Modul: Programmierung B-PRG 1
PRG1 und EPR

V01 Computer – Algorithmus – Programm

Prof. Dr. Detlef Krömker
Professur für Graphische Datenverarbeitung
Institut für Informatik
Fachbereich Informatik und Mathematik (12)

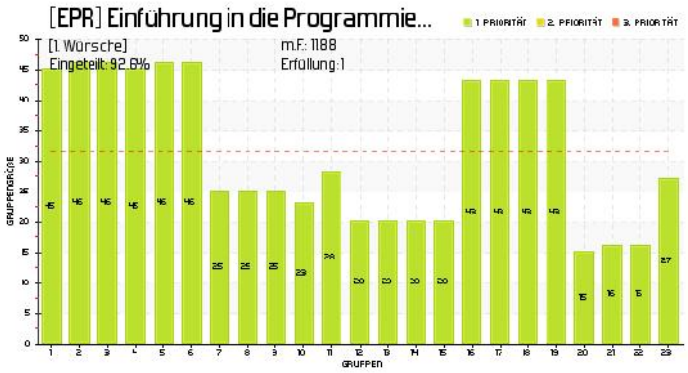
Rückblick

Organisation

1. **Sie sollten sich beim Moodle-Kurs angemeldet haben!**
   (Das haben leider noch nicht alle gemacht!)
1. Sie **sollten** sich zu einer Übungsgruppe angemeldet haben!

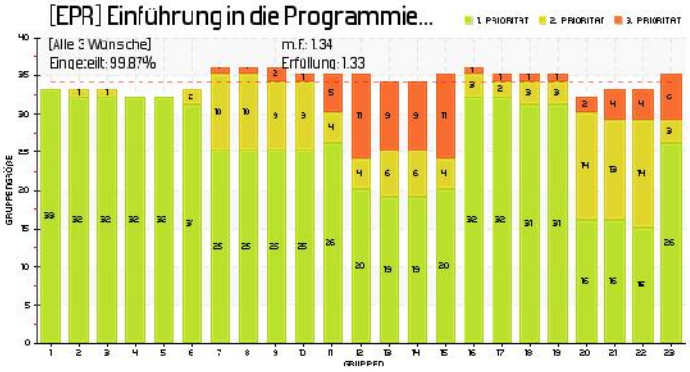
2. Sie sollten einen RBI-Account beantragen.
3. Sie sollten sich mit Linux vertraut machen.
4. **Sie sollten ihre eigene Arbeitsumgebung für Python anlegen**.
5. **Sie sollten Zweierteams (x2 = Lerngruppen) bilden!**

Übungsgruppen
Einteilungen EPR Erst-Wünsche



N = 787

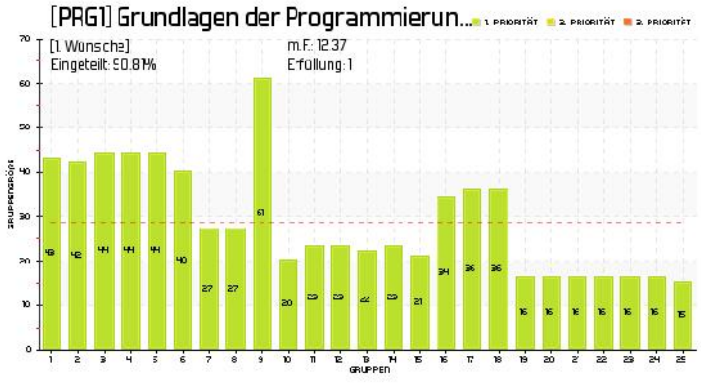
Einteilungen EPR Zuteilung



N = 787

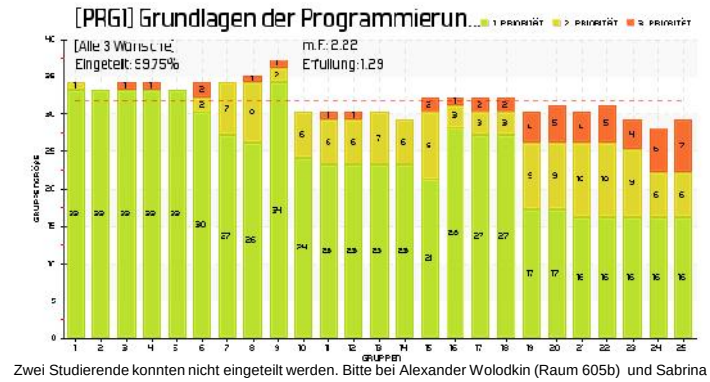
Eine(r) Studierende® konnte nicht eingeteilt werden. Bitte bei Alexander Wolodkin (Raum 605b) und Sabrina Safre melden.

Einteilungen PRG 1 Erst-Wünsche



N = 795

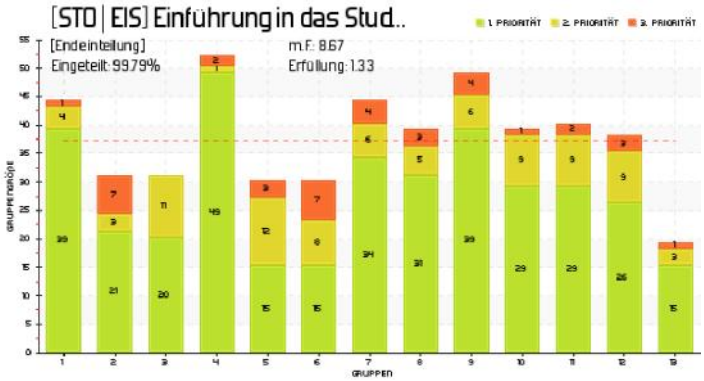
Einteilungen PRG1 Zuteilung



N = 795

Zwei Studierende konnten nicht eingeteilt werden. Bitte bei Alexander Wolodkin (Raum 605b) und Sabrina Safre melden.

Einteilungen STO/EIS Zuteilung



N = 487

alle eingeteilt

Kontaktinfos

- Alexander Wolodkin wolodkin@studiumdigitale.uni-frankfurt.de
- Raum 605b im Matheturm

- Sabrina Safre sabrinasafre@gmail.com

Noch nicht alle Gruppen in Moodle eingetragen, erfolgt heute im Laufe des Tages.

Danach dort Änderungswünsche annoncieren und Tauschpartner suchen!

FAQs (die klitzekleinen Trostpflästerchen)

- **Wenn ich den Anmelde-Termin verpasst habe, was dann?**
- **Wenn ich die Übungsgruppe wechseln will - egal warum?**

- Nachrücker/Wechsler können sich nächste Woche noch anmelden!
  - 1. In auge/Moodle schauen, wo noch Platz ist
  - 2. Direkt zum Tutor gehen und ihn/sie fragen, ob er/sie noch jemanden aufnimmt. Dieser gibt uns dann ggf. die Änderung bekannt.

Fragen / Kommentare

- ???

- Die Ergebnisse der Veranstaltungen MOD, Mathe 1 und GL-1 wurden **nicht** veröffentlicht. … Das machen die Veranstalter selbst!

