# Kapitel 2: Informationsdarstellung

## 2.1 Bits und Bytes, Zahl- und Zeichendarstellungen

**(siehe Gumm/Sommer * Kap.1.2/1.3)**

Damit Informationen von einer Maschine verarbeitet werden können, müssen sie in der Maschine *repräsentiert* werden. Üblich sind dabei Repräsentationsformen, die auf Tupeln oder Folgen über der Menge $\mathbb{B}$ aufbauen. Ein ***Bit (binary digit)*** ist die kleinste Einheit der Informationsdarstellung: es kann genau zwei Werte annehmen, z. B. **0** oder **1.** Genau wie es viele verschiedene Notationen der Menge $\mathbb{B}$ gibt, gibt es viele verschiedene Realisierungsmöglichkeiten eines Bits: an/aus, geladen/ungeladen, weiss/schwarz, magnetisiert/entmagnetisiert, reflektierend/lichtdurchlässig, …
Lässt eine Frage mehrere Antworten zu, so lassen sich diese durch eine ***Bitfolge*** (mehrere Bits) codieren.
**Beispiel:** Die Frage, aus welcher Himmelsrichtung der Wind weht, lässt 8 mögliche Antworten zu. Diese lassen sich durch Bitfolgen der Länge 3 codieren:

000 = Nord
001 = Nordost
010 = Ost
011 = Südwest
100 = Süd
101 = Südost
110 = West
111 = Nordwest

Offensichtlich verdoppelt jedes zusätzliche Bit die Anzahl der möglichen Bitfolgen, so dass es genau $2^n$ mögliche Bitfolgen der Länge $n$ gibt ($|\mathbb{B}|=2 \rightarrow |\mathbb{B}^n|=2^n$)

Ein *Byte* ist ein Oktett von Bits: **8 Bits = 1 Byte.** Oft betrachtet man Bytefolgen anstatt von Bitfolgen. Ein Byte kann verwendet werden, um z.B. folgendes zu speichern:

- ein codiertes Zeichen (falls das Alphabet weniger als $2^8$ Zeichen enthält)
- eine Zahl zwischen 0 und 255,
- eine Zahl zwischen -128 und +127,
- die Farbcodierung eines Punkts in einer Graphik, genannt „*Pixel*“ (*picture element*)

Gruppen von 16 Bits, 32 Bits, 64 Bits bzw. 128 Bits werden häufig als *Halbwort, Wort, Doppelwort* bzw. *Quadwort* bezeichnet. Leider gibt es dafür unterschiedliche Konventionen.

Zwischen 2-er und 10-er Potenzen besteht (näherungsweise) der Zusammenhang:

$$\mathbf{2^{10} = 1024 \cong 1000 = 10^3}$$

Für Größenangaben von Dateien, Disketten, Speicherbausteinen, Festplatten etc. benutzt man daher folgende Präfixe:

- k = $1024 = 2^{10}$ $\cong 10^3$ (k = Kilo)
- M = $1024^2 = 1048576 = 2^{20}$ $\cong 10^6$ (M = Mega)
- G = $1024^3 = 2^{30}$ $\cong 10^9$ (G = Giga)
- T = $1024^4 = 2^{40}$ $\cong 10^{12}$ (T = Tera)
- P = $1024^5 = 2^{50}$ $\cong 10^{15}$ (P = Peta)
- E = $1024^6 = 2^{60}$ $\cong 10^{18}$ (E = Exa)

Die Ungenauigkeit der obigen Näherungsformel nimmt man dabei in Kauf.

Mit *1 GByte* können also entweder $2^{30} = 1024^3 = 1.073.741.824$ oder $10^9 = 1.000.000.000$ Bytes gemeint sein.

**Anhaltspunkte für gängige Größenordnungen von Dateien und Geräten:**

- eine SMS: ~140 B (160 Zeichen zu je 7 Bit)
- ein Brief: ~3 kB
- ein „kleines“ Programm: ~300 kB
- Diskettenkapazität: 1,44 MB
- ein „mittleres“ Programm: ~ 1 MB
- ein Musiktitel: ~40 MB (im MP3-Format ~4 MB)
- CD-ROM Kapazität: ~ 680 MB
- Hauptspeichergröße: 1-8 GB
- DVD (Digital Versatile Disk): ~ 4,7 bzw. ~ 9 GB
- Festplatte: 250-1000 GB.

Für Längen- und Zeiteinheiten werden auch in der Informatik die gebräuchlichen Vielfachen von 10 benutzt. So ist z.B. ein 400 MHz Prozessor mit $400 \times 10^6 = 400.000.000$ Hertz (Schwingungen pro Sekunde) getaktet.
Das entspricht einer Schwingungsdauer von 2,5×10-9 sec, d.h. 2,5 ns. Der Präfix n steht hierbei für *nano*, d.h. den Faktor $10^{-9}$. Weitere Präfixe für Faktoren kleiner als 1 sind:

- m = $1/1000 = 10^{-3}$ (m = Milli)
- µ = $1/1000000 = 10^{-6}$ (µ = Mikro)
- n = $1/1000000000 = 10^{-9}$ (n = Nano)
- p = ... = $10^{-12}$ (p = Pico)
- f = ... = $10^{-15}$ (f = Femto)

**Beispiele:** 1mm = 1 Millimeter; 1 ms = 1 Millisekunde.

Für Längenangaben wird neben den metrischen Maßen eine im Amerikanischen immer noch weit verbreitete Einheit verwendet: 1" = 1 in = 1 inch = 1 Zoll = 2,54 cm = 25,4 mm. Teile eines Zolls werden als Bruch angegeben. Beispiel - Diskettengröße: 3 1/2".

## Darstellung natürlicher Zahlen, Stellenwertsysteme

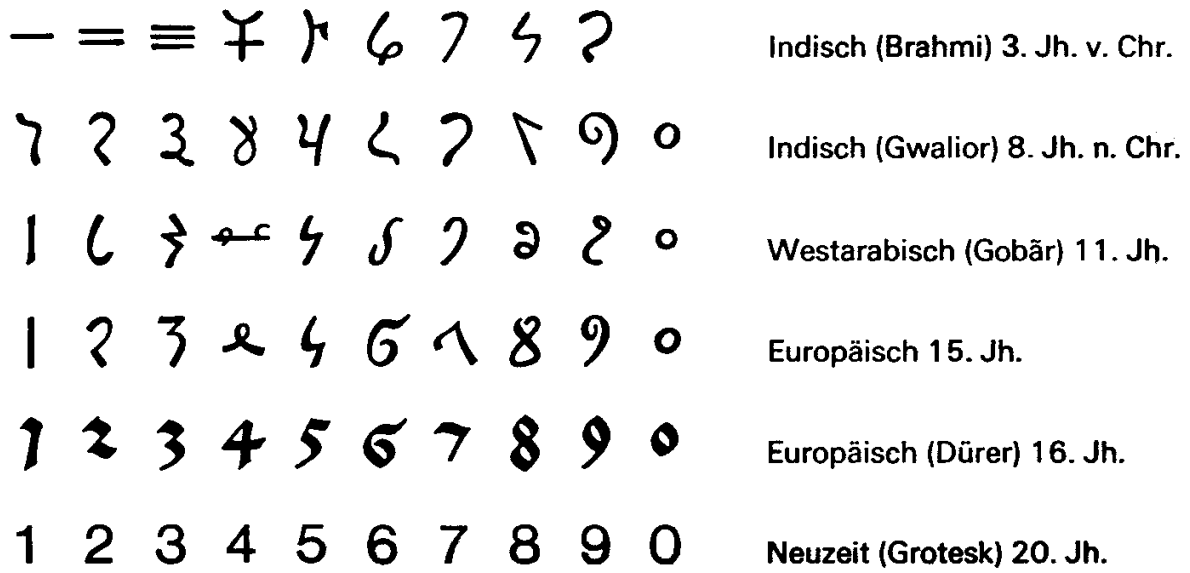
Die älteste Form der Darstellung von Zahlen ist die Strichdarstellung, bei der jedes Individuum durch einen Strich oder ein Steinchen repräsentiert wird (calculi=Kalksteinchen, vgl. kalkulieren). Bei dieser Darstellung ist die Addition besonders einfach (Zusammen- oder Hintereinanderschreibung von zwei Strichzahlen), allerdings wird sie für große Zahlen schnell unübersichtlich. Die Ägypter führten deshalb für Gruppen von Strichen Abkürzungen ein (http://de.wikipedia.org/wiki/Ägyptische_Zahlen, http://www.informatik.uni-hamburg.de/WSV/teaching/vorlesungen/WissRep-Unterlagen/WR03Einleitung-1.pdf).


Daraus entstand dann das römische Zahlensystem:

| arab. | 1 | 5 | 10 | 50 | 100 | 500 | 1 000 | 5 000 | 10 000 | 100 000 | 1 000 000 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| röm. | I | V | X | L | C | D | M | ↁ | ↂ | ↇ | ↈ |
| Bemer-kungen: | | halbes X | | halbes C | | halbes ↀ | entstanden aus ↀ | halbes ↂ | | entstanden aus C und M | zehnfaches ↇ |

Aus Übersichtlichkeitsgründen werden die großen Zahlen dabei zuerst geschrieben; prinzipiell spielt in solchen direkten Zahlensystemen die Position einer Ziffer keine Rolle. Die Schreibweise IV = 5-1 ist erst viel später entstanden!

