4. Daten

Was ist Informatik?

vgl. Kapitel 1

Begriffsbestimmung (Gegenstand):

"Informatik ist die Wissenschaft ... der maschinellen Verarbeitung, Speicherung und Übertragung von Information."

(Broy, Informatik, Teil I, Springer 1992).

- → Information verarbeiten (Algorithmen)
- → Information repräsentieren (Daten ... Datenbanken)
- → Informationen übertragen (Internet ... Kontoauszugsdrucker)

Programme = Daten + Algorithmen

Programme lösen Probleme

... durch Algorithmen, die über Daten operieren

Programme = Daten + Algorithmen

Problembereiche zu Daten

Programme = Daten + Algorithmen

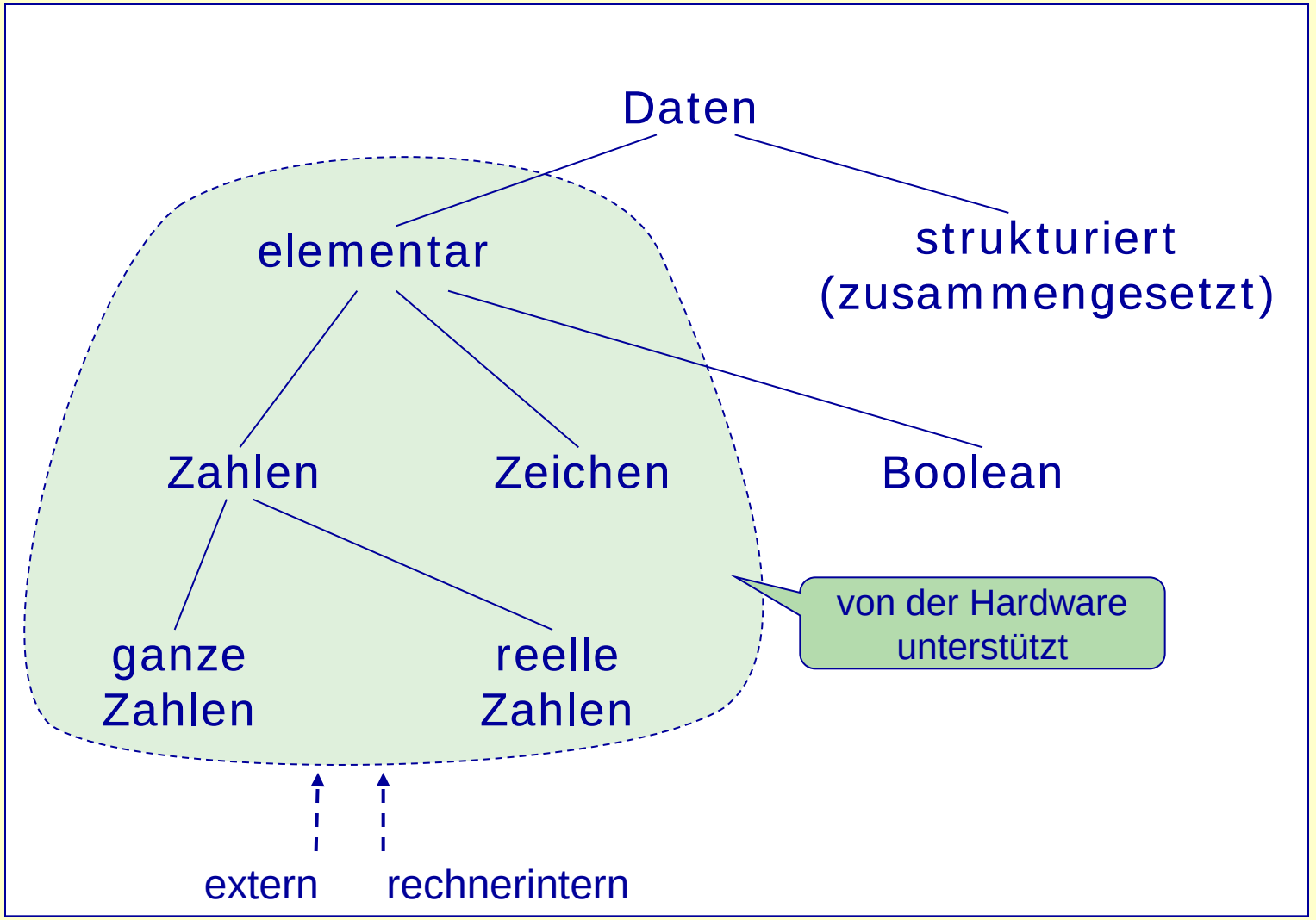
Problembereiche:

- **Repräsentationen** (Darstellung):
  extern (menschengerechte, lesbare Form, im Programm)
  ⇔ rechnerintern (Bitfolgen: 00110 ...)
- **Strukturierungsart:**
  elementare Daten (Zahlen, Zeichen)
  ⇔ zusammengesetzte (strukturierte) Daten

jetzt: elementare Daten in externer und rechnerinterner Repräsentation

später (Teil II): elementare und strukturierte Daten in Java

Klassifikation von Daten



Rechnerinterne Darstellung von Daten bei Nutzung höherer Programmiersprachen wichtig?

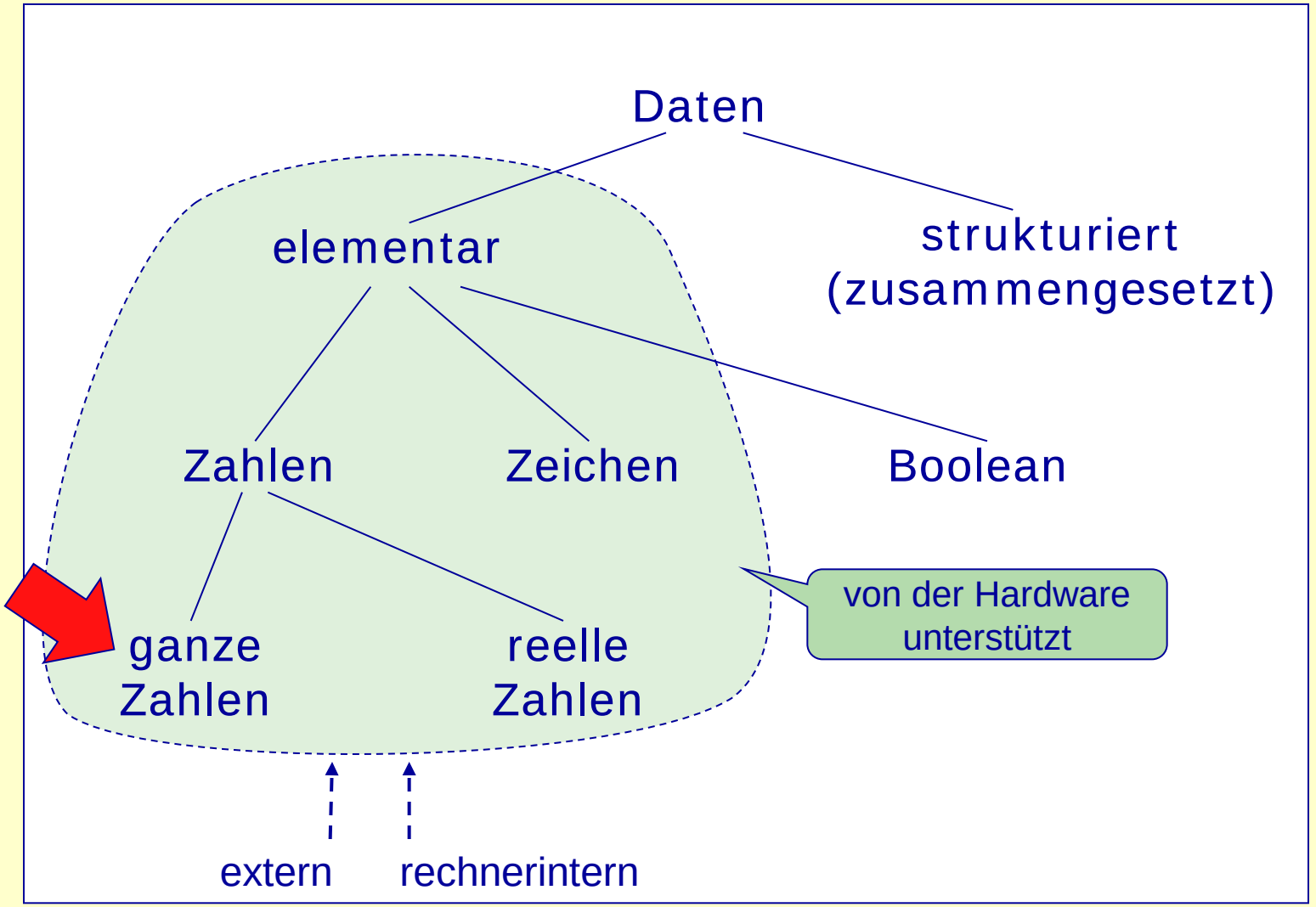
Adresse	Inhalt
60225	…
60226	00110011 01000110 01110000 10000001
60227	10110011 11100110 01110000 00011001
60228	00110011 00000110 01110000 11000001
60229	00000011 00000110 01110000 10000001
60230	10110011 00000110 01110000 01100001
60231	01110011 00000110 01110000 00000101
60232	01100111 00000001 01101011 01000101
60233	00111011 00000110 01110000 11100001
…	…
…	…
280461	00000000 00000000 00000000 00000001
…	…
…	…
440204	00000000 00000000 00000000 00000101
…	…