Unser heutiges Lernziel

Drei zentrale Grundbegriffe kennen lernen:

- **Computer**
- **Algorithmus** und erste Beschreibungsmöglichkeiten
- **Programm** (und Programmiersprachen: Genealogie und Paradigmen)

Begriffsbestimmung/-klärung ist unser Ziel – **nicht strenge Definition!**

Übersicht

- Was ist ein Computer? - Auch kurze Anmerkungen zur Geschichte
- Erste Begriffsklärung: Algorithmus
- Programm
- Compiler versus Interpreter … Virtuelle Maschine
- Programmiersprachen
  - Genealogie: Die Entwicklung von Programmiersprachen
  - Paradigmen

Was ist ein Computer? - Begriffsbestimmung

Eine **Maschine** (heute i.d.R. eine **digital-elektronische**) zur Speicherung und automatischen Verarbeitung von Daten resp. Informationen (z.B. für mathematische Berechnungen oder allgemeiner Zeichenersetzungen) durch Angabe einer **programmierbaren (Rechen-)vorschrift**.

Die programmierbare, d.h. veränderbare (Rechen-)vorschrift, nennen wir **Programm**

➔ **Hardware** (die Elektronik + Gehäuse), das **„unveränderbare“**

➔ **Software** (das Programm), das **„veränderbare“**

Aus der Begriffsbestimmung ergibt sich u.a.: Was war der **erste Computer**?

Der Begriff „Computer“

- **Computer =** Rechner oder Rechenanlage lässt sich aus dem lateinischen *computare* (zählen, rechnen) herleiten.
- entstammt im modernen Deutsch eher dem Englischen *computer.*
- abgeleitet vom Verb *to compute* (rechnen).
- **ursprünglich**: „Computer“ war ein Beruf: bezeichnete Menschen, die im Mittelalter für Astronomen die quälend langwierigen Berechnungen vornahmen.
- Zunächst wurden die „Arbeiter“, die Rechenmaschinen Computer bezeichnet,

  später ging der Begriff auf „diese“ Maschinen über.

Thomas J. Watson Erste IBM-Computer Ende 40er **„Calculator“**

von Neumann Architekur

benannt nach **John von Neumann** ist ein **Schaltungskonzept (eine Architektur)** zur Realisierung **universeller Rechner** (Von-Neumann-Rechner) welches folgende Komponenten enthält:

- **Rechenwerk** (führt Rechenoperationen und logische Verknüpfungen aus)
- **Speicher** (speichert sowohl Programme als auch Daten
- **Steuerwerk** (interpretiert die Anweisungen eines Programms und steuert die Ausführung dieser Befehle)
- **Eingabe-/Ausgabewerke** (steuert die Ein- und Ausgabe von Daten)
- (Verbindungssystem)

Das Rechen- und Steuerwerk (wird zusamen manchmal auch Leitwerk genannt) bilden die so genannte Zentraleinheit, den Prozessor (CPU).

von Neumann Architekur (2)

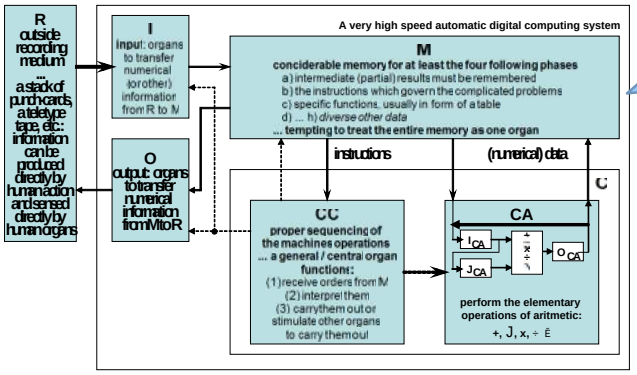
Dieses Konzept wurde 1945 in dem Papier "*First Draft of a Report on the EDVAC*" im Rahmen des Baus der EDVAC beschrieben
war zur Zeit seiner Entwicklung „revolutionär" (oder auch nicht: Konrad Zuse hatte schon 1938 seinen Z3 (Relaisrechner) realisiert).

Anderer Rechner jener Zeit hatten ein festes Programm, das z.B. durch Schaltdrähte „programmiert" wurde oder waren nicht programmierbar.

In einer Von-Neumann-Architektur war es nun möglich, Änderungen an Programmen sehr schnell durchzuführen oder in kurzer Folge ganz verschiedene Programme ablaufen zu lassen, ohne Veränderungen an der Hardware vornehmen zu müssen

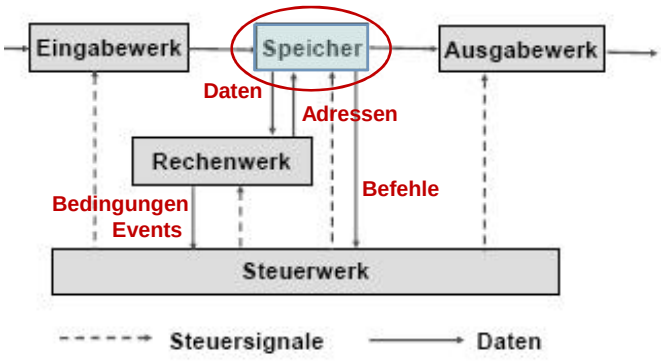
Diese von Neumann-Architektur setzte sich sehr schnell durch.

Die „Original“ von Neumann-Architektur. Extrahiert aus dem Text „First Draft of Report on the EDVAC“ mit Datum 30.6.1945



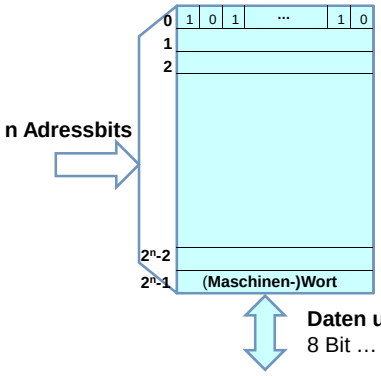
Dies Schema hat nur historische Bedeutung … muss man sich NICHT merken.

von Neumann Architektur modern dargestellt (3)



Das muss man sich merken!

Basis-Architektur des Speichers



- heute: 1-64 Gbyte, meist Byteweise (=8 Bit) adressierbar
- n = 16 ➔ 2^{16} = 65.536 Wörter (Bytes) Speicherkapazität
- n = 32 ➔ 2^{32} = 4 GByte

Die drei Kernideen der von Neumann Architektur

1. Ein Computer besteht aus **5 Funktionseinheiten**: Speicher, Steuerwerk (oder Leitwerk), Rechenwerk, Eingabe, Ausgabe (Steuerwerk und Rechenwerk bilden zusammen 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 (von der Adresse im Befehlszähler IC) — FETCH
   2. Befehl im Steuerwerk interpretieren — DECODE
   3. Operanden holen — FETCH OPERANDS
   4. Befehl ausführen — EXECUTE

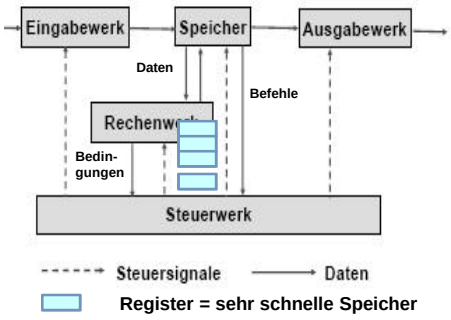
   (5. Erhöhen des Befehlszählers — UPDATE IC)

Wie funktioniert ein Computer?
Ein typisches von Neumann Maschinenbefehlswort

(Maschinen-)Befehl einer Registermaschine
bestehend aus einem (festen) Bitmuster

Op-Code kodiert die auszuführende Operation

Operandenfeld: kodiert die zugehörigen Operanden



Die Maschinenbefehle beziehen sich exakt auf die Hardware (Rechenwerk, Speicher, Eingabe und Ausgabewerk eines Typs von Maschine.

