

Inhalt

3	Hard- und Software eines Rechners (Teil II)	3-2
3.4	<i>Daten – codierte Informationen</i>	3-2
3.4.1	Externe und interne Daten	3-2
3.4.2	Interne Textdarstellung	3-5
3.4.3	Interne Zahlendarstellung – Ganze Zahlen $z \in \mathbb{Z}$	3-6
3.4.4	Interne Zahlendarstellung – Rationale Zahlen $z \in \mathbb{Q}$	3-7
3.5	<i>Software – Die Seele eines Rechners</i>	3-11
3.5.1	Ein Programm für einen Algorithmus	3-11
3.5.2	Programmhierarchien	3-13
3.5.3	Softwareklassifizierung	3-16
3.6	<i>Einführung der Begriffe in der Unterstufe</i>	3-17

Dateneinheiten

- **Bit** (binary digit) Ein einzelnes **Binärzeichen**, kleinste interne Organisationseinheit von Daten, hat den Wert 0 oder 1.
- **Byte** (8 Bit) **Kleinste adressierbare Einheit**, man kann den Inhalt eines Bytes lesen oder auf ein Byte schreiben.

1 Byte = 8 Bit

1 KB = 2^{10} = 1 024 Bytes KB ... Kilobyte

1 MB = 2^{20} = 1 048 576 Bytes MB ... Megabyte

1 GB = 2^{30} $\approx 10^9$ Bytes GB ... Gigabyte

ASCII-Code (*American Standard Code for Information Interchange*):

8-Bit-Zeichensatzes - Jedes externe Zeichen wird intern in einem Byte durch die Kombination der 8 Bit dargestellt.

1 Byte = 8 Bit $\Rightarrow 2^8 = 256$ verschiedene Zeichen.

\Rightarrow Inkompatible Kodierung in verschiedenen Ländern: Umlaute, ...

Unicode¹ (*Universal Character Set*), seit 1991:

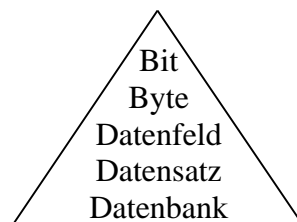
16-Bit-Zeichensatzes - Verschlüsselung eines Zeichens in 2 Bytes.

2 Byte = 16 Bit $\Rightarrow 2^{16} = 65\,536$ verschiedene Zeichen.

Alle wichtigen Sprachen und deren Besonderheiten können berücksichtigt werden. Der ASCII-Codes ist Bestandteil des Unicodes. Unicode wird zur Zeichenkodierung in **Java** verwendet.

- **Datenfeld** Bytefolge, der eine bestimmte Bedeutung zugeordnet wird: Möchte man einen Namen oder eine Kontonummer speichern, so benötigt man mehrere Bytes. *Zahlen* werden in mehreren Bytes hintereinander abgespeichert. Die Anzahl dieser zusammengehörigen Bytes muss bekannt sein.
- **Datensatz** Zusammenfassung zusammenhängende Datenfelder: Art und Anzahl der Datenfelder muss bekannt sein. Adressen bestehen zum Beispiel aus mehreren Datenfeldern wie Name, Vorname, Straße,
- **Datenbank** Zusammenfassung von Datensätzen gleichen Aufbaus: Art und Anzahl der Datensätze muss bekannt sein. Ein Adressbuch besteht aus einer Menge von Adressen, also aus mehreren Datensätzen gleicher Struktur.

