Vorlesung

1

Inhalt

Vorlesung

1

Computer - Algorithmus - Programm

Im ersten Teil dieser Veranstaltung sollen Sie die grundlegenden Mechanismen der ***imperativen Programmierung*** *(im Sinne der strukturierten, prozeduralen und modularen Programmierung) kennenlernen. Sie sollen dabei das Programmieren als Methode zur Problemlösung kennenlernen, das Programmieren als exakte und vom Computer interpretierbare Ausdrucksweise zur Formulierung von Bearbeitungsvorschriften , also Algorithmen, erlernen. Elementar und zentral sind hierfür die Begriffe: Computer, Algorithmus und Programm.*

Dies ist eine Einführung, ein Einstieg. Keiner der Begriffe kann abschließend oder nahezu vollständig diskutiert werden. Hierfür braucht man sehr viel mehr Zeit .

1. Was ist ein Computer?

In einem Satz nach heutigem Verständnis: „Ein Computer ist eine (digitalelektronische) Maschine zur **Speicherung** und **automatischen Verarbeitung** von Daten, respektive Informationen (z.B. für mathematische Berechnungen oder allgemeiner Zeichenersetzungen) durch Angabe einer **programmierbaren** (flexiblen, veränderbaren) **(Rechen-)vorschrift**.".

Vielleicht richtig, aber vollständig? Informatikerinnen und Informatiker sollten es schon etwas genauer wissen: Wir kommen unten darauf zurück.

Zwar behauptete Edsger Dijkstra, „*In der Informatik geht es genauso wenig um Computer wie in der Astronomie um Teleskope..*", aber, wenn wir nicht alle zu „Zauberlehrlingen" werden wollen, dann müssen wir auch unser Hauptgerät verstehen: In allen Informatik-Curricula ist die technische Informatik (auch wenn sie heute in die Systeminformatik eingegliedert wird) essentieller Bestandteil: Elektronik, Rechnertechnologien, Digitale Schaltungen, Rechnerarchitekturen (in Ihrem Studium mindestens durch die Module Hardware und das Hardwarepraktikum vertreten).

Der Begriff **Computer** steht also für Rechner oder Rechenanlage und lässt sich aus dem lateinischen *computare* (zählen, rechnen) herleiten. Im modernen Deutsch wurde es aber eher aus dem Englischen *computer* abgeleitet: Der englische Begriff *computer*, abgeleitet vom Verb *to compute* (rechnen), bezeichnete Menschen, die beginnend im späten Mittelalter für Astronomen die quälend langwierigen Berechnungen vornahmen. Zunächst wurden die Hilfskräfte, die dies „von Hand" ausführten oder einfache Rechenhilfen bedienten, als **Computer** bezeichnet, später wurde der Begriff Computer auch für diese Maschinen benutzt. Noch Ende der 40er und Anfang der 50er Jahren untersagte der damalige CEO[1] der IBM (International Business Machines) Thomas J. Watson (Senior) die Benutzung des Begriffs Computer für die IBM-Systeme. Man nannte die Maschinen dann „Calculator". Der Hauptgrund soll gewesen sein, dass er bei den Bedienern der damals üblichen Rechenmaschinen, den Computern, keine Angst bezüglich ihres Ersatzes und Arbeitsplatzverlustes hat aufkommen lassen wollen.

Zu Anfang war die Informationsverarbeitung mit Computern meist auf die Verarbeitung von Zahlen beschränkt (es waren flexible Rechenhilfen! – mit Ausnahme von Colossus, einer Maschine zum „Code-Knacken"). Mit zunehmender Leistungsfähigkeit eröffneten sich aber viele neue Einsatzbereiche, so dass Computer heute aus unserem modernen Leben nicht mehr wegzudenken sind.

1.1 Kommentare zur Geschichte des Computers

In vielen einführenden Büchern zur Informatik findet man hier mehr oder weniger geistreiche Ausführungen zur Geschichte. Oft beginnt man beim Abakus, beleuchtet kurz die Leibnitzschen Rechenmaschinen, kommt dann je nach Herkunft auf die Z1 und Z3 zu sprechen (Zuses Entwicklungen in Deutschland) oder auf die ENIAC (mit Mauchly und Eckert in den USA) oder auf die Analytical Engine (Babbage) oder gar Collossus und den Entwicklungen in England. Wir haben diesen Teil in einen Anhang „verbannt" – hier können Sie, wenn Sie Lust und Zeit haben, durchaus interessante Details nachlesen.

Um uns nicht mitten in den plumpen Streit über den „Erfinder" des modernen Computers hineinzubegeben (Wer baute den ersten Computer?), müssen wir uns kritisch mit den kennzeichnenden Attributen eines Computers

[1] Chief Executive Officer (CEO) ist im angloamerikanischen Raum die Bezeichnung für einen alleinigen Geschäftsführer, Vorstandsvorsitzenden oder Generaldirektor einer Gesellschaft.

auseinandersetzen, denn etwa ab Mitte der 30er Jahre des 20. Jahrhunderts gab es diverse Entwicklungen, die wegbereitend waren.

Ein Computer ist eine (digitalelektronische) Maschine zur **Speicherung** und **automatischen Verarbeitung** von Daten, respektive Informationen (z.B. für mathematische Berechnungen oder allgemeiner Zeichenersetzungen) durch Angabe einer **programmierbaren** (flexiblen veränderbaren) **(Rechen-)vorschrift**.

Die hieraus abzuleitenden Eigenschaften sind:

1. (digitalelektronisch)
2. Speicherung von Daten
3. automatische Verarbeitung von Daten
4. programmierbare (flexibel veränderbare) Rechenvorschrift, **d.h. Software**.

Zu 1.: Mit dieser Eigenschaft rein mechanische, elektromechanische (auf Relais basierende) oder analog arbeitende Lösungen auszuschließen ist nach heutigem Stand zwar kennzeichnend, aber dieses historisch als ausschließende Eigenschaft zu betrachten ist schon sehr willkürlich und steht deshalb in Klammern!

Zu 2.: Tatsächlich war die zuverlässige Speicherung von Daten zumindest in größerem Umfang (z.B. 1000 zehnstellige Zahlen) anfangs ein erhebliches Problem, deren unvollkommene (weil unzuverlässige) Lösung einige frühe Ansätze (Z1, ABC) zum Scheitern brachten. Hier gab es also erhebliche technologische Probleme zu lösen.

Zu 3.: Was heißt „automatisch"? - Automatisch bedeutet „selbsttätig ablaufend". Der Wortstamm ist aus zwei altgriechischen Stämmen zusammengesetzt und bedeutet etwa *von selbst tun*, *sich selbst bewegend.*

Was soll der Computer selbst tun? Nach der Eingabe der Aufgabe (insbesondere der Eingabeparameter) soll das Ergebnis also ohne weitere menschliche Intervention ggf. schrittweise errechnet werden. Was braucht man dazu minimal? Eine Antwort hierauf liefert die Theorie:

Alles was intuitiv berechenbar ist, kann mit einer Maschine schrittweise errechnet werden, die folgende Operationen ausführen und sich die Zwischenergebnis merken (speichern) kann:

- das Nullsetzen eines Wertes,
- das Inkrement bilden (um 1 erhöhen – hochzählen) eines Wertes und
- eine datenabhängige Verzweigung:, z.B. der Form: Wenn ein (Zwischen-)Wert 0 ist, dann führe die Operationsfolge 1 aus, sonst die Operationsfolge 2.

Eine solche Maschine nennt man Turing-vollständig (*Turing-complete*). Dies ist (erschreckend) wenig aber auch kaum praktisch (die Anzahl der Schritte, selbst für einfache Aufgaben, wäre viel zu groß, die nötigen Operationsfolgen schwer zu finden und setzt auch einen (theoretisch unendlich großen Speicher voraus, etc.) – aber dies ist **minimal** um alle (überhaupt) vom Menschen berechenbare Funktionen zu errechnen (Church-Turing-These).

Zu 4.: Die Programmierbarkeit unterscheidet einen Computer von Rechenhilfsmitteln wie dem Abakus oder einer Rechenmaschine, die nur einige spezielle Rechenoperationen (entweder einfache, wie Addition und Subtraktion oder auch aufwendigere, wie die Division komplexer Zahlen) ausführen kann und ggf. darauf angewiesen ist, dass der Benutzer die benötigten Rechenschritte für eine kompliziertere Berechnung nacheinander veranlasst. Wie die Programmiermöglichkeit realisiert wurde ist prinzipiell unerheblich. Hier wurden durchaus verschiedene Varianten implementiert:

- durch Umlöten von elektrischen Verbindungen (z.B. noch 1957: Versandsystem für Quelle von Karl Steinbuch „Preisänderung durch Lötkolben-Eingriff" - 11 Jahre in Betrieb)
- durch Stecken von Schaltdrähten,
- durch Lochstreifen oder Lochkarten,

- durch Eingabe und Speicherung eines Programms (speicherprogrammierbare Recchenanlage, *stored program computer*).

Auf der Basis dieser knappen Diskussion und beim Betrachten der im Anhang angegebenen Tabelle wird klar, dass es nicht möglich ist, einen einzigen Erfinder des Computers auszumachen. Viele haben mit ihren Ideen und Entwicklungen dazu beigetragen. Das ist nicht selbstverständlich: Als klar wurde, dass Computer auch wirtschaftlich bedeutend wurden, war es „unvermeidbar" das Patentstreite entstanden, insbesondere in den USA von Eckert und Mauchly (vs. Atanasoff) betrieben und in Deutschland von Zuse. Auch wenn die Gerichte lange benötigten (bis fast 1970), so wurde **kein** Patent für die Erfindung des Computers vergeben!

Je nachdem, welche Eigenschaften und Ideen man wie gewichtet:

- das Konzept oder die Realisierung
- die Technologie: Mechanik – Relais – Röhren – (Transistoren) – (oder gar integrierte Schaltkreise)
- die Programmierbarkeit und das Programmmedium
- die theoretische Turing-Vollständigkeit – oder (praktische) Verfügbarkeit eines bedingten Sprungs
- oder weitere Kriterien, die von Zeit zu Zeit angeführt werden, wie Dezimal vs. Binär Repräsentationen, Festkomma vs. Gleitkomma Repräsentation, sequentielle oder parallele Arithmetik, usw.

kommt man zu sehr **verschiedenen Ergebnissen**, was als **erster Computer** bezeichnet werden darf! Wer Lust hat, darf also weiter streiten!

1.2 Die von Neumann-Architektur

Seit 1948 basierten eigentlich alle Rechner auf den Ideen und Konzepten, die von Neumann zusammengefasst hatte.[2] Das Original-Papier selbst enthält keine Architekturzeichnung. Auch wenn die Prinzipien meist einheitlich beschrieben werden, so findet man in der Literatur doch sehr verschiedene Varianten. Nachfolgendes Bild, illustriert die Kernideen und ist direkt mit „Textauszügen" aus dem Originalpapier beschriftet.

[2] Natürlich gab es Weiterentwicklungen und auch die sogenannte Harvard Architektur (Aiken) mit der Trennung von Daten und Programm, die den so genannten von Neumann Flaschenhals vermeidet, aber das stored program Prinip wurde allgegenwärtig.

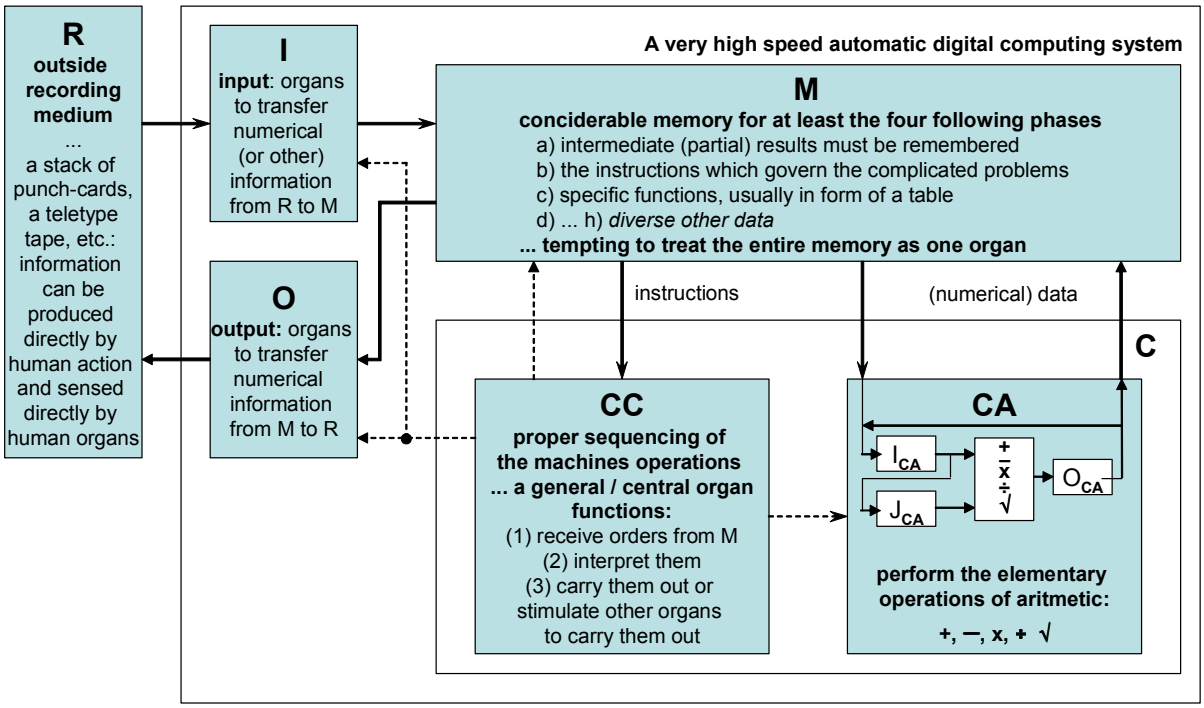

Abb. 1: Die Strukturelemente der von Neumann-Architektur

Die **Kernideen der von Neumann-Architektur sind**:

1. Ein Computer besteht aus **5 Funktionseinheiten**: Speicher **M**, Steuerwerk oder Leitwerk **CC**, Rechenwerk, Eingabe, Ausgabe (Steuerwerk und Rechenwerk bilden den **Prozessor**).