Beispiele

16 Bit-Befehl: „Lade den Inhalt der Speicherzelle Nr. 108 ins Register Nr. 5“

Binär	0 0 0 1	0 1 0 1	0 1 1 0	1 1 0 0
Hexadezimal	1	5	6	(12 =) C
	Op-Code	Zieloperand	<==	Quelloperand
	Lade	Register 5		Speicherzelle 108_{10}

Ein MIPS (32 Bit)-Maschinenprogramm binär und hexadezimal

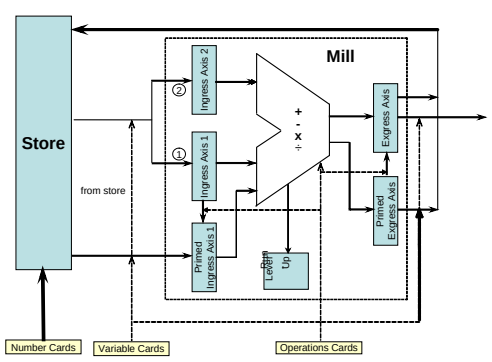
00100111101111010111111111100000	27BDFFE0
10101111101111110000000000010100	AF ...
10101111101001000000000000100000	
10101111101001010000000000100100	

Wie war es davor – bis etwa 1950 (1)

Name	Haupt-entwickler	Fertigstel-lung	Techno-logie	Program-mierbar?	Turing-vollständig
Analytical Engine (nicht realisiert)	Charles Babbage	**(1833 -1842)**	Mechanik	Lochstreifen	Ja
ABC Computer	John V. Atanasoff Clifford Berry	1941	Elektronen-röhren	Nein	Nein
Z 3	Konrad Zuse	1941	Relais	Lochstreifen	(Ja)
Colossus	M.H.Newman I.J. Good	1944	Elektronen-röhren	Datenband und Verkabelung	Nein

während des 2. Weltkrieges

Architektur der „Analytical Engine“ (1833 – 1842)



Auch dieses nicht lernen …
Nur zur Info!

Wie war es davor – bis etwa 1950 (1)

voll erfüllt
teilweise erfüllt
nicht erfüllt

Name	Haupt-entwickler	Fertigstel-lung	Techno-logie	Program-mierbar?	Turing-vollständig
Analytical Engine (nicht realisiert)	Charles Babbage	**(1833 -1842) 100 Jahre**	Mechanik	Lochstreifen	Ja
ABC Computer	John V. Atanasoff Clifford Berry	1941	Elektronen-röhren	Nein	Nein
Z 3	Konrad Zuse	1941	Relais	Lochstreifen	(Ja)
Colossus	M.H. Newman I.J. Good	1944	Elektronen-röhren	Datenband und Verkabelung	Nein

Wie war es davor – bis etwa 1950 (2)

Name	Haupt-entwickler	Fertigstel-lung	Techno-logie	Program-mierbar?	Turing-vollständig
Harvard Mark I (ASCC)	Howard H. Aiken **➔ IBM**	1944	Mechanik (und Relais)	Lochstreifen	Ja
ENIAC	Mauchley und Eckert	1946	Elektronen-röhren	Verkabelung	Ja
von Neumann Architektur (First Draft ... EDVAC „Electronic Discrete Variable Computer“ 1945 datiert, veröffentlicht 1946 aus der Gruppe Eckert+Mauchly heraus, beide ENIAC)					
Manchester Mark I Prototyp „Baby“	F. Williams U Man	**1948**	Elektronen-röhren	*„stored program“*	Ja

und alle weiteren!

Also, wer war der Erfinder des Computers?

Z3	Konrad Zuse?
ENIAC	John Mauchley, Presper Eckert?
Analytical Engine	Charles Babbage
	John von Neumann **30.6.1945**
	…

... also: Nicht eine Person (!):

Der Computer ist eine "Gemeinschaftserfindung"! (auch wenn sich die Erfinder nicht persönlich kannten!)

Entwicklung ab 1950 – „Elektronengehirne"

In den USA vor allem

- mit „Starthilfe" von Howard Aiken ➔ (der Computerhersteller) IBM
- Eckert und Mauchly (eigene Firma) ➔ UNIVAC
- F. Williams (U Manchester) ➔ Ferranti
- Konrad Zuse ➔ Zuse KG (AG)

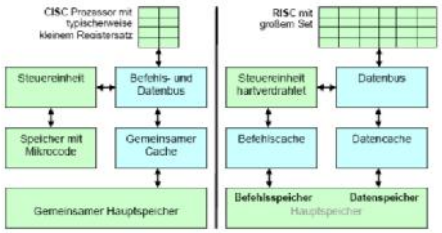
1960 in den USA: „**IBM und die sieben Zwerge**" – gemeint waren UNIVAC 12 % Marktanteil , Burroughs 3 %, NCR 3 %, CDC, RCA, Honeywell, GE: **IBM beherrschte 70% des US Marktes: „Big Blue"**

Heute

Die allermeisten Computer haben eine (modifizierte) von Neumann Architektur. Sind Multiprozessoren, d.h. haben oft mehrere Prozessorkerne (Cores). Wir unterscheiden:

CISC
Complex Instruction Set Computer

RISC
Reduced Instruction Set Computer



Durchatmen!

Computer – Algorithmus – Programm

✓

In dieser Veranstaltung nur „Nebenthema“
(Viel) mehr dazu in

Hardwarearchitekturen und Rechensysteme (SoSe)

Algorithmus

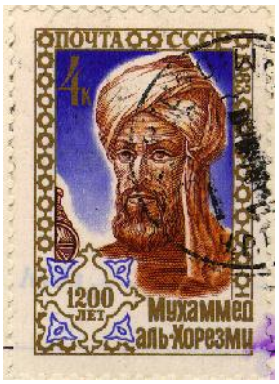
Das Wort Algorithmus ist eine Abwandlung oder Verballhornung des Namens von Muhammad ibn Musa al-Chwarizmi (* ca. 783, † ca. 850), dem Autor des Buchs Hisab al-dschabr wa-l-muqabala (825, Regeln zur Wiederherstellung und Reduktion), durch welches die Algebra im Westen verbreitet wurde.

Die lateinische Fassung beginnt mit „Dixit Algoritmi...“ (Algoritmus sprach...), womit der Autor gemeint war.

Das Wort Algebra stammt ebenfalls (al-Jabr – „Einrenkung“) aus dem Titel des Buches. Ursprünglich stand das Wort Algorism nur für die Regeln zur Arithmetik mit arabischen Ziffern.

Heute steht **Algorithmus** für alle geregelten "Prozeduren" (Handlungsvorschriften) , mit denen Probleme aller Art gelöst werden können.

Ein Porträt ?