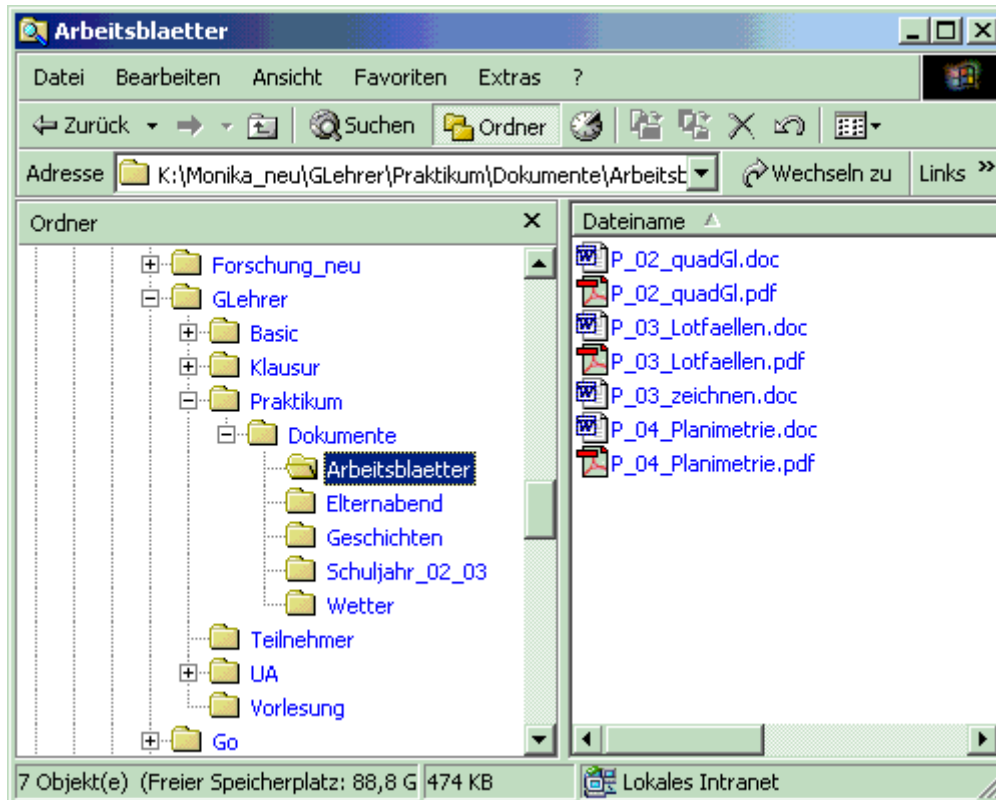
\Rightarrow **Hierarchie**



¹ Unicode-Konsortium <http://www.unicode.org/>

Dateiensystem

Daten und *Programme* werden in **Dateien (Files)** abgespeichert. Zusammengehörige Dateien werden in **Ordnern (Directories)** zusammengefasst. Diese können wiederum in Ordnern liegen. Mehrere Laufwerke lassen sich auf dem Rechner einrichten. Dadurch entsteht für jedes Laufwerk ein **Dateienbaum**: Beginnend mit der **Wurzel** wird eine *Dateihierarchie* aufgebaut. Durch diese Hierarchie und spezielle Dateinamenerweiterungen wird Übersicht und Orientierung auf den Speichermedien ermöglicht.



Erweiterungen zu den Dateinamen klassifizieren den Ursprung der Dateientwicklung und verbinden diese Datei mit einer Anwendung.

Erweiterung	Bedeutung
.txt	Text im ASCII-Code (Notepad)
.doc	Word-Dokument
.dot	Word-Vorlage
.odt	Writer-Dokument
.ott	Writer-Vorlage
.xls	Excel-Dokument
.ods	Calc-Dokument
.bmp	Bild-Dokument
.ppt	Power-Point-Dokument
.odp	Impress-Dokument
.pdf	Portables Dokument Format
.bas	Basic-Programm
.pas	Pascal-Programm
.c	C-Programm
.java	Java-Programm
.class	Bytecode eines Java-Programms
.exe	ausführbares Programm

3.4.2 Interne Textdarstellung

Text (engl. *String*) wird intern durch den *ASCII-Code* bzw. den *Unicode* seiner Zeichen dargestellt.

ASCII-Code-Tabelle

Länder spezifisch, hier Deutsch

0	NULL	32	space	64	@	96	`	128	€	160		192	À	224	à
1	SOH	33	!	65	A	97	a	129	⤎	161	ı	193	Á	225	á
2	STX	34	"	66	B	98	b	130	,	162	ç	194	Â	226	â
3	ETX	35	#	67	C	99	c	131	f	163	£	195	Ã	227	ã
4	EOT	36	\$	68	D	100	d	132	„	164	¤	196	Ä	228	ä
5	ENQ	37	%	69	E	101	e	133	...	165	¥	197	Å	229	å
6	ACK	38	&	70	F	102	f	134	†	166	ı	198	Æ	230	æ
7	BEL	39	'	71	G	103	g	135	‡	167	§	199	Ç	231	ç
8	BS	40	(72	H	104	h	136	^	168	¨	200	È	232	è
9	HT	41)	73	I	105	i	137	‰	169	©	201	É	233	é
10	LF	42	*	74	J	106	j	138	Š	170	ª	202	Ê	234	ê
11	VT	43	+	75	K	107	k	139	<	171	«	203	Ë	235	ë
12	FF	44	,	76	L	108	l	140	Œ	172	¬	204	Ì	236	ì
13	CR	45	-	77	M	109	m	141	⤎	173	-	205	Í	237	í
14	SO	46	.	78	N	110	n	142	Ž	174	®	206	Î	238	î
15	SI	47	/	79	O	111	o	143	⤎	175	-	207	Ï	239	ï
16	DLE	48	0	80	P	112	p	144	⤎	176	°	208	Ð	240	ð
17	DC1	49	1	81	Q	113	q	145	‘	177	±	209	Ñ	241	ñ
18	DC2	50	2	82	R	114	r	146	’	178	²	210	Ò	242	ò
19	DC3	51	3	83	S	115	s	147	“	179	³	211	Ó	243	ó
20	DC4	52	4	84	T	116	t	148	”	180	´	212	Ô	244	ô
21	NAK	53	5	85	U	117	u	149	•	181	µ	213	Õ	245	õ
22	SYN	54	6	86	V	118	v	150	–	182	¶	214	Ö	246	ö
23	ETB	55	7	87	W	119	w	151	—	183	·	215	×	247	÷
24	CAN	56	8	88	X	120	x	152	~	184	¸	216	Ø	248	ø
25	EM	57	9	89	Y	121	y	153	™	185	¹	217	Ù	249	ù
26	SUB	58	:	90	Z	122	z	154	š	186	º	218	Ú	250	ú
27	ESC	59	;	91	[123	{	155	›	187	»	219	Û	251	û
28	FS	60	<	92	\	124		156	œ	188	¼	220	Ü	252	ü
29	QS	61	=	93]	125	}	157	⤎	189	½	221	Ý	253	ý
30	RS	62	>	94	^	126	~	158	ž	190	¾	222	Þ	254	þ
31	US	63	?	95	_	127	□	159	Ÿ	191	¿	223	ß	255	ÿ

Texte können unterschiedlich lang sein. Zur Kennzeichnung des Textendes werden zwei Methoden verwendet:

1. *Stringmethode* mit Endzeichen als Kennung (NULL)
2. *Puffermethode* mit Längenangabe

Beispiel für die Verwendung von ASCII-Code

```

Auto      => 65 117 116 111      => 41h 75h 74h 6Fh
           => 0100 0001 0111 0101 0111 0100 0110 1111
zu 1.     => 0100 0001 0111 0101 0111 0100 0110 1111 0000 0000
zu 2.     => 0000 0100 0100 0001 0111 0101 0111 0100 0110 1111
    
```

3.4.3 Interne Zahlendarstellung – Ganze Zahlen $z \in \mathbb{Z}$

Die Darstellung von Zahlen richtet sich nach deren Typ. So wird der Code von natürlichen, ganzen und rationalen Zahlen völlig unterschiedlich erzeugt. Wir wollen uns hier auf die Darstellung von Zahlen, die in **Java** auftreten, beschränken.

