



Einführung in die Praktische Informatik

Prof. Björn Ommer HCI, IWR
Computer Vision Group



Grundbegriffe

- Formale Systeme & Sprachen
- Bäume und Graphen
- Automatentheorie, Turingmaschine

- Algorithmen
- Berechenbarkeit
- Reale Computer

- Programmiersprachen
- Grenzen der Harwareentwicklung



Formale Systeme

Im folgenden betrachten wir Zeichenketten über einem Alphabet.

Ein **Alphabet** \mathcal{A} ist eine endliche, nichtleere Menge (manchmal verlangt man zusätzlich, dass die Menge geordnet ist). Die Elemente von \mathcal{A} nennen wir Zeichen (oder Symbole).

Eine endliche Folge nicht notwendigerweise verschiedener Zeichen aus \mathcal{A} nennt man ein **Wort**. Das **leere Wort** ϵ besteht aus keinem einzigen Zeichen. Es ist ein Symbol für „Nichts“.

Die Menge aller möglichen Wörter inklusive dem leeren Wort wird als **freies Monoid** \mathcal{A}^* bezeichnet.

Beispiel: $\{0, 1\}^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \dots\}$

Formale Systeme dienen der Beschreibung interessanter Teilmengen von \mathcal{A}^* .

Formale Systeme

Definition: Ein **formales System** ist ein System von Wörtern und Regeln. Die Regeln sind Vorschriften für die Umwandlung eines Wortes in ein anderes.

Mathematisch: $F = (\mathcal{A}, \mathcal{B}, \mathcal{X}, \mathcal{R})$, wobei

- \mathcal{A} das **Alphabet**,
- $\mathcal{B} \subseteq \mathcal{A}^*$ die Menge der **wohlgebildeten Worte**,
- $\mathcal{X} \subset \mathcal{B}$ die Menge der **Axiome** und
- \mathcal{R} die Menge der **Produktionsregeln**

sind. Ausgehend von \mathcal{X} werden durch Anwendung von Regeln aus \mathcal{R} alle wohlgeformten Wörter \mathcal{B} erzeugt.

Das MIU-System [Hofstadter, 2007]

Das MIU-System handelt von Wörtern, die nur aus den drei Buchstaben M, I, und U bestehen.

- $\mathcal{A}_{\text{MIU}} = \{\text{M}, \text{I}, \text{U}\}$.
- $\mathcal{X}_{\text{MIU}} = \{\text{MI}\}$.
- \mathcal{R}_{MIU} enthält die Regeln:
 1. $\text{MxI} \rightarrow \text{MxIU}$. Hierbei ist $x \in \mathcal{A}_{\text{MIU}}^*$ irgendein Wort oder ϵ .
Beispiel: $\text{MI} \rightarrow \text{MIU}$. Man sagt **MIU wird aus MI abgeleitet**.
 2. $\text{Mx} \rightarrow \text{Mxx}$.
Beispiele: $\text{MI} \rightarrow \text{MII}$, MIUUI \rightarrow MIUUIIUUI.
 3. $xIIIy \rightarrow xUy$ ($x, y \in \mathcal{A}_{\text{MIU}}^*$).
Beispiele: $\text{MIII} \rightarrow \text{MU}$, $\text{UIIIIM} \rightarrow \text{UUIM}$, $\text{UIIIIM} \rightarrow \text{UIUM}$.
 4. $xUUy \rightarrow xy$.
Beispiele: $\text{UUU} \rightarrow \text{U}$, $\text{MUUUIII} \rightarrow \text{MUIII}$.

Das MIU-System

- \mathcal{B}_{MIU} sind dann alle Worte die ausgehend von den Elementen von \mathcal{X} mithilfe der Regeln aus \mathcal{R} erzeugt werden können, also

$$\mathcal{B} = \{\text{MI}, \text{MIU}, \text{MIUUI}, \dots\}.$$

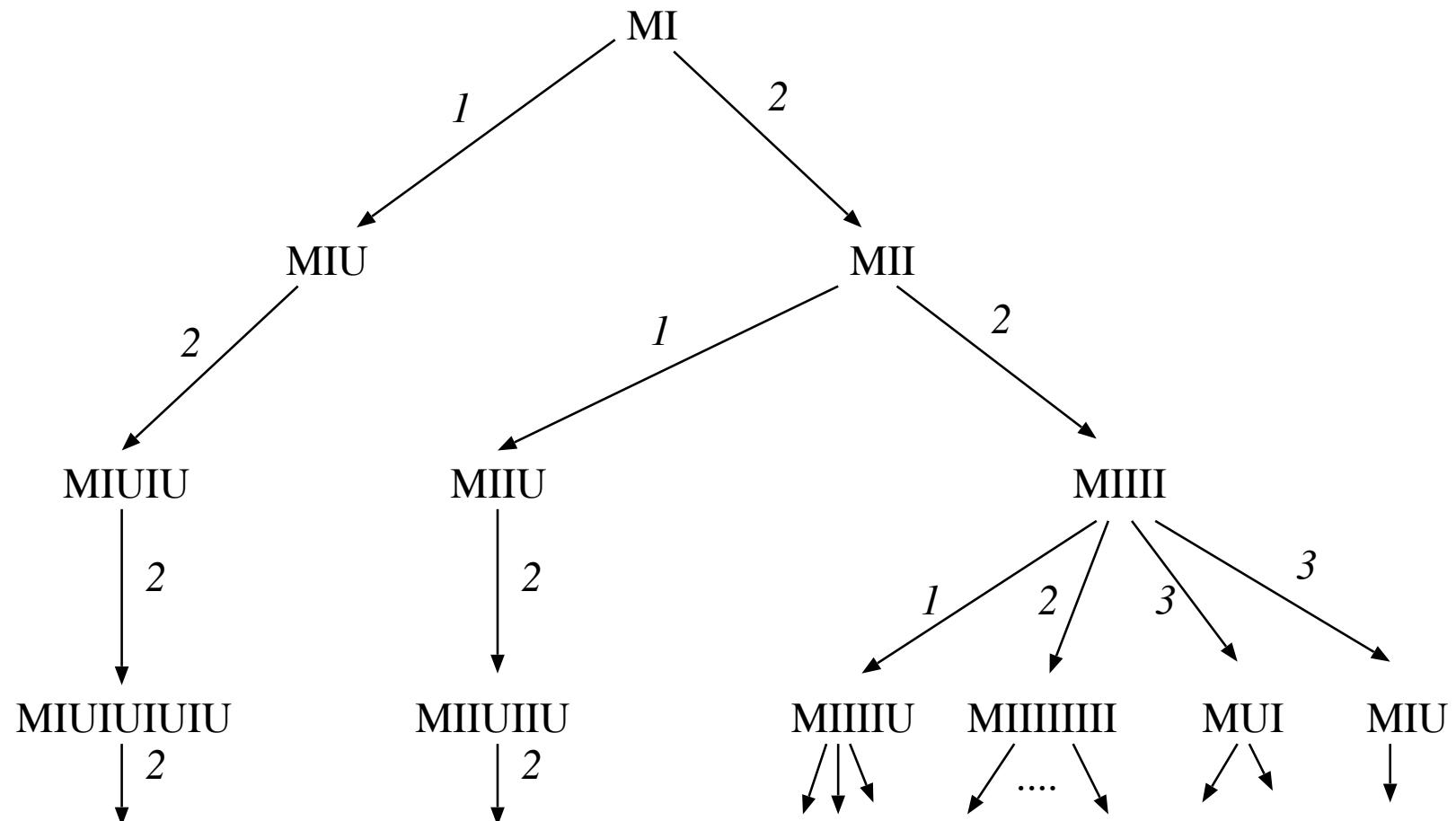
Beobachtung: \mathcal{B}_{MIU} enthält offenbar unendlich viele Worte.

Problem: (MU-Rätsel) Ist MU ein Wort des MIU-Systems?

Oder mathematisch: $\text{MU} \in \mathcal{B}_{\text{MIU}}$?

Systematische Erzeugung aller Worte des MIU-Systems

Dies führt auf folgende Baumstruktur:



Systematische Erzeugung aller Worte des MIU-Systems

Beschreibung: Ganz oben steht das Anfangswort MI. Auf MI sind nur die Regeln 1 und 2 anwendbar. Die damit erzeugten Wörter stehen in der zweiten Zeile. Ein Pfeil bedeutet, dass ein Wort aus dem anderen ableitbar ist. Die Zahl an dem Pfeil ist die Nummer der angewendeten Regel. In der dritten Zeile stehen alle Wörter, die durch Anwendung von zwei Regeln erzeugt werden können, usw.

Bemerkung: Wenn man den Baum in dieser Reihenfolge durchgeht (**Breitendurchlauf**), so erzeugt man nach und nach alle Wörter des MIU-Systems.

Systematische Erzeugung aller Worte des MIU-Systems

Beschreibung: Ganz oben steht das Anfangswort MI. Auf MI sind nur die Regeln 1 und 2 anwendbar. Die damit erzeugten Wörter stehen in der zweiten Zeile. Ein Pfeil bedeutet, dass ein Wort aus dem anderen ableitbar ist. Die Zahl an dem Pfeil ist die Nummer der angewendeten Regel. In der dritten Zeile stehen alle Wörter, die durch Anwendung von zwei Regeln erzeugt werden können, usw.

Bemerkung: Wenn man den Baum in dieser Reihenfolge durchgeht (**Breitendurchlauf**), so erzeugt man nach und nach alle Wörter des MIU-Systems.

Folgerung: Falls $MU \in \mathcal{B}_{\text{MIU}}$, wird dieses Verfahren in endlicher Zeit die Antwort liefern. Wenn dagegen $MU \notin \mathcal{B}_{\text{MIU}}$, so werden wir es mit obigem Verfahren nie erfahren!

Sprechweise: Man sagt: Die Menge \mathcal{B}_{MIU} ist **rekursiv aufzählbar**.

Frage: Wie löst man nun das MU-Rätsel?

Lösung des MU-Rätsels

Zur Lösung muss man Eigenschaften der Wörter in \mathcal{B}_{MIU} analysieren.

Beobachtung: Alle Ketten haben immer M vorne. Auch gibt es nur dieses eine M, das man genausogut hätte weglassen können.

Beobachtung: Die Zahl der I in einzelnen Worten ist niemals ein Vielfaches von 3, also auch nicht 0.

Beweis: Ersieht man leicht aus den Regeln, sei $\text{anzahli}(n)$ die Anzahl der I nach Anwendung von n Regeln, $n \in \mathbb{N}_0$. Dann gilt:

$$\text{anzahli}(n) = \begin{cases} 1 & n = 0, \text{Axiom}, \\ \text{anzahli}(n - 1) & n > 0, \text{Regel 1, 4}, \\ \text{anzahli}(n - 1) \cdot 2 & n > 0, \text{Regel 2}, \\ \text{anzahli}(n - 1) - 3 & n > 0, \text{Regel 3} \end{cases}$$

Ist $\text{anzahli}(n - 1) \bmod 3 \neq 0$, so gilt dies auch nach Anwendung einer beliebigen Regel.

Grundbegriffe

- **Formale Systeme & Sprachen**
- **Bäume und Graphen**
- **Automatentheorie, Turingmaschine**

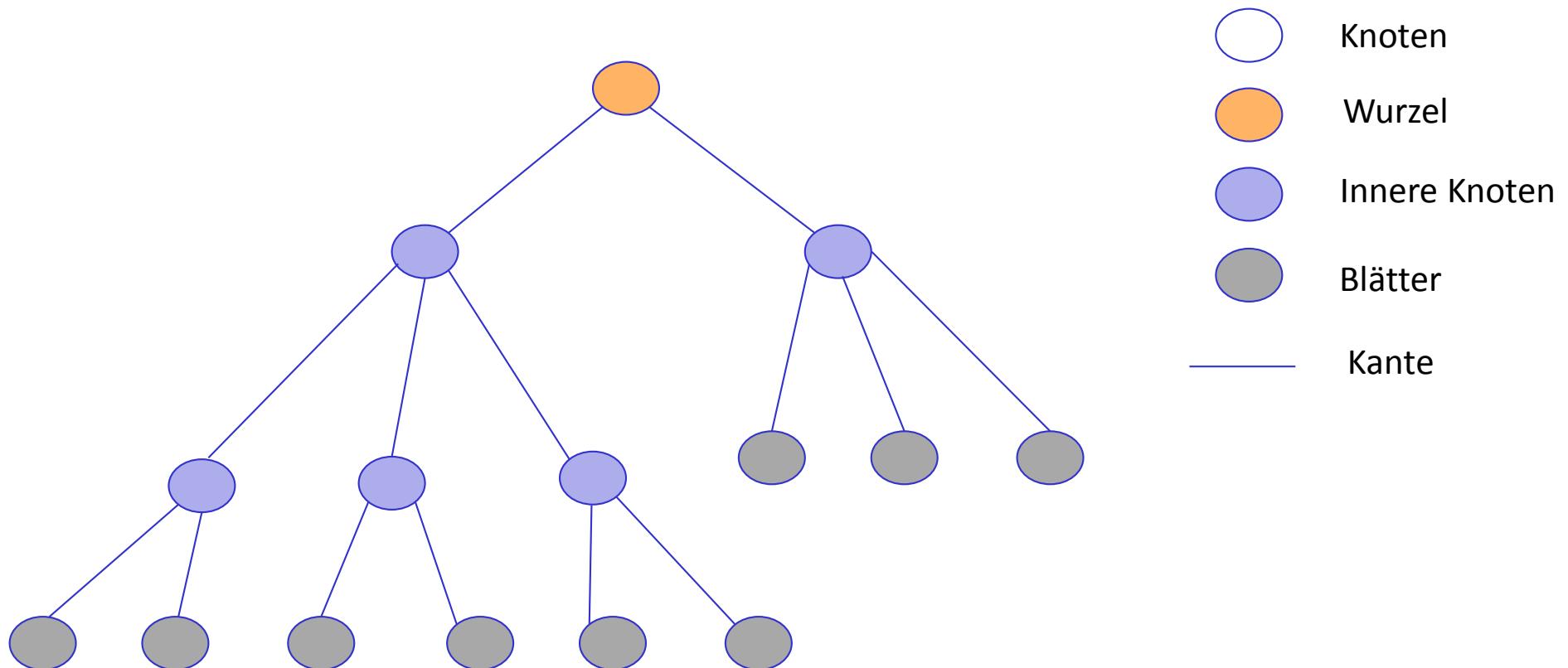
- **Algorithmen**
- **Berechenbarkeit**
- **Reale Computer**

- **Programmiersprachen**
- **Grenzen der Harwareentwicklung**



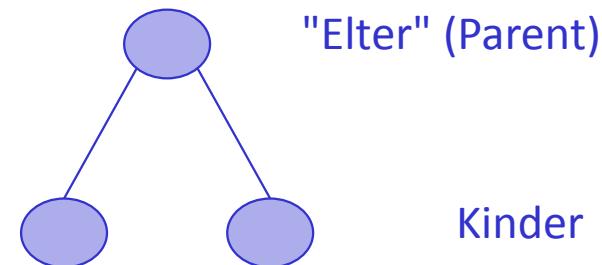
Bäume

- Zur Analyse der formalen Sprache MIU und zur Lösung des MU-Rätsels haben wir ein Baum-Diagramm benutzt

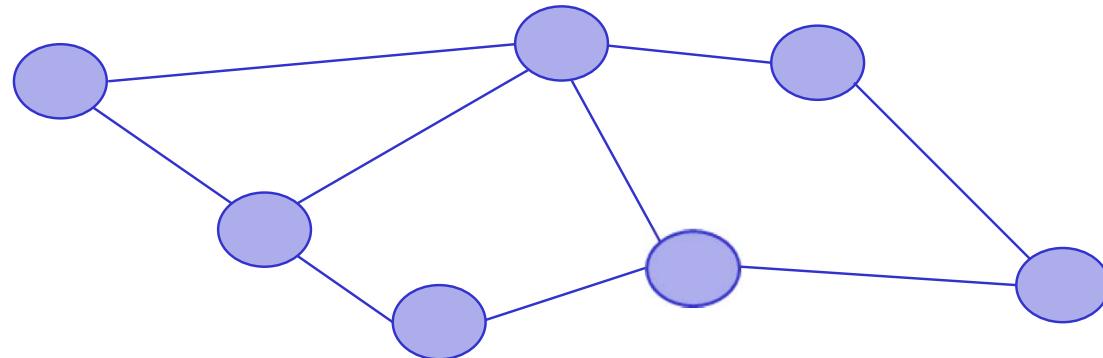


Bäume

- Jeder Baum besitzt genau einen Knoten, der keine eingehenden Kanten hat. Diese heißt Wurzel
- Knoten ohne ausgehende Kanten heißen Blätter, alle anderen Knoten heißen Innerer Knoten (je nach Definition wird die Wurzel dazugezählt oder nicht).
- Ein Baum, bei dem alle inneren Knoten (und die Wurzel) höchstens zwei Kinder haben, heißt Binärbaum.
- Ein Baum ist verbunden. Es gibt genau einen Weg von der Wurzel zu jedem Blatt
- Ein Baum ist eine spezielle Form eines Graphen (s. nächste Folie)
- Bezeichnung der Relationen im Baum:



Graphen



Definition: Ein Graph $G = (V, E)$ besteht aus

- einer nichtleeren Menge V , der sogenannten Menge der **Knoten**, sowie
- der Menge der **Kanten** $E \subseteq V \times V$.

$V \times V = \{(v, w) : v, w \in V\}$ bezeichnet das **kartesische Produkt**.

Teilmengen von $V \times V$ bezeichnet man auch als **Relationen**.

Typen von Graphen

- **Ungerichteter Graph:**

Ein Graph ist ungerichtet, wenn $(v, w) \in E \Rightarrow (w, v) \in E$

Sonst heißt der Graph **gerichtet**.

- **Symmetrischer Graph:**

Ein gerichteter Graph G heißt symmetrisch, falls G zu jeder Kante auch die entsprechende invertierte Kante enthält. Ein ungerichteter Graph lässt sich einfach in einen symmetrischen gerichteten Graphen umwandeln, indem man jede Kante (v, w) durch die zwei gerichteten Kanten (v, w) und (w, v) ersetzt.

- **Verbundener Graph:**

Ein ungerichteter Graph heißt verbunden, falls jeder Knoten von jedem anderen Knoten über eine Folge von Kanten erreichbar ist. Bei einem gerichteten Graphen ergänze erst alle Kanten der Gegenrichtung und wende dann die Definition an.