Wesen des Begriffs Algorithmus

- Unter einem Algorithmus versteht man (allgemein) **eine genau definierte Handlungsvorschrift** (diese beschreibt einen Weg, wie man ein bestimmtes Ziel erreichen kann) zur Lösung eines Problems oder einer bestimmten Art oder Klasse von Problemen.
- Alltagsalgorithmen: ein Kochrezept, ein Algorithmus? – zumindest dann, wenn alle Angaben **genau genug** sind und es für alle Teilaufgaben, wie Braten, Rühren, etc., ebenfalls Algorithmen gibt.
- Auch Reparatur- und Bedienungsanleitungen oder Hilfen zum Ausfüllen von Formularen, Verhaltensvorschriften (im Fall von Feuer, o.ä.) sind eigentlich Algorithmen ... aber oft **zu unpräzis.**

➔ Also: Welche **weiteren Eigenschaften** erwarten wir (in der Informatik) von einem Algorithmus?

Ein Algorithmus hat folgende sieben Eigenschaften (fast strenge Fassung):

- Ausführbarkeit
- Allgemeingültigkeit
- Determiniertheit
- Determinismus
- Statische Finitheit
- Dynamische Finitheit
- Terminiertheit

Ausführbarkeit

- Die Anweisungen (Teile der Verhaltensvorschrift) müssen verständlich formuliert und ausführbar sein.
- Für den "ausführenden" Teil (Mensch / Maschine / …) muss eine "Sprache" vereinbart sein, die "verstanden" wird.
- Nicht festgelegt ist die Form:

  "Der Algorithmus kann in jeder beliebigen, dem Menschen oder der ausführenden Maschine zugänglichen Form beschrieben werden, zum Beispiel auch gezeichnet, skizziert, getanzt oder sonst etwas werden."
- In der Informatik sind die ausführenden Maschinen:
  Computer oder Menschen.

Wie formuliert (beschreibt) man Algorithmen wegen „ genau definierte Handlungsvorschrift“

1. Durch eine präzise Beschreibung in einer natürlichen Sprache **(?)**
2. Mittels einer Pseudo-Programmiersprache (Pseudo-Code): Folge von

   Zuweisung: ***name ← Ausdruck***
   Verzweigung: *if <Bedingung>* ***then*** *(Aktivitätsfolge)*
   else *(Aktivitätsfolge)*

   while *<Bedingung>* ***do*** *(Aktivitätsfolge)*

   *(****und mehr braucht man eigentlich nicht --- nach dem Prinzip: alles was verständlich, präzis und eindeutig ist, reicht!)***
3. Durch eine Programmiersprache (formale Sprache) (später).
4. Durch geeignete Abbildungen / Diagramme.

Es gibt viele andere Möglichkeiten …
Wenn Sie es können, meinetwegen auch tanzen!

Allgemeingültigkeit

Der Algorithmus löst eine Menge ähnlicher Probleme, eine ganze Problemklasse (und nicht nur ein Einzelproblem).

Bsp. $2x^2 - 5x + 9 = 0$ (ist ein **Einzelproblem**)

"Alle quadratische Gleichungen" ist eine **Problemklasse**.

Meist behandeln wir **"Rechenvorschriften". Rechnen** wird dabei sehr weit gefasst: Schließt Textverarbeitung und Sensoren und Aktoren (Robotics) mit ein.

Algorithmus-Eigenschaft: Determiniertheit

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

Algorithmus-Eigenschaft: Determinismus, d.h. deterministisch (ein wichtiger Unterschied)

- 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 gibt es neben dem Determinismus auch den Nichtdeterminismus, der aber in der Praxis kaum Verwendung findet, aber z.B. bei Quantencomputern, welche auch solche Algorithmen erfolgreich ausführen.
- Es gilt übrigens**: Jeder deterministische Algorithmus ist auch determiniert**. Nicht jeder determinierte Algorithmus ist jedoch deterministisch.

Algorithmus-Eigenschaft: Statische Finitheit

Kurz: Die **Beschreibung des Algorithmus ist endlich.**

Die Beschreibung eines Algorithmus darf nicht unendlich groß sein.

Als statische Finitheit wird die Endlichkeit des (Programm-)Quelltextes bezeichnet. Der Quelltext darf nur eine begrenzte Anzahl, wenn auch bei Bedarf sehr viele Regeln/Anweisungen enthalten.

Algorithmus-Eigenschaft: Dynamische Finitheit

Kurz: **Menge an Daten inklusive Zwischenspeicherungen sind zu jeder Zeit endlich.**

Zu jedem Zeitpunkt der Ausführung darf der von einem Algorithmus benötigte Speicherbedarf nicht unendlich groß (über alle Maßen wachsen) sein. Andernfalls wäre der Algorithmus nicht ausführbar.

Algorithmus-Eigenschaft: Terminiertheit

Kurz: **Der Algorithmus bricht nach endlicher Zeit kontrolliert ab**.

- Algorithmen sind terminiert, wenn sie für jede mögliche Eingabe **nach** einer **endlichen Zahl von Schritten zu einem Ergebnis kommen**. Die tatsächliche Zahl der Schritte kann dabei beliebig groß (aber nicht unendlich) sein.
- Steuerungssysteme und Betriebssysteme und auch viele Programme, die auf Interaktion mit dem Benutzer zielen, erfüllen diese Eigenschaft nicht: Wenn der Benutzer keinen Befehl zum Beenden gibt, läuft das Programm endlos weiter.

Der informatische Begriff "*Algorithmus*"

- **Ist durchaus schwer zu fassen – nicht trivial!**
- Viele Begriffe, die wir zur Beschreibung der Eigenschaften benutzen, sind nicht präzis definiert. **Problem!**
- Unter bestimmten Bedingungen können manche Eigenschaften nicht gefordert werden – sind also nicht zwingend.
- Insbesondere in der Theorie reicht das **nicht:** Führt hier aber zu weit ➔ Algorithmik, z.B. in "*Algorithmen und Datenstrukturen*"
- *Eine Berechnungsvorschrift zur Lösung eines Problems heißt genau dann Algorithmus, wenn eine zu dieser Berechnungsvorschrift äquivalente Turingmaschine existiert, die für jede Eingabe, die eine Lösung besitzt, also stoppt.*
- ➔ Begriffe, wie "Turing vollständig" (siehe Computer)

Interessante Aussagen über Algorithmen:

- **Berechenbare Funktion:** Eine Funktion f: **M** → **N** heiß berechenbar, wenn es einen **Algorithmus** gibt, der für jeden Eingabewert m ∈ **M,** für den f (m) definiert ist terminiert und ein Ergebnis liefert.

  Anschaulich: Eine Funktion ist berechenbar, wenn man sie programmieren kann.
- **Auch: Es gibt nicht berechenbare Probleme. ??? ➔ Theorie!**
- **Churchsche These** (leider(??) nicht beweisbar)

Jede im intuitiven Sinne berechenbare Funktion ist Turing-berechenbar.

Turing-berechenbar besagt "durch eine Turing-Maschine berechenbar", ist äquivalent zu "mit eine Registermaschine berechenbar", ist äquivalent zu μ-rekursive Funktion, …

Interessante Aussagen über Algorithmen:

Das heißt aber auch (Umkehrschluss): *Was eine Turingmaschine nicht berechnen kann, kann auch kein Mensch berechnen!*

Oder ???

Alles das, was ein Mensch (berechnen) kann, kann auch ein Computer: Emotionen haben, Gefühle haben, leben (?), lieben(?), hassen(?)

Orwell lässt grüßen!