⇒ *Natürliche Zahlen* $z \in \mathbb{N}$: **Java kennt keine natürlichen Zahlen.**

⇒ *Ganze Zahlen* $z \in \mathbb{Z}$:

$z \geq 0$ *Direkter Code* - Dualdarstellung mit festgelegter Bitanzahl
 $z < 0$ *Komplement* des direkten Codes vom Betrag der Zahl z |

```

Dualdarstellung           63 = (11 1111)2

byte a = 63;              0011 1111
byte b = -63;             1100 0001

byte MAX = 127;          0111 1111
byte MIN = -128;         1000 0000
    
```

Beispiel: Darstellbare ganze Zahlen mit maximal 4 Bit (Vierbitzahl):

s	z'_2	z'_1	z'_0
---	--------	--------	--------

Dezimalzahl	Computerzahl	Berechnung
7	0111	$2^2 + 2^1 + 2^0 = 7$
6	0110	$2^2 + 2^1 = 6$
5	0101	...
4	0100	...
3	0011	...
2	0010	...
1	0001	$2^0 = 1$
0	0000	0

-1	1111	0001 \Rightarrow 1110 + 1
-2	1110	0010 \Rightarrow 1101 + 1
-3	1101	...
-4	1100	...
-5	1011	...
-6	1010	...
-7	1001	0111 \Rightarrow 1000 + 1
-8	1000	1000 \Rightarrow 0111 + 1

$$n = 4 \quad \text{MIN_VALUE} = -8 = -2^3 \quad \text{MAX_VALUE} = 7 = 2^3 - 1$$

Java:

Typ	kleinster Wert (MIN_VALUE)	größter Wert (MAX_VALUE)	Byte	Codierung
byte	-128 (-2^7)	127 (2^7-1)	1	Direktcode / Komplement
short	-32'768 (-2^{15})	32'767 ($2^{15}-1$)	2	
int	-2'147'483'648 (-2^{31})	2'147'483'647 ($2^{31}-1$)	4	
long	-9'223'372'036'854'775'808 (-2^{63})	9'223'372'036'854'775'807 ($2^{63}-1$)	8	UTF-16
char	0	65535 ($2^{16}-1$)	2	

3.4.4 Interne Zahlendarstellung – Rationale Zahlen $z \in \mathbb{Q}$

Wegen der Endlichkeit des Speichers kann nur ein Teil der rationalen Zahlen als sogenannte Gleitpunktzahlen im Rechner dargestellt werden.

\Rightarrow Rationale Zahlen $z \in \mathbb{Q}$

Darstellung einer rationalen Zahl als Gleitpunktzahl mit Mantisse M und Exponent E :

1. Ausgangspunkt ist die Dualdarstellung der Zahl.

$$3.15625 = 3 + 0.15625 = (11)_2 + (0.00101)_2 = (11.00101)_2$$

2. Einführung der wissenschaftlichen Notation.

Es erfolgt eine Normalisierung durch Stellenverschiebung auf **1.** ..., man erhält eine Mantisse M und einen Exponenten E :

$$(11.00101)_2 = (1.100101)_2 * (10)_2 = (1.100101)_2 * 2^1 \Rightarrow M = (1.100101)_2, E = 1$$

3. Darstellung als Maschinenzahl.

Die Mantisse M ist in ihrer Stellenzahl t beschränkt. Sie wird auf die entsprechende Stellenzahl gerundet. Da die Mantisse stets mit **1.** beginnt, lässt man aus Platzgründen diese **1** weg (*hidden bit*):

$$M \approx 1.M'$$

Exponenten E sind nur bis zu einer festgelegten Größe darstellbar, $E \in [r, R]$. Es können nicht beliebig kleine und beliebig große Zahlen dargestellt werden. Die Exponenten werden durch Addition einer Konstanten in den positiven Bereich verschoben:

$$E' = E + C > 0 \Rightarrow C = -r + 1$$

Maschineninterne Darstellung:

mit s als Vorzeichenbit + .. **0**; - .. **1**.

$$s \mid E' \mid M'$$

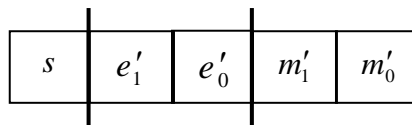
Eine **Sonderbehandlung** erfährt die Zahl **0**, sie wird nur durch Nullen codiert:

$$0 \mid 0\dots 0 \mid 0\dots 0$$

Die Menge der Maschinenzahlen ist durch die Stellenzahl t der Mantisse und durch den kleinsten und größten darstellbaren Exponenten r und R eindeutig bestimmt.

$$M_z(t, r, R)$$

Beispiel $M_z(3, -1, 1)$: Darstellbare positive rationale Zahlen mit 5 Bit (1 Vorzeichenbit, 2 Exponentenbits und 2 Mantissenbits):



$$M' = m'_1 m'_0 \text{ mit } M \approx 1.M' \text{ und } E' = e'_1 e'_0 \text{ mit } E' = E + 2.$$

$$z = 3.15625 = (11.00101)_2$$

$$\Rightarrow \text{Mantisse } M = (1.100101)_2, \text{ Exponent } E = 1$$

$$\Rightarrow M' = (10)_2, E' = 3 = (11)_2$$

$$\Rightarrow \text{Interne Darstellung: } 0 \mid 11 \mid 10$$

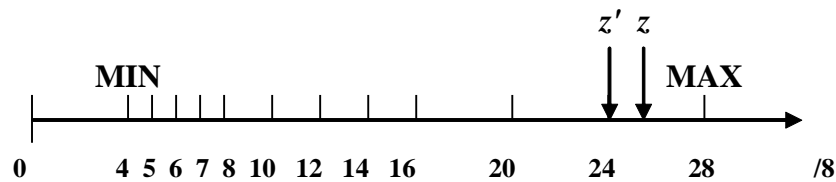
$$\Rightarrow z' = (1.10)_2 * 2^1 = (11)_2 = 3, \text{ d.h. intern wird } 3.15625 \text{ auf } 3 \text{ gerundet!}$$

