Digital Computers, ONR, Washington D.C., 1954, pp. 106-113.
Backus, J.W., Rj, Beeber, S. Best, R. Goldberg, L.M. Heibt, H.I,. Herrick, R.A. Nelson, D. Sayre, P.B. Sheridan, H. Stern, 1. Ziller, R.A. Hughes, and R. Nutt. Proamm 's Reference Manual, The Fortran Automatic Coding System for the IBM 704 EDPM, IBM Corporation, New York, 1956.
Backus, John W., "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference," Proc.. First Int'l. Conf. Information Processing, Butterworth, London, 1960, pp. 125-132.
Backus, John W., "Can Programming be Liberated from the Von Neumann Style? A Functional Style and Its Algebra of Programs," Comm. ACM.1 Vol. 21, 1978, pp. 613-641.

IBM 704, FORTRAN, Backus-Naur Notation

IBM 704

Die Auslieferung des IBM Rechners IBM 704 im Jahre 1954 gilt als einer der Hauptgründe für die Entwicklung von FORTRAN. Vorher hatte Backus am Rechner SSEC (IBM) gearbeitet, er entwickelte *Speedcoding* zur Unterstützung von Gleitkommarechnungen. Auch macht Backus Vorschläge für die Gleitkommaarithmetik der IBM 650.

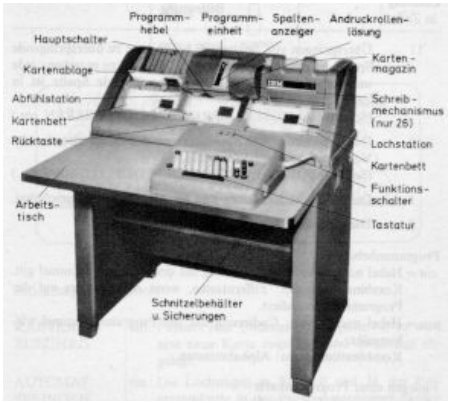
IBM 702, 704, 704, 1955 /BEAU68/, p. 135.

FORTRAN

Die Arbeit an FORTRAN begann 1953 mit einer von John Backus geleiteten Gruppe bei der *International Business Machines Corporation (IBM)*. FORTRAN ist eine Abkürzung für *FORmula TRANslating*. Der Name ist wichtig, da er die frühe Perspektive hinsichtlich der Programmiersprachen widerspiegelt. FORTRAN war eine höhere Programmiersprache (höher als Maschinensprache), da mit ihr mathematische Formeln ausgedrückt werden. Dennoch war sie nicht als universelle algorithmische Sprache gedacht, sondern als Schreibweise für mathematische Formeln, die von einer Maschine für eine Maschine übersetzt werden konnten.

Das Ziel war es, das Ausführen von Einzelheiten und das wiederholte Planen zu umgehen. 1954 glaubte man, FORTRAN würde die Programmierung und Fehlersuche gewissermaßen überflüssig machen. FORTRAN war näher am Denkprozeß, erforderte weniger Tastatureingaben und war leichter zu lernen. Statt jedoch die technische Entwicklung von Programmiersprachen abzuschließen, löste FORTRAN eine Suche nach Sprachen aus, die auf einer hohen Ebene mit dem Computer kommunizieren, einer Ebene, die der Art der Menschen zu denken näher ist.

Die Arbeit am FORTRAN-Übersetzer (oder *Compiler*, wie diese Programme heute auch genannt werden), begann 1955. Geplant war eine Dauer von sechs Monaten - doch sie endete erst 1957. Der erste FORTRAN Compiler bestand aus 25000 Zeilen Maschinencode. Einen Compiler in einer Assemblersprache zu schreiben, kann ziemlich mühsam und zeitaufwendig sein. Der FORTRAN I Compiler (siehe Handbuch rechts) für die IBM 704 benötigte bis zu seiner Fertigstellung 18 Mannjahre und es bedurfte etwa weiterer 10 Mannjahre, um ihn für die IBM 709 in die FORTRAN II Version zu erweitern. Damals schrieb ein Programmierer das Programm auf Papier und stanzte es dann in Lochkarten.



Bis in die sechziger Jahre wurden FORTRAN Programme mit Lochkartenmaschinen (siehe Bild rechts) in Lochkarten gestanzt.

FORTRAN hat sich ständig weiterentwickelt und tut dies noch immer. Die Sprache FORTRAN wurde unter der Schirmherrschaft *des American National Standards Institute (ANSI)* und der *International Standards Organization (ISO)* genormt, und zwar in den Jahren 1966, 1978 und 1992. Die neueste FORTRAN-Norm ANSI X.3198-1992 wurde am 21. September 1992 nach langer Diskussion verabschiedet.

Backus-Naur Notation

Zur Beschreibung der Syntax von ALGOL 60 wurde von Backus eine eigene Schreibweise entwickelt. Diese (und einige verwandte Versionen) sind unter dem Namen BNF bekannt, was zu Ehren von John Backus und Peter Naur, ihren Erfindern, für Backus-Naur Form (BNF) steht. Backus führte die Beschreibung von ALGOL58 1959 auf der Konferenz ACM-GAMM (American Computing Machinery-Gesellschaft für Mathematik und Mechanik), die sich mit der Spezifikation von ALGOL beschäftigte, ein. Die BNF-Definition wurde später von Naur leicht modifiziert und zur Beschreibung der Syntax von ALGOL60 verwendet. Die BNF-Definitionen sehr ähnlich zu Chomsky's kontext-freien Grammatiken ist.

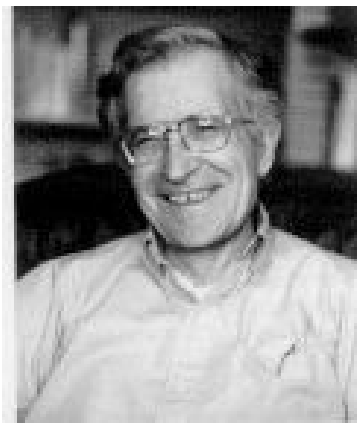
Bei BNF-Definitionen denken wir im allgemeinen an die Definition der Symbolfolgen, aus denen die Sprache besteht. Mit Symbolen sind hier die Bezeichner, Zahlen, Schlüsselwörter, Zeichensetzung usw. gemeint, aus denen die Sprache zusammengesetzt ist. Im ALGOL 60 Report ist z.B. die Definition einer Gleitkommazahl (Auszug) wie folgt beschrieben.

< vorzeichenlose Zahl >	-	< Dezimalzahl>
< vorzeichenlose Zahl >	-	< Exponententeil>
< vorzeichenlose Zahl >	-	< Dezimalzahl> <Exponenteil>
< Dezimalzahl >	-	< vorzeichenlose ganze Zahl >
< Dezimalzahl >	-	< Dezimalbruch >
< Dezimalzahl >	-	< vorzeichenlose ganze Zahl >
		< Dezimalbruch >
< vorzeichenlose ganze Zahl >	-	< Ziffer >
< vorzeichenlose ganze Zahl >	-	< vorzeichenlose ganze Zahl >
		< Ziffer >

< Dezimalbruch >	-	.< vorzeichenlose ganze Zahl >
< Exponententeil >	-	$_{10}$< ganze Zahl >
< ganze Zahl >	-	< vorzeichenlose ganze Zahl >
< ganze Zahl >	-	+<vorzeichenlose ganze Zahl >
< ganze Zahl >	-	-< vorzeichenlose ganze Zahl >

Eine vorzeichenlose ganze Zahl kann in der Backus-Naur Notation würde wie folgt geschrieben werden: <Vorzeichenlose ganze Zahl>:: 0|1|2|3|4|5|6|7|8|9|0. Dies bedeutet, eine vorzeichenlose ganze Zahl können die Ziffern von 0 - 9 sein.

7 [#] [$] [+] [K] Avram Noam Chomsky



Avram Noam Chomsky, geb, 1928 in Pennsylvania, ist Professor für Linguistik und Philosophie am *Massachussetts Institute of Technology*. Avram Noam Chomsky ist Mitglied der National Academy of Sciences, USA, und der Deutschen Akademie der Naturforscher Leopoldina.

Auszeichnungen (Auszug)

Die Liste der Auszeichnungen für Noam Chomsky ist sehr umfangreich. Aber besonders erwähnt werden sollte die Verleihung *Helmholtz-Medaillie*, die Noam Chomsky am 29. Juni 1996 von der *Berlin-Brandenburgischen Akademie der Wissenschaften* erhalten hat. *Die Berlin-Brandenburgische Akademie* führt dazu u.a. in einer Pressemitteilung vom 25. Juni 1996 aus: *Die Berlin-Brandenburgische Akademie der Wissenschaften hat Professor Dr. Avram Noam Chomsky in Würdigung seiner bahnbrechenden Arbeiten zur Theorie der natürlichen Sprache, die das Verständnis der fornialen Struktur, der psychologischen Mechanismen und der biologischen Grundlagen der menschlichen Sprachfähigkeit auf eine neue Stufe gehoben haben und prägend für die Entwicklung der kognitiven Wissenschaften geworden sind, die Helmholtz-Medaille verliehen*[2].

Bedeutende Publikationen

Chomsky, Noam, Three Models for the Description of language. IRE Transactions of Information Theory, IT-2, pp. 113-124, 1946.
Chomsky, Noam, Syntactic Structures, Mouton & Co., The Hague, 1957.
Chomsky, Noam, Chomsky: Selected Readings, Oxford University Press, Oxford, 1971.
Chomsky, Noam, Language and Mind, enl. cd., Harcourt Brace jovanovich, New York, 1972.
Chomsky, Noam, Topics in the Theory of Cenerative Crammar, Mouton & Co. The Hague, 1975.
Chomsky, Noam, 7he Logical Structure of Linguistic Theory, Plenum Press, New York, 1975.
Chomsky, Noam, Reflections on Language, Ist ed., Pantheon Books, New York, 1975.
Chomsky, Noam, Rules and Representations, Columbia University Press, New York, 1980.
Chomsky, Noam, Knowledge of Language. Its Nature, O'Yigin, and Use, Praeger, New York, 1986.
Chomsky, Noam, Language and Problems of Knowledge: the Managua Lectures, MIT Press, Cambridge, Mass., 1988.

[#] Avram Noam Chomsky
[$] Avram Noam Chomsky
[+] auto
[K] Avram Noam Chomsky

[2] Die Berlin-Brandenburgische Akademie der Wissenschaften (BBAW) setzt mit der Verleihung der Helmholtz-Medaille zur Würdigung herausragender wissenschaftlicher Leistungen eine langjährige Tradition der Preußischen Akademie der Wissenschaften fort. Erstmals wurde die Medaille am 2.Juni 1892 an den Physiologen Emil Du Bois-Reymond, die Physiker Robert Bunsen und Lord Kelvin sowie den Mathmatiker Karl Weierstraß verliehen. Nach so bedeutenden Wissenschaftlern wie dem Anatomen und Pathologen Rudolf Virchow, dem Physiker Max Planck und dem Chemiker Otto Hahn erhielt im Sommer 1990 das heutige Ehrenmitglied der Berlin-Brandenburgischen Akademie der Wissenschaften und Alt-Präsident der Leopoldina, Heinz Bethge, diese hohe Auszeichnung. Von der Berlin-Brandenburgischen Akademie der Wissenschaften wurde die Helmholtz-Medaille erstmals 1994 an Manfred Eigen, Direktor am Max-Planck-Institut für biophysikalische Chemie Göttingen und außerordentliches Mitglied der BBAW, verliehen.

Grammatiken, *Vater* der formalen Sprachen

Das Werk von Noam Chomsky beschränkt sich nicht nur auf den Bereich der Computerwissenschaften, sondern es umfaßt bahnbrechende Arbeiten auf dem Gebiet der natürlichen Sprache. Der Schwerpunkt seiner brillianten Arbeiten bezüglich der Programmiersprachen lag nicht in der Definition einer neuen Programmiersprache, sondern in der Definition einer Hierarchie von Grammatiken, die auch zur Definition von Programmiersprachen eingesetzt werden konnten.

Um die Wichtigkeit von Grammatiken für Sprachen zu verstehen, wollen wir uns einige Konzepte von Sprachen, die sich dann auf Programmiersprachen anwenden lassen, anschauen: Die Lehre der Sprache ist üblicherweise in drei Bereiche aufgeteilt: Syntax, Semantik und Pragmatik. Die Syntax einer Sprache bestimmt die Form, das Format, das Aussehen und den strukturellen Aufbau der Sprache. Der Satz

Gemalt zwei Tisch.

ist beispielsweise syntaktisch falsch. Dies ist ein syntaktischer Fehler. Aber Sprache beinhaltet jedoch mehr als nur Syntax. Vergleichen Sie die folgenden Beispiele von Noam Chomsky:

Revolutionäre neue Gedanken sind selten.
Farblose grüne Ideen schlafen wild.

Beide Sätze sind syntaktisch richtig, der zweite ist jedoch völlig sinnlos! Das Problem liegt nicht in seiner Syntax, die dieselbe ist wie beim ersten Satz; vielmehr liegt es in seiner Semantik. Die Semantik einer Sprache betrifft die Bedeutung und Auslegung der Sprache. Syntax und Semantik sind oft nur schwer auseinanderzuhalten. Betrachten Sie diese Sätze :

Zeit bewegt sich schnell wie ein Pfeil.
Obst bewegt sich schnell wie eine Banane.

Diese Beispiele verdeutlichen, daß die Bedeutung der Worte in einem Satz eng mit dessen syntaktischer Struktur zusammenhängt.

Die Pragmatik einer Sprache bezeichnet alles übrige: Ursprung, Anwendung und Wirkung einer Sprache. Die Fragen der Pragmatik sind von großer Bedeutung, können jedoch nur schwer systematisch untersucht werden. Der Einfluß von APL war beispielsweise beschränkt, weil zur Eingabe der Symbole der Sprache eine ungewöhnliche Tastatur erforderlich war.

Die Computerwissenschaft unterscheidet sich von der Linguistik darin, daß ihre Lehre präskriptiv oder normativ ist, die der Linguistik dagegen deskriptiv. Linguisten bleiben bei den natürlichen Sprachen, doch wir können Programmiersprachen erstellen. Wir können schlechte oder gute Programmiersprachen erstellen.

Die Syntax einer Sprache hat großen Einfluß auf die Einfachheit ihre ihrer Anwendung. Alle C-Programmierer haben unter den negativen Auswirkungen zu leiden gehabt, wenn anstatt der Gleichheitszeichen für relationale Operationen = = der Zuweisungsoperator = verwendet worden war. Zu den Geschichten um Programmiersprachen gehört auch die, daß folgendes Programmfragment zur Zerstörung einer Rakete bei deren Abschuß führte:

```
DO 10 I = 1.5
A(I)= X + B(I)
CONTINUE
```

Vermutlich sollte anstatt des Punktes (statt 1.5 solltec es 1,5 heißen) ein Komma die 1 und die 5 voneinander trennen. Das Problem war, daß die DO-Anweisung, da FORTRAN Leerzeichen ignoriert, als die Zuweisung

D010I = 1.5 statt DO 10 I = 1.5

betrachtet wurde. Anstatt einer Schleife wurde nun eine Zuweisung (an eine nicht deklarierte Variable) ausgeführt.

Es ist das Verdienst von Noam Chomsky, verschiedene Typen von Grammatiken, die für die Definition von Programmiersprachen verwendet weden können, definiert zu haben. Die Chomsky-Hierarchie vergleicht die Leistung verschiedener Grammatiken miteinander. Die Backus-Naur Notation für ALGOL60 basiert auf den Definitionen der Hierarchie der Chomsky Grammatiken von 1946. Daher kommt auch der Titel: *Vater der formalen Sprachen*, wie J.A.N. Lee /LEE.95/ es ausdrückt. Noam Chomsky hat bahnbrechende Arbeiten zur Theorie der natürlichen Sprache geleistet, die das Verständnis der formalen Struktur, der psychologischen Mechanismen und der biologischen Grundlagen der menschlichen Sprachfähigkeit auf eine neue Stufe gehoben haben und prägend für die Entwicklung der kognitiven Wissenschaften geworden sind

Mit dem Konzept der *Generativen Grammatik*, das unter anderem Ideen von Leibniz und Wilhelm von Humboldt mit Entwicklungen der modernen Logik von Frege bis Kleene und Turing verbindet, hat Noam Chomsky eine Wende in der Analyse und Erklärung der Struktur der menschlichen Sprachfähigkeit herbeigeführt, die nicht nur für die Linguistik eine neuartige Perspektive eröffnet und eine Fülle empirischer Untersuchungen ausgelöst hat, sondern auch die disziplinübergreifenden Dimensionen auf neuartige Weise zum Thema gemacht hat. Die Auswirkungen dieser Entwicklungen gehören zu den entscheidenden Impulsen der kognitiven Wissenschaft nach Piaget, aber auch der Sprachphilosophie und der Diskussion methodologischer Grundfragen im Bereich der *Philosophy of Mind*.

Über diesen fächerübergreifenden Einfluß im Bereich der Kognitionswissenschaften hinaus hat sich Chomsky - unter anderem mit analytischen Arbeiten zum Vietnamkrieg, zum Nahostkonflikt und zur Rolle der Massenmedien - nachhaltig an der Diskussion brennender philosophischer und politischer Fragen unserer Zeit beteiligt.

8 [#] [$] [+] [K] Ole-Johann Dahl; Kristen Nygaard

Dahl, Ole-Johann

1931
Geboren 1931 in *Mandal*, Norwegen.

1952-1963
Vertrag mit *Norwegian Defense Research Establishment (NDRE)*

Ab 1963
Professor an der *University of Oslo*.

Bedeutende Publikationen

Dahl, 0.-J., and K. Nygaard, "SIMULA-AN Algol Based SIMULAtion Language," Comm. ACM, Vol. 9, No. 9, Sept. 1966, pp, 671-682.
Dahl, 0.-J., and K. Nygaard, "Classes and Subclasses," in Buxton, j., cd., *,SIMULAtion Programming Lan ages,* North-Holland, Amsterdam, 1967.
Dahl, 0.-J., and C.A.R. Hoare, "Hierarchical Program Structures," in Dahl, 0.-J., E. Dijkstra, and C.A,R. Hoare, *Structured Programming,* Academic Press, New York, 1972.
Dahl, 0.-J., *Verifiable Programming,* Prentice-Hall, Englewood Cliffs, N j., 1992.

Kristen Nygaard

1926
Geboren in Oslo, Norwegen.

1956
Master Thesis: Theoretical Aspects of the MonteCarlo Methods.

1962
Direktor für Forschung am *Norwegian Computing Center*

1976-heute
Professor an der *University of Oslo.*

1973-1980
Entwicklung der Sprache DELTA (1973-1975) and BETA (1976-1980)

[#] Ole-Johann Dahl - Kristen Nygaard
[$] Ole-Johann Dahl - Kristen Nygaard
[+] auto
[K] Ole-Johann Dahl - Kristen Nygaard

Bedeutende Publikationen

Dahl, 0., and K Nygaard, "SIMULA-An Algol Based SIMULAtion Language," Comm. ACM, Vol. 9, No. 9, Sept. 1966, pp. 671-682.