…

Wissenschaften und die Grenzen ihrer Erkenntnismethoden

- Die **Mathematik** gewinnt ihre Erkenntnisse mit der **beweisenden Methode**. Gödel konnte zeigen, dass diese Methode ihre Grenzen hat, wenn man sie auf die Mathematik als Ganzes anwendet.
- Turing und Church haben gezeigt, dass auch die algorithmische Methode (**Informatik**) ihre Grenzen hat. Es gibt Probleme, die sich nicht mit einem Algorithmus automatisiert lösen lassen.
- In der **Physik** gelangt man über Experimente zu Einsichten über die Zusammenhänge der uns umgebende Welt. Heisenberg zeigte eine fundamenale Schwäche der messenden Methode auf: Man kann bestimmte Größen nicht gleichzeitig beliebig genau messen.

Absolute Grenzen sind interessant, teilweise weit weg – bedeutender sind aber erreichbare Ziele:

- Grace et.al. befragten Experten, die an führenden Konferenzen im Bereich *Machine Learning* und *Neural Information Processing Systems* im Jahr 2015 Fachvorträge gehalten hatten.
- n = 352 (1634 angefragt) Grace et.al. berechneten Medianwerte gemäß der Erwartungen, wann "Künstliche Intelligenz", d.h. *Computer mit geeigneten Algorithmen dies besser können als der Mensch:*
  - Übersetzen 2024
  - Schreiben von Aufsätzen fürs Gymnasium 2026
  - Fahren von Lastwagen 2027
  - Tätigkeiten im Einzelhandel 2031
  - Bestseller-Buch schreiben 2049
  - als Chirurg arbeiten 2053

Katja Grace et.al.: *When Will AI Exceed Human Perfor-mance? Evidence from AI Experts, 30.05.17* https://arxiv.org/pdf/170508807.pdf , *zuletzt abgerufen am 28.08.2017*

Ein wenig durchatmen!

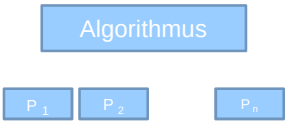
Computer – Algorithmus – Programm

✓ ✓ ⇧

Hier nur „Nebenthema“	hierzu gibt es noch	unser Kernthema
(Viel) mehr in Hardware 2	Einiges in PRG1-EPR	in PRG1/EPR/PS2

Die Themen "Die Vielgestalt moderner Computer" finden Sie in der eLecture-"Vertiefung"

Algorithmus vs. Programm



Algorithmen sind Abstraktionen von Programmen

ob sie jetzt auf eine spezielle Maschine zugeschnittenen sind oder auch mit einer höheren (maschinenunabhängigen) Programmiersprache formuliert sind (d. h., die Abstraktion erfolgt hier durch Weglassen der Details der realen Maschine, der realen Programmiersprache oder der Programmiererin)

das Programm ist eine konkrete Form des Algorithmus, angepasst an die Notwendigkeiten und Möglichkeiten der realen Maschine resp. der realen Programmiersprache

Programme werden in einer **Programmiersprache** formuliert.

Programmiersprachen

Programmiersprachen sind **formale Sprachen!**
(im Gegensatz zu natürlichen Sprachen)

zwei weiter Hauptkriterien :

- **das Paradigma**
- **die Generation**

und dann viele, viele Details:

Übrigens? Wie viele verschiedene Programmiersprachen mag es geben?

Programmiersprachen-Paradigmen (ganz grob)

- Paradigma bedeutet allgemein:
  Beispiel*, *Vorbild*, *Muster* oder *Abgrenzung

Paradigma		Unterformen	Beispiele
imperativ	Antwort auf das WIE	strukturiert prozedural modular objektorientiert	Pascal Modula **Java**, C++, **Python**
deklarativ	Antwort auf das WAS	constraint funktional logisch ==> Anwendung	Eiffel **Haskell** Prolog **SQL**
Sonstige		Datenstrom	

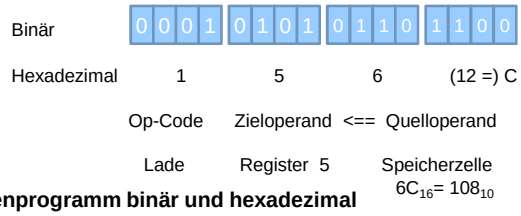
Zur Generation: Maschinensprachen sind Sprachen der ersten Generation

- Direkte Programmierung der Hardware durch
  - **Folge** elementarer im **Binärcode** repräsentierter Befehle eines bestimmten Prozessors (sehr uneinheitlich)
  - Diese Folge (das Programm) ist von der Hardware direkt ausführbar.

Was ist ein minimale Befehlssatz?

Beispiel Maschinensprache

16 Bit-Befehl: „Lade den Inhalt der Speicherzelle Nr. 108 ins Register Nr. 5“

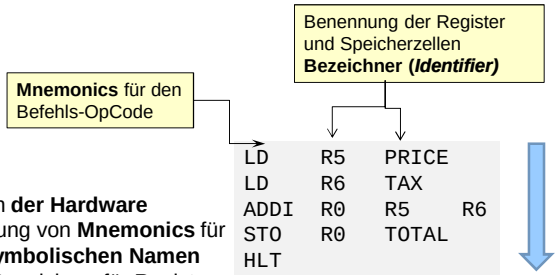


Ein MIPS (32 Bit)-Maschinenprogramm binär und hexadezimal

```
00100111101111010111111111100000    27BDFFE0
10101111101111110000000000010100    AF ...
10101111101001000000000000100000
10101111101001010000000000100100
```

So wollen wir **nicht** programmieren!

2. Generation : Assemblersprachen



Programmieren **der Hardware** unter Verwendung von **Mnemonics** für Befehle **und symbolischen Namen** für Adressen (Bezeichner für Register und Speicherzellen)

Programm in Assemblersprache

Auch noch zu mühsam!

Assembler: Programm, das ein Assemblerprogramm in ein Maschinenprogramm umsetzt. Vorgang: **assemblieren**

Assemblerprogramme

Vorteil: Ermöglichen die Erstellung sehr effizienter Programme
aber nur für einen speziellen Rechnertyp

Nachteile:

- abhängig vom konkreten Rechnertyp ➔ viel Portierarbeit
- Programme sind selbst für den Entwickler **schwer verständlich**
  ➔ kaum wartbar

nur noch sehr selten genutzt!

3. Generation: Höhere Programmiersprachen

- Hardware (Maschinen)-Unabhängigkeit des Source Codes
- Für den Menschen verständlicher und einfacher zu handhaben
- **Idee:** Die mühsame Codierarbeit (dafür brauchte man früher Programmierer) soll doch die Maschine selbst machen!

➔ spezielle Übersetzer-Programme (**Compiler** von *to compil*e = *erstellen, zusammenstellen* und **Interpreter**)

die Ersten schon Mitte der 50er Jahre:

FORTRAN (Formular Translation; Ausdrücke wie in der Mathematik)
COBOL
LISP

4. und 5. Generation

nicht wirklich wissenschaftlich, eher „Marketingbegriffe“

... bitte nicht ernsthaft benutzen!

Beispiele höherer Programmiersprachen (3. Generation)

FORTRAN
COBOL
LISP
Pascal

Python
C / C++
JAVA
Ruby

Haskell
PHP
Perl
…

Programmbeispiel in Python

```
price = float (input ("Bitte Preis eingeben:  "))
tax = 0.19 * price
total = price + tax
print ("Nettopreis:  ", price, \
"Steuer: ", tax, "Gesamtpreis:  ", total)
```

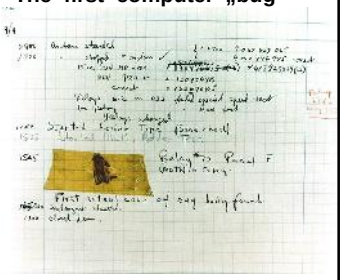
Abarbeitung

Wichtig: Alle Programmiersprachen sind gleich mächtig!