2. Im **Speicher** sind sowohl die zu bearbeitenden **Daten** als auch das **Programm** abgelegt.

3. Ein **Befehls-Ausführungszyklus** besteht aus der Folge:

1. Befehl aus dem Speicher holen	FETCH INSTRUCTION
2. Befehl im Steuerwerk interpretieren	DECODE
3. Operanden holen	FETCH OPERANDS
4. Befehl ausführen (von einer der Funktionseinheiten)	EXECUTE
5. (Erhöhen des Befehlszählers	UPDATE INSTRUCTION POINTER)

Bei Gelegenheit lesen Sie den Anhang zur Geschichte des Computers – auch dieser ist sehr knapp gefasst und Informatiker sollten diese Zusammenhänge kennen. Für das weitere Verständnis und Erreichung unseres Lernziels: „*Programmieren als Methode zur Problemlösung beherrschen lernen*“ ist dieses allerdings nicht essentiell.

2. Was ist ein Algorithmus?

2.1 Der Begriff ‚Algorithmus'

Unter einem Algorithmus versteht man eine **genau definierte** Handlungsvorschrift zur Lösung eines Problems oder einer bestimmten Klasse von Problemen.

Im täglichen Leben lassen sich leicht Beispiele für Algorithmen finden: Zum Beispiel ist ein Koch- oder Backrezept ein Algorithmus – zumindest dann, wenn alle Angaben genau genug sind und es ggf. für alle Teilaufgaben, wie Braten, Rühren, etc., ebenfalls (Teil-) Algorithmen gibt. Auch Reparatur- und Bedienungsanleitungen oder Hilfen zum Ausfüllen von Formularen sind prinzipiell Algorithmen.

Das Wort *Algorithmus* ist eine Abwandlung oder Verballhornung des Namens von **Muhammad ibn Musa al-Chwarizmi** (* ca. 783, † ca. 850), dem Autor des Buchs Hisab al-dschabr wa-l-muqabala (825, *Regeln zur Wiederherstellung und Reduktion*), durch das die Algebra im Westen verbreitet wurde. Die lateinische Fassung beginnt mit „Dixit Algoritmi..." (Algoritmus sprach...), womit der Autor gemeint war.

2.2 Formulierung von Algorithmen

Für Algorithmen gibt es unterschiedliche Repräsentationen, umgangssprachliche, graphische, als Pseudo-Programmiersprache oder in einer Programmiersprache, dann als „Maschinenprogramm", Quellprogramm, usw. Der erste für einen Computer gedachte Algorithmus wurde schon 1842 von Ada Lovelace, in ihren Notizen zu Charles Babbages Analytical Engine, geschrieben. Sie gilt deshalb als die erste Programmiererin.

Die mangelnde mathematische Genauigkeit des Begriffs Algorithmus störte viele Mathematiker und Logiker des 19. und 20. Jahrhunderts. Insbesondere steht die natürliche Sprache mit ihren Unschärfen und Widersprüchlichkeiten der Forderung nach Eindeutigkeit und Widerspruchsfreiheit im Wege: Betrachten Sie zum Beispiel ein Kochrezept oder eine Bedienungsanleitung. Wir benötigen um Computern Anweisungen zu geben eine erheblich höhere Präzision.

Insbesondere in der ersten Hälfte des 20. Jahrhunderts wurde eine ganze Reihe von Ansätzen entwickelt, um zu einer genauen Definition zu kommen. Formalisierungen des Berechenbarkeitsbegriffs sind die Turing-Maschine (Alan Turing), das Lambda-Kalkül (Alonzo Church), rekursive Funktionen, Chomsky-Grammatiken und Markow-Algorithmen – viele dieser Formalien werden Sie in Ihrem Studium noch kennenlernen.

Es wurde – unter maßgeblicher Beteiligung von Alan Turing selbst – gezeigt, dass all diese Methoden ebenso leistungsfähig (gleich *mächtig*) sind. Sie können durch eine Turing-Maschine emuliert (nachgebildet) werden, und sie können umgekehrt eine Turing-Maschine emulieren.

Mit Hilfe des Begriffs der Turing-Maschine kann folgende formale Definition des Begriffs formuliert werden:

Eine Berechnungsvorschrift zur Lösung eines Problems heißt Algorithmus genau dann, wenn eine zu dieser Berechnungsvorschrift äquivalente Turingmaschine existiert, die für jede Eingabe, die eine Lösung besitzt, stoppt.

Etwas informaler kann man Algorithmus auch wie folgt definieren:

Unter einem Algorithmus versteht man allgemein eine genau definierte Handlungsvorschrift zur Lösung eines Problems oder einer bestimmten Klasse von Problemen.

Dabei muß „natürlich" jeder Schritt der Handlungsvorschrift **ausführbar** sein. Im Allgemeinen fordert man insbesondere folgende weitere **Eigenschaften von einem Algorithmus**:

2.3 Eigenschaften von Algorithmen

- **Determiniertheit:** Bei jeder Ausführung mit gleichen Startwerten muss das gleiche Ergebnis berechnet werden. Algorithmen sind determiniert, wenn sie bei gleichen Parametern und Startwert stets das gleiche Resultat liefern.

- **Deterministisch:** Deterministisch heißen alle Algorithmen, bei denen zu jedem Zeitpunkt der Ausführung maximal eine Möglichkeit der Programmfortsetzung besteht. Gibt es mehrere Möglichkeiten der Programmfortsetzung und lassen sich diesen Wahrscheinlichkeiten zuweisen, so spricht man von stochastischen, randomisierten oder probabilistischen Algorithmen. In der Theorie werden auch nichtdeterministische Algorithmen betrachtet, die der aber in der Praxis kaum Verwendung finden.

  Anmerkung: Es gilt übrigens: Jeder deterministische Algorithmus ist auch determiniert. Nicht jeder determinierte Algorithmus ist jedoch deterministisch.

- **Statische Finitheit:** Die Beschreibung des Algorithmus ist endlich. Die Beschreibung eines Algorithmus darf nicht unendlich groß sein. Als statische Finitheit wird die Endlichkeit des Quelltextes bezeichnet. Der Quelltext darf nur eine begrenzte Anzahl, wenn auch bei Bedarf sehr viele Regeln enthalten.

- **Dynamische Finitheit:** Die Menge an Daten inclusive Zwischenspeicherungen sind zu jeder Zeit endlich. Zu jedem Zeitpunkt der Ausführung darf der von einem Algorithmus benötigte Speicherbedarf nicht unendlich groß sein. Andernfalls wäre der Algorithmus nicht ausführbar.

- **Terminiertheit:** Der Algorithmus bricht nach endlicher Zeit kontrolliert ab. Algorithmen sind terminierend, wenn sie für jede mögliche Eingabe nach einer endlichen Zahl von Schritten zu einem Ergebnis kommen. Die tatsächliche Zahl der Schritte kann dabei jedoch sehr groß sein. Es ist übrigens im Allgemeinen nicht möglich für jeden beliebigen Algorithmus zu bestimmen, ob er terminiert oder nicht, das klassische Halteproblem.

  Anmerkung: Die meisten Steuerungssysteme, Betriebssysteme und auch viele Programme, die auf Interaktion mit dem Benutzer aufbauen, erfüllen diese Eigenschaft nicht: Wenn der Benutzer keinen Befehl zum Beenden gibt, läuft das Programm endlos weiter.

Ein klassisches Beispiel: der **Euklidische Algorithmus**

Der Euklidische Algorithmus dient zur Ermittlung des größten gemeinsamen Teilers (ggT) zweier natürlicher Zahlen *A* und *B*. Der klassische Algorithmus wurde bereits um 300 v. Chr. von Euklid beschrieben und benutzt die Subtraktion:

Formulierung 1: Nach Euklid zieht man wiederholt die kleinere der beiden Zahlen von der größeren ab, solange bis die beiden Zahlen gleich groß sind. Diese Zahl ist dann der ggT (A,B).

Frage: Ist dieser Algorithmus präzis genug beschrieben? – Können Sie ihn ausführen? Nehmen Sie das Beispiel: ggT (12, 30).

Der Algorithmus funktioniert dann in folgender Weise:

Schritt i	A_i	B_i	$A_i - B_i$
0	30	12	18
1	18	12	6
2	12	6	6
3	6	6	

Das Ergebnis ist also ggT (30,12) = 6. Frage: Ist die Beschreibung wirklich eindeutig. Versuchen Sie, den Algorithmus präziser zu beschreiben als oben. Eine Möglichkeit ist, dieses in Pseudo-Code zu tun:

Formulierung 2:

```
EUCLID_KLASSISCH (a,b)                  # Name des Algorithmus und
                                        # der Parameter
1  solange b ≠ 0                        # Eine bedingte Wiederholung der
                                        # nachfolgend eingerückten
                                        # Anweisungsfolge
2      wenn a > b                       # Eine Verzweigung
3          dann a ← a - b               # Eine Wertzuweisung
4          sonst b ← b - a              # Die alternative Wertzuweisung
5  ende_und_ergebnis a                  # Die Beendigung des Algorithmus
                                        # die Rückgabe des Ergebnisses
```

Diese Beschreibung ist schon sehr ähnlich der einer (imperativen, siehe unten) Programmiersprache aber ohne viele Formalien. Als sprachliche Mittel ist alles zugelassen, was verständlich und (informell) eindeutig ist, also zum Beispiel die Nummerierung von Schritten, eine Anweisung zur Wiederholung von Schritten, eine Verzweigung gemäß einer Bedingung (wenn ... dann ... sonst), eine Konvention für Kommentare, hier #, usw. Allgemein gültige Konventionen gibt es für Pseudo-Code nicht: Entscheidend ist die Verständlichkeit für einen menschlichen Leser, so dass man sich in der Regel auf wenige „Sprachkonstrukte" beschränkt. Für eine Maschine ist weder die Syntax noch die Semantik der Sprachmittel präzis genug – unter Menschen aber oft ausreichend.