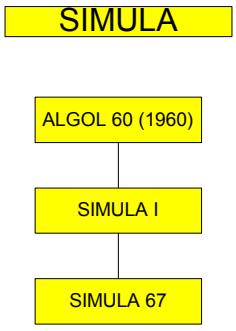
SIMULA I und SIMULA 67

Ole-Johann Dahl und Kristen Nygaard entwickelten die Sprache SIMULA. Die Entwicklung von SIMULA fand im *Norwegian Computing Center* statt, einer halbstaatlichen Forschungseinrichtung, die vom Königlich Norwegischen Amt für wissenschaftliche und industrielle Forschung geführt wird. SIMULA war eng mit ALGOL verwandt und hatte, wie ALGOL, einen bedeutenden Einfluß auf die Entwicklung von Programmiersprachen, wird jedoch nicht häufig verwendet.

Anstoß für die Entwicklung der Sprache SIMULA war der Bedarf an der Simulation von Situationen wie Warteschlangen in Supermärkten, Reaktionszeiten von Notdiensten oder Kettenreaktionen in Kernreaktoren - daher rührt auch der Name SIMULA. Kristen Nygaard und Ole-Johan Dahl entwickelten und implementierten die Sprache in den frühen 60er Jahren. Im wesentlichen handelte es sich bei SIMULA um eine Sprache zur Beschreibung von Systemen für diskrete Ereignisnetzwerke. Nach einiger Zeit der Anwendung konnten Nygaard und Dahl die Sprache überarbeiten.

Dahl und Nygaard schreiben: *Im Herbst 1966 verbrachten wir viel Zeit mit dem Versuch, Hoares Datensatzklasse an unseren Bedarf anzupassen, jedoch ohne Erfolg. Plötzlich, im Dezember 1966, bot sich eine Lösung in Form des »Prefixings«. Wir dachten dabei an eine Mautstelle auf einer Brücke, mit einer Warteschlange aus Fahrzeugen, bei denen es sich entweder um LKW oder Busse handelt. Wir schrieben gerade eine »zerlegte« Listenstruktur, die aus einem »Kopfteil« und einer variablen Anzahl von »Verbindungen« bestand, als wir feststellten, daß unsere Probleme beide durch einen Mechanismus gelöst werden konnten, der die verschiedenen Prozesse (LKW, Busse) mit je einer »Verbindung verknüpfte«, so daß jeder Verknüpfungsprozeß zu einer Ein-Block-Instanz wurde.*

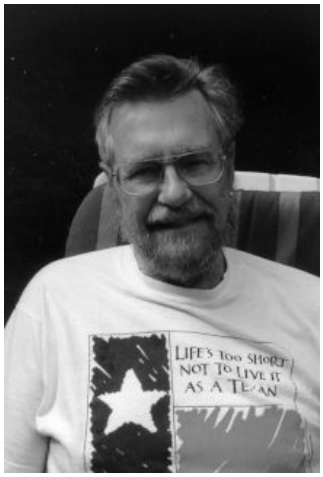
Die Sprache wurde zu einer Mehrzwecksprache und enthielt Objektklassen. Die Simulation physischer Einheiten gab den Ausschlag für die Idee von Objekten, die unabhängig voneinander existierten, also ohne die hierarchische Existenz von Datenelementen, die durch die statische Verschachtelung von Blöcken unter ALGOL bekannt geworden waren. Die neue Sprache wurde SIMULA 67 genannt und im Frühjahr 1969 vollendete das *Norwegian Computing Center* die Implementation von SIMULA 67 für die Rechner der Typen IBM System/360 und UNIVAC 1100. Über 20 Jahre später wurde ein Nachfolger von SIMULA entwickelt, der BETA genannt wird.



Eine derzeit sehr bekannte objektorientierte Sprache ist C++. Bjarne Stroustrup wurde von SIMULA inspiriert, der Programmiersprache C Klassen hinzuzufügen. Der Erfolg und die weite Verbreitung von C sowie das Fehlen jeglicher Unterstützung von abstrakten Datentypen in C führte zu einer weiten Verbreitung von C++. 1993 vergab ACM den *Grace Murray*

Hopper Award an Stroustrup, den außergewöhnlichen jungen Computerprofi, für eine einzige großartige technische Leistung.

9 [#] [$] [+] [K] Edsger Dijkstra



1930
Geboren in Rotterdam, Niederlande

1950
Studium der Mathematik und der theoretischen Physik an der *Universität Leiden,* Niederlande.

1956
Dijkstra entwickelt seinen ersten *non-trivial algorithm* (nicht trivialen Algorithmus). Es ist der *Shortest Path* Algorithmus (siehe auch unten), mit dem er u.a. die Leistungsfähigkeit der Rechenanlage ARMAC[3] zeigen wollte.

1959
Doktorgrad in Computerwissenschaften an der *Municipal Universität* von Amsterdam.

1962-1984
Professor an der *Eindhoven Universität*.

1984-heute
Professor an der *University of Texas.*

Auszeichnungen (Auszug)

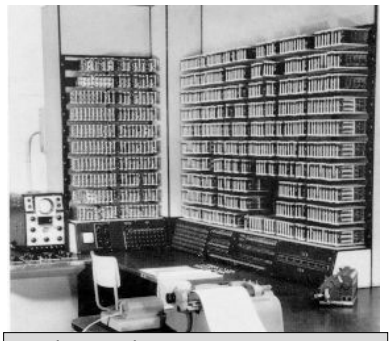
1980 IEEE Computer Society Pioneer Award; 1972 ACM Turing Award. 1980 IEEE Computer Society Pioneer Award.

Bedeutende Publikationen

Dijkstra, E.W., "Some Meditations on Advanced Programming," Proc. IFIP Congress, North-Holland, Amsterdam, 1962, pp. 535-538.

[#] Edsger Dijkstra
[$] Edsger Dijkstra
[+] auto
[K] Edsger Dijkstra
[3] ARMAC steht für *Automatische Rechenmaschine MAthematisch Centrum* in Amsterdam, die im Jahre 1955 konzipiert und 1956 fertiggestellt wurde.

Dijkstra, E.W., "Programming Considered as a Human Activity," Proc. IFIP Congress, 1965, pp. 213-217.
Dijkstra, E.W., "Solution to a Problem in Concurrent Programming Control," Comm. AGM, 1965, Vol. 8,1965, p. 569.
Dijkstra, E.W., "The Structure of the 'THE'-Multiprogramming System," ACM Symp. on Operating Systems, Comm. ACM, Vol. I 1, No. 5, May 1968, pp. 341-346.
Dijkstra, E.W., "GO TO Statement Considered Harmful," letter to the editor, Comm. A CM, Vol. I 1, No. 8, Aug. 1968, p. 538.
Dijkstra, E.W., A Short Introduction to the Art of Computer Programming, Technische Hogeschool, Eindhoven, 1971.
Dijkstra, E.W., "The Humble Programmer," Turing Award Lecture, Comm. ACM, Vol. 15, No. 10, Oct. 1972, pp. 859-866.
Dijkstra, Edsger, A Discipline of Programming, Prentice-Hall, Englewood Cliffs, Nj., 1976.



Rechenanlage ARMAC.

Programmieren als Ingenieurtätigkeit, Goto Considered Harmful, Strukturierte Programmierung, Dijkstra-Strukturen, *Shortest Path* Algorithmus

Programmieren als Ingenieurtätigkeit

Seit Anfang 1941 der erste funktionsfähige Computer von Konrad Zuse gebaut wurde, lassen Menschen Probleme durch diese Maschinen lösen, indem sie ein Programm schreiben. Wir nennen diese Tätigkeit Programmieren. Lange Zeit hindurch verfolgte die Programmierausbildung das Ziel, die Syntax einer oder mehrerer Programmiersprachen zu lernen. Dies geschah anhand exemplarischer Beispiele. Programmierkurse lehrten also nicht, wie ein Programmierer eine vorgegebene Aufgabe in elementare Arbeitsschritte zerlegen muß, um sie in eine algorithmische Sprache umsetzen zu können. Dieser Transformationsprozeß blieb weitgehend seiner Intuition und seinen Fähigkeiten überlassen.

Der Programmierprozeß war deshalb wenig strukturiert. Aus der Aufgabenstellung leitete der Programmierer unmittelbar ein Programm ab. Um seine Korrektheit zu überprüfen, führte er Tests durch. Dabei fand er zunächst syntaktische Fehler und erst wenn ein Programm ausgeführt werden konnte, Fehler in der Programmlogik. Indem der Programmierer einen logischen Fehler beseitigte, beging er häufig neue syntaktische Fehler. So entstand ein Teufelskreis aus Test und Neuprogrammierung, der sich bis in die Wartungsphase erstreckte. Viele Programmierer haben sich mit diesem Faktum wie mit einem Naturgesetz abgefunden.

Djikstra schrieb 1972 über Rechner und das Programmieren: *Als es noch keine Rechner gab, war auch das Programmieren noch kein Problem, als es dann ein paar leistungsschwache Rechner gab, war das Programmieren ein kleines Problem und nun, wo wir gigantische Rechner haben, ist auch das Programmieren zu einem gigantischen Problem geworden. In diesem Sinne hat die elektronische Industrie kein einziges Problem gelöst, sondern nur neue geschaffen. Sie hat das Problem geschaffen, ihre Produkte zu benutzen.*

Dijkstra wies 1972 auch darauf hin, daß Testen nur zeigt, es sind noch Fehler in einem Programm vorhanden. Testen kann nie die Fehlerfreiheit beweisen. Allerdings tut man dem Programmierer unrecht, wenn man ihm allein die Schuld für diese Situation anlastet; denn Programmieren heißt auch, die Komplexität einer Aufgabenstellung zu bewältigen. Dazu benötigt der Programmierer eine Methodik, die ihn lehrt, Komplexität zu beherrschen und zu reduzieren. Sie war lange Zeit hindurch unbekannt.

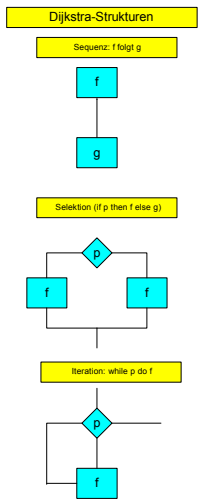
So entstand die Forderung nach einem ingenieurwissenschaftlich fundierten Programmierstil, der die Programmierkunst durch eine Programmiertechnologie ersetzt. Auf der *NATO-Konferenz* 1968 in Garmisch prägte F.L. Bauer [4] (siehe Bild) für diese neue Disziplin den Begriff *Software Engineering*. Es war als Herausforderung an die Wissenschaft gedacht, eine Konstruktionsmethodik zusammen mit unterstützenden Werkzeugen zu entwickeln, die gestattet, Programme als Industrieprodukte immaterieller Art zu gestalten und zu produzieren.

Goto Considered Harmful (GOTO Statement ist nachteilig)

Die Fachdiskussion über die ingenieurmäßige Entwicklung von Software-Produkten löste Dijkstra im Jahr 1968 mit seiner Leserzuschrift in den *Communications of the ACM* aus. Er berichtete in dieser Arbeit über seine Beobachtung, daß die Qualität eines Programms umgekehrt proportional ist zur Dichte der GOTO-Anweisungen in diesem Programm. Er fordert deshalb, diesen Anweisungstyp aus allen höheren Programmiersprachen zu verbannen (GOTO-lose Programmierung). Dijkstra begründete seine Beobachtung damit, daß viele Programmierer überfordert sind, wenn sie alle möglichen Zustände eines Programmablaufs an dem Punkt verfolgen sollen, zu dem er mit GOTO-Anweisungen verzweigt.

Strukturierte Programmierung und Dijkstra-Strukturen

Als Antithese zum herkömmlichen Programmierstil führte Dijkstra 1972 den Begriff *strukturierte Programmierung* ein. Der Praktiker, der zum ersten Mal hört, daß er Programme ohne GOTO-Anweisungen schreiben soll, wird fragen, ob er überhaupt in der Lage ist, diese Forderung zu erfüllen. Der Dijkstra'sche Ansatz wird deshalb erst auf dem Hintergrund einer Untersuchung von Böhm und Jacopini verständlich. Sie zeigten, daß jeder Algorithmus durch Kombination der unten aufgeführten Bausteine (die alle nur einen Eingang und einen Ausgang haben) darstellbar ist. Damit läßt sich das Prinzip der strukturierten Programmierung folgendermaßen definieren: Ein wohlstrukturiertes Programm basiert unabhängig von der Programmiersprache, in der es abgefaßt wurde, auf drei Grundbausteinen, nämlich:

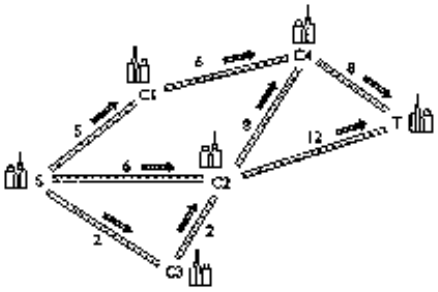


1. Der **Sequenz** als einer Folge von Programmbausteinen (z.B. Anweisungen, Programmodulen, Dialogschritten), die nacheinander ausgeführt werden;

[4] Professor Dr. Friedrich L. Bauer wurde am 10. Juni 1924 in Rcgensburg geboren. Er studierte Mathematik, Theoretische Physik und Astronomie an der Universität München. Er war Assistent und Privatdozent an der Technischen Hochschule München, wurde 1958 (als Professor für Angewandte Mathematik) an die Universität Mainz berufen. Seit 1963 ist er ordentlicher Professor für Mathematik an der Technischen Hochschule München.

2. Der **Selektion,** von der aus einer und nur einer von mehreren möglichen Programmzweigen ausgeführt wird;
3. Der **Iteration** als einem Programmbaustein, der null-, ein- oder mehrmals in der Programmausführung durchlaufen wird. (Der Sonderfall, daß eine Iteration vorzeitig abgebrochen wird, soll eingeschlossen sein.)

Shortest Path Algorithmus



Dijkstra hat auch signifikante Entwicklungen auf dem Gebiet der Semaphoren (Synchronisation von sequentiellen Prozessen) durchgeführt. 1956 entwickelte Dijkstra den *Shortest-Path Algorithm*, der jetzt auch als *Dijkstra's Algorithmus* bezeichnet wird. Dieser Algorithmus beschäftigt sich mit dem Problem der Bestimmung des kürzesten Weges zwischen zwei Orten.

10 [#] [$] [+] [K] Charles Anthony Richard Hoare

1934
Geboren am 11. Januar 1934.

1960-1962
Projektleiter bei der Implementierung von ALGOL60.

1968-1977
Professor of Computing Science an der *Queen's University of Belfast.*

1977-heute
Professor an der *Oxford University.*

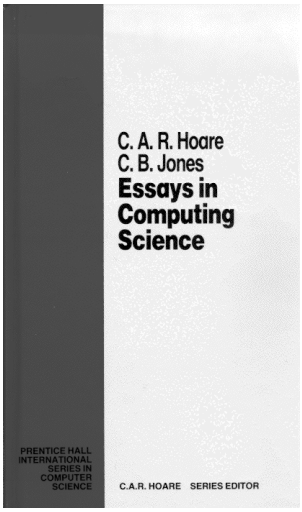
1986-1987
Admiral R. Inman Centennial Chair in Compting Theory, University of Texas.

1991-1993
Direktor des *University Computing Laboratory* in Oxford.

1993
James Martin Professor of Computing an der *Oxford University*

Auszeichnungen

Distinguished fellow of British Computer Society, 1978; DSc (Hon.), University of Southern Carolina, 1979; ACM Turing Award, 1980; AFIP Harry Goode Memorial Award, 1981; elected fellow of the Royal Society, 1982; IEE FarADAy Medal, 1985; DSc (Hon.), Warwick University, 1985; honorary doctor of science, University of Pennsylvania, 1986; DSc (Hon.), Queen's University of Belfast, 1987; foreign member, Accademia dei Lincei, 1988; honorary doctor of the University of York, 1989; member, Academia Europaea, 1989; IEEE Computer Society Pioneer Award, 1990; honorary doctor of the University of Essex, 1991; IEEE Computer Society Pioneer Award, 1991; Lee Kuan Yew Distinguished Visitor, Singapore, 1992.



Bedeutende Publikationen

Die beste Zusammenfassung der wissenschaftlichen Arbeiten von Hoare ist im Buch: *C.A.R. Hoare; C.B. Jones: Essays in Computing Science, 1989*, zu finden.

Hoare, C.A.R., E-W. Dijkstra, and 0-j. Dahl, Structured Programming, Academic Press, New York, 1972.
Hoare, C.A.R., and N. Wirth, "An Axiomatic Definition of the Programming Language PASCAL," A C7A *Informatica,* Vol. 2, 1973, pp. 335-355.
Hoare, C.A.R., "An Assessment of the Programming Language PASCAL," LF-FE, *Trans. SoftwareEng.,June* 1975.

[#] Charles Anthony Richard Hoare
[$] Charles Anthony Richard Hoare
[+] auto
[K] Charles Anthony Richard Hoare

Hoare, C.A.R., "The Emperor's Old Clothes," Comm. ACM., Vol. 24, No. 2, Feb. 1981, pp. 75-83.
Hoare, C.A.R., *Communicating,Sequential Processes,* Prentice-Hall International, New York, 1985.
Hoare, C.A.R., and C.B. jones, eds., *Essays in Computing Science,* Prentice-Hall, International, New York, 1989.

Definition und Entwurf von Programmiersprachen, Einfachheit von Programmiersprachen

Charles Anthony Richard Hoare ist einer der herausragenden Computerwissenschaftler. Hoare hat u.a. durch seinen axiomatischen Ansatz zur Spezifikation von Programmiersprachen, den *Hoare-Tripel*, weite wissenschaftliche Anerkennung gefunden. Herkömmliche Programmiersprachen und die Prädikatenlogik der ersten Stufe haben einige Gemeinsamkeiten. Diese Überschneidungen ermöglichen es, die Ausführung eines Konstrukts einer Programmiersprache mit einer logischen Vorschrift zu verbinden.