Rechnerinterne Form wichtig auch bei Nutzung höherer Programmiersprachen:

- Verstehen von Speicherinhalten bei Laufzeitfehlern (Speicherdump)
- Größenbeschränkungen von Zahlen in Programmen
- Rundungsprobleme bei reellen Zahlen

Ganze Zahlen

Klassifikation von Daten



Zahlendarstellungen:
Ganze Zahlen - externe Form

- C, C++, JAVA u. a.: unterschiedliche Datentypen

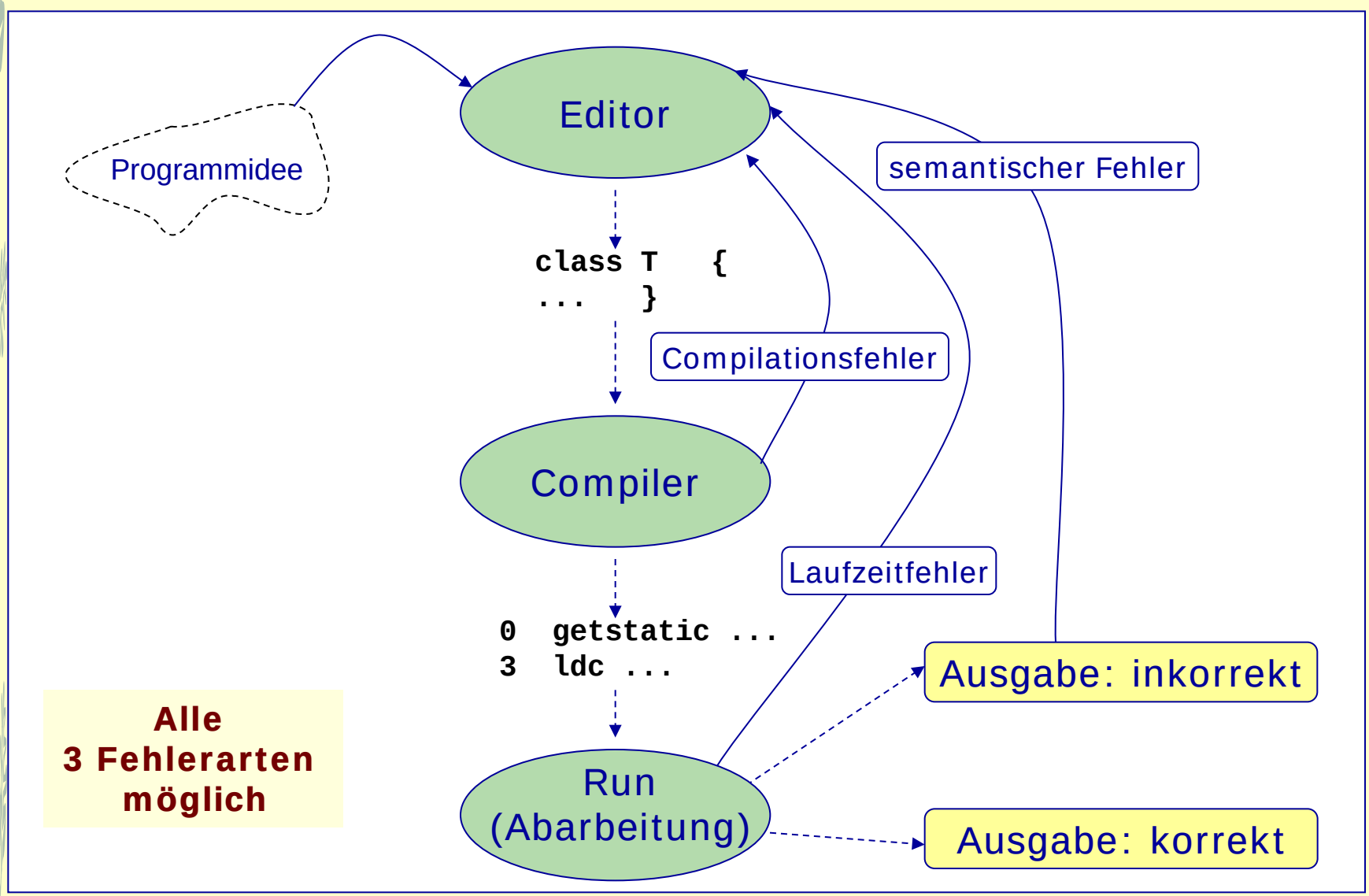
short:	im Bereich $-2^{15} \ldots 2^{15}-1$	(2 Byte = 16 Bit)
int:	im Bereich $-2^{31} \ldots 2^{31}-1$	(4 Byte = 32 Bit)
long:	im Bereich $-2^{63} \ldots 2^{63}-1$	(8 Byte = 64 Bit)

z. B.: 100, -1, 0

größte int: 2147483647 ($2^{31}-1$)
kleinste int: -2147483648 (-2^{31})

Inhaltliche Begründung?

Fehlerart bei Überschreitung der Zahlenbereiche?



Fehler: zu große bzw. zu kleine Zahlen

Bereich int:

a) im Programmtext: `x = 2147483647;(10*maxint)`

b) als Resultat einer Operation: `y = 2147483647 - b`

(zur Abarbeitungszeit: b<0)

Welche Fehlerart:

a) → Compilationsfehler (lexikalischer Fehler)

b) → Laufzeitfehler (integer overflow) oder semantischer Fehler

Achtung:

- Laufzeitfehler meist nicht angezeigt, sondern semantischer Fehler (abhängig von Rechnerarchitektur und z.T. von Programmiersprache)
- Damit: Resultat ist falsche Zahl, mit der weitergearbeitet wird → falsche Ergebnisse erscheinen als semantische Fehler

oft: größte int-Zahl + 1 → kleinste int-Zahl

2147483647 + 1 → -2147483648

d.h. $(2^{31}-1) + 1 \rightarrow (-2^{31})$

Integer Overflow

Europäische Trägerrakete Ariane 5: Absturz 1996

Positive ganze Zahlen: rechnerinterne Repräsentation

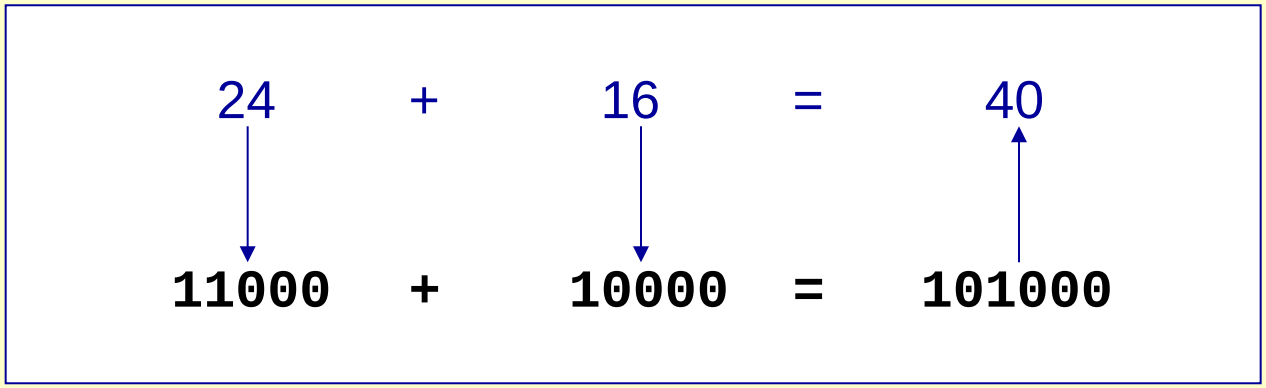
Binäre Darstellung (Dualzahl) mit zwei Ziffern: 0 und 1

0 → 0, 1→ 1, 2 → 10, 3 → 11, 4 → 100, 5 → 101 …

24 = 11000_2, intern als Typ ***int*** mit 4 Byte:

```
00000000 00000000 00000000 00011000
```

Maschineninterne Operationen mit Dualzahlen:



Wert einer Dualzahl (Binärzahl)

Dualzahl:

```
00000000 00010100
```

$b_{n+1}b_nb_{n-1}...b_2b_1$

Wert:

$$x = \sum_{i=1,...n} b_i \cdot 2^{i-1}$$

(short: n=15, int: n=31, long: n=63)

z.B. 10100_2 → $0*2^0 + 0*2^1 + 1*2^2 + 0*2^3 + 1*2^4 = 20$

→ 1. Bit "frei" für Vorzeichen (positiv: 0, neg.: 1)
→ in 15 Bit: größte positive short-Zahl: $2^{15} - 1$

```
01111111 11111111
```

Negative ganze Zahlen: rechnerinterne Repräsentation

Dualzahl (Binärzahl):

$$b_{n+1}b_n b_{n-1}\ldots b_2 b_1$$

erstes Bit entscheidet: positiv oder negativ?

