Was ist ein Algorithmus?

Duden online:

„Rechenvorgang nach einem bestimmten (sich wiederholenden) Schema“

Beispiele für Algorithmen bereits in der Antike, etwa der Euklidsche Algorithmus zur Berechnung des ggT:

„Wenn CD aber AB nicht misst, und man nimmt bei AB, CD abwechselnd immer das kleinere vom größeren weg, dann muss (schließlich) eine Zahl übrig bleiben, die die vorangehende misst.“

aus *Euklid: Die Elemente, Buch VII (Clemens Thaer)*

Was ist ein Algorithmus?

Duden online:

„Rechenvorgang nach einem bestimmten (sich wiederholenden) Schema“

Beispiele für Algorithmen bereits in der Antike, etwa der Euklidsche Algorithmus zur Berechnung des ggT:

„Wenn CD aber AB nicht misst, und man nimmt bei AB, CD abwechselnd immer das kleinere vom größeren weg, dann muss (schließlich) eine Zahl übrig bleiben, die die vorangehende misst.“

aus *Euklid: Die Elemente, Buch VII (Clemens Thaer)*

Was ist ein Algorithmus?

M. Broy: Informatik: Eine grundlegende Einführung

„Ein Algorithmus ist ein Verfahren

- mit einer **präzisen** (d.h. in einer genau festgelegten Sprache abgefassten),
- **endlichen** Beschreibung,
- unter Verwendung
  - **effektiver** (d.h. tatsächlich ausführbarer),
  - **elementarer** (Verarbeitungs-) Schritte.“

Was ist ein Algorithmus?

H. Rogers:
Theory of Recursive Functions and Effective Computability

„Ein Algorithmus ist eine

- deterministische Handlungsvorschrift,
- die auf eine bestimmte Klasse von Eingaben angewendet werden kann,
- und für jede dieser Eingaben eine korrespondierende Ausgabe liefert.“

Im weiteren Verlauf des Buches wird mathematische Theorie zur ↑Berechenbarkeit entwickelt

↑theoretische Informatik

Was ist ein Algorithmus?

Mathematische Definition Algorithmus

Eine Berechnungsvorschrift zur Lösung eines Problems heißt Algorithmus genau dann, wenn

- eine zu dieser Berechnungsvorschrift äquivalente Turingmaschine existiert,
- die für jede Eingabe, die eine Lösung besitzt, terminiert.

Alan Turing (1936): Turingmaschine als mathematisches Modell eines Computers

↑theoretische Informatik

3 Einführung

Ein Problem besteht darin, aus einer Menge von Informationen eine weitere (unbekannte) Information zu bestimmen.

mathematisch:

Ein Problem beschreibt eine Funktion $f : E \to A$, mit

$E =$ zulässige Eingaben und $A =$ mögliche Ausgaben.

Beispiele:

- Addition: $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$
- Primzahltest: $f : \mathbb{N} \to \{0, 1\}$
- Schach: $f : \mathcal{P} \to \mathcal{P}$, wobei $\mathcal{P}$ die Menge aller Schachpositionen ist, und $f(p)$ der beste Zug in Position p.

3 Einführung

Ein Problem besteht darin, aus einer Menge von Informationen eine weitere (unbekannte) Information zu bestimmen.

mathematisch:
Ein Problem beschreibt eine Funktion $f : E \to A$, mit
$E =$ zulässige Eingaben und $A =$ mögliche Ausgaben.

Beispiele:

- Addition: $f : \mathbb{Q} \times \mathbb{Q} \to \mathbb{Q}$
- Primzahltest: $f : \mathbb{N} \to \{\text{yes, no}\}$
- Schach: $f : \mathcal{P} \to Z$, wobei $\mathcal{P}$ die Menge aller Schachpositionen ist, und $f(P)$, der beste Zug in Position P.

3 Einführung

Ein Problem besteht darin, aus einer Menge von Informationen eine weitere (unbekannte) Information zu bestimmen.

mathematisch:
Ein Problem beschreibt eine Funktion $f : E \to A$, mit
$E =$ zulässige Eingaben und $A =$ mögliche Ausgaben.

