

# Imperative Programmierung (IPR)

## Kapitel 1: Einführung

**Univ.-Prof. Dr.-Ing. habil. Gero Mühl**

Lehrstuhl für Architektur von Anwendungssystemen (AVA)  
Fakultät für Informatik und Elektrotechnik (IEF)  
Universität Rostock

Universität  
Rostock



Traditio et Innovatio



# Inhalte

1. Was ist Informatik?
2. Was ist ein Algorithmus?
3. Algorithmen und Funktionen
4. Was ist eine Programmiersprache?
5. Geschichte der Programmiersprachen

## Kapitel 1.1

# Was ist Informatik?

# Definition und Ursprung des Begriff Informatik

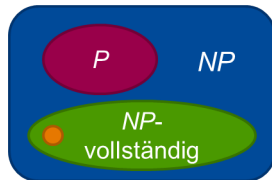
- **Informatik** ist die Wissenschaft der systematischen Verarbeitung von Informationen.
- Der Begriff Informatik entsteht durch Verschmelzen von INFORMATION und matheMATIK.
- Ursprung der Informatik sind (im Wesentlichen) die beiden Wissenschaften Mathematik und Elektrotechnik.
- Informatik als Begriff ist vor allem in Kontinentaleuropa gebräuchlich, hingegen **Computer Science** in den angelsächsischen Staaten.
- „*Computer science is no more about computers than astronomy is about telescopes.*“ (Edsger Dijkstra)

# Wo ist überall Informatik drin?



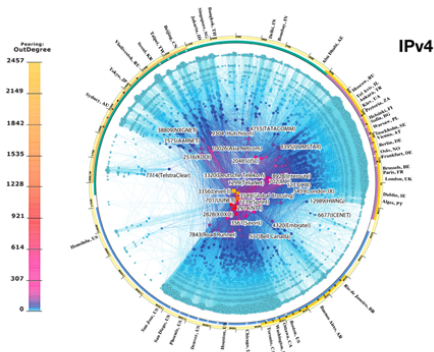
# Informatik als Grundlagenwissenschaft

- Informatik ist wie die Mathematik eine auf andere Wissensgebiete ausstrahlende Grundlagen- und Formalwissenschaft.
- Die Mathematik kann als die Wissenschaft des „formal Denkbaren“ gesehen werden.
- Die Informatik konzentriert sich hingegen auf das Realisierbare, das der maschinellen Verarbeitung zugänglich ist.
  - Programmiersprachen und ihre Semantik
  - Logiken, Kalküle und Beweisverfahren
  - Automaten, Schaltwerke und Maschinenmodelle
  - Datenstrukturen, Datentypen und Objekte
  - Algorithmen und ihre Komplexität
  - Programme und Prozesse
  - Naturanaloge Verfahren und Heuristiken
  - Sicherheit, Korrektheit und Zuverlässigkeit



# Informatik als Ingenieurswissenschaft

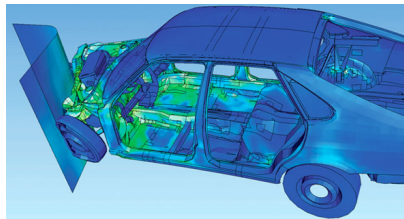
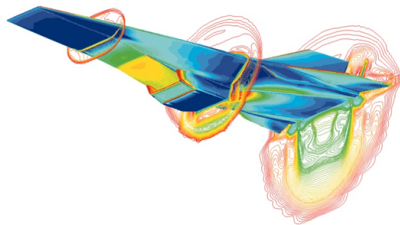
- Informatik ist vor allem ein Ingenieurswissenschaft, die sich mit Entwurf, Implementierung und Einsatz von Informationssystemen für diverse Anwendungsgebiete befasst.
  - Rechnerarchitektur und Chipentwurf
  - Integrierte Hardware-Softwaresysteme
  - Betriebssysteme und Verteilte Systeme
  - Rechner- und Kommunikationsnetze
  - Datenbanken und Informationssysteme
  - Eingebettete Systeme und Echtzeitsysteme
  - Computergrafik



copyright © 2009 UC Regents. all rights reserved.

# Informatik als Experimentalwissenschaft

- Informatik besitzt auch Aspekte einer Experimentalwissenschaft
  - Experimente in einem virtuellen Labor
  - Modellierung und Simulation von Szenarien, die sich physischen Experimenten verschließen (z.B. Störfälle in Atomkraftwerken)
- Teilaspekte
  - Modellierungs- und Simulationsmethodik
  - Parallele Algorithmen
  - Datenanalyse
  - Höchstleistungsrechnen
  - Wissensbasierte Systeme
  - Bildverarbeitung und -erkennung





# Teilgebiete der Informatik

## ■ Theoretische Informatik

- Formale Sprachen, Automatentheorie, Logik, Berechenbarkeit, Komplexitätstheorie, Verifikation, Algorithmik, Kryptographie, etc.

## ■ Praktische Informatik

- Softwaretechnik, Verteilte Systeme, Betriebssysteme, Datenbanken, Informationssysteme, Computergraphik, etc.

## ■ Technische Informatik

- Integrierte Schaltungen, Eingebettete Systeme, Netzwerke, Robotik, Echtzeitsysteme, Rechnerarchitekturen, etc.

## ■ Angewandte Informatik

- Anwendung von Informationsverarbeitung in Unternehmen, Medien Verwaltung, Fertigung, Medizin, Biologie, Visualisierung, ...



## Kapitel 1.2

# Was ist ein Algorithmus?

# Was ist ein Algorithmus?

- Folgende Definition des Begriffs Algorithmus ist die Basis für die weiteren Betrachtungen in dieser Vorlesung:

## Definition 1 (Algorithmus)

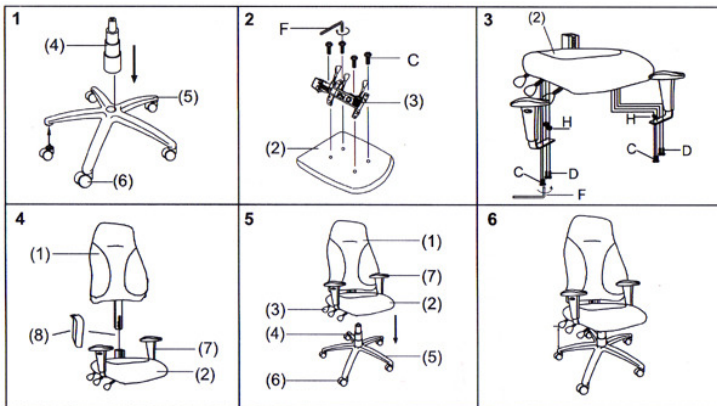
*Ein **Algorithmus** ist eine aus endlich vielen, wohldefinierten und effektiven Einzelschritten bestehende Handlungsvorschrift zur Lösung eines Problems oder einer Klasse von Problemen.*

- Direkt aus der obigen Definition lassen sich die folgenden Anforderungen an Algorithmen ableiten:
  - Finithheit:** Die Beschreibung muss eine endliche Länge haben.
  - Definierttheit:** Jeder Einzelschritt muss eindeutig definiert sein.
  - Effektivität:** Jeder Einzelschritt muss direkt ausführbar sein.

# Beispiele für Algorithmen

- Aus dem Alltag
  - Kochrezepte
  - Bedienungsanleitungen
  - Hilfen zum Ausfüllen von Formularen
  - ...
- Aus der Schulzeit
  - Schriftliches Addieren, Subtrahieren, Multiplizieren und Dividieren
  - Bestimmung der Nullstellen einer quadratischen Funktion
  - Differenzieren und Integrieren von Polynomen
  - ...
- Grundlegende Algorithmen aus der Informatik
  - Suchen
  - Sortieren
  - Bestimmung kürzester Wege in Graphen
  - ...

# Beispiel: Bauanleitung



Quelle: IKEA

# Beispiel: Rezept für einen Caipirinha

## ■ Zutaten

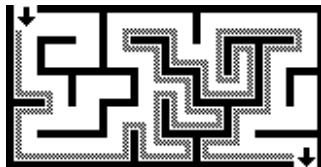
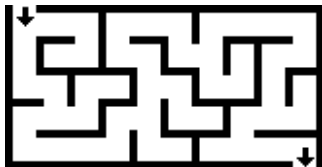
- 3 cl Zuckerrohrschnaps (Cachaca)
- Eine Limette
- Zwei Teelöffel brauner Zucker
- gecrushtes Eis

## ■ Zubereitung

- Limette in Achtelstücke schneiden, in ein Longdrinkglas geben und den braunen Zucker darüber streuen.
- Limettenstücke mit einem Holzmörser gut zerdrücken und mit dem Zucker vermischen.
- Anschließend den Zuckerrohrschnaps drübergießen und das Glas mit gecrushtem Eis auffüllen.
- Gut mischen und mit einem oder zwei Trinkhalmen servieren.



## Beispiel: Den Weg durch ein Labyrinth finden



### Regel

- Gehe immer so, dass Du mit der Rechten Hand die Wand berührst!

# Beispiel: Schriftliche Addition von Dezimalzahlen

- 1 Schreibe die Zahlen rechtsbündig so hin, dass jeweils zwei Ziffern in Spalten untereinander stehen.
- 2 Beginne mit der Spalte, die am weitesten rechts ist.
- 3 Addiere die Ziffern in dieser Spalte.
- 4 Schreibe die Einerstelle der Summe unter die Spalte.
- 5 Schreibe die Zehnerstelle der Summe über die Spalte links der aktuellen.
- 6 Wiederhole ab Schritt 3 mit nächster Spalte links der aktuellen, bis keine Spalte mehr übrig ist.
- 7 Die unterste Zeile ist das Ergebnis.