- → $b_{n+1} = 0$ für positive Zahlen
- → $b_{n+1} = 1$ für negative Zahlen

Verschiedene Varianten (abh. von Rechnerarchitektur)

Variante 1: einfache Vorzeichenbit-Technik

Dualzahl (Binärzahl): $\mathbf{b_{n+1}b_nb_{n-1}...b_2b_1}$

b_{n+1} = 0 für positive Zahlen
b_{n+1} = 1 für negative Zahlen
sonst wie bisher

24 = 11000_2, Typ short mit 2 Byte:

```
00000000 00011000
```

-24 = -11000_2, Typ short mit 2 Byte:

```
10000000 00011000
```

Variante 2: 1-er-Komplement

-x: bitweise Umkehrung der Dual-Ziffern

$24 = 11000_2$, Typ short mit 2 Byte:

```
00000000 00011000
```

$-24 = -11000_2$, Typ short mit 2 Byte:

```
11111111 11100111
```

Variante 3: 2-er-Komplement

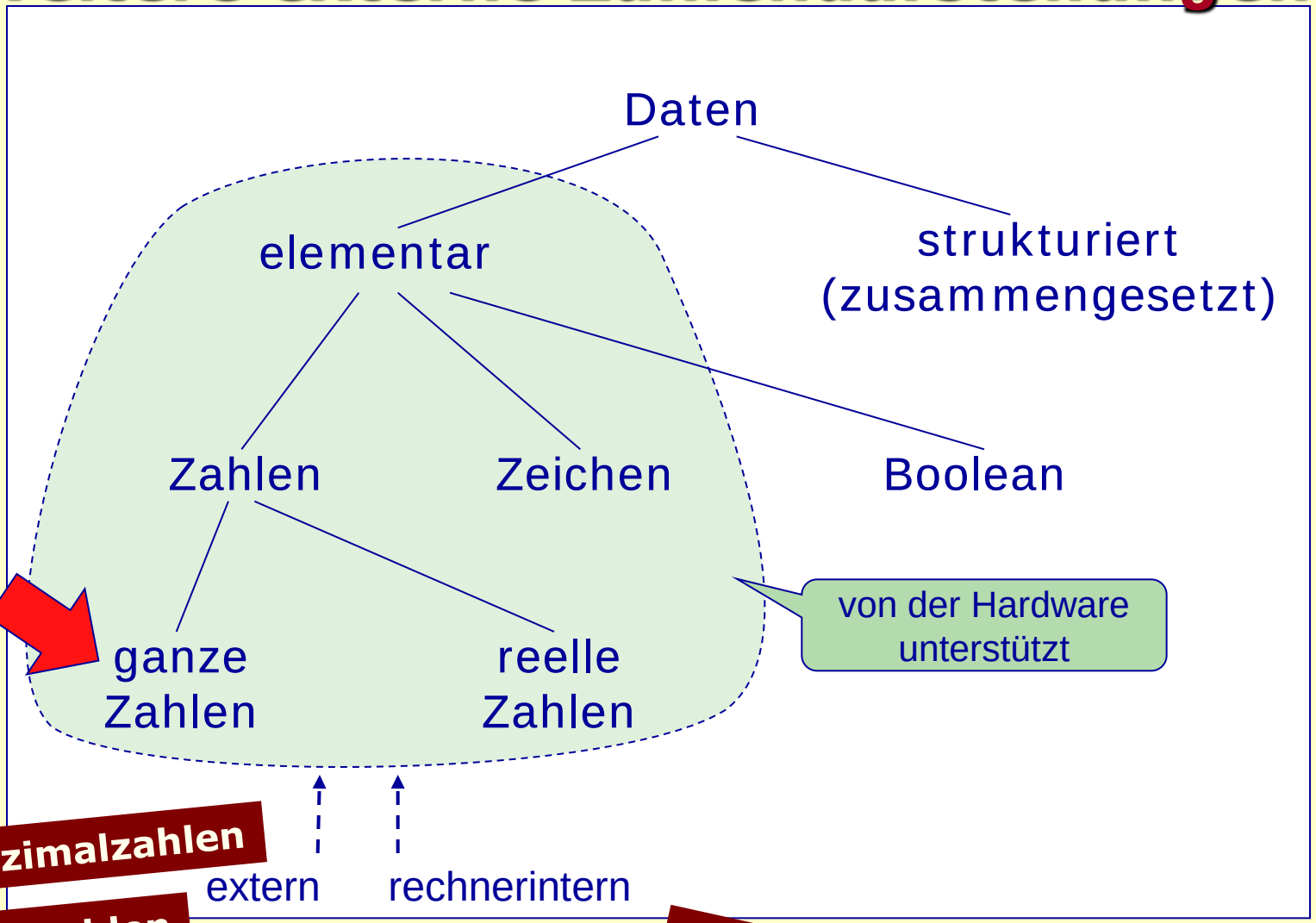
-x: Einerkomplement + 1

-24 = -11000_2, Typ short mit 2 Byte:

+ 24	**00000000 00011000**
Einerkomplement:	**11111111 11100111**
Zweierkomplement:	**11111111 11101000**

Heutige Rechner: meist 2-er-Komplement
(Addition positiver und negativer Zahlen einfach)

Ganze Zahlen: weitere externe Zahlendarstellungen



Hexadezimalzahlen

Dezimalzahlen

Oktalzahlen

Binärzahlen

Ganze Zahlen: weitere externe Zahlendarstellungen

(bisher Zahlen mit Basis: 10, 2)

a) *Oktal*-Zahl mit Ziffern 0, ..., 7
(passen in 3 Bit)

Wert der Oktalzahl $s_n s_{n-1} \ldots s_2 s_1$

$$x = \sum_{i=1,\ldots n} s_i \cdot 8^{i-1}$$

z. B. $14_8 = 12_{10}$

in C, Java u. a.: 014 (führende Null)

b) Hexadezimal-Zahl
mit Ziffern 0, ..., 9, A, ..., F
(passen in 4 Bit)

Wert der Hexadezimalzahl $s_n s_{n-1} \ldots s_2 s_1$

$$x = \sum_{i=1,\ldots n} s_i \cdot 16^{i-1}$$

z. B. $14_{16} = 20_{10}$, $1A_{16} = 26_{10}$

in C, Java u. a.: 0x14, 0x14,
0x1A, 0xCAFE

Zusammenfassendes Beispiel

$$3E8_{16} = 1000_{10} = 1750_{8} = 1111101000_{2}$$

→ Umrechnungsalgorithmen: Übungen

Zum Nachdenken

- Warum werden überhaupt unterschiedliche Zahlensysteme verwendet?

- Es kann eine negative Zahl mehr (als bei positiven Zahlen) im 2er-Komplement dargestellt werden.
  - Wieso?
  - Beispiel: Wie sieht die interne Darstellung der kleinsten (short-, d.h. 16 Bit-)Zahl aus?
    → größte Zahl:

    01111111 11111111

    → kleinste Zahl: ?

Warum verschiedene Zahlensysteme? (Basis 2, 8, 10, 16 u. a.)

- Dezimal: lesbar
- Binär: maschinenintern

  `0110 1111 0100 1100`
- Hexadezimal: komprimierte maschineninterne Form:

  Speicher*dump:* **6F4C**

  (1 Byte durch 2 Ziffern dargestellt)
- Oktal: passen in 3 Bit

Interne Darstellung: kleinste ganze Zahl?

Interne Darstellung: größte Zahl Zmax

01111111 11111111

short: $2^{15}-1$

Interne Darstellung: kleinste Zahl = -(Zmax + 1) ?