Anläßlich der Verleihung des *Turing Award* im Jahre 1980 hielt Hoare eine vielbeachtete Rede. Er erzählt von seinen ersten Erfahrungen als junger Programmierer bei einem kleinen Londoner Computerhersteller *Elliot Brothers Ltd*., wo er acht Jahre tätig war. In dieser Zeit entwickelt er den berühmt gewordenen Sortieralgorithmus *Quicksort*. Ein glücklicher Umstand spielt ihm die Kopie des *Report on the International Algorithmic Language Algol 60* zu, als er die Aufgabe erhält, eine neue höhere Programmiersprache für die nächste Computergeneration seiner Firma zu entwerfen. Es erfolgt der etwas überraschende Beschluß der Firmenleitung, eine Untermenge der Sprache ALGOL 60 auszuwählen und einen Compiler dafür zu entwickeln. Hoare bestimmt diesen Subset und macht sich an die Arbeit. Er beachtet vier grundlegende Punkte, die für ihn auch heute noch beim Entwurf von Programmiersprachen gültig sind:

- Jedes syntaktisch unrichtige Programm soll vom Compiler verworfen werden. Jedes syntaktisch korrekte Programm soll ein Ergebnis oder eine Fehlermeldung liefern. Beides soll an Hand der Quellsprache gedanklich nachvollzogen werden können. Beispiel:das Überschreiten von Feldgrenzen gehört während der Laufzeit abgefangen.
- Der Code, den der Compiler liefert, soll kurz und der Speicherbedarf während der Laufzeit minimal sein.
- Der Aufruf von Prozeduren und Funktionen soll möglichst effizient gestaltet werden.
- Der Compiler soll in einer einzigen Übersetzungsphase seinen Code liefern. Er soll aus rekursiven Prozeduren bestehen, die eine bestimmte syntaktische Einheit der Sprache - eine Anweisung, eine Deklaration etc. - analysieren und übersetzen.

Mit dem Fortschritt der Arbeiten entdeckt er Möglichkeiten, die Einschränkungen, die zum Subset führten, zu lockern, ohne die Prinzipien der Sprache ALGOL 60 zu verletzen. So gelingt ihm die Implementierung fast aller Möglichkeiten, die die Sprache bietet.

Seine Arbeit findet Beachtung. Im August 1962 wird er in die *Working Group 2.1* der *IFIP* (*International Federation of Information Processing*) berufen. Diese Arbeitsgruppe befaßt sich mit der Maintenierung und Weiterentwicklung von ALGOL. Bis zum Oktober 1966 nimmt er an jeder Sitzung teil. Der Entwurf von ALGOL X, der Nachfolgesprache von ALGOL 60 wird in Angriff genommen. Es gehen Vorschläge für neue Features ein, die Niklaus Wirth (der später PASCAL entwickeln wird) in einen Sprachentwurf einbringt. Hoare ist hoch erfreut über Wirths Arbeit, vermeidet sie doch all die bekannten Schwächen von ALGOL 60. Außerdem enthält sie mehrere neue Elemente, die einfach und effizient zu

implementieren sind und zu einem sicheren Gebrauch der Sprache führen. Weitere Vorschläge und Verbesserungen, auch von Hoares Seite, resultieren in einer Sprache, implementiert auf einer IBM 360 und von -den sehr zufriedenen Benutzern ALGOL W genannt.

Die verhängnisvolle Wende

Doch da tritt das für Hoare kaum faßbare Ereignis ein. Das ALGOL-Komitee beschließt, als Nachfolge für ALGOL 60 eine ganz neue, ehrgeizige und wie Hoare meint sehr unattraktive Sprache zu entwerfen. Hoare warnt verzweifelt vor der Undurchsichtigkeit, dem Überumfang und dem übertriebenen Ehrgeiz dieses Projekts. Aber er spricht in den Wind! Eine besonders schmerzliche Erfahrung, da er bei seiner Firma eben mit einem großen Softwareprojekt aus genau diesen Gründen Schiffbruch erlitten hatte. Ihm bleibt nur die resignierende Feststellung, daß es anscheinend zwei Wege des Softwareentwurfs gibt:

1. Die Gestaltung so einfach zu wählen, daß offensichtlich keine Mängel und Fehler enthalten sind.
2. Die Gestaltung so kompliziert zu wählen, daß es keine offensichtlichen Mängel und Fehler gibt.

Das ALGOL-Komitee schlägt den zweiten Weg ein. Von Sitzung zu Sitzung werden die Entwürfe zu einem immer dicker anwachsenden Dokument überarbeitet. Ohne überhaupt nur daran zu denken, die Sprache zu vereinfachen, werden die Autoren gebeten, immer noch mehr komplexe Komponenten aufzunehmen, wie z.B. parallele Verarbeitung. Schließlich erfolgt im Dezember 1968 in München die Geburt des Ungeheuers. Es erhält den Namen ALGOL 68. Was Hoare zusammen mit einigen Kollegen noch tun kann, ist die Herausgabe eines Minderheitenpapiers mit der Feststellung: ... *als Werkzeug zur Erstellung von gedanklich komplexen und trotzdem zuverlässigen Programmen ist die Sprache nicht geeignet*. Doch IFIP läßt dieses Papier in der Versenkung verschwinden. Der Berg der Sprachen und Supersprachen wächst und wächst.

So hat Hoare, mittlerweile Professor an der Universität Belfast, Einblick in ein weiteres ehrgeiziges Projekt. Als Vorsitzender des Komitees Nr.10 der *European Computer Manufacturers Association* beobachtet er die Entwicklung einer Sprache aller Sprachen. Der Welt größter Computerhersteller IBM gibt 1964 den Auftrag zum Entwurf eines ähnlich ehrgeizigen Vorhabens. Die Sprache erhält den Namen PL/1. Auch hier beobachtet Hoare mit gedämpfter Schadenfreude die gleichen Schwächen und Übel.

Programmierer sehen sich unvermeidlich komplexen Problemen gegenüber: Einerseits sollen die Computer immer mehr können. Andererseits aber, wenn das grundlegende Werkzeug, die Programmiersprache, sich zusätzlich als komplex und in der Handhabung schwierig erweist, wird die Sprache ein Teil des Problems und nicht ein Teil der Lösung.

Sind diese Fehlentwicklungen früherer Jahre heute noch relevant? fragt Hoare. Er befürchtet es und erinnert an ein Projekt, das als Entwurfsergebnisse die *Dokumente strawman, woodenman, tinman, ironman, steelman, green* und schließlich ADA hervorgebracht hat. Hoare sieht bis jetzt keine Anzeichen, daß diese Sprache irgend ein Problem vermieden hätte, das die Entwicklung früherer umfangreicher Sprachen schon geplagt hat. Die hoffnungsvollen Ansätze, wie Zuverlässigkeit, Lesbarkeit der Programme, formale Definition der Sprache und zunächst auch eine gewisse Einfachheit, wurden zugunsten von Mächtigkeit und unnötigem, sogar gefährlichem Ballast geopfert.

11 [#][$][+][K] Grace Brewster Murray Hopper

1906
Grace Brewster Murray Hopper wurde am 9 Dezember 1906 in New York City geboren.

1930-1934
Sie studierte Mathematik an der *Yale University*, wo ihr der M*aster's Degree*. (1930) und die Doktorwürde im Mathematik (1934) zuerkannt wurden. Zwischen 1862 und 1934 erhielten nur 1279 Wissenschaftler den Dr.Titel für Mathematik.

1944
Im Jahre 1944 trat sie dem *Bureau of Ordnance's Computation Project* bei, welches von Howard Aiken an der *Havard Univerity* geleitet wurde. Dort hatte Grace Hopper erstmalig mit Rechner zu tun.

1945-1992
Am Ende des zweiten Weltkrieges verließ Grace Hopper die Marine als Leutnant und ging als Mathematikerin zu der *Eckert-Mauchly Corporation*, wo die Rechner BINAC (Binary Automatic Computer) und UNIVAC konstuiert wurden. Sie war dort als Forscherin bis 1971 tätig.

1973, 1985
Während ihrer wissenschaftlichen Tätigkeit pflegte Grace Hopper weiterhin den Kontakt zur Marine. Sie wurde 1973 zum *Captain*, 1983 zum *Commodore*, und 1985 zum *Rank of Commodore known as Rear Admiral* ernannt.

1986
Am 1. September 1986 begann sie als ein *Senior Consultant* bei der Digital Equipment Corporation zu arbeiten.

1992
Grace Hopper starb am 1. Januar 1992.

Bedeutende Publikationen

Aiken, H.H., and Grace M. Hopper, "The Automatic Sequence Controlled Calculator," reprinted in Randell, Brian, Origins of Digital Computers: Selected Papers, Springer-Verlag, Berlin, 1982, pp. 203-222.
Hopper, Grace Murray, "The Education of a Computer," Proc. AGM Conf., reprinted Ann. Hist. Comp., Vol. 9, No. 3-4,1952, pp. 271-281.20

Biographie und Auszeichnungen

[#] Grace Brewster Murray Hopper
[$] Grace Brewster Murray Hopper
[+] auto
[K] Grace Brewster Murray Hopper

Charlene W. Billings hat 1989 /BILL89/ eine Biographie über die faszinierende Persönlichkeit Grace Hopper geschrieben. Die Liste der Auszeichnungen füllt mehr als drei Seiten.

Programmierin, Bugs, Programmiersprachen, Übersetzer, FLOW-MATIC, Mother of COBOL

Programmiererin, Bugs

Ab 1944 arbeitete Grace Hopper als Programmiererin an Aiken's MARK I. Sie bezeichnete sich als: *I became the third programmer on the world's first large-scale digital computer, Mark I.* Ihre Aufgabe war u.a. Fehler (bugs) der MARK I herauszufinden und ein Manual über die Funktionsweise der MARK I zu erstellen.

Grace Hopper gilt als die Person, die den heute noch verwendeten Begriff *bug* eingeführt hat: *One day in 1945, a moth flew into the Mark II computer and lodged between the contact plates of a relay. Thus there was literally a bug in the system. Hopper taped the bug to the log book and wrote, "First actual case of bug being found." (The log book, with moth still attached, is in the Naval Museum at the Naval Surface Weapons Center, in Dahlgren, Virginia.)*

Programmiersprachen, Übersetzer, FLOW-MATIC, MATH-MATIC


Rechner BINAC, 1949.

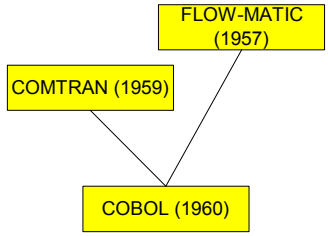
Für mehr als drei Dekaden forcierte Grace Hopper die Entwicklung von Programmiersprachen und Übersetzern. Inspiriert von John Mauchly's *Short Order Code* (BINAC, 1949), entwickelte Grace Hopper den ersten Übersetzer für Programmiersprachen, genannt A-0 (1952). Im August 1953 stellte Grace Hopper den Übersetzer A-2 vor, der mathematische Berechnungen unterstützen sollte. Der A-2 Übersetzer war sehr einfach. Es waren 60 Worte Speicher vorhanden und jede Gleitkommazahl (floating point number) benötigte 2 Worte. 1957 wurde in der Forschungsgruppe von Grace Hopper unter der Leitung von Charles Katz, die Sprache A-3 (MATH-MATIC) entwickelt. Sie enthielt Sprachelemente für die Lösung mathematischer Probleme und hatte ein gewisse Ähnlichkeit mit FORTRAN I. MATH-MATIC arbeitete sehr ineffizient und hatte nicht die Leistungsfähigkeit von FORTRAN.

Mother of COBOL (*Mutter von COBOL*)

Im Jahr 1957 entwickelte Grace Hopper den Übersetzer B-0, genannt FLOW-MATIC. Dieser Übersetzer konnte Anweisungen von der englischen Sprache in den Maschinencode übersetzen. Dies war der Vorläufer einer anwendungsorientierten Programmiersprachen. Mit FLOW-MATIC demonstrierte Grace Hopper die Erzeugung von Maschinencode aus Programmen, die in englische, französische und deutsche Schlüsselworte (keywords) enthielten. Grace Hopper vertrat schon sehr früh die Auffassung, daß mathematische Programme in einer mathematischen Notation und datenverarbeitende Programme in

englischen Worten formuliert sein sollten. Grace Hopper hatte damit schon sehr früh erkannt, daß Rechner nicht nur für Operationen mit Zahlen geeignet sind.

Im Mai 1959 veranstaltete das Verteidigungsministerium der Vereinigten Staaten eine Konferenz. Ziel dieser Konferenz war die Erstellung einer allgemeinen Programmiersprache für Datenverabeitungsanwendungen - einen für große Organisationen wie das Verteidigungsministerium wichtigen Anwendungsbereich. Eines der Ergebnisse der Zusammenkunft war die Gründung des CODASYL-Vorstandes *(COmmittee on DAta SYstems Languages*) zur Koordination der Arbeit sowie zweier Arbeitsgruppen, die das Problem kurzfristig und in der näheren Zukunft untersuchen sollten. Nach sechs Monaten hatte das erste Komitee eine neue Sprache namens COBOL (*COmmon Business Oriented Language*) etabliert, die als *Lückenbüßer* dienen sollte, um das Wachstum der Sprachen einzelner Computerhersteller (unter anderem IBM, Honeywell, RCA und Sylvania) zu kürzen. Obwohl Grace Murray Hopper kein Mitglied des Komitees war, beeinflußte sie die Sprache durch ihre vorangegangene Arbeit am Entwurf des ersten Datenverarbeitungscompilers FLOW-MATIC. Im April 1960 gab das Verteidigungsministerium die ersten Spezifikationen für COBOL. Im Dezember desselben Jahres waren die ersten Compiler auf RCA- und *Remington-Rand-UNIVAC-Rechnern* implementiert. 1968, 1974 und 1985 erschienen entsprechende ANSI-Normen. COBOL wird auch heute noch bei Datenverarbeitungsanwendungen eingesetzt.



Eines der Hauptziele beim Entwurf der Sprache war eine dem Englischen näher stehende Programmiersprache, mit der Finanzdaten verarbeitet werden konnten. Da FORTRAN für Wissenschaftler und Ingenieure gedacht war, bestand seit 1960 Bedarf für eine solche Sprache. COBOL, das zur Verarbeitung großer Mengen von Daten (zum Beispiel einer Lohnliste) konzipiert war, besaß besondere Einrichtungen für die rechnerabhängige Umgebung, einschließlich der Dateien. COBOL führte zur Organisation von Dateien in einer Sammlung von Datensätzen eine Art einfache Struktur ein.

Zur wissenschaftlichen Entwicklung von Programmiersprachen hat COBOL keinen großen Beitrag geleistet. Es entwickelte sich im großen und ganzen unabhängig von den enormen Ereignissen, die sich rund um ALGOL abspielten. Keine andere Sprache hat auf eine so wortreiche Syntax zurückgegriffen wie COBOL. Mittlerweile bieten die meisten Sprachimplementationen, unabhängig vom verwendeten System, Abstraktionen für Dateien, zumeist in Form eines Zeichenstromes. Dadurch wird eine übergeordnete, rechnerunabhängige Schnittstelle möglich, die für Programmierer einfacher ist.

Jean E. Sammet und COBOL

Die weitere Entwicklung von COBOL wurde u.a. von Jean E. Sammet entscheident beeinflußt. Sie war 1959 und 1960 Vorsitzende von zwei Untergruppen der CODASYL COBOL Gruppe, wo die detaillierten Spezifikationen der Sprache COBOL entwickelt wurden. Neben vielen anderen Aktivitäten, hat Jean Sammet u.a. einen hohen Bekanntsheitsgrad wegen ihrer Arbeiten zu der Historie von Programmiersprachen erlangt /SAMM69/.

12 [#] [+] [$] [K] Kenneth E. Iverson; Adin D. Falkhoff

Kenneth E. Iverson

1950
Studium in Mathematik und Physik an der *Queen's University* in *Kingston, Ontario.*

1954
Doktorgrad in angewandter Mathematik.

1955-1960
Assistant Professor in angewandter Mathematik an der *Havard University.*

1960-heute
IBM Corporation.

Auszeichnung

Honors and Awards: IBM Fellow, 1970; AFIPS Harry Goode Award, 1975; ACM Turing Award, 1979; IEEE Computer Pioneer Award, 1982; National Medal of Technology, 1991; member, National Academy of Engineering.

Bedeutende Publikationen

Falkoff, A.D., and K.E. Iverson, "The Design of APL," IBM J. Research and Development, Vol. 17, No. 4, 1973, pp. 324-334.
Iverson, Kenneth E., Machine Solutions of Linear Differential Equations: Applications to a Dynamic Economic Model, PhD thesis, Harvard Univ. , 1954.
Iverson, Kenneth E., A Programming Language, John Wiley, New York, 1962.
Iverson, Kenneth E., Elementary Functions, Science Research Associates, Chicago, 1966.

Adin D. Falkhoff

[#] Kenneth E. Iverson - Adin D. Falkhoff
[+] auto
[$] Kenneth E. Iverson - Adin D. Falkhoff
[K] Kenneth E. Iverson - Adin D. Falkhoff

1921
Geboren am 19 Dezember 1921 in New Jersey.

1977-1987
Manager der *APL Design Group*, *IBM Thomas J. Watson Research Center*

1987
Research Staff Member, IBM T.J. Watson Research Center.

Auszeichnungen:

IBM Outstanding Contribution Awards for Development of APL and APL / 360, 1983.
ACM Award for outstanding contribution to the development and application of APL, 1983.

APL

Kenneth E. Iverson und Adin D. Falkhoff begannen schon vor 1960 mit der Entwicklung von APL. Zunächst sollte die Sprache nicht implementiert werden. Sie war dazu entwickelt worden, Computerarchitekturen zu beschreiben, und wurde vom Gebiet der linearen Algebra beeinflußt. Vor allem Felder stellen die wichtigste Datenstruktur dar.

APL ist für seine kompakte Schreibweise und seinen ungewöhnlichen Zeichensatz bekannt. Die geradezu kryptische Schreibweise wird von ihren Anhängern wegen ihrer Leistungsfähigkeit zur exakten Beschreibung von Operationen sehr geschätzt. Die Schreibweise von APL wurde erstmals 1962 in dem Buch *A Programming Language von* Iverson beschrieben. Die ersten Buchstaben des Titels, APL, gaben der Sprache ihren Namen. Kurz darauf wurden von IBM die ersten Implementationen entwickelt.


IBM 360 Computer System

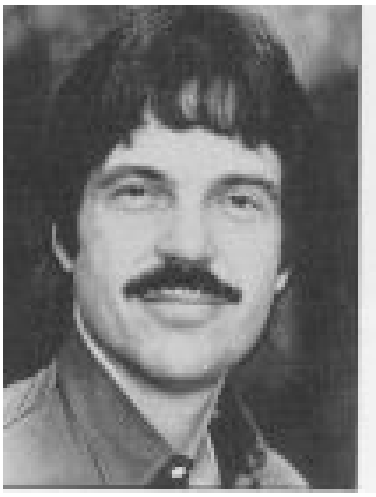
Iverson und Falkhoff berichteten: Anfang 1966 dachten wir an eine Implementation auf dem System/360, eine Arbeit, mit der wir dann im Juli ernsthaft begannen und die im Herbst ein lauffähiges System hervorbrachte. Die Tatsache, daß diese interpretative und experimentelle Implementation sich als bemerkenswert praktisch und effizient erwies, haben wir den Fähigkeiten der Personen, die sie implementiert haben, zu verdanken. Diese Fähigkeiten wurden 1973 gewürdigt, als die Hauptpersonen: L.M. Breed, R.H. Lathwell und R.D.Moore den *Grace Murray Hopper Award* von ACM erhielten.