Grace Murray Hopper (1906 – 1992)
„Erfinderin des Compilers“ im Mark 1 Team (Aiken)

She did this - the invention of the compiler -, she said, because she was lazy and hoped that "the programmer may return to being a mathematician."

The first computer „bug“

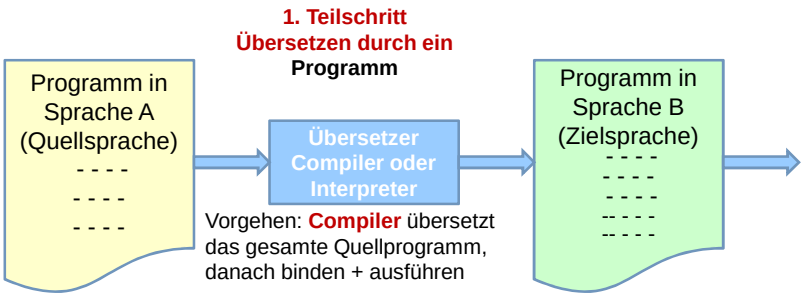


Hauptziel:
Maschinenunabhängiges Programmieren.

Charakteristika von Programmiersprachen
(= formale Sprachen - im Gegensatz zu natürlichen Sprachen)

- **eindeutige Lexikalität**: Festlegung der gültige Zeichen bzw. Wörter (= Symbole/*Token*) aus denen Programme zusammengesetzt werden.
- **eindeutige Syntax**: legt fest, ob eine Folge von Token ein gültiger „Satz“ ist (syntaktisch korrekt ist).
- **eindeutige Semantik**: welche Auswirkungen hat die Ausführung eines Satzes (Anweisung, *statement*) ?
- (Pragmatik definiert den Einsatzbereich der Sprache)

Übersetzer (ab) 3. Generation: Compiler oder Interpreter



Vorgehen: **Compiler** übersetzt das gesamte Quellprogramm, danach binden + ausführen

Quellprogramm ***(Source)***

Interpreter macht alle Teilschritte (übersetzen, binden, ausführen) nur für die nächste (vollständige) Anweisung (Statement).

Zielprogramm, z.B. in Maschinen-sprache (kann auch Assembler oder andere Hochsprache sein)

Beispiel: Prinzip der Übersetzung und des Bindens: ➔ Compiler 1

Quellcode Hochsprache ➔ **Symbolischer Assemblercode** + Bindeanweisungen ➔ **Objektcode (binden) Maschinencode**

```
a = b + c
import (math)
d = cos (a)
```

```
// R 2 Parameterübergabe
LD    R5   b
LD    R6   c
ADDI  R0   R5   R6
STO   R0   a
IMPRT math
MOV   R2   R0
CALLI cos  R2
STO   R2   d
```

Symbolische Adessen:

```
// b in Speicherzelle 100
// c in Speicherzelle 101
// a in Speicherzelle 102
// d in Speicherzelle 103
```

```
0001 0101 0110 0100
0001 0110 0110 0101
...
```

Funktion „binden“

- Viele häufig benötigte Programmteile, z.B. mathematische Funktionen (wie Kosinus im Beispiel) , Betriebssystemfunktionen, Textmanipulationen, etc. sind in sogenannten **„Bibliotheken“** gesammelt und bereitgestellt.
- Diese Programmteile sind fertig „ausprogrammiert“ und schon übersetzt in Bibliotheken vorhanden.
- Kann auch für selbstprogrammierte Programmteile genutzt werden.
- Die hier bereitgestellten „Funktionen (Programmteile)“ und deren Namen müssen dem Übersetzer bekannt gemacht werden ➔ **import**
- Sie müssen dann nur noch geladen und zu dem individuellen Programm hinzugebunden werden ➔ **Binder**

Beispiel: Prinzip der Übersetzung: ➔ Compiler 2 (ohne Assembler, ohne Zwischencode)

Quellcode ➔ Hochsprache

```
a = b + c
import (math)
d = cos (a)
```

Objektcode ➔(binden) Maschinencode

```
// b in Speicherzelle 100
// c in Speicherzelle 101
// a in Speicherzelle 102
// d in Speicherzelle 103
// R 2 Parameterübergabe

0001 0101 0110 0100
0001 0110 0110 0101
...

IMPRT math
0010 0010 0000 0000
CALLI cos  R2
0011 0010 0110 0111
```

Beispiel: Prinzip der Übersetzung: Interpreter

(Statements werden in „Unterprogramm-Aufrufe“ im Interpreter abgebildet, (Teil-)Statements sind dort in der Quellsprache des Interpreter programmiert und werden direkt ausgeführt)

Quellcode ➔

```
// b in Speicherzelle 100
// c in Speicherzelle 101
// a in Speicherzelle 102
// d in Speicherzelle 103
// R 2 Parameterübergabe
```

```
a = b + c
import (math)
d = cos (a)
```

```
Call Add (100,101,102)

IMPORT math: cos

CALL cos (102,103)
```

ausführen, d.h.
in Speicherzelle 102
steht danach der Wert b+c

ausführen, d.h.
in Speicherzelle 103
steht danach der Wert cos (102)

Übersetzungsphasen im Compiler (Interpreter)

Lexikalische Analyse	Quellprogramm wird in eine Folge von „Token (=Wörter)“ zerlegt (Blanks, Leerzeilen, Kommentare entfernen).
Syntaktische Analyse	Überprüft, ob das Quellprogramm der Syntax (Grammatik) der Quellsprache entspricht: „Fügt Wörter zu Sätzen zusammen: „*Statements*“.
Semantische Analyse	Überprüft z.B., ob die Typregeln eingehalten wurden: Ein Text kann nicht ohne Weiteres einer Zahl zugewiesen werden.
Optimierung und (ggf. Zwischencodeerzeugung)	Befehlsreihenfolgen vertauschen, Parallelitäten nutzen, Vorausberechnungen, Dead Code Elimination**,** Inlining, etc. **(keine globale Optimierung beim Interpreter)**
Code Generierung	Zielprogramm erzeugen

Compiler

Analysiert das vollständige Quellprogramm und erzeugt Assembler oder Objektcode der Zielmaschine (in der Regel als Dateien)

Assemblieren und „Binden und Laden“ erzeugt dann ausführbaren Maschinencode“

Unterscheidbar: Übersetzungszeit und Laufzeit der Programme (➔ weniger Laufzeitfehler)

Globale Optimierung: Befehlsreihenfolgen vertauschen, Parallelitäten nutzen, Vorausberechnungen, Dead Code Elimination, Inlining, etc.

➔ **Maximale Performance möglich**

Interpreter (simpel)

Analysiert das Programm Statement für Statement und übersetzt jedes Statement in einen entsprechenden Unterprogrammauf (und führt diesen dann aus).

Muss dieselbe Quellzeile ggf. mehrfach analysieren.

Laufzeit und Übersetzungszeit sind vermischt ➔ „nur“ Laufzeitfehler

Globale Optimierungen **nicht** möglich!