Zmax + 1:

(0) 10000000 00000000

falls nicht nur 16 Bit

1er Komplement:

(1) 01111111 11111111

2er Komplement: + 1

10000000 00000000

short: -2^{15}

Überschreitung von Zahlenbereichen: semantischer Fehler entsteht

value = ... größte int-Zahl

01111111 11111111

short: $2^{15}-1$

value = value + 1; ... kleinste int-Zahl

10000000 00000000

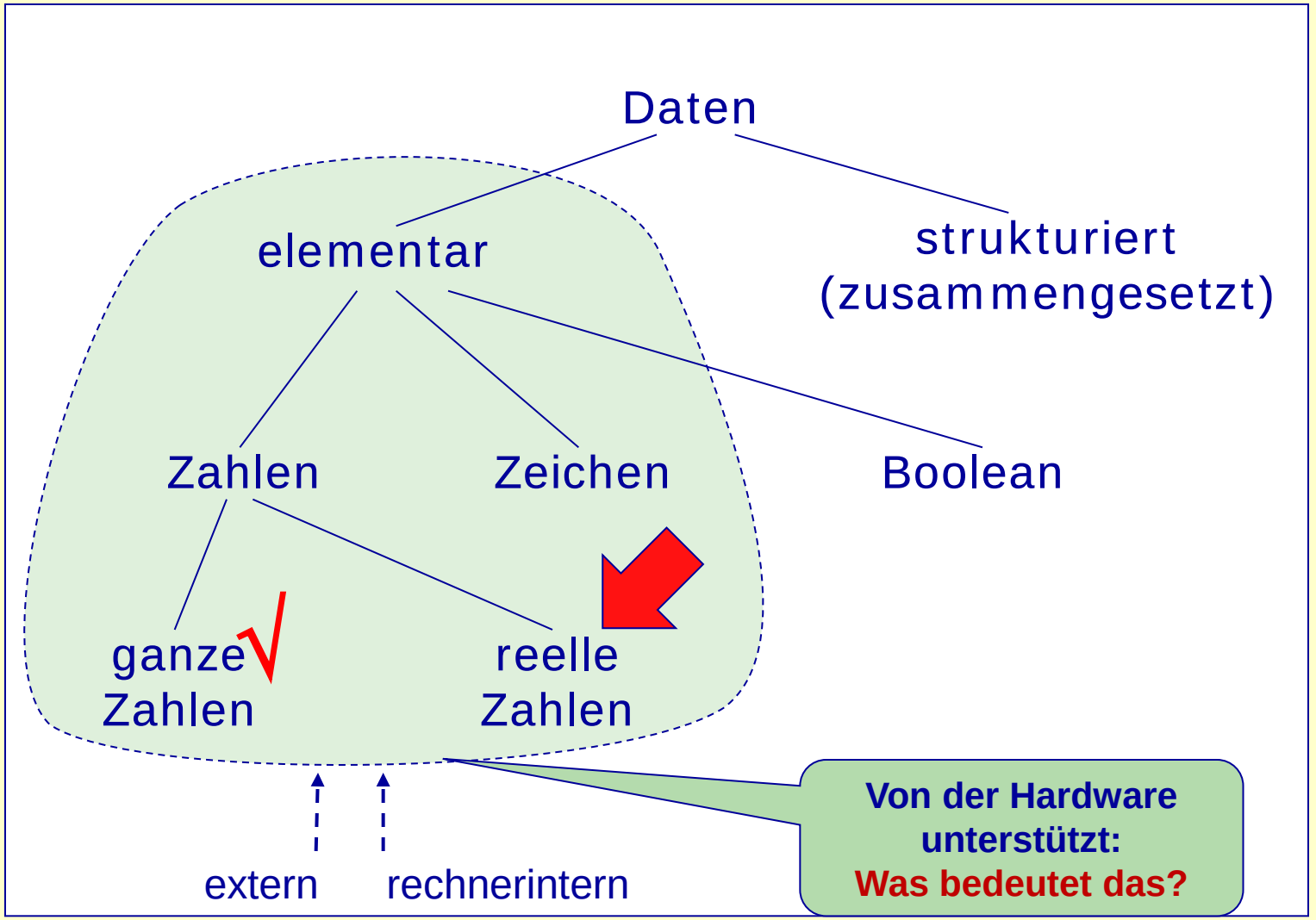
short: -2^{15}

Reelle Zahlen

Gleitkommazahlen

Floating-point numbers

Klassifikation von Daten



Reelle Zahlen: externe Form

C, C++, JAVA u. a.: 2 Typen (mit 2 Bereichen)

float (4 Byte) im Bereich

$-3.40282347 * 10^{38}$ ... $+3.40282347 * 10^{38}$

double (8 Byte) im Bereich

$-1.7976931348623157 * 10^{308}$ ... $+1.7976931348623157 * 10^{308}$

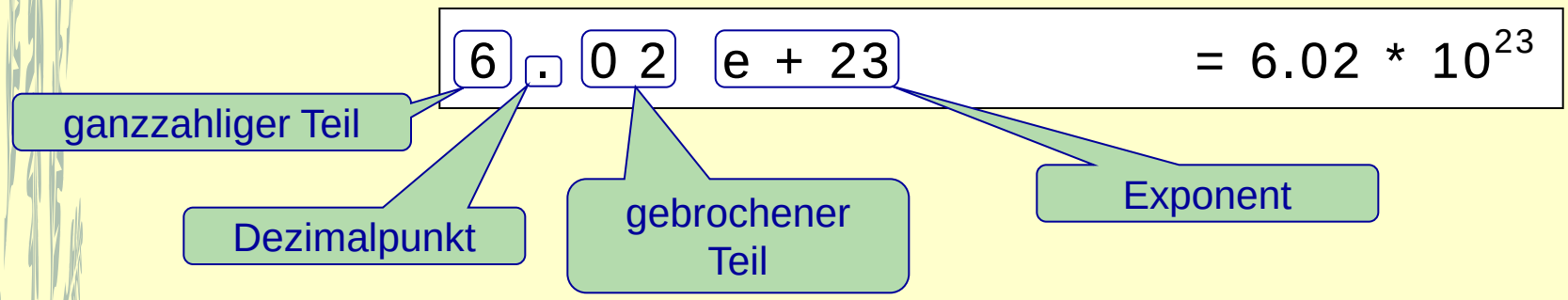
- Begründung für 'krumme' Bereiche:
  interne Repräsentation
- Nicht jede reelle Zahl ist repräsentierbar – warum?

  In jedem endlichen Intervall [a , b] liegen unendlich viele reelle Zahlen, aber nur endlich viele davon können in n Bytes (n = 4 oder 8) repräsentiert werden!

➔ Rundungsfehler normal, z.B. Zahl Pi

Reelle Zahlen: externe Form (cont.)

(C, C++, Java, Pascal)



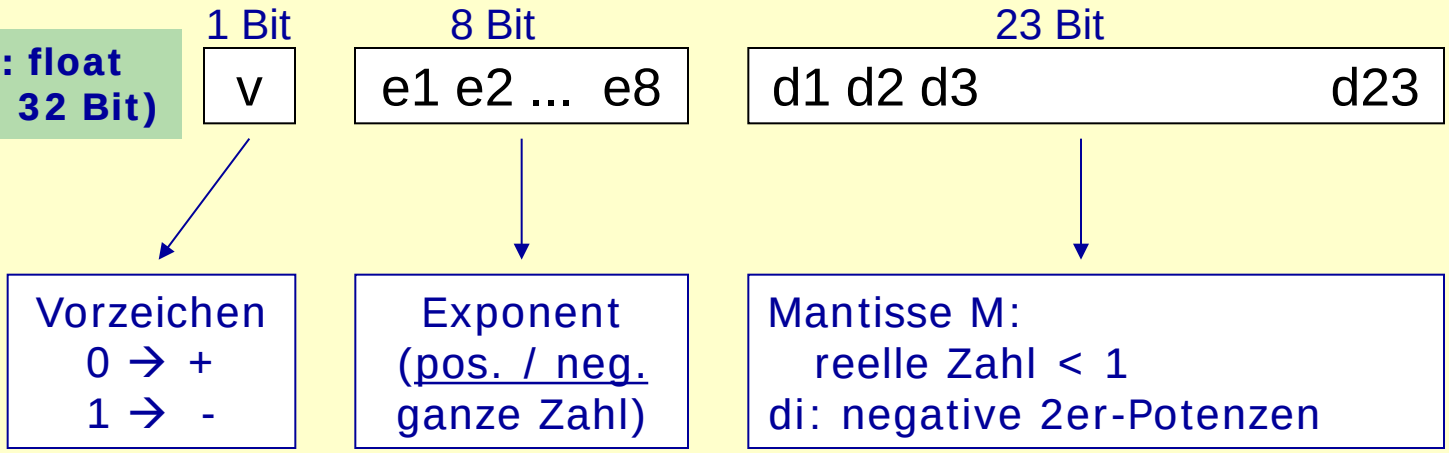
Teile optional:

1 e -2	(kein gebrochener Teil)
3.14	(kein Exponent)
.3	(kein ganzzahliger Teil)
2.	(kein gebrochener Teil)
6.02 e 2	(kein Vorzeichen im Exponenten)

2 keine reelle Zahl, sondern anderer Zahlenbereich:
ganze Zahl

Reelle Zahlen: rechnerinterne Form

Bereich: float (4 Byte = 32 Bit)



Für float/4 Byte: v, e1, e2, ... e8, d1, d2 ... d23:

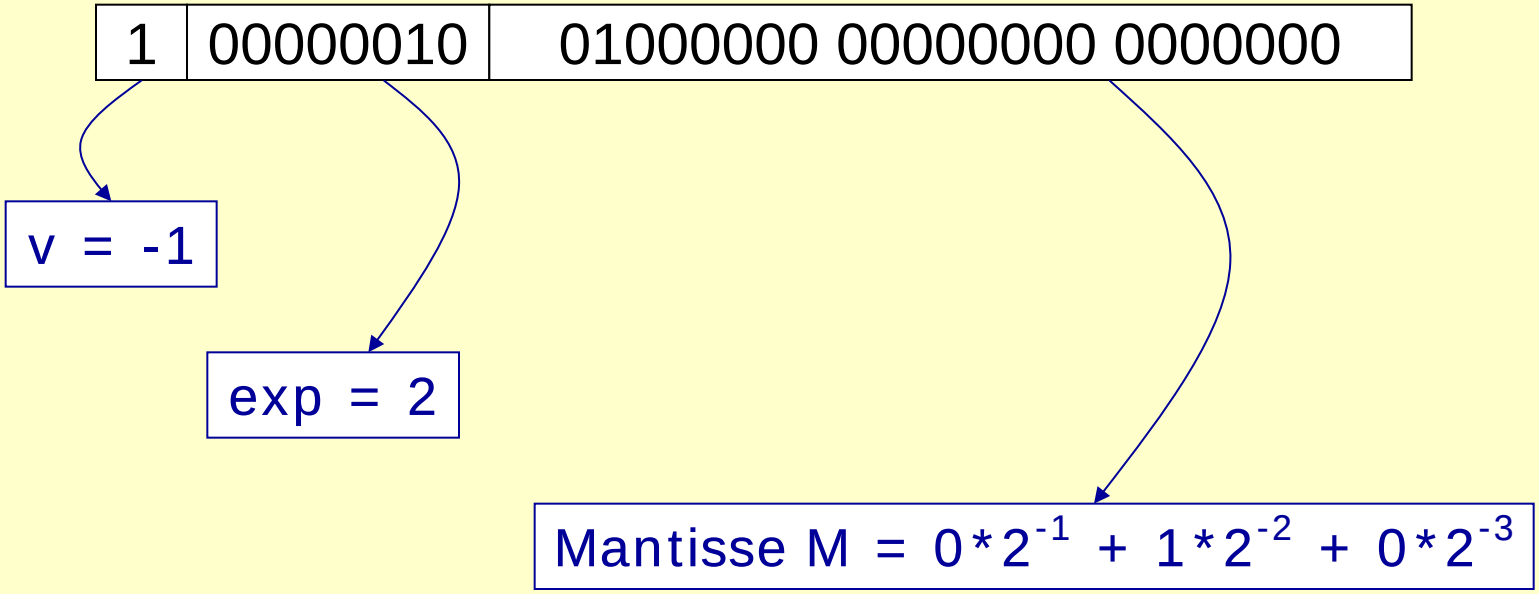
Exponent *exp* zwischen 2^7-1 und -2^7 (127 und – 128)

$$M = d1 * 2^{-1} + d2 * 2^{-2} + \ldots + d23 * 2^{-23}$$

Dargestellte reelle Zahl $r = [+/-]\ 2^{exp}\ (1+M)$

Andere Varianten beim Exponenten möglich: exp immer positiv interpretiert: $2^{exp-127}$

Beispiel



$r = (-1) * 2^2 * (1 + 1 / 4) = - 4 * 5 / 4 = \mathbf{-5}$

Andere Variante bei Interpretation des Exponenten:
$\mathbf{2^{2-127} = 2^{-125}}$

$r = (-1) * 2^{-125} * (1 + 1 / 4) = - 2^{-125} * 5 / 4 = - 2^{-127} * 5$

Dargestellte reelle Zahl $r = [+/-]\ 2^{exp}\ (1+M)$

Repräsentation der reellen Zahl 0.0

Spezielle Festlegungen: bisher 0.0 nicht repräsentierbar

$$r = [+/-]\ 2^{exp}\ (1+M)$$

Hardware: keine einheitliche Regelung

ein Beispiel: **IEEE 754 floating-point standard**

(Hennessy, Patterson: Computer Architecture)

Mantisse M: 0

Exponent exp: kleinste mögliche negative Zahl im Exponentenfeld

0	10000000	00000000 00000000 00000000 00000000

Ganze Zahl ←→ reelle Zahl

Notation im Programm: -5.0 oder -5

11111111 11111111 11111111 11111011

anders repräsentiert (anderer Zahlenbereich)

1	00000010	01000000 00000000 0000000

v = -1

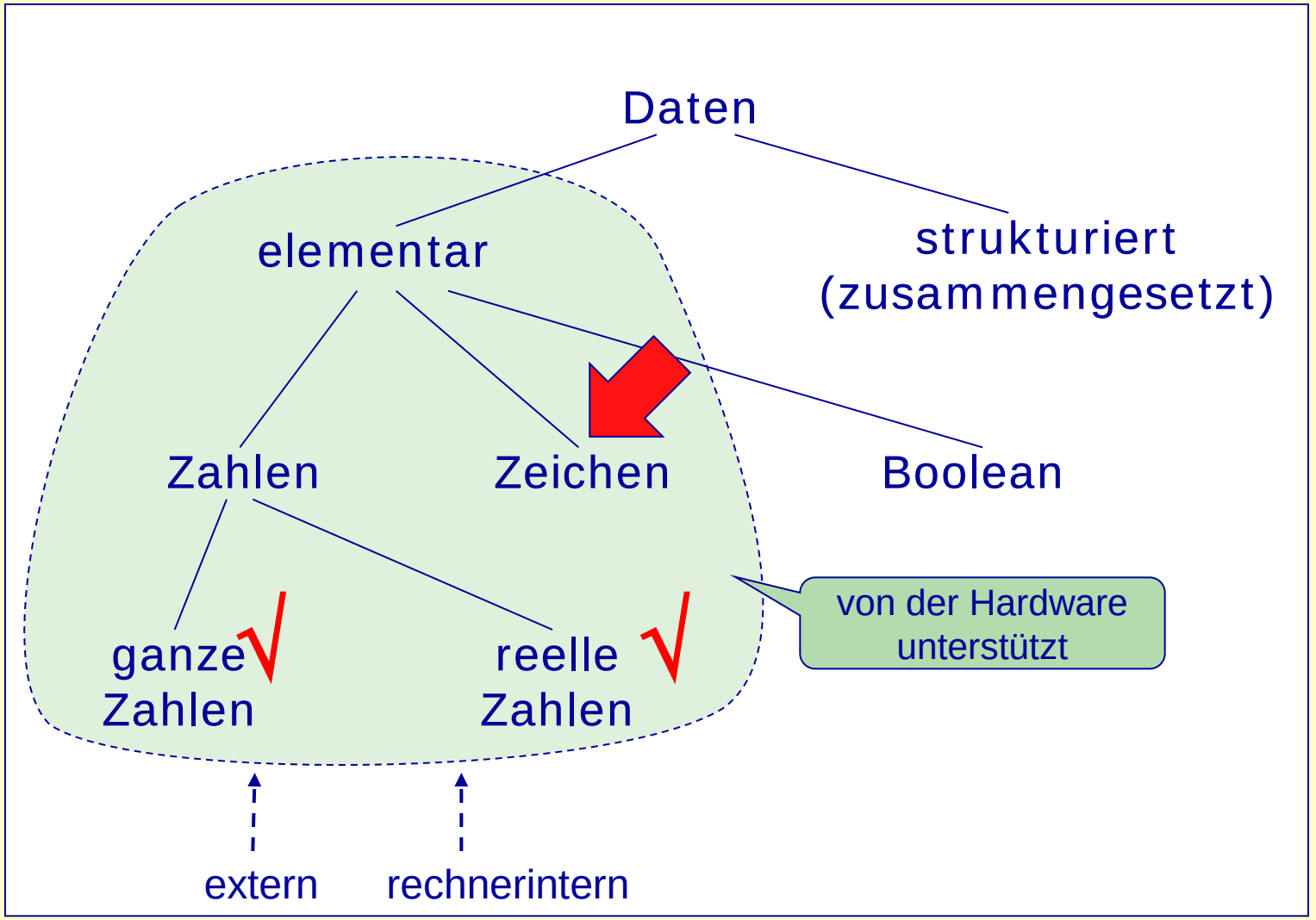
exp = 2

Mantisse M = $0*2^{-1} + 1*2^{-2} + 0*2^{-3}$

r = $(-1) * 2^{2} * (1 + 1 / 4) = - 4 * 5 / 4 =$ **-5**

Zeichen

Klassifikation von Daten



Zeichen (char)

Schreibmaschinenzeichen und weitere Zeichen,
u. a. Steuerzeichen (CTRL ..)

z.B.