- **Zyklischer Graph:**

Es gibt, ausgehend von einem Knoten, eine Folge von Kanten mit der man wieder beim Ausgangsknoten landet.

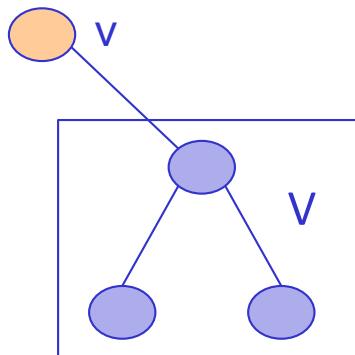
- **Endlicher Graph:**

die Menge der Knoten ist endlich. Sonst spricht man von einem unendlichen Graphen.

Nochmal Bäume

- Ein Baum ist ein zyklenfreier, verbundener Graph.
- Es gibt gerichtete und ungerichtete Bäume
- andere Definition (rekursiv):
Wir definieren die Menge der Bäume rekursiv über die Anzahl der Knoten
 - $(\{v\}, \emptyset)$ ist ein Baum.
 - Sei $B = (V, E)$ ein Baum, so ist $B' = (V', E')$ ebenfalls ein Baum, wenn

$$V' = V \cup \{v\}, \quad v \notin V, \quad E' = E \cup \{(w, v) : w \in V\}.$$



Grundbegriffe

- **Formale Systeme & Sprachen**
- **Bäume und Graphen**
- **Automatentheorie, Turingmaschine**

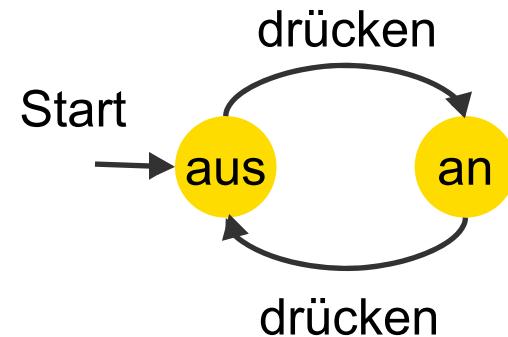
- **Algorithmen**
- **Berechenbarkeit**
- **Reale Computer**

- **Programmiersprachen**
- **Grenzen der Harwareentwicklung**



Endliche Automaten: Beispiel Kippschalter und Lexikon

Schalter



drücken	
aus	an
an	aus

Informelle Einführung: Abstrakter Automat

- ein **abstrakter Automat** ist ein **mathematisches Modell** für einfache Maschinen/Programme, die bestimmte Probleme lösen
- beschreibt nicht einen bestimmten Automaten, sondern gemeinsame **Grundprinzipien** einer **Klasse** von Automaten
- **Grundlegende Komponenten**
 - Zustände
 - Eingabesymbole
 - Zustandsübergänge: reagiert auf Eingaben durch Übergang in einen anderen Zustand



Determinierter abstrakter Automat

Mengentheoretische Definition

determinierter abstrakter Automat

$A = (X, Y, Z, \delta, \lambda, z_0)$ heißt determinierter abstrakter Automat,
falls

- a) X, Y, Z beliebige nichtleere Mengen sind
- b) Startzustand $z_0 \in Z$
- c) Zustandsübergangsfkt $\delta: Z \times X \rightarrow Z$
- d) Ausgabefkt $\lambda: Z \times X \rightarrow Y$

oder c) & d) zusammengefasst: $\gamma: Z \times X \rightarrow Y \times Z$

Interpretation

X Menge der Eingabesymbole

Y Menge der Ausgabesymbole

Z Menge der Zustände

Endlicher Automat

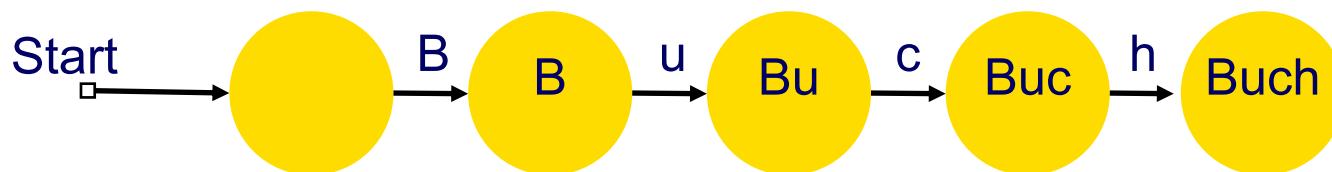
Mengentheoretische Definition

endlicher determinierter Automat

Ein determinierter Automat $A = [X, Y, Z, \delta, \lambda, z_0]$ heißt **X-endlich**, **Y-endlich** bzw. **Z-endlich** bzw. **(X,Y)-endlich** usw., wenn die jeweils angegebenen Mengen endlich sind. **(X,Y,Z)-endliche** Automaten bezeichnen wir schlechthin als **endlich** ■

Endliche Automaten haben kein Gedächtnis

- Mengen der Zustände, der Eingabesignale, der Ausgabesignale sind endlich
- kein Gedächtnis zur Speicherung durchlaufener Zustände:
 - Übergang von Zustand zur Zeit t in Zustand zur Zeit $t+1$ nur abhängig von
 - Zustand zur Zeit t und
 - Eingabe im Zustand zur Zeit t
 - Vorhergehende Zustände nur dadurch wirksam, dass
 - sie über eine bestimmte Eingabe in den aktuellen Zustand geführt haben, und
 - dieser aktuelle Zustand ein bestimmtes Ergebnis repräsentiert.



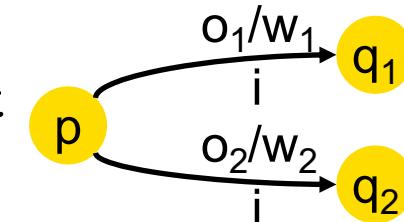
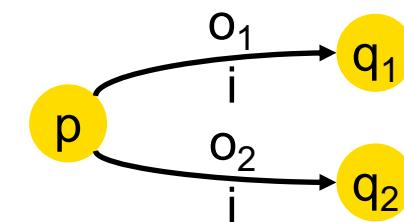
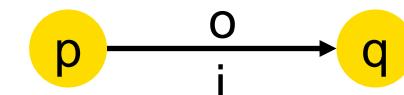
Typen endlicher Autotmaten

- **Akzeptoren**
 - Automaten ohne Ausgabe
 - **Transduktoren**
 - Automaten mit Ausgabe
-

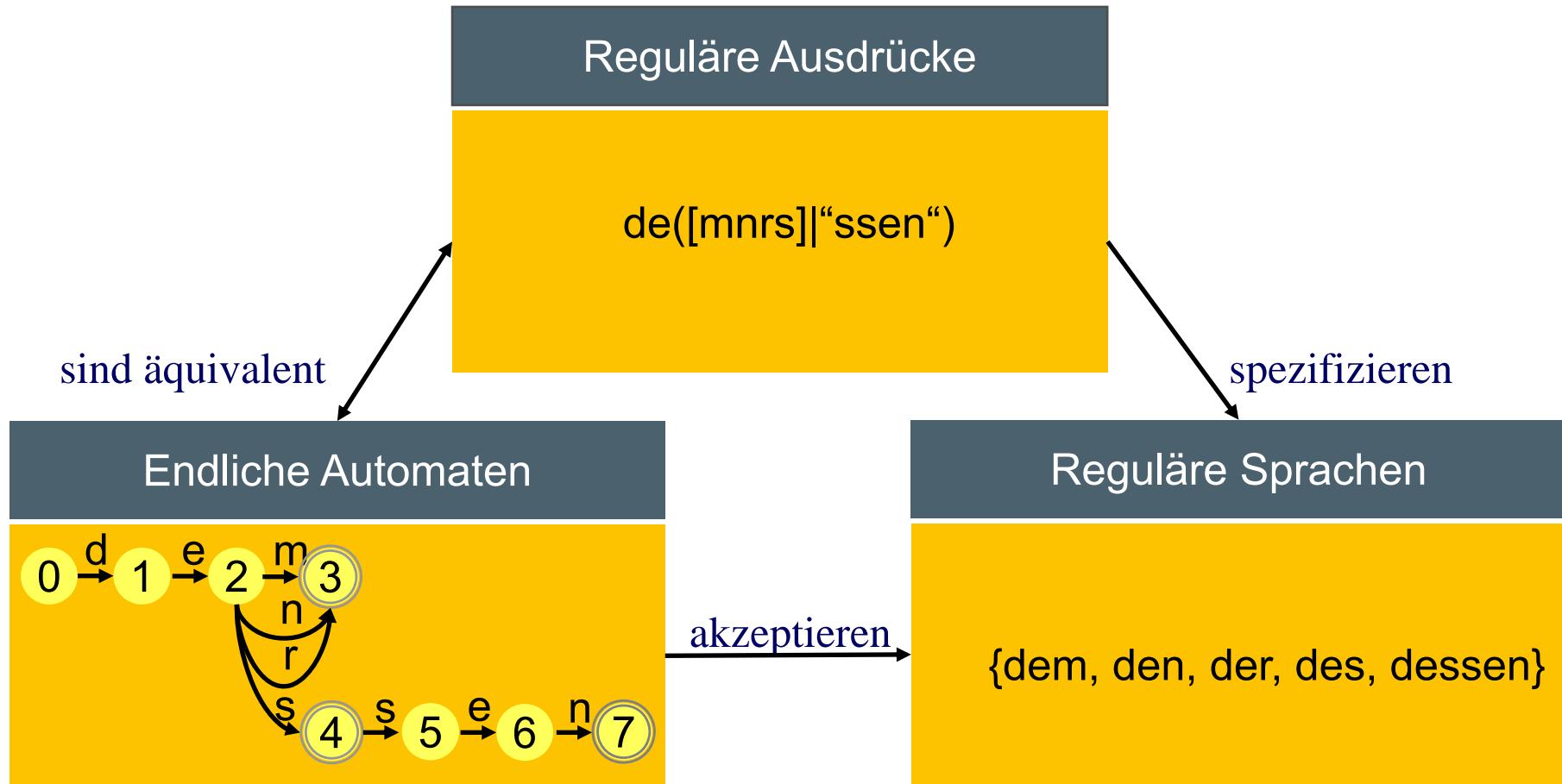
$$p \xrightarrow{i} q$$

$$p \xrightarrow{i : o} q$$

- **deterministisch**
 - jedem Paar $[p,i]$ ist ein Paar $[o,q]$ eindeutig zugeordnet
- **nicht-deterministisch**
 - einem Paar $[p,i]$ können mehrere mögliche Paare $[o,q]$ zugeordnet sein
- **stochastisch**
 - jedem Paar $[p,i]$ ist für ein Paar $[o,q]$ ein Wahrscheinlichkeitsmaß zugeordnet



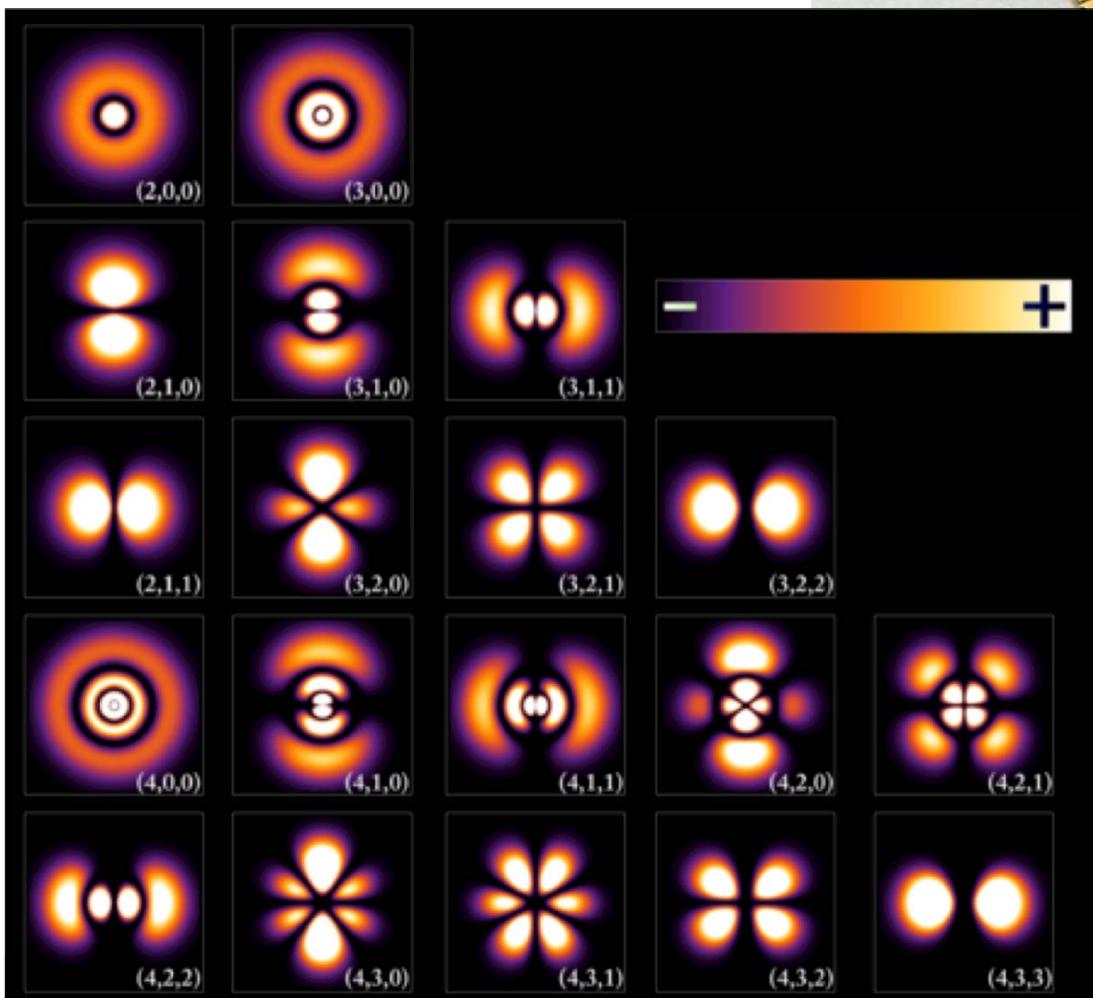
Äquivalenzen: Endliche Automaten, reguläre Sprachen, reguläre Ausdrücke



Automatentheorie

- klassifiziert Algorithmen nach der Art des Speichers, der für die Implementierung zum **Merken von Zwischenergebnissen gebraucht wird**

Spezialisierungen ↓	Automat	Speicher
	Turingmaschine	unendlich großer Speicher
	linear beschränkter Automat	endlich großer Speicher
	Kellerautomat (Push Down Automaton)	Kellerspeicher (Stack)
	endlicher Automat	kein zusätzlicher Speicher



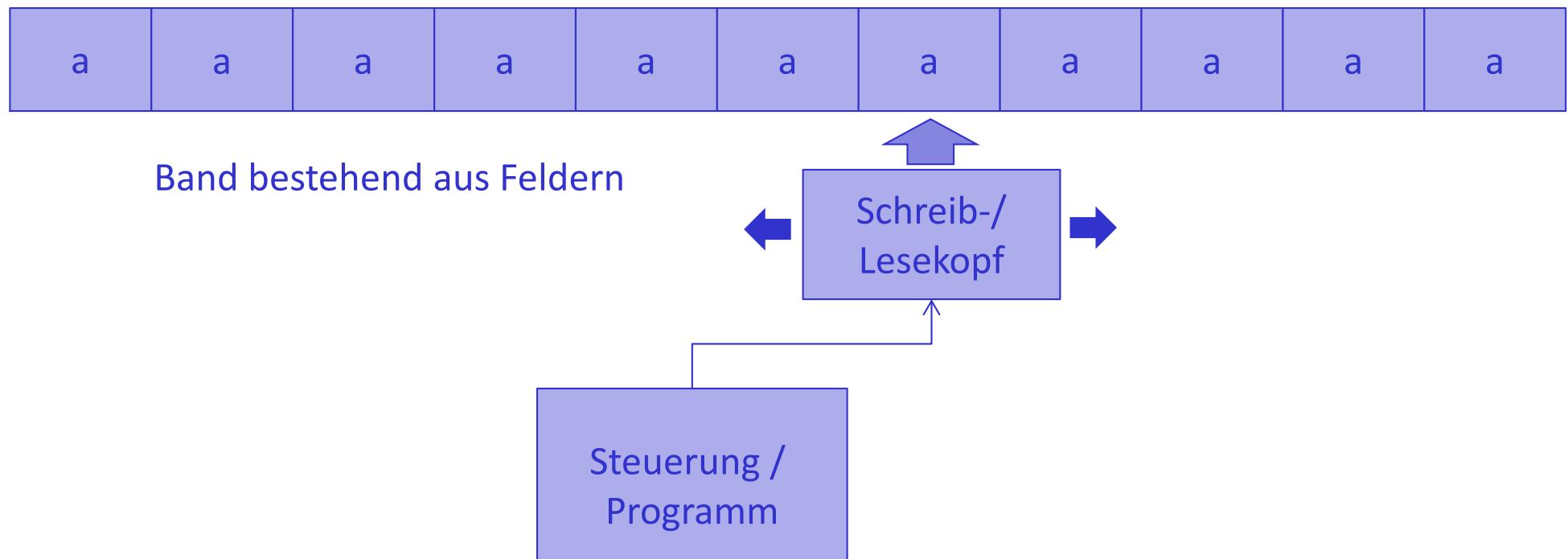
Turingmaschine

- Alain Turing konzipierte 1936 eine "rechnende Maschine", die heute im zu Ehren als Turingmaschine bezeichnet wird.
- Die Turingmaschine (TM) ist keine wirkliche Maschine (obwohl es diverse praktische Anschauungsbeispiele gibt), sondern mehr ein Gedankenmodell zum theoretischen Studium der Berechenbarkeit
- Eine TM besteht aus einem festen Teil ("Hardware") und einem variablen Teil ("Software").
- Die TM ist damit nicht eine Maschine, die genau eine Sache tut, sondern kann eine ganze Menge von verschiedenen Maschinen definieren. Alle Maschinen sind aber nach einem festen Schema aufgebaut.
- Diese Beschreibung suggeriert, dass eine TM als eine Art primitiver Computer verstanden werden kann. Dies war aber nicht die Absicht von Alan Turing.