➔ **Geringere Performance aber gute Eigenschaften während der Entwicklung**

Beispiel für beschränkte Performance (sehr simpler Interpreter)

Das relativ aufwendige Analysieren (lexikalisch, syntaktisch, semantisch) des Quellprogramms muss in diesem Beispiel für den Schleifenrumpf bei jedem Schleifendurchlauf durchgeführt werden.

Python Programbeispiel:

```
x = 0
while x < 6:
    print (x)
    x = x + 1
```

Ausgabe

```
>>>
0
1
2
3
4
5
>>>
```

Eigenschaften von kompilierten Programmen

- maximale Geschwindigkeit durch Optimierung des Codes erreichbar
  aber: das **übersetzte** Programm läuft nur auf dem jeweiligen Maschinentyp; ggf. nur neu übersetzen

aber, weil „Dienste des Betriebssystems“ benötigt werden, sind die Programme trotzdem **Plattform-abhängig.**

das Resultat: **viele** Dialekte einer Sprache und **viele** Compiler-Versionen.

> *„The difference between theory and practice is that in theory, there is no difference between theory and practice, but in practice there is.“*

Zusammenfassung: Mit dem Compiler arbeiten

1. **Übersetzt** das Quellprogramm „als Ganzes“
   kann dann viel optimieren, d.h. Befehlsreihenfolgen vertauschen, Parallelitäten nutzen, Vorausberechnungen machen, etc. Wird während der Entwicklung oft nicht genutzt, wegen der Zeit und des „Debugging“

2. Danach **binden,** d.h. „vorübersetzte“ Funktionen hinzufügen und die „Namen“ (Bezeichner) durch Adressen ersetzen
   entweder statisch oder dynamisch (DLL),

3. **Laufen lassen** und zurück zu 1.

Lange „turn-around“ Zeiten bei der Programmentwicklung.

Ein alternatives Vorgehen: Interpreter

Ein Interpreter führt das Quellprogramm Zeile für Zeile nach der Übersetzung direkt aus:

Arbeitsweise:

1. Lexikalische Analyse **nur einer Anweisung** im Programm
2. Syntaktische Analyse
3. Übersetzung der Hochsprache in eine Befehlsfolge der Maschinensprache oder einer Zwischensprache; dynamisch binden
4. **Ausführung der generierten Befehlsfolge**

   Wiederholung der Schritte 1 bis 4 (nächstes Statement)

Vor (und Nachteile) eines Interpreters

- schnelles Ändern, auch ausprobieren gut möglich (!?!)
- sehr gut für Prototypen, kleine Programme
- geänderte Programme sind sofort ausführbar (kein langes übersetzen und binden)
- sehr hilfreich beim Programmieren und Testen
- **Ein wichtiger Grund dafür, dass wir Python benutzen!**
- **Aber:** die Ausführungszeiten sind gegenüber kompilierten Programmen größer.

Bisher nur die klassischen „Reinformen“ betrachtet

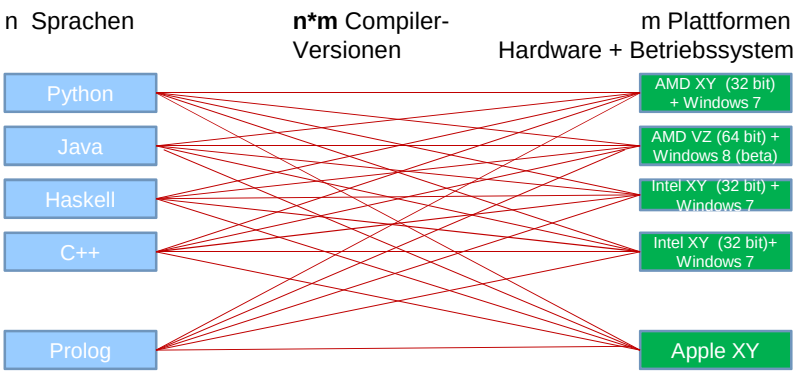
Heutige Compiler und Interpreter sind fast immer Hybridformen

Zwei Haupt-Maßnahmen um die jeweiligen Probleme **zu lindern**:

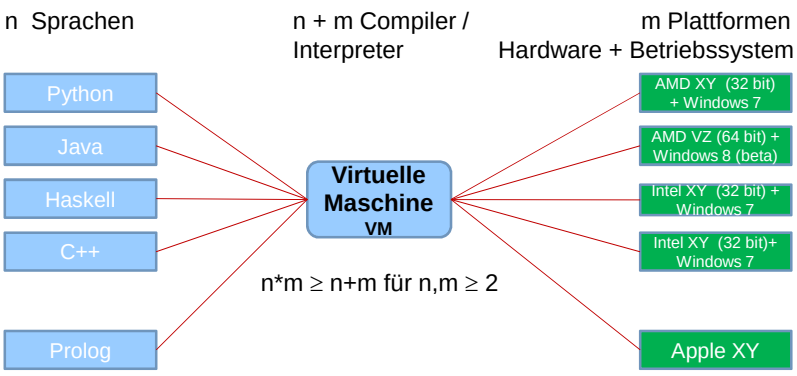
Virtuelle Maschinen mit sogenannten plattformunabhängigen **„Zwischencode“,** der Maschinencode der virtuellen Maschine

JIT (Just In Time) – Compiler (seit ca. 1990)

Sprachen und Versionenproblem bei Compilern



Lösungsidee:
Sprachen und Versionenproblem



n*m ≥ n+m für n,m ≥ 2

Vorgehen

- Der Quellcode (z.B. Python) wird in einen **Bytecode (Zwischencode)** übersetzt und oft abgespeichert. Beispiele: **.pyc** in Python, **.class** in Java
  Der Python-Interpreter prüft dann, ob ein .py-File neu ist.
- Bytecode ist kompakter als Quellcode aber **noch maschinenunabhängig** und häufig in 8 Bit codiert (**deshalb Bytecode** genannt (8 Bit = 1Byte))
- Bytecode ist von der Struktur her einer „Assemblersprache“ sehr ähnlich: eben die **„Assemblersprache“ einer virtuellen Maschine**,
- Bytecode wird von der Virtuellen Maschine meist
  - **interpretiert** (Bytecode-Befehl für Befehl, so bei Python und auch in der traditionellen JAVA Virtual Machine) oder
  - J**IT-Compiliert**, d.h insbesondere für zeitkritische Teile des Codes Maschinencode erzeugt.

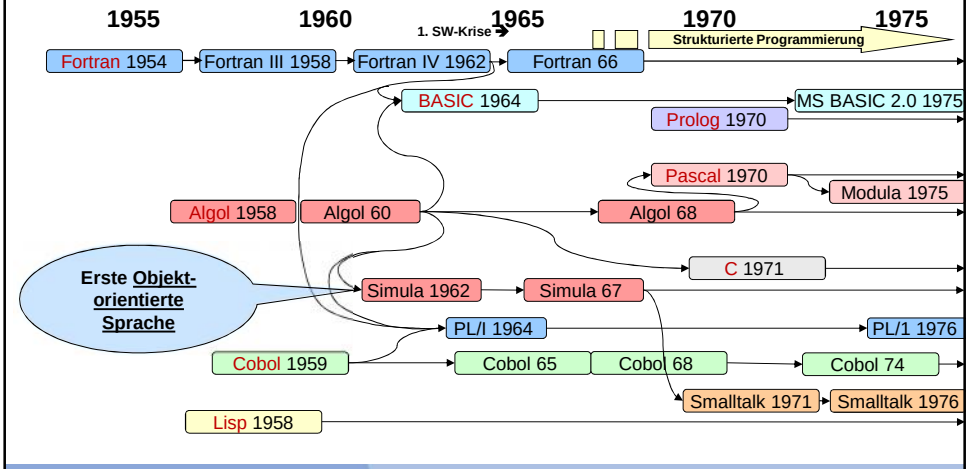
Vor- und Nachteile

- Nur ein Compiler/Interpreter für jede Sprache und allen Prozessoren und Plattformen
- (natürlich braucht man eine VM pro Plattform, oft ein Interpreter)
- Bytecode-Programme laufen auf allen Prozessoren und Plattformen
- (etwas) langsamer als Maschinencode ➔ z.B. JIT Compiler