Iverson gewann 1979 den *ACM-Turing Award* vor allem wegen der Entwicklung der Programmiersprache APL. IBM entwickelt die Sprache auch weiterhin. Mittlerweile ist eine kommerzielle Version namens APL2 erhältlich, und am *IBM Cambridge Scientific Center* wurde eine Schnittstelle zwischen APL2 und dem *X-Windows-System* entwickelt. Ein aktueller Nachfahre von APL ist die Sprache J. Das folgende Bild zeigt ein APL Program.

APL-Ausdruck	**Wert**
N-2	5
uN-2	12345
l+LN-2	23456
(1 +uN-2) IN	11321
O;d (1 + uN - 2) IN	11111
A/O;- (1 + uN - 2) IN	1

APL erfordert eine spezielle Tastatur und erlaubt sehr komplexe Probleme in wenigen Programmzeilen zu formulieren. Wegen der Kompliziertheit der Sprache APL hat sie nicht die Verbreitung wie COBOL oder FORTRAN gefunden.

13 [#] [$] [+] [K] Alan Kay



1940
Geboren in *Springfield, Massachusetts*.

1940
Aufenthalt in Australien, wo Kay's Vater geboren wurde.

1966
Examen der *University of Colarado*.

1969
Doktorgrad an der *University of Utah* in Computerwissenschaften und Entwicklung des ersten graphischen objektorientierten Computers.

1972
Kay beendet den ersten Entwurf von SMALLTALK.

Auszeichnungen

Apple Fellow, ACM Software Award, 1987

Bedeutende Publikationen

Kay, Alan, "Computer Software," *Scientific* American, Sept. 1984.
Kay, Alan, "The Early History of SMALLTALK," *ACM SIGPLAN Notices,* Vol. 28, No. 3, Mar. 1993, pp. 69-96.

Zitate

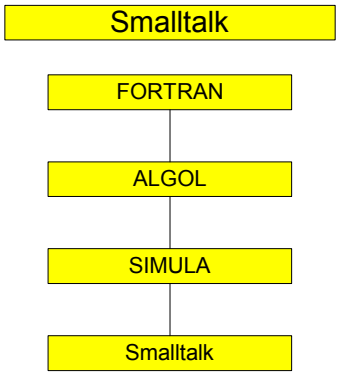
Computers are to computing as instruments are to music. Software is the score whose interpretation amplifies our reach and lifts our spirits. Lconardo da Vinci called music "the shaping of the invisible," and his phrase is even more apt as a description of software. (Kay 1984).

I think that since children appear to have to construct the world inside their heads in order to become human beings, then people must be natural constructors. Computers are the best construction material we have ever come up with outside of our brains.

Some people worry that artificial intelligence will make us feel inferior, but then, anybody in his right mind should have an inferiority complex every time he looks at a flower.

[#] Alan Kay
[$] Alan Kay
[+] auto
[K] Alan Kay

SMALLTALK, Erste Objektorientierte Sprache



Der Vorläufer des Modells der objektorientierten Programmierung (OOP) ist das Programmiersystem SMALLTALK. Der Erfinder von SMALLTALK ist Alan Kay. Die Entwicklung von SMALLTALK wurde von der Programmiersprache SIMULA beeinflußt. SMALLTALK gilt auch heute noch als die objektorientierte Sprache an sich. Die meisten Definitionen beziehen sich auf SMALLTALK.

SMALLTALK entstand seit etwa 1970 im Palo Alto Research Center der Firma XEROX. SMALLTALK war für Kinder konzipiert. Es wurde in der Entwicklungsphase von Kindern getestet. Es war Kay's Überzeugung, daß Kinder besser durch Bilder als durch Text lernen. Es handelt sich, wie die Einleitung des Buches von Goldberg und Robson (1983) feststellt, um viel mehr als eine Programmiersprache, nämlich um

- eine Vision;
- ein System auf der Grundlage nur weniger Konzepte, das aber mit einer ungewöhnlichen Terminologie definiert ist;
- einen interaktiven graphischen Arbeitsplatz zum Programmieren;
- ein großes System.

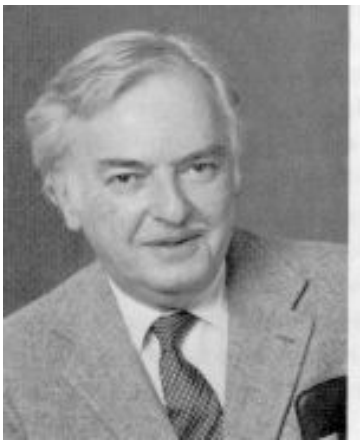
1980 verließ SMALLTALK die Grenzen von Xerox. Die formale Beschreibung von SMALLTALK-80 wurde von vier Firmen überarbeitet und verbessert: *Apple Computer, Digital Equipment Corporation*, *Hewlett-Packard* und *Tektronix.* Jede dieser Firmen implementierte das System auf ihrer eigenen Hardware. Später erhielt die *University of California* in Berkeley eine weitere Forschungslizenz. Die dabei gesammelten Erfahrungen wurden in einer von Adele Goldberg und anderen autorisierten dreibändigen Buchreihe im *Addison-Wesley-Verlag* festgehalten. Zur Zeit steht SMALLTALK auf einer Vielzahl von Plattformen zur Verfügung. Bei der Verbreitung des objektorientierten Programmierungsmodells spielte SMALLTALK eine wesentliche Rolle.

1981 BYTE Magazin

Daß es sich bei SMALLTALK um ein vollständiges Programmiersystem in und von sich selbst handelt, ist sowohl ein großer Vor- als auch ein Nachteil. Ein Vorteil liegt in der integrierten Programmentwicklungsumgebung, die dem Anwender große Flexibilität und Steuerungsmöglichkeiten gibt. Ein Nachteil ist der große Aufwand, der nötig ist, um das ganze System zu lernen, sowie die Schwierigkeit der gleichmäßigen Unterstützung eines vielseitigen Systems auf verschiedenen Hardware-Plattformen.

14 [#] [$] [+] [K] John G. Kemeny; Thomas E. Kurtz

John Kemeny:



1926
John Kemeny wurde am 31. Mai 1926 in Budapest geboren.

1943
Studium der Mathematik in *Princeton / New Jersey.*

Ab 1943
Los Alamos Projekt. Er berechnete dort Differentialgleichungen mit Hilfe von IBM Buchungsmaschinen.

1948
Assistent von Albert Einstein. Arbeit auf dem Gebiet der Vereinheitlichung der Feldtheorie.

1953-1990
Professor für Mathematik am *Dartmouth College.*

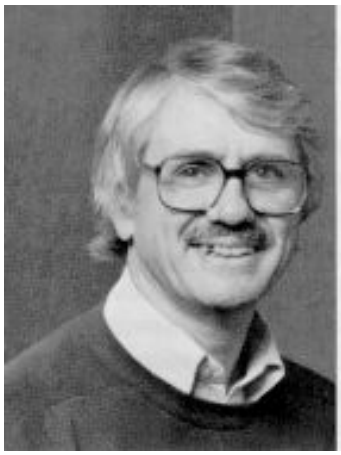
1992
John Kemeny verstarb am 26. Dezember 1992 in *Lebanon, New Hampshire, USA.*

Auszeichnungen:

Priestley Award, Dickinson College, 1976; IEEE Computer Society Pioneer Award, 1985; AFIPS Education Award, 1983; New York Academy of Sciences Award, 1984; fellow, American Academy of Arts and Sciences; First Louis Robinson Lifetime Achievement Award, EDUCOM, 1990.

Thomas Kurtz:

[#] John G. Kemeny - Thomas E. Kurtz
[$] John G. Kemeny - Thomas E. Kurtz
[+] auto
[K] John G. Kemeny - Thomas E. Kurtz



1928
Thomas Kurtz wurde am 22. Februar 1928 in *Oak Park in Illinois* geboren.

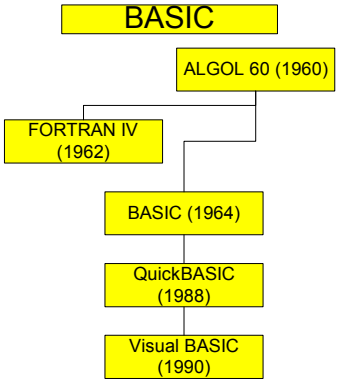
1956
Studium und Dissertation in Statistik.

1974-1981
Chairman of *American Bureau of Standards Committee X3J2*, welches der Entwicklung eines Standards für BASIC diente.

Auszeichnungen:

1974	AFIPS Pioneer Award.
1991	IEEE Computer Science Pioneer Award.

BASIC



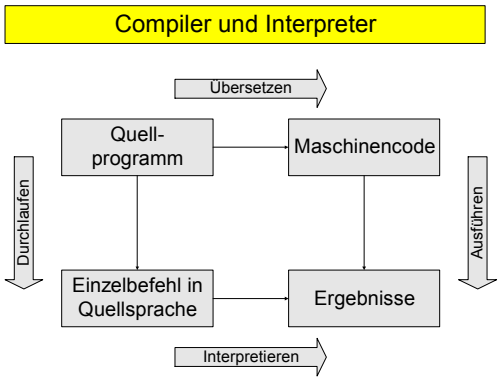
BASIC *(Beginner's All-purpose Symbolic Instruction Code)* wurde am Dartmouth College von John G. Kemeny und Thomas E. Kurtz für Studenten entwickelt. Es sollte eine interaktive Sprache erstellt werden, die leicht zu erlernen und schnell zu compilieren war und in der man Fehler leicht finden konnte. BASIC hat sich aus den Sprachen FORTRAN and ALGOL 60 entwickelt.

BASIC sollte folgende Ziele erreichen:

1. Es sollte leicht zu erlernen sein, besonders für nicht wissenschaftlich arbeitete Studenten.
2. Es sollte eine einfache Programmiersprache im Sinne ihrer Konzeption sein.
3. Es sollte für Hausarbeiten, also für Kleinrechner mit wenig Speicherplatz geeignet sein.
4. Es sollte für den Benutzer möglich sein, interaktiv zu programmieren. Der Benutzer soll nicht auf die Übersetzung der Programme warten müssen.

Variablen bestanden in BASIC entweder aus einem einzelnen Buchstaben oder aus einem Buchstaben, dem eine einzelne Ziffer folgt. Für die Erstellung schneller Programme schien dies ausreichend. Wie in FORTRAN werden Variablen nicht deklariert, und es wird nicht zwischen Ganzzahlen und reellen Zahlen unterschieden. Ein Programm wird durch Zeilennummern geordnet, wie das folgende BASIC-Programm zeigt.

```
 20   DIM A(ll)
 30   FOR I = 1 TO 11
 40   INPUT A(I)
 50   NEXT I
 60   FOR J = 1 TO 11
 70   LET I = 11 - J
 80   LET Y = A(I+l) + 5 * A(I+l)
 90   IF Y > 400.0 TREN 120
100   PRINT I+l, Y
110   GOTO 130
120   PRINT I, "ZU GROSS"
130   NEXT J
140   STOP
150   END
```



Compiler und Interpreter: Der Compiler erhält ein vollständiges Quellprogramm und liefert ein entsprechendes Maschinenprogramm, welches dann noch gebunden (link) werden muß. Bei der Programmausführung ist der Compiler unbeteiligt. Interpreter arbeiten anders: Das Programm wird nicht übersetzt, sondern in der Programmiersprache ausgeführt. Der Interpreter analysiert jeweils den anstehenden Befehl und interpretiert ihn, d.h. führt in direkt aus.

Unterstützt durch die Personalcomputer-Revolution, breitete sich BASIC schnell aus. Die Einfachheit von BASIC kommt all jenen Menschen entgegen, die den Computer zwar verwenden, nicht jedoch das Programmieren lernen möchten. Der Bekanntheitsgrad von BASIC brachte viele Implementationen der Sprache mit unterschiedlichen Eigenschaften hervor. Auch ein ANSI-Standard für BASIC im Jahre 1978 konnte die Situation nicht ändern. 1985 erstellten Kemeny und Kurtz eine neue BASIC-Version namens True BASIC, die neben vielen anderen neuen Fähigkeiten auch interaktive Grafik aufwies. Der Befehlsvorrat des Ur-BASIC ist folgender:

LET variable = ausdruck	{Wertzuweisung}
GOTO zeilennummer	{Sprung}
GOSUB zeilennummer	{Unterprogr.-Sprung}
RETURN	{Unterpr.-Ruecksprung}
IF ausdruck vergleichssymbol ausdruck THENzeilennummer	{Bedingter Sprung}
FOR variable = ausdruck TO ausdruck STEP ausdruck	{Laufschl.}
NEXT variable	{Ende Laufschleife}
READ variable, variable,	{Eingabe}
PRINT zeichenreihe-oder-ausdruck,zeichenreihe-oder-ausdruck, ...	{Ausgabe}
STOP	{Ende Programm-Lauf}
END	{Ende des Programms}
DIM variable (integer)	{eindim. Feld}
DIM variable (integer, integer)	{zweidim. Feld}
DATA zahl, zahl,…	{Konstanten}
REM kommentar	{Kommentarzeile}
DEF FN zeichen (variable) = ausdruck	{Funktion}

BASIC läßt sich leicht als Interpreter implementieren. Die Ausführungszeit kann bei der Interpretation länger sein. Dies ist vor allem dann sehr ineffizient, wenn einige wenige Befehle sehr oft durchlaufen werden. Der Implementierungsaufwand für einen Compiler ist beträchtlich höher, vor allem durch einen gewissen Grundaufwand. Der Sprachumfang beeinflußt einen Interpreter weitaus stärker als einen Compiler. Interpreter brauchen weniger Speicherplatz als Compiler, schon deshalb, weil sie keinen Maschinencode erzeugen. Das Entwickeln von Programmen mit Unterstützung von Interpretern ist für den Programmierer oder Anwender einfach, da der Interpreter sofort syntaktische, aber teilweise auch semantische Fehler anzeigt. Dies trifft bei der Schnelligkeit der heutigen Prozessoren auch auf die Entwicklung größerer Systeme zu.

1990 gab Microsoft in Form von VISUAL BASIC der Sprache BASIC einen neuen Aufschwung. VISUAL BASIC wurde aus QUICKBASIC (1988) entwickelt, ist leicht erlernbar wie das Original BASIC und es eignet sich in guter Weise für die Programmierung von verschiedenen Bildschirmasken unter Verwendung einer Menüleiste (Siehe Bild oben). Während VISUAL BASIC 3 noch kaum objektorientierte Eigenschaften aufwies, hat sich dies bei VISUAL BASIC 4/5/6 geändert.

15 [#] [$] [+] [K] Donald Knuth

1938
Donald Knuth wurde am 10. Januar 1938 in *Milwaukee in Wisconsin*, USA geboren.

1956
Donald Knuth studierte Mathematik und Physik am *Case Institute of Technology in Cleveland, Ohio,* wo er auch im Rechenzentrum an einer IBM 650 arbeitete und Assembler Programme, aber auch Übersetzer erstellte. Er studierte dort nebenbei auch analytische Geometrie.

1969
Im Sommer 1969 arbeitete er am *Pasadena Office of Burroughs*. Dort schrieb er Software, u.a. einen ALGOL Compiler. Er erhielt dafür 5500$.

1960
Ab 1960 arbeitete er am *California Institute of Technology*, wo er 1963 den Doktorgrad in Mathematik erhielt.

1962
Der Verlag *Addison Wesley* bat Knuth ein Buch über Computer und speziell Compiler zu schreiben.

1968, 1993-heute
Seit 1968 Professor an *der Stanford University* und seit 1993 *Professor Emeritus of The Art of Computer Programming.*

Auszeichnungen:

[#] Donald Knuth
[$] Donald Knuth
[+] auto
[K] Donald Knuth

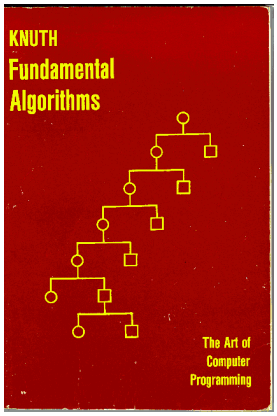
Donald Knuth hat unzählige Auszeichnungen erhalten. Die wichtigste Auszeichnung dürfte die *National Medal of Science* sein, die ihm 1979 von Präsident Carter verliehen wurde (siehe Bild rechts).

Bedeutende Publikationen:

Knuth, Donald E., "On the Translation of Languages from Left to Right," Information and Control, Vol. 8, 1965, pp. 607-639.
Knuth, Donald E., "Von Ncumann's First Computer Program," Computing Surveys, Vol. 2, 1970, pp. 247-260.
Knuth, Donald E., "Fortran: An Empirical Study of Fortran Programs," Software: Practice and Expe7ience, Vol. 1, 197 1, pp. 105-133.
Knuth, Donald E., "Ancient Babylonian Algorithms," Comm. ACM, Vol. 15, No. 7, July 1972, pp. 671-677 (Errata Vol. 19, No. 2, 1976, p. 108).
Knuth, Donald E., The Art of Computer Programming, 3 Vols., Addison-Wesley, Reading, Mass., 1968.
Knuth, Donald E., Computers and Typesetting, 5 Vols., Addison-Wesley, Reading, Mass., 1986.

Lehrbücher: *Art of Computer Programming (Kunst der Programmierung),* Analyse von Algorithmen, Erfinder von TeX, METAFONT

Die Leistung von Donald Knuth auf dem Gebiet der Computerwissenschaften kann nicht mit einem Satz beschrieben werden. Das Werk, welches ihn schon Mitte der 60-ziger Jahre berühmt machte, waren und sind die drei von sieben geplanten Bänden*: Donald Knuth: The Art of Computer Programming.*



Art of Computer Programming (Kunst der Computer Programmierung)

Zweck einer Sprache ist die Kommunikation. Menschen verwenden natürliche Sprachen, um untereinander zu kommunizieren. Programmiersprachen dienen der Kommunikation mit nüchternen Maschinen. Zumeist versetzen uns die Unterschiede bei diesen Kommunikationsarten in Staunen, in mancher Hinsicht sind sie sich jedoch sehr ähnlich. Beim Schreiben von Programmen verhält es sich wie beim Verfassen eines Aufsatzes; viele können in ihrer Sprache schreiben, doch nur wenige schreiben auch gut. In diesem Sinne hat Donald Knuth drei Bände seines geplanten sieben bändigen Riesenwerkes: *Die Kunst der Computerprogrammierung* geschrieben. Die sieben Bände sollten die folgenden Themen behandeln:

Band 1: Fundamental Algorithms.
Band 2: Seminumerical Algorithms.
Band 3: Sorting and Searching.
Band 4: Combinatorial Algorithms.
Band 5: Syntactical Algorithms.
Band 6: Theory of Languages.
Band 7: Compilers.