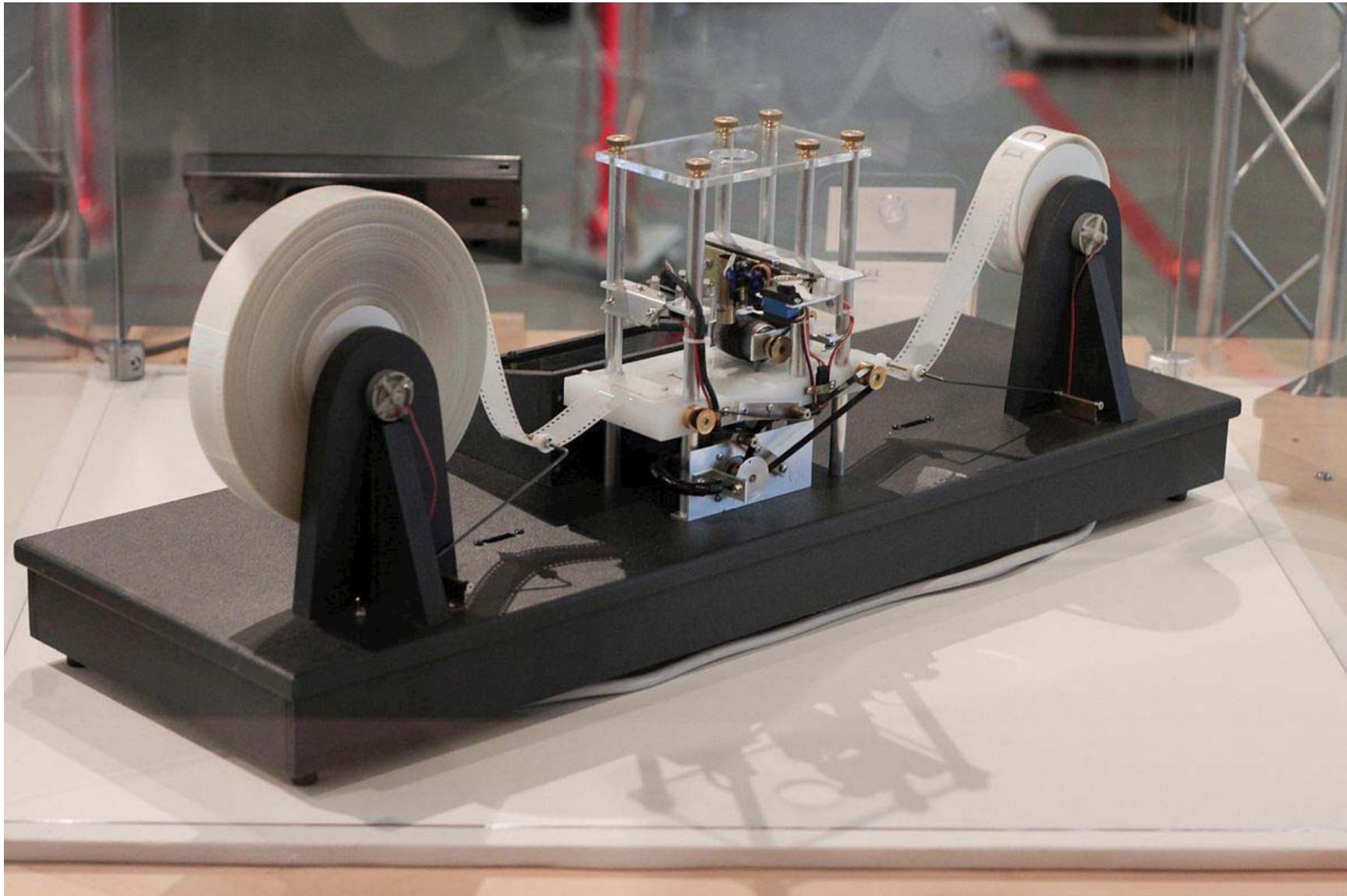


Turingmaschine - Hardware

- Die Hardware besteht aus
 - einem unendlich langen Band welches aus einzelnen Feldern besteht
 - einem Schreib-/Lesekopf
 - der Steuerung



Gedankenmodell der Turingmaschine



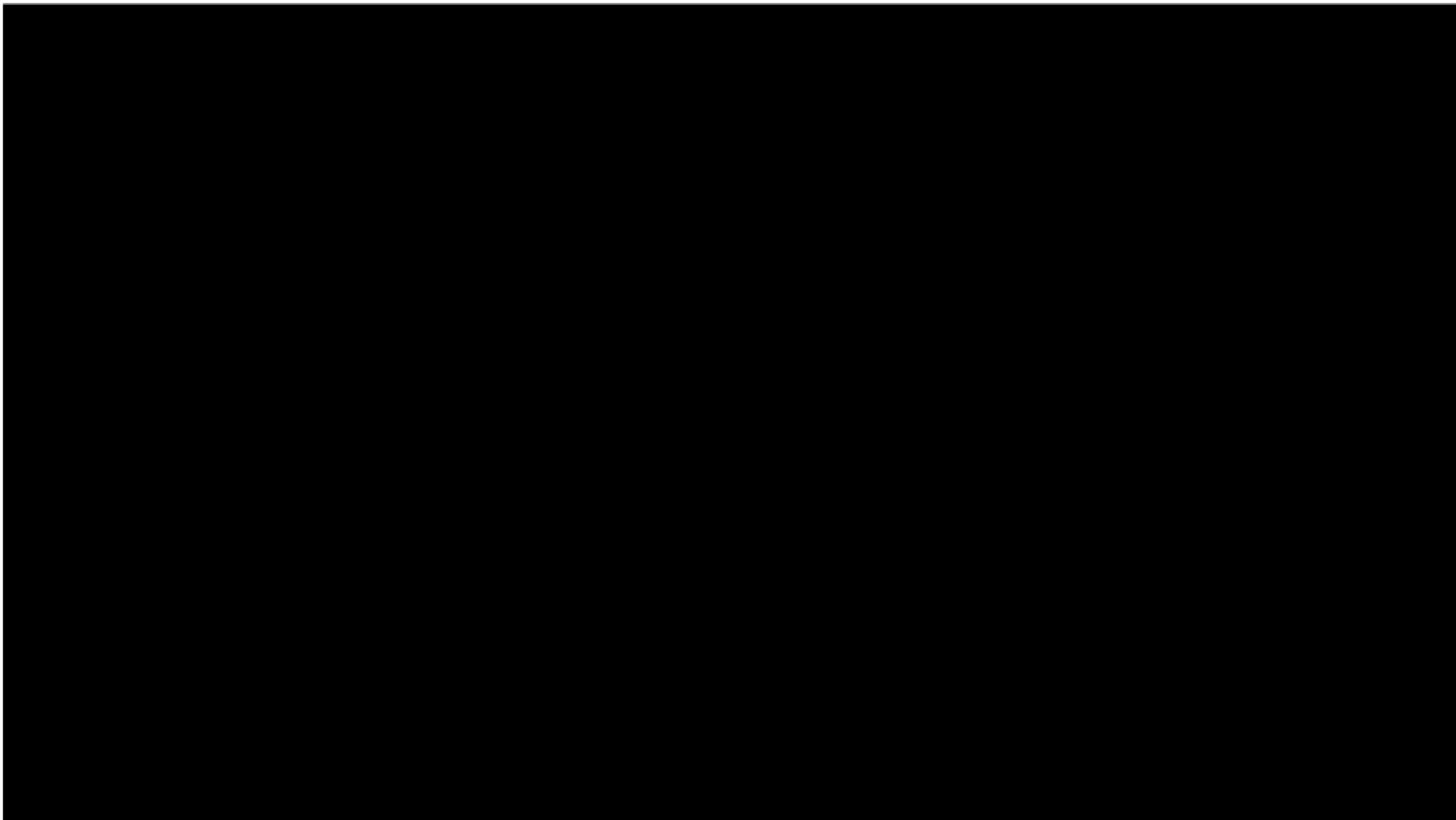
Turingmaschine

- Jedes Feld des Bandes trägt ein Zeichen aus einem frei wählbaren (aber für eine Maschine festen) Bandalphabet (Menge von Zeichen).
- Der Schreib-/Lesekopf ist über einem Feld positioniert, welches dann nach Anweisung eingelesen und/oder beschrieben werden kann.
- Der Schreibkopf kann sich nach (gemäß Anweisung) nach rechts oder links bewegen (oder auch stehen bleiben)
- Die Maschine kann sich in einem von endlich vielen verschiedenen Zuständen befinden (je nach Maschine verschieden).
- Die Tätigkeit von Turingmaschinen kann man durch ein 5-Tupel definieren: (Zustand, Eingabezeichen, Ausgabezeichen, Kopfbewegung, Folgezustand)

Turingmaschine - Steuerung

Die Steuerung, der variable Teil der Maschine, befindet sich in einem von endlich vielen Zuständen und arbeitet auf der Basis ihres speziellen Programms wie folgt:

1. Am Anfang befindet sich die Maschine im sog. Startzustand, das Band ist mit einer Eingabe belegt und die Position des Schreib-/Lesekopfes ist festgelegt.
2. Lese das Zeichen unter dem Lesekopf vom Band.
3. Abhängig vom gelesenen Zeichen und dem aktuellen Zustand der Steuerung führe alle folgende Aktionen aus:
 - Schreibe ein Zeichen auf das Band,
 - bewege den Schreib-/Lesekopf um ein Feld nach links oder rechts,
 - überführe die Steuerung in einen neuen Zustand.
4. Wiederhole diese Schritte solange bis ein spezieller Endzustand erreicht wird.



Turingmaschine

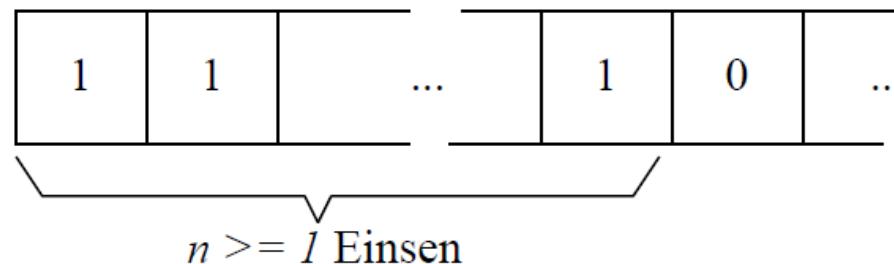
- Die auszuführenden Aktionen kann man in einer Übergangstabelle notieren.
Diese Tabelle stellt das Programm der Turingmaschine dar.
- Beispiel:

Zustand	Eingabe	Ausgabe	Bewegungsrichtung	Folgezustand
1	0	0	L	2
2	1	1	R	1
*	*	*	**	*

- * Optionen verschieden - je nach Alphabet und Programm der TM
- ** Optionen L , R und N bzw. 0

Beispiel 1: TM zum Löschen einer Einserkette.

- Das Bandalphabet enthalte nur die Zeichen 0 und 1.
- Zu Beginn der Bearbeitung habe das Band folgende Gestalt:



- Der Kopf steht zu Beginn auf der Eins ganz links. Folgendes Programm mit zwei Zuständen löscht die Einserkette und stoppt:

Zustand	Eingabe	Ausgabe	Bewegungsrichtung	Folgezustand
1	1	0	R	1
1	0	0	R	2
2 (Endzustand)				



Beispiel 2: Ungerade Paritätsprüfung

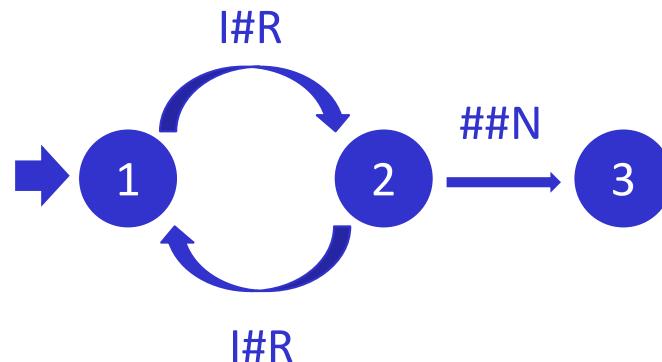
- TM soll alle Strichfolgen mit einer ungeraden Anzahl an Strichen | (bits) akzeptieren
 - Es gibt Striche und Leezeichen (#)
 - Zu Beginn befindet sich der Lese-Schreib-Kopf auf dem ersten Stich ganz links im Zustand 1.
 - Zustände:

1 - gerade Zahl gelesen	3 - Strichfolge akzeptiert
2 - ungerade Zahl gelesen	

Zustand	Eingabe	Ausgabe	Bewegungsrichtung	Folgezustand
1		#	R	2
2		#	R	1
2	#	#	N	3
1	#			Abbruch
3 (erfolgreicher Endzustand)				Ende

Das selbe Beispiel kann auch auf andere Arten dargestellt werden:

- Als Zustandsgraph / Übergangsgraph



- Jeder Knoten ist ein Zustand.
- Jeder Pfeil entspricht einer Zeile der Übergangstabelle.

- als Turingtabelle:

		gelesene Eingabezeichen				steht für undefiniertes Zeichen hält an / Band nicht akzeptiert
		#	1	*		
Zustände	1		#R2			
	2	#N3	#R1			

Ausgabezeichen / Kopfbewegung / neuer Zustand

Binärdarstellung

Dezimalzahlen

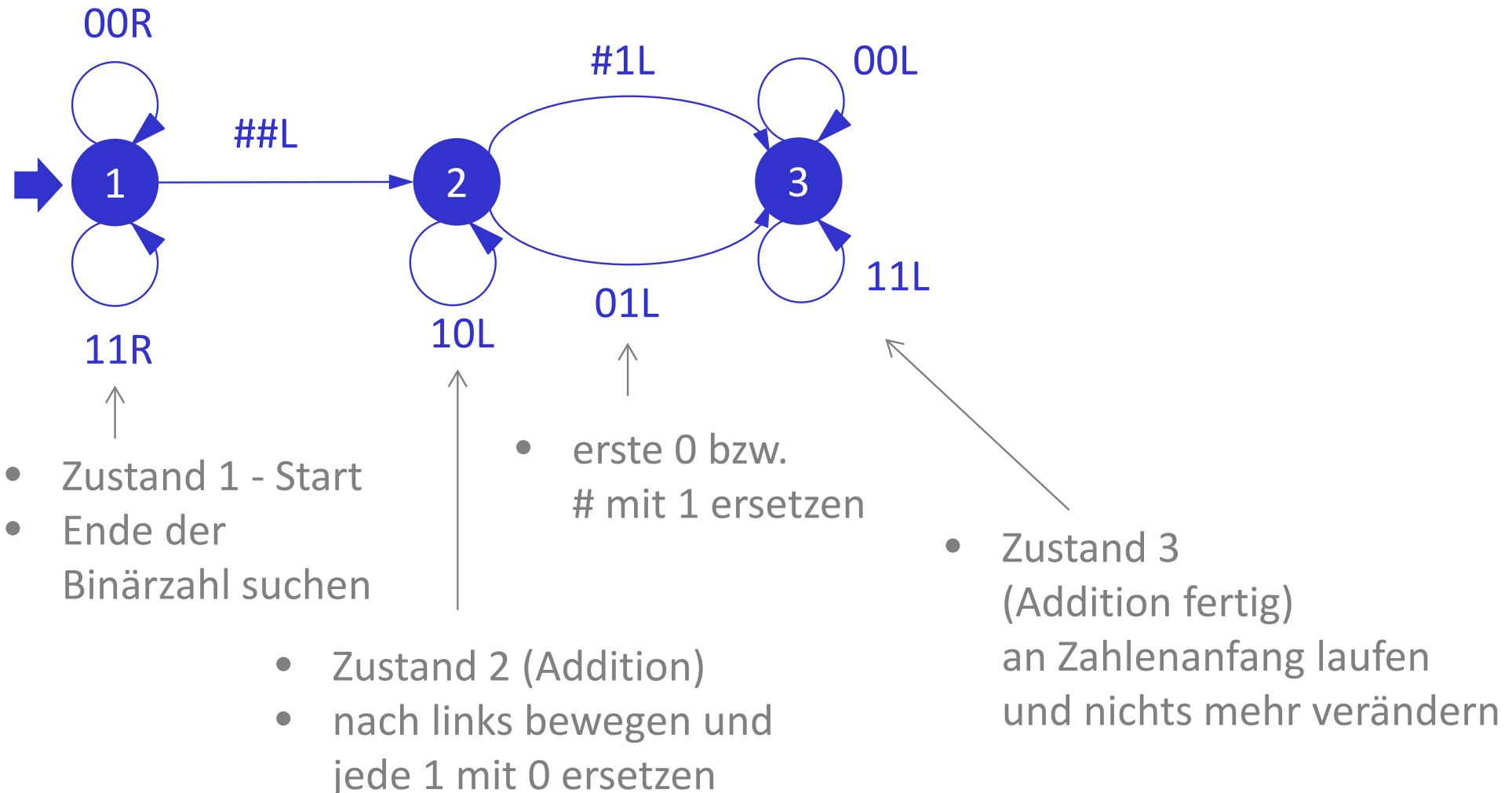
0 bis 15

im Dualsystem

Wertigkeit:	8 4 2 1
Null:	0 0 0 0
Eins:	0 0 0 1
Zwei:	0 0 1 0
Drei:	0 0 1 1
Vier:	0 1 0 0
Fünf:	0 1 0 1
Sechs:	0 1 1 0
Sieben:	0 1 1 1
Acht:	1 0 0 0
Neun:	1 0 0 1
Zehn:	1 0 1 0
Elf:	1 0 1 1
Zwölf:	1 1 0 0
Dreizehn:	1 1 0 1
Vierzehn:	1 1 1 0
Fünfzehn:	1 1 1 1

$$[1101]_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = [13]_{10}$$

Beispiel 3 - Binäre Addition von 1



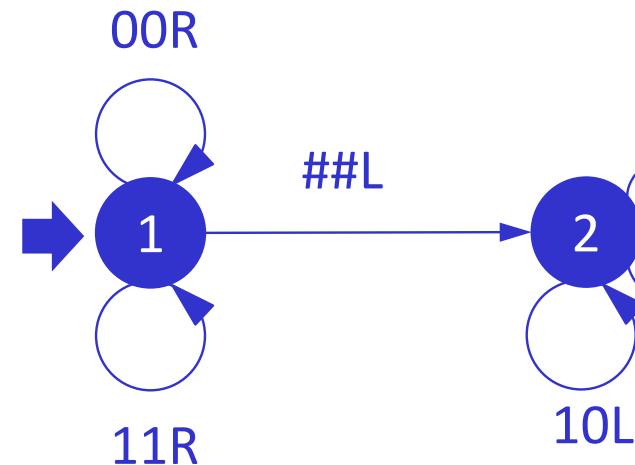
Varianten von Turingmaschinen

- Es gibt verschiedene Varianten von Turingmaschinen, die aber gleichwertige Leistungsfähigkeiten aufweisen
- Beispiele:
 - Band nur einseitig unendlich, an anderer Seite begrenzt
 - TM mit mehreren Bändern und Lese-/Schreibköpfen
 - TM mit mehreren Spuren (mehrere Ein-/Ausgabe pro Feld)
 - TM mit großem Eingabealphabet
 - einer oder mehrere zulässige Endzustände

Berechnungsprozesse

Bemerkung: Wir erkennen die drei wesentlichen Komponenten von Berechnungsprozessen:

- Grundoperationen
- Selektion
- Wiederholung



Grundbegriffe

- **Formale Systeme & Sprachen**
- **Bäume und Graphen**
- **Automatentheorie, Turingmaschine**

- **Algorithmen**
- **Berechenbarkeit**
- **Reale Computer**

- **Programmiersprachen**
- **Grenzen der Harwareentwicklung**



Problem, Algorithmus, Programm

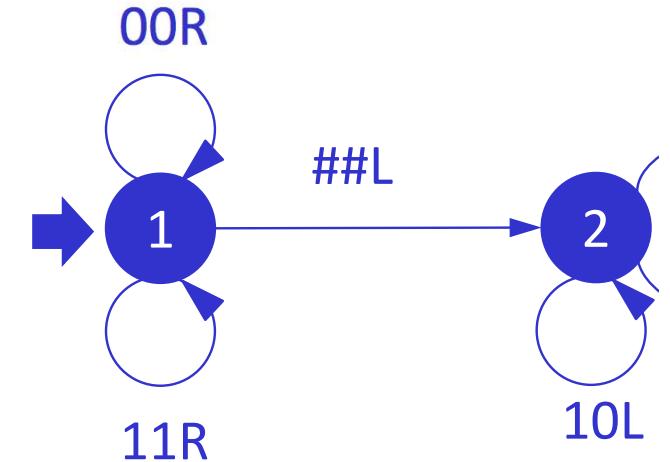
Definition: Ein **Problem** ist eine zu lösende Aufgabe. Wir sind daran interessiert Verfahren zu finden, die Aufgaben in einer Klasse von Problemen zu lösen. Das konkrete zu lösende Problem wird mittels **Eingabeparameter** ausgewählt.