Direkte Zahlensysteme haben einige Nachteile: Die Darstellung großer Zahlen kann sehr lang werden, und arithmetische Operationen lassen sich in solchen Systemen nur schlecht durchführen. In Indien (und bei den Majas) wurde ein System mit nur zehn verschiedenen Ziffernsymbolen verwendet, bei der die Position jeder Ziffer (von rechts nach links) ihre Wertigkeit angibt. Die wesentliche Neuerung ist dabei die Erfindung der Zahl Null (ein leerer Kreis). Der schon genannte Muhammed ibn Musa al-Khwarizmi verwendete das Dezimalsystem in seinem Arithmetikbuch, das er im 8. Jahrhundert schrieb. Bereits im 10. Jahrhundert wurde das System in Europa eingeführt, durchsetzen konnte es sich jedoch erst im 12. Jahrhundert mit der Übersetzung des genannten Arithmetikbuchs ins Lateinische (durch Fibonacci, siehe oben).(*)




Wir haben schon erwähnt, dass das *Dualzahlen-* oder *Binärsystem* mit nur zwei Ziffern in Europa 1673 von Gottfried Wilhelm Leibnitz (wieder-)erfunden wurde. Allgemein gilt: In Stellenwertsystemen wird jede Zahl als Ziffernfolgen $x_{n-1}... x_0$ repräsentiert, wobei - bezogen auf eine gegebene *Basis b* - jede Ziffer $x_i$ einen *Stellenwert* ($x_i * b^i$) bekommt:

$$[x_{n-1} \dots x_0]_b = \sum_{i=0}^{n-1} x_i * b^i$$

Werden dabei nur Ziffern $x_i$ mit Werten zwischen 0 und *b-1* benutzt, so ergibt sich eine eindeutige Darstellung; dafür werden offenbar genau *b* Ziffern benötigt. Zu je zwei Basen *b* und *b'* gibt es eine umkehrbar eindeutige Abbildung, die $[x_{n-1}... x_0]_b$ und $[ y_{n'-1}... y_0]_{b'}$ mit $0 \le x_i \le b$ und $0 \le y_i \le b'$ ineinander überführt.

**Beispiele:**

**b = 2:** $[10010011]_2 = 1*2^7 + 1*2^4 + 1*2^1 + 1*2^0 = [147]_{10}$

**b = 3:** $[12110]_3 = 1*3^4 + 2*3^3 + 1*3^2 + 1*3^1 = [147]_{10}$

**b = 8:** $[223]_8 = 2*8^2 + 2*8^1 + 3*8^0 = [147]_{10}$

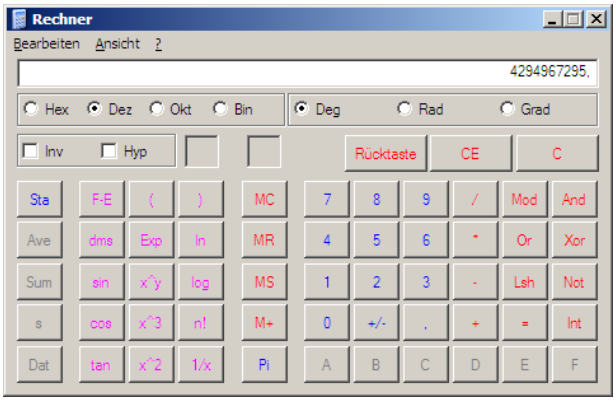
**b = 10:** $[147]_{10} = 1*10^2 + 4*10^1 + 7*10^0 = [10010011]_2$

Wichtige Spezialfälle sind b=10, 2, 8 und 16. Zahlen mit b=8 bezeichnet man als *Oktalzahlen.* Unter *Sedezimalzahlen* (oft auch *Hexadezimalzahlen* genannt) versteht man Zifferndarstellungen von Zahlen zur Basis 16. Sie dienen dazu, Dualzahlen in komprimierter (und damit leichter überschaubarer) Form darzustellen und lassen sich besonders leicht umrechnen. Je 4 Dualziffern werden zu einer *„Hex-ziffer“* zusammengefasst. Da man zur Hexadezimaldarstellung 16 Ziffern benötigt, nimmt man zu den Dezimalziffern 0 ... 9 die ersten Buchstaben A ... F hinzu.

**Beispiele:** Umwandlung von Dezimal - in Oktal- / Hexadezimalzahlen und umgekehrt:

| | | |
|---|---|---|
| $[1]_{10}=[1]_8=[1]_{16}=[1]_2$ | $[16]_{10}=[20]_8=[10]_{16}$ | $[256]_{10}=[400]_8=[100]_{16}$ |
| … | $[17]_{10}=[21]_8=[11]_{16}$ | $[1000]_{10}=[3E8]_{16}$ |
| $[7]_{10}=[7]_8=[7]_{16}=[111]_2$ | … | … |
| $[8]_{10}=[10]_8=[8]_{16}=[1000]_2$ | $[32]_{10}=[40]_8=[20]_{16}$ | $[4096]_{10}=[10000]_8=[1000]_{16}$ |
| $[9]_{10}=[11]_8=[9]_{16}=[1001]_2$ | $[33]_{10}=[41]_8=[21]_{16}$ | … |
| $[10]_{10}=[12]_8=[A]_{16}=[1010]_2$ | … | $[10000]_{10}=[2710]_{16}$ |
| $[11]_{10}=[13]_8=[B]_{16}=[1011]_2$ | $[80]_{10}=[100]_8=[50]_{16}$ | … |
| $[12]_{10}=[14]_8=[C]_{16}=[1100]_2$ | … | $[45054]_{10}=[AFFE]_{16}$ |
| $[13]_{10}=[15]_8=[D]_{16}=[1101]_2$ | $[160]_{10}=[240]_8=[A0]_{16}$ | $[65535]_{10}=[177777]_8=[FFFF]_{16}$ |
| $[14]_{10}=[16]_8=[E]_{16}=[1110]_2$ | … | $[10^6]_{10}=[F4240]_{16}$ |
| $[15]_{10}=[17]_8=[F]_{16}=[1111]_2$ | $[255]_{10}=[377]_8=[FF]_{16}$ | $[4294967295]_{10}=[FFFFFFFF]_{16}$ |

Diese Umrechnung ist so gebräuchlich, dass sie von vielen Taschenrechnern bereit gestellt wird.




Um mit beliebig großen natürlichen oder ganzen Zahlen rechnen zu können, werden diese als Folgen über dem Alphabet der Ziffern repräsentiert. Diese Darstellung wird beispielsweise im **bc** verwendet. Numerische Algorithmen mit solchen Repräsentationen sind allerdings häufig komplex, und in vielen Anwendungen wird diese Allgemeinheit nicht benötigt. Daher werden Zahlen oft als *Dual-* oder *Binärwörter* einer festen Länge *n* repräsentiert. Mit Hilfe von *n* Bits lassen sich $2^n$ Zahlenwerte darstellen:

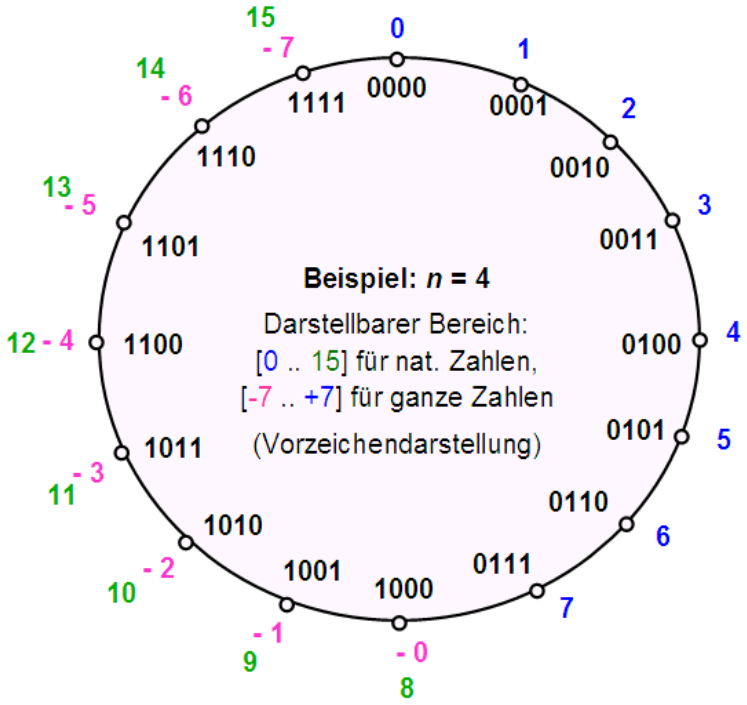
- die ***natürlichen Zahlen*** von 0 bis $2^n$ - 1 oder
- die ***ganzen Zahlen*** zwischen $-2^{n-1}$ und $2^{n-1}$ - 1 oder
- ein Intervall der ***reellen Zahlen*** (mit begrenzter Genauigkeit)

**Beispiel:**

| *Länge* | *darstellbare Zahlen* |
|---|---|
| 4 | 0 .. 15 |
| 8 | 0 .. 255 |
| 16 | 0 .. 65535 |
| 32 | 0 .. 4 294 967 295 |

## Darstellung ganzer Zahlen

Für die Darstellung ganzer Zahlen $\mathbb{Z}$ wird ein zusätzliches Bit (das "*Vorzeichen-Bit*") benötigt. Mit Bitfolgen der Länge $n$ kann also (ungefähr) der Bereich $[-2^{n-1} .. 2^{n-1}]$ dargestellt werden. Nahe liegend ist die dabei *Vorzeichendarstellung*: Das erste Bit repräsentiert das *Vorzeichen* (**0** für '+' und **1** für '-') und der Rest den *Absolutwert*.