CR = \carriage return" (Wagenrücklauf)

LF = \line feed" (neue Zeile)

Wichtige Zeichensätze:

1. ASCII Zeichensatz

(American Standard Code for Information Interchange)

7-bit-Code (1 Byte mit führendem Bit = 0)

→2^7 = 128 Zeichen

→ausreichend für "normale" Programmierung

(mit lateinischen Zeichen, Ziffern, Operatoren, Spezielles)

Zeichen (char)

2. LATEIN1 (ISO-8859-1) Zeichensatz (umfasst ASCII)

8-bit-Code (1 Byte)

→2^8 = 256 Zeichen

3. Unicode Zeichensatz (umfasst ISO-8859-1)

16-Bit-Code (2 Byte - Basisversion)

→ 2^{16} = 65536 Zeichen

Schriftzeichen der wichtigsten Sprachen codiert:
Griechisch, Chinesisch, Kyrillisch, Arabisch ...

Unicode Version 6.0 (Okt. 2010): 109.242 Zeichen (32 Bit)

JAVA verwendet Unicode

ASCII-Code

Anwendung:
Hexadezimale Zahlendarstellung
(je Byte: 2 Hex-Ziffern)

Zeichen „/“:
47 –> 2F = 0010 1111

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

Beispiel: ASCII-Code

binär:

0010 1100	0100 0011	0101 1100	0110 0001

hexadezimal:

2C	43	5C	61

als Zeichenfolge (dekodiert nach ASCII-Tabelle):

, C \ a

Unicode: dargestellte Schriftsätze

The Unicode Character Code Charts By Script

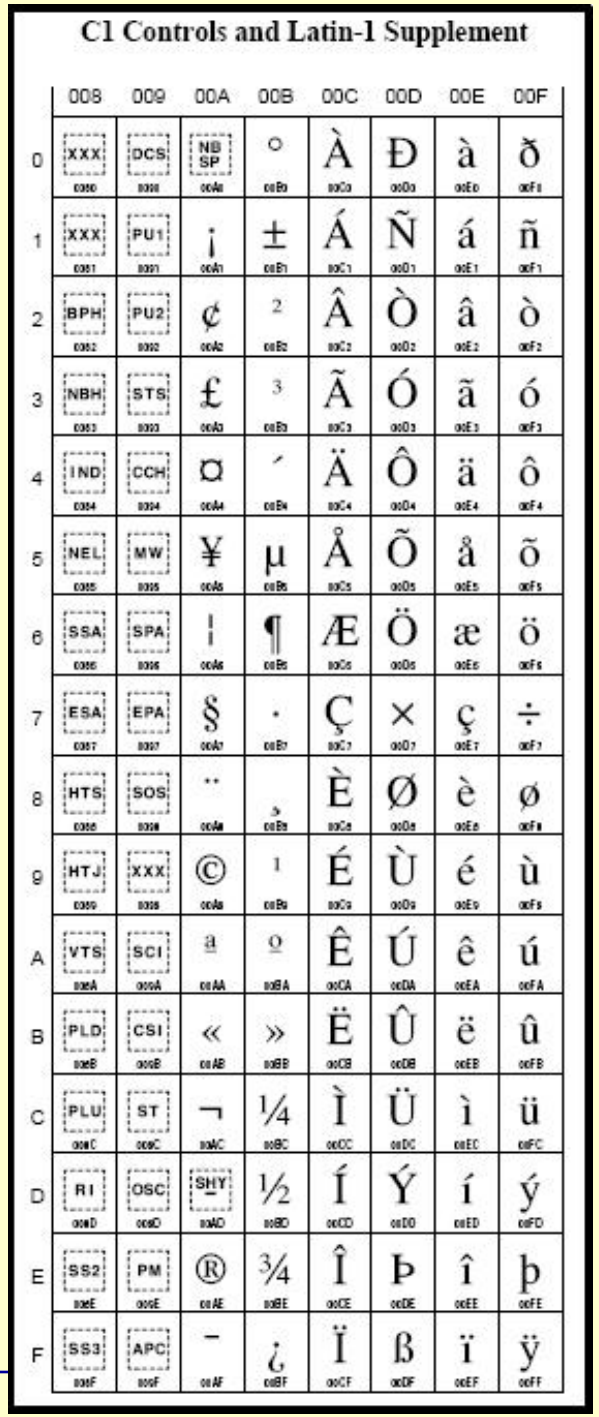
SYMBOLS AND PUNCTUATION | NAME INDEX | HELP AND LINKS

European Alphabets	African Scripts	Indic Scripts	East Asian Scripts	Central Asian Scripts
(see also **Comb. Marks**)	***Ethiopic***	Bengali	***Han Ideographs***	Kharoshthi
Armenian	Ethiopic	Devanagari	Unified CJK Ideographs (5MB)	Mongolian
Armenian	Ethiopic Supplement	Gujarati	CJK Ideographs Ext. A (2MB)	Phags-Pa (5.0)
Armenian Ligatures	Ethiopic Extended	Gurmukhi	CJK Ideographs Ext. B (13MB)	Tibetan
Coptic	***Other African scripts***	Kannada	Compatibility Ideographs (.5MB)	
Coptic	N'Ko (5.0)	Limbu	Compatibility Ideo. Suppl. (.5MB)	
Coptic in Greek block	Tifinagh	Malayalam	Kanbun	
Cyrillic	**Middle Eastern Scripts**	Oriya	(see also **Unihan Database**)	**Ancient Scripts**
Cyrillic	***Arabic***	Sinhala	***Radicals and Strokes***	***Ancient Greek***
Cyrillic Supplement	Arabic	Syloti Nagri	CJK Radicals	Ancient Greek Numbers
Georgian	Arabic Supplement	Tamil	KangXi Radicals	Ancient Greek Musical
Georgian	Arabic Presentation Forms A	Telugu	CJK Strokes	***Cuneiform***
Georgian Supplement	Arabic Presentation Forms B		Ideographic Description	Cuneiform (5.0)
Greek	***Hebrew***	**Philippine Scripts**	***Chinese-specific***	Cuneiform Numbers (5.0)
Greek	Hebrew	Buhid	Bopomofo, Extended	Old Persian
Greek Extended	*Hebrew Presentation Forms*	Hanunoo	***Japanese-specific***	Ugaritic
(see also ***Ancient Greek***)	***Other ME Scripts***	Tagalog	Hiragana	***Linear B***
Latin	Syriac	Tagbanwa	Katakana,	Linear B Syllabary
Basic Latin	Thaana		Katakana Phonetic Ext.	Linear B Ideograms
Latin-1	**American scripts**	**South East Asian**	*Halfwidth Katakana*	***Other Ancient Scripts***
Latin Extended A	Canadian Syllabics	Buginese	***Korean-specific***	Aegean Numbers
Latin Extended B	Cherokee	Balinese (5.0)	Hangul Syllables (4MB)	Counting Rod Num. (5.0)
Latin Extended C (5.0)	Deseret	Khmer	Hangul Jamo	Cypriot Syllabary
Latin Extended D (5.0)		Khmer Symbols	Hangul Compatibility Jamo	Gothic
Latin Extended Additional	**Other Scripts**	Lao	*Halfwidth Jamo*	Old Italic
Latin Ligatures	Shavian	Myanmar	***Yi***	Ogham
Fullwidth Latin Letters	Osmanya	New Tai Lue	Yi (.6MB)	Runic
Small Forms	Glagolitic	Tai Le	Yi Radicals	Phoenician (5.0)
(see also **Phonetic Symbols**)		Thai		