Übersicht über alle darstellbaren positiven rationalen Zahlen mit 5 Bit als Maschinenzahlen

$$M_z(3, -1, 1)$$

Maschinenzahl	Wert	interne Darstellung
$(0.00)_2 * 2^0$	= 0	0 00 00
$(1.00)_2 * 2^{-1}$	= 4/8	0 01 00
$(1.01)_2 * 2^{-1}$	= 5/8	0 01 01
$(1.10)_2 * 2^{-1}$	= 6/8	0 01 10
$(1.11)_2 * 2^{-1}$	= 7/8	0 01 11
$(1.00)_2 * 2^0$	= 8/8	0 10 00
$(1.01)_2 * 2^0$	= 10/8	0 10 01
$(1.10)_2 * 2^0$	= 12/8	0 10 10
$(1.11)_2 * 2^0$	= 14/8	0 10 11
$(1.00)_2 * 2^1$	= 16/8	0 11 00
$(1.01)_2 * 2^1$	= 20/8	0 11 01
$(1.10)_2 * 2^1$	= 24/8	0 11 10
$(1.11)_2 * 2^1$	= 28/8	0 11 11

$\Rightarrow z'$



$$M_z(3, -1, 1) \quad \text{MIN} = 2^{-1} \quad \text{MAX} = 2 * 2^1 - 2^{-2} < 2^2$$

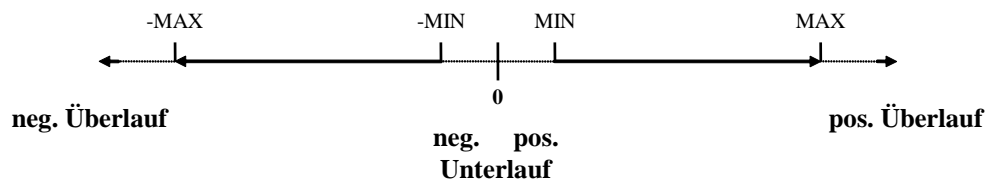
$$M_z(t, r, R) \quad \text{MIN} = 2^r \quad \text{MAX} = 2 * 2^R - 2^{-t+1} < 2^{R+1}$$

Der Zahlenstrahl mit den darstellbaren Zahlen lässt erkennen, dass es Bereiche gibt, in denen Zahlen nicht darstellbar sind und dass die Abstände der darstellbaren Zahlen von MIN nach MAX größer werden.

Die gleichen Aussagen treffen auch für die *negativen* Gleitpunktzahlen zu.

Zusammenfassung

- Wertebereich für Gleitpunkttypen: $[-MAX, -MIN] \cup \{0\} \cup [MIN, MAX]$



- Gleitpunktzahlen sind mit größer werdendem Betrag dünner darstellbar.
- Je größer die Mantissenstellenanzahl t , desto dichter die Zahlendarstellung.
- Das Exponentenintervall $[r, R]$ bestimmt die größte und die kleinste darstellbare Zahl.

Behandlung nicht darstellbarer Zahlen:

- $z > MAX$ oder $z < -MAX$:
Überlauf (**Infinity, -Infinity**) \Rightarrow Abbruch: **Laufzeitfehler**
- $-MIN < z < 0$ oder $0 < z < MIN$:
Unterlauf \Rightarrow **Sonderbehandlung**, Runden auf darstellbare Zahlen: **Rundungsfehler**
- $-MAX < z < -MIN$ oder $MIN < z < MAX$:
Maschinenzahlbereich \Rightarrow Runden auf darstellbare Zahlen: **Rundungsfehler**

Standard für den rationalen Zahlenbereiche:
(IEEE – Standard², Institute of Electrical and Electronics Engineers)

Bezeichnung	Byteanzahl	s [Bit]	M [Bit]	E [Bit]	r	R	C
single	4	1	24	8	-126	127	127
double	8	1	53	11	-1022	1023	1023

s Vorzeichenbit: 0 ... +, 1 ... -
 M Länge der Mantisse, einschließlich *hidden bit*
 E Anzahl der Exponentenbit
 r kleinster Exponent
 R größter Exponent
 C Verschiebungskonstante

² <http://www.h-schmidt.net/FloatApplet/IEEE754de.html>

Sonderbehandlung:

	$M' = 0$	$M' \neq 0$
$E' = 0$ ($E = r - 1$)	$z = 0.0$	$z = (-1)^s * 2^r * (0.M')_2$; zusätzliche Zahlen im Unterlauf
$E' = E'_{\max}$ ($E = R + 1$)	$z = (-1)^s * \infty$; -Infinity, Infinity	NaN (Not a Number); keine gültige Gleitpunktzahl

Beispiel für single (Java: float):

Dezimalzahl \Rightarrow Maschinenzahl

$$3.15625 = (11.0010\ 1)_2 = (1.1001\ 01)_2 * 2^1$$

$$\Rightarrow M' = 1001\ 01, E' = E + C = 1 + 127 = 128$$

$$\Rightarrow 0|100\ 0000\ 0|100\ 1010\ 0000\ 0000\ 0000\ 0000$$

$$0.1 = (0.00011)_2 = (1.1001)_2 * 2^{-4}$$

$$\Rightarrow M' = 100\ 1100\ 1100\ 11001100\ 1100, E' = E + C = -4 + 127 = 123$$

$$\Rightarrow 0|011\ 1101\ 1|100\ 1100\ 1100\ 11001100\ 1100$$

\Rightarrow **Die Zahl 0.1 ist als Maschinenzahl nicht exakt darstellbar.**

Maschinenzahl \Rightarrow Dezimalzahl

$$1|011\ 1111\ 1|000\ 0000\ 0000\ 0000\ 0000\ 0000$$

$$\Rightarrow M' = 0, E' = 127 \Rightarrow M = 1.0, E = E' - C = 127 - 127 = 0$$

$$\Rightarrow -1. * 2^0 = -1.0$$

Intern sind Datendarstellungen fehlerbehaftet.

