Modul: Programmierung B-PRG (ALT) und B-PRG1 Grundlagen der Programmierung 1 und Einführung in die Programmierung EPR WS 11/12

V3 Elementare Datentypen

Prof. Dr. Detlef Krömker
Professur für Graphische Datenverarbeitung
Institut für Informatik
Fachbereich Informatik und Mathematik (12)

www.uni-frankfurt.de

Rückblick

Programmieren – Erste Schritte

Übungsblätter 1: Schon angefangen – fertig?

Zuordung Studiengänge und PRG1(NEU) + EPR und PRG1(ALT)

Fach	Studienbeginn ab WS 11/12	früherer Studien beginn	
Informatik Bachelor	NEU	ALT	
Wirtschaftsinformatik Master	NEU	ALT	
Mathematik Bachelor	ALT	ALT	
Mathematik Master	ALT	ALT	
Bioinformatik Bachelor	ALT	ALT	
Physik Master	in Arbeit	ALT	
Physik der Informationstechnik	läuft aus	ALT	

Fach	Studienbeginn ab WS 11/12	früherer Studienbeginn	
Informatik L3 (modular)	(NEU)	ALT	
Informatik L2/L5	(NEU)	ALT	
Informatik L3 (nicht modular)	(NEU)	ALT	
als Nebenfach	ALT	ALT	

Unser heutiges Lernziele

- *Elementare Datentypen sind solche, die von der Programmiersprache direkt unterstützt werden und meist direkt auf Speicherzellen abgebildet sowie durch entsprechende Hardwarekomponenten effizient behandelt werden können.*

- *Ziel ist es, diese Datentypen als Programmiererin beherrschen zu lernen und ihre Eigenarten zu kennen.*

- *Besonders wichtig und sehr unterschiedlich in verschiedenen Programmiersprachen ist das sogenannte Typsystem, starke versus schwache Typisierung; statische versus dynamische Typisierung.*

- *Dies ist ein sehr umfangreiches Kapitel. Die Konzepte sind allerdings wenig schwierig – Sie müssen sich allerdings durch Übung mit diesen Gegebenheiten vertraut machen. Nutzen Sie die e-Kurse!*

Übersicht

- *Numerische Datentypen*
  - *Ganze Zahlen (Integer)*
  - *Gleitpunktzahlen (floating point number)*
- *(Boolescher Datentyp)*
- *Datentypen für Zeichen und Zeichenketten (Text)*
- *Typisierung*
  - *Starke und Schwache Typisierung*
  - *Statische und Dynamische Typisierung*
  - *Typwandlung*

Grundsätzliches

- **Zahlen** und (Schrift-)**Zeichen** sind zweifellos elementare Datenstrukturen.
- Jede übliche Programmiersprache stellt diese Datentypen als elementare Datentypen zur Verfügung.
- Bei Formulierungen in Programmiersprachen versucht man dabei, die üblichen Notationen der Mathematik weitgehend beizubehalten, in der Regel die **angloamerikanischen** Konventionen.

Mathematische Grundlagen

- Es lohnt sich, die mathematischen Grundlagen kurz zu rekapitulieren, bevor wir die Rechnerrepräsentationen betrachten.

- Aus der Mathematik sind uns insbesondere folgende Zahlenmengen geläufig:

Die Menge der Natürlichen Zahlen	**N** oder $\mathbb{N}$
Die Menge der Ganzen Zahlen	**Z** oder $\mathbb{Z}$
Die Menge der Rationalen Zahlen	**Q** oder $\mathbb{Q}$
Die Menge der Reellen Zahlen	**R** oder $\mathbb{R}$
Die Menge der Komplexen Zahlen	**C** oder $\mathbb{C}$
(Die Menge der Quaternionen)	**H** oder $\mathbb{H}$)

- Allgemein gilt, das $\mathbf{N} \subset \mathbf{Z} \subset \mathbf{Q} \subset \mathbf{R} \subset \mathbf{C} \subset \mathbf{H}$

Dezimalschreibweisen

- Wir schreiben Zahlen gewöhnlich als **Vorzeichenbehaftete Dezimalzahl**, also in einem Stellenwertsystem zur Basis 10. Beispiele sind:

  0 -42 1,414 3,141 -1,59 oder

- Wert **Z** einer Dezimalzahl: $z_m z_{m-1} \ldots z_1 z_0 , z_{-1} z_{-2} \ldots z_{-n}$ $\mathbf{Z} = \sum_{i=-n}^{m} z_i \cdot 10^i$

- Anstelle des Kommas als Dezimaltrenner (kennzeichnet die Stelle (10^0, 10^{-1}) verwendet man im angloamerikanischen Raum den Punkt

- diese Punkt-Schreibweise findet sich in allen (mir bekannten) Programmiersprachen wieder! Also:

  0 -42 1.414 3.141 -1.59

Ganze Zahlen (*Integer*) als Dualzahlen

- Die Dyadik (dyo, griech. = Zwei), also die Darstellung von Zahlen im Dualsystem wurde schon Ende des 17. Jahrhunderts von Leibniz entwickelt.

- Die Ziffern z^i (0 oder 1) werden wie im Dezimalsystem ohne Trennzeichen hintereinander geschrieben, ihr Stellenwert entspricht allerdings der zur Stelle passenden **Zweierpotenz** und **nicht** der Zehnerpotenz.