Nachteile dieser Darstellung:

- Die Darstellung der 0 ist nicht mehr eindeutig.
- Beim Rechnen "über die 0" müssen umständliche Fallunterscheidungen gemacht werden, Betrag und Vorzeichen sind zu behandeln.
  Beispiel:3 + (-5) = 0011+1101=1010

Eine geringfügige Verbesserung bringt die *Einserkomplement-Darstellung*, bei der jedes Bit der Absolutdarstellung umgedreht wird. Meist wird jedoch die so genannte *Zweierkomplement-Darstellung* (kurz: 2c) benutzt. Sie vereinfacht die arithmetischen Operationen und erlaubt eine eindeutige Darstellung der 0. Bei der Zweierkomplementdarstellung gibt das erste Bit das Vorzeichen an, das 2. bis $n$-te Bit ist das Komplement der um eins verringerten positiven Zahl. Die definierende Gleichung für die Zweierkomplementdarstellung ist:

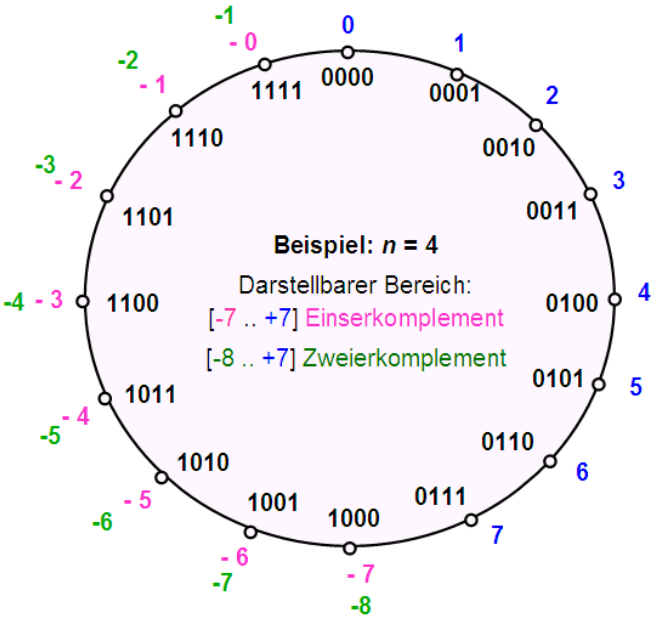
$$[-x_{n-1}... x_0]_{2c} + [x_{n-1}...x_0]_{2c} = [10...0]_{2c} = 2^{n+1}$$

Um zu einer gegebenen positiven Zweierkomplement-Zahl die entsprechende negative zu bilden, invertiert man alle Bits und addiert 1. Die Addition kann mit den üblichen Verfahren berechnet werden (Beweis: eigene Übung!).

**Beispiele:** $n = 4$: $+5 \rightarrow [0101]_{2c}$ , also $-5 \rightarrow (1010 + 0001)_{2c} = [1011]_{2c}$

$3 + (-5) = [0011]_{2c} + [1011]_{2c} = [1110]_{2c}$

$4 + 5 = [0100]_{2c} + [0101]_{2c} \rightarrow [1001]_{2c} = -7$ (Achtung!!!)




Wichtiger Hinweis: In vielen Programmiersprachen wird ein Zahlbereichsüberlauf nicht abgefangen und kann beispielsweise zur Folge haben, dass die nagelneue Rakete abstürzt!

## Darstellung rationaler und reeller Zahlen

Prinzipiell kann man rationale Zahlen $\mathbb{Q}$ als Paare ganzer Zahlen (Zähler und Nenner) darstellen. Ein Problem ist hier die Identifikation gleicher Zahlen: (7,-3)=(-14,6). Hier müsste man nach jeder Rechenoperation mit dem ggT normieren; das wäre sehr unpraktisch. Daher werden in der Praxis rationale Zahlen $\mathbb{Q}$ meist wie reelle Zahlen $\mathbb{R}$ behandelt.
Für reelle Zahlen gilt:

- Es gibt überabzählbar viele reelle Zahlen $\mathbb{R}$. Also gibt es auch reelle Zahlen, die sich nicht in irgend einer endlichen Form aufschreiben lassen, weder als endlicher oder periodischer Dezimalbruch noch als arithmetischer Ausdruck oder Ergebnis eines Algorithmus. Echte reelle Zahlen lassen sich also nie genau in einem Computer speichern, da es für sie definitionsgemäß keine endliche Darstellung gibt.
- Die so genannten „reals“ im Computer sind mathematisch gesehen immer Näherungswerte für reelle Zahlen mit endlicher Genauigkeit.

Für reelle Zahlen $\mathbb{R}$ gibt es die Festkomma- und die Gleitkommadarstellung. Bei der *Festkommadarstellung* steht das Komma an einer beliebigen, festen Stelle. Für $x=[x_{n-1}x_{n-2}\ldots x_1x_0x_{-1}x_{-2}\ldots x_{-m}]_2$ ist $x=\sum_{i=-m}^{n-1} x_i \cdot 2^i$. Diese Darstellung gestattet nur einen kleinen Wertebereich und hat auch sonst einige Nachteile (Normierung von Zahlen erforderlich). Daher verwendet man sie in elektronischen Rechenmaschinen nur in Ausnahmefällen (z.B. beim Rechnen mit Geld). Ziel der *Gleitkommadarstellung* (IEEE754 Standard) ist es,

- ein möglichst großes Intervall reeller Zahlen zu umfassen,
- die Genauigkeit der Darstellung an die Größenordnung der Zahl anzupassen: bei kleinen Zahlen sehr hoch, bei großen Zahlen niedrig.

Daher speichert man neben dem Vorzeichen und dem reinen Zahlenwert - der so genannten *Mantisse* - auch einen *Exponenten* (in der Regel zur Basis 2 oder 10), der die Kommaposition in der Zahl angibt.

- Das *Vorzeichenbit v* gibt an, ob die vorliegende Zahl positiv oder negativ ist.
- Die *Mantisse m* besteht aus einer *n*–stelligen Binärzahl $m_1....m_n$

- Der *Exponent e* ist eine *L*-stellige ganze Zahl (zum Beispiel im Bereich -128 bis +127), die angibt, mit welcher Potenz einer *Basis b* die vorliegende Zahl zu multiplizieren ist.

Das Tripel (*v, m, e*) wird als $(-1)^v * m * b^{e-n}$ interpretiert. Bei gegebener Wortlänge von 32 Bit verwendet man beispielsweise 24 Bit für Vorzeichen und Mantisse ($n$=23) sowie $L$=8 Bit für den Exponenten. Beispiele mit n=L=4: (0, 1001, 0000) = $1 * 9 * 2^{-4} = [0{,}1001]_2$=0,5625 und (1, 1001, 0110) = $-1 * 9 * 2^{6-4} = [-100100]_2$ = -36 und (0, 1001, 1010) = $1 * 9 * 2^{-6-4}$ = 0,00878906… Anstatt wie in diesen Beispielen *e* in Zweierkomplementdarstellung abzuspeichern, verwendet man die sogenannte *biased-Notation*: E=e+e', wobei e'=$2^{L-1}$-1. Damit ist 0≤E≤($2^L$-1) positiv, und (*v, m, E*) wird als $(-1)^v * m * b^{E-(e'+n)}$ interpretiert. Zum Beispiel ist für L=4 der Wert von e'=7, also ein Exponent E=1010 entspricht e=3. Damit ergibt sich (0, 1001, 1010) = $9 * 2^{3-4}$ = 4,5. Bei 32-Bit Gleitkommazahl mit 8-Bit Exponent gilt: *e'*=127, bei 64-Bit Gleitkommazahlen ist *e'*=1023.

In Groovy und Java gibt es die folgenden Datentypen für Gleitpunktzahlen:

- Datentyp `float` (32 Bit) mit dem Wertebereich (+/-)$1.4*10^{-45}$..$3.4*10^{38}$.(7 relevante Dezimalstellen: 23 Bit Mantisse, 8 Bit Exponent)
- Datentyp `double` (64 Bit) mit dem Wertebereich (+/-)$4.9*10^{-324}$..$1.79*10^{308}$.(15 relevante Stellen: 52 Bit Mantisse, 11 Bit Exponent)
- Datentyp `BigDecimal` mit fast beliebiger Präzision, bestehend aus Mantisse und 32-bit Exponent, Wert: (Mantisse * (10 ** (-Exponent)))
  Beispiel: `new BigDecimal(123, 6 ) == 0.000123`

## Zeichendarstellung

Zeichen über einem gegebenen Alphabet **A** werden meist als Bitfolgen einer festen Länge $n \geq \log(|\mathbf{A}|)$ codiert. Oft wird *n* so gewählt, dass es ein Vielfaches von 4 oder von 8 ist.
**Beispiel:** Um Texte in einem Buch darzustellen, benötigt man ein Alphabet von 26 Kleinbuchstaben, ebenso vielen Großbuchstaben, einigen Satzzeichen wie etwa Punkt, Komma und Semikolon und Spezialzeichen wie "+", "&", "%". Daher hat eine normale Schreibmaschinentastatur eine Auswahl von knapp hundert Zeichen, für die 7 Bit ausreichen. Bereits in den 1950-ern wurden die Codes ASCII und EBCDIC hierfür entwickelt. (siehe ASCII-Tabelle in Kap.1.3 Alphabete).