IEEE:

Typ	single	double
MAX $< 2^{R+1}$	$\pm 3.40282347\ E +38$	$\pm 1.797693138462315750\ E +308$
MIN = 2^r	$\pm 1.17549435\ E -38$	$\pm 2.225073858507201383\ E -308$
MIN Sonderbehandlung	$\pm 1.40239846\ E -45$	$\pm 4.940656458412465\ E -324$
gültige Dezimalstellen	7	15

Java:

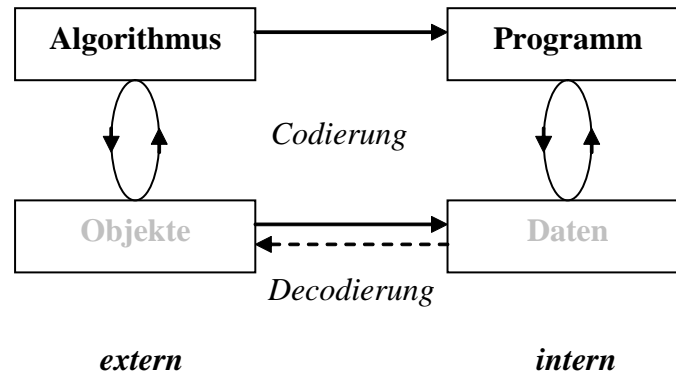
Typ	kleinster Wert (MIN_VALUE)	größter Wert (MAX_VALUE)	Codierung
float	$\pm 1.40239846\ E -45$	$\pm 3.40282347\ E +38$	IEEE 754 ³ , single
double	$\pm 4.940656458412465\ E -324$	$\pm 1.797693138462315750\ E +308$	IEEE 754, double

³ <http://www.h-schmidt.net/FloatApplet/IEEE754de.html>

3.5 Software – Die Seele eines Rechners

Der Rechner ist ein spezieller Prozessor, welcher Algorithmen ausführt. Das aber setzt voraus, dass der Prozessor den Algorithmus interpretieren kann, d.h. dieser muss so formuliert werden, dass er

- (a) die *Arbeitsanweisungen* versteht und
- (b) die *Basisoperationen* ausführen kann.



Ein Programm ist ein codierter Algorithmus.

3.5.1 Ein Programm für einen Algorithmus

Ein **Programm** für einen Algorithmus ist eine *Folge notwendiger Arbeitsanweisungen bei gegebenen Basisoperationen*. Es legt somit fest, welche Basisoperationen in welcher Reihenfolge zur Ausführung des Algorithmus abzuarbeiten sind.

Problem $s = \text{kgV}(m, n) = m * n / \text{ggT}(m, n)$

Bestimmen des kleinsten gemeinsamen Vielfachen s zweier natürlichen Zahlen m und n unter Verwendung des euklidischen Algorithmus zur Berechnung des größten gemeinsamen Teiler $r = \text{ggT}(m, n)$:

Programm $r = \text{ggT}(m, n)$

```

a = m;
b = n;
c = a % b;
while( c != 0)
{
    a = b;
    b = c;
    c = a % b;
}
r = b;
    
```

Programm $s = \text{kgV}(m, n)$, drei Varianten:

```

1. a = m;
   b = n;
   c = a % b;
   while( c != 0)
   {
       a = b;
       b = c;
       c = a % b;
   }
    
```

} **ggT(m, n)**

```
s = m * n;
s = s / b;
```

Dieses Programm kann durch unseren Modellrechner verarbeitet werden.

```
2. r = ggT ( m, n );
   s = m * n;
   s = s / r;
```

Ein *Aufruf eines anderen Programms* (als Unterprogramm oder Funktion) innerhalb eines Programms ist in den Basisoperationen unseres Modellrechners *nicht* vorgesehen!

```
3. s = m * n / ggT( m, n)
```

Komplexe Anweisungen können mittels der angegebenen Basisoperationen unseres Modellrechners *nicht* verarbeitet werden!

Modernen Programmiersprachen ermöglichen eine *komplexe* Syntax analog 2. und 3. Solche Programme nennt man **problemorientiert**. Einzelne Arbeitsanweisungen sind nicht sofort ausführbar, eine *Zerlegung* in die vorhandenen Basisoperationen ist notwendig. Diese wird in der Praxis automatisch durch spezielle Programme (**Compiler, Interpreter**) erledigt. Das 1. Programm, könnte Ergebnis einer solchen **Übersetzung** sein. Es besteht nur aus ausführbaren Arbeitsanweisungen unseres Modellrechners. Man nennt es ein **maschinenorientiertes** Programm.

Rekursion

Schließlich wird noch ein weiteres problemorientiertes Programm für den euklidischen Algorithmus betrachtet. Dieses wurde als **Rekursion** entwickelt, denn *es ruft sich selbst auf*.