- Der Wert **Z** der Dualzahl ergibt sich durch Addition dieser Ziffern, welche vorher jeweils mit ihrem Stellenwert 2^i multipliziert werden:

$$z_m z_{m-1} \ldots z_1 z_0 \qquad \mathbf{Z} = \sum_{i=0}^{m} z_i \cdot 2^i$$

Beispiel

Die Ziffernfolge **1101**, stellt nicht (wie im Dezimalsystem) die „Tausendeinhundertundeins“ dar, sondern die Dreizehn, denn im **Dualsystem** berechnet sich der Wert durch

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

$$1101_{(2)} = 13_{(10)}$$

Wenn Sie hiermit nicht vertraut sind, müssen Sie üben ...
Siehe eKurs: Zahlendarstellung

In der Hardware ist die Verarbeitungsbreite, also die Stellenzahl festgelegt, üblich sind heute **32 Bit** und zunehmend häufiger **64 Bit**;

Leibnitz (Anekdote)

Leibnitz sah das Dualsystem ein besonders überzeugendes Sinnbild des christlichen Glaubens an. So schrieb er an den chinesischen Kaiser Kangxi:

„Zu Beginn des ersten Tages war die 1, das heißt Gott.
Zu Beginn des zweiten Tages die 2, denn Himmel und Erde wurden während des ersten geschaffen.
Schließlich zu Beginn des siebenten Tages war schon alles da; deshalb ist der letzte Tag der vollkommenste und der Sabbat, denn an ihm ist alles geschaffen und erfüllt, und deshalb schreibt sich die 7 111, also ohne Null.
Und nur wenn man die Zahlen bloß mit 0 und 1 schreibt, erkennt man die Vollkommenheit des siebenten Tages, der als heilig gilt, und von dem noch bemerkenswert ist, dass seine Charaktere einen Bezug zur Dreifaltigkeit haben."

Andere übliche Basen

Andere übliche Basen sind

8 (Oktalsystem)
16 (Hexadezimalsystem)

Warum können amerikanische Informatiker (Computer Scientists) Weihnachten (25. Dezember) nicht von Halloween (31. Oktober) unterscheiden?

(Antwort: Weil 25(dez) = 31(oct)) ;-)

Kodierung ganzer Zahlen (Zweierkomplement)

- Um auch negative Zahlen darstellen zu können, wird der Bereich der verarbeitbaren positiven Zahlen eingeschränkt und die frei werdenden Kodeworte für negative Zahlen genutzt. Das fast ausschließlich genutzte Verfahren ist das **Zweierkomplement** auch echtes Komplement, 2-Komplement).

- Weitere Verfahren sind das so genannte Einerkomplement; hierbei wird für eine negative Zahl die Kodierung der entsprechenden positiven Zahl stellenweise invertiert. (Nachteil: die Null hat zwei Repräsentationen, es gibt also -0 und +0) und die Arithmetik ist etwas komplizierter.

Zweierkomplement

Voraussetzung ist eine feste Stellenzahl, sagen wir für das folgende Beispiel 8.

Negative Zahlen werden mit einer führenden **1** dargestellt und wie folgt kodiert: Sämtliche Ziffern der entsprechenden positiven Zahl werden invertiert . Zum Ergebnis wird 1 addiert.

Beispiel zur Umwandlung einer negativen Dezimalzahl, hier -4

- 1. Vorzeichen ignorieren und ins Binärsystem konvertieren: $4_{(10)} = 00000100_{(2)}$
- 2. Invertieren, da negativ: 11111011 (Einerkomplement)
- 3. Eins addieren: 11111011 + 00000001 = $11111100_{(2}$ = $-4_{(10}$

Darstellungsbereiche

Mit n Bits lassen sich Zahlen von - 2^{n-1} bis + 2^{n-1} -1 darstellen, also z.B.

- bei 32 Bit: $-2.147.483.648_{(10)}$ bis $+2.147.483.647_{(10)}$

Leider gibt es noch eine weiteres Detail zu beachten:

- Die **Byte-Reihenfolge *(Byte order)***. Im Speicher eines Rechners sind die Speicherzellen in Bytes organisiert und in der Regel auch so adressierbar

Byteorder ... das *„NUXI-Problem“*

Für die Speicherung einer 32-Bit-Binärzahl, hier in Hex-Schreibweise A4B3C2D1 im Hauptspeicher gibt es verschieden Möglichkeiten:

natürlich ?

Adresse	3	2	1	0
Inhalt	A4	B3	C2	D1

(Adresse) Speicher-offset	**Little Endian**	**Big Endian**	Middle Endian obsolet	
0	D1	A4	C2	B3
1	C2	B3	D1	A4
2	B3	C2	A4	D1
3	A4	D1	B3	C2

Noch eine Anekdote

Auch die Etymologie (Herkunft) dieser kuriosen Bezeichnungen „Endian“ ist interessant:

Sie lehnen sich an den satirischen Roman *Gullivers Reisen* ("Gulliver's Travels") von Jonathan Swift (1726) an, in dem der Streit darüber, ob ein Ei am spitzen oder am dicken Ende aufzuschlagen sei, die Bewohner von Liliput in zwei verfeindete Lager spaltet –

die „Little-Endians“ und die „Big-Endians“

in der deutschen Übersetzung des Buches übrigens „Spitz-Ender“ und „Dick-Ender“ ... aber das sagt in der Fachsprache niemand ;-)

Wo findet man was?

- Im Internet ist das **Big Endian** als *Network Byte Order* festgelegt. Die sogenannte *Host Byte Order* ist ggf. bei heute gebräuchlichen Systemen verschieden:

- **Little Endian:** Intel-x86-Prozessoren und auch das Betriebssystem Windows

- **Big Endian;** Power PC (umschaltbar), Motorola-68000-Familie, MIPS Prozessoren, HP-UX, **Internet**

Gleitpunktzahlen (Gleitkommazahlen; *floating point numbers*)

Eine Gleitkommazahl
(auch: Gleitpunktzahl, manchmal auch Fließkommazahl; **halblogarithmische Darstellung**, engl: *floating point number*)

ist eine **meist** approximierte Kodierung einer rationalen Zahl in einer festgelegten Anzahl von Bits (meist 32, 64, seltener 128 oder gar 256 Bits).