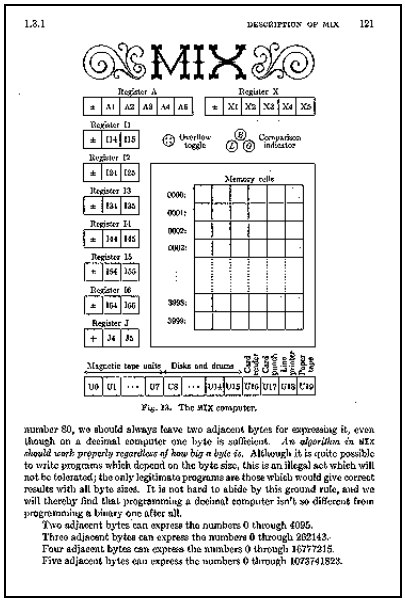
Der Computer IBM 650, mit dem Donald Knuth so schöne Abende verbrachte.

Knuth's Verbundenheit mit Computern und den Computerwissenschaften ist aus der Widmung im Band I zu ersehen:

> *The book is affectionately dedicated to the Type 650 computer once installed at Case Institute of Technology, with whom I have spent many pleasant evenings.*
> (Dieses Buch ist dem IBM 650 Computer gütigst gewidmet, einst installiert *am Case Institute of Technology,* mit dem ich so viele erfreuliche Abende verbracht habe.

Die drei Bücher: *Art of Computer Programming*, entstanden Mitte der sechziger Jahre, sie sind die berühmtesten Werke von Donald Knuth. Im Juni 1965 Knuth hatte den ersten Entwurf von ehemals zwölf geplanten Kapiteln fertiggestellt, es waren 3000 handgeschriebene Seiten. Die drei Bände wurden in viele Sprachen, wie Chinesisch, Russisch, Japanisch, Spanisch, usw. übersetzt.

Bemerkenswert ist, daß Knuth von *der Kunst des Programmierens* spricht. Seit der Erfindung des Computers standen Programmierer im Ruf Künstler zu sein, die in der Lage waren, die komplexe Situation des Programmierens intuitiv zu beherrschen und die auch immer eine Lösung, wenn auch trickreich, fanden. In dieser Weise bestimmten ihre Fähigkeiten, aber auch ihre Launen, die Qualität eines Programms.



1.3.1 DESCRIPTION OF MIX 121

MIX

Fig. 10. The MIX computer.

number 80, we should always have two adjacent bytes for expressing it, even though on a decimal computer one byte is sufficient. *An algorithm in* MIX *should work properly regardless of how big a byte is.* Although it is quite possible to write programs which depend on the byte size, this is an illegal act which will not be tolerated; the only legitimate programs are those which would give correct results with all byte sizes. It is not hard to abide by this ground rule, and we will thereby find that programming a decimal computer isn't so different from programming a binary one after all.

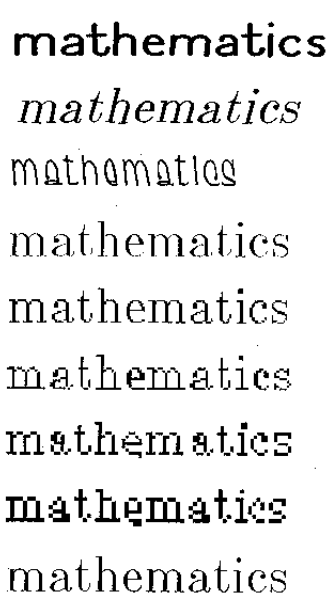
Two adjacent bytes can express the numbers 0 through 4095.
Three adjacent bytes can express the numbers 0 through 262143.
Four adjacent bytes can express the numbers 0 through 16777215.
Five adjacent bytes can express the numbers 0 through 1073741823.

Die unglaubliche Vielfalt von Techniken, Algorithmen, Theoremen diente als Grundstein für viele Kurse in den Computerwissenschaften. Das rechte Bild zeigt die Seite 121 aus dem ersten Band /KNUT72/, wo er eine virtuellen Computer MIX diskutiert.

Erfinder von TeX

Knuth ist der Erfinder von TeX, der ersten standardisierten Sprache für Computer Typographie. TeX ist eine der bedeutendsten Erfindungen in der Historie des Setzens von Buchmanuskripten. Es wurde mit der Erfindung des Buchdrucks von Gutenberg verglichen. TeX erlaubt die Kreation von Buchstaben jeder Art in hoher Eleganz und Genauigkeit. Eine TeX Benutzergruppe wurde 1979 etabliert und umfaßte 1989 mehr als 3000 Teilnehmer.

Das rechte Bild mit dem Begriff *mathematics* demonstriert Knuth's Unzufriedenheit mit seinem und den anderen existierenden Textsystemen. Von Computern erzeugte Texte sind zu genau, die Unregelmäßigkeiten der durch konventionelle Setzmaschinen erzeugten Texte geht verloren. Knuth versuchte, solche Unregelmäßigkeiten durch zufällige Unregelmäßigkeiten im Text zu erzeugen.

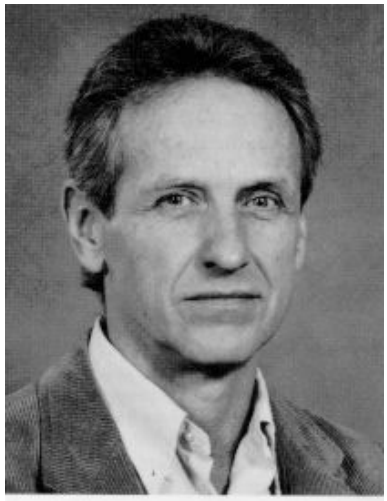


METAFONT

METAFONT erlaubt einem Entwickler die elektronische Kreation eines Schriftsatzes (Fonts) in sehr kurzer Zeit. In der Vergangenheit dauerte die Entwicklung eines Schriftsatzes Wochen oder gar Jahre. 1986 veröffentlichte Knuth das fünfbändige Werk*: Computers and TypeSetting*. Band I ist eine Benutzerhandbuch (User's Guide) für TeX, Band II enthält den kompletten Quellkode für TeX, Band III sind ein Benutzerhandbuch für METAFONT inklusive des Quellkodes, and Band V enthält mehr als 500 Beispiele für die Programmierung von METAFONT.

16 [#] [+] [$] [K] Alain Colmerauer; Philippe Roussel; Robert Kowalski

Alain Colmerauer (Bild); Philippe Roussel und Robert Kowalski entwickelten in den frühen 70er Jahren an der Universität von Marseilles, Frankreich, die Sprache PROLOG. Die Sprache wurde PROLOG genannt (PROgrammation en LOGique), weil sie auf dem Auflösungsprinzip von Robinson beruht. Das Auflösungsprinzip besteht aus einer einzigen Regel, aus der wahre Formeln der Prädikatenlogik hergeleitet werden können.



Robert Kowalski

PROLOG ist ein Beispiel für eine sogenannte nichtprozedurale Sprache. Eine *nichtprozedurale Sprache* führt nicht einen Schritt nach dem anderen aus, um ein Ergebnis zu berechnen, sondern legt fest, welche Eigenschaften das Resultat besitzen soll. Die Ausführung dieser Festlegung führt zur Lösung. Betrachten Sie das Problem, ein Feld zu sortieren. In einer prozeduralen Sprache muß der Programmierer einen Algorithmus, sagen wir QuickSort, angeben, um ein korrektes Ergebnis zu erhalten. In einer nichtprozeduralen Sprache hingegen braucht der Programmierer nur festzulegen, daß das Feld in einer nichtabsteigend
Dies stellt dann einen interessanten Ansatz zur Lösung eines Problems dar, wenn das allgemeine Verfahren die Lösung auf effiziente Weise anbietet.

Wir betrachten ein kleines Beispiel:

Die Programmierung in PROLOG beruht auf der interaktiven Verständigung zwischen dem Anwender und dem PROLOG-System, in dem der Anwender z.B. folgendes vornehmen kann:

1. Fakten für Objekte und Beziehungen deklarieren,
2. Regeln für Objekte und Beziehungen definieren sowie
3. über Objekte und Beziehungen Anfragen stellen.

Es folgen einige Beispiele für die Deklarierung von Fakten.

Valuable (Gold).	/*	Gold ist wertvoll.	*/
Valuable (Money).	/*	Geld ist wertvoll.	*/
Father (John, Mary).	/*	John ist der Vater von Mary.	*/
Gives (John, Book, Mark).	/*	John gibt Mark dasBuch.	*/
King (John, France).	/*	John ist König vonFrankreich.	*/
Iam.	/*	Ich bin.	*/

[#] Alain Colmerauer - Philippe Roussel - Robert Kowalski
[+] auto
[$] Alain Colmerauer - Philippe Roussel - Robert Kowalski
[K] Alain Colmerauer - Philippe Roussel - Robert Kowalski

In diesen Beispielen haben wir die Relationssymbole *Valuable, Father, Gives, King* sowie *Iam* groß geschrieben. Dasselbe haben wir mit den atomaren Objekten *Gold, Money, John, Book, Mark* und *France* getan.

Wir haben nun eine sehr kleine PROLOG Datenbank und können an das PROLOG-System einige Fragen stellen:

Valuable (Gold)? /* Ist Gold wertvoll*/
Yes

Wir erhalten hier eine positive Antwort, da das Literal Valuable (Gold) in der Liste der sechs Fakten auftaucht.

Wir können auch fragen, wer Vater von Mary ist:

Father (x,Mary)?
x=John

Wir erhalten als Antwort John. Wir können aber auch fragen:

Valuable (x)?
x=Gold
x=Money

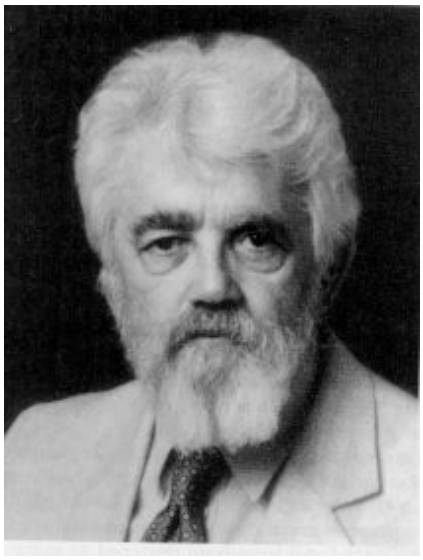
Hier erhalten wir als Antwort Gold und Money. Wir können aber auch fragen, ob zwei oder mehr Fakten gleichzeitig wahr sind. Die folgende Anfrage ermittelt, ob John irgend etwas wertvolles an Mark übergibt.

Gives (John,x,Mark), Valuable (x)?
Die Antwort ist negativ, da es kein Objeckt x gibt.

Die meisten heutigen PROLOG Implementierungen sind Interpreter. Sobald jedoch große Anwendungen laufen und man die interpretative Verarbeitung, d.h. langsame Verarbeitung, zu spüren bekommt, ist das Interesse für einen PROLOG-Compiler geweckt. Einige Anbieter haben diese Marktnachfrage erkannt und bieten für unterschiedliche Rechner PROLOG-Compiler an. Da PROLOG-Implementierungen bereits kommerziellen Software-Standards entsprechen und eine brauchbare Anwenderumgebung haben, z.B. IF/PROLOG, wird PROLOG immer öfters eingesetzt. Praktischen Einsatz findet PROLOG an den Universitäten und in kommerziellen Forschungseinrichtungen beim Bau von Expertensystemen. Expertensysteme nehmen ein besonders großes Gebiet im Bereich PROLOG-Anwendungen ein, da die Struktur eines PROLOG-Programm-Systems dem Aufbau eines Expertensystems entspricht.

17 [#] [$] [+] [K] John McCarthy

1948
Doktorgrad in Mathematik an der *Princeton University in Virginia.*



1951-1953
Procter Fellow an der *Princeton University,* und *Higgins Research Instructor.*

1962-heute
Director of the Stanford Artificial Intelligence Laboratory (SAIL)

1987-heute
Charles M. Pigott Professor of Computer Science in Stanford.

Auszeichnungen

ACM Turing Award, 1971, President, American Association for Artificial Intelligence (AAAI), 1983-1984; First Research Excellence Award, International joint Conference on Artificial Intelligence, 1985; IEEE Computer Society Pioneer Award, 1985; member, National Academy of Engineering, 1987; Kyoto Prize, Inamori Foundation, 1988; member, National Academy of Sciences, 1989; National Medal of Science, 1990; fellow, American Association for Artificial Intelligence (AAAI), 1990; member, American Academy of Arts and Sciences.

Bedeutende Publikationen

McCarthy, John, "A Time-Sharing Operator Program for Our Projected IBM 709," unpublished memorandum, MIT, Cambridge, Mass., reprinted in Ann. Hist. Comp., Vol. 14, No. 1, 1992, pp. 20-21.
McCarthy, John, Paul Abrahams, Daniel Edwards, Timothy Hart, and Michael Levin, LISP 1.5 Programmer's Manual, MIT Press, Cambridge, Mass., 1962.
McCarthy, John, S. Boilen, E. Fredkin, and J.C.R. Licklider, "A Time-Sharing Debugging System for a Small Computer," Proc. Spyingjoint Computer Conf., Vol. 23, Spartan Books, Washington, D.C., 1963, pp. 51-57.
McCarthy, John, "A Basis for a Mathematical Theory of Computation,' in Braffort, P., and D. Hischberg, eds., Computer Programming and Formal Systems, North-Holland, Amsterdam, 1963, pp. 33-70.
McCarthy, John, "Programs with Common Sense," in Minsky, M., ed., Semantic Information Processing, MIT Press, Cambridge, Mass., 1968.
McCarthy, John, and P.J. Hayes, "Some Philosophical Problems from the Standpoint of Artificial Intelligence," in Michie, D., ed., Machine Intelligence 4, Ainerican Elsevier, New York, 1969.
McCarthy, John, "Ascribing Mental Qualities to Machines," in Ringle, Martin, ed., Philosophical Perspectives in Artificial Intelligence, Harvester Press, July 1979.
McCarthy, John, "Applications of Circumscription to Formalizing Common Sense Knowledge," Artificial Intelligence, Apr. 1986.
McCarthy, John, "Generality in Artificial Intelligence," Comm. ACM, Vol. 30, No. 12, 1987, pp. 1030-1035; reprinted in A CM Turing Award Lectures: The First Twenty Years, ACM Press, New York.

[#] John McCarthy
[$] John McCarthy
[+] auto
[K] John McCarthy

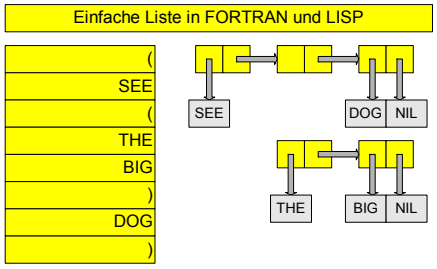
LISP

LISP gehört mit FORTRAN und COBOL zu den ältesten noch verwendeten Programmiersprachen. Die Geschichte von LISP ist eng mit John McCarthy verbunden. Sie beginnt Ende der 50er Jahre am MIT (*Massachussetts Institute of Technology*). John McCarthy suchte nach einer Programmiersprache, mit der er seine Ideen im Forschungsbereich der künstlichen Intelligenz ausprobieren konnte. Es handelte sich dabei um den sogenannten *Advice Taker*, ein lernfähiges Programm, das mit Hilfe eines menschlichen Lehrers sein Wissen erweitern kann. Bis heute konnte allerdings noch keiner ein Programm schreiben, das diesen Traum erfüllt. Der *Advice Taker* stellte bestimmte Forderungen an die zu verwendende Programmiersprache, die damals von keiner existierenden Sprache erfüllt wurden. Zum einen sollten Ausdrücke für den *Advice Taker* symbolisch (und nicht ausschließlich numerisch) darstellbar sein. Und zum anderen sollten Manipulationen auf diesen Ausdrücken durch Funktionen im mathematischen Sinne durchführbar sein. Listen erfüllen die erste Forderung. Rekursion, Konditional und Funktionen, die Listen manipulieren, erfüllen die zweite. Damit war LISP geboren.

Nicht ganz: bevor es einen LISP-Interpreter oder -Compiler gab, kam John McCarthy auf die Idee, formal zu beweisen, daß seine geplante Programmiersprache gleichmächtig zu Turing-Maschinen war. Dies ist eine wichtige Eigenschaft, wenn man an die Idee des *Advice Taker* denkt. Mit LISP sollte sich alles berechnen lassen, was sich nur berechnen läßt. Er ließ sich von der Idee der *Universellen Turing-Maschine* inspirieren, d.h. der *Turing-Maschine*, mit der sich alle anderen Turing-Maschinen simulieren lassen.

Er schrieb also ein (Papier-) Programm in LISP, das in der Lage ist, jedes beliebige LISP-Programm auszuführen (evaluieren sagen wir heute, nachdem J. McCarthy diese LISP-Funktion EVAL genannt hatte). Einer seiner Programmierer, S. R. Russel, las den Code und meinte, dieses Programm lasse sich programmieren. Worauf J. McCarthy antwortete, er verwechsle wohl Theorie mit Praxis: dies sei ein theoretisches Programm.

LISP ist eine Sprache, deren vorrangige Datenstruktur eine Liste mit Symbolen war. Eine solche Symbolliste konnte die Wörter eines Satzes wiedergeben, eine Liste von Attributen, ein Lohnkonto oder eine symbolische Differentialgleichung. Zunächst scheint es merkwürdig, daß eine so komplizierte Datenstruktur wie eine Liste als wichtigste Datenstruktur einer Sprache dienen könnte. Eine Liste besteht jedoch nur aus einem *nil*-Element und einem Paar-Element - einem Zeigerpaar, von dem einer auf ein Element der Liste zeigt, der andere auf den Rest der Liste. Das Beispiel rechts zeigt eine Liste in FORTRAN und in LISP.

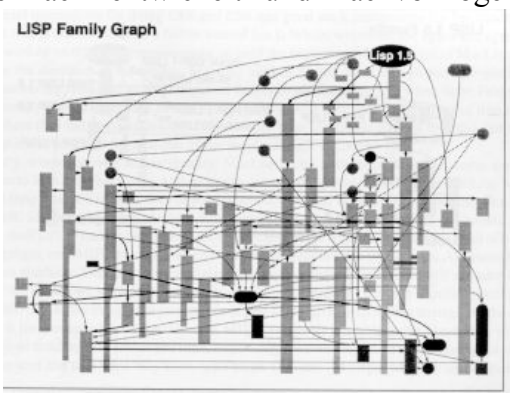


Eigentlich sind Programme selbst Listen von Symbolen. LISP stellt sowohl Daten als auch Programme als Listen dar. Eine einheitliche Darstellung von Daten und Programmen wie diese gibt es nur in LISP. In LISP kann ein Programm ein anderes Programm (eine Liste) genau wie einen Wert erstellen und dann ausführen.