Beispiel: Finde die kleinste von $n \geq 1$ Zahlen x_1, \dots, x_n , $x_i \in \mathbb{N}$.

Grundoperationen / Einzelschritte

Bsp.: MIPS Architektur (32 bit long instructions):

6 5 5 5 5 6 bits					
[op		rs		rt	rd shamt funct] R-type
[op		rs		rt	address/immediate] I-type
[op				target address] J-type



Addiere Register 1 & 2, Platziere Resultat in Register 6:

[op		rs		rt	rd shamt funct]	
0		1		2	6 0 32	decimal
000000	00001	00010	00110	00000	100000	binary

Lade Wert aus der in Register 3 angegebenen Speicherzelle + 68 Zellen in Register 8:

[op		rs		rt	address/immediate]	
35		3		8	68	decimal
100011	00011	01000	00000	00001	000100	binary

Problem, Algorithmus, Programm

Definition: Ein **Problem** ist eine zu lösende Aufgabe. Wir sind daran interessiert Verfahren zu finden, die Aufgaben in einer Klasse von Problemen zu lösen. Das konkrete zu lösende Problem wird mittels **Eingabeparameter** ausgewählt.

Beispiel: Finde die kleinste von $n \geq 1$ Zahlen x_1, \dots, x_n , $x_i \in \mathbb{N}$.

Definition: Ein **Algorithmus** beschreibt, wie ein Problem einer Problemklasse mittels einer Abfolge bekannter Einzelschritte gelöst werden kann.

Beispiel: Das Minimum von n Zahlen könnte man so finden: Setze $\min = x_1$. Falls $n = 1$ ist man fertig. Ansonsten teste der Reihe nach für $i = 2, 3, \dots, n$ ob $x_i < \min$. Falls ja, setze $\min = x_i$.

Herkunft des Wortes Algorithmus

- benannt nach dem persischen Mathematiker und Astronom Ibn Musa **Al-Chwarismi**
- schrieb im 9. Jahrhundert ein wichtiges mathematisches Lehrbuch, das u.a. sehr zur Verbreitung der "indisch-arabischen" Ziffern beitrug

Nes χ i: ۱ ۲ ۳ ۴ ۰ ۶ ۷ ۸ ۹

Gobar: ۱ ۲ ۳ ۴ ۵ ۶ ۷ ۸ ۹

Syakat: ۱ ۰ ۱ ۲ ۳ ۴ ۵ ۶ ۷ ۸ ۹

1 2 3 4 5 6 7 8 9



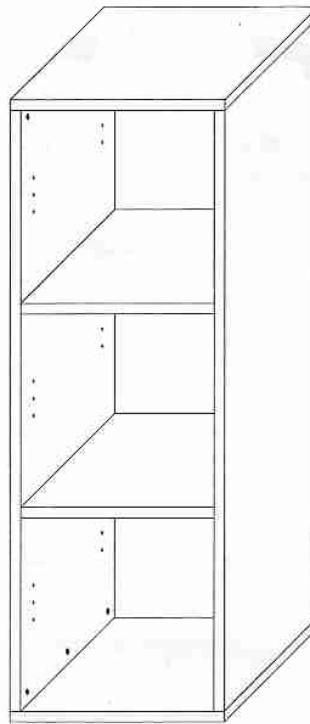
- **Erster Computeralgorithmus:**

wurde 1843 von Ada Lovelace in ihren Notizen zur Analytical Engine von Charles Babbage niedergeschrieben. Tatsächliche Implementierung blieb aus, weil Charles Babbage seine Analytical Engine nicht vollenden konnte (s. 1. Vorlesung)

Aufbauanleitungen sind keine Algorithmen

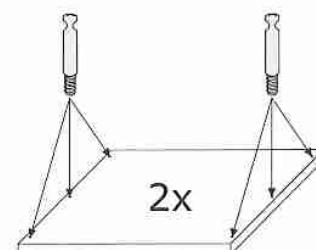
AUFWAUAUANLEITUNG

WÜRFELSYSTEM XERO



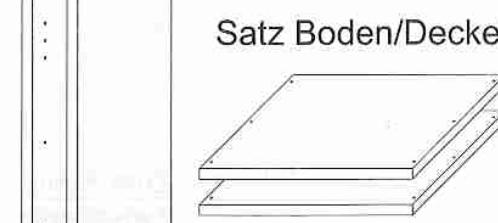
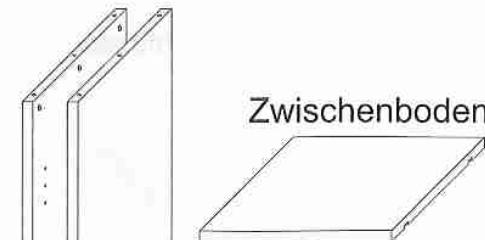
Aufbauanleitung am Beispiel eines Würfelsystems XERO in 116 cm Höhe mit 3 Zwischenböden.

1. Je sechs Verbinderbolzen mit einem geeigneten Schraubendreher in die Bohrungen von Boden und Deckel einschrauben.



Stückliste

Die Anzahl der Zwischenböden, sowie die Maße der Seitenteile richtet sich nach Ihrer Bestellung.



2 Seitenteile

Mögliche Höhen: 36 / 74 / 112 / 150 oder 188 cm

Zubehör

pro Satz Boden/Deckel

12x Verbinder

12x Verbinderbolzen

12x Gewindestift

1x Imbusschlüssel

pro Zwischenboden

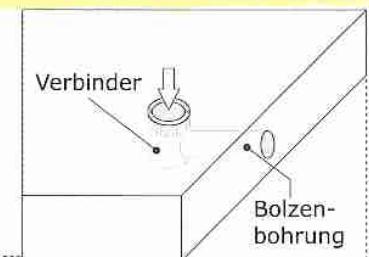
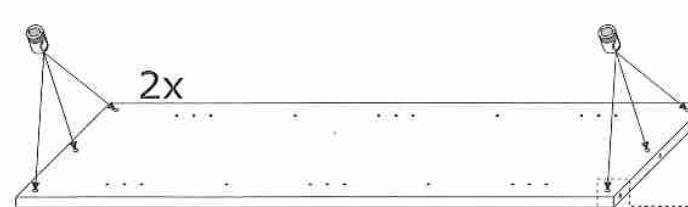
4x Excenterbolzen

4x Excenter

Tipp: Fetten oder ölen Sie die Gewinde der Bolzen vorher leicht ein, um das Eindrehen in das Massivholz zu erleichtern!

Bitte beachten Sie, dass ab einer Seitenhöhe von 150 cm zur zusätzlichen Stabilisierung mindestens ein Zwischenboden montiert werden muss!

2. Die Verbinder in die seitlichen Bohrungen der Seitenteile einstecken, so dass die seitliche Bohrung des Verbinder in Richtung der Bolzenbohrung weist.



Eigenschaften von Algorithmen

Ein Algorithmus muss gewisse Eigenschaften erfüllen:

- Ein Algorithmus beschreibt ein generelles Verfahren zur Lösung einer Schar von Problemen.
- Trotzdem soll die Beschreibung des Algorithmus endlich sein. Nicht erlaubt ist also z. B. eine unendlich lange Liste von Fallunterscheidungen.
- Ein Algorithmus besteht aus einzelnen Elementaroperationen, deren Ausführung bekannt und endlich ist. Als Elementaroperationen sind also keine „Orakel“ erlaubt.

Algorithmus

Def. Algorithmus: schematische^[1] Vorgehensweise, mit der jedes Problem einer bestimmten Klasse^[2] mit endlich vielen^[3] elementaren^[4] Schritten/Operationen gelöst wird.

1. Schematisch: man kann den Algorithmus ausführen ohne ihn zu verstehen (\Rightarrow Computer)
2. Klasse: z.B. die Wurzel aus jeder beliebigen nicht-negativen Zahl, mit nicht nur $\sqrt{11}$ (eine einzige Eingabe)
3. Endlich viele Schritte: man kommt nach endlich vielen Schritten zur Lösung.
4. Elementare Schritte/Operationen: führen die Lösung auf Operationen (Teilprobleme) zurück, die wir schon gelöst haben

Spezielle Algorithmen sind:

- **Terminierende Algorithmen:** Der Algorithmus stoppt für jede zulässige Eingabe nach endlicher Zeit.
- **Deterministische Algorithmen:** In jedem Schritt ist bekannt, welcher Schritt als nächstes ausgeführt wird.
- **Determinierte Algorithmen:** Algorithmus liefert bei gleicher Eingabe stets das gleiche Ergebnis.
- Ein terminierender, deterministischer Algorithmus ist immer determiniert. Terminierende, nichtdeterministische Algorithmen können determiniert sein oder nicht. Determinierte Algorithmen sind nicht unbedingt deterministisch.
- **Definition:** Ein **Programm** ist eine Formalisierung eines Algorithmus. Ein Programm kann auf einer Maschine (z. B. TM, PC etc.) ausgeführt werden.
- **Beispiel:** Das Minimum von n Zahlen kann mit einer TM berechnet werden. Die Zahlen werden dazu in geeigneter Form kodiert (z. B. als Einserketten oder als Binärzahlen) auf das Eingabeband geschrieben.

Problem, Algorithmus, Programm

Wir haben also das Schema:

Problem => Algorithmus => Programm.

Die Informatik beschäftigt sich damit, algorithmische Problemlösungen systematisch zu finden:

- Zunächst muss das Problem analysiert und möglichst präzise formuliert werden. Dieser Schritt wird auch als Modellierung bezeichnet.
- Im folgenden entwirft man einen effizienten Algorithmus zur Lösung des Problems. Dieser Schritt ist von zentralem Interesse für die Informatik.
- Schließlich muss der Algorithmus als Computerprogramm formuliert werden, welches auf einer konkreten Maschine ausgeführt werden kann.

Berechenbarkeit und Turing-Äquivalenz

Es sei \mathcal{A} das Bandalphabet einer TM. Wir können uns die Berechnung einer konkreten TM (d. h. gegebenes Programm) auch als Abbildung vorstellen:

$$f : \mathcal{A}^* \rightarrow \mathcal{A}^*.$$

Hält die TM für einen Eingabewert nicht an, so sei der Wert von f undefiniert.

Dies motiviert folgende allgemeine

Definition: Eine Funktion $f : E \rightarrow A$ heisst **berechenbar**, wenn es einen Algorithmus gibt, der für jede Eingabe $e \in E$, für die $f(e)$ definiert ist, terminiert und das Ergebnis $f(e) \in A$ liefert.

Welche Funktionen sind in diesem Sinne berechenbar?

Turing- Äquivalenz

Auf einem PC mit unendlich viel Speicher könnte man mit Leichtigkeit eine TM **simulieren**. Das bedeutet, dass man zu jeder TM ein äquivalentes PC-Programm erzeugen kann, welches das Verhalten der TM Schritt für Schritt nachvollzieht. Ein PC (mit unendlich viel Speicher) kann daher alles berechnen, was eine TM berechnen kann.

Interessanter ist aber, dass man zeigen kann, dass die TM trotz ihrer Einfachheit alle Berechnungen durchführen kann, zu denen der PC in der Lage ist. Zu einem PC mit gegebenem Programm kann man also eine TM angeben, die die Berechnung des PCs nachvollzieht! Computer und TM können dieselbe Klasse von Problemen berechnen!

Bemerkung: Im Laufe von Jahrzehnten hat man viele (theoretische und praktische) Berechnungsmodelle erfunden. Die TM ist nur eines davon. Jedes Mal hat sich herausgestellt: Hat eine Maschine gewisse Mindesteigenschaften, so kann sie genausoviel wie eine TM berechnen. Dies nennt man **Turing-Äquivalenz**.

Berechenbarkeit

- **Church'sche-These (bzw. Church-Turing-These)**

(Alonzo Church, US-amerikanischer Mathematiker, Logiker und Philosoph, 1903 - 1995)

Alles was man für intuitiv berechenbar hält, kann man mit einer TM ausrechnen.
Dabei heißt intuitiv berechenbar, dass man einen Algorithmus dafür angeben kann.

- Mehr dazu in den Vorlesungen zur Theoretischen Informatik.

- Diese These ist nicht beweisbar, da der Begriff „intuitiv berechenbare Funktion“ nicht exakt formalisiert werden kann.

- Berechenbare Probleme kann man mit Algorithmen lösen, die in verschiedenen Programmiersprachen implementiert sein können. Unterschiede dieser Implementierungen bestehen

- in der Länge und Eleganz der dafür nötigen Programme

- der zur Erstellung notwendigen Zeit

- auch Effizienz ihrer Ausführung kann sehr unterschiedlich sein, hängt auch sehr von der Compilerimplementation ab (dazu später mehr)

Nicht berechenbare Probleme

Bemerkung: Es gibt auch nicht berechenbare Probleme! So kann man z. B. keine TM angeben, die für jede gegebene TM entscheidet, ob diese den Endzustand erreicht oder nicht (**Halteproblem**).

Dieses Problem ist aber noch partiell-berechenbar, d. h. für jede terminierende TM erfährt man dies nach endlicher Zeit, für jede nicht-terminierende TM erfährt man aber kein Ergebnis.

Grundbegriffe

- Formale Systeme & Sprachen
- Bäume und Graphen
- Automatentheorie, Turingmaschine

- Algorithmen
- Berechenbarkeit
- Reale Computer

- Programmiersprachen
- Grenzen der Harwareentwicklung



Reale Computer

Algorithmen waren schon vor der Entwicklung unserer heutigen Computer bekannt, allerdings haperte es mit der Ausführung. Zunächst arbeiteten Menschen als „Computer“!

- Lewis Fry Richardson⁵ schlägt in seinem Buch *Weather Prediction by Arithmetical Finite Differences* vor, das Wetter für den nächsten Tag mit 64000 (!) menschlichen Computern auszurechnen. Der Vorschlag wird als unpraktikabel verworfen.
- In Los Alamos werden Lochkartenmaschinen und menschliche Rechner für Berechnungen eingesetzt. Richard Feynman⁶ organisierte sogar einen Wettbewerb zwischen beiden.

⁵Lewis Fry Richardson, brit. Meteorologe, 1881–1953.

⁶Richard P. Feynman, US-amerik. Physiker, Nobelpreis 1965, 1918–1988.



Menschliche Rechner Ende des 19. Jahrhunderts

Der Startpunkt der Entwicklung realer Computer stimmt (zufällig?) relativ genau mit der Entwicklung theoretischer Berechenbarkeitskonzepte durch Church und Turing überein.

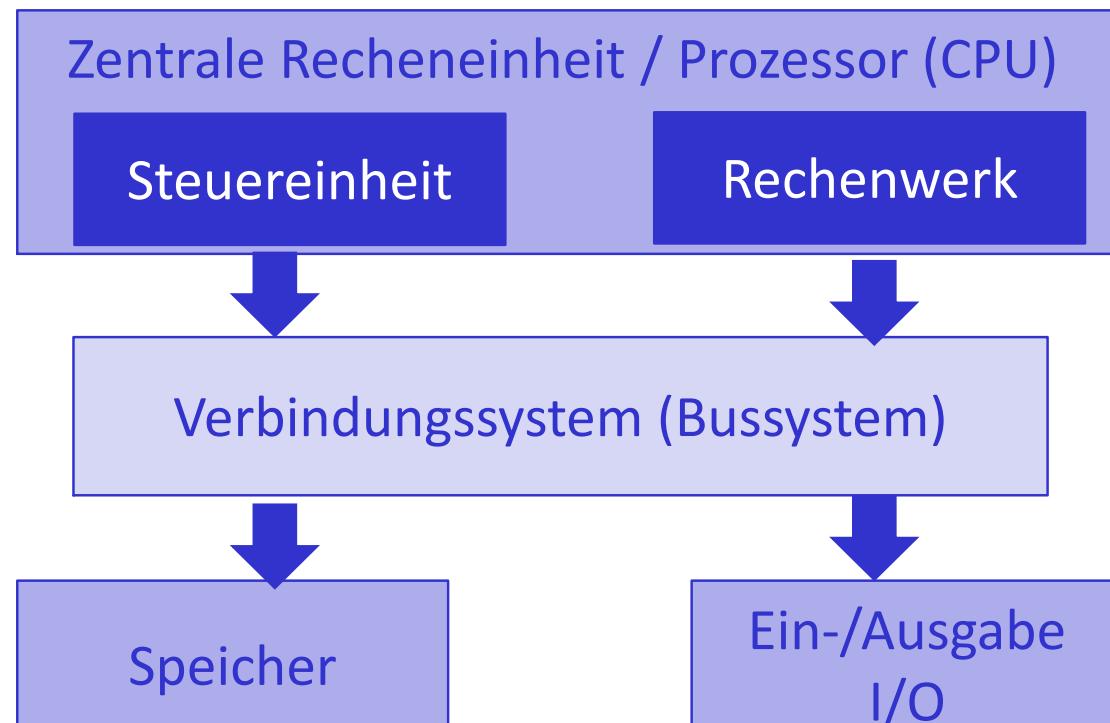
Dabei verstehen wir Computer bzw. (Universal-)Rechner als Maschinen zur Ausführung beliebiger Algorithmen in obigem Sinne (d. h. sie können nicht „nur“ rechnen im Sinne arithmetischer Operationen).