Die Menge der Gleitkommazahlen ist eine endliche **Teilmenge** der rationalen Zahlen, meist erweitert um einige Spezialelemente (+Unendlich, –Unendlich, NaN (="Not A Number"), –0, usw.

Gleitpunktzahlen (allgemein)

- Zur Kodierung einer Zahl wird eine **Mantisse m** und ein **Exponent e** zu einer bestimmten, festen Basis b benutzt
  $8{,}432 \cdot 10^{23}$ (wissenschaftliche Zahlendarstellung)

- Eine Zahl $a \neq 0$ wird durch zwei Zahlen *m* und *e* solcherart dargestellt, dass $a = m \cdot b^e$ gilt.
  Dabei ist die **Basis b** (auch: Radix) eine beliebige natürliche Zahl >1.

- Die Zahl *m* wird **Mantisse** genannt und ist eine Zahl mit ***p*** Stellen (der so genannten Präzision, engl. ***precision***) der Form ±z0,z-1z-2... zm-1 .

- Hierbei steht z für eine Ziffer zwischen 0 und *b* – 1.

Normalisierte und normierte **Mantisse**

- Liegt die Mantisse im Wertebereich $1 \leq m < b-1$ (im Fall b=2 ist die Vorkommazahl 1), so spricht man von einer **normalisierten Mantisse**.
- Liegt die Mantisse im Wertebereich $1/b \leq m < 1$ (also im Fall b=2, ist die Vorkommazahl 0 und die erste Nachkommastelle ist ungleich 0), so spricht man von einer **normierten Mantisse** (0.xxxx-Form).
- Gegenüber einer Integerdarstellung kann mit Gleitkommazahlen bei gleichem Speicherplatzbedarf ein viel größerer Wertebereich abgedeckt werden.

IEEE 754 und IEEE 754r

Das gebräuchliche und häufig auch durch Hardware unterstützte Format ist in der Norm **IEEE 754** (ANSI/IEEE Std 754-1985; IEC-60559 - International version) festgelegt.

Diese Norm legt die Standarddarstellungen für **binäre** Gleitkommazahlen fest und definiert Verfahren für die Durchführung mathematischer Operationen, insbesondere für Rundungen.

Beinahe alle modernen Prozessoren folgen diesem Standard.
Ausnahmen:

- **Java Virtual Machine** mit den Java Typen float und double, die nur einen Subset der IEEE 754 Funktionalität unterstützt.

IEEE 754 r

benutzt **normalisierte** Gleitkommazahlen (NZ)

auf der Basis b = 2.

Das **Vorzeichen *s*** = $(-1)^S$ wird in einem Bit S
(S = 0 positive Zahlen und S = 1 negative Zahlen)

Der **Exponent e** ergibt sich aus der in den Exponentenbits gespeicherten nichtnegativen Binärzahl E durch Subtraktion eines festen **Biaswertes B**:

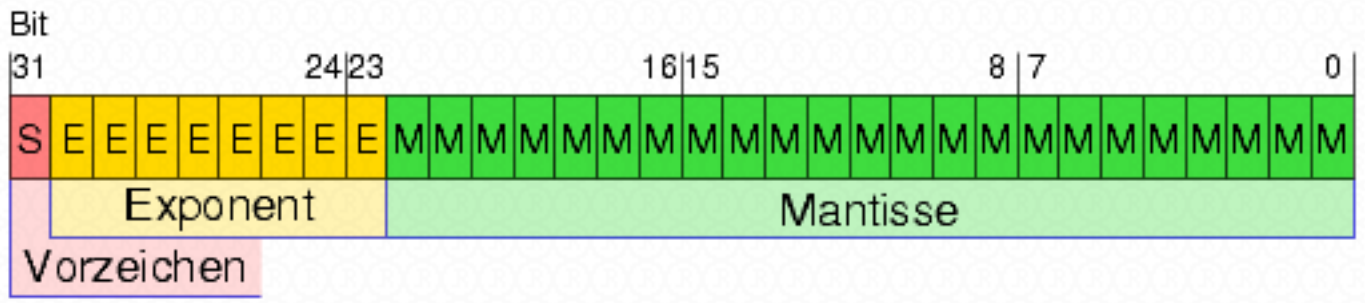
$e = E - B$.

IEEE 754

Schließlich ist die Mantisse ein Wert, der sich aus den p Mantissenbits M berechnet als $m = 1 + M / 2p$. Einfacher ausgedrückt, denkt man sich an das Mantissenbitmuster M links eine 1. angehängt:

- m = 1.M.
- $s = (-1)S$
- $e = E - B$

Da die Mantisse immer mit 1. beginnt, braucht dieses Bit nicht mehr gespeichert zu werden (**hidden Bit**).
Damit gewinnt man ein zusätzliches Bit Genauigkeit



single precission

„*The Perils* (Risiken) *of Floating Point*“

- Gleitkommazahlen sind nicht gleich dicht (im gleichen absoluten Abstand) auf dem Zahlenstrahl:

- Bei *single* Precission liegen z.B. **zwischen 1 und 2** 8.388.607 verschiedene Gleitkommazahlen, zwischen **1023 und 1024** dagegen nur 8191, weil eben nicht die absolute Genauigkeit konstant ist, sondern die relative, also die Anzahl der signifikanten Stellen.

- Weitere Programmierempfehlungen und -regeln betrachten wir im eKurs.