LISP war für die an künstlicher Intelligenz interessierten Kreise schon immer von besonderer Bedeutung. Der ursprüngliche Anstoß für LISP lag in der Lösung allgemeiner Probleme, worin Menschen zwar gut sind, Computer sich jedoch ungeschickt anstellen. In den 60er und

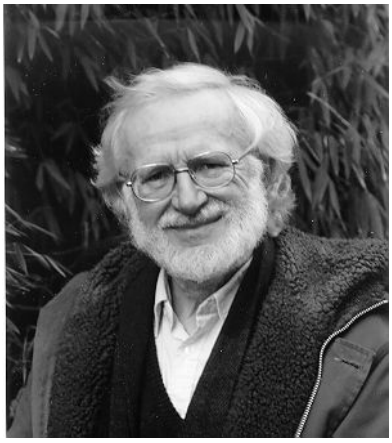
70er Jahren waren LISP und die künstliche Intelligenz eng miteinander verbunden. Inzwischen führte der Fortschritt bei der Compilertechnik und die klare Semantik der rein funktionalen Programmierung zur Bildung einer zweiten Hauptfamilie der Programmiersprachen, die sich von der ursprünglichen imperativen Familie unterscheidet.

Aufgrund der unterschiedlichen Syntax verhielt sich LISP ursprünglich wie alle herkömmlichen Sprachen, da die Programmierung eine Reihe von Nebeneffekten produzierte. LISP besaß jedoch einen konditionalen Aufbau und Funktionen. Die Bedeutung rekursiver Funktionen als Programmierungsmuster wurde mehr und mehr erkannt. Anstelle von Iteration, wie bei den DO-Schleifen unter FORTRAN, konnten rekursive Funktionen verwendet werden. Funktionen ohne Nebeneffekte können einfach entwickelt und nachvollzogen werden. Sprachen, die sich ausschließlich auf Funktionen und deren Anwendung verlassen, werden als *funktionale Sprachen bez*eichnet. Rein funktionale Sprachen sind selten; die meisten ermöglichen einige Nebeneffekte und Umordnungen.



Die Sprache LISP wurde im Laufe der Zeit (Siehe den *LISP Family Graph*: LISP-Familien Darstellung) verfeinert, genormt, und man einigte sich auf einige der zahlreichen Erweiterungen, die LISP bis dahin in viele Dialekte aufgeteilt hatten. Von 1975 bis 1985 wurden u.a. die Sprachen SCHEME und COMMON LISP entwickelt.

18 [#] [$] [+] [K] Peter Naur



1928
Geboren am 25. Oktober 1928 in Frederiksberg bei Kopenhagen.

1947-1949
Studium an der Universität Kopenhagen.

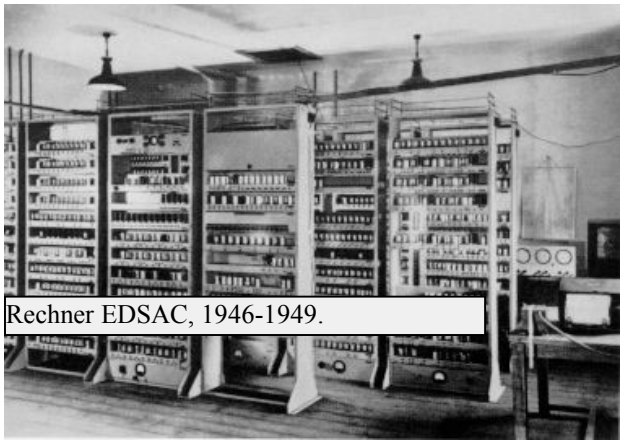
1950-1951
Research Student am *King's College, Cambridge, England.* Entwicklung eines Programs zur Untersuchung der Bewegung von Planeten und Kometen mit dem in England gebauten Rechner EDSAC.

1969-heute
Professor an der *Copenhagen University Institute of Datalogy*.

Auszeichnungen

G.A. Hageman Medal, 1963. Jens Rosenkjar Prize, 1966. Computer Pioneer Award of the IEEE Computer Society, 1986.

Bedeutende Publikationen


Rechner EDSAC, 1946-1949.

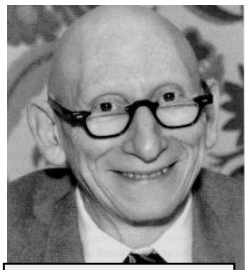
Neben vielen bedeutenden Veröffentlichungen auf dem Gebiet der Astronomie, des Buches: *Peter Naur: Concise Survey of Computer Methods, 1974*, kann das Buch: *Peter Naur: Computing: A Human Activity*, *ACM Press*, als die beste Zusammenfassung von Peter Naur's Veröffentlichungen angesehen werden.

Nach einer frühen Karriere in Astronomie, arbeitete Peter Naur für zehn Jahre in der Computer Software Industrie, bevor er zu seiner jetzigen Position als Professor von *Datology at Kopenhagen Universität,* ernannt wurde.. Er war u.a. Ehrenpräsident von *Dansk Selskab für Datalogi,* Mitglied von mehreren IFIP Arbeitsgruppen, und ist Mitherausgeber von BIT, der des *Nordic Journal for Numerical Mathematics and Computer Science.*

ALGOL 60, Backus-Naur Notation, Computing: A Human Activity, Rekursivität

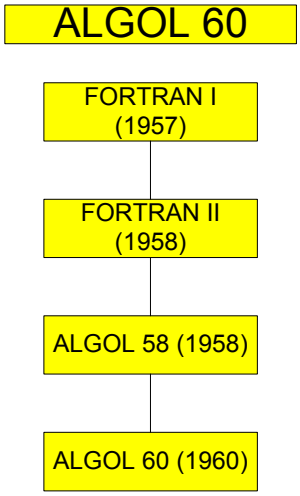
[#] Peter Naur
[$] Peter Naur
[+] auto
[K] Peter Naur

Die Programmiersprache ALGOL 60 wurde unter wesentlicher Mitwirkung von Peter Naur, aber auch Wissenschaftlern, wie Alan Perlis[5], Klaus Samuelson, Rutishauser und Friedrich Bauer, entwickelt. Obwohl es in den Vereinigten Staaten nie sehr bekannt wurde, hatte ALGOL einen bedeutenden Einfluß auf die Entwicklung von Programmiersprachen. Viele Sprachen, wie PASCAL und ADA, sind ALGOL-ähnlich. Dies bedeutet, sie weisen folgende Eigenschaften auf:


Alan Perlis

- Variablen werden geändert.
- Blöcke und Prozeduren sind die Grundbausteine.
- Prozeduren können sich selbst rekursiv aufrufen.
- Daten werden in verschiedene Typen untergliedert.
- Bezeichner haben lexikalische Reichweite.

Es ist keine Übertreibung zu behaupten, daß die meisten Programmiersprachen diese Eigenschaften aufweisen und man sie daher als von ALGOL abgeleitet bezeichnen kann. ALGOL (kurz für *ALGOrithmic Language*) wurde von einem Komitee unter wesentlicher Mitwirkung von Peter Naur entwickelt. In den späten 50er Jahren wollten Komitees der GAMM (*Gesellschaft für angewandte Mathematik und Mechanik*) und der ACM (*Association for Computing Machinery*) eine universelle Programmiersprache entwickeln, mit deren Hilfe Programme mit Anwendern und Computern kommunizieren konnten. Im Jahre 1958 trafen sich Mitglieder beider Organisationen an der Eidgenössischen Technischen Hochschule in Zürich. Von europäischer Seite waren u.a Bauer und Samuelsen vertreten. Sie erstellten den ersten Entwurf der neuen Sprache. Die Amerikaner bezeichneten sie zunächst als IAL (*International Algebraic Language*), bekannt wurde sie jedoch bald als ALGOL. 1960 trafen sich die Mitglieder beider Organisationen erneut, diesmal in Paris. Vorschläge für Blöcke, Wertübergabe und Adreßübergabe sowie die Rekursion wurden in die Sprache aufgenommen. Dabei kam eine revolutionäre neue Sprache heraus. Das Verfahren des Entwurfs und der Implementation der Sprache hat viel zum Feld der Programmiersprachen beigetragen. Um die Syntax von ALGOL zu beschreiben, wurde die BNF-Schreibweise (Backus-Naur-Form) entwickelt, und für ALGOL gab es erstmals eine Übersetzertechnik für blockstrukturierte Sprachen.



In den 60er Jahren gab es eine Menge Diskussionen und Unstimmigkeiten über eine Überarbeitung der Sprache. Allmählich entwickelte ein Komitee der *International Federation of Information Processing* die Version ALGOL 68. Schon zuvor stellte Niklaus Wirth eine neue Version von ALGOL 60 vor, die ALGOL W genannt wurde. ALGOL 68 besaß eine lange Reihe von Merkmalen. Parallele Berechnung und Semaphoren wurden verwendet. Es besaß implementationsabhängige Konstanten (z.B. die Obergrenze für ganze Zahlen) und Vorgaben. Die Sprache verfügte über einen großen Vorrat an Datentypen: komplexe Zahlen, Bitmuster, lange und kurze Zahlen, Zeichenketten und flexible Felder. Sie besaß anwenderdefinierte Überlagerungen und die von Hoare erfundene case-Anweisung. ALGOL 68 vereinfachte die barocke for-Konstruktion, die in ALGOL 60 eingeführt worden war.

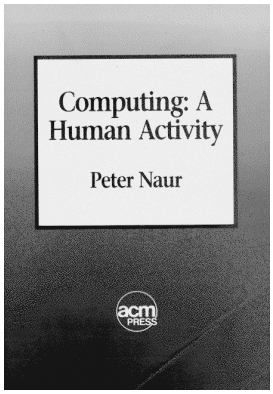
[5] Alan Perlis wurde am 1. April 1922 in Pittsburg geboren und starb am 7. Februar 1990 in New Haven / Conneticut. Perlis gilt als einer der Computerpioniere, der die Computerwissenenschaften als eine eigenständige Disziplin eingeführt hat. Er war in die Definitionen von ALGOL60 eingebunden, hat aber auch entscheidende Beiträge zu dem Gebiet des *empirical software engineering* geleistet.

Als die Vielseitigkeit der Computerprogrammierung immer größer wurde, ging der Traum von einer universellen algorithmischen Sprache jedoch langsam zu Ende. ALGOL 68 wurde nicht sehr häufig eingesetzt. Die Sprache war schwer zu implementieren: Ein Compiler der Sprache benötigte sechs Durchgänge. Nur die frühere Sowjetunion besaß einen offiziellen Standard für ALGOL 68. Hat ALGOL 68 eine Zukunft? Die ehrliche Antwort muß heute *Nein* lauten. Die Welt hat sich weiter entwickelt. Um heute akzeptiert zu werden, benötigt ihre Sprache Module, Ausnahmebehandlung, Polymorphie, sichere Parallelbearbeitung und saubere Schnittstellen.

Backus-Naur-Notation

Zur Beschreibung der Syntax von ALGOL 60 wurde von Backus eine eigene Schreibweise entwickelt. Diese (und einige verwandte Versionen) sind unter dem Namen BNF bekannt, was zu Ehren von John Backus und Peter Naur, ihren Erfindern, für Backus-Naur Form (BNF) steht. Backus führte die Beschreibung von ALGOL58 1959 auf der Konferenz ACM-GAMM (Gesellschaft für Mathematik und Mechanik), die sich mit der Spezifikation von ALGOL beschäftigte, ein. Die BNF-Definition wurde später von Naur leicht modifiziert und zur Beschreibung der Syntax von ALGOL60 verwendet.

Computing: A Human Activity



Peter Naur betrachtete die Entwicklung von Programmen immer als eine menschliche Aktivität. Naur meinte, daß die Schwierigkeiten bei der Erstellung großer Softwaresysteme nicht alleine durch neue Methoden bei der Softwareenetwicklung gelöst werden können. Viele dieser Fragen werden von Peter Naur im Buch: *Peter Naur: Computing: a Human Activity, ACM Press*, diskutiert.

Rekursivität

Wer je in einer Spiegelgalerie sich fortwährend gespiegelt hat, weiß intuitiv, was Rekursivität ist. Viele Sprachen enthalten die Möglichkeit des rekursiven Aufrufs von Prozeduren, wie dies auch in ALGOL60, im Gegensatz zu FORTRAN, aber auch schon im Plankalkül von Zuse, vorgesehen war. Heutzutage beherrschen auch einfache Sprachen, wie z.B. VISUAL BASIC den rekursiven Aufruf. In: *Bauer; Goos: Informatik, 1971, Springer Verlag,* findet sich eine *hübsche* Erklärung des Begriffes Rekursivität: **…***daß dieser Wurm an Würmern litt, die wiederum an Würmern litten (*Joachim Ringelnatz).

19 [#] [$] [+] [K] Dennis Ritchie; Kenneth Thompson

Dennis Ritchie wurde am 9. September 1941 in *Mount Vernon, New York,* geboren, er machte 1963 das Examen in Physik an der *Harvard University und arbeitet seit* 1968 bei den *Bell Laboratories.*

Ken Thompson wurde am 4. Februar 1943 in *New Orleans, Louisiana* geboren und arbeitet seit 1966 bei den *Bell Laboratories*

UNIX, Programmiersprache C

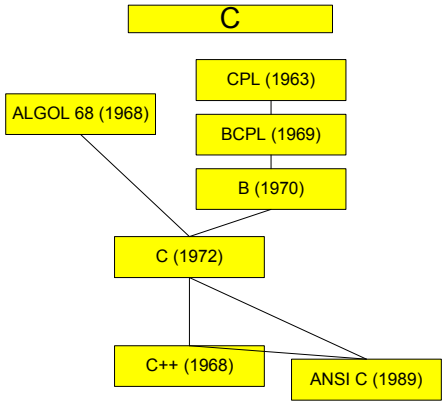
UNIX

Dennis Ritchie und Kenneth Tompson gelten als die Erfinder des UNIX Systems. UNIX ist ein Betriebssystem, daß heutzutage weltweite Verbreitung gefunden hat. Die Entwicklung von UNIX geht zurück in die 60-ziger Jahre in die Bell Laboratories. UNIX war als Ersatz für MULTICS gedacht. UNIX war 1985 auf mehr als 270000 Rechnern installiert und 1984 mehr als 740 Universitäten hatten eine UNIX Lizenz. Die Entwicklung der Programmiersprache C ist ein Seiteneffekt von UNIX.

Programmiersprache C

Die Programmiersprache C wurde ursprünglich 1972 von Dennis M. Ritchie bei den *Bell Laboratories* in New Jersey entwickelt und implementiert. Sie wurde stark von der Programmiersprache B beeinflußt, die Ken Thompson für das erste UNIX-System auf dem PDP-11 entwickelt hatte. Die Programmiersprache B war wiederum stark von BCPL geprägt, einer Systemprogrammiersprache, die zum Compilerbau entwickelt worden war.

[#] Dennis Ritchie - Kenneth Thompson
[$] Dennis Ritchie - Kenneth Thompson
[+] auto
[K] Dennis Ritchie.- Kenneth Thompson

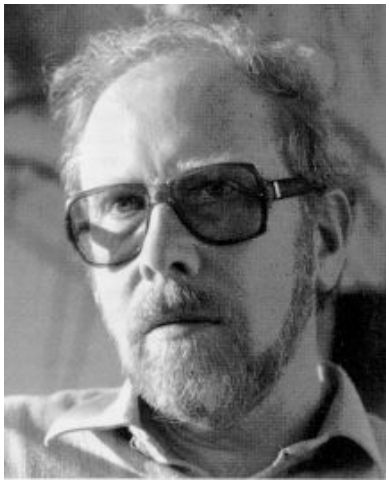


Nach zehn Jahren allmählichen Wachstums gab es in den 80er Jahren einen starken Anstieg bei der Verwendung von C. Eine große Zahl verschiedener Compiler wurde geschrieben, und man portierte Implementationen für beinahe jede Rechnerarchitektur und jedes Betriebssystem. Sowohl die Hobbyanwender mit ihren Personalcomputern als auch die Programmierer kommerzieller Software verwendeten häufig C. Man kann sich denken, warum dies der Fall war. Die konkurrierenden Programmiersprachen waren FORTRAN, COBOL und PASCAL. FORTRAN wurde und wird von den wissenschaftlichen Anwendern eingesetzt. In den 80er Jahren war diese Gruppe nicht gerade repräsentativ für die neuen Anwender von Computern. COBOL ist eher für umfangreiche Datenverarbeitungsoperationen geeignet, und auch diese Operationen nahmen nicht übermäßig zu. PASCAL war wesentlich einfacher zu erlernen als FORTRAN und COBOL und wurde sogar der nächsten größeren Generation der neuen Computeranwender gelehrt. Seine Anwendung ist einfacher und sicherer als die von C, doch aufgrund seiner *Isolation* von den *echten* Computeranwendungen ist es für einige Anwendungen nicht geeignet. Vor allem für grafische Anwendungen und Netzwerkprogrammierung benötigt man besseren Zugriff auf die Hardware-Umgebung. Diese Anwendungen wurden immer beliebter und bedeutender. C ist für derartige Anwendungen ideal, da es Zugriff auf Wörter und Bits des Computerspeichers auf niedriger Ebene ermöglicht und weil die *#include-Fähigkeit* das flexible Hinzufügen von systemabhängigen Definitionen erlaubt. C gehört zu den gewachsenen Sprachen, nicht zu den geschaffenen. Die Sprache neigt zu Seiteneffekten und damit zu schwer erkennbaren Fehlern.

Weiterentwicklung von C nach C++

Eine derzeit sehr bekannte objektorientierte Sprache ist C++. Bjarne Stroustrup wurde von SIMULA inspiriert, der Programmiersprache C in einem vorbereitenden Schritt des Compilers Klassen hinzuzufügen. Der Erfolg und die weite Verbreitung von C sowie das Fehlen jeglicher Unterstützung von abstrakten Datentypen in C führte zu einer weiten Verbreitung von C++. 1993 vergab ACM den *Grace Murray Hopper Award* an Stroustrup, den außergewöhnlichen jungen Computerprofi, für eine einzige technische Leistung. Im Jahre 1984 suchte Stroustrup Ideen für einen neuen Namen und wählte C++, weil es kurz war, schöne Interpretationen besaß und von der Form »Adjektiv C« abwich. In C kann ++ je nach Kontext als *Nächstes*, *Nachfolger* oder *Inkrement* gelesen werden, obwohl es immer *plus plus* ausgesprochen wird. Der Name C++ wurde von Rick Mascitti vorgeschlagen.

20 [#] [$] [+] [K] Niklaus Wirth



1934
Geboren in Winterthur, Schweiz.

1954-1958
Studium der Elektrotechnik an der ETH in Zürich.