Mit einem Byte lassen sich 256 Zeichen codieren. Viele PCs benutzen den Code-Bereich [128 .. 255] zur Darstellung von sprachspezifischen Zeichen wie z.B. "ä" (Wert 132 in der erweiterten Zeichentabelle), "ö" (Wert 148) "ü" (Wert 129) und einigen Sonderzeichen anderer Sprachen. Leider ist die Auswahl der sprachspezifischen Sonderzeichen eher zufällig und bei weitem nicht ausreichend für die vielfältigen Symbole fremder Schriften. Daher wurden von der „International Standardisation Organisation" (ISO) verschiedene ASCII-Erweiterungen normiert. In Westeuropa ist dazu die 8-Bit ASCII-Erweiterung „Latin-1" nützlich, die durch die Norm ISO8859-1 beschrieben wird.

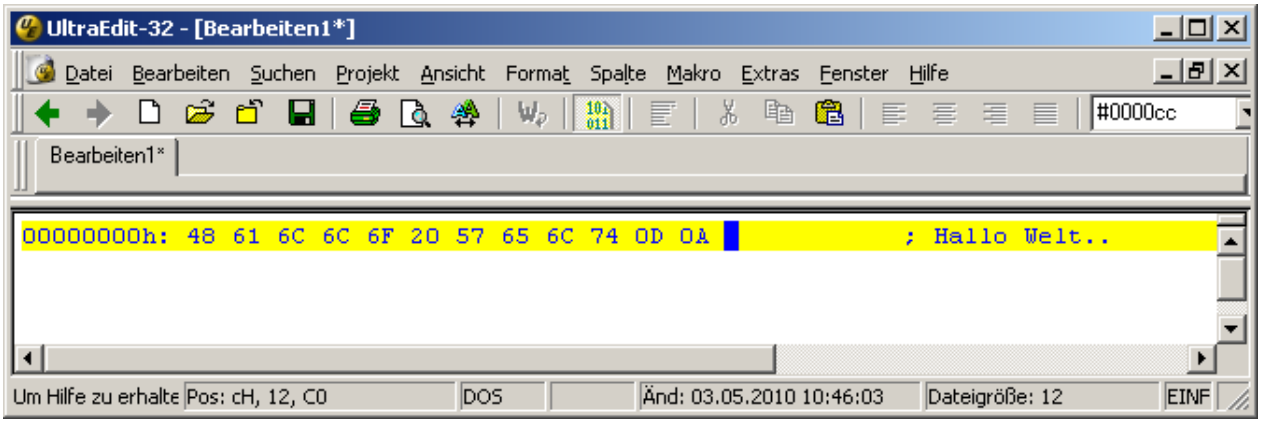
Mit steigender Speicherverfügbarkeit geht man heutzutage von 8-Bit Codes zu 16-Bit oder noch mehr stelligen Codes über. Hier gibt es den Standard UCS bzw. Unicode.

- [ISO10646]: "Information Technology -- Universal Multiple-Octet Coded Character Set (UCS) -- Part 1: Architecture and Basic Multilingual Plane", ISO/IEC 10646-1:1993.
- [UNICODE]: "The Unicode Standard: Version 2.0", The Unicode Consortium, Addison-Wesley Developers Press, 1996. Siehe auch http://www.unicode.org

Die ersten 128 Zeichen dieser Codes sind ASCII-kompatibel, die ersten 256 Zeichen sind kompatibel zu dem oben genannten Code ISO-Latin-1. Darüber hinaus codieren sie alle gängigen Zeichen dieser Welt.

Herkömmliche Programmiersprachen lassen meist keine Zeichen aus ASCII-Erweiterungen zu. Java und Groovy erlauben die Verwendung beliebiger Unicode-Zeichen in Strings. (Allerdings heißt dies noch lange nicht, dass jede Java-Implementierung einen Editor zur Eingabe von Unicode mitliefern würde!)

Zeichenketten (strings) werden üblicherweise durch Aneinanderfügen einzelner codierter Zeichen repräsentiert; da die Länge oftmals statisch nicht festliegt, werden sie intern mit einem speziellen Endzeichen abgeschlossen..
Beispiel:
Dem Text "Hallo Welt" entspricht die Zeichenfolge
"H", "a", "l", "l", "o", " ", "W", "e", "l", "t"
Diese wird in ASCII folgendermaßen codiert:
072 097 108 108 111 032 087 101 108 116.
In Hexadezimal-Schreibweise lautet diese Folge:
48 61 6C 6C 6F 20 57 65 6C 74.
Dem entspricht die Bitfolge:
01001000 01100001 01101100 01101100 01101111
00100000 01010111 01100101 01101100 01110100.
Obwohl diese Repräsentation sicher nicht die speichereffizienteste ist, ist sie weit verbreitet; bei Speichermangel greift man eher auf eine nachträgliche Komprimierung als auf andere Codes zurück.




## Zeit- und Raumangaben

Oft muss man Datumsangaben wie „4. Mai 2010, 9 Uhr 15 pünktlich“ oder Raumangaben, etwa „0310, Rudower Chaussee 26, 12489 Berlin-Adlershof“ in Programmen repräsentieren. Für Termine könnte man das Tripel (4, 5, 2010) durch drei natürliche Zahlen oder mit Uhrzeit als (4, 5, 2010, 9, 15, 0) in sechs Zahlen (Speicherwörtern) ablegen. Um Platz zu sparen, kann man auch eine Folge von 14, 8 oder 6 Zeichen verwenden: "20100504091500", "20100504", "100504". Im letzten Fall bekommt man allerdings ein Y2K-Problem, was sich aber auch im Januar 2010 im Kreditkartenwesen wiederholte.
Um den aktuellen Tag in nur einer ganzen Zahl zu codieren, könnte man eine Stellenwertrechnung wie bei den Gleitkommazahlen verwenden. Dies ist jedoch zu aufwändig; besser ist es, die Zeiteinheiten ab einem bestimmten Anfangszeitpunkt zu zählen. Die Römer zählten ab der vermuteten Gründung der Stadt; in der christlichen Zeit zählt man ab der vermuteten Geburt Jesu. In der Unixzeit zählt man die Sekunden ab dem 1.1.1970, wobei Schaltsekunden nicht mitgezählt werden. Dies kann ggf. zu einem „Jahr-2038-

Problem“ führen. Die Umrechnung von Unixzeit in christliche Zeit erfolgt mit dem Kommando `date`. Da die inkrementelle Zeitrechnung eine immer größer werdende Differenz zur durch die Erdrotation gegebene „natürliche“ Zeit aufweist, fanden öfters Anpassungen und Korrekturen statt, die einschneidendste durch die Einführung des Gregorianischen Kalenders am 4.10./14.10.1582. Seither erfolgt diese Korrektur durch die Einführung von Schalttagen und Schaltsekunden. Ein Schaltjahr ist definiert dadurch, dass die Jahreszahl durch 4 teilbar ist, aber nicht durch 100, oder durch 400.

```
boolean schaltjahr (x) {(x % 4 == 0) & (x % 100 != 0) | (x % 400 == 0)}
assert schaltjahr (2012) & schaltjahr (2000) & ! schaltjahr(1900)
```

Um zu einem gegebenen Datum im Gregorianischen Kalender den Wochentag zu ermitteln, kann man die Zellersche Formel (siehe http://de.wikipedia.org/wiki/Zellers_Kongruenz) verwenden.

Um die Zeitrepräsentation auf der ganzen Erde einheitlich referenzieren zu können, wurde 1968 die koordinierte Weltzeit UTC eingeführt. Die lokale Zeit ergibt sich durch Angabe des Offsets zur UTC, etwa `11:30 UTC+1:00` für die mitteleuropäische Ortszeit (MEZ oder CET), die der UTC eine Stunde voraus ist. Die mitteleuropäische Sommerzeit (CEST) ist `UTC+2:00`. Als Alternative zur koordinierten Weltzeit wurde 1998 die Internetzeit eingeführt (und von der Firma Swatch propagiert), bei der ein Tag aus 1000 Zeiteinheiten besteht und die überall gleich gezählt wird.

In Groovy und in der Java-Bibliothek gibt es die Klassen Date und GregorianCalendar, mit der man direkt mit Datumsangaben rechnen kann:

```
def t = new Date ()
println t // heutiges Datum
println t+7 // eine Woche später
println t.time // Unixzeit
def c=new GregorianCalendar()
println c.time // heutiges Datum
println c.timeInMillis // etwas später...
```

Zur Repräsentation von Ortsinformationen auf der Erde gibt es sehr viele unterschiedliche Postanschriftssysteme. Für Koordinatenzuweisungen beim Geo-tagging hat sich das System (Längengrad, Breitengrad) durchgesetzt.

### Darstellung sonstiger Informationen

Natürlich lassen sich in einem Computer nicht nur Bits, Zahlen und Zeichen repräsentieren, sondern z.B. auch visuelle und akustische Informationen. Für Bilder gibt es dabei prinzipiell zwei verschiedene Verfahren, nämlich als Vektor- und als Pixelgrafik. Auch für Töne gibt es verschiedene Repräsentationsformen: als Folge von Noten (*Midi*), Schwingungsamplituden (*wav, au*) oder komprimiert (*mp3*). Diese Repräsentationsformen lassen sich jedoch auffassen als strukturierte Zusammensetzungen einfacher Datentypen; wir werden später noch detailliert auf verschiedene Strukturierungsmöglichkeiten für Daten eingehen.
**Wichtig:** Der Bitfolge sieht man nicht an, ob sie die Repräsentation einer Zeichenreihe, einer Folge von ganzen Zahlen oder reellen Zahlen in einer bestimmten Genauigkeit ist. Ohne Decodierungsregel ist eine codierte Nachricht wertlos!