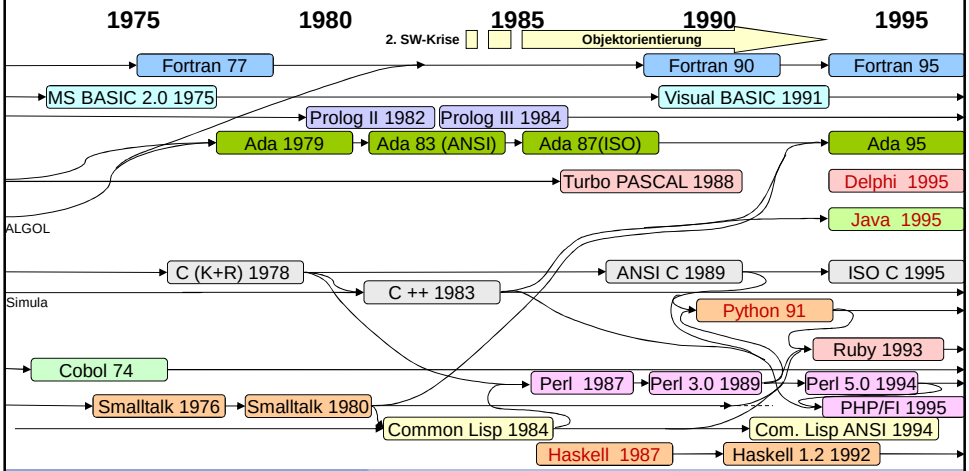
Zusammenfassung

- **Zu unterscheiden**:
  Paradigmen und Generationen
  (in der Realität nicht ganz unabhängig)
- An Übersetzungstechniken arbeiten InformatikerInnen seit den 50er Jahren: ➔ viele Varianten – Hauptunterscheidungen
  - Compiler vs. Interpreter
  - mit Byte-Code oder ohne
  - Just in time Compiler oder nicht
- Insgesamt sehr viele verschiedene Programmiersprachen

Anhang: Ideengeschichte der Programmiersprachen

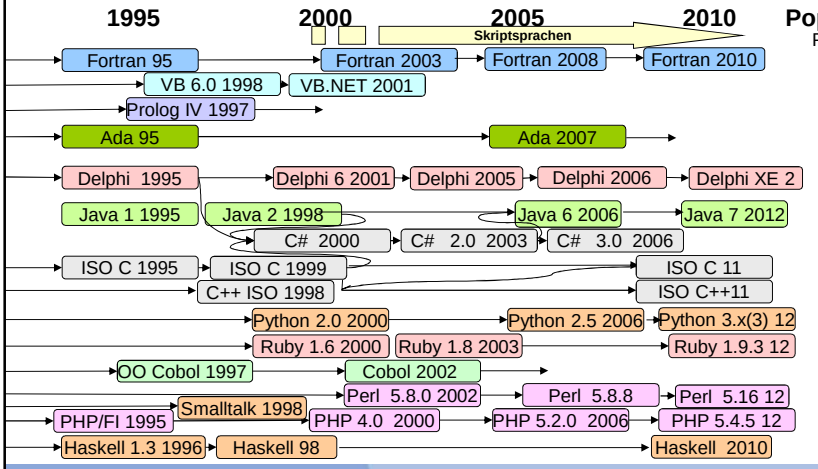


Entwicklung der höheren Programmiersprachen



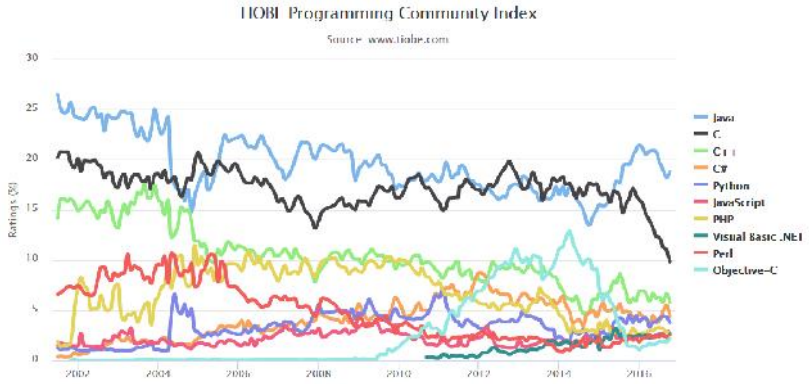
Entwicklung der höheren Programmiersprachen

nach Tiobe-Index



Popularität	Oct/1
Platz	in %
28	0,7
8	2,7
36	0,5
31	0,7
12	1,6
1	**18,8**
4	4,4
2	9,8
3	5,8
5	**3,8**
13	0,2
24	1,1
9	2,5
7	2,7
34	**0,5**

Zeitliche Entwicklung



Geschafft! – Das war echt viel!

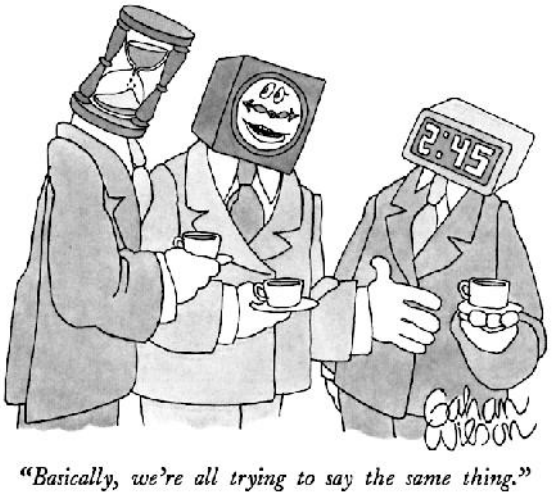
Computer – Algorithmus – Programm

✓ ✓ ✓

Fragen und (hoffentlich) Antworten

Stellen Sie bitte in der nächsten Vorlesung!

Danke für Ihre Aufmerksamkeit



Gahan Wilson: „Basically, We're All Trying To Say The Same Thing", New Yorker Cartoons, *unter:* https://condenaststore.com/featured/basically-were-all-trying-to-say-the-same-thing-gahan-wilson.html; *(abgerufen am 16.07.2017*

Ausblick

- Also, nicht vergessen:
  - Das Reading "Ada Lovelace …" lesen
  - Ergänzung "Die Vielgestalt des Computers" und
  - Ergänzung "Programm" anschauen
  - Aufgaben für PRG1 und EPR anschauen, um ggf. im nächsten Tutorium Fragen stellen zu können
  - Das Quizz "Computer – Algorithmen – Programm" durchführen
- Am kommenden Montag geht's ans Programmieren … **in Python**

… und danke für Ihre Aufmerksamkeit