1963
Doktorgrad der Elektrotechnik an der *University of California, Berkeley*, 1963.

1963-1967
Assistant Professor an der *Standford University*. Entwicklung der Programmiersprache PL360 (zusammen mit der *IFIP Working Group 2.1).*

1968-1970
Entwicklung der Programmiersprache PASCAL.

1979-1981
Entwicklung der Programmiersprache MODULA-2.

1972-heute
Professur an der ETH in Zürich.

1977,1985
Forschungsaufenthalte bei *XEROX Palo Alto Research Center*.

Auszeichnungen

Niklaus Wirth ist u.a. Träger des *ACM Turing Awards* 1984. Seine Rede zur Preisverleihung gibt Einblicke und überraschende Erkenntnisse, die er auf seiner Suche nach angemessenen Sprachformalismen zur Programmierung gewonnen hat. NELIAC, ein ALGOL-Dialekt, EULER, PL360 als Vorläufer von ALGOL W, PASCAL, MODULA-2 und schließlich der Personalcomputer Lilith - das sind die markanten Punkte auf Niklaus Wirths Suche nach verbesserten Programmiertechniken.

Bedeutende Veröffentlichungen

Wirth, N., "The Programming Language PASCAL," *Acta Informatica,* Vol. 1, June 1971, pp. 35-63.

[#] Niklaus Wirth

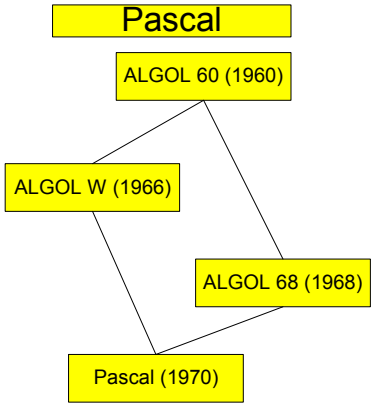
[$] Niklaus Wirth

[+] auto
[K] Niklaus Wirth

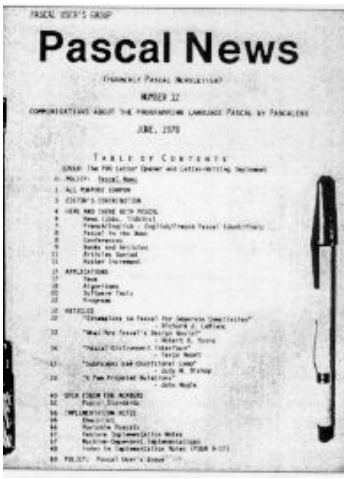
Hoare, C.A.R., and N. Wirth, "An Axiomatic Definition of the Programming Language PASCAL," *Acta In 'formatica,* Vol. 2, 1973, pp. 335~355.
Wirth, N., *PASCAL-User Manual and Report* (with Kathy jensen), Springer-Verlag, Berlin, 1974.
Wirth, N., *Algorithms, Data Structures, Programs,* Prentice~Hall, Englewood Cliffs, Nj., 1975.
Wirth, N., Programming in Modula-2, Springer-Verlag, Heidelberg, New York, 1982.
Wirth, N., "The Programming Language Oberon," *Software-Practice and ExpMence,* Vol. 18, No. 7,1985, pp. 671- 690.
Wirth, N., *Programming in Oberon* (with M. Reiser), Addison-Wesley, Reading, Mass., 1992.

ALGOL W, PASCAL, Modula-2, Oberon

Die Programmiersprache ALGOL hatte enorme Möglichkeiten demonstriert. All die Feinheiten mußten jedoch zu einem Ganzen vereint werden, das viele miteinander unvereinbare Anforderungen erfüllen sollte. So war einige Ordnung erforderlich, die Niklaus Wirth durchsetzte, der zunächst versuchte, die Entwicklung von ALGOL 68 durch seinen Vorschlag (ALGOL W) zu beeinflussen und dann mit der Entwicklung von PASCAL seinen eigenen Weg ging. Der erste Entwurf der Sprache besaß keine dynamischen Felder oder rekursive Prozeduren und wurde 1969 von einem einzigen diplomierten Studenten in FORTRAN auf dem CDC 6000 implementiert.



Eine zweite Version der Sprache, diesmal mit Rekursion, wurde in PASCAL selbst geschrieben. Die öffentliche Beschreibung der Sprache erschien 1971. In diesem Jahr wurde PASCAL in Programmierungsgrundkursen an der *Eidgenössischen Technischen Hochschule* in Zürich eingesetzt. PASCAL wurde auf der ganzen Welt eingesetzt, um Studenten das Programmieren zu lehren. Auch die Bemühungen um eine Norm Ende der 70er und Anfang der 80er Jahre konnten die Ausbreitung von Implementationen mit eigenen Erweiterungen nicht stoppen. Vor allem das Problem der dynamischen Felder, die unbedingt benötigt wurden, für die jedoch keine Lösung in Sicht war, bereitete Grund zur Sorge.



Mit dem Einfluß von PASCAL verbunden war auch der strukturierte Ansatz zur Programmethodik, der als *schrittweise Verbesserung*[6] *(stepwise refinement)* bezeichnet wird.

Anwenderdefinierte Datentypen und strukturierte Programmkonstruktionen, wie *if-* und *while* Anweisungen, läuteten ein neues Zeitalter der Programmierung ein, das auf einer *höheren Ebene* begann, als dies vorher möglich war. Man konnte bei der Formulierung des Problems

[6] Schon Descartes hat beschrieben, wie man ein komplexes Problem lösen könne; man zerlege es in Teile, und zerlege diese weiter, bis die Teile klein genung sind, um einfacher lösbar zu sein. Dieses rekursive Prinzip hat Wirth auf die Programmierung übertragen.

beginnen und dies in immer weitere Zwischenschritte gliedern, bis das Problem mit einem Computerprogramm gelöst war.

Die Axiomatische Definition von PASCAL

Die Existenz von PASCAL hat noch weitere bemerkenswerte Entwicklungen angestoßen und bestehende weiter vorangetrieben. Die Stärke der Sprache ALGOL 60 als akademische Antwort auf FORTRAN aus der Industrie Ende der fünfziger Jahre war ihre Syntaxbeschreibung auf der Theorie der formalen Sprachen. Hoare, in den letzten Jahren auch durch seine Warnungen vor ADA bekannt geworden, hatte sich gegen Ende der sechziger Jahre Gedanken über eine axiomatische Definition auch der Semantik einer Programmiersprache gemacht. Hoare und Dijkstra präsentierten 1973 die erste vollständige axiomatische Definition der Programmiersprache PASCAL.

Concurrent PASCAL

Concurrent PASCAL erweiterte die sequentielle Programmiersprache PASCAL mit nebenläufigen Prozessen. Die entscheidenden Arbeiten gingen von Brinch Hansen aus.

Modula-2, Oberon

Im Verlauf der 70er Jahre wurde deutlich, daß PASCAL und alle anderen Sprachen für eine Programmierung in großem Maßstab ungeeignet waren. Große Programme konnten nicht rein hierarchisch von einem engen Startpunkt bis zu einer breiten Basis gegliedert werden. Das Hauptproblem ist bis heute die Organisation dieser Riesen in Einheiten, die klar, effizient und nahtlos geschrieben und ausgetauscht werden können.

Die Programmiersprache MODULA-2 war Niklaus Wirths Antwort auf diese Forderungen. Die Version der Sprache, die als MODULA-2 bekannt ist, hat einen gewissen Bekanntheitsgrad erlangt, wenn auch keinen so großen wie PASCAL. Die Konkurrenz zu anderen Sprachen mit Modulen (vor allem ADA), anderen bekannten Sprachen wie C und C++ und die allgegenwärtigen Sprachen FORTRAN und COBOL ließen für MODULA-2 nur wenig Raum. Nun ging sogar Wirth mit seiner neuen Sprache OBERON mit typisierten objektorientierten Eigenschaften einen Schritt weiter.

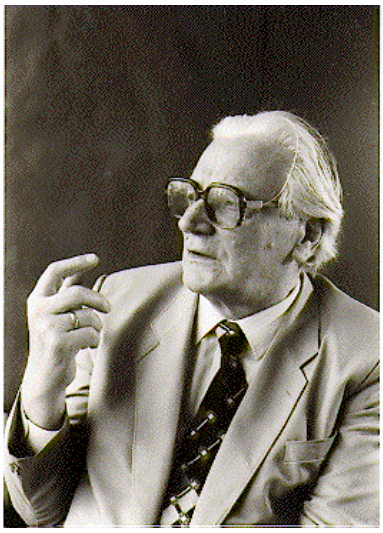
Entwicklung von ADA aus PASCAL

Ein weiteres wichtiges Ereignis der 80er Jahre war die Entwicklung von ADA, die auf u.a. auf den fundamentalen Prinzipien von PASCAL aufbaute. Seit Mitte der 70er Jahre unterstützte das amerikanische Verteidigungsministerium (DoD) einen Versuch, die wachsenden Software-Kosten zu verringern, die durch die hohe Zahl der von der Verteidigung verwendeten Sprachen verursacht wurden.

Es wurden 23 bereits bestehende Programmiersprachen nach diesen Zielen beurteilt. Da keine dieser Sprachen für ausreichend befunden wurde, wurde ein Zyklus von Sprachspezifikationen und überarbeiteten Entwürfen durchlaufen, um eine neue Sprache zu entwickeln. Fünf Firmen einigten sich schließlich darauf, konkurrierende Entwürfe der Sprache zu entwickeln. Ein Team von *Honeywell Bull* aus Frankreich erstellte unter der Leitung von Jean Ichbaih den Entwurf, der schließlich vom Verteidigungsministerium der USA (DoD) gewählt wurde. Zu Ehren von *Ada Augusta*, der *Gräfin von Lovelace*, wurde die Sprache mit Einverständnis ihrer

Nachkommen ADA genannt. Aufgrund ihrer gemeinsamen Arbeit mit Babbage an der analytischen Maschine wird sie als die erste Programmiererin betrachtet. Der Sprachstandard für ADA erschien 1983.

21 [#] [+] [$] [K] Konrad Zuse



1910
Konrad Zuse wurde am 22. Juni 1910 in Berlin (Wilmersdorf) geboren.

1928
Abitur am *Reform-Real-Gymnasium in Hoyerswerda*

1934
Beginn der Entwicklungsarbeiten von programmgesteuerten Rechenmaschinen

1935
Diplom-Hauptexamen an der Fakultät für Bauingenieurwesen, *Technische Hochschule Berlin Charlottenburg.*

Nach dem Studium Statiker bei den *Henschel-Flugzeugwerken* in Berlin-Schönefeld.

1936
In den Jahren 1936-1938; Fertigstellung der ersten noch ganz mechanisch arbeitenden Rechenmaschine, die Zuse Z1 (Versuchsmodell), deren Nachbau heute im *Museum für Verkehr und Technik* in Berlin steht.

1941
Nach Unterbrechung der Arbeiten durch Einberufung bei Kriegsausbruch entstand 1941 das Gerät Z3, der erste voll funktionsfähige programmgesteuerte Rechner der Welt (in elektromechanischer Relaistechnik). Ein Nachbau befindet sich heute im *Deutschen Museum* in München.

1942-1945/46
Entwicklung einer universalen algorithmischen Sprache unter der Bezeichnung *Plankalkül.*

1949
Gründung der ZUSE KG in Neukirchen (damals Kreis Hünfeld). 1957 wurde der Betrieb nach Bad Hersfeld verlegt. Mitarbeit an der Entwicklung weiterer programmgesteuerter Rechengeräte in elektromechanischer Technik, Röhren- und Transistortechnik (Modelle Z11, Z22, Z23, Z25, und Z31).

1959
Entwicklung eines sehr genau arbeitenden automatischen Zeichentisches (Graphomat 64), der die Anwendung von Rechenmaschinen im graphischen Bereich demonstriert.

1964
Ausscheiden aus der ZUSE KG als aktiver Teilhaber.

[#] Konrad Zuse

[+] auto

[$] Konrad Zuse

[K] Konrad Zuse

Ab 1966
Beschäftigung u.a. mit den theoretischen Grundlagen der Computer-Technik, wissenschaftlichen Veröffentlichungen, Biographie, Petri-Netze aus der Sicht des Ingenieurs, Veröffentlichung des Plankalküls 1971.

1995
Gestorben am 18. Dezember 1995 in Hünfeld.

Auszeichnungen

Konrad Zuse hat unzählige Auszeichnungen erhalten. Es sind eine Ehren- und eine Honorarprofessur und sieben Ehrendoktoren zu erwähnen. Fünfzehn inländische Auszeichnungen, acht Ehrenmitgliedschaften, das Ehrenbürgerrecht der Stadt Hünfeld, und fünfzehn Namensgebungen von Schulen und Häusern sind zu erwähnen. Die ausländischen Auszeichnungen sind die folgenden:

Ausländische Auszeichnungen / Ehrungen

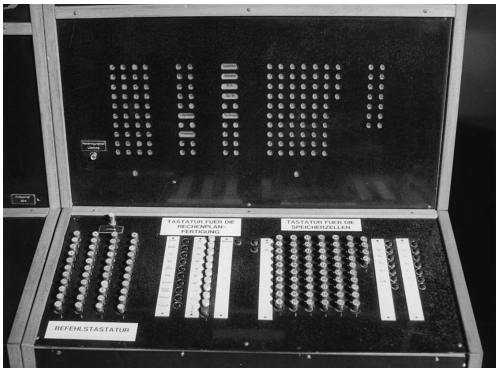
1965 *Harry-Goode-Memorial-Award* (AFIPS, Las Vegas)
1972 *Wilhelm-Exner-Medaille* (Österreichischer Gewerbeverein ÖCV, Wien)
1981 *Foreign Associate* (National Academy of Engineering CNAE, Washington)
1981 *Gustave-Transenster-Medaille* (Universite de Liege, Lüttich/Belgien)
1982 *Computer-Pioneer-Award* (The Institute of Electrical and Electronics Engineers Inc., IEEE, Cal.)
1999 Posthum Fellow des Computer museum History Center in Mountain View / California (Zusammen mit Mccarthy und Kay).

Bedeutende Publikationen

Zuse, K.: Über den Plankalkül als Mittel zur Formulierung schematisch-kombinativer Aufgaben. In: Archiv der Mathematik, Band 1 (1948/49)
Zuse, K.: Die mathematischen Voraussetzungen für die Entwicklung logistisch-kombinativer Rechenmaschinen. Sonderdruck aus "Zeitschrift für angewandte Mathematik und Mechanik", Band 29, Heft 1/2 (Jan./Feb. 1949)
Zuse, K.: Die Feldrechenmaschine. MTW-Mitteilungen, Nr. V/4 (1958)
Zuse, K.: Über den Plankalkül. Sonderdruck aus Elektronische Rechenanlagen, 1.
Jahrg., Heft 2. R. Oldenbourg-Verlag München (1959)
Zuse, K.: Entwicklungslinien einer Rechengeräteentwicklung von der Mechanik zur Elektronik. Hrsg.: Walter Hoffmann. Sonderdruck aus "Digitale Informationswandler". Friedrich Vieweg & Sohn Braunschweig (1962)
Zuse, K.: Über sich selbst reproduzierende Systeme. In: Elektronische Rechenanlagen,
Heft 2. R. Oldenbourg-Verlag München (1967)
Zuse, K.: Gesichtspunkte zur sprachlichen Formulierung in Vielfachzugriffsystemen unter Berücksichtigung des Plankalküls. NTG-Tagung "Teilnehmerrechensysteme". R. Oldenbourg-Verlag München (1968)
Zuse, K.: Rechnender Raum, Schriften zur Datenverarbeitung Band 1. Friedr. Vieweg &
Sohn Braunschweig (1969)
Zuse, K.: Der Plankalkül, BMBW-GMD-63. Berichte der Gesellschaft für Mathematik und Datenverarbeitung. Schloß Birlinghoven, St. Augustin (1972)
Zuse, K.: Gesichtspunkte zur Beurteilung allgemeiner. Sprachen, BMFr-GMD-105. Berichte der Gesellschaft für Mathematik und Datenverarbeitung mbH Bonn. Schloß Birlinghoven, St. Augustin (1975)
Zuse, K.: Ansätze einer 1 heorie des Netzautomaten, Nova Acta Leopoldina, Abhandlungen der Deutschen Akademie der Naturforscher Leopoldina, i. A. des Päsidiums Hrsg. von J.-H. Scharf, Director Ephemeridum der Akademie, Neue Folge, Nr. 220, Band 43 (1975)

Planfertigungsteil, Erste universelle Programmiersprache der Welt: Plankalkül, Schach

Planfertigungsgerät



Schon in der Z4 (1942-1945 in Berlin, dann ETH-Zürich von 1950-1955) war zur Unterstützung der Programmierung ein sogenanntes Planfertigungsteil[7] /ZUSE98/ realisiert. Mit diesem Gerät war es möglich, die Programmierung der Z4 in ca. drei Stunden zu erlernen. Dieses Gerät erlaubte eine Art symbolische Programmierung, ähnlich zu einem einfachen Assembler, der Z4, die ca. 20 Befehle (Siehe dazu /ZUSE98/) enthielt.

Plankalkül

In Hinterstein im Allgäu, wo Konrad Zuse von 1945-1947 mit seiner Frau lebte, schrieb er in fast völliger Isolation von der Außenwelt 1945/46 die Endfassung des *Plankalkül. Der Plankalkül*, basierend auf dem Aussagen- und Prädikatenkalkül, gilt als die erste universelle Programmiersprache der Welt für Computer.

Konrad Zuse Vision /ZUSE98/ war, den Plankalkül zusammen mit den logistischen Rechenmaschinen einzusetzen. Rechner, wie die Z3 und Z4, bezeichnete er als algebraische Rechenmaschinen. Mit Hilfe des Plankalkül sollten kombinatorische Aufgaben beschrieben werden und dann in Programme für algebraische Rechengeräte umgesetzt werden. Der zweite Weltkrieg machte Konrad Zuse's Vision zunichte (Siehe dazu auch /ZUSE98/).

Der Plankalkül wurde nie implementiert, er diente aber als Diskussionsgrundlage für Programmiersprachen, wie z.B. ALGOL58 / 60. Auch in der objektorientierten Sprache EIFFEL, wie in: *Robert Sebesta: Concepts of Programming Languages, Addison Wesley Publishing Company, 1996,* auf den Seiten 55 und 97, zu lesen ist, ist ein Kunstrukt von Konrad Zuse zu finden (ASSERTION, Siehe dazu die Einleitung).