Operationen auf Zahlen

geordnet nach Vorrangregeln

Auszug aus Programmier-handzettel 1

Operation	Beschreibung
`+x, -x,` `~x`	Einstellige Operatoren Invertiere x
`x ** y`	Exponential-Bildung x^y (Achtung: rechts-assoziativ)

`x * y`
`x / y`
`x % y`
`x // y`
	Multiplikation (Wiederholung)
	Division
	Modulo (-Division) =
	(Ganzzahliger) Rest
	Restlose Division [2])

`x + y` `x - y`	Addition (Konkatenation) Subtraktion
`x << y, x >> y`	Bitweises Schieben (nur bei Integer)
`x & y`	Bitweises Und (nur bei Integer)
`x ^ y`	Bitweises exklusives Oder (nur bei Integer)
`x | y`	Bitweises Oder (nur bei Integer)

Klammerung

Wie in der Mathematik üblich, kann man die Auswertereihenfolge durch Klammerung (…) beeinflussen. Zugelassen sind allerdings **nur runde Klammern**. Diese können aber beliebig geschachtelt werden. Hierzu einige Beispiele:

```
>>> 1+2*3+4
11

>>> (1+2)*(3+4)
21

>>> ((1+2)*3)+4
13
```

Übersicht

- *Numerische Datentypen*
  - *Ganze Zahlen (Integer)*
  - *Gleitpunktzahlen (floating point number)*
- *Boolescher Datentyp ➔ nächste Woche*
- *Datentypen für Zeichen und Zeichenketten (Text)*
- *Typisierung*
  - *Starke und Schwache Typisierung*
  - *Statische und Dynamische Typisierung*
  - *Typwandlung*

Übersicht

Was ist Text?

Welche Schriftarten existieren?

Wie werden diese im Computer codiert?

Was sind Zeichenketten?

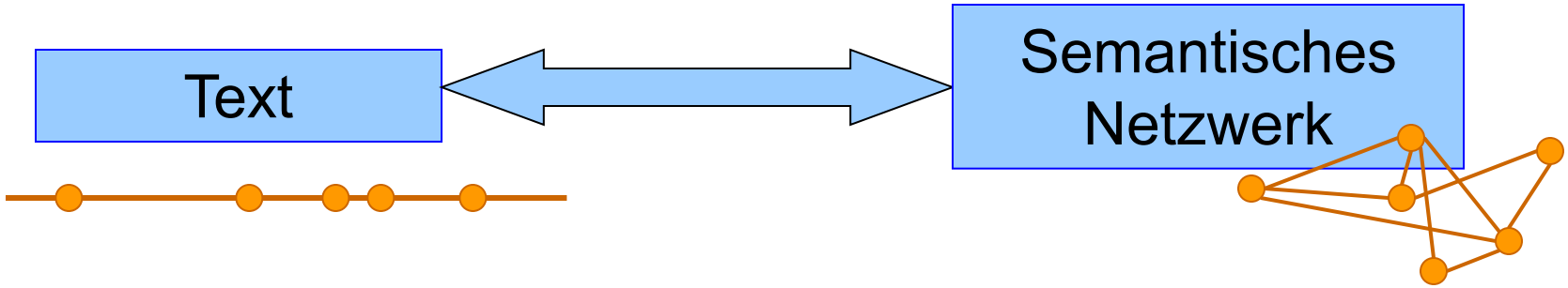
Text

= geschriebene Sprache (im engeren Sinne)

kommt von lat. *textus* =

- Gewebe
- Geflecht
- Stoff

Vielleicht eine Erklärung:
Funktionale Verbindung zwischen Gewebe und Text



linear-temporäre Aneinanderreihung von Zeichen und Worten wird mental in eine semantische Netzwerkrepräsentation transformiert

=>

Text „verstrickt“ und „verflechtet“ Objekte und Akteure miteinander,

Darstellung des Textes

benötigt eine Schrift, deren Zeichen wahlweise

- Phoneme (bei uns)
- Silben
- Wörter bzw. Begriffe

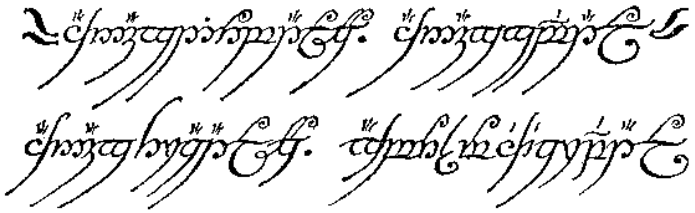
kodieren.

Alphabetschrift

Alphabet (von gr. αλφάβητο [alfáwito] **- Alpha & Beta)**

Beispiele:

- **Lateinische Schriften:** **Den småne skriften**
- **IPA-Lautschrift:** [alfáwito]
- **Kyrillisch:** **Ду бист кайн Весси!**
- **Griechische Schrift:** αλφάβητο
- **Tengwar, Sindarin ... :**



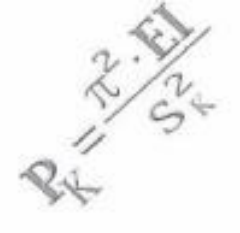
Wortbildschrift, Logogrammschrift

Beispiele:

- **Hanzi (chinesisch)**
- **Hanja (koreanisch)**
- **Maja-Schrift**
- **Jurjen**
- **Tangut**