■ Beispiel: Addition von 782 und 341:

$$\begin{array}{r} \phantom{0}1 \\ 7 \phantom{0} 8 \phantom{0} 2 \\ + 3 \phantom{0} 4 \phantom{0} 1 \\ \hline 1 \phantom{0} 1 \phantom{0} 2 \phantom{0} 3 \\ \hline \hline \end{array}$$



## Beispiel: Algorithmus zum Verdoppeln aus dem Jahr 1574

# Dupliren

**S**chreibe wie du ein zahl zweyfaltigen solt. Thu ihm also: Schreibe die zahl vor dich/mach ein Linien darunder/hebe an zu forderst/Duplir die erste Figur. Kompt ein zahl die du mit einer Figur schreiben magst/so setz die vnden. Wo mit zweyen/schreibe die erste/Die ander behalt im sinn. Darnach duplir die ander/vnd gib darzu/das du behalten hast/vnd schreib abermals die erste Figur/wo zwei vorhanden/vnd duplir fort biß zur letzten/die schreibe ganz auß/als folgende Exempel aufzuweisen.

$$\begin{array}{r} 41232 \quad 98765 \quad 68704 \\ \hline 82464 \quad 197530 \quad 137408 \\ \quad \quad \quad \text{iii} \quad \text{Prob.} \end{array}$$

- Verdoppeln nach Adam Riese. Aus: A. Risen, Rechenbuch, Frankfurt 1574. Faksimile-Druck Satyr-Verlag, Brensbach/Odw, 1978

# Beispiel: Berechnung des größten gemeinsamen Teilers (ggT)

## Originaler Euklidischer Algorithmus

Eingabe: zwei (positive) natürliche Zahlen  $a$  und  $b$ .

Ausgabe: eine natürliche Zahl (der ggT von  $a$  und  $b$ ).

- ①  $x := a$
  - ②  $y := b$
  - ③ Wenn  $x > y$ , dann  $x := x - y$ , sonst  $y := y - x$ .
  - ④ Wenn  $y \neq 0$ , dann weiter mit Schritt 3.
  - ⑤ Gib den Wert von  $x$  zurück.
- Der ggT wird z. B. benötigt, um Brüche zu kürzen. In diesem Fall werden Zähler  $Z$  und Nenner  $N$  durch  $\text{ggT}(Z, N)$  geteilt.
  - Allerdings ist die oben angegebene originale Version des euklidischen Algorithmus für stark unterschiedlich große Zahlen ineffizient.

# Beispiel: Effizientere Berechnung des ggTs

## Verbesserter Euklidischer Algorithmus

Eingabe: zwei (positive) natürliche Zahlen  $a$  und  $b$ .

Ausgabe: eine natürliche Zahl (der ggT von  $a$  und  $b$ ).

- ①  $x := a$
- ②  $y := b$
- ③  $r := x \bmod y$
- ④  $x := y$
- ⑤  $y := r$
- ⑥ Wenn  $y \neq 0$ , dann weiter mit Anweisung 3.
- ⑦ Gib den Wert von  $x$  zurück.

### Beispiel 1 (Berechnung des GGT von 1071 und 1029)

$x$		$y$		$q$		$r$
1071	:	1029	=	1	Rest	42
1029	:	42	=	24	Rest	21
42	:	21	=	2	Rest	0
<b>21</b>		0				

- Hinweis: Gilt initial  $x < y$ , so vertauscht der erste Schleifendurchlauf lediglich die Werte von  $x$  und  $y$ .

# (Statische) Finitheit

- Algorithmen können auf verschiedene Weise beschrieben werden, zum Beispiel mit Hilfe von Texten oder auch Grafiken.
  - **Textuelle Beschreibung**
    - Natürliche Sprache
    - Pseudocode
    - ...
  - **Grafische Beschreibung**
    - Struktogramme (Nassi-Shneiderman)
    - Programmablaufpläne
    - ...
- Es muss stets gewährleistet sein, dass die Beschreibung eines Algorithmus eine endliche Länge hat.

# Definiertheit

- Jeder Schritt des Algorithmus muss eindeutig sein.
- Betrachte folgenden Algorithmus zur Übernahme von Microsoft:

## Übernahme von Microsoft

- 1 Gehe nach New York.
  - 2 Mache eine Million Dollar.
  - 3 Spekuliere an der Börse mit der Million bis Du 2.500 Milliarden Dollar zusammen hast.
  - 4 Kaufe alle Aktien von Microsoft auf.
- Offensichtlich sind die Anweisungen 2 und 3 nicht eindeutig definiert und wahrscheinlich auch nicht eindeutig definierbar.

# Ausführbarkeit

- Jeder Teil eines Algorithmus sollte direkt oder mit Hilfe eines anderen Algorithmus ausführbar sein.
- Test: Prinzipiell muss alles mit Stift und Papier nachvollziehbar sein.
- Betrachte folgenden Algorithmus zu Primzahlzwillingen:

- 1 Lese zwei Zahlen  $x_1$  und  $x_2$  ein.
- 2 Wenn  $x_1$  keine Primzahl ist oder wenn  $x_2$  keine Primzahl ist, dann gebe „Keine zwei Primzahlen“ aus und gehe zu Schritt 5.
- 3 Wenn  $\|x_1 - x_2\| \neq 2$ , dann gebe „Keine Primzahlzwillinge“ aus und gehe zu Schritt 5.
- 4 Wenn  $x_1$  und  $x_2$  die größten Primzahlzwillinge sind, gebe „größte Primzahlzwillinge“ aus, sonst gebe „Primzahlzwillinge“ aus.
- 5 Beende Algorithmus.

- Alle Schritte sind exakt definiert, aber es bis heute nicht bekannt, **ob** es ein größtes Primzahlzwillingspaar gibt oder **wie** man dieses ggf. bestimmt
- Der Algorithmus ist also (zumindest bis auf weiteres) nicht ausführbar.

# Weitere Eigenschaften von Algorithmen

**Abstrahierung:** Der Algorithmus soll nicht nur ein Problem lösen, sondern eine Klasse von Problemen.

**Terminierung:** Für *jede gültige* Eingabe ist der Algorithmus nach einer endlichen Anzahl von Schritten beendet.

**Effizienz:** Der Algorithmus soll möglichst wenige Schritte bis zur Ausgabe des Ergebnisses benötigen.

**Korrektheit:** Für *jede gültige* Eingabe soll der Algorithmus die gewünschte Ausgabe liefern.

**Dynamische Finitheit:** Der Algorithmus nutzt stets nur eine endliche Menge von Ressourcen

**Operative Finitheit:** Der Algorithmus darf nur endlich viele Elementaroperationen voraussetzen.

# Weitere Eigenschaften von Algorithmen

**Determinismus:** Zu jedem Zeitpunkt gibt es höchstens eine Möglichkeit der Fortsetzung. Der Algorithmus führt bei gleicher Eingabe also immer die gleichen Schritte aus.

**Determiniertheit:** Unter den gleichen Anfangsbedingungen gelten am Ende stets die gleichen Endbedingungen. Der Algorithmus liefert also bei gleicher Eingaben immer die gleiche Ausgabe.

## Merke

- Ein deterministischer Algorithmus ist stets determiniert.
- Die Umkehrung gilt jedoch im Allgemeinen nicht.
- Zum Beispiel gibt es randomisierte Algorithmen, die bei gleicher Eingabe auf verschiedenen Wegen stets die gleich Ausgabe liefern.



## Merke

- Zusätzlich zu den in der Definition genannten Eigenschaften verlangen wir von einem Algorithmus explizit die Eigenschaften Terminierung und Korrektheit sowie dynamische und operative Finitheit.
- Es gibt Algorithmen, die eine oder mehrere der anderen weiteren Eigenschaften *nicht* erfüllen. Wir betrachten aber hauptsächlich Algorithmen, die abstrahierend, deterministisch und determiniert sind.
- Außerdem interessieren uns vor allem Algorithmen, die auf einem *Computer* möglichst effizient ausgeführt werden können.

# Notation eines Algorithmus

- Algorithmen (und Programme) können auf verschiedene Weise dargestellt werden.
- Eine beliebte graphische Darstellung von Algorithmen sind die **Nassi-Shneiderman-Diagramme**, die 1972/73 von ISAAC NASSI und BEN SHNEIDERMAN entwickelt wurden.
- Sie erlauben die Darstellung von Programmentwürfen im Rahmen der Methode der **strukturierten Programmierung**.
- Da Nassi-Shneiderman-Diagramme Programmstrukturen darstellen, werden sie auch als **Struktogramme** bezeichnet.
- Die Vorgehensweise entspricht der **Top-Down-Programmierung**, bei der zunächst ein Gesamtkonzept entwickelt wird, welches dann verfeinert wird.
- Hierbei wird das Gesamtproblem in immer kleinere Teilprobleme zerlegt, bis schließlich nur noch elementare Grundstrukturen wie Sequenzen und Kontrollstrukturen übrig bleiben.