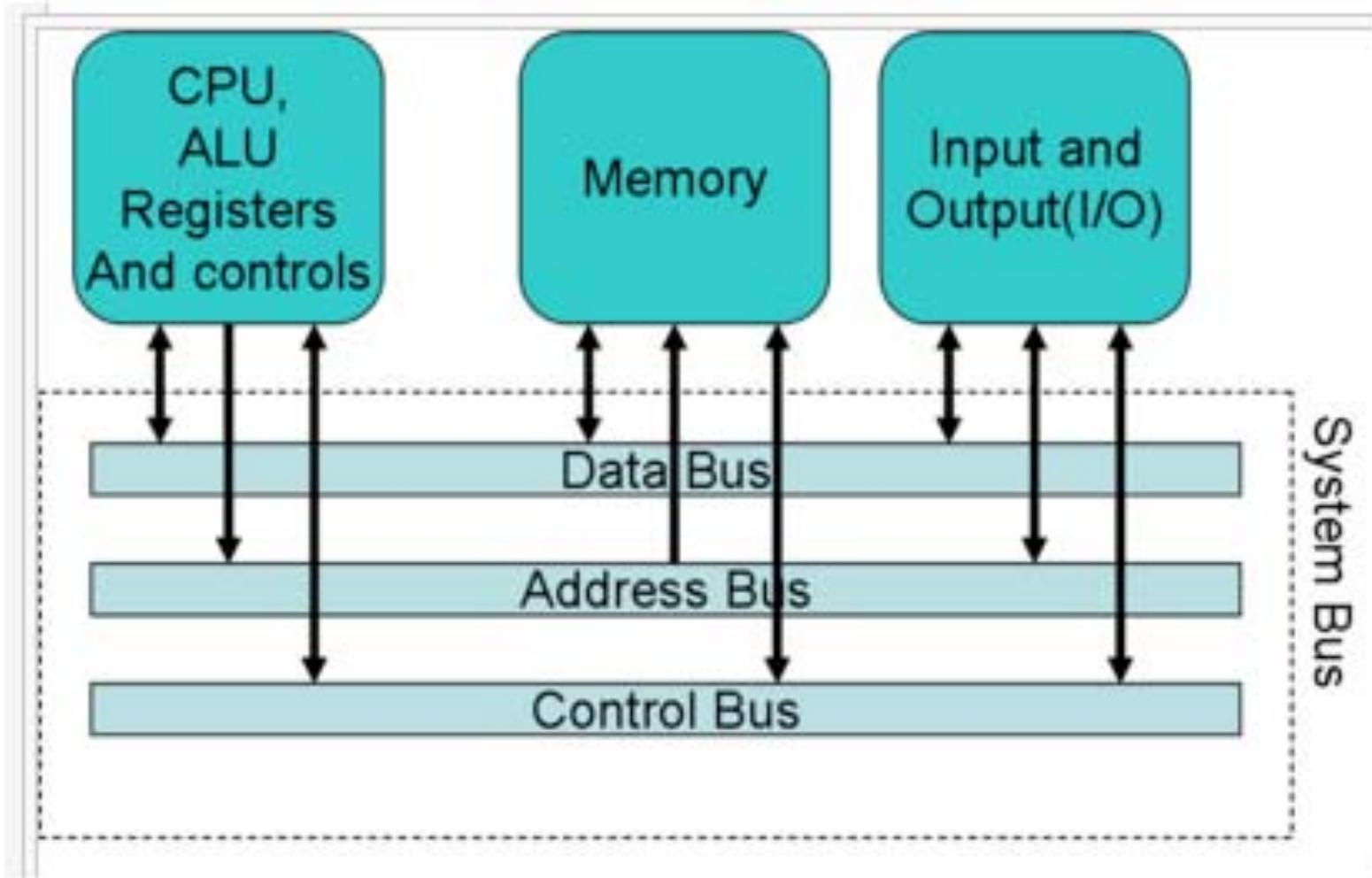
Einige der wichtigsten frühen Rechenmaschinen waren:

- Zuse Z3, Mai 1941, mechanisch, turing-vollständig (aber nicht als solcher konstruiert), binäre Gleitkommaarithmetik
- Atanasoff-Berry-Computer, Sommer 1941, elektronisch (Röhren), nicht turing-mächtig, gebaut zur Lösung linearer Gleichungssysteme (29×29)
- Colossus, 1943, elektronisch, nicht turing-mächtig, Kryptographie
- Mark 1, 1944, mechanisch, turing-vollständig, Ballistik
- ENIAC, 1946, elektronisch, turing-vollständig, Ballistik
- EDVAC, 1949, elektronisch, turing-vollständig, Ballistik, erste „Von-Neumann-Architektur“

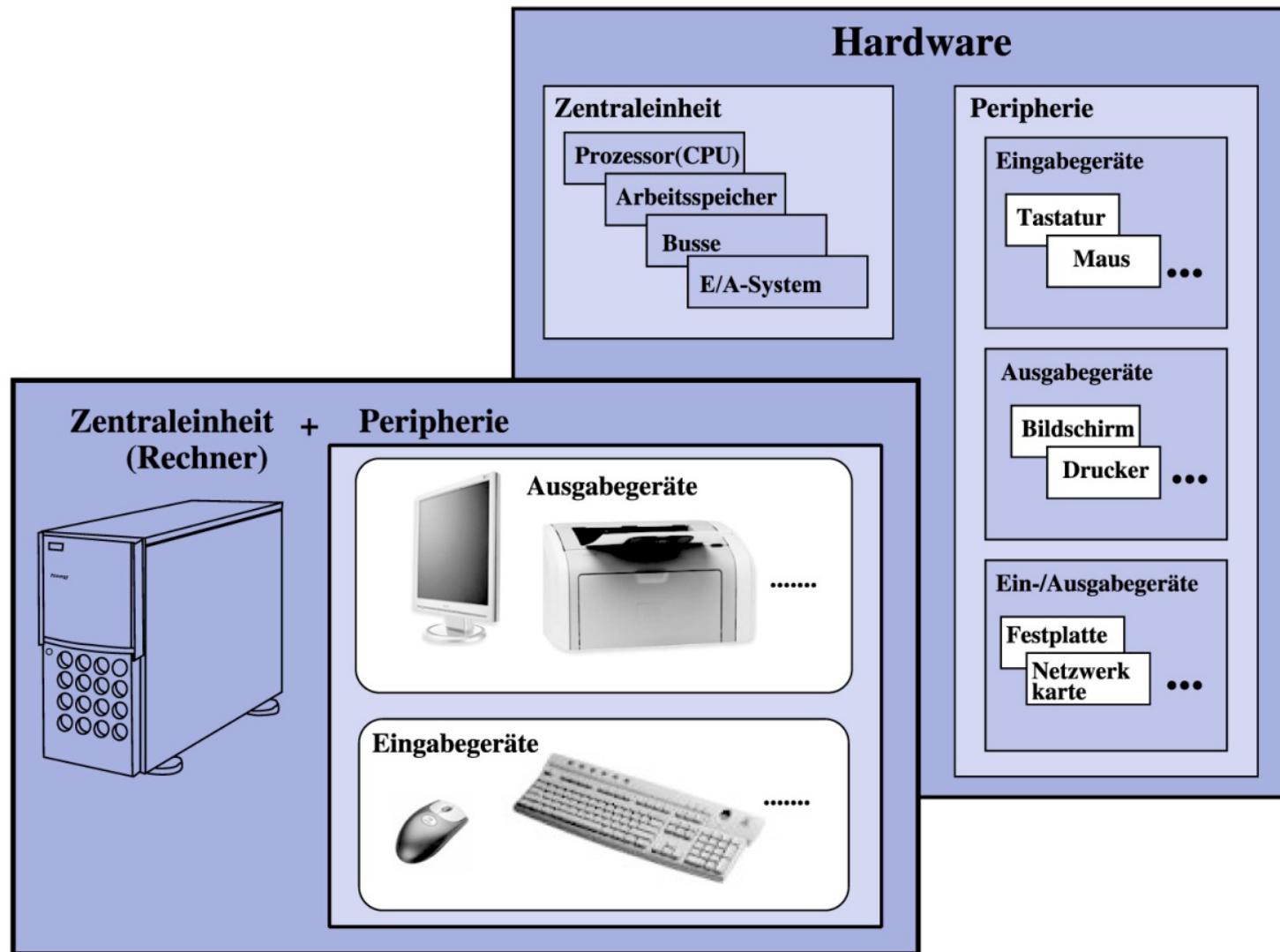
von Neumann Rechnerarchitektur

- Praktische Computer basieren meist auf dem von John von Neumann 1945 im Rahmen der EDVAC-Entwicklung eingeführten Konzept.
- John von Neumann (ursprünglich Janos Neumann Margittai), Mathematiker österreichisch-ungarischer Herkunft, 1903 - 1957. Von ihm stammt die Spieltheorie, die mathematische Begründung der Quantenmechanik, und er lieferte wichtige Beiträge zu Informatik und Numerik.
- Architektur / Komponenten:





Computersystem = Zentraleinheit + Peripherie

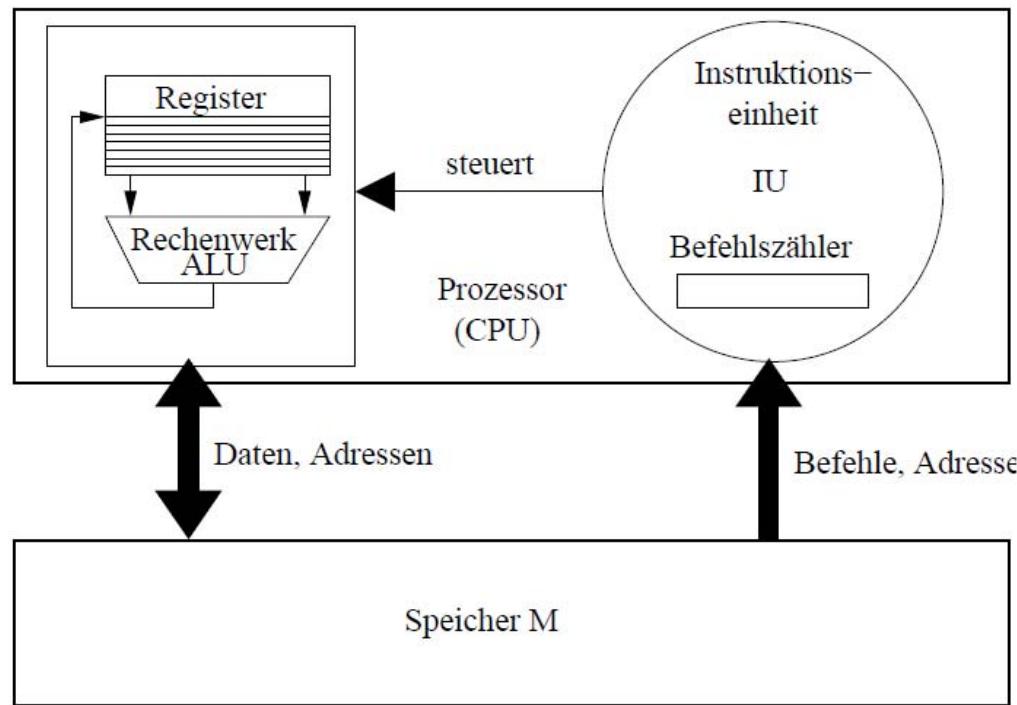


Herold, Lurz, Wohlrb - Grundlagen der Informatik

Peripheriegeräte

Geräte	Eingabe	Ausgabe	Massenspeicher
Tastatur, Maus, Scanner, Digitalkamera	X		
DVD-ROM-Laufwerk	X		X
Grafikkarte, Bildschirm, Drucker, Lautsprecher		X	
Netzwerkkarte, Modem, Soundkarte	X	X	
Festplatte, USB-Stick, DVD-Brenner	X	X	X

CPU & Speicher



Der **Speicher M** besteht aus endlich vielen Feldern, von denen jedes eine Zahl aufnehmen kann. Im Unterschied zur TM kann auf jedes Feld ohne vorherige Positionierung zugegriffen werden (**wahlfreier Zugriff, random access**).

Speicher

Zum Zugriff auf den Speicher wird ein Index, auch **Adresse** genannt, verwendet, d. h. wir können den Speicher als Abbildung

$$M : A \rightarrow D$$

auffassen.

Für die Adressen gilt $A = [0, N - 1] \subset \mathbb{N}_0$ wobei aufgrund der **binären** Organisation $N = 2^n$ gilt. n ist die Anzahl der erforderlichen Adressleitungen.

Für D gilt $D = [0, 2^m - 1]$ mit der **Wortbreite** m , die meistens ein Vielfaches von 8 ist. m ist die Anzahl der erforderlichen Datenleitungen.

Die Gesamtkapazität des Speichers ist demnach $m \cdot 2^n$ **Bit**. Jedes Bit kann zwei Werte annehmen, 0 oder 1. In der Praxis wird die Größe des Speichers in **Byte** angegeben, wobei ein Byte aus 8 Bit besteht. Damit enthält ein Speicher mit n Adressleitungen bei Wortbreite m genau $(m/8) \cdot 2^n$ Byte.

Speicher

Gebräuchlich sind auch noch die Abkürzungen 1 Kilobyte = 2^{10} Byte = 1024 Byte, 1 Megabyte = 2^{20} Byte, 1 Gigabyte = 2^{30} Byte.

Der Speicher enthält sowohl Daten (das Band in der TM) als auch Programm (die Tabelle in der TM). Den einzelnen Zeilen der Programmtabelle der TM entsprechen beim von Neumannschen Rechner die Befehle. Die Vereinigung von Daten und Programm im Speicher (**stored program computer**) war der wesentliche Unterschied zu den früheren Ansätzen.

Befehle werden von der **Instruktionseinheit** (instruction unit, IU) gelesen und dekodiert.

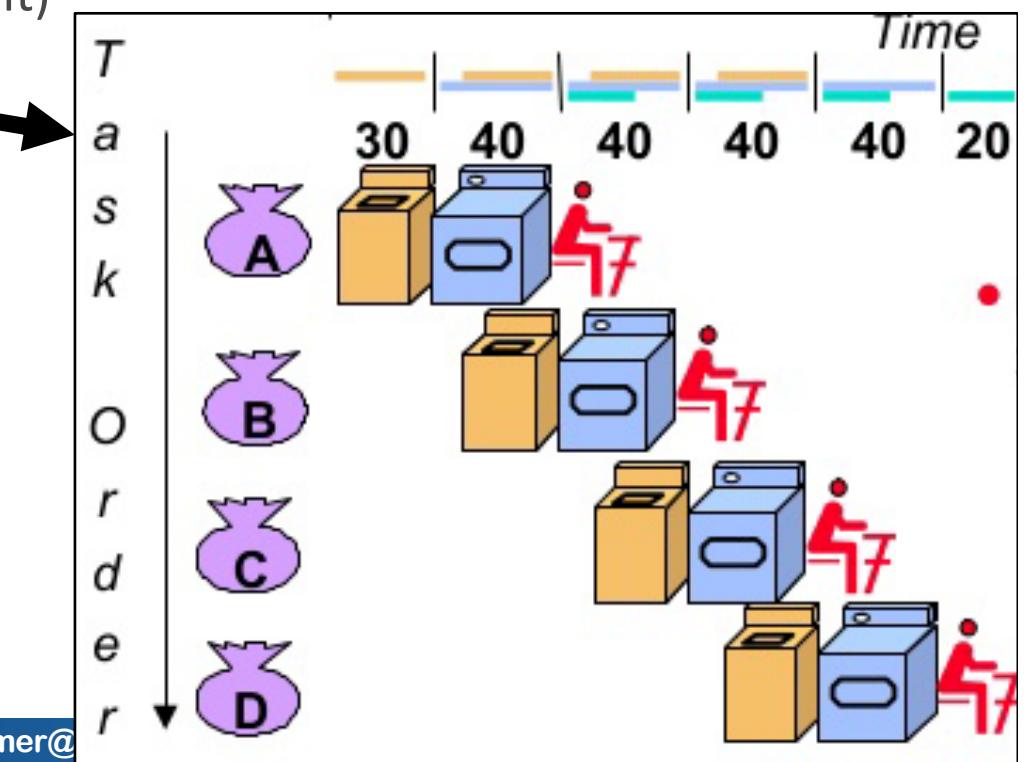
Die Instruktionseinheit steuert das Rechenwerk, welches noch zusätzliche Daten aus dem Speicher liest bzw. Ergebnisse zurückschreibt.

Befehlszyklus

- Die Maschine arbeitet zyklisch die folgenden Aktionen ab:
 - Befehl holen
 - Befehl dekodieren
 - Befehl ausführen
- Dies nennt man Befehlszyklus. Viel mehr über Rechnerhardware erfährt man in der Vorlesung "Technische Informatik".
- Bemerkung: Hier wurde insbesondere die Interaktion von Rechnern mit der Umwelt, die sog. Ein- und Ausgabe, in der Betrachtung vernachlässigt. Moderne Rechner haben insbesondere die Fähigkeit, auf äußere Einwirkungen hin (etwa Tastendruck) den Programmuss zu unterbrechen und an anderer Stelle (Turingmaschine: in anderem Zustand) wieder aufzunehmen.