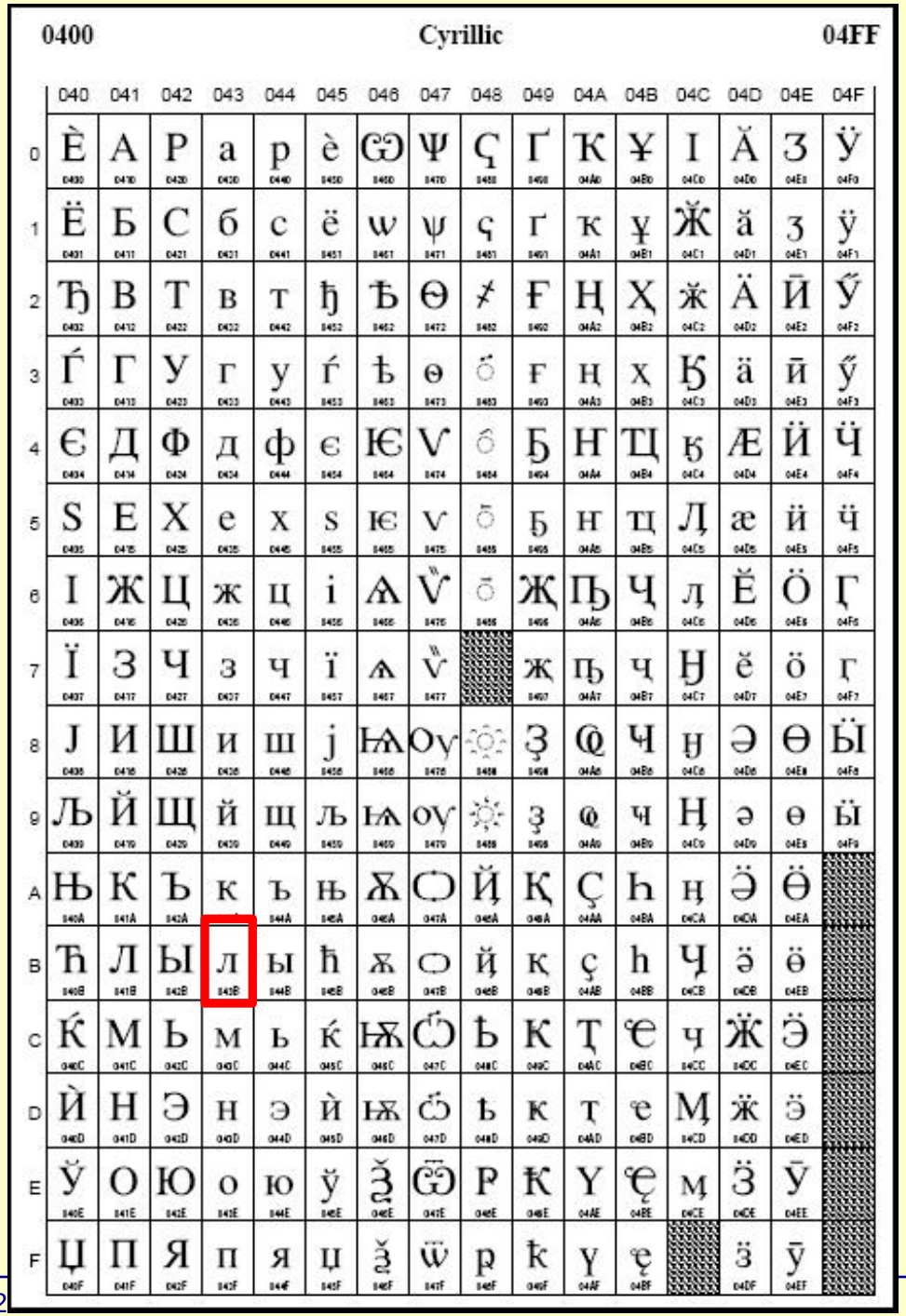
Unicode: Basic Latin (ASCII) und Latin-1

Zu lesen:

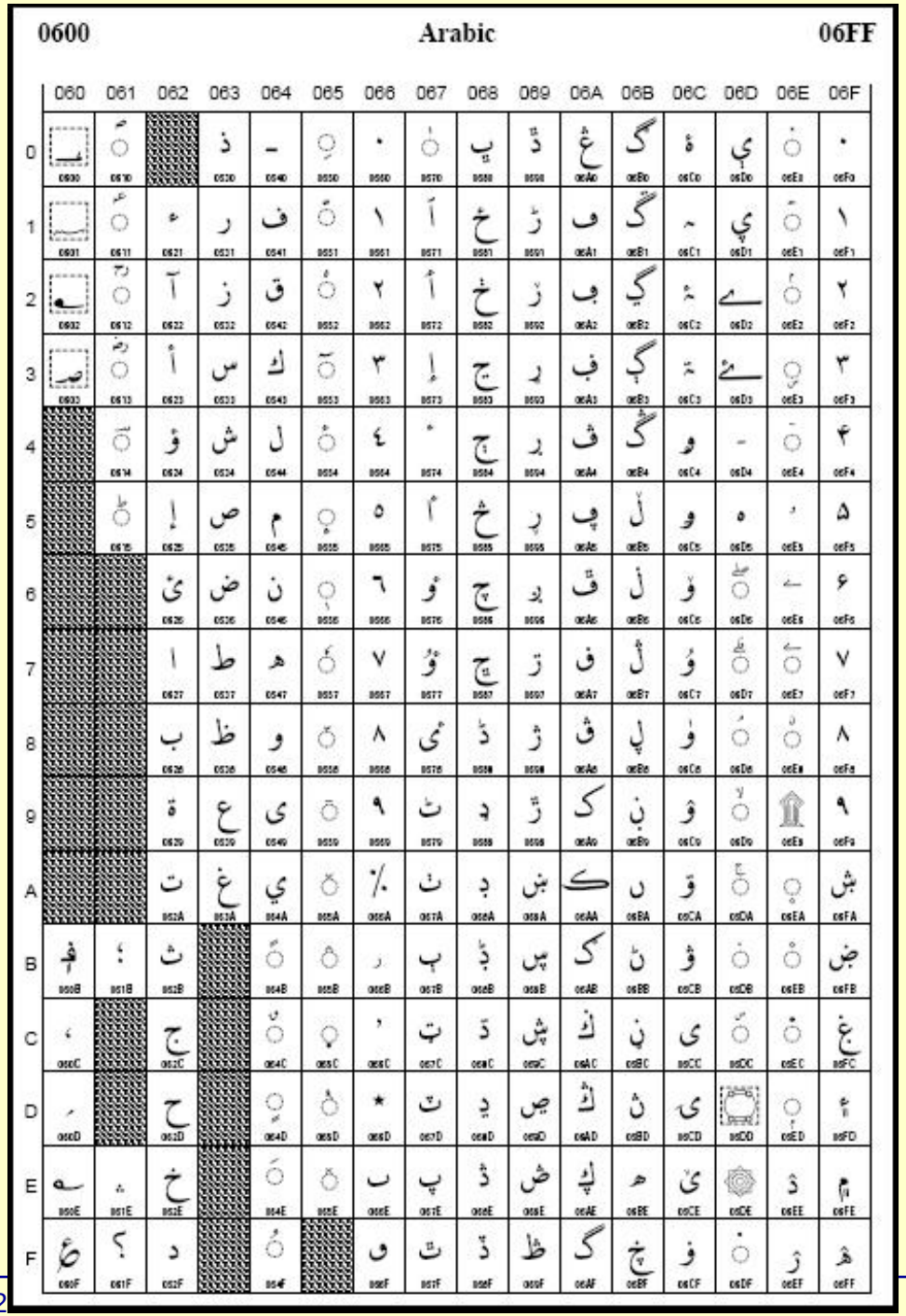
- **Alles hexadezimal**
- **z.B. Spalte 004, Zeile 1**
  → Gesamt 2 Byte 0041
  →´A` repräsentiert