# Struktogramme

## ■ Einfache Anweisung

<Anweisung>

$z := k * m$

## ■ Anweisungsfolge

<Anweisung>

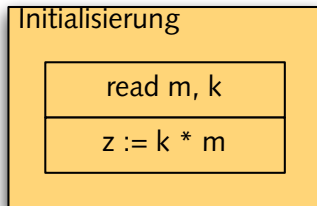
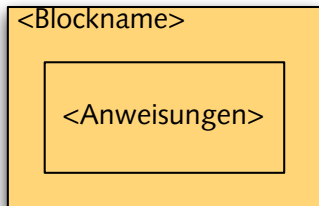
<Anweisung>

read m, k

$z := k * m$

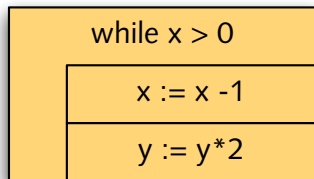
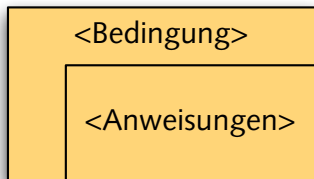
# Struktogramme

## ■ Benannter Block

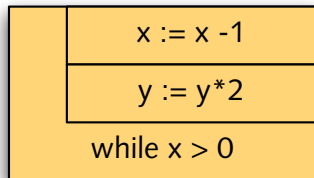
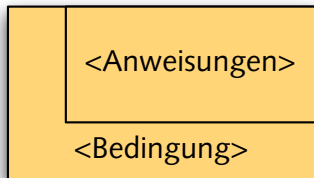


# Struktogramme

## ■ Wiederholung

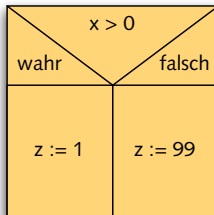
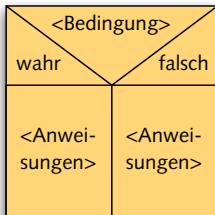


## ■ Wiederholung, mindestens einmal

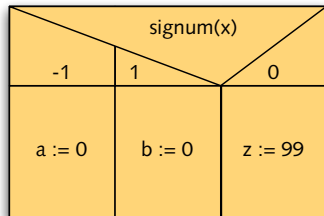
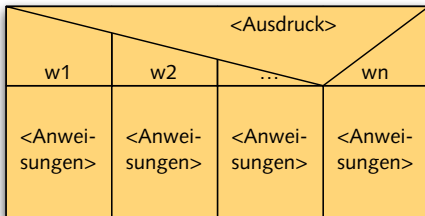


# Struktogramme

## ■ Fallunterscheidung (binär)

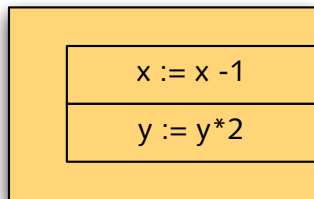
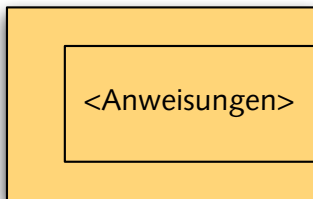


## ■ Fallunterscheidung ( $n$ -Weg)



# Struktogramme

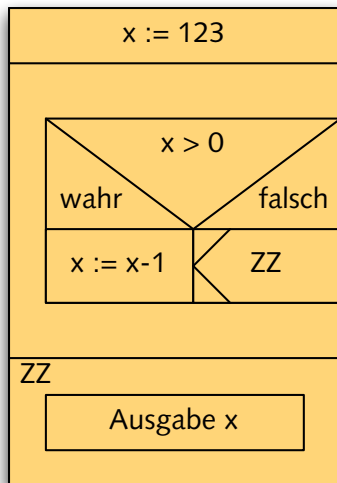
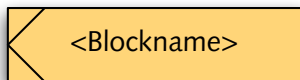
- Unbegrenzte Wiederholung



- Nur sinnvoll mit Abbruch-Konstrukt (siehe nächste Folie)!

# Struktogramme

## ■ Nichtlokaler Abbruch





# Entwurf von Algorithmen

- Algorithmen zu entwerfen ist teilweise sehr komplex.
- Leider gibt es auch keinen (exakten) Algorithmus, der beschreibt, wie beim Entwickeln von Algorithmen vorzugehen ist.
- Hilfreich sind in jedem Fall
  - die Fähigkeit Probleme analysieren zu können,
  - Abstraktionsvermögen,
  - Mathematikkenntnisse,
  - Kreativität und
  - Erfahrung.
- Jede dieser Eigenschaften kann durch Training teils erheblich verbessert werden.

## Kapitel 1.3

# Algorithmen und Funktionen

# Analogie zu mathematischen Funktionen

- Ein Algorithmus kann als **mathematische Abbildung** betrachtet werden, die die gültigen Eingaben als **Definitionsbereich** und die potentiell gelieferten Ausgaben als **Bildbereich** hat.
- Bei einem (determinierten) Algorithmus, entspricht dies dem Modell einer **mathematischen Funktion**

$$f : E \longrightarrow A$$

von der Menge der zulässigen Eingaben  $E$  in die Menge der zulässigen Ausgaben  $A$ .

- Für eine Eingabe  $e \in E$  liefert

$$a = f(e)$$

dann das Ergebnis des Algorithmus, wobei  $a \in A$  gilt.

# Analogie zu mathematischen Funktionen

- Durch die Äquivalenz mit einer Funktion wird ein Algorithmus zu einem **Berechnungsproblem**.

## Definition 2 (Berechnete Funktion)

*Ein Algorithmus  $P$  **berechnet eine Funktion**  $f : E \rightarrow A$ , wenn  $P$  bei Eingabe eines beliebigen  $e \in E$  nach einer endlichen Zahl von Schritten den Wert  $f(e)$  ausgibt und bei allen anderen Eingaben nicht terminiert.*

## Definition 3 (Berechenbare Funktionen)

*Eine Funktion heißt **berechenbar**, wenn es einen Algorithmus gibt, der sie berechnet.*

# Berechenbarkeit

- Lässt sich aber jede Funktion, die formal definiert werden kann, auch berechnen?
- Gibt es also für jede denkbare Funktion einen Algorithmus?
- Leider nein, da es viele Funktionen gibt, die **nicht berechenbar** sind. Hierzu gibt es z. B. folgende Aussagen:
  - **Äquivalenzproblem**: Es gibt keinen Algorithmus, der für zwei *beliebige* Algorithmen entscheiden kann, ob diese dieselbe Funktion berechnen
  - **Halteproblem**: Es gibt keinen Algorithmus, der berechnen kann, ob ein *beliebiger* anderer Algorithmus immer terminiert.
  - Hinweis: In beiden Fällen werden die zu untersuchenden Algorithmen geeignet als Eingabe des **Kontrollalgorithmus** kodiert.
- Es gibt sogar viel mehr nicht berechenbare Funktionen als berechenbare.

# Berechenbarkeit

- Daneben gibt es Funktionen, für die nicht bekannt ist, ob sie berechenbar sind.
- Es ist z. B. nicht bekannt, ob die folgende Funktion berechenbar ist:

$$f(n) = \begin{cases} 0, & \text{falls in } \pi \text{ die Ziffer 3 genau } n\text{-mal hintereinander auftritt} \\ 1, & \text{sonst} \end{cases}$$

- Solche Probleme der Grenzen der **Berechenbarkeit** werden in der theoretischen Informatik untersucht.
- Dort wird mit Hilfe von **Maschinenmodellen** (z. B. Turing-Maschine) die Mächtigkeit von Maschinen in Bezug auf die von ihnen berechenbaren Funktionen formal untersucht.
- Hierbei wird versucht, jeweils die *einfachste* Maschine für eine **Klasse berechenbarer Funktionen** zu verwenden.

# Die Turing-Maschine

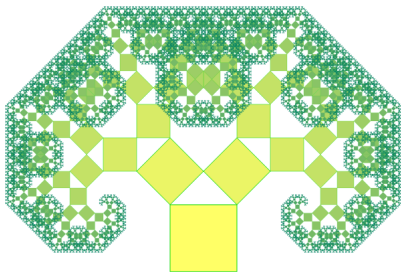
- Die Turing-Maschine ist eine einfache Maschine, mit der laut der (bisher unbewiesenen) **Church-Turing-These** *alle intuitiv berechenbaren* Funktionen berechnet werden können.
- Es gibt viele andere Maschinen, deren Mächtigkeit mit der der Turing-Maschine übereinstimmt → **Turing-Mächtigkeit**.
- Zu diesen gehören auch die gängigen Programmiersprachen, die eine weitaus komfortablere Programmentwicklung ermöglichen.
- Mit Hilfe des Begriffs der Turing-Maschine kann folgende alternative Definition des Begriffs Algorithmus formuliert werden:

## Definition 4 (Algorithmus)

*Eine Berechnungsvorschrift ist ein **Algorithmus**, wenn eine zu dieser Berechnungsvorschrift äquivalente Turing-Maschine existiert, die für genau jede gültige Eingabe stoppt und die korrekte Ausgabe liefert.*

# Rekursive Funktionen

- Als **Rekursion** (lat. recurrere „zurücklaufen“) wird die Technik bezeichnet, eine Funktion durch sich selbst zu definieren.
- Zentraler Bestandteil einer **rekursiven Definition** ist ihre **Selbstbezüglichkeit**.
- Werden mehrere Funktionen durch wechselseitige Verwendung voneinander definiert, so liegt **wechselseitige Rekursion** vor.
- Das Konzept der Rekursion ist nicht auf Funktionen beschränkt, sondern kann z. B. auch auf Punktmengen angewendet werden  
→ **Fraktale**