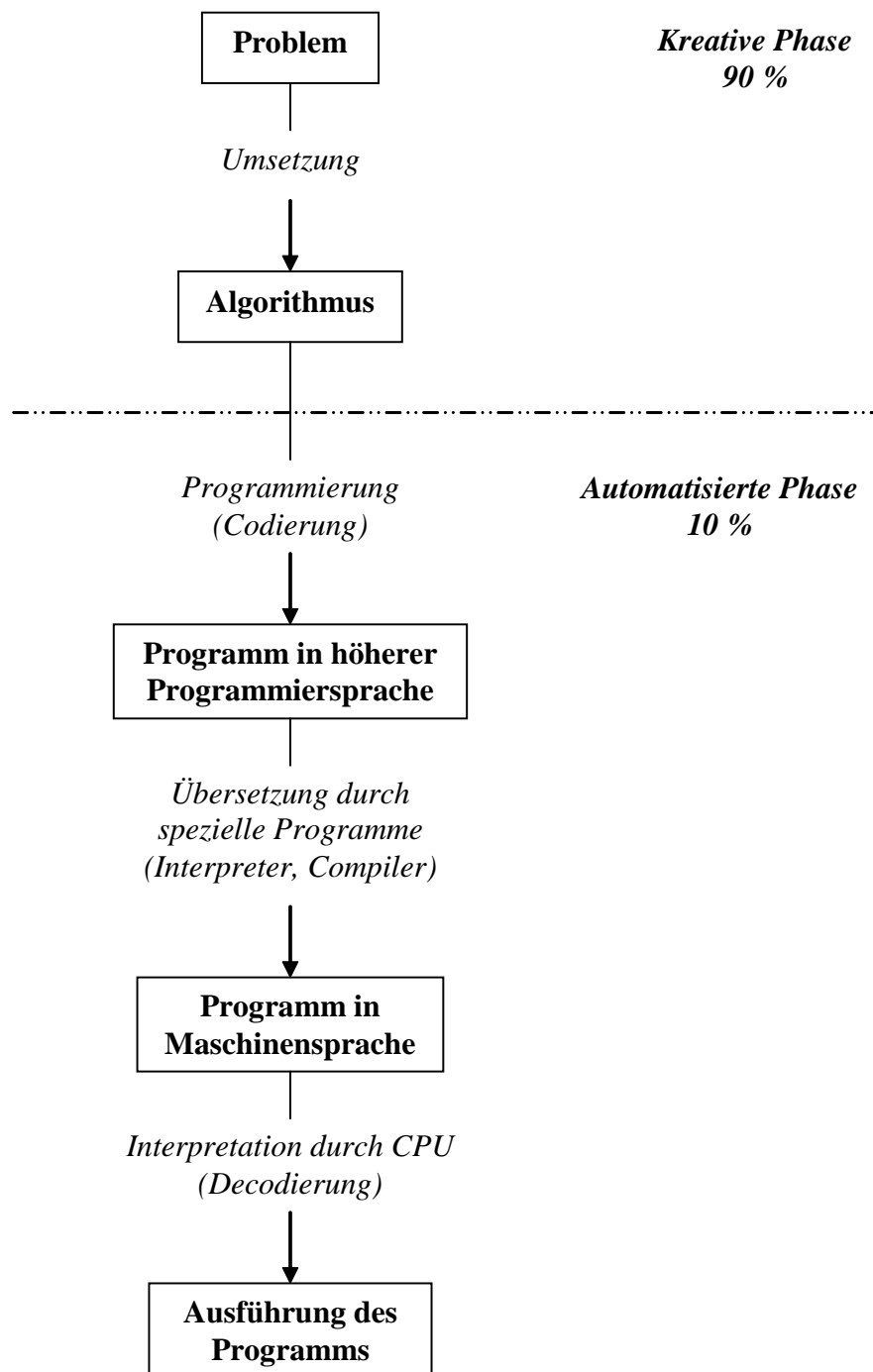
Programm $r = \text{ggT}(m, n)$ mit Rekursion

```
a = m;
b = n;
if( b != 0)
{
    r = ggT( b, a mod b);
}
else
{
    r = a;
}
```

Rekursion, problemorientiert

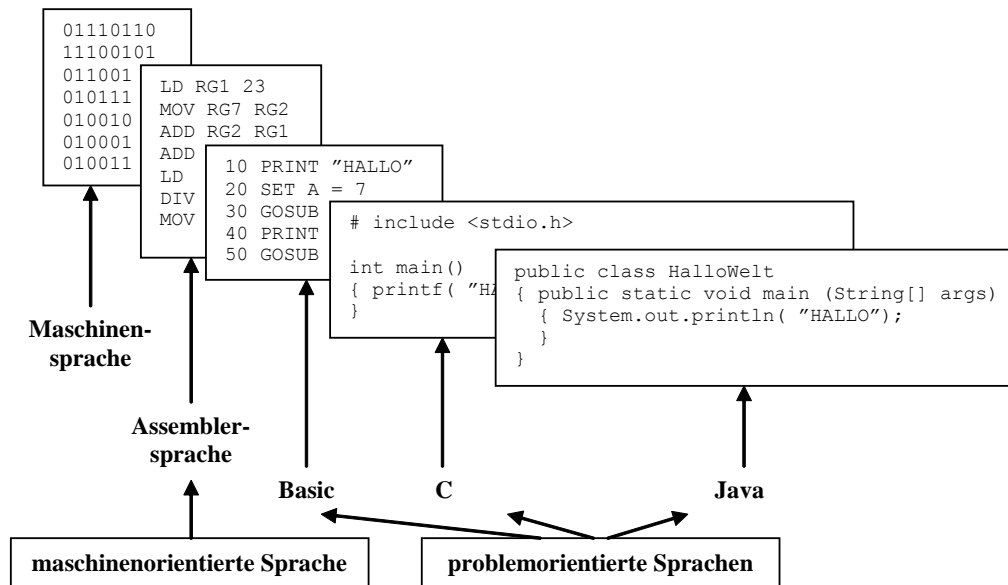
Zu einem Algorithmus kann man verschiedene Programme formulieren, zum einen in Abhängigkeit von den vorhandenen Basisoperationen, zum anderen aber auch in Abhängigkeit von den verwendeten Arbeitsanweisungen.