Rechnerarchitekturen

- Bemerkung: Heutige Rechner sind wesentlich komplizierter als dieses einfache Modell.
- Insbesondere sind viele Möglichkeiten der parallelen Verarbeitung (von Daten und von Befehlen) enthalten.
- Wichtige Konzepte in modernen Rechnern sind:
 - Hierarchisch organisierter Speicher mit Caches (schnelle Zwischenspeicher zur Erhöhung der Zugriffsgeschwindigkeit)
 - Pipelining des Befehlsholzyklus
 - SIMD Instruktionen, Superskalarität
 - Multicorerechner



Grundbegriffe

- Formale Systeme & Sprachen
- Bäume und Graphen
- Automatentheorie, Turingmaschine

- Algorithmen
- Berechenbarkeit
- Reale Computer

- Programmiersprachen
- Grenzen der Harwareentwicklung



Programmiersprachen

Die Befehle, die der Prozessor ausführt, nennt man **Maschinenbefehle** oder auch **Maschinensprache**. Sie ist relativ umständlich, und es ist sehr mühsam größere Programme darin zu schreiben. Andererseits können ausgefeilte Programme sehr kompakt sein und sehr effizient ausgeführt werden.

Beispiel: Ein Schachprogramm auf einem 6502-Prozessor findet man unter

<http://www.6502.org/source/games/uchess/uchess.pdf>

Es benötigt weniger als 1KB an Speicher!

Programmiersprachen

Die weitaus meisten Programme werden heute in sogenannten **höheren Programmiersprachen** erstellt. Sinn einer solchen Sprache ist, dass der Programmierer Programme möglichst

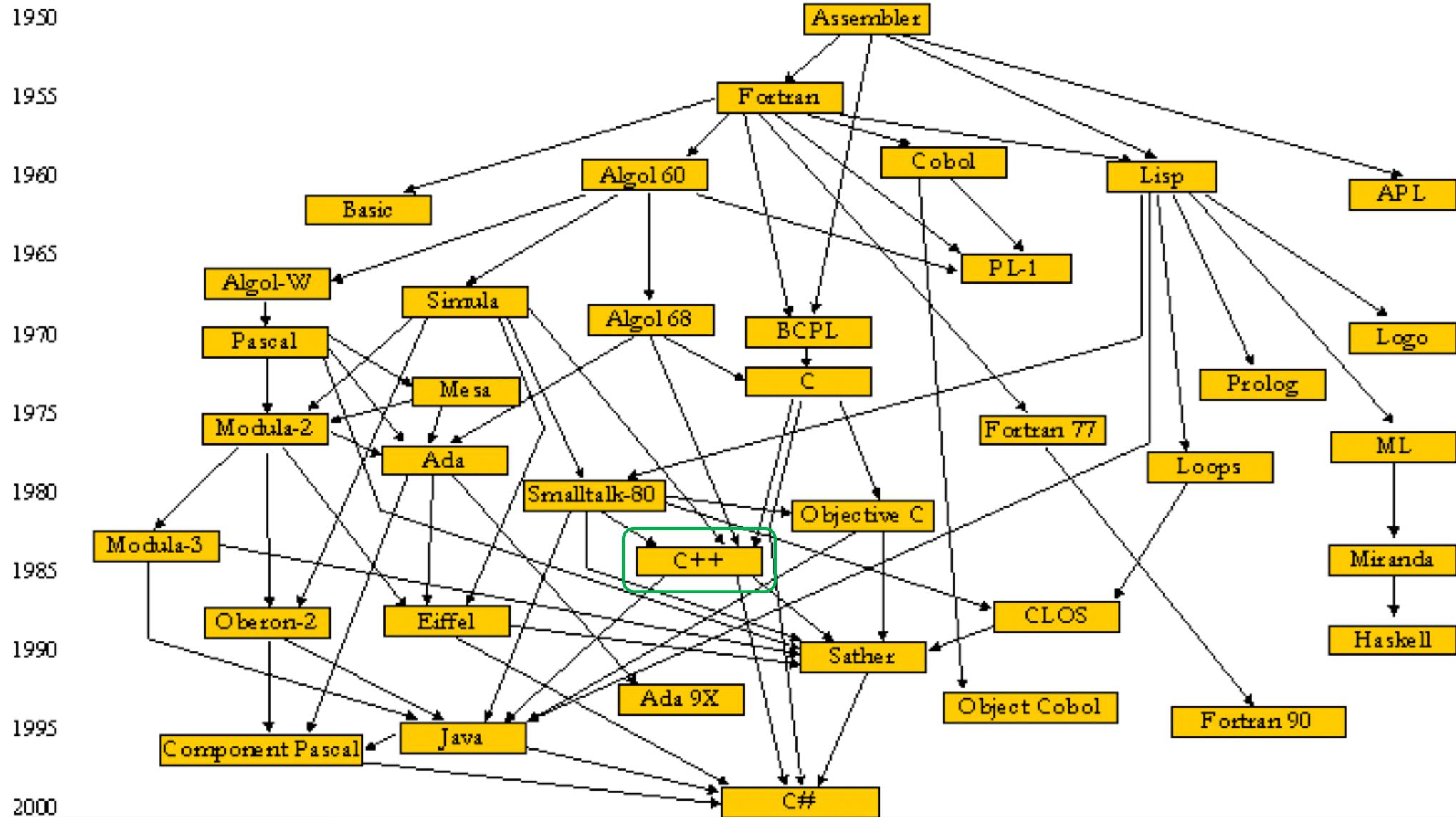
- schnell (in Sinne benötigter Programmiererzeit) und
- korrekt (Programm löst Problem korrekt)

erstellen kann.

Wir lernen in dieser Vorlesung die Sprache C++. C++ ist eine Weiterentwicklung der Sprache C, die Ende der 1960er Jahre entwickelt wurde.

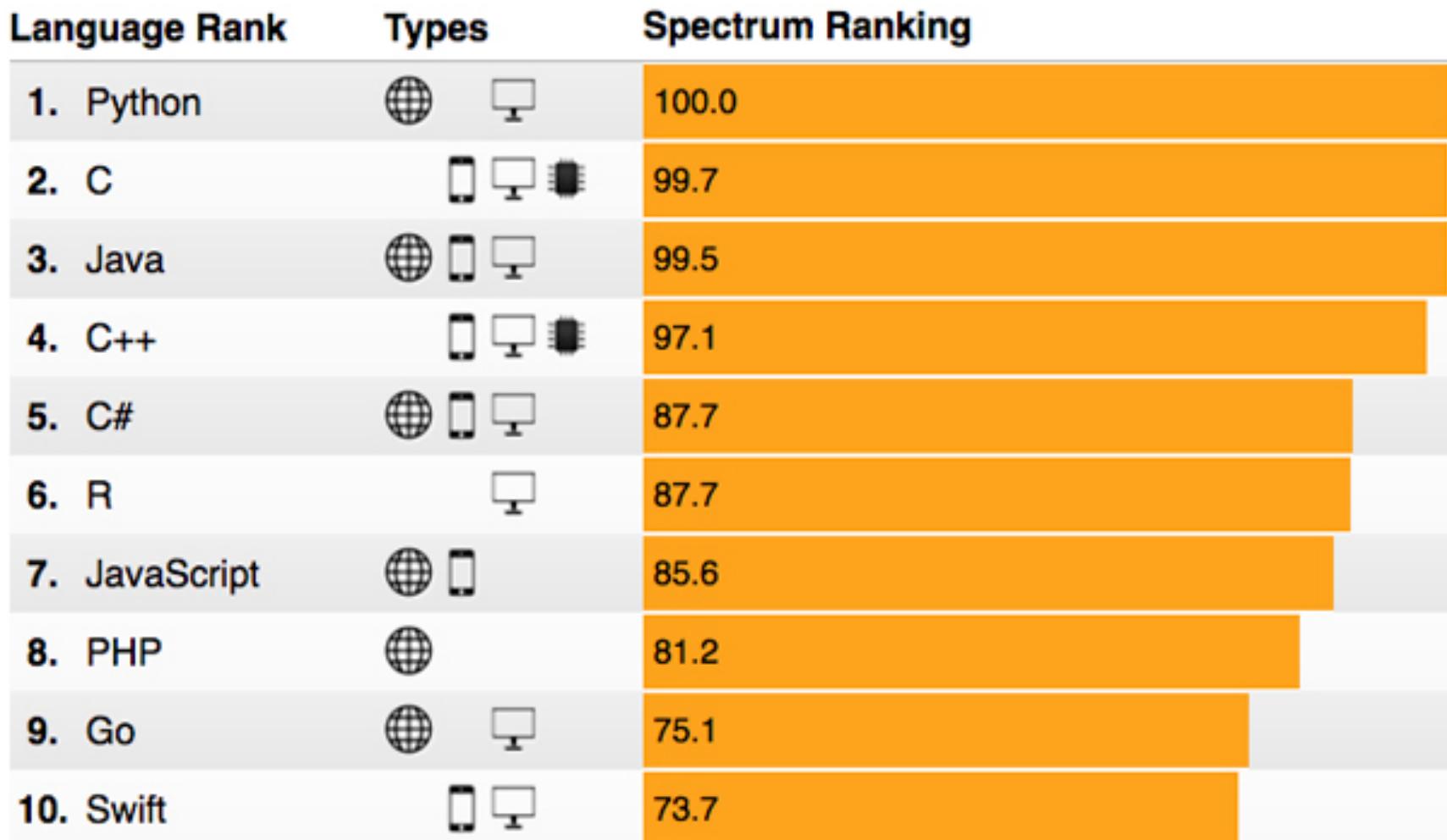
Programming Language Family Tree

F | H | A D G A



2017 Most used prog languages...

- for the typical IEEE Spectrum Reader (Informatiker & Ingenieure)



- deckt sich mit Angaben der Webpage [hackernoon.com](https://hackernoon.com/the-most-used-programming-languages-in-2017)

Top 10 Programming Languages

					
Paradigm	Multi-paradigm: object-oriented, imperative, functional, procedural, reflective	Imperative (procedural), structured	Multi-paradigm: object-oriented (class-based), structured, imperative, generic, reflective, concurrent	Multi-paradigm: procedural, functional, object-oriented, generic	Multi-paradigm: structured, imperative, object-oriented, event-driven, task-driven, functional, generic, reflective, concurrent
Designed by	Guido van Rossum	Dennis Ritchie	James Gosling	Bjarne Stroustrup	Microsoft
Developer	Python Software Foundation	Dennis Ritchie & Bell Labs (creators), ANSI X3J11 (ANSI C), ISO/IEC	Sun Microsystems (now owned by Oracle corporation)	Bell Labs	Microsoft
First appeared	20 February 1991 (26 years ago)	1972 (45 years ago)	May 23 1995 (22 years ago)	1983 (34 years ago)	2000 (17 years ago)
Typing discipline	Duck, dynamic, strong	Static, weak, manifest, nominal	Static, strong, safe, nominative, manifest	Static, nominative, partially inferred	Static, dynamic, strong, safe, nominative, partially inferred
Platform	Cross-platform	Cross-platform	Windows, Solaris, Linux, OS X	Linux, MacOS, Solaris	Common Language Infrastructure
Filename extensions	.py, .pyc, .pyo (prior to 3.5), .pyw, .pyz (since 3.5)	.c, .h	.java, .class, .jar	.cc, .cpp, .C, c++, .h, .hh, .hpp, .hxx, .h++	.cs

- 6-10

	 R	 JavaScript	 PHP	 Go	 Swift
Paradigm	Multi-paradigm: array, object-oriented, imperative, functional, procedural, reflective	Multi-paradigm: object-oriented (prototype-based), imperative, functional, event-driven	Imperative, object-oriented, procedural, reflective	Compiled, concurrent, imperative, structured	Multi-paradigm: protocol-oriented, object-oriented, functional, imperative, block-structured
Designed by	Ross Ihaka and Robert Gentleman	Brendan Eich	Rasmus Lerdorf	Robert Griesemer, Rob Pike, Ken Thompson	Chris Lattner and Apple Inc
Developer	R Core Team	Netscape Communications Corporation, Mozilla Foundation, Ecma International	The PHP Development Team, Zend Technologies	Google Inc.	Apple Inc
First appeared	August 1993 (24 years ago)	December 4, 1995 (21 years ago)	June 8, 1995 (22 years ago)	November 10, 2009 (7 years ago)	June 2, 2014 (3 years ago)
Typing discipline	Dynamic	Dynamic, duck	Dynamic, weak, gradual (as for PHP 7.0.0)	Strong, static, inferred, structural	Static, strong, inferred
Platform	UNIX platforms, Windows, MacOS	Cross-platform	Unix-like, Windows	Linux, macOS, FreeBSD, NetBSD, OpenBSD, Windows, Plan 9, DragonFly BSD, Solaris	Darwin, Linux, FreeBSD
Filename extensions	.r, .R, .RData, .rds, .rda	.js	.php, .phtml, .php3, .php4, .php5, .php7, .phps	.go	.swift

Hackernoon.com

Programmiersprache in IPI: C++

Warum C++ ?

- C++ ist eine ausgereifte Sprache, die für sehr viele große Softwareprojekte verwendet und ständig weiterentwickelt wird.
- Objektorientierung: Modellierung komplexer Daten ist wichtiger als der Kontrollfluss.
- C++ erlaubt sowohl maschinennahe Programmierung als auch hohes Abstraktionsniveau.
- C++ ist sehr effizient
- Compiler sind frei verfügbar und in jeder Linux-Distribution vorinstalliert
- Sehr flexibel und erweiterbar

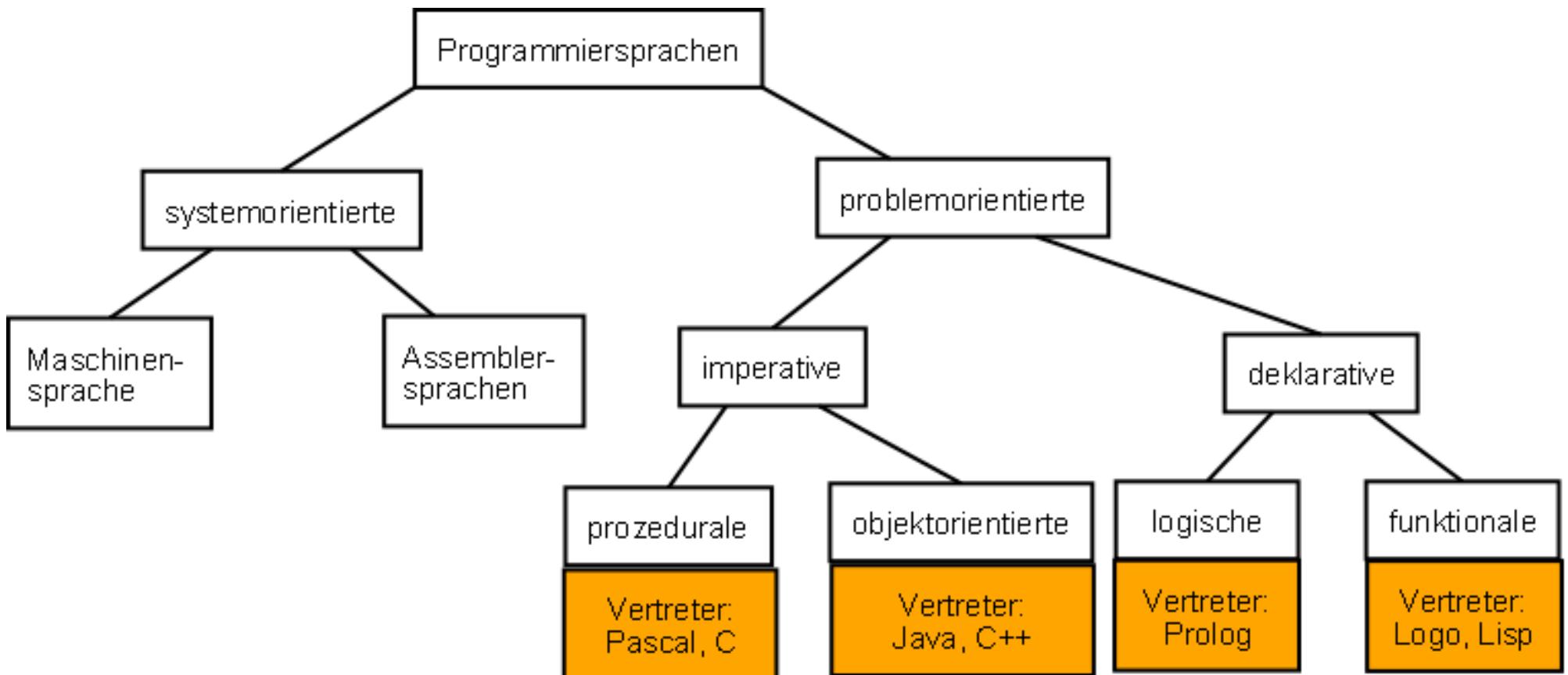
C++ ist sehr komplex, daher werden wir uns auf einen Ausschnitt beschränken.
Vollständigkeit ist kein Ziel dieser Vorlesung!

Warum gibt es verschiedene Programmiersprachen ?

Wie bei der Umgangssprache: teils sind Unterschiede historisch gewachsen, teils sind die Sprachen wie Fachsprachen auf verschiedene Problemstellungen hin optimiert.

Andererseits sind die Grundkonzepte in vielen prozeduralen bzw. objektorientierten Sprachen sehr ähnlich. Eine neue Sprache in dieser Klasse kann relativ leicht erlernt werden.

Programmiersprachen und -paradigmen



Programmiersprachen und -paradigmen

	Imperative/prozedurale Sprachen	Deklarative/prädiktive Sprachen
Merkmal	<ul style="list-style-type: none">• Benutzer legt durch die Sprache fest, wie das Problem gelöst werden soll.• Der Lösungsweg wird als eine Folge von Einzelschritten aufgeschrieben.• Die Schritte bestehen aus Anweisungen, die aus den algorithmischen Grundstrukturen Sequenz, Bedingungen und Schleifen zu komplizierteren Verbundanweisungen verknüpft werden.	<ul style="list-style-type: none">• Der Benutzer beschreibt, was das Problem ist, indem er bekanntes Wissen angibt.• An das System werden Anfragen gestellt, die entsprechend mit den Lösungsmechanismen Unifikation, Resolution und Backtracking abgearbeitet werden.• Es findet ein auf logische Schlüsse basierendes Beweisen in einem System von Tatsachen und Schlussfolgerungen statt.
Beispiel: Beschreibung eines Kreises	<ol style="list-style-type: none">1. Leg den Mittelpunkt fest2. Rotiere mit dem Zirkel um 360° um den Mittelpunkt mit der gegebenen Spanweite (Radius)	<ul style="list-style-type: none">• Menge aller Punkte, die zu einem vorgebenden Punkt den gleichen Abstand haben
typische Vertreter	ALGOL, FORTRAN, PASCAL, BASIC, Delphi (Object PASCAL), C++, Java, Oberon	LISP, PROLOG

Vom Programm zum maschinenlesbaren Programm

Programme in einer Hochsprache lassen sich *automatisch* in Programme der Maschinensprache übersetzen. Ein Programm, das dies tut, nennt man **Übersetzer** oder **Compiler**.