## 2.2 Sprachen, Grammatiken, Syntaxdiagramme

Die obigen Darstellungsformen sind geeignet zur Repräsentation einzelner Objekte. Häufig steht man vor dem Problem, Mengen von gleichartigen Objekten repräsentieren zu müssen. Für endliche Mengen kann dies durch die Folge der Elemente geschehen; für unendliche Mengen geht das im Allgemeinen nicht. Daher muss man sich andere (symbolische) Darstellungsformen überlegen.

### Darstellung von Sprachen

Ein besonders wichtiger Fall ist die Darstellung von unendlichen Mengen von Wörtern, die einem bestimmten Bildungsgesetz unterliegen; zum Beispiel die Menge der syntaktisch korrekten Eingaben in einem Eingabefeld, oder die Menge der Programme einer Programmiersprache.
Eine ***Sprache*** ist eine Menge von Wörtern über einem Alphabet **A.**

- z.B. {a, aab, aac}
- z.B. Menge der grammatisch korrekten Sätze der dt. Sprache
- z.B. Menge der Groovy-Programme

Man unterscheidet zwischen ***natürlichen Sprachen*** wie z.B. deutsch und englisch und ***formalen Sprachen*** wie z.B. Java, Groovy, C++ oder der Menge der Primzahlen in Hexadezimaldarstellung. Da Leerzeichen in der Lehre von den formalen Sprachen genau wie andere Zeichen behandelt werden, gibt es hier keinen Unterschied zwischen Wörtern und Sätzen.

Unter ***Syntax*** versteht man die Lehre von der Struktur einer Sprache

- Welche Wörter gehören zur Sprache?
- Wie sind sie intern strukturiert?
  z.B. Attribut, Prädikatverbund, Adverbialkonstruktion

Unter ***Semantik*** versteht man die Lehre von der Bedeutung der Sätze

- Welche Information transportiert ein Wort der Sprache?

In der Linguistik betrachtet man manchmal noch den Begriff ***Pragmatik,*** darunter versteht man die Lehre von der Absicht von sprachlichen Äußerungen

- Welchen Zweck verfolgt der Sprecher mit einem Wort?

Berühmtes Beispiel für den Unterschied zwischen Semantik und Pragmatik ist die Beifahrer-Aussage „Die Ampel ist grün“.

**Grammatiken**
Grammatiken sind Werkzeuge zur Beschreibung der Syntax einer Sprache. Eine Grammatik stellt bereit

- **A** : ***Alphabet*** oder „***Terminalzeichen***“
- **H** : ***Hilfssymbole*** = syntaktische Einheiten (<Objekt<, <Attribut>,..) oder „***Nonterminalzeichen***“

Aus A und H bildet man ***Satzformen*** (Schemata korrekter Sätze)
z.B. “<Subjekt> <Prädikat> <Objekt>”, “ Heute <Prädikat> < Subjekt> <Objekt> “

- **R**: ***Ableitungsregeln*** – erlaubte Transformationen auf Satzformen
- **s**: Ein ausgezeichnetes *Hilfssymbol* („Gesamtsatz”, „***Axiom***“)

Formal ist eine ***Grammatik*** ein Tupel G = [**A, H, R, s**], wobei

- **A** und **H** Alphabete sind mit $\mathbf{A} \cap \mathbf{H} = \emptyset$
- $\mathbf{R} \subseteq (\mathbf{A} \cup \mathbf{H})^{+} \times (\mathbf{A} \cup \mathbf{H})^{*}$
- $\mathbf{s} \in \mathbf{H}$

Die Relation **R** wird meist mit dem Symbol → oder ::= (in Infixschreibweise) notiert.
Zwischen Satzformen definieren wir eine Relation –> (***direkte Ableitungsrelation***)

w–>w’, falls $w = w_1 \circ u \circ w_2$, w’ = $w_1 \circ v \circ w_2$ und $(u,v) \in \mathbf{R}$

Die ***Ableitungsrelation*** => ist die reflexiv-transitive Hülle der direkten Ableitungsrelation:

w=>w’, falls w=w’ oder es gibt ein $v \in (\mathbf{A} \cup \mathbf{H})^*$ mit w–>v und v=>w’

Die von der Grammatik **G *beschriebene Sprache*** $L_G$ ist definiert durch

$\mathbf{L_G} = \{ w \mid s => w \text{ und } w \in \mathbf{A}^* \}$

**Beispiel**
**A**={“große“, “gute“, “jagen“, “lieben”, “Katzen”, “Mäuse”}
**H**={<attribut>,<objekt>,<prädikatsverband>,<satz>,<subjekt>,<substantiv>,<verb>}
**s=**<satz>
**R**: <satz> → <subjekt> <prädikatsverband> “.”
<subjekt> → <substantiv>
<subjekt> → <attribut> <substantiv>
<attribut>→“gute”
<attribut>→“große”
<substantiv>→“Katzen”
<substantiv>→“Mäuse”
<prädikatsverband>→<verb> <objekt>
<verb>→“lieben”
<verb>→“jagen”
<objekt>→<substantiv>
<objekt>→<attribut> <substantiv>

**Beispiel für Generierung**:
<satz> –> <subjekt> <prädikatverband> “.”
–> <attribut> <substantiv> <verb> <objekt> “.”
–> “gute” “Katzen” “jagen” “Mäuse” “.”

**Beispiel für Akzeptierung:**

| | | | | | |
|---|---|---|---|---|---|
| | “Katzen | lieben | große | Mäuse | .” |
| <– | <substantiv> | <verb> | <attribut> | <substantiv> | “.” |
| <– | <subjekt> | <verb> | <objekt> | | “.” |
| <– | <subjekt> | <prädikatverband> | | | “.” |
| <– | <satz> | | | | |

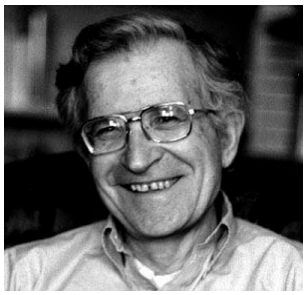
**Noch ein Beispiel**

<S> → <E>
<S> → <S> “+” <E>
<E> → <T>
<E> → <E> “•” <T>
<T> → <F>
<T> → “-” <F>
<F> → “x”
<F> → “0”
<F> → “1”
<F> → “(“ <S> “)”

Ableitung aus dem Beispiel: <S> => “1•(1+x)+0•-1”

<S> –> <S> “+” <E>
–> <E> “+”<E> “•” <T>
–> <E>“•” <T> “+”<T>“ •” “-” <F>
–> <T>“•” <F> “+” <F>“•” “-” “1”
–> <F>“•” “ (“ <S> “)” “+” “0” “•” “-” “1”
–> “1” “•” “(“ <S>“+” <E> “)” “+” “0” “•” “-” “1”
–> “1” “•” “(“ <E>“+” <T> “)” “+” “0” “•” “-” “1”
–> “1” “•” “(“ <T>“+” <F> “)” “+” “0” “•” “-” “1”
–> “1” “•” “(“ <F>“+” “x” “)” “+” “0” “•” “-” “1”
–> “1” “•” “(“ “1” “+” “x” “)” “+” “0” “•” “-” “1”

## Grammatiktypen – Die Chomsky-Hierarchie



Die Lehre von den Grammatiken wurde vom amerikanischen Linguisten Noam Chomsky (geb. 1928, http://web.mit.edu/linguistics/people/faculty/chomsky/index.html) („America's most prominent political dissident“, http://www.zcommunications.org/chomsky/index.cfm) in den späten 1950-ern entwickelt. Chomsky unterscheidet vier Typen von Grammatiken:

- **Typ 0** - beliebige Grammatiken
- **Typ 1** - ***kontextsensitive*** Grammatiken
  - Ersetzt wird ein einziges Hilfssymbol, nichtverkürzend
  - Alle Regeln haben die Form u+h+w → u+v+w mit h ∈ H
  - u , w heißen linker bzw. rechter Kontext
  - Sonderregel für das leere Wort
- **Typ 2** – ***kontextfreie*** Grammatiken
  - Ersetzt wird ein einziges Hilfssymbol, egal in welchem Kontext
  - Alle Regeln haben die Form h → v mit h ∈ H
- **Typ 3** – ***reguläre*** Grammatiken
  - Wie Typ 2, in v kommt aber max. ein neues Hilfssymbol vor, und zwar ganz rechts

Abhängig vom Typ spricht man auch von einer ***Chomsky-i-Grammatik***.

**Beispiele** (**A**={a, b, c}, **H**={s, x, y, z, …}):
Chomsky-0-Regeln sind z.B. die folgenden: xyz ::= zyx, abc ::= x, axby ::= abbxy
Eine Chomsky-0-Grammatik für $\{a^n b^n c^n \mid n>0\}$ erhält man durch folgende Regeln:
s::= xs’z; s’ ::= s’s’; s’ ::= abc; ba ::= ab; ca ::= ac; cb ::= bc;
xa ::= ax; x ::= y; yb ::= by; y ::= z; zc ::= cz; zz ::=ε

xyz ::= xyxyz (Chomsky-1-Regel) Anwendung: axyzxa -> axyxyza -> axyxyxyza -> …
y ::= yxy (Chomsky-2-Regel)
x::= ax; x::= b (Chomsky-3-Sprache $a^*b$ )

Eine Sprache heißt ***Chomsky-i-Sprache,*** wenn es eine Chomski-i-Grammatik für sie gibt, aber keine Chomsky-(i-1)-Grammatik. Die vier Typen bilden eine echte Hierarchie, d.h., für i=0,1,2 kann man jeweils eine Sprache finden, die durch eine Chomsky-i-Grammatik, nicht aber durch eine Chomsky-i+1-Grammatik beschreibbar ist. Es gilt:

- Mit beliebigen Grammatiken lassen sich alle Sprachen beschreiben, die überhaupt berechenbar sind
- Kontextsensitive Grammatiken sind algorithmisch beherrschbar
- Für die meisten Programmiersprachen wird die Syntax in Form kontextfreier Grammatiken angegeben.
- Einfache Konstrukte innerhalb von Programmiersprachen (z.B. Namen, Zahlen) werden durch reguläre Grammatiken beschrieben.