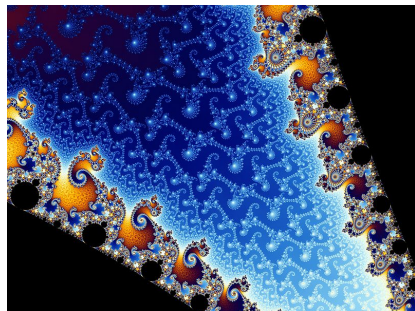
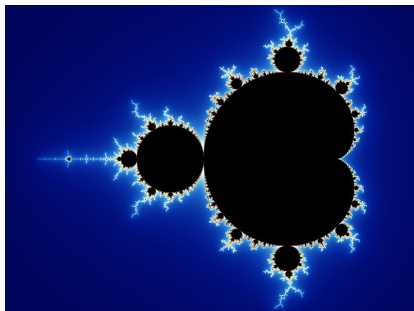
**Baum des Pythagoras**



# Bsp.: Mandelbrotmenge (nach Benoît Mandelbrot)

## Definition 5 (Mandelbrotmenge)

Die **Mandelbrot-Menge** ist die Menge aller komplexen Zahlen  $c$ , für welche die rekursiv definierte Folge komplexer Zahlen  $z_0, z_1, z_2, \dots$  mit dem Bildungsgesetz  $z_{n+1} = z_n^2 + c$  und dem Anfangsglied  $z_0 = 0$  beschränkt bleibt. Die Darstellung erfolgt meist in der komplexen Zahlenebene.



# Beispiele für rekursive Funktionen

## Beispiel 2 (Rekursive Definition der Fakultät)

$$fak(n) = \begin{cases} 1, & \text{falls } n = 0 \\ n \cdot fak(n-1), & \text{falls } n > 0 \end{cases}$$

$$fak(0) = 1$$

$$fak(1) = 1 \cdot fak(0) = 1$$

$$fak(2) = 2 \cdot fak(1) = 2$$

$$fak(3) = 3 \cdot fak(2) = 6$$

$$fak(4) = 4 \cdot fak(3) = 24$$

$$fak(5) = 5 \cdot fak(4) = 120$$

$$fak(6) = 6 \cdot fak(5) = 720$$

$$fak(7) = 7 \cdot fak(6) = 5.040$$

$$fak(8) = 8 \cdot fak(7) = 40.320$$

$$fak(9) = 9 \cdot fak(8) = 362.880$$

## Beispiel 3 (Iterative Definition der Fakultät)

$$fak(n) = \prod_{i=1}^n i = 1 \cdot 2 \cdot \dots \cdot n - 1 \cdot n$$

# Beispiele für rekursive Funktionen

## Beispiel 4 (Rekursive Definition der Fibonacci-Zahlen)

$$\text{fib}(n) = \begin{cases} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2), & \text{falls } n > 1 \end{cases}$$

$\text{fib}(2) = 1$	$\text{fib}(7) = 13$	$\text{fib}(12) = 144$
$\text{fib}(3) = 2$	$\text{fib}(8) = 21$	$\text{fib}(13) = 233$
$\text{fib}(4) = 3$	$\text{fib}(9) = 34$	$\text{fib}(14) = 377$
$\text{fib}(5) = 5$	$\text{fib}(10) = 55$	$\text{fib}(15) = 610$
$\text{fib}(6) = 8$	$\text{fib}(11) = 89$	$\text{fib}(16) = 987$

- Benannt ist die Folge nach nach LEONARDO FIBONACCI, der damit 1202 das Wachstum einer Kaninchenpopulation beschrieb.

# Wann ist Rekursion einsetzbar?

- Rekursion ist ein grundlegendes *mathematisches* Konzept.
- Rekursion ist dann angezeigt, wenn eine Funktion  $p$  für die Lösung eines Problems der Größe  $n$  unter der Annahme formuliert werden kann, dass  $p$  bereits Teilprobleme einer Größe kleiner als  $n$  löst.
- Wenn beispielsweise die Lösung von  $p(n)$  auf  $p(n-1)$  zurückgeführt werden kann, so lässt sich folgende **Rekursionsgleichung** aufstellen:

$$p(n) = f(p(n-1), \dots)$$

- Hier beschreibt die Funktion  $f$ , wie die Lösung für  $p(n-1)$  transformiert werden muss, um die Lösung für  $p(n)$  zu erhalten.
- Eine *notwendige* Bedingung für die Terminierung ist, dass die Rekursion immer schließlich auf Probleme führt, deren Lösung direkt ohne Anwendung der Funktion  $p$  möglich ist → **Rekursionsabbruch**.

# Rekursive Definition von Mengen

- Rekursion lässt sich auch einsetzen, um Mengen (z. B. die Menge der natürlichen Zahlen) rekursiv zu definieren → **Rekursive Definition**.
- Die rekursive Definition ist hierbei ein (zur vollständigen Induktion analoges) Verfahren, bei der die Elemente der Menge durch einen **Rekursionsanfang** und einen **Rekursionsschritt** definiert werden.
- Daher wird für die rekursive Definition häufig auch der (veraltete) Begriff der **induktiven Definition** benutzt.
- Die rekursive Definition einer Menge  $A$  erfolgt in zwei Schritten:
  - ① Zunächst wird eine Menge von **Grundelementen**  $B$  definiert, die allesamt in  $A$  enthalten sind.
  - ② Dann wird eine Menge von **Konstruktionsregeln** definiert, mit denen
    - (a) sich alle restlichen Elemente von  $A$  konstruieren lassen und
    - (b) die Elemente von  $A$  ausreichend charakterisiert werden.

# Beispiel: Natürliche Zahlen

- Die Menge der natürlichen Zahlen  $\mathbb{N}$  wird durch folgende Axiome induktiv definiert:
  - ① 0 ist eine natürliche Zahl.
  - ② Jede natürliche Zahl  $n$  hat genau einen Nachfolger  $n'$ , der ebenfalls eine natürliche Zahl ist.
  - ③ Es gibt keine natürliche Zahl, deren Nachfolger 0 ist.
  - ④ Jede natürliche Zahl ist Nachfolger höchstens einer natürlichen Zahl. (Zwei unterschiedliche natürliche Zahlen  $n$  und  $m$  besitzen stets unterschiedliche Nachfolger  $n'$  und  $m'$ ).
  - ⑤ Enthält eine Menge  $X$  die Zahl 0 und mit jeder natürlichen Zahl  $n$  auch stets deren Nachfolger  $n'$ , so enthält  $X$  bereits alle natürlichen Zahlen.
- Dies ist die bekannte induktive Charakterisierung der natürlichen Zahlen durch die **Peano-Axiome** (GUISEPPE PEANO, 1889)
- Axiom 5 wird auch *Induktionsaxiom* genannt. Es bildet die Grundlage für die Beweismethode der vollständigen Induktion.

# Beispiel: Natürliche Zahlen

- Wie sehen nun die Elemente  $n \in \mathbb{N}$  aus?
- Nach Axiom 2 gibt es einen Nachfolger, der aufgrund von Axiom 4 durch eine **Nachfolgerfunktion** berechnet werden kann:

$$suc : \mathbb{N} \rightarrow \mathbb{N}$$

- Durch wiederholtes Anwenden der Nachfolgerfunktion auf die Elemente der Ausgangsmenge, die nur aus der 0 besteht, können daher alle natürlichen Zahlen erhalten werden:

$$0, suc(0), suc(suc(0)), suc(suc(suc(0))), \dots$$

- Man sagt daher auch:  $\mathbb{N}$  wird auf Basis der Peano-Axiome durch 0 und *suc* *rekursiv* definiert.
- Wenn wir die Nachfolgerfunktion anders darstellen, z. B. durch einen Anführungsstrich: ' , so erhalten wir:  $0, 0', 0'', 0''', 0'''', 0''''' , \dots$
- Werden die Nullen weggelassen, so ergibt sich die Strichdarstellung, oder **unäre Darstellung** der natürlichen Zahlen (außer Null).

# Funktionen auf rekursiv definierten Mengen

- Auf rekursiv definierten Mengen können Funktionen durch Rekursion definiert werden.