Anmerkung: Der Euklidische Algorithmus wird heute i.d.R. anders formuliert: Man ersetzt die im klassischen Algorithmus auftretenden wiederholten Subtraktionen durch die Division mit Rest. Wir beginnen mit den beiden Zahlen *a* und $b = r_0$ deren größter gemeinsamer Teiler bestimmt werden soll:

$a = q_1 \cdot r_0 + r_1$ $\quad$ $q_1 = a : r_0$ (Ganzzahlige Division) und $r_1 = a \bmod r_0$ (Modulo-Funktion: Rest der Divison)

In jedem weiteren Schritt wird mit dem Divisor und dem Rest des vorhergehenden Schritts eine erneute Division mit Rest durchgeführt. Und zwar solange bis eine Division aufgeht, das heißt der Rest Null ist.

$$r_0 = q_2 \cdot r_1 + r_2$$
$$r_1 = q_3 \cdot r_2 + r_3$$
$$\vdots$$
$$r_{n-1} = q_{n+1} \cdot r_n + 0$$

Der Divisor r_n der letzten Division ist dann der größte gemeinsame Teiler: $ggT(a,b) = r_n$

Da sich die Zahlen in jedem Schritt mindestens halbieren, ist das Verfahren auch bei großen Zahlen schnell. In Pseudo-Code lässt sich der Algorithmus wie folgt formulieren:

```
EUCLID_MODERN(a,b)
1  solange b ≠ 0
2       h ← a mod b    # a mod b liefert den ganzzahligen Rest von a:b
3       a ← b
4       b ← h
5  return a
```

3 Was ist ein Programm?

Programmieren bezeichnet die Tätigkeit mithilfe einer Programmiersprache ein Programm zu formulieren / zu schreiben. Programme sind mittels einer Programmiersprache formulierte Berechnungsvorschriften (Algorithmen; wir werden diesen Begriff später noch präziser fassen). Eine **Programmiersprache** ist eine künstlich geschaffene Sprache (eine formale Sprache) zur maschinenlesbaren Repräsentation von Algorithmen. **Programmiersprachen müssen Algorithmen (= Bearbeitungsvorschriften) sowohl in einer für einen Computer, als auch in einer für den Menschen lesbaren und verständlichen Form ausdrücken.** Sie sind notwendig, da natürliche Sprachen (deutsch, Englisch, Französisch, etc.) für eine genügend detaillierte und präzise Beschreibung von Computerberechnungen zu vieldeutig sind.
Die durch eine Programmiersprache ausgedrückte, von einem Menschen möglichst einfach lesbare (und von einem Computer interpretierbare) Beschreibung nennen wir **Quelltext (Quellcode)** eines Programms. Programme nennen wir Software und diese soll insbesondere

1. fehlerfrei und korrekt sein und genau die „Berechnungsleistung" erbringen, für die sie geschrieben wurden,
2. möglichst einfach wartbar (verbesserbar, veränderbar, anpassbar) sein. Dabei muss berücksichtigt werden, dass in der Praxis ca. 80% der Veränderungen von einem anderen Menschen durchgeführt werden als dem ursprünglichen Programmierer.

Neben der Programmiersprache (und den zugehörigen Programmen zur Übersetzung: Compiler oder Interpreter, siehe unten) helfen hierbei die sogenannte **Laufzeitumgebung** (zur plattformunabhängigen Nutzung von Hardware und Betriebssystem: Lesen und Schreiben von Dateien, Daten über Netzwerke transportieren; Steuerung von Ein- und Ausgabegeräten; Behandlung von Ausnahmen / Fehlern, etc.) und einer **Programmentwicklungsumgebung** (*Integrated Development Environment* (IDE)).

In (Programm-) Quelltexten sollten Algorithmen also möglichst einfach und übersichtlich formuliert werden können. Die Anforderungen sind:

- Programmieren soll einerseits möglichst einfach und übersichtlich sein, andererseits aber redundanzfrei und eindeutig.
- Übliche Konventionen (z.B. aus der Mathematik, aus Anwendungsgebieten) wollen wir übernehmen, andererseits aber eindeutig, ohne Interpretationsspielraum, formulieren.
- Einerseits sollen Programmiersprachen universell sein, andererseits ggf. spezielle Anwendungsbereiche unterstützen.
- Einerseits sollen Programme ein effizientes Laufzeitverhalten haben (schnell sein) -- andererseits möglichst abstrakt (d.h. unabhängig von Prozessor, Betriebssystem) sein.
- Programmiersprachen sollen verschiedene Programmierparadigmen unterstützen,
- usw.

Diese widersprüchlichen Anforderungen machen es so schwierig, Programmiersprachen zu entwickeln. Der Sprachdesigner muss sich immer wieder entscheiden, welcher Anforderung er mehr Gewicht gibt und dies

führte historisch zu den existierenden mehr als 8500 Programmiersprachen. Es gibt auch keine optimale Sprache für alle Zwecke.

Zusätzlich sind Meinungen durchaus verschieden, welche Eigenschaften eine Programmiersprache besitzen sollte. Allgemein wird jedoch akzeptiert, dass zumindest die grundlegende mathematische **Arithmetik und einfache Zeichenersetzung** einfach ausgedrückt werden können sollte. Oft erscheinen der von der Programmiersprache vorgegebene oder der besonders unterstützte Programmierstil oder die Zweckgebundenheit der Programmiersprache wichtig.

3.1 Generationen von Programmiersprachen

Eine klassische Einteilung der Programmiersprachen erfolgt nach sogenannten Generationen.

1. Generation: Tatsächlich war es bis in die 50er Jahre hinein notwendig, in der jeweiligen **Maschinensprache** zu programmieren. Das war nicht nur sehr mühsam, sondern auch sehr fehleranfällig. In dieser Zeit entstand der Beruf des Programmierers – also derjenige der programmiert oder codiert.

Jeder Prozessor bietet eine dem jeweiligen Prozessor-Typ eigene „Programmiersprache" an, die so genannte **Maschinensprache.** Diese Sprache kann der Prozessor direkt ausführen. Eine Maschinensprache ist nichts anderes als eine Bitfolge mit einer speziellen Interpretationsvorschrift und einfachen Semantik. Diese Sprache reflektiert direkt die Prozessorarchitektur. Ein typischer von Neumann Maschinenbefehl sieht etwa folgendermaßen aus:

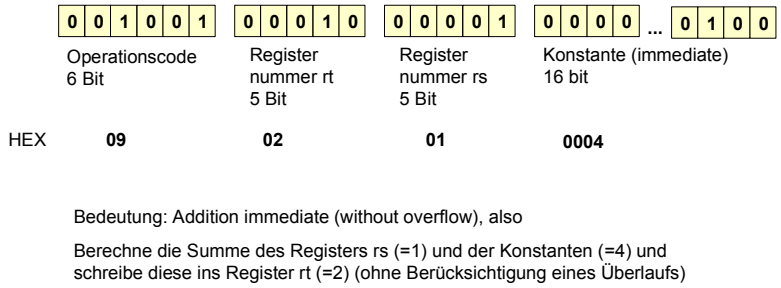

Abb. 2: Beispiel eines Maschinenbefehls aus dem MIPS-Prozessor (RISC-Prozessor)

Maschinensprachen sind also für den Menschen kaum lesbarer Binärcode. Etwas Hilfe bieten ggf. spezielle Programme, so genannte Maschinensprachemonitore (abgekürzt auch einfach Monitor genannt) mit deren Hilfe Binärprogramme bearbeitet werden können – das ist aber glücklicherweise in ganz, ganz seltenen Fällen nötig.

```
00100111101111011111111111100000        27BDFFE0
10101111101111110000000000010100        AF ...
10101111101001000000000000100000
10101111101001010000000000100100
...
```

Binärdarstellung Hexadezimaldarstellung

Abb. 3 Ein MIPS-Maschinenprogramm binär und hexadezimal:

2. Generation: Assembler ersetzen die Zahlencodes der Maschinensprache durch symbolische Bezeichner (Mnemonics). Eine Assembleranweisung lässt sich Eins-zu-Eins in einen Maschinenbefehl umsetzen, ist also immer

noch direkt abhängig von der Maschine. Ein Assembler übersetzt ein Assemblerprogramm in Maschinencode. Für unser obiges Beispiel des MIPS-Prozessors würde der Beispiel-Befehl dann folgendermaßen aussehen:

allgemein:	`addiu rt,rs,imm`
konkret das Beispiel:	`addiu 2,1,4`

Assemblersprachen bieten typischerweise zusätzlich die Möglichkeit Speicherzellen mit symbolischen Namen zu versehen, also anstelle einer Adresse (eine Zahl zwischen 0 und höchster verfügbare Speicherzelle) einen Namen, wie etwa „Startwert" oder „Ergebnis" zu verwenden. Verglichen mit der direkten Maschinenprogrammierung erleichtert ein Assembler die Programmierung, aber komfortabel kann man das nicht nennen. Der Anteil der Assemblerprogrammierung ist dementsprechend heute sehr gering – wird z.B. aber noch genutzt, wenn man für bestimmte Programmteile eine maximale Geschwindigkeit benötigt, z.B. in embedded systems.

3. Generation: Höhere Programmiersprachen (High Level Languages) führen Konzepte wie z.B. Variablen ein, um leichter verständlichen Quelltext schreiben zu können. Ziel der Sprachen der 3. Generation war es, weitgehend maschinenunabhängig und leichter verständlich zu programmieren. *„She did this – the invention of the compiler -, she said, because she was lazy and hoped that „the programmer may return beeing a mathematician"*, so ein Zitat über Grace Murray Hopper (1906 – 1992) „Erfinderin des Compilers" im Mark 1 Team (Aiken).

Die heute ganz überwiegend eingesetzten Programmiersprachen sind Programmiersprachen der Dritten Generation. Objektorientierte Sprachen sind ebenfalls Sprachen der 3. Generation. Um jedoch ihre konzeptionelle Sonderrolle zu betonen, werden sie in der Literatur oft als *„OO-Generation"* bezeichnet.

Beispiel: Es soll die Addition „3 + 4" durchgeführt werden. In allen Programmiersprachen dieser Generation werden z.B. mathematische Notationen für die Operation „addiere" direkt unterstützt. Man kann also schreiben:

```
int Summe;
Summe = 3 + 4;
```

In diesem Beispiel wird eine Variable „Summe" vom Datentyp int (Integer) reserviert und dann in dem zweiten Statement dieser Variable der Wert 7 (3+4) zugewiesen.

Die ersten derartigen Sprachen wurden in den 50er Jahren in den USA entwickelt und verbreiteten sich schnell:

- FORTRAN (**FOR**mula **TRAN**slator) von John Backus et al., 1954: für technisch-wissenschaftliche Anwendungen.
- COBOL (**CO**mmon **B**usiness **O**riented **L**anguage) von Grace Hopper et al., 1959 für kaufmännische Anwendungen.
- LISP (**LIS**t **P**rocessor) von John McCarthy et al., 1959, vor allem im akademischen Umfeld genutzt.

Die vorgenannten Sprachen existieren mit ihren Nachfolgern bis heute. Vor allem LISP beeinflusste die später entwickelten Programmiersprachen stark.

Sprachen der **4. Generation: Fourth Generation Language (4GL)** sind anwendungsbezogene (applikative) Sprachen. Sie stellen alle wichtigsten Gestaltungsmittel von Sprachen der 3. Generation zur Verfügung, zusätzlich jedoch Sprachmittel zur Auslösung relativ komplexer, anwendungsbezogener Operationen, beispielsweise zum Zugriff auf Datenbanken (meist per SQL) oder zur Gestaltung von graphischen Benutzungsoberflächen (GUIs).

Der Begriff 4GL wurde im Marketing einiger Hersteller von Entwicklungsumgebungen fälschlich missbraucht, um das Produkt als vermeintlich moderner darzustellen. Deshalb ist die Benutzung dieses Begriffes problematisch!

5. Generation: (Very High Level Language, VHLL) Sprachen der 5.Generation gestatten das Beschreiben von Sachverhalten und Problemen. Sie kommen vor allem im Bereich der KI (künstlichen Intelligenz) zum Einsatz. Die Wahl des Problemlösungsweges kann dem jeweiligen System (weitgehend) überlassen werden. Bekanntestes Beispiel für eine Sprache der 5. Generation ist **PROLOG**. Auch hier ist die Benutzung dieses Begriffes problematisch!

3.2 Programmiersprachenparadigmen

Programmiersprachen lassen sich bezüglich des zugrundeliegenden **Programmierparadigmas** einteilen. Paradigma bedeutet allgemein *Beispiel*, *Vorbild*, *Muster* oder *Abgrenzung*.

- **Imperative Programmierparadigmen**
  - Strukturierte Programmierung
  - **Prozedurale Programmierung**
  - **Modulare Programmierung**
  - Programmierung mit abstrakten Datentypen
  - **Objektorientierung**

- **Deklarative Programmierparadigmen**
  - Constraint Programmierung
  - **Funktionale Programmierung**
  - Logische Programmierung

- **Sonstige Ansätze**
  - Agentenorientierte Programmierung
  - Aspektorientierte Programmierung
  - Generische Programmierung
  - Datenstromorientierte Programmierung

Grundlegend für die Eigenschaften von Programmiersprachen sind die Paradigmen der **imperativen** und der **deklarativen** Programmierung. Beim letzteren sind als wichtige Ausprägungen die Paradigmen der **funktionalen Programmierung** und der logischen Programmierung zu nennen. Alle weiteren Programmierparadigmen sind Verfeinerungen und Spezialisierungen dieser Prinzipien.

Wir beschränken uns hier auf die Diskussion der Kernbegriffe. Bei allen **imperativen Programmiersprachen** versteht man ein Computerprogramm als lineare Folge von Befehlen, die der Rechner in einer definierten Reihenfolge abarbeitet. Imperative Programmiersprachen bilden die Architektur des von-Neumann-Rechners auf die Programmierung ab. Dieses Konzept wird durch **Befehle** (Anweisungen) realisiert. Die Befehle manipulieren dabei den Zustand der Speicherbereiche oder Speichermedien, die die zu verarbeitenden und die auszugebenden Daten enthalten. Daten werden in sogenannten „Variablen" gespeichert. Die Werte in Variablen können sich im Programmablauf ändern. Daher kann man sie auch als **zustandsorientierte Programmierung** bezeichnen. Durch die Reihenfolge der Befehle ist die zeitliche Abfolge vorgegeben.