**Aufschreibkonventionen für kontextfreie Grammatiken**
***Backus-Naur-Form (BNF)*** verwendet u ::= v | w als Abkürzung für {u ::= v, u ::= w}

**Beispiel:**

```
<S> ::= <Expression> | <S> + <Expression>
<Expression> ::= <Term> | <Expression> • <Term>
<Term> ::= <Factor> | - <Factor>
<Factor> ::= x | 0 | 1 | ( <S> )
```

***Erweiterte Backus-Naur-Form (EBNF)*** löst direkte Rekursion durch beliebigmalige Wiederholungsklammern {} auf

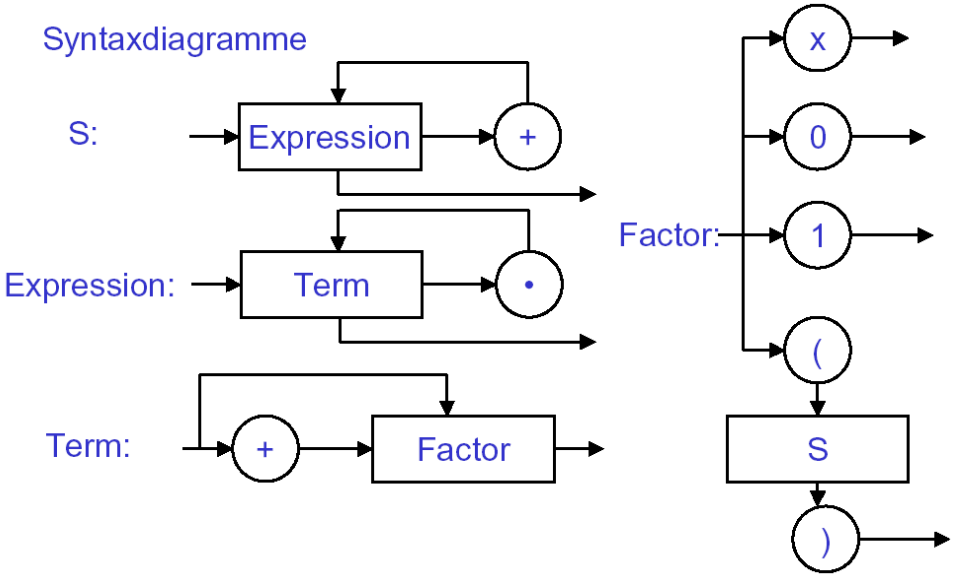
**Beispiel:**

```
S = Expression { “+” Expression }
Expression = Term { “•” Term }
Term = [ “-” ] Factor
Factor = “x” | “0” | “1” | “(” S “)”
```

Manchmal verwendet man auch zählende Wiederholungen $[...]_i^j$ mit der Bedeutung „…mindestens *i* und höchstens *j* mal“, wobei *j* auch durch einen Stern ersetzt werden kann (für „beliebig mal“). Es gilt $\{x\} = [x]_0^*$ und $[x] = [x]_0^1$.

**Beispiele:**
$[a]_0^3 = \{\varepsilon, a, aa, aaa\}$; $[a]_2^*$ = {aa, aaa, aaaa, …}

### *Syntaxdiagramme*




Syntaxdiagramme werden zur anschaulichen Notation von Programmiersprachen verwendet.

### *Reguläre Ausdrücke*

Für reguläre Sprachen gibt es die Möglichkeit, sie durch reguläre Ausdrücke aufzuschreiben. Das sind Ausdrücke, die gebildet sind aus

- der leeren Sprache, die keine Wörter enthält
- den Sprachen bestehend aus einem Wort bestehend aus einem Zeichen des Alphabets
- der Vereinigung von Sprachen (gekennzeichnet durch +)
- Konkatenation (gekennzeichnet durch Hintereinanderschreibung, · oder ;)
- beliebiger Wiederholung (gekennzeichnet durch *)

**Beispiele:**

($aa$)* (gerade Anzahl von a)
$aa$* (mindestens ein $a$, auch $a^+$ geschrieben)
(($a$ + $b$)$b$)* (jedes zweite Zeichen ist $b$)

Reguläre Ausdrücke werden z.B. in Editoren verwendet, um bestimmte zu suchende Zeichenreihenmengen zu repräsentieren. Beispiel für eine Suche ist: „Suche eine Zeichenreihe, die mit c beginnt und mit c endet und dazwischen eine gerade Anzahl von a enthält.“
In Groovy sind reguläre Ausdrücke („*patterns*“) ein fester Bestandteil der Sprache. Sie werden in / … / eingeschlossen, und es gibt vielfältige Operatoren:
==~ prüft ob eine Zeichenreihe in einem regulären Ausdruck enthalten ist
=~ gibt ein Matcher-Objekt

Regular Expression Operators

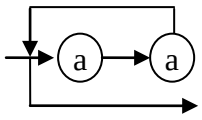
| | |
|---|---|
| a? | matches 0 or 1 occurrence of *a* |
| a* | matches 0 or more occurrences of *a* |
| a+ | matches 1 or more occurrences of *a* |
| a\|b | match *a* or *b* |
| . | match any single character |
| [woeirjsd] | match any of the named characters |
| [1-9] | match any of the characters in the range |
| [^13579] | match any characters not named |
| (ie) | group an expression (for use with other operators) |
| ^a | match an *a* at the beginning of a line |
| a$ | match an *a* at the end of a line |

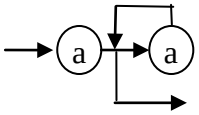
Beispiele:

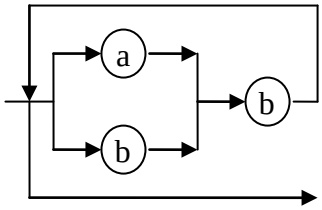
```
"aaaa" ==~ /(aa)+/
"abbbab" ==~ /(.b)*/
"..." ==~ /\.*/
"Hallo Welt" ==~ /\w+\s\w+/
["Hut", "Rot", "Rat"].each {assert it ==~ /(H|R)[aeiou]t/}
["cc", "caac", "cabac"].each {assert it ==~ /c[^a]*(a[^a]*a)*[^a]*c/}
```

*Endliche Automaten* sind Syntaxdiagramme ohne Abkürzungen, d.h. es gibt keine „Kästchen“. Ein endlicher Automat hat genau einen Eingang und mehrere mögliche Ausgänge.

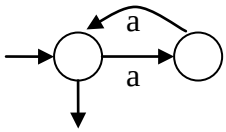
Beispiele:

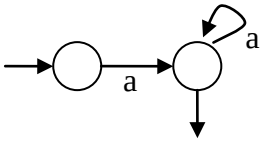


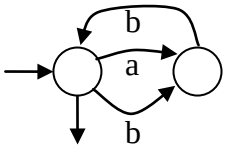




Üblicherweise werden bei Automaten die Kreuzungen als Kreise gemalt und *Zustände* (states) genannt, die Zustände werden mit Pfeilen verbunden (sogenannten *Transitionen*, transitions), die mit Terminalsymbolen beschriftet sind.




Ein fundamentaler Satz der Theorie der formalen Sprachen besagt, dass mit regulären Ausdrücken genau die durch reguläre Grammatiken definierbaren Sprachen beschrieben werden können, welches wiederum genau die Sprachen sind, die durch endliche Automaten beschrieben werden können.

## 2.3 Darstellung von Algorithmen

Ein ***Algorithmus*** ist ein *präzises*, *schrittweises* und *endliches* Verfahren zur Lösung eines Problems oder einer Aufgabenstellung (insbesondere zur Verarbeitung von Informationen, vgl. Kap. 0). Das bedeutet, an einen Algorithmus sind folgende Anforderungen zu stellen:

- Präzise Beschreibung (relativ zu den Kommunikationspartnern) der zu bearbeitenden Informationen und ihrer Repräsentation

- Explizite, eindeutige und detaillierte Beschreibung der einzelnen Schritte (relativ zu der Person oder Maschine, die den Algorithmus ausführen soll)
- Endliche Aufschreibung des Algorithmus, jeder Einzelschritt ist in endlicher Zeit effektiv ausführbar, und jedes Ergebnis wird nach endlich vielen Schritten erzielt.

Um Algorithmen von einem Computer ausführen zu lassen, müssen sie (genau wie andere Informationen) in einer für die jeweilige Maschine verständlichen Form *repräsentiert* werden. Eine solche Repräsentation nennt man ***Programm***. Zum Beispiel kann das eine Folge von Maschinenbefehlen sein, die ein bestimmter Rechner ausführen kann. Die tatsächliche Ausführung eines Algorithmus bzw. Programms nennt man einen ***Prozess.*** Sie findet auf einem (menschlichen oder maschinellen) ***Prozessor*** statt. Ein Algorithmus bzw. Programm ***terminiert***, wenn seine Ausführung nach einer endlichen Zahl von Schritten (Befehlen) abbricht. Ein Algorithmus (bzw. ein Programm) heißt ***deterministisch***, wenn für jeden Schritt der nächste auszuführende Schritt eindeutig definiert ist. Er bzw. es heißt ***determiniert***, wenn die Berechnung nur ein mögliches Ergebnis hat.