Konrad Zuse nannte seine Programmiersprache *Plankalkül* bzw. *Planrechnung.* Er entwickelte eine Schreibweise mit drei Adressen. Die Schreibweise erstreckte sich über mehrere Zeilen. Indizes erschienen beispielsweise direkt unterhalb der Variablennamen. Das folgende Beispiel demonstriert eine Zuweisung des Ausdrucks A(4) + 1 nach A(5). Die Zeile gekennzeichnet mit V ist für den Index, und die Zeile gekennzeichnet mit S ist für den Datentyp. 1.n bedeutet eine Integerzahl bestehend aus n Bits.

```
| A + 1 =>      A
V| 4            5
S| 1.n          1.n
```

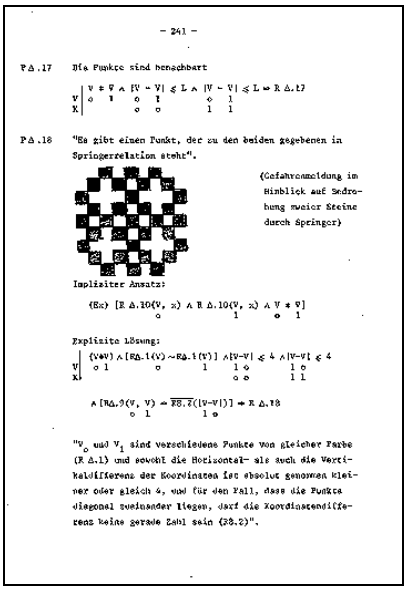
[7] Bei dem Rechner Z4 wurde der Lochstreifen auf einem sogenannten Planfertigungsteil erstellt. Dieser enthielt Tasten, welche mit den Befehlen beschriftet waren. Durch Niederdrücken der Taste wurde der entsprechende Befehl gelocht. Diese Arbeit gestaltete sich sehr einfach, da der Mathematiker nicht überlegen mußte, in welcher Weise die Maschine die Befehle verschlüsselt. Es bestand auch die Möglichkeit, einen Rechenplan zu kopieren. Das Planfertigungsteil war ein Vorläufer des Planfertigungsgerätes, mit dem eine symbolische Programmierung möglich sein sollte.

Mit Unterstützung der *Gesellschaft für Mathematik und Datenverarbeitung* (GMD) (Professor Krückeberg) und der *Siemens AG* (Professor Gumin) konnte Konrad Zuse das ursprüngliche Manuskript aus dem Jahr 1945/46 überarbeiten. Dabei übernahm er bewußt die zeitbedingten sachlichen Fehler. *Der Plankalkül* erschien schließlich 1972. Die Arbeit wurde ins Englische mit dem Titel: *The Plankalkuel*, übersetzt und erschien 1976. Danach schrieb K. Zuse zwei weitere Bücher zu diesem Thema: *Beschreibung des Plankalküls* und *Gesichtspunkte zur Beurteilung algorithmischer Sprachen.* 1972 haben es F.L. Bauer und H.Wössner unternommen, dem Plankalkül seinen Platz in der Geschichte der Computersprachen zuzuweisen. Der Titel ihrer Arbeit, die in *Communications of ACM* auch in englischer Fassung 1972 erschien, lautete: *The Plankalkül of Konrad Zuse: A Forerunner of Today's Programming Languages.*

Obwohl der Plankalkül erst 1972 international veröffentlicht wurde, spielte der Plankalkül 1958 eine Rolle bei der Definition des Zuweisungszeichens von Variablen beim Entwurf von ALGOL 58 unter der Leitung von Alan Perlis. Ideen des Plankalkül waren Rutishauser bekannt, der als einer der führenden Wissenschaftler an der ETH Zürich war, wo die Z4 von 1950-1955 aufgestellt war. Konrad Zuse hatte die Ergibtanweisung im Plankalkül eingeführt. Es wurde diskutiert, ob die Notation *expression => variable* verwendet werden sollte, wie Zuse es definiert hatte. Man entschied sich dann allerdings auf Druck amerikanischer Wissenschaftler für *variable := expression.*

Besonders interessant am Plankalkül war seine umfangreiche Sammlung an Datenstrukturen. Sie begann mit Daten von einem Bit Länge, deren Wert entweder 0 oder 1 war. Auch Datentypen für Fließkommazahlen (Gleitkommazahlen) konnten erstellt werden. Dabei konnte angegeben werden, ob es sich um eine reelle oder imaginäre Zahl handelte und ob sie positiv, null oder negativ war. Eine derart allgemeine und fortgeschrittene Datenstrukturierung wie im Plankalkül sollte es bis zum Erscheinen von ALGOL 68 (1968) und PASCAL (1971) nicht mehr geben. Der Plankalkül ist keine typische imperative Sprache wie z.B. PL/1. er muß mehr als ein logische Programmiersprache (z.B. PROLOG) gesehen werden, der auch auf dem Prädikatenlogik basierte.

Programmierung von Schachprogrammen



Konrad Zuse demonstrierte die Mächtigkeit des Plankalküls mit der Programmierung von Schachprogrammen (siehe Bild rechts). In seiner Biographie /ZUSE86/ schreibt Konrad Zuse u.a. über den Plankalkül:

Die Computerfachleute sprechen vom Programm, von formalen und algorithmischen Sprachen, von Spezialcodes, von systemorientierten Assemblersprachen usw.…Ich selbst habe schon während des Krieges und kurze Zeit danach versucht, eine algorithmische Formelsprache zu entwickeln, die die numerischen Rechnungen als Nebenzweck behandelt, in erster Linie aber rein logischen Kombinationen dient. Das Schachspiel diente mir dabei als ein gutes Modell. Nicht, weil ich es als besonders wichtig erachtete, daß man eines Tages einen Schachspieler mit dem Computer besiegt, sondern nur, weil ich ganz bewußt ein Gebiet

suchte, auf dem möglichst alle komplizierten und komplexen Kombinationen vorkommen (Bild aus: /ZUSE72/, Seite 241).

Ausführungen von Giloi über den Plankalkül

In /GILO97/ schreibt. Giloi u.a. über den Plankalkül (PK). *Oberflächlich gesehen nahm der PK wesentliche Züge der späteren algorithmischen Sprachen, wie ALGOL vorweg. So gibt es Entsprechungen für die gängigen Kontrollkonstrukte wie IF und REPEAT-UNTIL-Anweisungen. Und man findet auch alle in höheren Programmiersprachen übliche skalare Datentypen wie:*

- *boolean (logische Werte)*
- *integer (ganze Zahlen)*
- *real (relle Zahlen)*
- *complex (komplexe Zahlen)*

Eine genauere Betrachtung zeigt aber doch wesentliche Unterschiede zu den frühen höheren Programmiersprachen (die erst über ein Jahrzehnt später entstanden) auf, nämlich:

- *die Sichtbarkeit der binären Darstellung (representation) der genannten Datentypen*
- *die Existenz von Datenstrukturen*

Der Begriff des Datentyps und seiner Repräsentation ist zu erläutern. Mit Datentypen bezeichnet man Objektarten, mit denen in den Anwendungen gerechnet wird (z.B. logische Größen, Zahlen der verschiedensten Art, abstrakte Bezeichner, usw.), einschließlich der Operationen, die auf die einzelnen Objektarten angewandt werden können. Auf der Hardwareebene werden alle diese Objektarten immer durch Bitketten, d.h. Folgen von Nullen und Einsen, repräsentiert. Bereits bei den frühen höheren Programmiersprachen FORTRAN und ALGOL galt das Prinzip, daß der Programmierer nach Möglichkeit nur die Typen der Sprache sieht, nicht jedoch die sie repräsentierenden Bitketten (die nur der Maschine bekannt sein müssen). Das Verbergen der Datenrepräsentation gibt dem Programmierer eine abstrakte, anwendungsbezogene Sicht der Datentypen und macht so die Programmierung einfacher und fehlerfreier. Dadurch wird dem Programmierer andererseits die Möglichkeit genommen, eigene Typen definieren zu können.

Der PK kennt genau betrachtet nur einen elementaren Datentyp: das einzelne Bit. Alle anderen, anwendungsorientierten Datentypen werden vom Programmierer bereits als Bitmuster, d.h. als Strukturen deklariert, deren Interpretation nicht wie in den höheren Programmiersprachen a priori definiert sind, sondern erst durch den Programmierer zu lieferende Typenspezifikation. Dadurch sind die den Anwendungs-Datentypen zugrundeliegenden Bitmuster dem Benutzer sichtbar, und (im Vergleich zu Fortran und Algol) hat der PK hier eine niedrigere Abstraktionshöhe.

Andererseits gibt es im PK Datenstrukturtypen wie den binären Baum, das Array (Feld) und die Liste, insbesondere die Liste von Wertepaaren, die die Darstellung von generalisierten Graphen (d.h. beliebige Relationen) ermöglicht. damit können z.B. geometrische Strukturen aufgebaut werden. Strukturen können im PK dynamisch sein, d.h. während der Programmausführung erzeugt werden.

Listen sind immer dynamisch; sie können wachsen und schrumpfen. Die leere Variable dient als Platzhalter für dynamisch erzeugte Elemente, und es gibt Listenoperationen für

- *Erzeugen von Unterlisten aus denjenigen Elementen, die ein bestimmtes Prädikat erfüllen;*
- *Abfrage der Anzahl der Listenelemente;*
- *Lesen der ersten oder letzten Listenelementes;*
- *Suche nach dem kleinsten oder größte Element;*
- *Konkatenation von zwei Listen.*

Diese Züge des PK sind erst in den ein Jahrzehnt später entstandenen Sprachen LISP und APL erneut zu finden. LISP basiert auf der Liste als Grundtyp und wurde damit zur Sprache der künstlichen Intelligenz. APL (A Programming Language) war die erste implementierte Sprache für die dynamische Erzeugung und Manipulation von Datenstrukturen. Alle diese Eigenschaften geben dem PK einen sehr viel höheren Abstraktionsgrad, als ihn die üblichen höheren Programmiersprachen haben, die letztlich nur die Fähigkeit der von Neumann-Maschine abbilden, mit jedem Rechenschritt den Inhalt eines einzelnen Speicherplatzes transformieren, statt eine ganze Datenstruktur.

Konrad Zuse hatte bereits 1945 bei der Entwicklung des Plankalküls eine klare Vorstellung eines abstrakten Programmiermodells mit abstrakten Datentypen und Operationen und einer für numerische wie nicht-numerische Anwendungen gleichermaßen geeignete Datenräpresentation, Konzepte, die eigentlich erst in den siebziger Jahren ihren Einzug in die Informatik hielten.

22 [+#$K] Literatur

Die folgende Literatur diente zur Erstellung dieses Berichtes. Weitere Informationen zur Historie der Rechnerentwicklung (Computer), aber auch der Softwareentwicklung, können aus /ZUSE98/ entnommen werden.

/BAUE71/ Bauer, F.L.; Goos, G.: Informatik - Eine einführende Übersicht. Springer-Verlag, 1971.

/BAUE75/ Bauer, F.L.: Software Engineering. In: Software Engineering as an Advanced Course, Springer 1975.

/BAUE72/ Bauer, F.L.; Wössner, H.: Zuses Plankalkül, ein Vorläufer der Programmiersprachen - gesehen vom Jahr 1972. Elektronische Rechenanlagen, 14 (1972), Heft 3, S.111-118.

/BEAU68/ Beauclair, W. de: Rechnen mit Maschinen, Vieweg und Sohn, 1968.

/BERG96/ Bergin, Thomas, J.; Gibson, Richard, G.: History of Programming Languages, Addison-Wesley Publishing Company, 1996.

/BILL89/ Billings, Charlene, W.: Grace Hopper - Navy Admiral and Computer Pioneer. Enslow Publishers, Inc., Hillside, New Jersey, 1989.

/BOEH79/ Boehm, B.W.: Software Engineering. In: Classics in Software Engineering, Yourdon Press, 1979.

/CERR98/ Ceruzzi,, Paul, E.: A History of Modern Computing. MIT Press, 1998.

/DENN75/ Dennis, J.B.: The Design and Construction of Software systems. in: Software Engineering as an Advanced Course, Springer 1975.

/FAIR85/ Fairley, R.: Software Engineering Concepts. McGraw Hill, 1985.

/GILO97/ Giloi, W.: Konrad Zuse's Plankalkül: The First High-Level, „non von Neumann" Programming Language. IEEE Annals of the History of Computing, Vol. 19, No 2, 1997.

/HOAR89/ Hoare, C.A.E.; Jones, C.B.: Essays in Computing Science. Prentice Hall., 1989.

/HOHM75/ Hohmann, Joachim: Eine Untersuchung des Plankalküls im Vergleich mit algorithmischen Sprachen. Gesellschaft für Mathematik und Datenverarbeitung. Nr. 105, BMBW - GMD - 105, 1975.

/HOMM79/ Hommel, G.; Jäckel; Jähnichen, S.; Koch, W.; Koster, K.: ELAN – Sprachbeschreibung. Akademische Verlagsgesellschaft Wiesbaden, 1979.

[+] auto

[#] Literatur

[$] Literatur

[K] Literatur

/KNUT72/ Knuth, Donald: Fundamental Algorithms - The Art of Computer Programming. Addison-Wesley Publishing Company, 1972. Neu aufgelegt 1998.

/LEE.95/ Lee, J.A.N.: Computer Pioneers. IEEE Computer Society Press, 1995.

/LUDE86/ Ludewig, Ludwig: Sprachen für die Programmierung - Eine Übersicht. BI Hochschultaschenbücher, Band 622, 1986.

/MEYE88/ Meyer, Bertrand: Object-Oriented Software Construction. Prentice Hall, 1988.

/MUEL67/ Müller, K.H.; Sreker, I.: FORTRAN IV - Programmierungsanleitung. BI - Hochschulskripten, 1967.

/PARN74/ Parnas, D.L.: Software engineering or Methods for the Multi-Person Construction of Multi-Version Programs. Lecture Notes in Computer Science, Programming Methodology, Springer 1974.

/SAMM69/ Sammet, J.: Programming Languages: History and Fundamentals. Prentice Hall, Englewood Cliffs, New Jersey, 1969.

/SCHÄ71/ Schärf, Julius: Programmieren - Leicht und Schnell Erlernbar - Einführung in BASIC. Oldenburg Verlag, 1971.

/SEBE96/ Robert Sebesta: Concepts of Programming Languages, Addison Wesley Publishing Company, 1996, pp. 55, 97.

/SHAS95/ Shasha, Dennis; Lazere, Cathy: Out of Their Minds - The Lives and Discoveries of 15 Great Computer Scientists. Copernicus, An Imprint of Springer-Verlag, 1995.

/SIEF98/ Siefkes, D.; Braun, A., Eulenhöfer, P.; Stach, H.; Städler, K.: Pioniere der Informatik. Springer Verlag, 1999.

/SOMM85/ Sommerville, I.: Software Engineering. Addison Wesley, 2. Edition, 1985.

/STAN95/ Stansifer, Ryan: Theorie und Entwicklung von Programmiersprachen. Prentice Hall Publisher, 1995.

/SWAD93/ Swade, Doron, D.: Der mechanische Computer des Charles Babbage. Spektrum der Wissenschaft, April 1993.

/VORN82/ Vorndran, Edgar, P.: Entwicklungsgeschichte des Computers, VDE-Verlag GmbH, 1982.

/ZUSE45/ Zuse, Konrad: Theorie der angewandten Logistik - 2. Buch. der Zuse Apparatebau Berlin, 1944.

Bemerkung von Horst Zuse: Es kann heutzutage nicht exakt festgestellt werden, ob dieses Werk 1945 oder 1945 vollendet wurde. Sicher ist, daß Konrad Zuse die Arbeiten am Plankalkül 1942 begonnen hat.

/ZUSE48/ Zuse, Konrad: Über den Plankalkül als Mittel zur Formulierung schematisch-kombinativer Aufgaben. In: Archiv Mathematik, Band I (1948/49).

/ZUSE72/ Zuse, Konrad: Der Plankalkül. Gesellschaft für Mathematik und Datenverarbeitung. Nr. 63, BMBW - GMD - 63, 1972.

/ZUSE75/ Zuse, Konrad: Gesichtspunkte zur Beurteilung algorithmischer Sprachen. Gesellschaft für Mathematik und Datenverarbeitung. Nr. 105, BMBW - GMD - 105, 1975.

/ZUSE77/ Zuse, Konrad: Beschreibung des Plankalküls. Gesellschaft für Mathematik und Datenverarbeitung. Nr. 112, GMD, 1977.

/ZUSE86/ Zuse, Konrad: Der Computer - Mein Lebenswerk. 2. Auflage, Springer-Verlag, 1986.

/ZUSE86a/ Zuse, Konrad: Zusammenstellung von Unterlagen bezüglich der Programmspeicherung und damit zusammenhängender Gedanken. Band I, II, III, 6.1.1986.

Bemerkung von Horst Zuse: Hier handelt es sich um eine Zusammenstellung (siehe auch /ZUSE98/ aus Patentanmeldungen und Schriften von 1937-1948 bezogen auf die Ideen der Programmspeicherung (Stored-Program Computer).

/ZUSE93/ Zuse, Konrad: The Computer - My Life. Springer-Verlag, 1993.

/ZUSE97/ Zuse, Horst: A Framework of Software Measurement. DeGruyter Publisher,1997.

/ZUSE98/ Zuse, Horst: Konrad Zuse Multimedia Show (CD). Eigenvertrieb: Dr.Horst Zuse, Schaperstraße 21, 19719 Berlin, 1998. Siehe auch im Internet: http://home.t-online.de/home/horst.zuse.

[#] [$] [K] Professor Dr. Bauer

Er war Assistent und Privatdozent an der Technischen Hochschule München, wurde 1958 als Professor für Angewandte Mathematik an die Universität Mainz berufen. Seit 1963 ist er ordentlicher Professor für Mathematik an der Technischen Hochschule München. Professor Dr.Bauer erhielt 1988 den *IEEE Computer Society Pioneer Award*. Er ist u.a. eine weltweit anerkannte Persönlichkeit auf dem Gebiet der Informatik (*Computer Science).*

[#] **Bauer**
[$] **Bauer**
[K] **Bauer**

[#] [$] [K] **Adresse zur Korrespondenz von Horst Zuse**

Privat:
Dr.-Ing. Horst Zuse
Schaperstraße 21
10719 Berlin

Tel..: +49-30-881 59 88
Fax: +49-30-886 81 678
E-mail: Horst.Zuse@T-Online.de
WWW-Adresse: http://home.t-online.de/home/horst.zuse

Technische Universität Berlin:
FR 5-3
Franklinstraße 28/29
10587 Berlin

Tel. +49-30-314-24788 / 73240
Fax: +49-30-314-73489
WWW-Adresse: http://www.cs.tu-berlin.de/~zuse
E-mail: zuse@cs.tu-berlin.de

Bei der t-online WWW-Adresse bitte auf die Groß- und Kleinschreibung achten (Horst.Zuse geht z.B. nicht).

[#] Adresse zur Korrespondenz von Horst Zuse
[$] Adresse zur Korrespondenz von Horst Zuse
[K] Adresse zur Korrespondenz von Horst Zuse