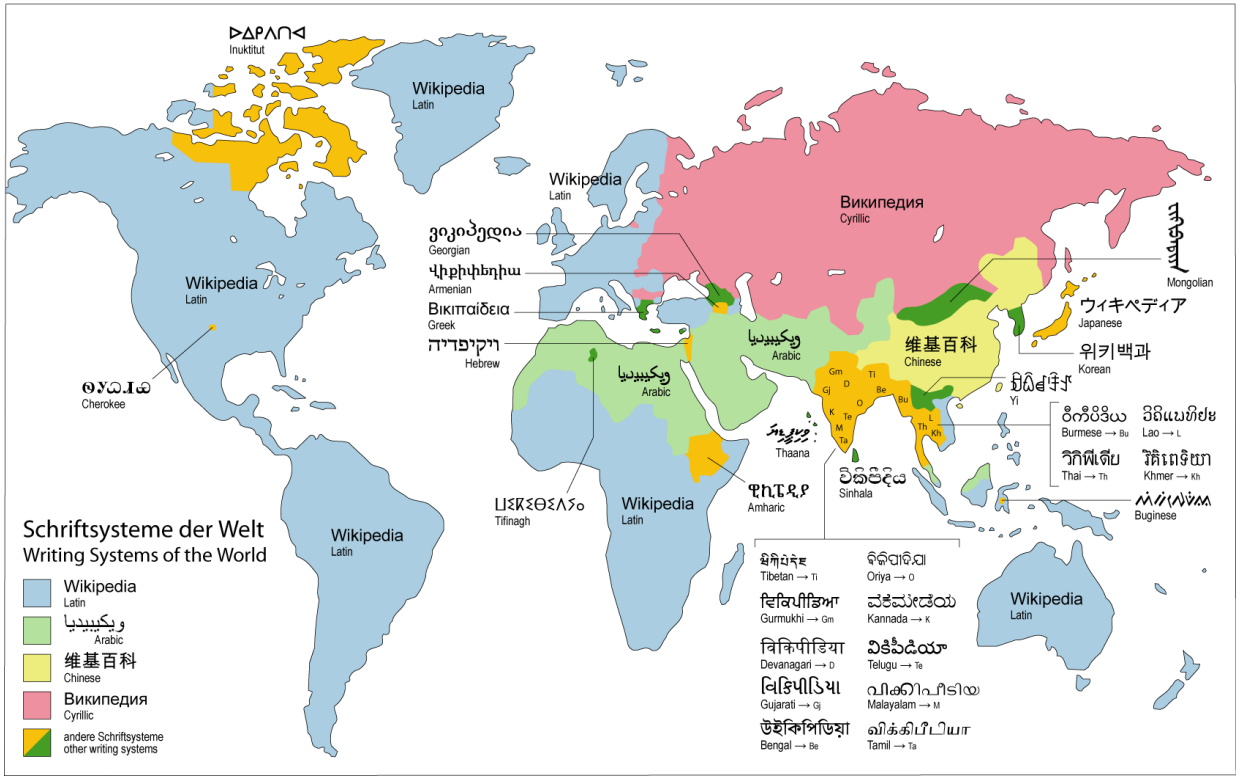
Sonderformen der Logogramme

Ziffernsysteme
&
Symbolik der Mathematik
oder der Chemie



Piktogramme

Schriftsysteme der Welt



http://de.wikipedia.org/wiki/Schrift

Begriff: Alphabet in der Informatik

ist in der Informatik weiter gefasst als in der Lingustik:

Unter **Alphabet** versteht man (z.B. nach DIN 44300)
eine **total geordnete endliche Menge** (oft nichtleere endliche Menge) von unterscheidbaren Symbolen (Zeichen).

Häufig symbolisiert durch Σ

Begriff: Zeichenreihe

Zeichenreihen = endliche lineare Reihen von Zeichen eines Alphabets **Σ**

Übrigens: Auch die Zeichenreihe, die keine Zeichen enthält, ist ein Wort - das leere Wort. Es wird mit ε bezeichnet.

Begriff: Zeichensatz

Die Zuordnung von alphanumerischen Zeichen (Buchstaben und Ziffern) sowie Sonderzeichen zu einem Zahlencode.

Beispiele:

- **ASCII (meist verwendet)**
- IBM **EBCDIC (verliert an Bedeutung)**

Begriff: Schriftart

Synonyme: font, typeface

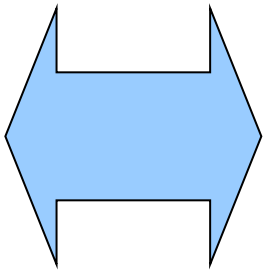
Bekannte Schriftarten:
Arial
Times New Roman
Frutiger light
Courier New
Impact
Century Gothic

Proportionale vs. Nicht-proportionale Schriften

Proportionale Schriften

(veränderliche Typenbreite)

(Arial, Times, Frutiger, ..)



Nichtproportionale Schriften

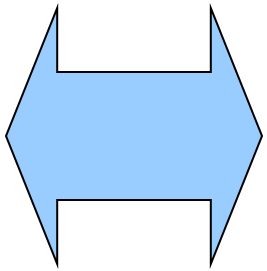
(fixe Typenbreite)

(Courier New, Lucida Console, ...)

Grotesk vs. Antiqua

Grotesten-Schriften
(ohne Serifen)

(Arial, Frutiger, `Lucida Console, ...`)