**Beispiel:** Wechselgeldbestimmung (nach Kröger / Hölzl / Hacklinger)
*Aufgabe:* Bestimmung des Wechselgeldes *w* eines Fahrscheinautomaten auf 5 Euro bei einem Preis von *p* Euro (zur Vereinfachung fordern wir, *p* sei ganzzahliges Vielfaches von 10 Cent; es werden nur 10-, 20- und 50-Cent Münzen zurückgegeben).
Der Algorithmus ist nicht determiniert! Damit das Ergebnis eindeutig bestimmt ist, legen wir zusätzlich fest: „es sollen möglichst wenige Münzen zurückgegeben werden".
Als erstes muss die Repräsentation der Schnittstellen festgelegt werden: *p* könnte zum Beispiel als natürliche Zahl in Cent, als rationale Zahl oder als Tupel (Euro, Cent) repräsentiert werden, und *w* als Multimenge oder Folge von Münzen. Dann erfolgt die Aufschreibung des Algorithmus.

*1. Darstellung* in **natürlicher Sprache**: „Die Rückgabe enthält maximal eine 10-Cent-Münze, zwei 20-Cent-Münzen, und den Rest in 50-Cent-Münzen. 10-Cent werden zurückgegeben, falls dies nötig ist, um im Nachkommabereich auf 0, 30, 50, oder 80 Cent zu kommen. Eine oder zwei 20-Cent-Münzen werden zurückgegeben, um auf 0, 40, 50 oder 90 Cent zu kommen. Wenn der Nachkommaanteil 0 oder 50 ist, werden nur 50-Cent-Münzen zurückgegeben."

Ein Vorteil der natürlichen Sprache ist, dass sie sehr flexibel ist und man damit (fast) alle möglichen Ideen aufschreiben kann. Nachteile bei der Darstellung von Algorithmen in natürlicher Sprache sind, dass keine vollständige formale Syntax bekannt ist, d.h. es ist nicht immer leicht festzustellen, ob eine Beschreibung syntaktisch korrekt ist oder nicht, und dass die Semantik weitgehend der Intuition überlassen bleibt. Dadurch bleibt immer eine gewisse Interpretationsfreiheit, verschiedene Prozessoren können zu verschiedenen Ergebnissen gelangen.

*2. Darstellung* als nichtdeterministischer **Pseudo-Code**:
**Sei** $w=\{\}$ Multimenge;
**Solange** $p<5$ **tue**
[] erste Nachkommastelle von $p \in \{2,4,7,9\}$:
$p = p + 0.10$, $w = w \cup \{10c\}$
[] erste Nachkommastelle von $p \in \{1,2,3,6,7,8\}$:
$p = p + 0.20$, $w = w \cup \{20c\}$
[] erste Nachkommastelle von $p \in \{0,1,2,3,4,5\}$:
$p = p + 0.50$, $w = w \cup \{50c\}$
**Ergebnis** *w*

Auch Pseudo-Code verwendet keine feste Syntax, es dürfen jedoch nur solche Sprachelemente verwendet werden, deren Bedeutung (Semantik) klar definiert ist. Damit ist die Menge der möglichen Ergebnisse zu einer Eingabe eindeutig bestimmt.

Berechnungsbeispiel:
$p$=3.20, $w$={} → $p$=3.70, $w$={50c} → $p$=3.80, $w$={50c, 10c} → $p$=4.00, $w$={50c, 10c, 20c} → $p$=4.50, $w$={50c, 10c, 20c, 50c} → $p$=5.00, $w$={50c, 10c, 20c, 50c, 50c}

*3. Darstellung* **mathematisch-rekursiv**:

$$\text{Wechselgeld}(p) = \begin{cases} \{\}, \text{ falls } p=5 \\ 0{,}50+\text{Wechselgeld}(p+0.50), \text{ falls } p\neq 5 \text{ und } 5-p \geq 0.50 \\ 0{,}20+\text{Wechselgeld}(p+0.20), \text{ falls } p\neq 5,\ 5-p < 0.50 \text{ und } 5-p \geq 0.20 \\ 0{,}10+\text{Wechselgeld}(p+0.10), \text{ sonst} \end{cases}$$

Beispiel zum Berechnungsablauf:
Wechselgeld(3.20) =0.50+Wechselgeld(3.70) =0.50+0.50+Wechselgeld(4.20) =0.50+0.50+0.50+Wechselgeld(4.70) =0.50+0.50+0.50+0.20+Wechselgeld(4.90) =0.50+0.50+0.50+0.20+0.10+Wechselgeld(5.00) =0.50+0.50+0.50+0.20+0.10

Die Syntax und Semantik in dieser Aufschreibung folgt den üblichen Regeln der Mathematik bzw. Logik, ist fest definiert, aber jederzeit erweiterbar. In der Schreibweise der Informatik könnte man dasselbe etwa wie folgt aufschreiben:

Wechselgeld ($p$) =
Falls 5-$p$=0 dann Rückgabe({}) sonst
falls 5-$p$ ≥ 0.50 dann Rückgabe(0.50+Wechselgeld($p$+0.50)) sonst
falls 5-$p$ ≥ 0.20 dann Rückgabe(0.20+Wechselgeld($p$+0.20)) sonst
Rückgabe(0.10+Wechselgeld($p$+0.10))

Diese Notation lässt sich unmittelbar in funktionale Programmiersprachen übertragen. Als Groovy--Programm aufgeschrieben sieht das etwa so aus:

```
int wg(p){
  if (5-p==0) return(0) else
   if (5-p>=0.50) return(100+wg(p+0.50)) else
    if (5-p>=0.20) return(10+wg(p+0.20)) else
     return(1+wg(p+0.10))}
```

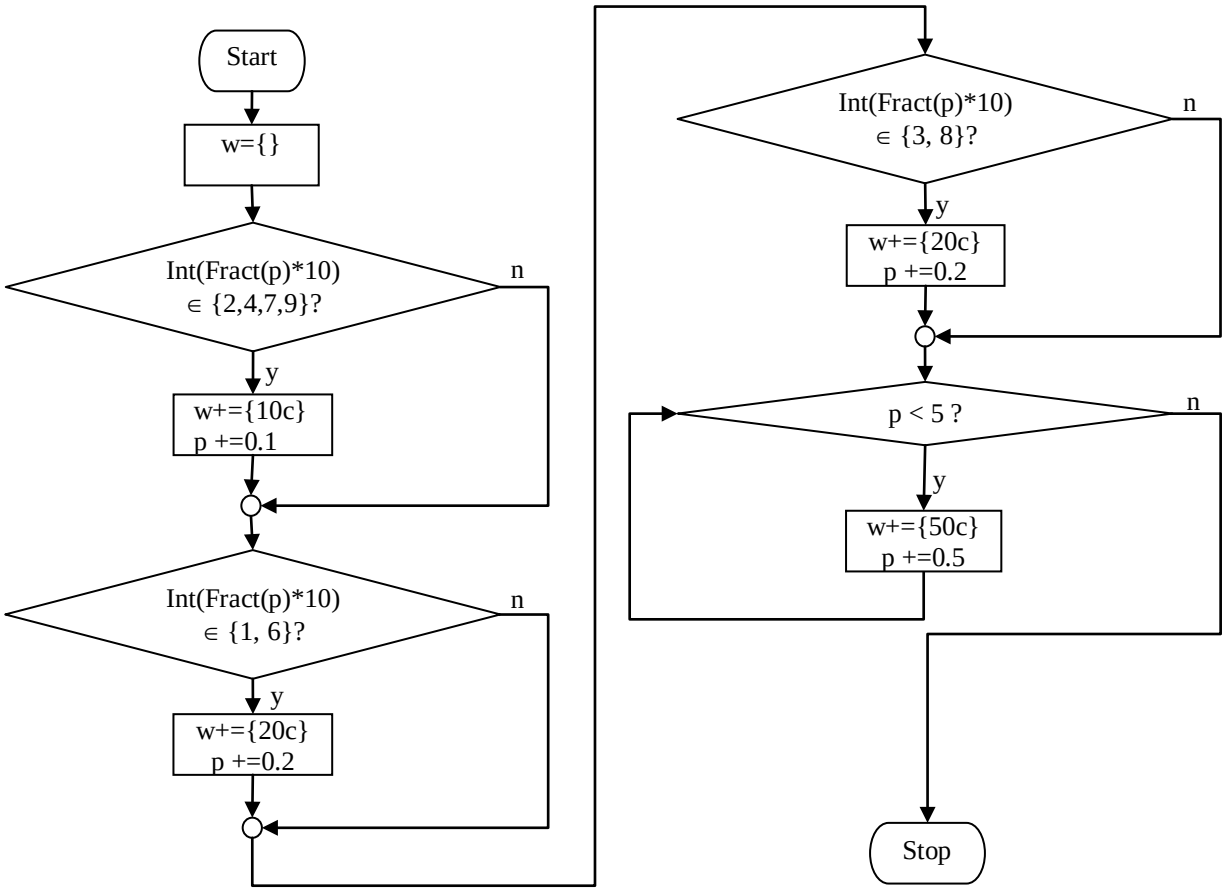
Hier wählen wir zur Darstellung des Ergebnisses eine dreistellige natürliche Zahl, deren erste Stelle die Anzahl der 50-Cent-Münzen, die zweite die Anzahl der 20-Cent-Münzen und die letzte die Anzahl der 10-Cent-Münzen angibt.

```
wg(2.70)
411
wg(3.10)
320
```

Eine andere Form der Ergebnisdarstellung (Multimenge!) wäre etwa als Liste:

```
def wg(p){
  if (5-p==0) return([]) else
  if (5-p>=0.50) return([50] + wg(p+0.50)) else
  if (5-p>=0.20) return([20] + wg(p+0.20)) else
    return([10] + wg(p+0.10))}
assert wg(2.70) == [50, 50, 50, 50, 20, 10]
```

*4. Darstellung* als **Ablaufdiagramm** oder **Flussdiagramm.**




Ein Ablaufdiagramm ist prinzipiell ein endlicher Automat! Zusätzlich dürfen jedoch Variablen, Bedingungen, Zuweisungen, Unterprogramme und andere Erweiterungen verwendet werden.