3.5.2 Programmhierarchien



**Ein Programm ist ein codierter Algorithmus.
Ein Algorithmus wird durch ein Programm interpretiert.**

Ein Rechner versteht einen Algorithmus nur als **Programm**, d.h. wenn dieser in eine spezielle Sprache, einer **Programmiersprache**, formuliert ist. Es gibt eine große Anzahl von Programmiersprachen auf unterschiedlichem Niveau.



Die Entwicklung vom Algorithmus zum ausführbaren Programm erfolgt in mehreren Schritten:

Maschinensprachen

Einfachste Sprachen, die der Rechner unmittelbar versteht. Jede Arbeitsanweisung ist eine Basisoperation und kann sofort ausgeführt werden. Der Rechner kann nur Signale verarbeiten. Programme in einer Maschinensprache (*Maschinencode*) sind somit *Bitfolgen*. Um die Programmierung im Maschinencode zu vereinfachen, wurde *Assemblercode*, ein symbolisierter Maschinencode, entwickelt. Ein spezielles Programm, der Assembler, übersetzt den Assemblercode in Maschinencode. In solchen *maschinenorientierten Sprachen* kommt man nur in kleinen Schritten vorwärts. Sie erzeugen sehr lange Programme. Das Entwickeln der Programme ist mühsam und birgt eine große Fehlerwahrscheinlichkeit in sich.

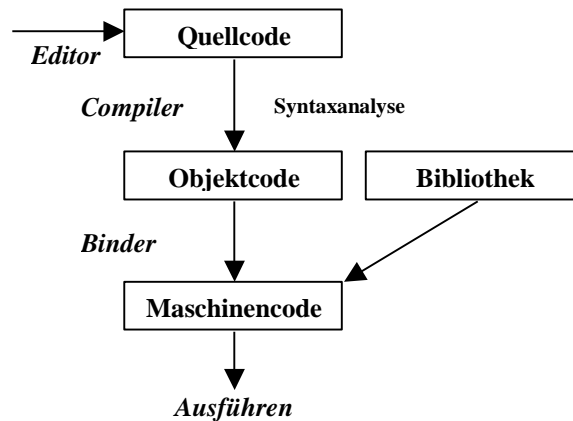
(11 011 101 \triangleq MOV 3 5 \triangleq Transportiere Inhalt des Register 3 nach Register 5)

Höhere Programmiersprachen

Problemorientierte Sprachen erzeugen Programme, welche nicht unmittelbar vom Prozessor verstanden werden. Man kann mittels einer Anweisung *mehrere Maschinenprogrammsschritte* zusammenfassen. Solche Programmiersprachen sind komfortabel. Damit wird das Programmieren erleichtert und die Fehlerwahrscheinlichkeit ist deutlich niedriger. Das zu einem Programm in höherer Programmiersprache gehörige Maschinenprogramm wird automatisch erzeugt, indem jede Arbeitsanweisung des problemorientierten Programms solange verfeinert wird, bis sie durch eine *Folge von Basisoperationen* ersetzt werden kann. Dazu braucht man ein spezielles **Übersetzungsprogramm**, einen *Interpreter* oder einen *Compiler*.

Compiler

Die Textdatei des Programms, der **Quellcode**, wird durch einen **Compiler** auf korrekte Syntax überprüft. Werden keine Fehler gefunden, so erzeugt der Compiler einen neuen Code, den **Objektcode**. Dieser Code wird anschließend vom **Binder** mit benötigten Bibliotheksprogrammen zum ausführbaren **Maschinencode** zusammengefügt. Die Übersetzung liegt beim Entwickler. Der Anwender braucht nur das Programm im Maschinencode für seinen Computer. In welcher Sprache das Programm ursprünglich entwickelt wurde, interessiert ihn nicht.

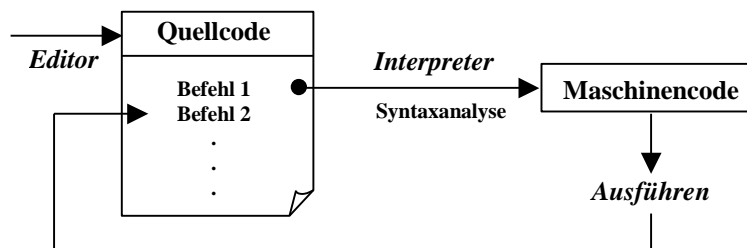


Übersetzte Programme sind im Allgemeinen sehr *schnell*, da die Übersetzung bereits vorliegt. Dafür sind sie aber nur auf der Systemplattform (Hardware und Betriebssystem) lauffähig, für die sie übersetzt wurden. Deshalb muss man beim Installieren von Software stets das vorhandene Betriebssystem angeben.

Für den Gebrauch im Rahmen des Internets ist das ein Nachteil, da hier eine Vielzahl von verschiedenen Systemplattformen untereinander vernetzt ist.

Interpreter

Der Quellcode des Programms bleibt bis zur Ausführung unbearbeitet. Die Übersetzung liegt beim Nutzer. Zum Starten braucht man außer diesem einen *Interpreter*. Der Interpreter liest zur Laufzeit einen Befehl des Quellcodes ein, wandelt ihn in **Maschinencode** um und führt ihn aus. Erst dann wird der nächste Befehl des Quellcodes vom Interpreter behandelt.