Die Idee der **deklarativen Programmierung** ist der historisch jüngere Ansatz. Im Gegensatz zu imperativen Programmierparadigmen, bei denen das „Wie berechnen" im Vordergrund steht, fragt man in der deklarativen Programmierung nach dem „Was berechnen". Es wird also nicht mehr der Lösungsweg programmiert, sondern „nur noch" angegeben, welches Ergebnis gewünscht ist. Zu diesem Zweck beruhen deklarative Paradigmen auf mathematischen, rechnerunabhängigen Theorien.

Angestrebt wird, dass keine so genannten Nebeneffekte existieren. Insbesondere sind Beweise (z.B. Korrektheitsbeweis, Beweise über Programmeigenschaften) dank der strikten mathematischen Basis (u.a. Lambda-

Kalkül) durchführbar. Allerdings haben diese Sprachen in der Praxis eine vergleichsweise geringe Akzeptanz (man spricht gelegentlich von Akademikersprachen).

In der **funktionalen Programmierung** wird die Aufgabenstellung und die bekannten Prämissen als funktionaler Ausdruck formuliert. Das selbstständige Anwenden von Funktionsersetzung und Auswertung seitens des Interpreters oder Compilers lösen dann die Aufgabenstellung. Das Programm kann als Abbildung (Funktion) der Eingabe auf die Ausgabe aufgefasst werden. Ein Beispiel (Haskell) hierzu lernen Sie in PRG 2 kennen.

In der logischen Programmierung werden die Aufgabenstellung und ihre Prämissen als logische Aussagen (Regeln) formuliert. Der Interpreter versucht dann, die gewünschte Lösungsaussage mit Hilfe des Horn-Verfahrens herzuleiten. (Siehe auch Horn-Klauseln). Dabei werden die Regeln gegen eine Datenmenge auf ihre Instanziierbarkeit geprüft. Aus allen Regelinstanziierungen wird eine (mehrere, alle) ausgewählt und die zur Regel gehörenden Anweisungen werden ausgeführt. Ein Beispiel für diese Art von Programmiersprachen ist PROLOG.

Viele der in der Praxis benutzten Programmiersprachen unterstützen mehrere Paradigmen in verschiedenen Ausprägungen:

Programmiersprache	**imperativ**	objektorientiert	**deklarativ**	funktional	logisch
C	x				
C++	x	x			
JAVA	x	x			
Python	x	x	(x)	(x)	
Haskell			x	x	
Prolog			x		x
Ada	x	x			

Tabelle 1: Eigenschaften und Einteilung aktueller Programmiersprachen

3.3 Compiler versus Interpreter

Ein **Compiler** (auch **Kompilierer** oder **Übersetzer**) ist ein Computerprogramm, das ein in einer Quellsprache geschriebenes Programm - genannt Quellprogramm (Quellcode) - in ein semantisch äquivalentes Programm einer Zielsprache (Zielprogramm) umwandelt. Üblicherweise handelt es sich dabei um die Übersetzung eines in einer Programmiersprache mindestens der dritten Generation geschriebenen **Quellprogramms in Assemblersprache, Bytecode oder direkt in Maschinensprache**. Das Übersetzen eines Quellprogramms in ein Zielprogramm durch einen Compiler wird als Kompilierung oder auch als Übersetzung bezeichnet.

Der Übersetzungsvorgang (das Compilieren) erfolgt in verschiedene Phasen, die nacheinander (sequentiell) ausgeführt werden.

1. Lexikalische Analyse: Die lexikalische Analyse zerteilt den eingelesenen Quelltext in zusammengehörende „Token" verschiedener Klassen, z. B. Schlüsselwörter, Bezeichner, Kommentare, Zahlen und Operatoren. Die genaue Bedeutung dieser Klassen klären wir später.

2. Syntaktische Analyse: Die syntaktische Analyse überprüft, ob der eingelesene Quellcode tatsächlich ein Programm der zu übersetzenden Quellsprache ist, d. h. der Syntax (Grammatik) der Quellsprache entspricht.

3. Semantische Analyse: Die semantische Analyse überprüft die „statische" Semantik, also über die syntaktische Analyse hinausgehende Bedingungen an das Programm. Zum Beispiel muss eine Variable in vielen Programmiersprachen erst deklariert worden sein, bevor sie verwendet wird, und Zuweisungen müssen mit kompatiblen (verträglichen) Datentypen erfolgen.

4. Zwischencodeerzeugung: Viele moderne Compiler erzeugen zunächst einen Zwischencode, der schon relativ maschinennah sein kann und führen auf diesem Zwischencode Programmoptimierung durch.

5. Programmoptimierung: Der Zwischencode ist Basis vieler Programmoptimierungen. Ein großes Potential bieten die Programmschleifen, d.h. Wiederholungen von bestimmten Teilen des Programms, indem man z. B.

- möglichst viele Variable in Registern hält,
- Berechnungen innerhalb der Schleife, die in jedem Durchlauf dasselbe Ergebnis liefern, nur einmal vor der Schleife ausführt.
- zwei Schleifen, die über denselben Wertebereich gehen, zu einer Schleife zusammenfasst. Damit fällt der Verwaltungsaufwand für die Schleife nur einmal an, usw.

6. Codegenerierung: Bei der Codegenerierung wird der Programmcode der Zielsprache erzeugt.

Falls die Zielsprache eine Maschinensprache ist, kann das Ergebnis direkt ein ausführbares Programm sein oder eine so genannte Objektdatei, die durch das „Linken" mit der Laufzeitbibliothek und evtl. weiteren Objektdateien (z.B. Dienste des Betriebssystems) zu einem ausführbaren Programm wird.

Der typische Zyklus bei der Programmentwicklung ist also „Übersetzen – Binden – Laufenlassen". Um übersichtlich zu programmieren, wird die Programmiererin bewusst kleine (= übersichtliche) Programme schreiben, ggf. sehr viele in verschiedenen Bibliotheken. Das heißt aber auch, dass größere Programmsysteme unter Umständen erhebliche Bindezeiten haben.

Hiervon zu unterscheiden sind **Interpreter**, die einen Programm-Quellcode im Gegensatz zu Compilern **nicht** in eine auf dem Zielsystem ausführbare Datei umwandeln, sondern den

- Quellcode (Statement für Statement) einliest,
- Quellcode analysiert (lexikalisch, syntaktisch, sematisch, siehe oben bei Compiler)
- (Maschinen-)Code generiert, ggf. nötige Bibliotheksfunktionen anbindet und
- (Maschinen-)Code für ein Statement direkt ausführt und dann das nächste Statement bearbeitet.

Die Analyse des Quellcodes erfolgt also zur Laufzeit des Programmes. Diese Arbeitsweise ist auch der größte Nachteil der Interpreter, nämlich die im Vergleich zu compilierten Programmen deutlich langsamere Ausführungsgeschwindigkeit. Die während der Ausführungszeit (Laufzeit) notwendige Analyse des Quellcodes und Codegenerierung verbraucht viel Zeit. Ein vom Compiler in Maschinencode übersetztes Programm kann hingegen nach dem Binden mit maximaler Geschwindigkeit direkt vom Prozessor ausgeführt werden.

Auch werden bei einem Interpreter zur Speicherung von Daten Typinformationen (Interpretationsvorschriften) benötigt, um Typüberprüfungen etc. zur Laufzeit durchführen zu können, was der Compiler schon beim Übersetzungsvorgang erledigt hat. Dies führt zu erheblich größeren Datenbereichen.

Vorteile haben Interpreter vor allem während der Programmentwicklung:

- Geänderte Programme sind sofort ausführbar (Übersetzen und binden sind nicht explizit nötig), das spart viel Zeit beim Testen ,
- schnelles Ändern, auch Ausprobieren ist möglich ,
- sehr gut für Prototypen (und kleine Programme) geeignet.

Interpreter findet man vor allem für so genannte **Skriptsprachen** (häufig auch Scriptsprachen). Dieses sind Programmiersprachen, die vor allem für kleine, überschaubare Programmieraufgaben gedacht sind. Sie verzichten auf bestimmte Sprachelemente, deren Nutzen erst bei der Bearbeitung größerer Projekte zum Tragen kommt. Etwa wird in Skriptsprachen oft auf den Deklarationszwang von Variablen verzichtet (dynamisches Typing) – vorteilhaft

zur schnellen Erstellung von kleinen Programmen (z.B. zum Rapid Prototyping), bei großen Programmen hingegen von Nachteil, etwa wegen der fehlenden Überprüfungsmöglichkeit auf Tippfehlern bei Variablennamen.

Programme, die in Skriptsprachen geschrieben sind, werden auch *Skripte* genannt. Skripte werden fast ausschließlich in Form von Quelltextdateien ausgeliefert, um so ein einfaches Bearbeiten und Anpassen der Programme zu ermöglichen. Bekannte Skriptsprachen sind: **Perl** – die erste Skriptsprache, die eine weite Verbreitung bei Webservern fand, **PHP** – wurde speziell für Programmieraufgaben an Webservern konzipiert und **Python** für denselben Zweck oder als Programmiersprache zur Ergänzung von Anwendungssystemen, (z. B. in OpenOffice, Blender und Gimp). Clientseitig ist vor allem JavaScript verbreitet und wird heute von allen modernen Browsern unterstützt.

3.4 Virtuelle Maschinen

Virtuelle Maschinen (in unserem Kontext) sind „virtuelle" Laufzeitumgebung für Programme auf einer Gastgeber-(Host-)Maschine, der realen Maschine.

Sie können entweder vollständig in Software oder mittels einer Kombination aus Software und Hardware implementiert werden. Bei sogenannten eingebetteten Systemen werden z.B. auch Java-CPUs für eine effiziente Ausführung von Java-Bytecode angeboten.

Virtuelle Maschinen sind also Abstraktionen von realen Maschinen. Die große Mehrzahl realer CPUs verwalten mehrere Operanden in einer **beschränkten Zahl** direkt adressierbarer Register (Registermaschinen), die große Mehrzahl der Virtuellen Maschinen verwalten ihre Operanden hingegen in einem **unbeschränkten Stapel (Stack)**. Ansonsten ist die Ausdrucksfähigkeit sehr ähnlich zu Assemblercode.

Virtuelle Maschinen spielen heute eine bedeutende Rolle, da Microsoft mit der .NET Architektur dem Beispiel von Sun mit der virtuellen Java-Maschine (Java VM) folgte.

Programme werden also nicht direkt in die Maschinensprache der CPU übersetzt wurde, sondern in einen einfach strukturierten sogenannten Zwischencode (vergleiche oben beim Compiler). Dieser Zwischencode wird dann auf dem Zielsystem durch einen Interpreter ausgeführt. Die Speicherung des Zwischencodes kann unterschiedlich ausfallen, etwa als Bytecode oder als Baumstruktur. Technisch kann dies als Vorkompilierung betrachtet werden, bei der die Analyseschritte eines Compilers durchlaufen werden (Frontend des Compilers), jedoch keine maschinenorientierte Anpassung an eine spezielle CPU erfolgt (Backend des Compilers), sondern die Anpassung an die abstrakten Ausführungseigenschaften der virtuellen Maschine erfolgt.

Bytecode nennt man ein Programm, das aus Befehlen für eine virtuelle Maschine besteht. Die virtuelle Maschine, im Falle von Java, die Java Virtual Machine (JVM), führt dann dieses Zwischenergebnis der Übersetzung aus, indem sie den Bytecode in Maschinencode, für den jeweiligen Prozessor, zur Laufzeit übersetzt. Dabei ist zu beachten, dass die Virtual Machine für jede Rechnerplattform, auf der das Programm laufen soll, bereits vorliegen muss.

Vorteile einer portablen virtuellen Maschine:

- Plattformunabhängigkeit: Programme für eine virtuelle Maschine laufen auf allen realen Maschinen, für die die virtuelle Maschine implementiert ist.
- Eine Dynamische Optimierung ist möglich.

Nachteile einer portablen virtuellen Maschine:

- Die Ausführung eines portablen Programms auf einer portablen virtuellen Maschine ist i.d.R. langsamer als die native Ausführung von Programmen, die speziell für die Zielumgebung in Maschinencode übersetzt wurden.

- Bei Verwendung eines Interpreters ergeben sich zusätzliche Indirektionen z.B. für Variablen, was weniger effizient als eine direkte Ausführung in Registern ist.

Diese Nachteile können zum Teil durch geeignete Optimierung verringert werden.

Java ist heute nur eines der prominentesten Beispiele für eine Bytecode-basierte Programmiersprache. Andere Sprachen, die Bytecodes verwenden, sind zum Beispiel Python, C#, Perl und Prolog.