Ein Vorteil dieses Vorgehens ist auch, dass Programme der Hochsprache in verschiedene Maschinensprachen (**Portabilität**) übersetzt und andererseits verschiedene Hochsprachen auch in ein und dieselbe Maschinensprache übersetzt werden können (**Flexibilität**).

Es gibt auch sog. interpretierte Sprachen. Dort werden die Anweisungen der Hochsprache während der Ausführung „on the fly“ in Maschinensprache übersetzt. Beispiele: Python, Shell, Basic.

Schließlich gibt es Mischformen, bei denen von der Hochsprache in eine Zwischensprache übersetzt wird, die dann interpretiert wird. Beispiel: Java.

Kompilieren und Linken

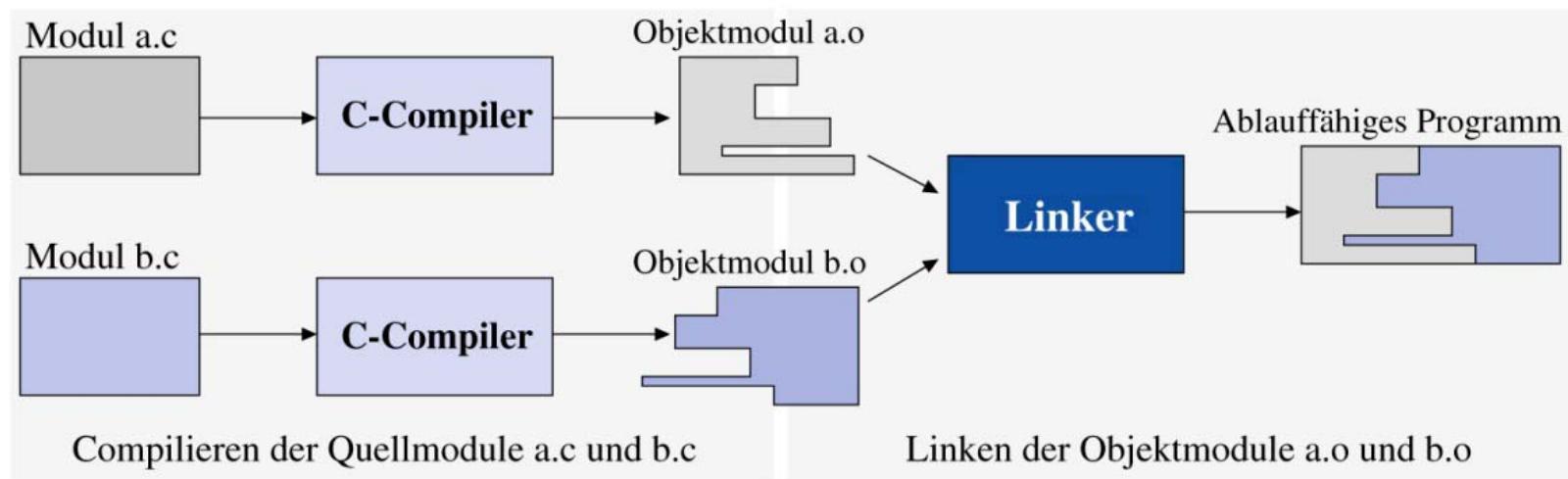


Abbildung 6.3: Compilieren und Linken von Modulen

Herold, Lurz, Wohlrab - Grundlagen der Informatik

Linken von eigenen Programmen mit Bibliotheken

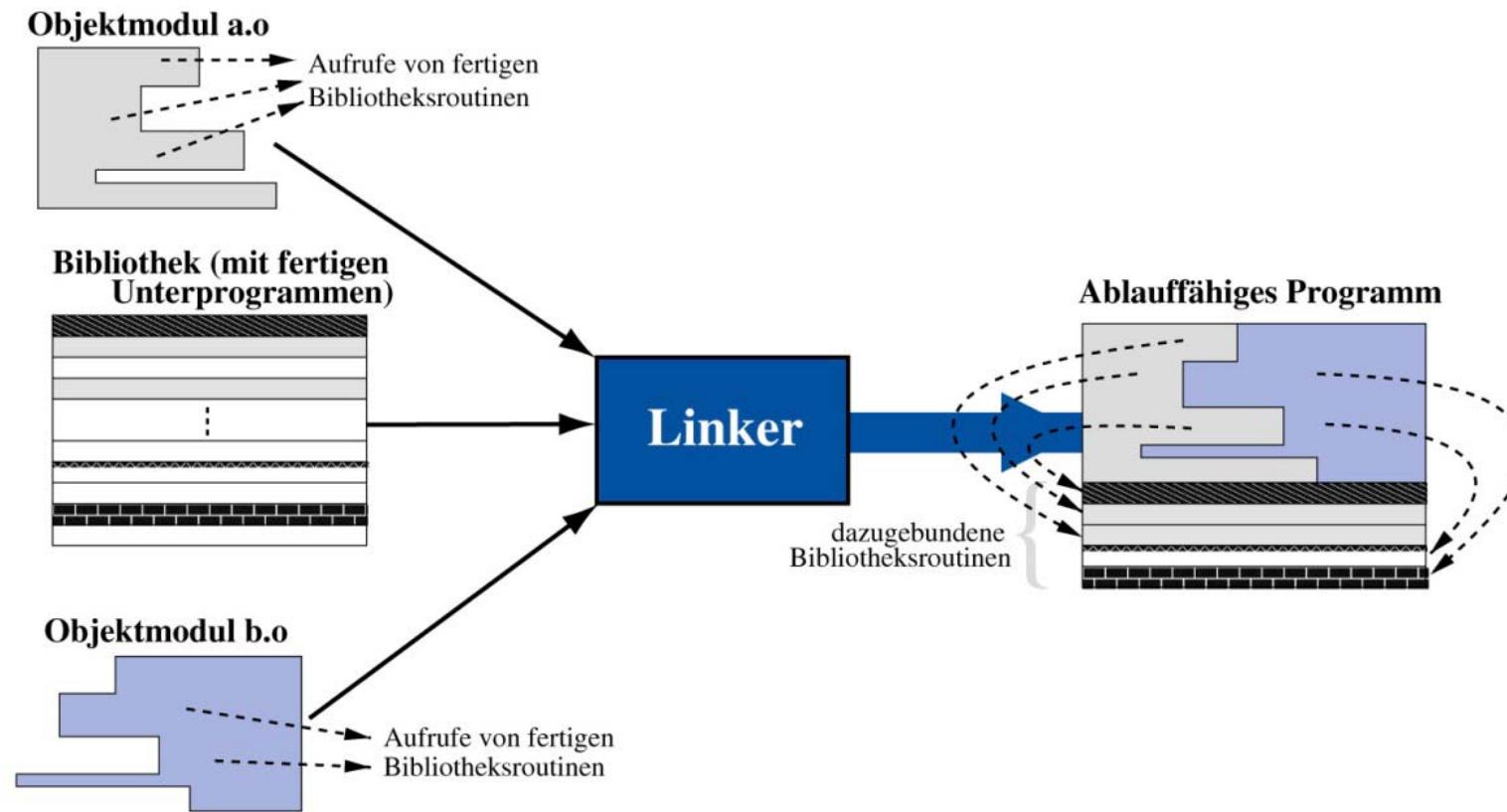
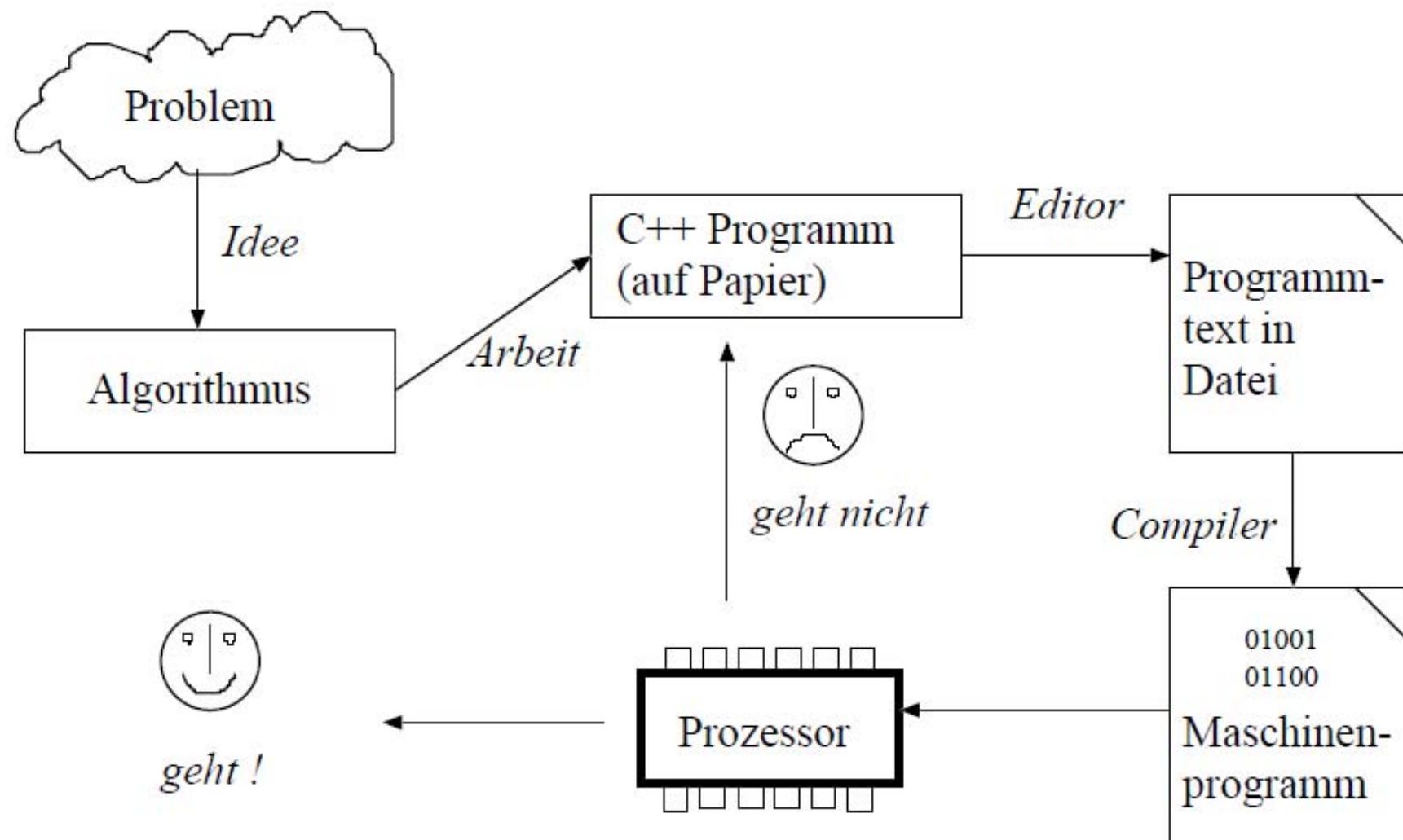


Abbildung 6.4: Linken der beiden Objektmodule a.o und b.o mit Hinzubinden von Bibliotheks routinen

Schritte der Programmierung



Die einzelnen Schritte bei der Programmerstellung im Überblick.

Grundbegriffe

- Formale Systeme & Sprachen
- Bäume und Graphen
- Automatentheorie, Turingmaschine

- Algorithmen
- Berechenbarkeit
- Reale Computer

- Programmiersprachen
- Grenzen der Harwareentwicklung

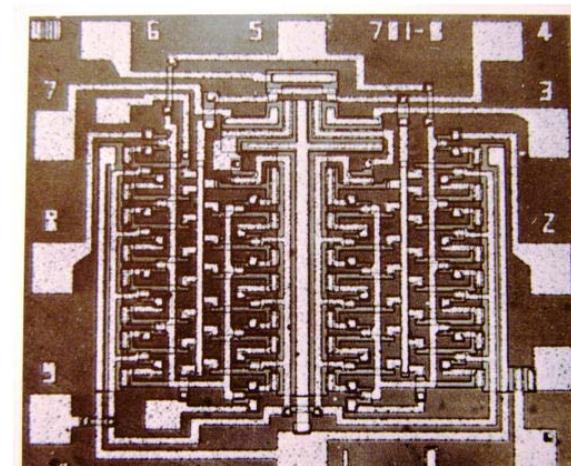
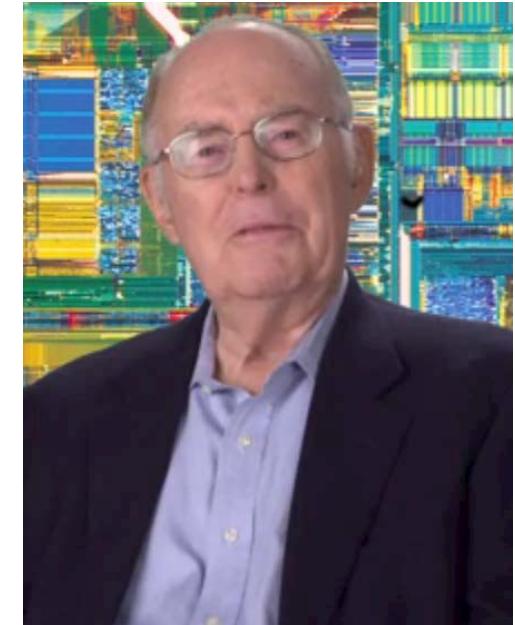


Grenzen der Harwareentwicklung

- Komplexität von Programmen wächst schnell
- Aber auch die der Hardware

Moore's law

- Wer ist Moore?
Gordon Moore (*1929 in San Francisco)
Mitbegründer von Fairchild Semiconductors (1957)
Mitbegründer von Intel (1968)
 - Moore's law ist kein richtiges Gesetz im naturwissenschaftlichen und auch nicht im juristischen Sinne, sondern eine Beobachtung, die Moore im Rahmen eines Papers in der Zeitschrift "Electronics" im Jahr 1965 veröffentlicht hat, kurze Zeit nach der Erfindung der integrierten Schaltkreise.



Moore's law

- Moore's law beschreibt die Beobachtung, dass sich die Komplexität integrierter Schaltkreise (=Zahl der Komponenten pro Schaltkreis) jedes Jahr verdoppelt, und prognostizierte, dass dies noch mindestens ein Jahrzehnt so weitergeht
- Moore korrigierte 1975 den Verdopplungs-Zeitraum für die nächste Dekade auf 2 Jahre
- Bezeichnung "Moore's law" stammt von Carve Mead, 1970

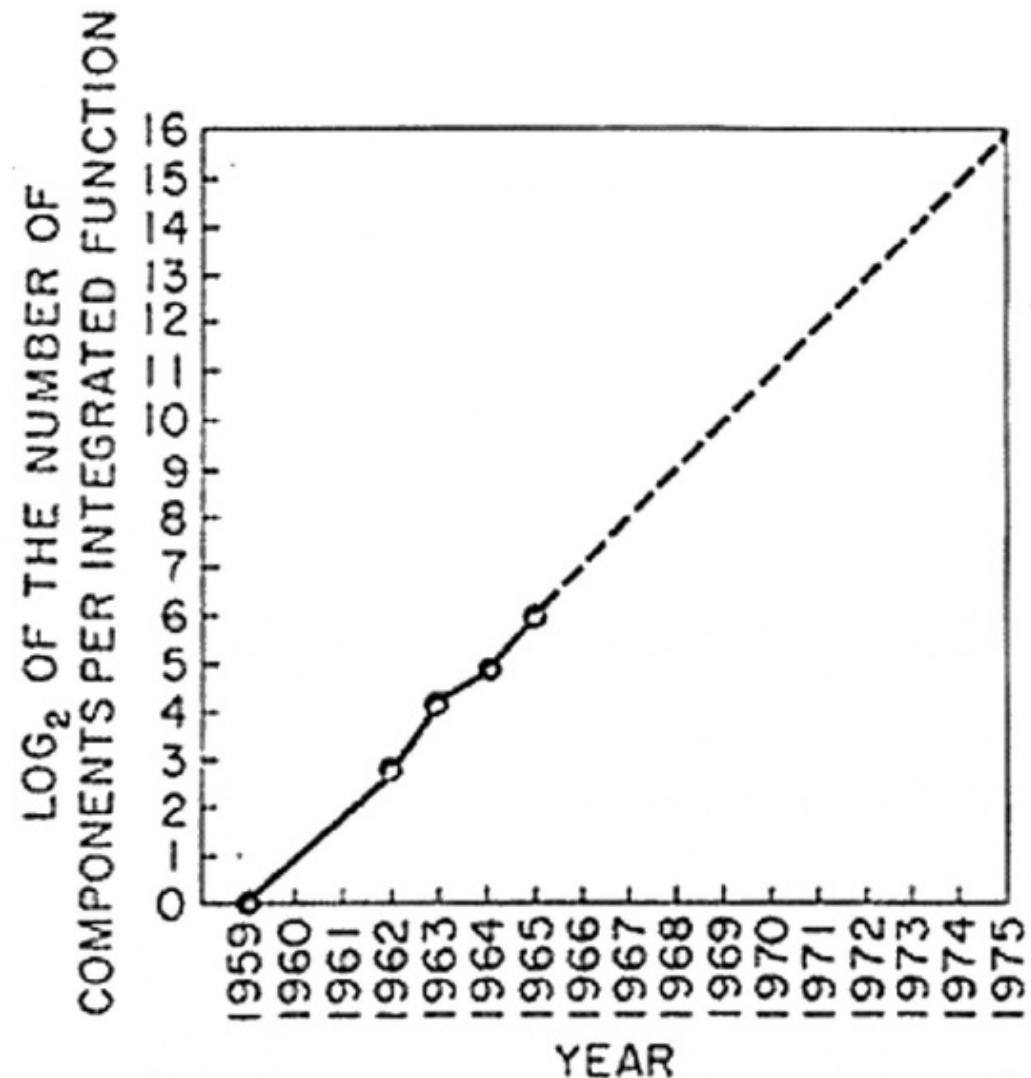
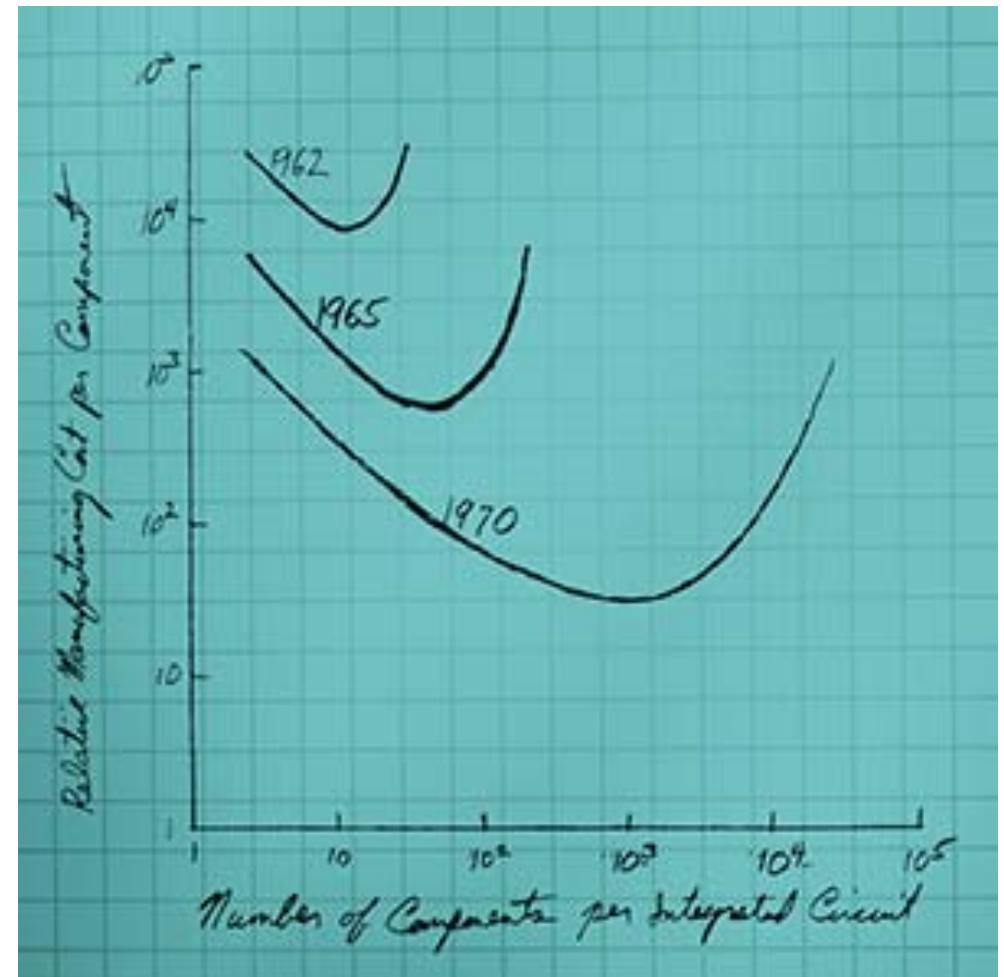