Beispiele:

- Addition: $f : \mathbb{Q} \times \mathbb{Q} \to \mathbb{Q}$
- Primzahltest: $f : \mathbb{N} \to \{\text{yes}, \text{no}\}$
- Schach: $f : \mathcal{P} \to \mathcal{Z}$, wobei $\mathcal{P}$ die Menge aller Schachpositionen ist, und $f(P)$, der beste Zug in Position P.

3 Einführung

Ein Problem besteht darin, aus einer Menge von Informationen eine weitere (unbekannte) Information zu bestimmen.

mathematisch:
Ein Problem beschreibt eine Funktion $f : E \to A$, mit
$E =$ zulässige Eingaben und $A =$ mögliche Ausgaben.

Beispiele:

- Addition: $f : \mathbb{Q} \times \mathbb{Q} \to \mathbb{Q}$
- Primzahltest: $f : \mathbb{N} \to \{\text{yes}, \text{no}\}$
- Schach: $f : \mathcal{P} \to \mathcal{Z}$, wobei $\mathcal{P}$ die Menge aller Schachpositionen ist, und $f(P)$, der beste Zug in Position P.

3 Einführung

Ein Problem besteht darin, aus einer Menge von Informationen eine weitere (unbekannte) Information zu bestimmen.

mathematisch:
Ein Problem beschreibt eine Funktion $f : E \to A$, mit
$E =$ zulässige Eingaben und $A =$ mögliche Ausgaben.

Beispiele:

- Addition: $f : \mathbb{Q} \times \mathbb{Q} \to \mathbb{Q}$
- Primzahltest: $f : \mathbb{N} \to \{\text{yes}, \text{no}\}$
- Schach: $f : \mathcal{P} \to \mathcal{Z}$, wobei $\mathcal{P}$ die Menge aller Schachpositionen ist, und $f(P)$, der beste Zug in Position P.

Algorithmus

Ein Algorithmus ist ein exaktes Verfahren zur Lösung eines Problems, d.h. zur Bestimmung der gewünschten Resultate.

Man sagt auch ein Algorithmus berechnet eine Funktion f.

Ausschnitt aus Briefmarke, Soviet Union 1983
Public Domain

Abu Abdallah Muhamed ibn Musa al-Chwarizmi, ca. 780–835

Algorithmus

Beobachtung:
Nicht jedes Problem läßt sich durch einen Algorithmus lösen (↑Berechenbarkeitstheorie).

Beweisidee:

- es gibt überabzählbar unendlich viele Probleme
- es gibt abzählbar unendlich viele Algorithmen

Algorithmus

Beobachtung:
Nicht jedes Problem läßt sich durch einen Algorithmus lösen (↑Berechenbarkeitstheorie).

Beweisidee:

- es gibt überabzählbar unendlich viele Probleme
- es gibt abzählbar unendlich viele Algorithmen

Algorithmus

Das exakte Verfahren besteht i.a. darin, eine Abfolge von elementaren Einzelschritten der Verarbeitung festzulegen.

Beispiel: Alltagsalgorithmen

Resultat	*Algorithmus*	*Einzelschritte*
Pullover	Strickmuster	eine links, eine rechts, eine fallen lassen
Kuchen	Rezept	nimm 3 Eier . . .
Konzert	Partitur	Noten

Beispiel: Euklidischer Algorithmus

Problem: geg. $a, b \in \mathbb{N}, a, b \neq 0$. Bestimme $\mathrm{ggT}(a, b)$.

Algorithmus:

1. Falls $a = b$, brich Berechnung ab. Es gilt $\mathrm{ggT}(a, b) = a$. Ansonsten gehe zu Schritt 2.
2. Falls $a > b$, ersetze a durch $a - b$ und setze Berechnung in Schritt 1 fort. Ansonsten gehe zu Schritt 3.
3. Es gilt $a < b$. Ersetze b durch $b - a$ und setze Berechnung in Schritt 1 fort.