Viele interpretierte Sprachen verwenden intern auch Bytecode (z.B. Python), was bedeutet, dass der Bytecode an sich unsichtbar für den Programmierer und Endbenutzer gehalten wird und automatisch als Zwischenschritt der Interpretation des Programmes erzeugt wird. Beispiele für aktuelle Sprachen, die zu diesem Trick greifen sind Perl, PHP, Python und Prolog.

4 Zusammenfassung

Das Begriffstripel „Computer – Algorithmus – Programm“ sollte Ihnen jetzt vertraut sein. Diese Kernbegriffe werden Sie ihr Ganzes Studium hindurch begleiten.

Im Fokus des Moduls PRG1 (PRG1+EPR) steht das Programmieren. Algorithmen werden Sie in ganz vielfacher Ausgestaltung kennen- und untersuchen lernen, vor allem auch in den theoretischen Veranstaltungen „Algorithmen und Datenstrukturen“ und in der „Algorithmentheorie“. Aber auch in vielen Veranstaltungen der praktischen Informatik spielen konkrete Algorithmen eine Rolle. Mit dem Computer, seinen Architekturvarianten und seinen Realisierungsmöglichkeiten werden Sie sich in der Veranstaltung „Hardwarearchitekturen und Rechensysteme“ vertieft beschäftigen.

Alle Themen sind für Vertiefungen offen und erlauben ein vielfältiges, individualisiertes und interessantes Studium zum Bachelor Informatik. Viel Spaß und viel Erfolg.

Anhang

1 Die Entwicklung des Computers

Die Zeit bis 1950

Charles Babbages hatte Mitte des 19. Jahrhunderts eine wegweisender Entwicklung gemacht, die **Analytical Engine** (die er allerdings nicht realisieren konnte). Es brauchte fast **einhundert Jahre,** bis viele seiner Ideen neuentwickelt und umgesetzt wurden. Obwohl (die Mathematiker) Howard Aiken in Havard, Vannevar Bush. und Georg Stibitz an den Bell Laboratories seine Ideen kannten, war der direkte Einfluss vermutlich sehr beschränkt. Konrad Zuse kannte diese Ideen nach eigenem bekunden nicht!

Man muss auch feststellen, dass viele Entwicklungen im Zeitraum **1939 – 1945** ihre Finanzierung aufgrund der Kriegsereignisse und der Einstufung als „kriegswichtig" erlangten, so insbesondere Colossus in England (zur Dechiffrierung der deutschen ENIGMA-Codes), Havard Mark 1 (ASCC) zum Berechnen von Tabellenwerken von ballistischen Kurven für die Marine, ENIAC (ballistische Geschoßbahnberechnung /Tafelwerke für die Army). Auch Zuse hat dem deutschen Militär seine Entwicklungen mehrfach angeboten ohne Erfolg. Dieses militärische Einflussnahme setzte sich in der Geschichte fort: Der Koreakrieg beflügelte die US-Computerindustrie enorm und später die Entwicklungen „für" den kalten Krieg.

Bei den ersten Entwicklungen gab es sowohl Tischgroße Systeme (Z 1, Z 3, ABC) als auch „mechanische Ungetüme", so die Havard Mark 1: Gewicht 35 Tonnen sowie eine Frontlänge von 16 Metern., und auch elektronische Ungetüme: ENIAC: Die komplette Anlage war in U-Form aufgebaut, beanspruchte eine Fläche von 10m x 17m und wog 27t.

Bis 1948 hatte **keines** der realisierten Rechensysteme alle geforderten Qualitäten erfüllt, die einen Computer ausmachen, siehe nachfolgende Tabelle.

Name	Haupt-entwickler	Jahr der Fertigstellung	Rechen-Technologie	Programmierbar?	Turing-vollständig
Analytical Engine nicht realisiert	Charles Babbage	**(1833 -1842)**	Mechanik	Lochstreifen	Ja
Z 1	Conrad Zuse	1938	Mechanik	Lochstreifen	Nein
CNC 1 Complex Number Computer	George Stibitz S.Williams Bell Labs	1940	Relais	Nein	Nein
Atanasoff-Berry Computer (ABC) nicht voll funktionsfähig realisiert	John V. Atanasoff Clifford Berry Iowa State	1941	Elektronen-röhren	Nein	Nein
Z 3	Konrad Zuse	1941	Relais	Lochstreifen	Ja
Bletchley Park	M.H. Newman I.J. Good	1944	Elektronen-röhren	Datenband und Verkabelung	Nein
Harvard Mark I (ASCC)	Howard H. Aiken Havard Univ. ➔ **IBM**	1944	Mechanik	Lochstreifen	Ja
ENIAC	John Mauchley Presper Eckert U Pennsylvania	1946	Elektronen-röhren	Verkabelung	Ja

Z 4	Conrad Zuse ➔ **Zuse AG**	1944 1950 an ETH	Relais	Lochstreifen	Ja (1950 bedingter Sprung ergänzt!)
von Neumann Architektur	John von Neumann	**30.6.1945**	**Architekturkonzept**		

Tabelle 2: Kerneigenschaften früher Rechensysteme – die ersten „Computer“

Die von Neumann - Architektur

Ein entscheidender und bedenkenswerter Wendepunkt trat ein, als John von Neuman den „First Draft of Report on the EDVAC“ mit Datum **30.6.1945, der „Geburtstag“ der von Neumann-Architektur und des stored program computers** schrieb, siehe http://www.virtualtravelog.net/entries/2003-08-TheFirstDraft.pdf .

Er war zu jener Zeit militärischer Consultant (Manhattan Projekt, Target Group) und Mitglied des Teams um Eckert und Mauchly (Moore School an der University of Pennsylvania). Dieses Team hatte stand kurz vor der Inbetriebnahme der ENIAC. Viele Mitglieder des Teams hatten Urlaub genommen und von Neuman machte wohl nicht mehr und nicht weniger, als die Ideen und Diskussionen zusammen zu fassen, die wohl maßgeblich von Eckert und Mauchly geführt worden waren – so zumindest eine Anekdote. Das Papier war lediglich als internes Memo gedacht, das einen Zwischenstand dokumentierte, nicht als Veröffentlichung. Das dieses Papier dann sehr weit bekannt wurde und seinen grundlegenden Einfluß auf die weitere Computerentwicklung ausübte, lag wohl daran, dass es viel Streit gab im Team um Eckert und Mauchly gab und diese auch im Frühjahr 46 ausschieden (Sie gründeten eine Firma.) Nachdem das Papier fast ein Jahr „herum gelegen“ hatte (als handschriftliche Notiz), entschloss Herman Hine Goldstine, es auf Maschine abschreiben zu lassen, setzte von Neumann als alleinigen Autor darauf und verschickte am 25.6.46 mindestens 24 Kopien an Teammitglieder und an viele Kollegen an Universitäten in den ganzen USA und auch nach England – damit war es öffentlich und entwickelte seine bahnbrechende Wirkung!

Dieses Papier war wohl den meisten Teilnehmers des Seminars an der Moore School der U Pennsylvania “*Theorie and Techniques of Electronic Digital Computers”* im Zeitraum 8.7. bis 31.8. 1946 bekannt. Quasi alle bedeutenden Computerentwickler waren vertreten: Stibitz (Bell Labs), Eckert und Mauchly (früher Moore School), Aiken (Havard), von Neuman, Goldstine (Princeton), D.R. Hartree (UK), usw. von Neumann spielte eine Nebenrolle, aber hielt zumindest auch einen Vortrag. Die notwendige Neuorientierung vieler an zuvor militärischen Projekten beteiligten hat diese erste Computer Conference ermöglicht und sicher zur Verbreitung der „von Neumann Architektur entscheident beigetragen.

Ab 1948 basierten eigentlich alle Rechner auf den Ideen und Konzepten, die von Neumann zusammengefasst hatte.[3] Das Papier selbst enthält keine Architekturzeichnung. Auch wenn die Prinzipien meist einheitlich beschrieben werden, so findet man in der Literatur doch sehr verschiedene Varianten. Nachfolgendes Bild, illustriert die Kernideen und ist direkt mit „Textauszügen“ aus dem Originalpapier beschriftet.

Die **Kernideen der von Neumann Architektur sind**:

1. Ein Computer besteht aus **5 Funktionseinheiten**: Speicher **M**, Steuerwerk oder Leitwerk **CC**, Rechenwerk, Eingabe, Ausgabe (Steuerwerk und Rechenwerk bilden den **Prozessor**)

[3] Natürlich gab es Weiterentwicklungen und auch die sogenannte Harvard Architektur (Aiken) mit der Trennung von Daten und Programm, die den so genannten von Neumann Flaschenhals vermeidet, aber das stored program Prinip wurde allgegenwärtig.

2. Im **Speicher** sind sowohl die zu bearbeitenden **Daten** als auch das **Programm** abgelegt.

3. Ein **Befehls-Ausführungszyklus** besteht aus der Folge:

6.	Befehl aus dem Speicher holen	FETCH
7.	Befehl im Steuerwerk interpretieren	DECODE
8.	Operanden holen	FETCH OPERANDS
9.	Befehl ausführen (von einer der Funktionseinheiten)	EXECUTE
10.	(Erhöhen des Befehlszählers	UPDATE INSTRUCTION POINTER)

(Das Konzept des Befehlszählers ist in dem Konzept nicht explizit erwähnt, aber notwendig.)

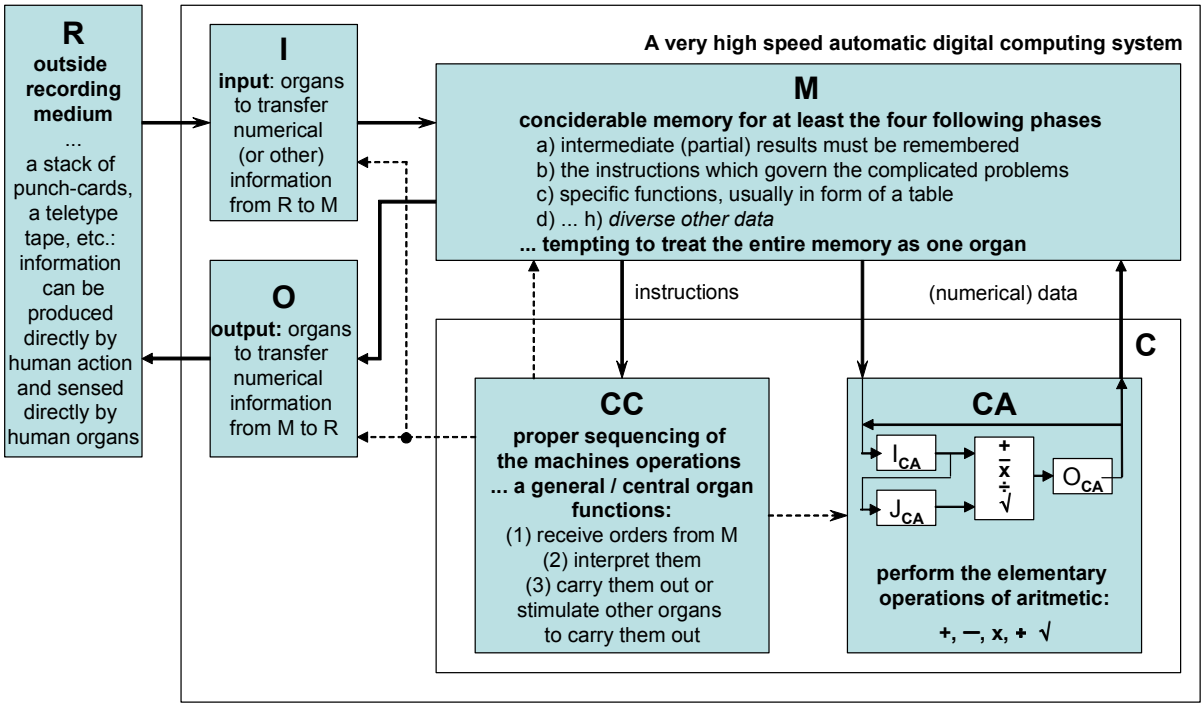

Abb. 4: Die „Original" von Neumann-Architektur. Extrahiert aus dem Text „First Draft of Report on the EDVAC" mit Datum **30.6.1945.** (DK)

Der Moderne Computer

Bis heute hat sich trotzt vielfältiger Kritik, Überlegungen und Variantenbildung die von Neumann Architektur erhalten. Fast alle Computer haben die von Neumann Architektur.

In den heutigen Computern sind die ALU (Arithmetic Logic Unit) und die Steuereinheit meistens in einem Baustein verschmolzen, der so genannten CPU (Central Processing Unit, zentraler Prozessor).

Der Speicher ist eine Anzahl von 0 ganzzahlig durchnummerierte „Zellen", die heute in der Regel 8 bit (0 oder 1) umfassen. Die Durchnummerierung nennt man Adresse. Ein Charakteristikum der „Von Neumann-Architektur" ist, dass der Inhalt einer Speicherzelle entweder (als Teil) eines Datums (also zum Beispiel der Buchstabe „A"), oder auch als (Teil eines) Befehls für die CPU („Addiere ...") betrachtet werden kann.