## Beispiel 5 (Rekursive Definition einiger Funktionen)

- Addition

$$\text{add}(x, 0) = x$$

$$\text{add}(x, \text{suc}(y)) = \text{suc}(\text{add}(x, y))$$

- Multiplikation:

$$\text{mult}(x, 0) = 0$$

$$\text{mult}(x, \text{suc}(y)) = \text{add}(x, \text{mult}(x, y))$$

- Vorgängerfunktion:

$$\text{pred}(0) = 0$$

$$\text{pred}(\text{suc}(y)) = y$$



# Funktionen auf rekursiv definierten Mengen

- Die rekursive Definition von *add* ist als Berechnungsvorschrift nutzbar:

*add*(4, 2)

= *add*(*suc*(*suc*(*suc*(*suc*(0)))), *suc*(*suc*(0)))

= *suc*(*add*(*suc*(*suc*(*suc*(*suc*(0)))), *suc*(0))) [Def. *add*(*x*, *suc*(*y*))]

= *suc*(*suc*(*add*(*suc*(*suc*(*suc*(*suc*((0))))), 0))) [Def. *add*(*x*, *suc*(*y*))]

= *suc*(*suc*(*suc*(*suc*(*suc*(*suc*(0)))))) [Anwendung Def. *add*(*x*, 0)]

= 6

# Funktionen auf rekursiv definierten Mengen

- Weiterhin lässt sich die Assoziativität von *add* beweisen:

- **Behauptung:**

$$(a + b) + c = a + (b + c)$$

oder in funktionaler Notation

$$\text{add}(\text{add}(a, b), c) = \text{add}(a, \text{add}(b, c))$$

- **Beweisidee:**

Induktion über *c*:

Zeige

$$\text{add}(\text{add}(a, b), c) = \text{add}(a, \text{add}(b, c))$$

$$\Rightarrow \text{add}(\text{add}(a, b), \text{succ}(c)) = \text{add}(a, \text{add}(b, \text{succ}(c)))$$

# Funktionen auf rekursiv definierten Mengen

- Assoziativität von *add* beweisen:

**Induktionsanfang.** Zeige Behauptung für  $c = 0$ .

$$add(add(a, b), 0)$$

$$= add(a, b)$$

$$[wg. \text{ Def } add(x, 0) = x]$$

$$= add(a, add(b, 0))$$

$$[wg. b = add(b, 0)]$$

**Induktionsannahme.**

$$add(add(a, b), c) = add(a, add(b, c))$$

**Induktionsschluss.**

$$add(add(a, b), suc(c))$$

$$= suc(add(add(a, b), c))$$

$$[wg. add(x, s(y)) = s(add(x, y))]$$

$$= suc(add(a, add(b, c)))$$

$$[wg. \text{ Induktionsannahme}]$$

$$= add(a, suc(add(b, c)))$$

$$[wg. s(add(x, y)) = add(x, s(y))]$$

$$= add(a, add(b, suc(c)))$$

$$[wg. s(add(x, y)) = add(x, s(y))]$$

# Äquivalenz von Turing-Maschinen und partiell-rekursiven Funktionen

- Die betrachteten rekursiv definierten Funktionen sind Beispiele für Funktionen aus der Klasse der **primitiv-rekursiven Funktionen**.
- Die Klasse der primitiv-rekursiven ist wiederum in der Klasse der **partiell-rekursiven** Funktionen enthalten.
- Die Funktionen beider Klassen lassen sich aus wenigen, sehr einfachen **Basisfunktionen** mit Hilfe weniger Bildungsregeln erzeugen.

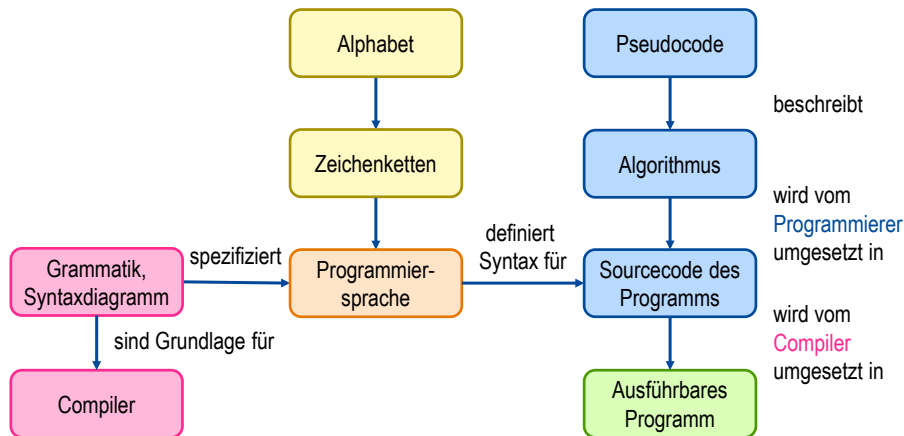
## Äquivalenz von Turing-Maschinen und partiell-rekursiven Funktionen

Jede Turing-Maschine lässt sich durch eine partiell-rekursive Funktion simulieren, und umgekehrt lässt sich jede partiell-rekursive Funktion mittels einer Turing-Maschine berechnen.

## Kapitel 1.4

# Was ist eine Programmiersprache?

# Vom Algorithmus zum ausführbaren Programm



# Vom Algorithmus zum ausführbaren Programm

- Algorithmen sind häufig abstrakt spezifiziert (z. B. in Pseudocode).
- Bevor ein Computer einen Algorithmus ausführen kann, muss ein **Programm** erstellt werden, das diesen Algorithmus umsetzt.
- Der Vorgang des Erstellens eines Programms wird auch als **Programmierung** bezeichnet.
- Programme können in diversen **Programmiersprachen** erstellt werden, wobei Menschen die Programmierung in **Hochsprachen** (z. B. Java, C, C++, Haskell) bevorzugen.
- Der Computer selbst kann jedoch nur **Maschinencode** ausführen, der eine sehr primitive Sprache ist.
- Daher muss ein Programm, das nicht in Maschinencode geschrieben ist, zur Ausführung in Maschinencode übersetzt werden.
- Hierfür werden spezielle Werkzeuge genutzt (**Compiler**, **Assembler**, **Linker** etc.). Eine Alternative ist der **Interpreter**.

# Sprachebenen

- Programmierung in High-Level-Sprache
  - Beispiel:  $A + B$
  - Compiler übersetzt Quellcode in Assembler
- Programmierung in Assembler
  - Beispiel: `add AX,BX`
  - Assembler übersetzt Assemblercode in Maschinensprache
- Programmierung in Maschinensprache
  - Anweisungen in Maschinensprache sind Bitsequenzen
  - Beispiel: 1000110010100000
  - Bedeutung: Addition zweier Zahlen

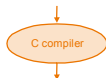


# Hochsprachen: Abstraktion von spezifischen Maschinengegebenheiten

High-level  
language  
program  
(in C)

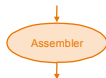
```
swap(int v[], int k)
{int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

← Hardware  
unabhängig



Assembly  
language  
program  
(for MIPS)