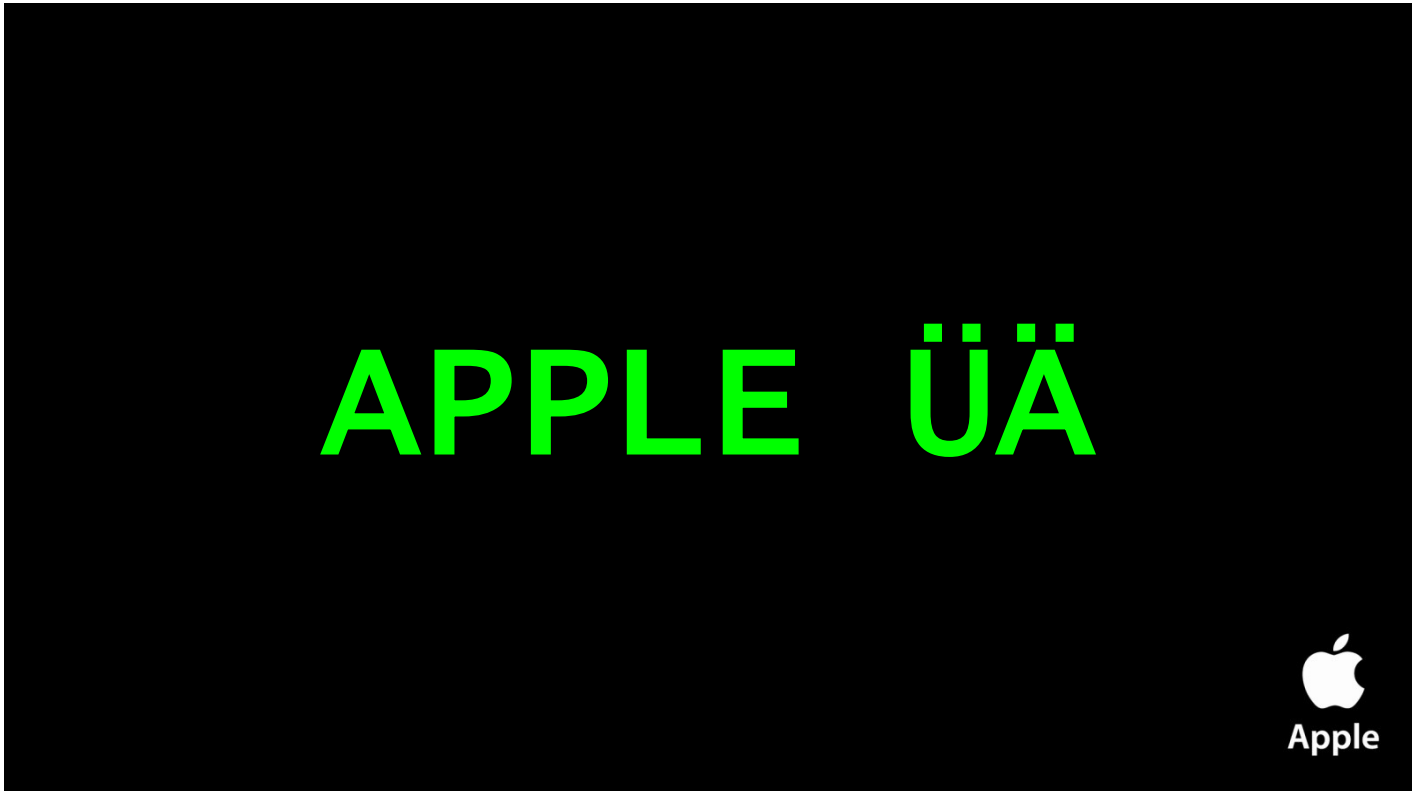
Antiqua-Schriften
(mit Serifen)

(Times, `Courier New, ..`)

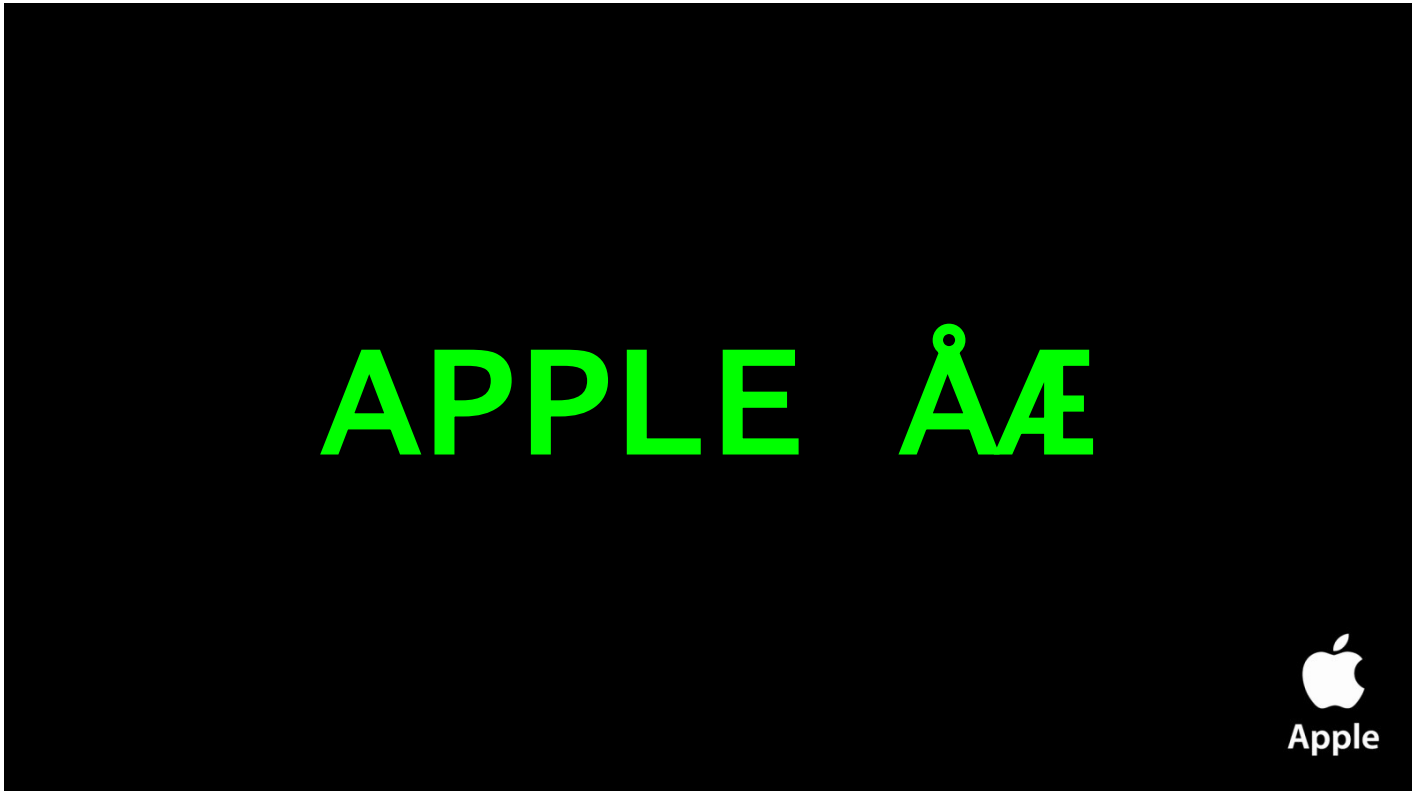
Internationale Zeichensätze

ASCII	Einer der ältesten Computer-Zeichensätze – 7 Bit	1963	Sehr weit verbreitet, (Programmier-sprachen, Internet-Adressen, etc.)
ISO/IEC 8859	15 verschiedenen Kodierungen zur Abdeckung europäischer Sprachen sowie Arabisch, Hebräisch, Thailändisch und Türkisch	1986	u.a. in Linux und MS Windows verwendet.
Unicode ISO/IEC 10646 -– 1991	Internationaler Standard – (7) 8, 16 oder 32 Bit	1991	nimmt an Bedeutung **stark zu**