Beispiel: Euklidischer Algorithmus

Warum geht das?

Wir zeigen, für $a > b$: $\mathrm{ggT}(a,b) = \mathrm{ggT}(a-b,b)$.

Seien $g = \mathrm{ggT}(a,b)$, $g' = \mathrm{ggT}(a-b,b)$.

Dann gilt:

Beispiel: Euklidischer Algorithmus

Warum geht das?

Wir zeigen, für $a > b$: $\mathrm{ggT}(a,b) = \mathrm{ggT}(a-b,b)$.

Seien $g = \mathrm{ggT}(a,b)$, $g' = \mathrm{ggT}(a-b,b)$.

Dann gilt:

$$\begin{array}{rcl} a & = & q_a \cdot g \\ b & = & q_b \cdot g \end{array} \quad \text{und} \quad \begin{array}{rcl} a-b & = & q'_{a-b} \cdot g' \\ b & = & q'_b \cdot g' \end{array}$$

Beispiel: Euklidischer Algorithmus

Warum geht das?

Wir zeigen, für $a > b$: $\mathrm{ggT}(a,b) = \mathrm{ggT}(a-b,b)$.

Seien $g = \mathrm{ggT}(a,b)$, $g' = \mathrm{ggT}(a-b,b)$.

Dann gilt:

$$\begin{aligned} a &= q_a \cdot g \\ b &= q_b \cdot g \end{aligned} \quad \text{und} \quad \begin{aligned} a-b &= q'_{a-b} \cdot g' \\ b &= q'_b \cdot g' \end{aligned}$$

$$\begin{aligned} a-b &= (q_a - q_b) \cdot g \\ b &= q_b \cdot g \end{aligned} \quad \text{und} \quad \begin{aligned} a &= (q'_{a-b} + q'_b) \cdot g' \\ b &= q'_b \cdot g' \end{aligned}$$

Beispiel: Euklidischer Algorithmus

Hier sind $q_a, q_b, q'_{a-b}, q'_b \in \mathbb{Z}$.

Warum geht das?

Wir zeigen, für $a > b$: $\mathrm{ggT}(a,b) = \mathrm{ggT}(a-b,b)$.

Seien $g = \mathrm{ggT}(a,b)$, $g' = \mathrm{ggT}(a-b,b)$.

Dann gilt:

$$\begin{aligned} a &= q_a \cdot g \\ b &= q_b \cdot g \end{aligned} \quad \text{und} \quad \begin{aligned} a-b &= q'_{a-b} \cdot g' \\ b &= q'_b \cdot g' \end{aligned}$$

$$\begin{aligned} a-b &= (q_a - q_b) \cdot g \\ b &= q_b \cdot g \end{aligned} \quad \text{und} \quad \begin{aligned} a &= (q'_{a-b} + q'_b) \cdot g' \\ b &= q'_b \cdot g' \end{aligned}$$

Das heißt g ist Teiler von $a-b, b$ und g' ist Teiler von a, b.

Daraus folgt $g \leq g'$ und $g' \leq g$, also $g = g'$.

Eigenschaften

Ein klassischer Algorithmus erfüllt alle Eigenschaften. Häufig spricht man aber auch von Algorithmen wenn einige dieser Eigenschaften verletzt sind.

(statische) Finitheit. Die Beschreibung des Algorithmus besitzt endliche Länge. (↑nichtuniforme Algorithmen)

(dynamische) Finitheit. Die bei Abarbeitung entstehenden Zwischenergebnisse sind endlich.

Terminiertheit. Algorithmen, die nach endlich vielen Schritten ein Resultat liefern, heißen terminierend. (↑Betriebssysteme, ↑reaktive Systeme)

Determiniertheit. Bei gleichen Eingabedaten gibt ein Algorithmus das gleiche Ergebnis aus. (↑randomisierte Algorithmen, ↑nicht-deterministische Algorithmen)

Determinismus. Der nächste anzuwendende Schritt im Verfahren ist stets eindeutig definiert. (↑randomisierte Algorithmen, ↑nicht-deterministische Algorithmen)

Eigenschaften

Ein klassischer Algorithmus erfüllt alle Eigenschaften. Häufig spricht man aber auch von Algorithmen wenn einige dieser Eigenschaften verletzt sind.