Unicode: Kyrillisch

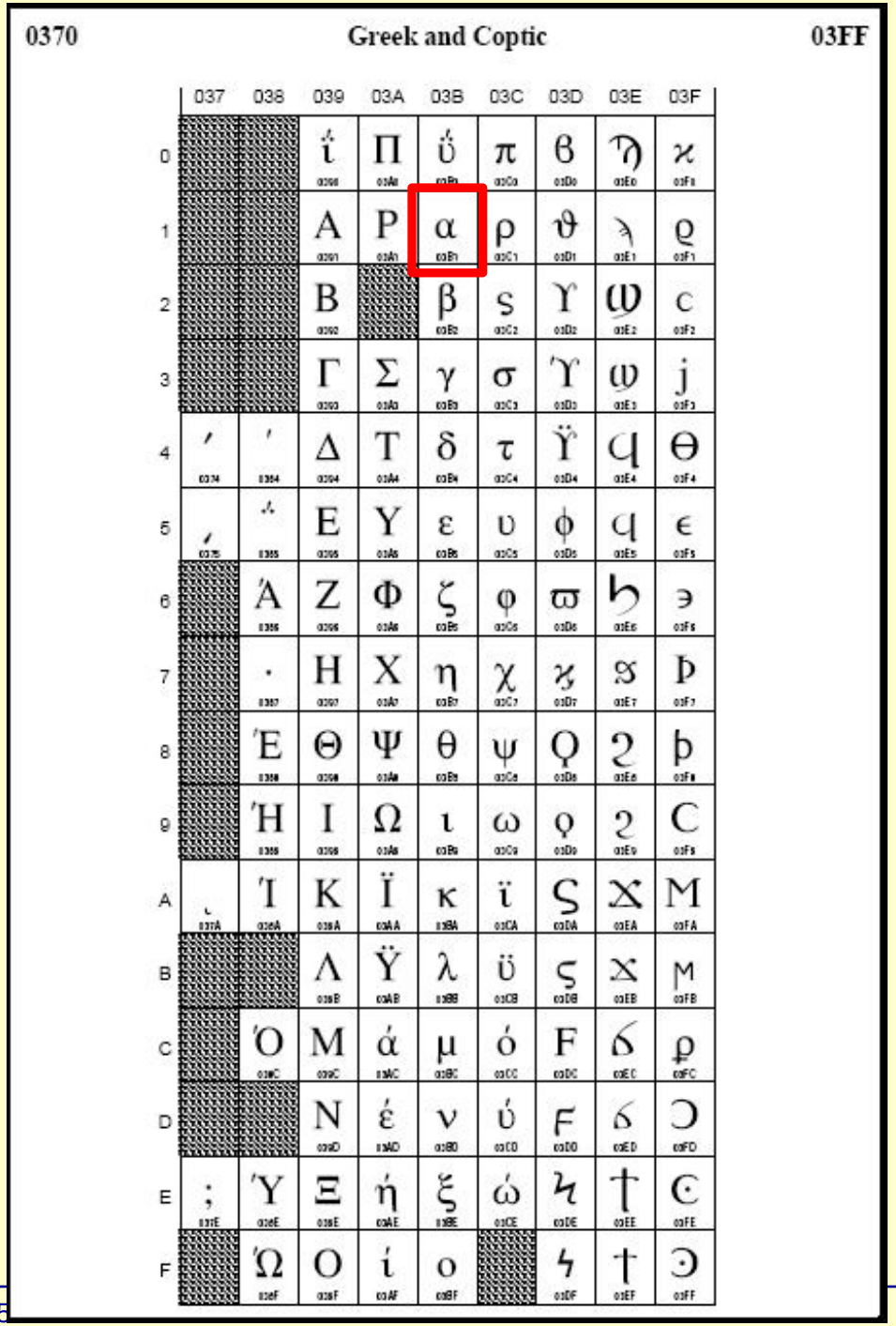


Unicode: Arabisch

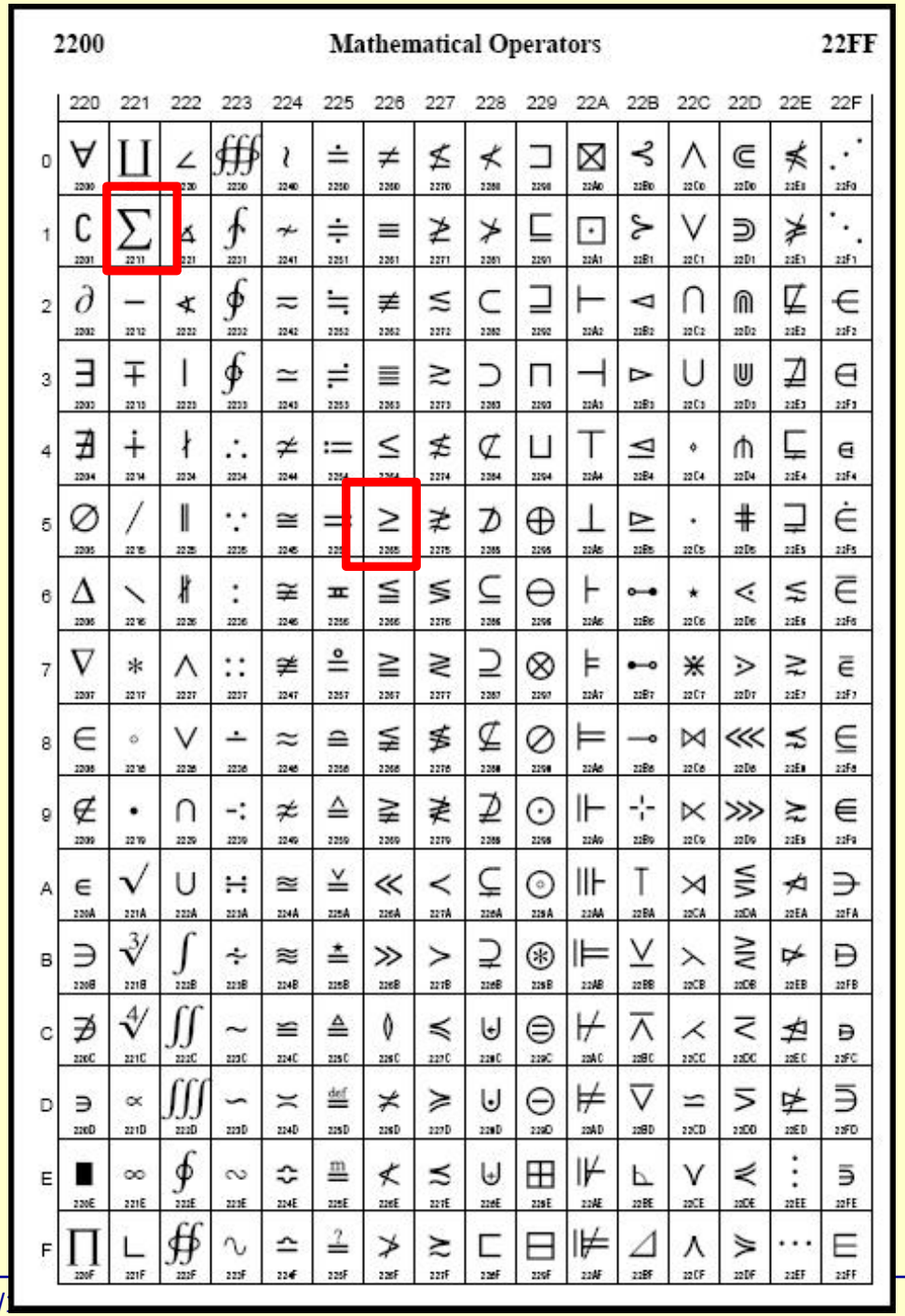


Unicode: Chinesisch (Han) 82 Tabellen, ca. 20.000 Zeichen

Unicode: Griechisch und Koptisch

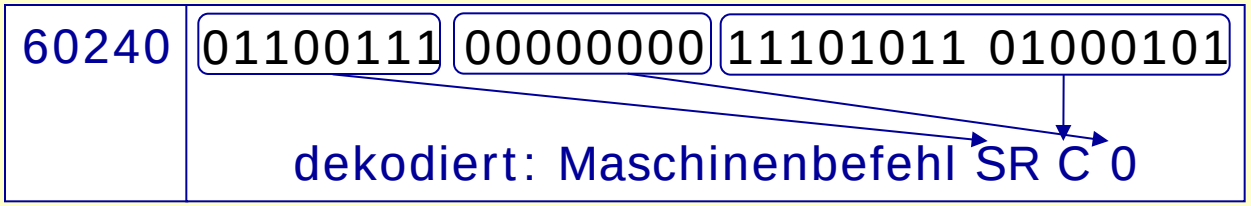


Unicode: Mathematische Operatoren



Speicherbelegung: ein Beispiel

- Maschinenbefehl: SR C 0



- ganze Zahl: 1728113477

  intern dargestellt als:

  01100111 00000000 11101011 01000101

- reelle Zahl:

  $2^{-50} * (1+2^{-8}+2^{-9}+2^{-10}+2^{-12}+2^{-14}+2^{-15}+2^{-17}+2^{-21}+2^{-23})$

  intern dargestellt als:

  01100111 00000000 11101011 01000101

- 4 Zeichen (erw.) ASCII: g NUL δ E

  intern dargestellt als:

  01100111 00000000 11101011 01000101

Welcher Inhalt wird denn nun wirklich durch das 4-Byte-Wort
01100111 00000000 11101011 01000101 repräsentiert ?

Steuerwerk (Prozessor)