ASCII-Anekdote



ASCII-Anekdote

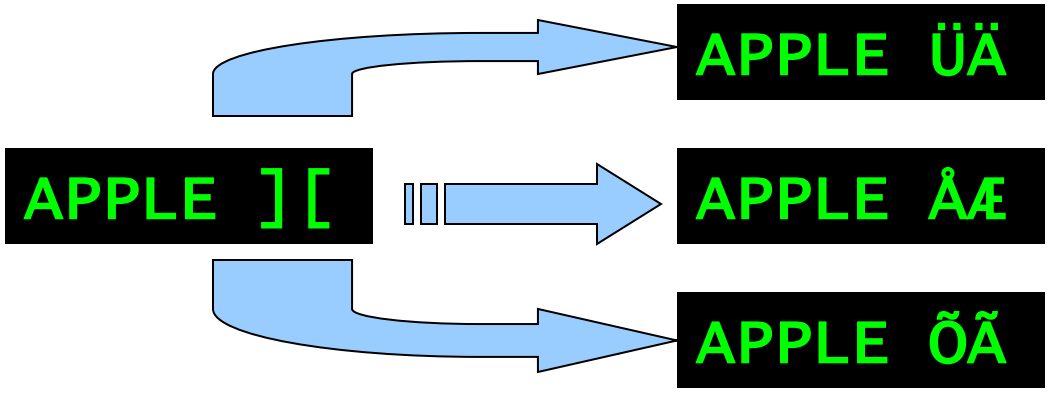


ASCII genauer unter die Lupe genommen

Problem: ASCII enthält keine diakritischen Zeichen!

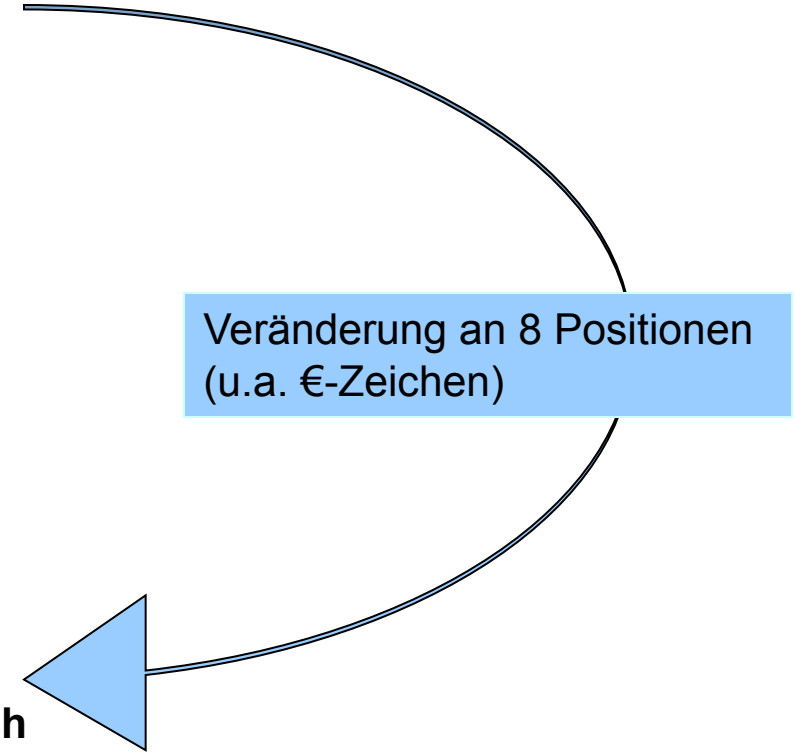
⇒ **1972 Einführung von ISO 646-IRV**

⇒ **Internationale Kompatibilitätsprobleme**



ISO IEC 8859 (fast nur noch historisch bedeutsam!)

-1	Latin-1	Westeuropäisch
-2	Latin-2	Osteuropäisch
-3	Latin-3	Südeuropäisch
-4	Latin-4	Baltisch
-5		Kyrillisch
-6		Arabisch
-7		Griechisch
-8		Hebräisch
-9	Latin-5	Türkisch
-10	Latin-6	Nordisch
-11		Thai
-13	Latin-7	Baltisch
-14	Latin-8	Keltisch
-15	Latin-9	Westeuropäisch
-16	Latin-10	Südosteuropäisch



Unicode

- **Identisch mit Universal Character Set (UCS nach ISO 10646 ISO 10646)**
- **32 Bit / Zeichen => über vier Milliarden Möglichkeiten**
- **Aber nur ~1 Million prinzipiell erlaubte Code-Werte**

  - *Unicode provides a unique number for every character,*
    no matter what the platform,
    no matter what the program,
    no matter what the language.