```
swap:
  muli $2, $5, 4
  add $2, $4, $2
  lw $15, 0($2)
  lw $16, 4($2)
  sw $16, 0($2)
  sw $15, 4($2)
  jr $31
```

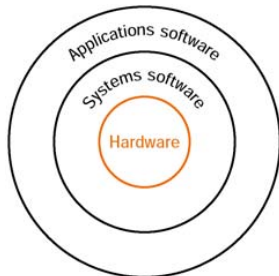


Binary machine  
language  
program  
(for MIPS)

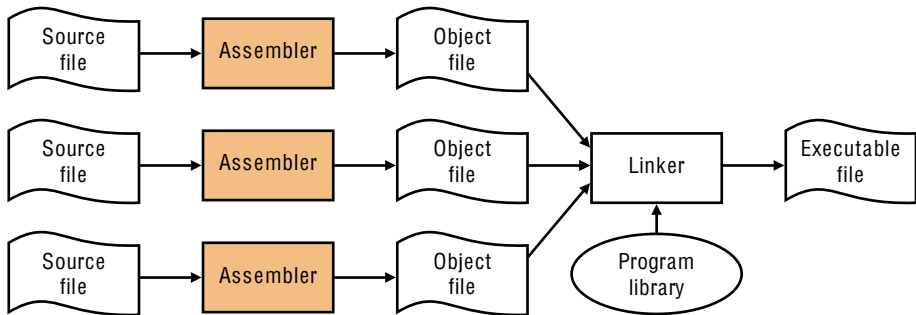
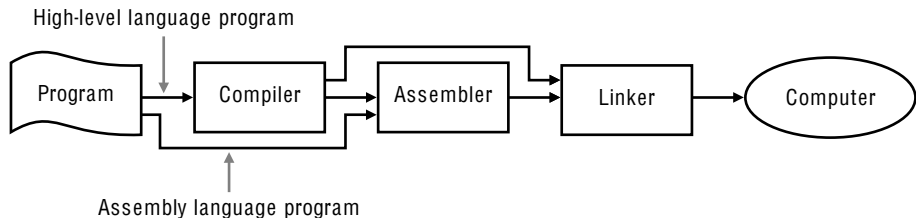
```
000000001010000100000000000011000
00000000100011100001100000100001
10001100011000100000000000000000
100011001111001000000000000000100
10101100111100100000000000000000
101011000110001000000000000000100
00000011111000000000000000001000
```

← Hardware  
abhängig

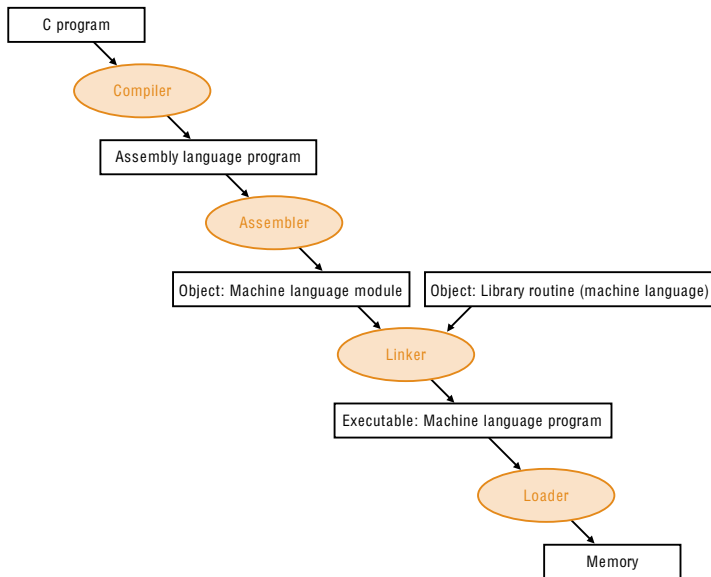
hierarchische  
HW-/SW-Schichten



# Hochsprachen: Übersetzung



# Hochsprachen: Übersetzung



# Alphabete

- Im Sinne der Mathematik und auch der Informatik ist eine Sprache eine Menge von Zeichenreihen über einem Alphabet.

## Definition 6 (Alphabet)

Ein **Alphabet**  $\mathcal{A}$  ist eine nicht leere, endliche Menge von Zeichen.

## Beispiel 6 (Exemplarische Definition einiger Alphabete)

- $\mathcal{A}_1 = \{"A", \dots, "Z"\}$
- $\mathcal{A}_2 = \{"a"\}$
- $\mathcal{A}_3 = \{"A", \dots, "Z", "a", \dots, "z", "0", \dots, "9"\}$
- $\mathcal{A}_4 = \{"\#", "0"\}$
- $\mathcal{A}_5 = \{"er", "sie", "liest", "ein", "Buch"\}$
- $\mathcal{A}_6 =$  Menge aller Verkehrszeichen

# Zeichenreihen über einem Alphabet

## Definition 7 (Menge aller Zeichenreihen)

Die **Menge aller Zeichenreihen**  $\mathcal{A}^*$  über einem Alphabet  $\mathcal{A}$  wird rekursiv definiert:

- ①  $\epsilon \in \mathcal{A}^*$  (die leere Zeichenreihe  $\epsilon$  ist eine Zeichenreihe)
- ②  $a \in \mathcal{A} \Rightarrow a \in \mathcal{A}^*$  (jedes Zeichen ist eine Zeichenreihe)
- ③  $x, y \in \mathcal{A}^* \Rightarrow x \cdot y \in \mathcal{A}^*$   
(die Verkettung zweier Zeichenketten ergibt eine Zeichenkette)
- ④  $\mathcal{A}^*$  enthält keine anderen Zeichenketten neben den durch die Regeln 1–3 gebildeten.

- Die Funktion „ $\cdot$ “ repräsentiert die **Verkettung von Zeichenketten**:  
"foo"  $\cdot$  "bar" = "foobar".

# Zeichenreihen über einem Alphabet

## Beispiel 7

- "ATA", "ATTTNBNM"  $\in \mathcal{A}_1^*$
- "a", "aa", "aaa", "aaaa", ...  $\in \mathcal{A}_2^*$
- "Paul", "Omega", "x12"  $\in \mathcal{A}_3^*$ ;
- "#", "#0#0000"  $\in \mathcal{A}_4^*$
- "er er er sie sie sie"  $\in \mathcal{A}_5^*$

# Sprache als Teilmenge der Zeichenreihen

## Definition 8 (Sprache)

Eine **Sprache**  $\mathcal{L}_{\mathcal{A}}$  ist eine Teilmenge der Menge aller Zeichenreihen über einem Alphabet  $\mathcal{A}$ :  $\mathcal{L}_{\mathcal{A}} \subseteq \mathcal{A}^*$ .

- Beispiel:

$$\text{Englisch} \subset \{ "A" \dots "Z", "a" \dots "z", "0" \dots "9", \\ " ", ",", ".", "!", "?" \}^*$$

- Falls *Englisch* geeignet definiert ist, gilt:

$$\text{"This is a cat."} \in \text{Englisch}$$

- In jedem Fall gilt:

$$\text{"Rübenkraut"} \notin \text{Englisch}$$

Warum?

# Festlegung einer Sprache

- Wie können die Zeichenreihen festgelegt werden, die zu einer bestimmten Sprache gehören sollen?
- Dafür gibt es mehrere Möglichkeiten:
  - ① **Aufzählung aller Zeichenreihen oder Wörter**
    - $L_1 = \{"a", "aaa", "aaaaaa"\}$
    - $L_2 =$  Sprache der Verkehrszeichen, wobei nicht mehr als 5 Zeichen zu einem Wort gehören.
  - ② **Mathematische Charakterisierung** der zur Sprache gehörenden Wörter (formale Sprachen aus der Theoretischen Informatik).
    - $L_3 = \{a^n \mid n \geq 3\}$
    - $L_4 = \{a^n b^n c^n \mid n \geq 1\}$
  - ③ Festlegung durch Verwendung einer **Grammatik**. Diese ermöglichen eine *endliche* (konstruktive) Beschreibung einer im allgemeinen unendlichen Menge und sind zur Beschreibung von Sprachen geeignet.



## Definition 9 (Grammatik)

Eine **Grammatik**  $G$  ist ein Viertupel  $G = (T, N, P, S)$  mit:

- Einer endlichen Menge  $T$  von **Terminalsymbolen**, dem Alphabet.
- Einer endlichen Menge  $N$  von **Nichtterminalsymbolen**, den syntaktischen Variablen. Es gilt  $T \cap N = \emptyset$ .
- Einer endlichen Menge  $P$  von **Produktionsregeln**, die jedem Nichtterminal mögliche Ersetzung (Sequenz von Terminalen und Nichtterminalen) zuordnen.
- Einem **Startsymbol**  $S$  mit  $S \in N$ .

# Produktionsregeln einer Grammatik

- Wir beschränken uns hier auf **kontextfreie Grammatiken**.
- Kontextfreie Grammatiken enthalten nur solche Ersetzungsregeln, bei denen immer genau ein Nichtterminal durch eine Folge von Nichtterminalen und Terminalen ersetzt wird.
- Dabei steht das zu ersetzende Nichtterminal allein (kontextfrei) auf der linken Seite der Ersetzungsregel.
- In diesem Fall sind Produktionsregeln Paare der Form  $(p, q)$ .
- Hierbei repräsentiert  $p \in N$  das zu ersetzende Nichtterminalsymbol und  $q \in (T \cup N)^*$  seine Ersetzung.
- Produktionsregeln der Form  $(p, q)$  werden auch geschrieben als

$$p = q$$

- In einer **Satzform**  $f$  (einer Folge von Nichtterminalen und Terminalen, d. h.,  $f \in (T \cup N)^*$ ) kann ein Nichtterminal durch die rechte Seite einer Regel ersetzt werden, auf deren linken Seite es steht.

# Sprache einer Grammatik

## Definition 10 (Von einer Grammatik erzeugte Sprache)

*Die von einer Grammatik  $G = (T, N, P, S)$  **erzeugte Sprache**  $L(G)$  besteht aus allen aus dem Startsymbol  $S$  mit Hilfe der Produktionsregeln in  $P$  ableitbaren Zeichenketten, die nur aus Terminalen bestehen.*

- Die kontextfreien Grammatiken erzeugen die sogenannten **kontextfreien Sprachen**.

## Beispiel 8

- Betrachte  $G_0 = (T_0, N_0, P_0, S_0)$  mit

$T_0 = \{ "a", "b", ":", "+", "1", "0" \}$

$N_0 = \{ \text{Zuweisung}, \text{Variable}, \text{Ausdruck}, \text{Konstante} \}$

$P_0 = \{ \text{Zuweisung} =_1 \text{Variable} " := " \text{Ausdruck},$   
 $\text{Ausdruck} =_2 \text{Variable}, \text{Ausdruck} =_3 \text{Konstante},$   
 $\text{Ausdruck} =_4 \text{Ausdruck} "+" \text{Ausdruck},$   
 $\text{Variable} =_5 "a", \text{Variable} =_6 "b"$   
 $\text{Konstante} =_7 "1", \text{Konstante} =_8 "0" \}$

$S_0 = \text{Zuweisung}$

- Wie lässt sich "a := a + b + 1" aus *Zuweisung* ableiten?

## Beispiel 9

- Ableitung von "a := a + b + 1" aus *Zuweisung* für  $G_0$ :

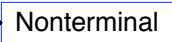
<i>Zuweisung</i> $\rightarrow$ <i>Variable</i> " := " <i>Ausdruck</i>	[Regel 1]
$\rightarrow$ "a := " <i>Ausdruck</i>	[Regel 5]
$\rightarrow$ "a := " <i>Ausdruck</i> "+" <i>Ausdruck</i>	[Regel 4]
$\rightarrow$ "a := " <i>Ausdruck</i> "+" <i>Konstante</i>	[Regel 3]
$\rightarrow$ "a := " <i>Ausdruck</i> "+" <i>Ausdruck</i> "+" <i>Konstante</i>	[Regel 4]
$\rightarrow$ "a := " <i>Variable</i> "+" <i>Ausdruck</i> "+" <i>Konstante</i>	[Regel 2]
$\rightarrow$ "a := " <i>Variable</i> "+" <i>Ausdruck</i> "+ 1"	[Regel 7]
$\rightarrow$ "a := a +" <i>Ausdruck</i> "+ 1"	[Regel 5]
$\rightarrow$ "a := a +" <i>Variable</i> "+ 1"	[Regel 2]
$\rightarrow$ "a := a + b + 1"	[Regel 6]


# Darstellung von Grammatiken

- Grammatiken können genutzt werden,
  - um die Worte einer Sprache zu erzeugen oder
  - um festzustellen, ob ein gegebenes Wort zur Sprache gehört.
- Wir konzentrieren uns zunächst auf die Erzeugung der Wörter einer Sprache und betrachten dazu zwei Darstellungen von Grammatiken:
  - **Syntaxdiagramme** (grafische Notation)
  - **Erweiterte Backus-Naur-Form (EBNF)** (textuelle Notation)
- Beide Notationen ermöglichen eine große Anzahl von Produktionen kompakt und (vergleichsweise) leicht verständlich darzustellen.

# Syntaxdiagramme

- Ein **Syntaxdiagramm** ist ein knotenmarkierter, gerichteter Graph.
- Ein **gerichteter** Graph besteht aus einer Menge von **Knoten**  $V$  und einer Menge **gerichteter Kante**  $E \subseteq V \times V$ .
- Jedes Syntaxdiagramm hat eine **Bezeichnung** und genau einen **Eingangsknoten** sowie genau einen **Ausgangsknoten**.
- Neben dem Ausgangs- bzw. Endknoten gibt es zwei Knotentypen:

→  → Bezeichnung eines anderen Syntaxdiagramms

→  → Zeichenreihe, die in den Satz eingefügt wird

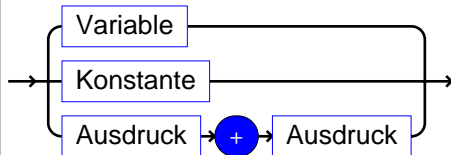
- Jeder Weg durch ein Syntaxdiagramm liefert durch Aneinanderreihen der dabei erreichten terminalen Markierungen ein (syntaktisch korrektes) Wort der Sprache.

# Beispiel: $G_0$ als Syntaxdiagramm

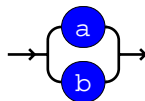
Zuweisung



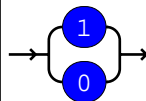
Ausdruck



Variable

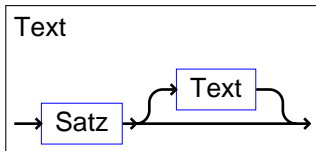
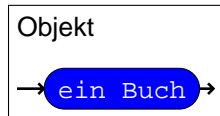
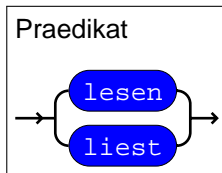
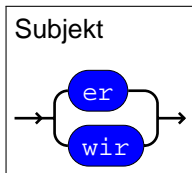
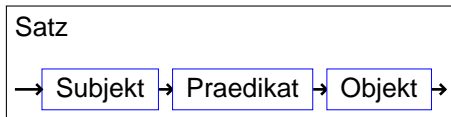


Konstante

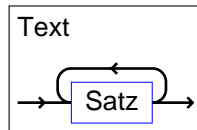




# Beispiel: Deutsche Texte als Syntaxdiagramm



oder



# Erweiterte Backus-Naur-Form

- Die **erweiterte Backus-Naur-Form (EBNF)** ist eine formale Metasyntax (Metasprache), zur kompakten und einfacher verständlichen Darstellung kontextfreier Grammatiken.
- Im Gegensatz zur „normalen“ Definition von Grammatikregeln, kann die rechte Seite von Regeln neben Terminalen und Nichtterminalen auch **Metasymbole** mit besonderer Bedeutung enthalten:

Symbol	Bedeutung
"ttt"	terminales Symbol, ist in dieser Form Bestandteil des Satzes einer Sprache.
[ ]	optionales (null- oder einmaliges) Auftreten der geklammerten Elemente
{ }	null- oder beliebig oftmaliges Auftreten der geklammerten Elemente
( )	Gruppierung von Elementen (einmaliges Auftreten)
	Trennung von Alternativen
=	Trennung linker von rechter Regelseite.
.	Ende der Regel
<i>Bezeichner</i>	Nichtterminal, wird durch weitere Regeln erklärt.

# Erweiterte Backus-Naur-Form

## Beispiel 10

*Zuweisung* = *Variable* " :=" *Ausdruck*

*Ausdruck* = *einfacher\_Ausdruck* [*Relations\_Operator* *einfacher\_Ausdruck*]

- Hier sind „[“ und „]“ Metasymbole, die einen optionalen Teil im Ersetzungstext kennzeichnen.

## Beispiel 11

*name* = *buchstabe* { *buchstabe* | *ziffer* }

## Beispiel 12

*zahl* = [ "-" ] *ziffer* *ohnenull* { *ziffer* } | "0"

- Es gibt eine Reihe von Varianten der EBNF (z. B.. mit „?“ , „+“ und „\*“), siehe <http://de.wikipedia.org/wiki/Backus-Normalform>.

## Kapitel 1.5

# Geschichte der Programmiersprachen

# Programmierparadigma

## ■ Imperative Programmierung

- Die imperative Programmierung ist das bekannteste Programmierparadigma.
- Ein imperatives Programm beschreibt eine Berechnung durch eine **Abfolge von Anweisungen**, die den Programzustand verändern.

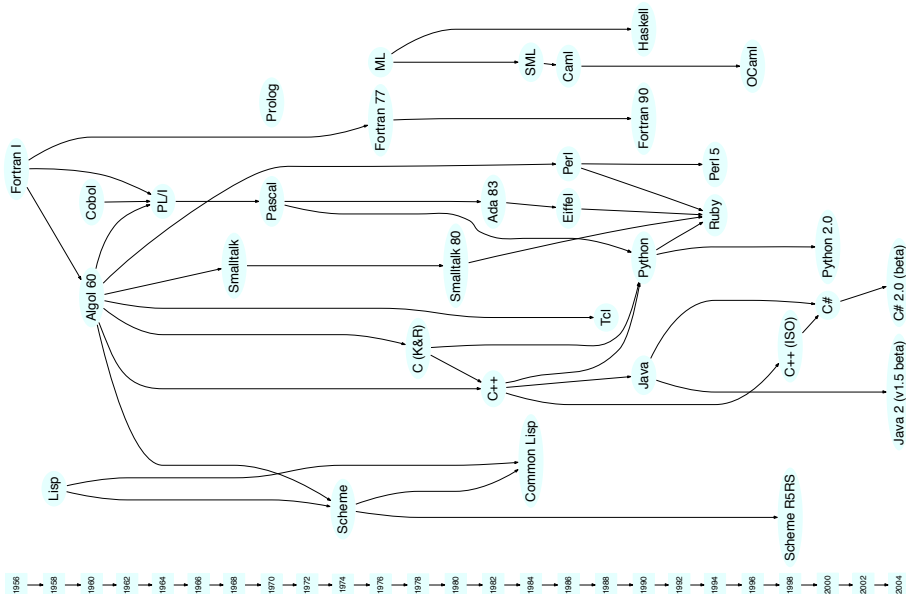
## ■ Funktionale Programmierung

- Ein funktionales Programm besteht nur aus **zustandslosen Funktionen**, die zur Lösung des Gesamtproblems kombiniert und transformiert werden.

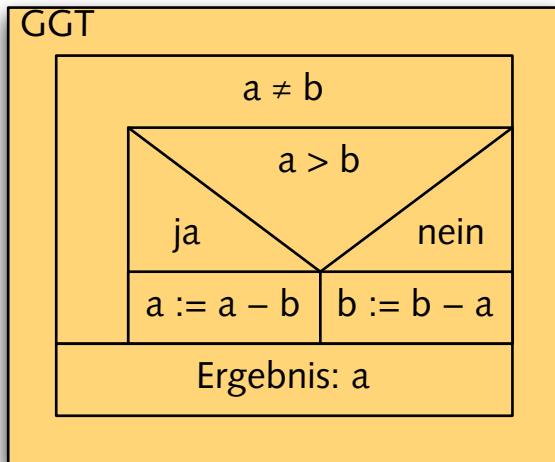
## ■ Logische Programmierung

- Ein logisches Programm besteht aus einer **Menge von Axiomen**. Stellt der Benutzer eines Programms eine Anfrage, so versucht ein Interpreter die Lösungsaussage allein aus den Axiomen abzuleiten.

# Geschichte der Programmiersprachen



# Ein Algorithmus in verschiedenen Sprachen



# FORTRAN (FORmula TRANslation)

- Erste jemals tatsächlich realisierte höhere Programmiersprache (Zuses Plankalkül wurde nicht realisiert); 1953 Vorschlag von John W. Backus, Programmierer bei IBM. Erstes Fortran-Programm am 20. September 1954 von Harlan Herrick, dem Erfinder der später heftig kritisierten Goto-Anweisung.
- Vorgesehen und optimiert für numerische Berechnungen (z. B. von Beginn an Potenz-Operator \*\* und komplexe Zahlen). Anfangs keine Rekursion. Fortran90 standardisiert Vektor- und Matrix-Operationen.
- Es gibt noch viele unentbehrliche Bibliotheken für wissenschaftliche und numerische Berechnungen in FORTRAN.