Ausführungsbeispiel:
*p*=3.20, *w*={} → *p*=3.30, *w*={10c} → *p*=3.50, *w*={10c, 20c} → *p*=4.00, *w*={10c, 20c, 50c} → *p*=4.50, *w*={10c, 20c, 50c, 50c} → *p*=5.00, *w*={10c, 20c, 50c, 50c, 50c}

*5. Darstellung* als **Groovy- oder Java-Programm** (mit heftigem Gebrauch von Mathematik)

**Groovy:**

```
int fuenfzigCentStuecke(float p) {
    ((5 - p) * 10 / 5)}
int zehnCentStuecke(float p) {
    int cent = (p-(int)(p)) * 100
    cent in [20, 40, 70, 90] ? 1 : 0}
int zwanzigCentStuecke(float p) {
    int tencent = ((p-(int)(p)) * 10)
    coins = [0,2,1,1,0,0,2,1,1,0]
    coins[tencent]}
```

**Java:**

```
static int fünfzigCentStücke(double p) {
      return (int)((5 - p) * 10 / 5) ;}
static int zehnCentStücke(double p) {
      int cent = (int)((p-(int)(p)) * 100);
      return((cent==20 || cent==40 || cent==70 || cent==90) ? 1 : 0) ;}
static int zwanzigCentStücke(double p) {
      int tencent = (int)((p-(int)(p)) * 10);
      int[] coins = {0,2,1,1,0,0,2,1,1,0};
      return coins[tencent] ;}
```

Syntax und Semantik sind eindeutig definiert, jedes in einer Programmiersprache aufschreibbare Programm ist normalerweise determiniert und deterministisch (Ausnahme: Verwendung paralleler Threads). Wie man in diesem Fall auch leicht sieht, terminiert das Programm auch für jede Eingabe. Eine wichtige Frage ist die nach der *Korrektheit*, d.h. berechnet das Programm wirklich das in der Aufgabenstellung verlangte?

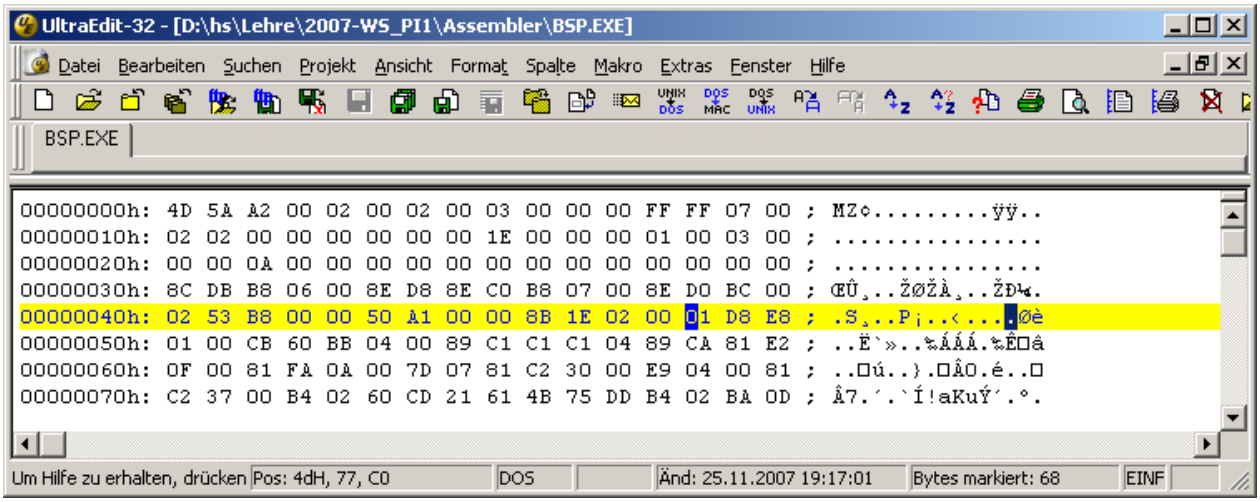
*6. Darstellung* als **Assemblerprogramm**

In der Frühzeit der Informatik wurden Rechenmaschinen programmiert, indem eine Folge von Befehlen angegeben wurde, die direkt von der Maschine ausgeführt werden konnte. Assemblersprachen sind Varianten solcher Maschinensprachen, bei denen die Maschinenbefehle in mnemotechnischer Form niedergeschrieben sind. Häufig verwendete Adressen (Nummern von Speicherzellen) können mit einem Namen versehen werden und dadurch referenziert werden. Viele Assemblersprachen erlauben auch *indirekte Adressierung*, d.h. der Inhalt einer Speicherzelle kann die Adresse einer anderen Speicherzelle. Die in einer Assemblersprache verfügbaren Befehle hängen stark von der Art der zu programmierenden Maschine ab. Typische Assemblerbefehle sind z.B. mov rx ry (Bedeutung: transportiere / kopiere den Inhalt von rx nach ry) oder jgz rx lbl (Bedeutung: wenn der Inhalt von rx größer als Null ist, gehe zur Sprungmarke lbl)

```
// Eingabe: Preis in Register p in Cent (wird zerstört)
// Ergebnisse in fc, zc, tc (fünfzigCent, zwanzigCent, tenCent)
mov 0, fc
mov 0, zc
mov 0, tc
loop:
  mov p, ac  // lade p in den Akkumulator
  sub ac, 450 // subtrahiere 450
  jgz hugo // jump if greater zero to hugo
  add ac, 500 // addiere 500
  mov ac, p // speichere Akkumulator nach p
  mov fc, ac
  add ac, 1
  mov ac, fc // fc := fc + 1
  goto loop
hugo:
  mov p, ac  // wie oben
  sub ac, 480
  jgz erna
  add ac, 500
  mov ac, p
  mov zc, ac
  add ac, 1
  mov ac, zc
  goto hugo
erna:
  mov p, ac
  sub ac, 490
  jgz fertig
  mov 1, tc
fertig:
```

*7. Darstellung* als **Maschinenprogramm**

Ein Maschinenprogramm ist eine Folge von Befehlen, die direkt von einer Maschine eines dafür bestimmten Typs ausgeführt werden kann. Jedem Assemblerbefehl entspricht dabei eine bestimmte Anzahl von Bytes im Maschinenprogramm.



**Programmiersprachen**

Ein Programm ist, wie oben definiert wurde, die Repräsentation eines Algorithmus in einer Programmiersprache. Die Menge der syntaktisch korrekten Programme einer bestimmten Programmiersprache (JAVA, C, Delphi, …) wird im Allgemeinen durch eine kontextfreie Grammatik beschrieben. Maschinen- und Assemblersprachen sind dabei sehr einfache Sprachen (die sogar durch reguläre Grammatiken definiert werden könnten). Das Programmieren in einer Maschinensprache oder Assembler ist außerordentlich mühsam und sehr fehleranfällig. ***Höhere Programmiersprachen*** sind nicht an Maschinen orientiert, sondern an den Problemen. Programme, die in einer höheren Programmiersprache geschrieben sind, können nicht unmittelbar auf einem Rechner ausgeführt werden. Sie werden entweder von einem speziellen Programm ***interpretiert*** (d.h., direkt ausgeführt) oder von einem ***Compiler*** in eine Folge von Maschinenbefehlen ***übersetzt*** und erst dann ***ausgeführt.*** Bei einigen Prgrammiersprachen (Java, C#, USCD-Pascal) erfolgt die Übersetzung zunächst in die Maschinensprache einer *virtuellen Maschine*, d.h. einer nicht in Hardware realisierten Maschine, welche daher unabhängig von einer speziellen Hardwaretechnologie ist. Die Ausführung von Maschinenprogrammen der virtuellen auf einer realen Maschine erfolgt von speziellen Interpretern (der natürlich für jede reale Maschine neu entwickelt oder angepasst werden muss). Die Sprachen, die einer maschinellen Behandlung nicht zugänglich sind und von Menschen in Programme überführt werden müssen, nennen wir ***Spezifikationssprachen***.

Seit Beginn der Informatik wurden mehr als 1000 unterschiedliche Programmiersprachen erfunden, von denen die meisten in vielen verschiedenen Varianten und Dialekten existieren. Beispielsweise gibt es von Java inzwischen sieben Hauptvarianten (Versionen), und Groovy kann als eine Erweiterung von Java betrachtet werden. Im vierten Kapitel werden wir Möglichkeiten zur Klassifikation von Programmiersprachen betrachten.