Zeichenketten in Python

Python implementierte in der Version 2.X zwei verschiedene Basis-Typen für Zeichenketten (*strings*):

- **Strings** (8-Bit, ein Oktett, ein Byte)
- **Unicode-Strings** (variable Codelänge von 8-32 Bits)

Ab Version 3.0 nur noch strings

Strings (variable Codelänge von 8-32 Bits)

Übersicht

- *Numerische Datentypen*
  - *Ganze Zahlen (Integer)*
  - *Gleitpunktzahlen (floating point number)*
- *Boolescher Datentyp*
- *Datentypen für Zeichen und Zeichenketten (Text)*
- *Typisierung*
  - *Starke und Schwache Typisierung*
  - *Statische und Dynamische Typisierung*
  - *Typwandlung*

Typisierung

Beispiel:

X = ‚ANTON'
Y = , & '
Z = ‚BERTA'.
U = X + Y + Z

arithmetisch interpretiert macht das keinen Sinn, erst recht nicht dieses auf Datenebene plump zu errechnen.

Vielmehr bedeutet der Operator + bei Zeichenketten eine Aneinanderreihung, also etwa

U = ‚ANTON & BERTA'

Datentypen

- Die Bedeutung von Operatoren in Ausdrücken hängt von der Art (=dem **Typ**, engl. *type*) der Daten ab

  Gleiche Operatorenzeichen (z.B, +) können abhängig vom Datentyp durchaus Verschiedenes bedeuten

- Ein **Datentyp** in der Informatik ist die **Zusammenfassung von Objektmengen mit den darauf definierten Operationen**.

- Grundsätzlich dürfen nur gleiche Datentypen miteinander verknüpft werden! (➔ starke Typisierung).

- In Python ist dann auch das Ergebnis vom gleichen Typ (➔ dynamische Typisierung).

Beispiel (1):

- 2 + 3.5 + '0001'
- Wir als Mensch interpretieren vermutlich:

  2.0 + 3.5 + 1.0 = 6.5 (als Gleitpunktzahl)
- Aber warum nicht:
- '2' + '3.5' + '0001' = '23.50001' (als String) interpretieren?
- Also: Es ist entscheidend wichtig, den Typ der Operanden zu kennen!

Beispiel (2)

```
a = 2  # Integer 32 bit
c = '0001'  # String 4 byte
b = 3.5  # Float 64 bit
```

- Diese drei Variablen könnten im Speicher etwa wie folgt repräsentiert sein:
-
- a ➔ 00000002 32-Bit Integer-Kodierung für 2
- c ➔ 30303031 ASCII-Kodierung für den String '0001'
- b ➔ 400C0000 64-Bit Float-Kodierung für 3.5
  00000000

a als Float interpretiert, so hätte a den Wert $4.643426084 \cdot 10^{-314}$,
c als Float interpretiert, so hätte c den Wert $1.39804468550329 \cdot 10^{-76}$,
b als Integer interpretiert, so hätte es den Wert 1,074,528,256 ,

- kurz, ein totales Tohuwabohu.

Beispiel (3)

Achtung: Bei der Division von Integer-Zahlen können Brüche (➔ float) entstehen: z.B.

$7 / 2 = 3\ ½$ oder 3.5 (float) aber $6 / 2 = 3$ (integer)

In der Mathematik führen wir diese „Coercion“ automatisch durch, oder?
In Python (ab Version 3.X) auch!

Aber es gibt auch die ganzzahlige Division:

$7 // 2 = 3.$

Und die Modulo-Divison (Rest):

$7 \% 2 = 1$

Also:

Es ist daher offensichtlich, dass es gilt, solche Situationen in jedem Fall zu vermeiden und mögliche Programmierfehler so früh wie möglich zu entdecken Hierzu unterscheiden wir:

Python

- -
- -

- schwache Typisierung (*weak typing*)
- statische Typisierung (*static typing*

Starke Typisierung

Bei der **starken Typisierung** (*strong typing* oder strengen, strikten Typisierung) bleibt eine einmal durchgeführte Bindung zwischen Variable und Datentyp in jedem Fall bestehen. Eine nicht stark typisierte Sprache bezeichnet man als **schwach typisiert**.

- stark typisierte Sprachen: Java, Python, Pascal
- schwach typisierte Sprachen: C / C++, PHP, Perl, JavaScript
- Schützt vor vielen Programmierfehlern!

Dynamische – statische Typisierung

- **dynamischen Typisierung** (engl. *dynamic typing*) erfolgt die Typzuweisung der Variablen zur Laufzeit eines Programms, z.B. durch eine Zuweisung
- Dies erspart es dem Programmierer, die Typisierung „von Hand“ durch eine Deklaration.

- Bei der **statischen Typisierung** muss zur Übersetzungszeit der Datentyp von Variablen bekannt sein. Dies erfolgt in der Regel durch Deklaration.

- Unter **Deklaration** versteht man die Festlegung von Bezeichner, Datentyp, Dimension und weiteren Aspekten einer Variablen.

Casting und Coercion

- **implizite** Typkonvertierung oder **coercion** (*engl. Nötigung, Zwang*)
- **explizite** Typkonvertierung oder **cast(ing)** (engl eingießen, formen, werfen, …)
- **Coercion** finden wir sehr häufig bei Zahlen, also:

  Integer ➔ Float ➔ Complex

  Nicht unproblematisch, aber macht Sinn ... Trotzdem Vorsicht!

Casting-Funktionen in Python

Typ (Kürzel)	Konvertierungsfunktionen
Integer (int)	`int(O[,basis])` `ord(C)`
Long Integer (long)	`long(O[,basis])`
Flaot (float)	`float(O)`
Complex (complex)	`complex(O)`
Boolean (bool)	`bool(O)`

String (str)	`str(O)`
	`repr(O)` oder `` `I` `` (Backticks)
	`chr(I)`
	`hex(I)`
	`oct(I)`

Zusammenfassung

Viele, viele Details, ich weiß!

Jetzt müssen Sie üben, mit den eKursen und dem Interpreter

Ausblick ... nächsten Mittwoch

Programmflußkontrolle:

- **Verzweigungen**
- **Schleifen**
- **Funktionen und Prozeduren**

... und, danke für Ihre Aufmerksamkeit!