```
C      FORTRAN
10     IF (A .EQ. B) GOTO 100
IF (A .GT. B)
  *GOTO 50
B=B-A
GOTO 10
50     A=A-B
GOTO 10
100    CONTINUE
```



# LISP (LISt Processing)

- Zweitälteste Programmiersprache, die heute noch im Einsatz ist. Entwickelt von John McCarthy 1958 am MIT. Design beschrieben 1960 in Communications of the ACM, „Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I“. (Teil 2 wurde nie veröffentlicht.)
- Grundlage: **Lambda-Kalkül** von Alonzo Church. Ermöglicht den Aufbau einer vollständigen Sprache allein mit einer Notation für Funktionen.
- Ursprung der Funktionalen Programmierung mit Rekursion zentraler Basis.
- Programme als Daten: **Higher-Order-Programming** (Funktionen die Funktionen als Eingabe / Ausgabe haben)
- Manipulation von Listen und Symbolen, besonders geeignet für Wissensrepräsentation (KI)

```
; LISP
(de ggt (a b)
  (cond ((equal a b) a)
        ((lessp a b) (ggt a (difference b a)))
        (t          (ggt (difference a b) b))))
```

# Algol (Algorithmic Language)

- Algol 60 wurde von 1958–1963 unter der Führung der Association for Computing Machinery (ACM) und der Gesellschaft für Angewandte Mathematik und Mechanik (GAMM), später dann der International Federation for Information Processing (IFIP), entwickelt. Beteiligt unter anderem John Backus, Friedrich Ludwig Bauer, John McCarthy, Peter Naur, Alan J. Perlis, Heinz Rutishauser, Klaus Samuelson.
- Wesentliche Aspekte heutiger imperativer Sprachen: Blockstruktur, Rekursion und Stack (ab Algol 68 Heap), Backus-Naur-Form, Call-By-Value und Call-By-Name.
- Heute keine Bedeutung mehr (aber Konzepte über C, Scheme, Perl in heutige Programmiersprachen eingeflossen).