Das Steuerwerk ist dafür zuständig, den aktuell auszuführende Befehl, d.h. die Speicherzelle wo dieser steht zu kennen. Diese Adresse wird in einem speziellen Register, dem Befehlszähler (*program counter*) gehalten. Das

Steuerwerk liest einen Befehl aus dem Speicher, erkennt zum Beispiel „01101110“, erkennt dies als „Springe zu“ und holt den Operanden, in diesem Fall die Sprungadresse (ein bis acht Byte lang). Dann setzt sie den Befehlszähler auf eben diese Speicherzelle, um dort wiederum ihren nächsten Befehl auszulesen; der Sprung ist vollzogen. Wenn der Befehl zum Beispiel „Addiere“ lauten würde, dann würde sie aus der in der Folge angegebenen Adresse den Inhalt auslesen, um ihn dann beispielsweise an die ALU als Operanden weiterzuleiten.

Operanden können in heutigen Prozessoren sehr verschieden adressiert werden: direkt (die vollständige Adresse folgt direkt dem Begfehl) oder indirekt, z. B. über ein weiteres spezielles Register, dem Indexregister., als Einadress-, Zweiadress oder Dreiadressbefehl, usw. Hier gibt es sehr viele Varianten.

Die ALU hat neben der Logik zur Ausführung der Operationen in der Regel mindestens ein Register, den so genannten Akkumulator, meistens aber einen Registersatz, die als Zwischenspeicher für Operanden dienen.

Nach wie vor aktuell ist die Diskussion um RISC (*reduced instruction set computer) oder* CISC (complex *instruction set computer).* Historisch hatten sich immer mehr Befehlsarten entwickelt, was einen erheblichen Dekodierungsaufwand bedeutete. Eine Beschleunigungsidee war, mit möglichst wenigen Befehlen auszukommen und dadurch die Geschwindigkeit, den Prozessortakt erheblich erhöhen zu können.

Worin unterscheiden sich RISC- und CISC-Prozessoren besonders?

Eigenschaften	CISC	RISC
Anzahl der Register	Wenige (ca. 20)	Viele (bis zu 200)
Befehlssatz	Ca. 300 Befehle und mehr als 50 Befehlstypen	Nur rund 100 meist Register-orientierte Befehle
Adressierungsarten	Viele, z.B. 12	Nur 3 bis 5 Arten
Speicher /Caches	Gemeinsame Caches, aber heute auch getrennte	Getrennte Daten und Befehls-Caches (nach Harvard)
Taktzyklen zur Ausführung eines Befehls (CPI)	1 bis 20, durchschnittlich z.B. 4	1 bei den meisten Befehlen, im Schnitt 1,5
Befehlssteuerung	Mikrocode, z.T. auch hartverdrahte	Meistens hartverdrahtet
Beispielprozessoren	Intel x86 und AMD	Sun UltraSparc und PowerPC

Tabelle 3: Eigenschaften von CISC- und RISC-Computern

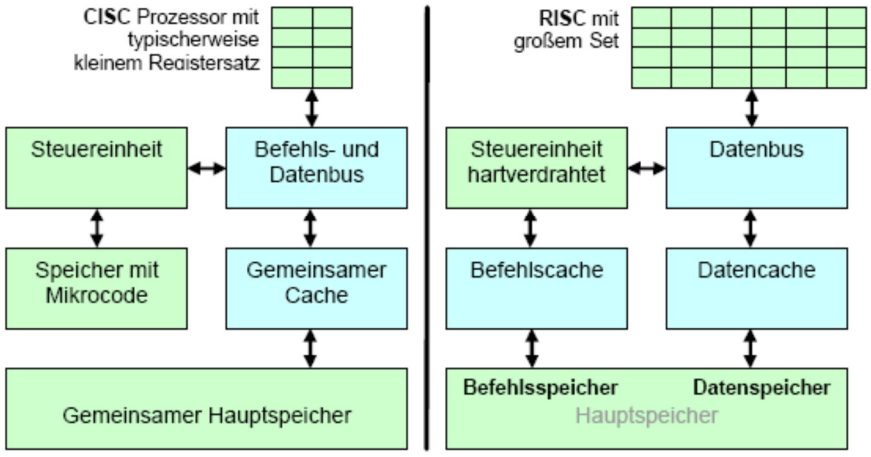

Abb. 5: Grundstruktur eines CISC und eines RISC-Computers

Hierzu werden Sie noch viele weitere Details in der Veranstaltung Hardware im 2. Semester hören.

2 Die Entwicklung der Programmiersprachen

Einteilung in Generationen

Erste Arbeiten im Bereich der Programmierung stammen von **Ada Lovelace** (ab etwa 1840 für die Analytical Engine), einer Bekannten von Charles Babbage. Sie erkannte das mathematische Potential der Maschine, verfasste eine relativ klare (wesentlich bessere) Beschreibung der Maschine als BABBAGE selbst und schrieb auch ein ziemlich kompliziertes Programm zur Berechnung von Bernoulli-Zahlen. (Randell 1982). Begeistert von den Möglichkeiten der Maschine schrieb sie, dass die Maschine **"algebraische Muster webe"**, so wie der JACQUARDsche Webstuhl gemusterte Stoffe herstelle. Ihre Programmiersprache war eine spezielle Tabellennotation. Leider starb sie relativ jung, ohne irgendeine Anerkennung für diese Arbeit erfahren zu haben; heute ist sie als **die erste Programmiererin** bekannt 1979 wurde die Programmiersprache Ada nach ihr benannt.

Wichtige theoretische Vorarbeiten wurden von Mathematikern in den dreißiger Jahren geleistet noch bevor Computer selbst existierten. Zu nennen sind insbesondere

- das Lambda-Kalkül (Stephen Cole Kleene und Alonzo Church), ~1935
- Grundsätzliches zur Berechenbarkeit (Alan Turing, Kurt Gödel) ~1936

Ein Vorläufer höherer Programmiersprachen war sicherlich auch der **Plankalkül** von Konrad Zuse in den Jahren 1942 bis 1946 entwickelt – allerdings erst 1972 komplett veröffentlicht. Ein Einfluss auf die heutigen Programmiersprachen ist nicht gegeben. Wenn man so will, war Plankalkül die erste höhere Programmiersprache der Welt. (Die Programmiersprache umfasst unter anderem Zuordnungsanweisungen (assignment statements), Funktionsaufrufe (subroutines), bedingte Anweisungen (conditional statements), Schleifen, Gleitkommaarithmetik (floating point arithmetic), Feldvariablen (Arrays), zusammengesetzte Datentypen, Ausnahmebehandlung und andere besondere Merkmale wie „zielgerichtete Ausführung“.)

Programmiersprachen werden üblicherweise in verschiedene Generationen eingeteilt, um damit zum einen die geschichtliche Entwicklung nachzuzeichnen. Andererseits ist dies ein Ansatz, die Sprachgenerationen als Grad der Abstraktion von der konkreten Maschinenarchitektur zu verstehen. Damit wird der Abstraktionsgrad von der zugrunde liegenden Technik mit jeder Generation erhöht. Die 1. Generation erfordert damit genaueste Kenntnis über die Funktionsweise der zu programmierenden Maschine, während man sich in den höheren Generationen immer mehr der natürlichen Sprache annähert. Dies ist bisher noch nicht vollständig realisiert, da die eindeutige Übersetzung für den Computer gewährleistet sein muss.

1. Generation: Tatsächlich war es bis in die 50er Jahre hinein notwendig, in der jeweiligen **Maschinensprache** zu programmieren. Das war nicht nur sehr mühsam, sondern auch sehr Fehleranfällig. In dieser Zeit entstand der Beruf des Programmierers – also derjenige der programmiert oder codiert.

2. Generation: Assembler ersetzen die Zahlencodes der Maschinensprache durch symbolische Bezeichner (Mnemonics). Eine Assembleranweisung lässt sich 1-zu-1 in einen Maschinenbefehl umsetzen, ist also immer noch direkt abhängig von der Maschine. Ein Assembler übersetzt ein Assemblerprogramm in Maschinencode. Für unser obiges Beispiel des MIPS-Prozessors würde der Beispiel-Befehl dann folgendermaßen aussehen:

allgemein:	`addiu rt,rs,imm`
konkret das Beispiel:	`addiu 2,1,4`

Assemblersprachen bieten typischerweise zusätzlich die Möglichkeit Speicherzellen mit symbolischen Namen zu versehen, also anstelle einer Adresse (eine Zahl zwischen 0 und höchster verfügbare Speicherzelle) einen Namen, wie etwa „Startwert“ oder „Ergebnis“ zu verwenden. Verglichen mit der direkten Maschinenprogrammierung erleichtert ein Assembler die Programmierung, aber konfortabel kann man das nicht nennen. Der Anteil der Assemblerprogrammierung ist dementsprechend heute sehr gering – wird z.B. aber noch genutzt, wenn man für bestimmte Programmteile eine maximale Geschwindigkeit benötigt, z.B. in embedded systems.

3. Generation: Höhere Programmiersprachen (High Level Languages) führen Konzepte wie z.B. Variablen ein, um leichter verständlichen Quelltext schreiben zu können. Ziel der Sprachen der 3. Generation war es, weitgehend maschinenunabhängig und leichter verständlich zu programmieren. *„She did this – the invention of the compiler -, she said, because she was lazy and hoped that „the programmer may return beeing a mathematician“*, so ein Zitat über Grace Murray Hopper (1906 – 1992) „Erfinderin des Compilers“ im Mark 1 Team (Aiken).

Abb. 6 Grace Murray Hopper (1906 – 1992) „Erfinderin des Compilers“

Die ganz überwiegend heute eingesetzten Programmiersprachen sind Programmiersprachen der Dritten Generation. Objektorientierte Sprachen sind ebenfalls Sprachen der 3. Generation. Um jedoch ihre konzeptionelle Sonderrolle zu betonen, werden sie in der Literatur oft als *„OO-Generation“* bezeichnet.

Beispiel: Es soll die Addition „3 + 4“ durchgeführt werden. In allen Programmiersprachen dieser Generation werden z.B. mathematische Notationen für die Operation „addiere“ direkt unterstützt. Man kann also schreiben:

```
int Summe;
Summe = 3 + 4;
```

In diesem Besipiel wird eine Variable „Summe“ vom Datentyp int (Integer) reserviert und dann in dem zweiten Statement dieser Variable der Wert 7 (3+4) zugewisen.

Die ersten derartigen Sprachen wurden in den 50er Jahren in den USA entwickelt und verbreiteten sich schnell:

- FORTRAN (**FOR**mula **TRAN**slator) von John Backus et al., 1954: für technisch-wissenschaftliche Anwenungen
- COBOL (**CO**mmon **B**usiness **O**riented **L**anguage) von Grace Hopper et al., 1959 für kaufmännische Anwendungen
- LISP (**LIS**t **P**rocessor) von John McCarthy et al., 1959, vor allem im akademischen Umfeld genutzt
- ALGOL (**ALGO**rithmic **L**anguage), eine Komitee-Entwicklung der *Association for Computing Machinery* (ACM) und der *Gesellschaft für Angewandte Mathematik und Mechanik* (GAMM), später dann der *International Federation for Information Processing* (IFIP), entwickelt. Beteiligt waren unter anderem John Backus, Friedrich Ludwig Bauer, John McCarthy, Peter Naur, Alan J. Perlis, Heinz Rutishauser und Klaus Samelson.

Die vorgenannten Sprachen existieren mit ihren Nachfolgern bis heute. Vor allem LISP beeinflusste die später entwickelten Programmiersprachen stark.

Ein großer Meilenstein war die Entwicklungvon ALGOL zwischen 1958 und 1962„ als ein internationales Komittee während einer Tagungsreihe eine „neue Sprache für Algorithmen" entwarf. Dies mündete in die Entwicklung von **Algol 60** (**ALGO**rithmic **L**anguage). In einem Tagungsbericht wurden viele Ideen, die zu dieser Zeit in der Fachgemeinschaft kursierten, aufgenommen und ebenso zwei Neuerungen: Zum einen die Verwendung der **Backus-Naur Form** (BNF) zur Beschreibung der Syntax der Programmiersprache. Nahezu alle folgenden Programmiersprachen benuzten die BNF, um die Syntax als kontextfreie Grammatik darzustellen. Zum anderen war die Einführung von **Gültigkeitsbereichen** neu (mehr dazu in der nächsten Vorlesung).