Fig. 2 Number of components per integrated function for minimum cost per component extrapolated vs time.

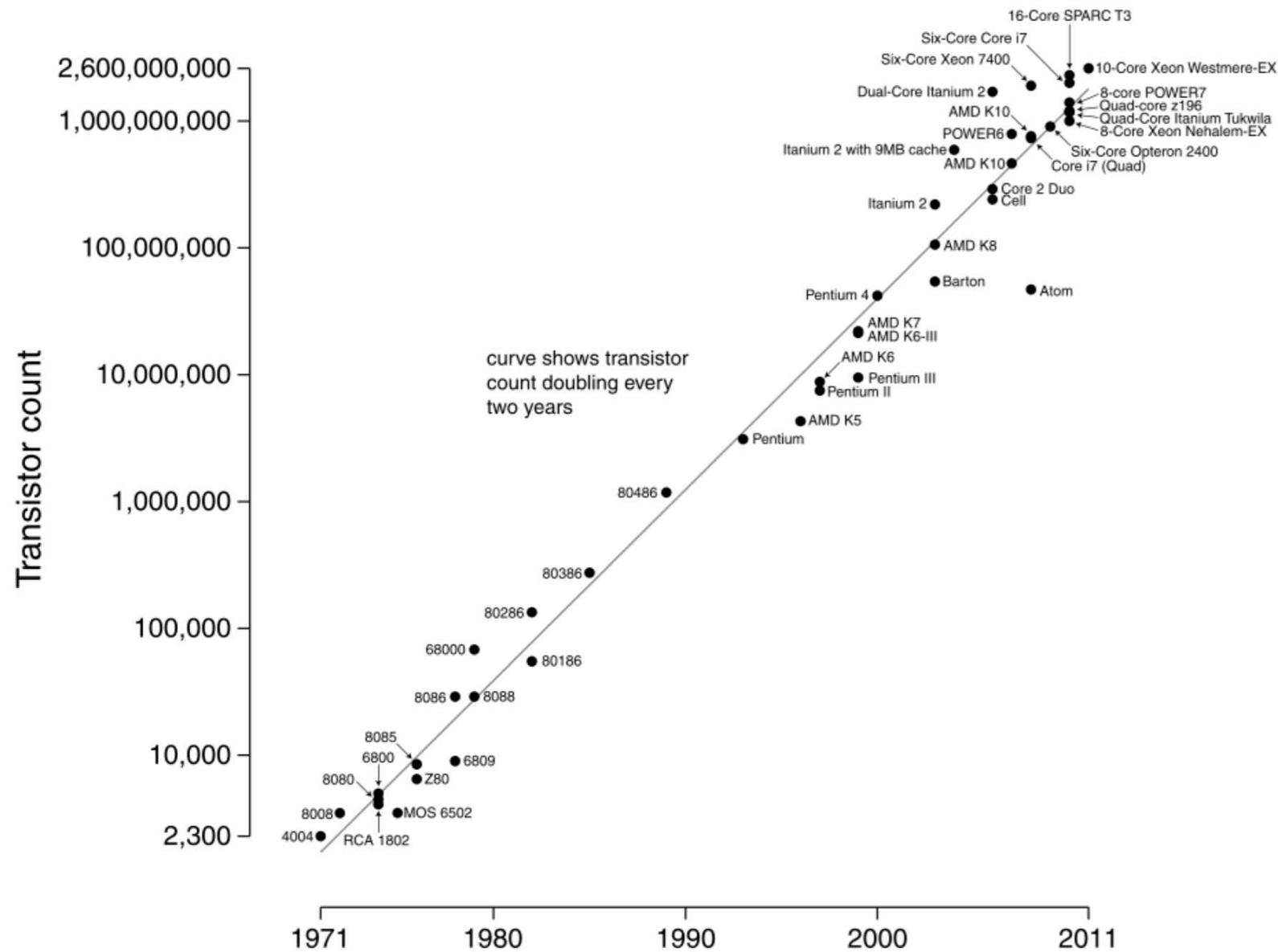
Moore's law

Tatsächlich ging es in seinem Paper auch sehr um wirtschaftliche Aspekte / Herstellungskosten:

- Für jede Generation von Produktionstechniken gibt es eine spezifische Kostenkurve: Herstellungskosten sind mit höherer Packdichte bis zu spezifischen Punkt (sweet spot), danach steigen sie
- Dieser sweet Spot bewegt sich mit der Zeit zu höhere Packdichte = immer komplexeren ICs



Microprocessor Transistor Counts 1971-2011 & Moore's Law

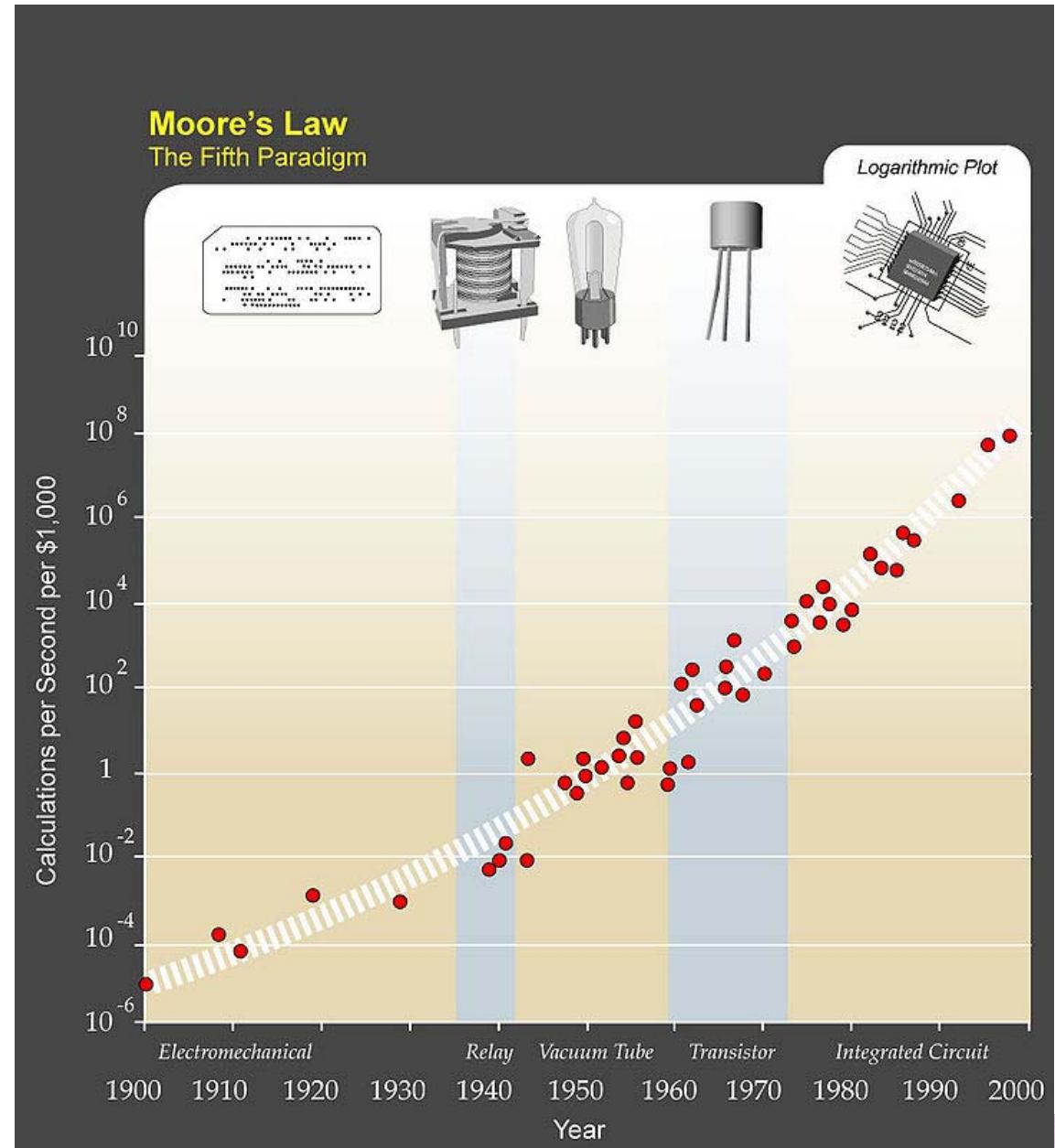


Moore's law

- Ähnliche Voraussage von David House (ebfalls Intel):
 - Chip Performance verdoppelt sich alle 18 Monate
 - ist eine Kombination der Effekte einer größeren Prozessorenzahl und höherer Leistungsfähigkeit der einzelnen Prozessoren
- Diese Aussage wird oft mit Moore's law verwechselt bzw. kombiniert
- Moore's law wird sehr oft sehr frei interpretiert und "zitiert"
- Moore's law wurde zu einem weithin akzeptierten Ziel der Halbleiterhersteller, da die Technologieentwicklung stark angetrieben hat und damit einer Art selbst-erfüllenden Prophezeiung,

Moore's law

- viele ähnliche Beobachtungen werden ebenfalls als "Moore's law" bezeichnet, sind aber nicht das, was er eigentlich gesagt hat



Nicht verwechseln mit Murphy's law!

- Murphys Gesetz:

“Anything that can go wrong *will* go wrong”

bzw. in der Urfassung:

“If there’s more than one possible outcome of a job or task, and one of those outcomes will result in disaster or an undesirable consequence, then somebody will do it that way.”
- geht auf E. A. Murphy von der Air Force zurück, der 1949 an einem Testprogramm für Raketenwagen teilnahm, bei dem die Beschleunigungen am menschlichen Körper und dessen Belastungsfähigkeit vermessen werden sollten. Für die Anbringung der 16 Messsensoren gab es 2 Möglichkeiten, eine richtige und eine falsche. Alle 16 Sensoren wurden falsch angebracht und das teure Experiment brachte damit kein Ergebnis!
- Moore fand seine Beobachtung dagegen sehr überraschend, weil sie so optimistisch war: "Moore's law is a violation of Murphy's law. Everything gets better and better."

The End of Moore's Law

Intelligent Machines

Moore's Law Is Dead. Now What?

Shrinking transistors have powered 50 years of advances in computing—but now other ways must be found to make computers more capable.

by Tom Simonite May 13, 2016



TECH

End of Moore's Law: It's not just about physics



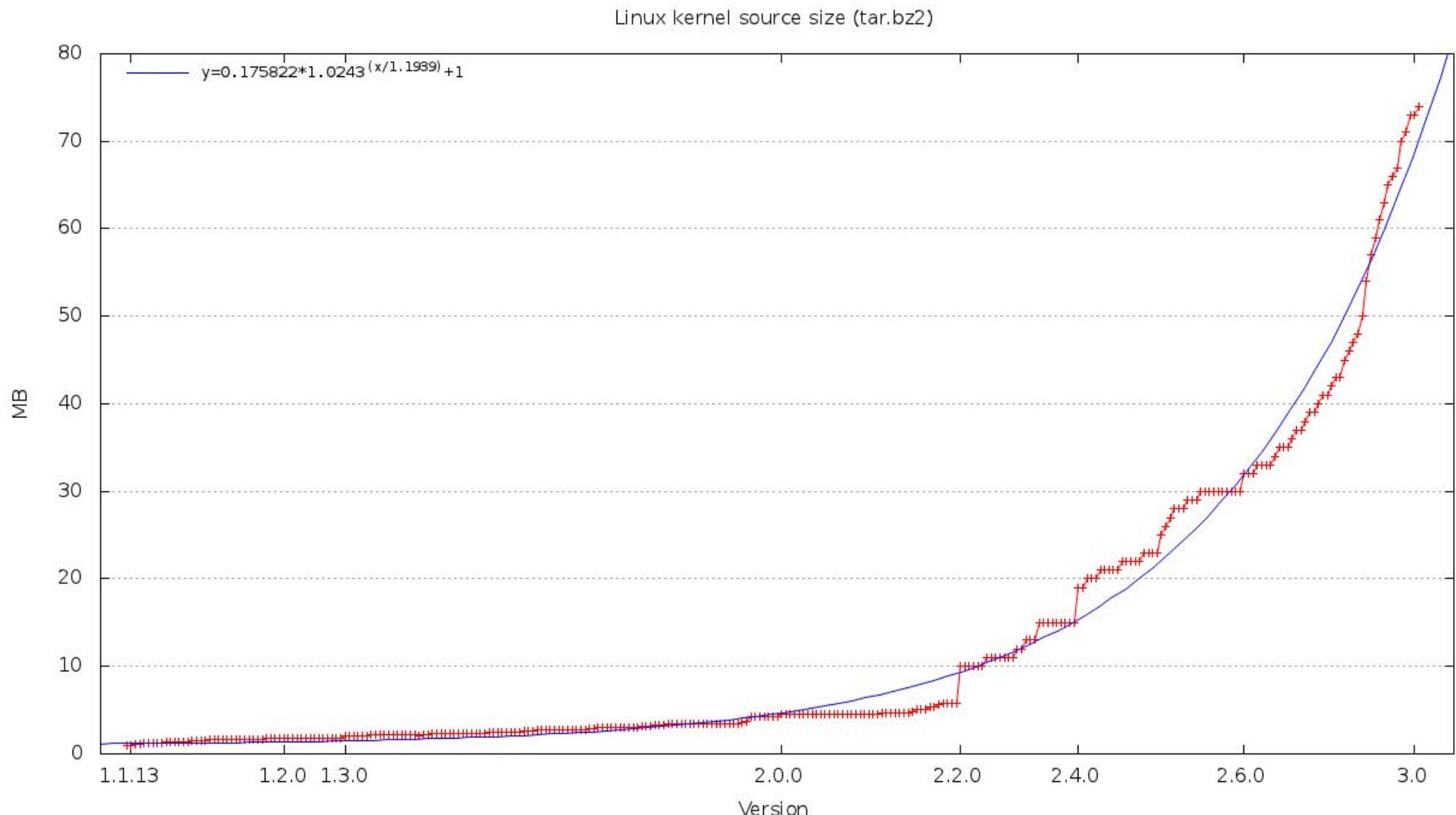
[The End of Moore's Law](#)

JUL 26, 2016 @ 10:28 AM 4,911 

Economics Is Important - The End Of Moore's Law

Beispiel: Entwicklung von Taktgeschwindigkeit, Speichergröße und Größe des Linux-Kernel.

Zeit	Proz	Takt	RAM	Disk	Linux Kernel (.tar.gz)
1982	Z80	6	64KB	800KB	6KB (CPM)
1988	80286	10	1MB	20MB	20KB (DOS)
1992	80486	25	20MB	160MB	140KB (0.95)
1995	PII	100	128MB	2GB	2.4MB (1.3.0)
1999	PII	400	512MB	10GB	13.2MB (2.3.0)
2001	PIII	850	512MB	32GB	23.2MB (2.4.0)
2004	P4 (Prescott)	3.8 GHz	2048 MB	250 GB	36 MB (2.4.26)
2010	i7 (Westmere)	3.5 GHz	8196 MB	1024 GB	84 MB (2.6.37.7)



Grenzen der Hardwareentwicklung:

- Je höher die Takfrquenz desto höher die Wärmeentwicklung.
- Mehr an Transistoren wird in viele unabhängige Cores gesteckt.
- Erfordert **parallele Programmierung**.

Grenzen der Softwareentwicklung:

- Die benötigte Zeit zum Erstellen großer Programme **skaliert** mehr als linear, d. h. zum Erstellen eines doppelt so großen Programmes braucht man mehr als doppelt so lange.
- Verbesserte Programmiertechnik, Sprachen und Softwareentwurfsprozesse. Einen wesentlichen Beitrag leistet hier die **objektorientierte Programmierung**, die wir in dieser Vorlesung am Beispiel von C++ erlernen werden.