```
comment ALGOL 60;  
integer procedure ggt(a,b); value a, b; integer a, b;  
begin  
    integer x;  
    for x:=a while a≠b do  
        if a>b then a:=a-b else b:=b-a;  
    ggt := a;  
end
```

# COBOL (COmmon Business Oriented Language)

- COBOL entstand aus dem dringenden Wunsch, eine hardware-unabhängige, standardisierte und problemorientierte Sprache für die Erstellung von Programmen für den betriebswirtschaftlichen Bereich zu haben.
- Die Programmierung kaufmännischer Anwendungen unterscheidet sich von technisch-wissenschaftlichen Anwendungen durch die Handhabung großer Datenmengen statt der Ausführung umfangreicher Berechnungen.
- Das Ergebnis wurde dann 1960 als COBOL-60 von CODASYL verabschiedet und in der Folgezeit weiterentwickelt und von nationalen und internationalen Normierungsinstituten (ANSI, ISO) standardisiert.

NOTE    COBOL

ANFANG.

IF A EQUAL TO B THEN NEXT SENTENCE

ELSE

IF A GREATER B THEN SUBTRACT B FROM A GOTO ANFANG

ELSE SUBTRACT A FROM B GOTO ANFANG.

# BASIC

- BASIC (Beginner's All-purpose Symbolic Instruction Code) wurde 1964 von John George Kemeny und Thomas Eugene Kurtz am Dartmouth College entwickelt, um den Elektrotechnikstudenten den Einstieg in die Programmierung mit Algol und Fortran zu erleichtern.
- Sehr einfache Sprache.
  - Keine Struktur
  - Keine Prozedurparameter, keine lokalen Variablen
  - Keine Nutzerdefinierten Datentypen, kein dynamischer Speicher

```
10 REM BASIC:
20 IF A = B THEN GOTO 99
30 IF A > B THEN
40     LET A = A - B
50 ELSE
60     LET B = B - A
70 ENDIF
80 GOTO 20
99 END
```

# Pascal

- 1970 durch Niklaus Wirth an der ETH Zürich als Antwort auf die Mammutsprachen PL/1 und ALGOL 68 entwickelt.
- Bietet die Algol-Konzepte von Blockstruktur, nutzerdefinierten Datentypen, rekursiven Prozeduren etc. in einer kompakten, schnell kompilierbaren Sprache.
- Durch kostenlose Compiler, kompakte Sprachdefinition und sehr gute Strukturierbarkeit von Programmen in der Ausbildung sehr beliebt.

```
{Pascal}
```

```
function ggt(a, b : integer) : integer;  
begin  
    while a<>b do  
        if a>b then a:=a-b else b:=b-a;  
    ggt:=a  
end;
```

# Prolog (Programming in Logic)

- Prolog wurde maßgeblich von Alain Colmerauer, einem französischen Informatiker, Anfang der 1970er Jahre entwickelt.
- Prolog gehört zur Familie der logischen Programmiersprachen und ist eine Vertreterin des Paradigmas der deklarativen Programmierung.
- Prolog beruht auf den mathematischen Grundlagen der Prädikatenlogik.
- Ein Prolog-Programm ist eine Sammlung prädikatenlogischer Aussagen in Form von so genannten **Horn-Klauseln**.
  - Beispiel:  $A := B, C$  bedeutet, dass  $A$  gilt, wenn  $B$  und  $C$  gelten.

```
/* PROLOG */
```

```
ggt(A,A,C) :- C is A.
```

```
ggt(A,B,C) :- A > B, A1 is A-B, ggt(A1,B,C).
```

```
ggt(A,B,C) :- A < B, B1 is B-A, ggt(A,B1,C).
```

```
?- ggt(12,20,C).
```

```
C is 4;
```

- C wurde von Ken Thompson und Dennis Ritchie in den frühen 70er Jahren für das neu entwickelte Betriebssystem Unix entworfen. Die grundlegenden Programme aller Unix-Systeme und die Kerne vieler Betriebssysteme sind in C programmiert.
- Ken Thompson passte zunächst die Programmiersprache BCPL an und nannte die so entstandene Sprache „B“ (nach den Bell Laboratories, in denen die Sprache entwickelt wurde). Aus dieser Sprache entstand dann C.
- C ist (vor allem im Vergleich zu Pascal) eine sehr liberale und maschinennahe Sprache (Casts, Pointer-Arithmetik), die für Unaufmerksame das Programmieren zum Abenteuer macht.
- Beispiel: Zeichenweise Kopieren eines Strings: `while(*a++ = *b++);`

```
/* C */
int ggt(int a, int b) {
    while(a != b)
        if (a>b) a = a-b; else b = b-a;
    return a;
}
```

- Haskell ist eine funktionale Programmiersprache, benannt nach dem US-amerikanischen Mathematiker Haskell Brooks Curry, dessen Arbeiten zur mathematischen Logik eine maßgebliche Grundlage heutiger funktionaler Programmiersprachen bilden.
- Haskell wurde in den 1980er Jahren mit der Motivation entwickelt, die funktionale Programmierung durch die Einführung einer standardisierten, modernen Sprache zu vereinheitlichen.
- Die aktuelle Version des Sprachstandards liegt unter dem Namen Haskell 2010 vor.

```
-- haskell
ggt a b | a == b = a
| a > b   = ggt (a-b) b
| True   = ggt a (b-a)
```



- Java ist eine reflexive, objektorientierte Sprache, die von James Gosling und anderen bei Sun Microsystems entwickelt wurde.
- Die erste öffentliche Version von Java wurde 1995 an Beta-Tester verteilt.
- 1996 wurde JDK 1.0 veröffentlicht. Momentan ist die Version 11 aktuell.
- Java bietet
  - „Write-Once, Run Everywhere“ auf Basis einer virtuellen Maschine
  - Mobilen Code durch die Möglichkeit Code nachzuladen
  - Automatisiertes Management von dynamischem Speicher → **Garbage Collection**
  - Abgesicherte Ausführung durch eine sogenannte **Sandbox**

```
/* Java */  
class Ggt {  
    public static int ggt(int a, int b) {  
        while(a != b)  
            if (a>b) a = a-b; else b = b-a;  
        return a;  
    }  
}
```

# Fragen zur Lernkontrolle

- 1 Was ist ein Algorithmus?
- 2 Erläutern Sie die drei wichtigsten Eigenschaften eines Algorithmus!
- 3 Welche weiteren Eigenschaften kann ein Algorithmus haben?
- 4 Worin besteht der Unterschied zwischen einem deterministischen und einem determinierten Algorithmus?
- 5 Was wird unter strukturierter Programmierung verstanden?
- 6 Was ist ein Struktogramm und wozu dient es?
- 7 Beschreiben Sie die wichtigsten Elemente von Struktogrammen!

# Fragen zur Lernkontrolle

- 8 Erläutern Sie die Modellierung eines Algorithmus als mathematische Abbildung oder Funktion!
- 9 Wie ist die berechnete Funktion eines Algorithmus definiert?
- 10 Wann ist eine Funktion berechenbar?
- 11 Sind alle Funktionen berechenbar?
- 12 Was besagt die Church-Turing-These?
- 13 Was versteht man unter Turing-Mächtigkeit?
- 14 Was versteht man unter Rekursion, einer Rekursionsgleichung und dem Rekursionsabbruch?
- 15 Wie können Mengen rekursiv definiert werden?
- 16 Mit welcher Menge rekursiver Funktionen sind Turing-Maschinen im Sinne der Berechenbarkeit äquivalent?

# Fragen zur Lernkontrolle

- 17 Wie ist ein Alphabet, die Menge aller Zeichenketten über einem Alphabet und eine Sprache formal definiert?
- 18 Wie ist eine Grammatik formal definiert?
- 19 Wozu dient eine Grammatik?
- 20 Wann ist eine Grammatik kontextfrei?
- 21 Was ist ein Syntaxdiagramm?
- 22 Erläutern Sie die erweiterte Backus-Naur-Form!
- 23 Worin besteht der wesentliche Unterschied zwischen imperativer, funktionaler und logischer Programmierung?

# Vielen Dank für Ihre Aufmerksamkeit!

Univ.-Prof. Dr.-Ing. Gero Mühl

`gero.muehl@uni-rostock.de`  
`https://www.ava.uni-rostock.de`