Obwohl Algol 60 sich in Nordamerika **nicht** durchsetzte (hauptsächlich aus geschäftspolitischen Gründen zum „Schutz von von IBMs **PL/1**", teilweise aber auch aus dem Grund, die Ein- und Ausgabe nicht Teil der Sprachdefinition zu machen, wurde Algol in der Folgezeit zum Standard in der (west)europäischen Welt. Sie beeinflusste die Ausbildung einer ganzen Generation von Informatikern und das Design späterer Sprachen, insbesondere von Simula, Pascal und Scheme.

In der Folgezeit wurde eine große Zahl weiterer Programmiersprachen entwickelt. Den größten Erfolg hatten dabei Weiterentwicklungen der bereits vorhandenen Programmiersprachen. Beispielsweise wurde um 1964 **BASIC** (**B**eginner's **A**ll-purpose **S**ymbolic **I**nstruction **C**ode) entwickelt, um Studenten den Einstieg in die Programmierung mit Algol und FORTRAN zu erleichtern. Diese BASIC-Schüler machten sie schließlich auch in dem Ende der 70er Jahre entstehenden Heimcomputerbereich populär. Auch die Programmiersprache **C**, 1972 für das neu entwickelte Betriebssystem Unix entworfen, hat ihre Wurzeln in Algol. Alle grundlegenden funktionalen Teile (insbesondere der sogenannte Kernel) von UNIX sind in C programmiert. Sie setzte sich aber auch für die Entwicklung allgemeiner Anwendungsprogramme durch. Beide Sprachen haben bis heute ein großes Feld von Varianten nach sich gezogen.

Als Reaktion auf die erste Software-Krise entwickelte Niklaus Wirth und Kathleen Jensen 1971 die Programmiersprache PASCAL, mit dem gänzlichen Verzicht auf einen Sprungbefehl – man benötigte nur wesentlich besser zu beherrschende Konstrukte wie Schleifen Siehe hierzu **Go To Statement Considered Harmful** von Edsger W. Dijkstra, Communications of the ACM, Vol. 11, No. 3, March 1968, pp. 147-148.

Es entstanden in dieser Zeit jedoch auch andere neue Konzepte, die bis heute dominierenden Programmierparadigmen (siehe unten). Große Bedeutung erlangte in der 2. Software-Krise das Objektorientierte Programmieren, das Daten-, Prozedur- und Referenzaspekte in dem einzigen Konzept des Objekts vereinigt: **Simula** von Kristen Nygaard und Ole-Johan Dahl begründete die Objektorientierte Programmierung, 1965.

Auch **Smalltalk** wurde bereits seit den frühen 70er Jahren im Xerox Palo Alto Research Center entwickelt, jedoch erst in den 80ern allgemein freigegeben. Bemerkenswert ist die konsequente Umsetzung des objektorientierten Programmierparadigmas in der Nachfolge von Simula und die Integration der Sprache in einer innovativen graphischen Benutzeroberfläche. Es folgten Sprachen wie **Ada**, 1980 und **C++** (Bjarne Stroustrup), die objektorientierte Erweiterung von C, 1983

Das schnelle Wachstum des Internets war Anfang der neunziger Jahre eine neue Herausforderung. Das Internet bildete eine völlig neue Grundlage für die Erstellung von Softwaresystemen und damit auch für die Entwicklung neuartiger Programmiersprachen. Dass sie sich schon früh in Webbrowsern integrierte, zeichnet die Programmiersprache **Java** aus und begründete ihre Popularität. Auch setzten sich verschiedenste Skriptsprachen für die Entwicklung von Webserver-Anwendungen durch: **Perl** und auch **Python** sind hierfür Beispiele. Obwohl keine der Sprachen fundamentale Neuerungen im Sprachdesign mit sich brachte, wurden nun Aspekte wie automatische Speicherbereinigung Typisierung stärker berücksichtigt. Immer größere Beachtung fand auch die Codesicherheit

und die Portabilität des Programmcodes, dies führte zur Entwicklung von so genannten virtuellen Maschinen (siehe unten) als Laufzeitumgebungen für die Programme.

Gelegentlich wird eine Fortsetzung dieser Genealogie von Programmiersprachen betrieben, aber die Eigenschaften dieser sprachen wird deutlicher, wenn man die jeweiligen Programmierparadigmen betrachtet.

4. Generation: Fourth Generation Language (4GL) Sprachen der 4. Generation sind anwendungsbezogene (applikative) Sprachen. Sie stellen u. a. die wichtigsten Gestaltungsmittel von Sprachen der 3. Generation zur Verfügung, zusätzlich jedoch Sprachmittel zur Auslösung von relativ komplexen, anwendungsbezogenen Operationen, beispielsweise zum Zugriff auf Datenbanken (meist per SQL) und zur Gestaltung von Benutzeroberflächen (GUIs).

Im Ansatz, die Sprachgenerationen als Abstraktion von der realen Maschine zu verstehen, sind 4GL-Sprachen deskriptive (beschreibende) Sprachen. Die Programmierung ist einen weiteren Schritt von der Technik entfernt. Als Beispiel für eine beschreibende Sprache der vierten Generation kann man SQL heranziehen, eine Datenbankabfragesprache. Hier wird z. B. der Befehl übermittelt: *„Gib mir alle Kunden im Ort Frankfurt am Main"*. Wie das Ergebnis erzielt wird (etwa Öffnen von Dateien, Positionieren eines Zeigers, etc.), ist im Gegensatz zu 3GL-Sprachen nicht mehr zu beschreiben.

Der Begriff 4GL wurde im Marketing einiger Hersteller von Entwicklungsumgebungen fälschlich missbraucht, um das Produkt als vermeintlich moderner darzustellen. Deshalb ist die Benutzung dieses Begriffes problematisch!

5. Generation: (Very High Level Language, VHLL) Sprachen der 5.Generation gestatten das Beschreiben von Sachverhalten und Problemen. Sie kommen vor allem im Bereich der KI (künstliche Intelligenz) zum Einsatz. Die Wahl des Problemlösungsweges kann dem jeweiligen System (weitgehend) überlassen werden. Bekanntestes Beispiel für eine Sprache der 5. Generation ist **PROLOG**.

Aktuelle Programmiersprachen – Popularität und Bedeutung

Zumindest für Programmieranfänger ist dies eine wichtige Frage. Schließlich will man ökonomisch lernen und nicht die Zeit damit „verplempern" Programmiersprachen zu erlernen, mit denen man später kaum etwas anfangen kann.

Um hier eine Antwort zu finden, verschaffen wir uns zunächst einen Überblick über die in den letzten rund 5o Jahren entwickelten Programmiersprachen: Dies waren und sind sehr, sehr viele: Schon 1969[4] bemühte Jean E. Sammet in Ihrem Buch "PROGRAMMING LANGUAGES: History and Fundamentals" das Bild der „Babylonischen Sprachverwirrung".

[4] Beachten Sie: Erst in 1969 wurde der Begriff „Informatik" in Deutschland gebräuchlich. Die „Sprachverwirrung fand also schon sehr früh statt.

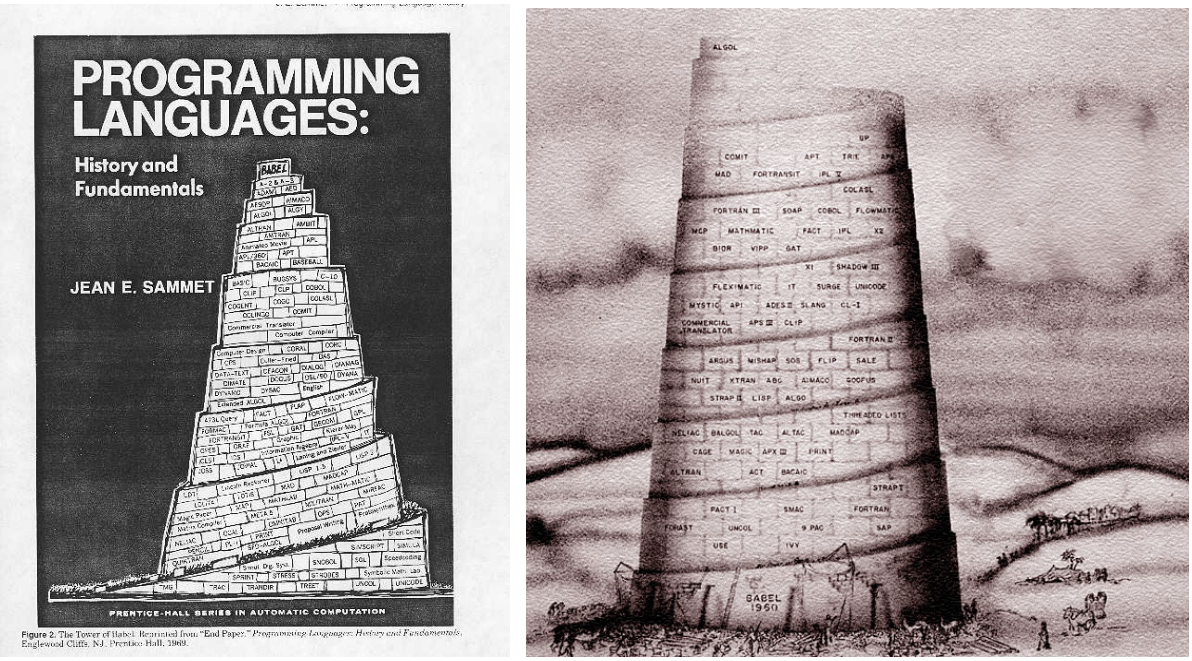

Abb. 7: Titelblatt des Buches von Jean S. Sammet und eine Variante des vielfach bemühten Bildes [aus: http://hopl.murdoch.edu.au/].

Die Website http://www.people.ku.edu/~nkinners/LangList/Extras/langlist.htm dokumentiert **mehr als 2500 verschiedene Programmiersprachen**. Viele Sprachen existieren zudem in mehreren Varianten/Dialekten.

Aber das ist längst nicht alles: Je nachdem, welche Kriterien man an die Auswahl der Sammlung legt, kann man auch noch deutlich mehr Programmiersprachen finden: Diarmuid Pigott listet in „ HOPL, the History of Programming Languages" [http://hopl.murdoch.edu.au/] mehr als **8500 Sprachen** mit fast 18.000 Bibliographischen Referenzen (Stand 18. März 2007).

Eine weitere interessante Informationsquelle findet man in dem „Dictionary of Programming Languages" von Neal Ziring [http://cgibin.erols.com/ziring/cgi-bin/cep/cep.pl]. Hier sind zwar nur 146 Programmiersprachen angeführt (Stand 18. März 2007) aber dafür in einer einheitlichen Form beschrieben und mit einem Programmierbeispiel bereichert.

Eine sehr bekannte Internetressource ist die von Éric Lévénez "Computer Languages History - Computer Languages Timeline" [http://www.levenez.com/lang/] mit zwar nur 50 betrachteten Programmiersprachen, dafür aber jeweils die Entwicklung aufzeigend und zudem „verwandtschaftliche" Zusammenhänge. Die folgende Abbildung ist ein erweiterter Extrakt:

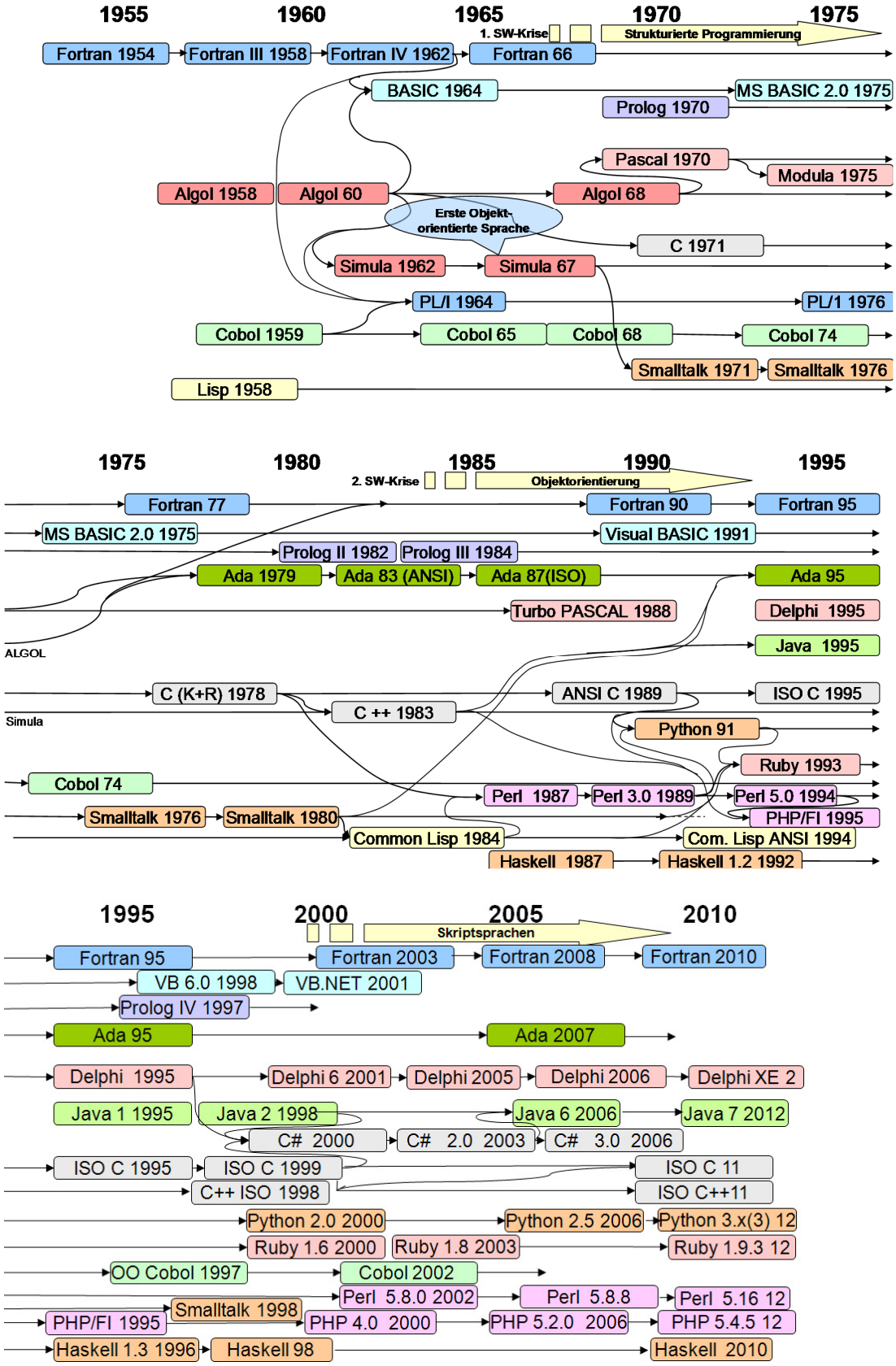

Abb. 8: Die Ideengeschichte der höheren Programmiersprachen 1955-2012.

Sowohl in der Abbildung als auch in der Tabelle erkennen wir zunächst erst einmal die „Dinosaurier“ der Programmiersprachen: vor allem Fortran (die erste höhere Programmiersprache überhaupt) und Cobol. Insbesondere im Kontext der 1. Programmierkrise wurde in „Informatiker-Kreisen“ über diese Sprachen gelächelt und diese wurden nicht mehr gelehrt – trotz alledem: Sie haben überlebt, die Haupt-Anwender (Fortran: Theoretische Physik, Theoretische Chemie, …; Cobol: kaufmännische Anwendungen) blieben diesen Sprachen treu und diese werden bis heute weiterentwickelt und den neuen Paradigmen angepasst.

Andere Programmiersprachen starben aus: Pascal (lebt weiter in der modernen Variante Delphi) sowie Algol und PL/1 (der Versuch von IBM, die damaligen Haupt-Programmiersprachen Fortran, Cobol und Algol „zusammenzufassen“). Einen anderen „Vereinigungsversuch“ stellt Ada dar. Das US Verteidigungsministerium hat ab Mitte der siebziger Jahre versucht, zumindest für seine Projekte eine Programmiersprache „vorzuschreiben“. In diesem Anwendungsbereich „lebt“ Ada noch – konnte sich aber weder in zivilen noch akademischen Kreisen verbreiten.

Auffallend ist auch die Zunahme der häufiger genutzten Programmiersprachen:

1960	4 Sprachen,
1980	ca. 10 Sprachen,
2000	ca. 20 Sprachen.

Versucht man die Popularität der Programmiersprachen zu bestimmen (siehe unten: Tiobe-Index), so stellt man fest:

Die 5 häufigsten Sprachen erreichen eine Popularität von kanpp 60%,
die 10 häufigsten von knapp 80% und
die 20 häufigsten von etwa 90%.

In Benutzung sind deutlich mehr als 100 Programmiersprachen – eine Unmöglichkeit diese alle kennenzulernen: Sie müssen die Prinzipen kennen und einige Exemplare, daraus lässt sich die Welt der Programmiersprachen erobern.

Bei den deklarativen Sprachen gibt es zwei Hauptlinien: Prolog (logisch) und Lisp/Haskell (funktional). Beide Linien sind nicht sehr ausgeprägt.

Obwohl die Objektorientierung eigentlich ein „frühes“ Konzept ist (Simula, Smalltalk) konnte es sich erst nach der zweiten Softwarekrise wirklich durchsetzen: C++, Java, Python, Ruby. Auch die „Dinosaurier“ konnten sich diesem Trend nicht widersetzen. In den meisten modernen Sprachen wird Objektorientierung unterstützt.

Aktuell gibt es einen deutlichen Trend zu Skriptsprachen, das heißt zu interpretierten Sprachen. Tatsächlich versprechen diese Sprachen schnelleres entwickeln, aber auch das Risiko, dass Programmierfehler zur Übersetzungszeit nicht erkannt werden und damit zu „Laufzeit-Fehlern“ werden, die schwerer und teurer zu beheben sind.

Wie orientiert man sich also in diesem „Babylon“ – was sollte man, was muss man kennenlernen? Ein Hinweis darauf könnte die Popularität von Programmiersprachen liefern. Leider ist diese nicht einfach zu messen: Was sind die Kriterien? – Ein Beispiel ist der „Tiobe Programming Index“
http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html .

Dieser Index wird seit 2001 einmal pro Monat veröffentlicht und basiert (angeblich) auf:

- der Verfügbarkeit von „Experten“ (in dieser Sprache),
- angebotenen Programmierkursen und
- Produkten und kommerziellen Angeboten

Konkret wird in mehreren Suchmaschinen (Google, MSN, Yahoo!, Google newsgroups and blogs) für jede Programmiersprache das folgende Suchwort eingegeben und die Hits gezählt:

```
+"<language> programming"
```

Dabei wird auch nach Synonymen gesucht, z.B. für C# auch nach C-Sharp und C Sharp – aber nur das Maximum in die Wertung einbezogen. „In the future we will do a better job and take the union (from mathematical set theory) of all the hits." – Na gut, es ist sowieso nur rein grober Indikator!

Eine Beschreibung der Vorgehensweise findet man unter http://www.tiobe.com/index.php/content/paperinfo/tpci/tpci_definition.htm . Diesen Index kann man also als Indikator für die Popularität einer Programmiersprache ansehen – ein absolutes Maß ist es aber nicht! Zu den Ergebnissen des Juli 2012:

Position Jul 2012	Position Jul 2011	Delta in Position	Programming Language	Ratings Jul 2012	Delta Jul 2011	Status
1	2	↑	C	18.331%	+1.05%	A
2	1	↓	Java	16.087%	-3.16%	A
3	6	↑↑↑	Objective-C	9.335%	+4.15%	A
4	3	↓	C++	9.118%	+0.10%	A
5	4	↓	C#	6.668%	+0.45%	A
6	7	↑	(Visual) Basic	5.695%	+0.59%	A
7	5	↓↓	PHP	5.012%	-1.17%	A
8	8	=	Python	4.000%	+0.42%	A
9	9	=	Perl	2.053%	-0.28%	A
10	12	↑↑	Ruby	1.768%	+0.44%	A
11	10	↓	JavaScript	1.454%	-0.79%	A
12	14	↑↑	Delphi/Object Pascal	1.157%	+0.27%	A
13	13	=	Lisp	0.997%	+0.09%	A
14	15	↑	Transact-SQL	0.954%	+0.15%	A
15	25	↑↑↑↑↑↑↑↑↑↑	Visual Basic .NET	0.917%	+0.43%	A
16	16	=	Pascal	0.837%	+0.17%	A
17	19	↑↑	Ada	0.689%	+0.14%	B
18	11	↓↓↓↓↓↓	Lua	0.684%	-0.89%	B
19	21	↑↑	PL/SQL	0.645%	+0.10%	A--
20	26	↑↑↑↑↑↑	MATLAB	0.639%	+0.19%	B

Akzeptiert man dieses Popularitätsmaß, so sind auch folgende Zahlen interessant:

Category	Ratings July 2012	Delta July 2011
Object-Oriented Languages	57.1%	+1.5%
Procedural Languages	37.4%	-0.5%
Functional Languages	3.6%	-1.3%
Logical Languages	1.9%	+0.3%

Category	Ratings July 2012	Delta July 2011
Statically Typed Languages	71.6%	-0.1%
Dynamically Typed Languages	28.4%	+0.1%

Interessant sind aber vor allem die längerfristigen Trends, die in der folgenden Graphik wiedergegeben sind.

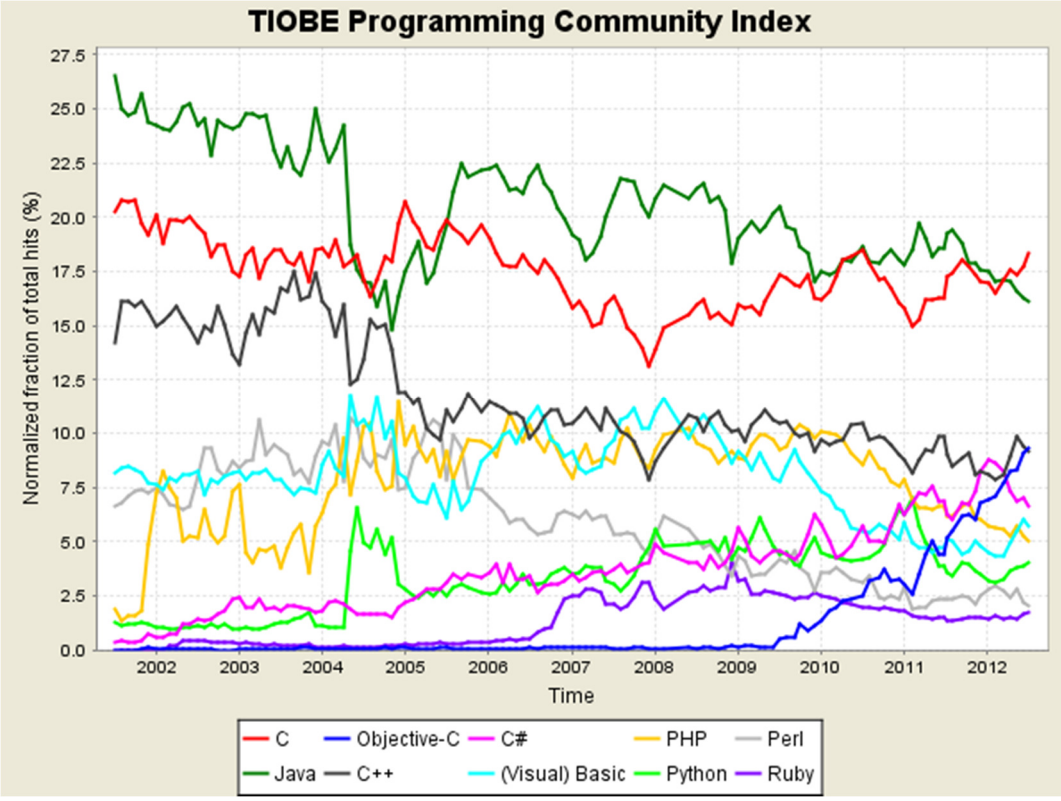

Abb. 9: Tiobe Programming Community Index 2001 – 2012
http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

Bezüglich ihrer Popularität sind bei den top-ten klar drei Gruppen erkennbar:

- Die Spitzenreiter: Java und nach wie vor C,
- Die Mittelgruppe: C++, Visual Basic, PHP und Perl,
- Die noch populären Programmiersprachen: Python, C#, Ruby und als „Newcomer" Objective-C (Apple).

Dies sind alles imperative Sprachen; eine deklarative (funktionale z.B. Haskell auf Platz 37 oder logische Prolog auf Platz 34) ist vorne nicht dabei. In der Praxis konnten sich diese Ansätze nicht durchsetzen; sie sind eben doch „akademische Sprachen".

Die nachfolgende Tabelle zeigt die langfristige Entwicklung der Popularität der TOP 10 Programmiersprachen:

Programming Language	Position July 2012	Position July 2007	Position July 1997	Position July 1987
C	1	2	1	1
Java	2	1	5	-
Objective-C	3	46	-	-
C++	4	3	2	6
C#	5	7	-	-
(Visual) Basic	6	4	4	5
PHP	7	5	-	-
Python	8	8	23	-
Perl	9	6	6	-
Ruby	10	10	-	-
Lisp	13	15	16	3
Ada	17	16	12	2
COBOL	30	17	3	11

Vielleicht sollte man sich einen „Spruch" merken:

There are only two kinds of languages: the ones people complain about and the ones nobody uses.

Bjarne Stroustrup (C++Entwickler)
nach http://www.cs.technion.ac.il/~imaman/stuff/hopl.pdf