Da der Quellcode bei jedem Programmaufruf immer wieder neu übersetzt werden muss, sind zu interpretierende Programme *langsamer*. Syntaktische Fehler machen sich erst zur Laufzeit bemerkbar und führen zu Programmabbrüchen. Die Entwicklung selbst geht etwas schneller, da die Fehleranalyse solcher Programme leichter ist.

Interpreter	Compiler
<ul style="list-style-type: none"> • Ein Interpreter übersetzt jede Anweisung einzeln und führt sie aus. • Bei jeder Ausführung wird das Programm neu übersetzt. 	<ul style="list-style-type: none"> • Ein Compiler übersetzt das Programm als Ganzes und erzeugt einen Maschinencode. • Der Maschinencode wird ausgeführt, Quellcode wird für die Ausführung nicht benötigt.
<p>Beispiele Basic, Betriebssystemkommandosprachen</p>	<p>Pascal, C</p>

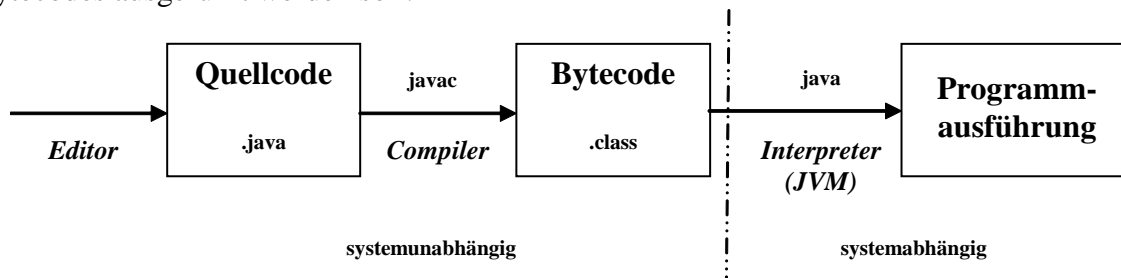
Vor- und Nachteile

- | | |
|--|---|
| <ul style="list-style-type: none"> • Strukturierung ist nicht so ausgeprägt. • Fehlersuche reduziert sich auf wenige Anweisungen. • Programmausführung dauert länger. | <ul style="list-style-type: none"> • Strukturierung und Modularisierung • Fehlersuche ist mühsamer, oft auf größere Bereiche ausgedehnt. • Laufzeit ist viel kürzer. |
|--|---|

Java-Programme brauchen sowohl einen Compiler als auch einen Interpreter:

Das Prinzip von **Java** basiert auf eine *Kombination beider Übersetzungstechniken*. Java-Programme werden nach ihrer Erstellung vom *Entwickler kompiliert*. Dabei entsteht ein Zwischencode, der sogenannte **Bytecode**. Dieser ist *nicht* ausführbar im obigen Sinn, dafür aber vollkommen unabhängig von der Hardware und dem Betriebssystem und kann über das *Internet* an andere Nutzer verschickt werden.

Ein *Interpreter (JVM .. Java Virtual Machine)* des *Nutzers* übernimmt die weitere Verarbeitung und muss deshalb für die konkrete Plattform vorhanden sein, auf der der Bytecodes ausgeführt werden soll.

**3.5.3 Softwareklassifizierung**

Wie schon festgestellt wurde, arbeitet jeder Rechner nach dem **EVA-Prinzip**. Das spiegelt sich auch in den Programmen wieder. Ein Programm benötigt Informationen, diese werden vom ihm verarbeitet und schließlich werden Ergebnisse der Verarbeitung wieder zur Verfügung gestellt.

Die Gesamtheit aller **Programme** einschließlich der von ihnen benötigten **Daten**, die auf einer Rechenanlage ihren Einsatz finden, wird als **Software** bezeichnet.

Systemsoftware

Programme, die für den korrekten Ablauf einer Rechenanlage erforderlich sind, sowie Programme, welche die Programmerstellung unterstützen und allgemeine Dienstleistungen bereitstellen.

- **Basissoftware:** Programme des **Betriebssystems** (dienen der Kommunikation zwischen Nutzer und Rechner; DOS, Windows, Unix, Mac OS)
- **systemnahe Software:**
 - Dienstprogramme** (Verwaltung von Dateien, Zugriff auf CD/DVD, Internetbrowser, Anti-Viren-Programme, ...)
 - Interpreter bzw. Compiler** - Übersetzungsprogramme für Programmiersprachen (Basic; Pascal, C, Java, ...)
 - Programmiersysteme** (Text-, Tabellenkalkulations-, Datenbanksysteme wie in MS Office/ Open Office, ...)

Anwendersoftware

Sie dient der Lösung von Nutzerproblemen.

- spezielle **Projekte**, zum Beispiel für Verwaltungsaufgaben (Fahrkarten, Post, Flug, Reisen, Renten, Versicherungen, Banken, ...)
- **Nutzerprogramme:** ggT, Lösen eines Integrals, Spiele, ...

Fehleranalyse

Bei der Entwicklung von Software können Fehler auftreten. Manche Fehler entdeckt man erst während der Anwendung. Deshalb ist eine **Wartung** der Software notwendig. Verbesserte Software wird dann als **neue Version** dieser vertrieben.

- **Syntaxfehler**: Falscher Einsatz von Sprachelementen, wird sofort erkannt.
- **Semantikfehler**: Falscher logischer Aufbau des Programms (oft zurückzuführen auf fehlerhafte Problemanalyse), wird oft erst in der Testphase oder noch später entdeckt.

3.6 Einführung der Begriffe in der Unterstufe

Hardware: Alles was man anfassen kann.

Software: Programme, die der Rechner braucht, um zu arbeiten.

Programm: Endliche Folge ausführbarer Befehle
(Beispiel: Muttis Programm für den Nachmittag)

Hinweis: *Sendung mit der Maus – Computer 1989, Internet*