(statische) Finitheit. Die Beschreibung des Algorithmus besitzt endliche Länge. (↑nichtuniforme Algorithmen)

(dynamische) Finitheit. Die bei Abarbeitung entstehenden Zwischenergebnisse sind endlich.

Terminiertheit. Algorithmen, die nach endlich vielen Schritten ein Resultat liefern, heißen terminierend. (↑Betriebssysteme, ↑reaktive Systeme)

Determiniertheit. Bei gleichen Eingabedaten gibt ein Algorithmus das gleiche Ergebnis aus. (↑randomisierte Algorithmen, ↑nicht-deterministische Algorithmen)

Determinismus. Der nächste anzuwendende Schritt im Verfahren ist stets eindeutig definiert. (↑randomisierte Algorithmen, ↑nicht-deterministische Algorithmen)

Eigenschaften

Ein klassischer Algorithmus erfüllt alle Eigenschaften. Häufig spricht man aber auch von Algorithmen wenn einige dieser Eigenschaften verletzt sind.

(statische) Finitheit. Die Beschreibung des Algorithmus besitzt endliche Länge. (↑nichtuniforme Algorithmen)

(dynamische) Finitheit. Die bei Abarbeitung entstehenden Zwischenergebnisse sind endlich.

Terminiertheit. Algorithmen, die nach endlich vielen Schritten ein Resultat liefern, heißen terminierend. (↑Betriebssysteme, ↑reaktive Systeme)

Determiniertheit. Bei gleichen Eingabedaten gibt ein Algorithmus das gleiche Ergebnis aus. (↑randomisierte Algorithmen, ↑nicht-deterministische Algorithmen)

Determinismus. Der nächste anzuwendende Schritt im Verfahren ist stets eindeutig definiert. (↑randomisierte Algorithmen, ↑nicht-deterministische Algorithmen)

Eigenschaften

Ein klassischer Algorithmus erfüllt alle Eigenschaften. Häufig spricht man aber auch von Algorithmen wenn einige dieser Eigenschaften verletzt sind.

(statische) Finitheit. Die Beschreibung des Algorithmus besitzt endliche Länge. (↑nichtuniforme Algorithmen)

(dynamische) Finitheit. Die bei Abarbeitung entstehenden Zwischenergebnisse sind endlich.

Terminiertheit. Algorithmen, die nach endlich vielen Schritten ein Resultat liefern, heißen terminierend. (↑Betriebssysteme, ↑reaktive Systeme)

Determiniertheit. Bei gleichen Eingabedaten gibt ein Algorithmus das gleiche Ergebnis aus. (↑randomisierte Algorithmen, ↑nicht-deterministische Algorithmen)

Determinismus. Der nächste anzuwendende Schritt im Verfahren ist stets eindeutig definiert. (↑randomisierte Algorithmen, ↑nicht-deterministische Algorithmen)

Eigenschaften

Ein klassischer Algorithmus erfüllt alle Eigenschaften. Häufig spricht man aber auch von Algorithmen wenn einige dieser Eigenschaften verletzt sind.

(statische) Finitheit. Die Beschreibung des Algorithmus besitzt endliche Länge. (↑nichtuniforme Algorithmen)

(dynamische) Finitheit. Die bei Abarbeitung entstehenden Zwischenergebnisse sind endlich.

Terminiertheit. Algorithmen, die nach endlich vielen Schritten ein Resultat liefern, heißen terminierend. (↑Betriebssysteme, ↑reaktive Systeme)

Determiniertheit. Bei gleichen Eingabedaten gibt ein Algorithmus das gleiche Ergebnis aus. (↑randomisierte Algorithmen, ↑nicht-deterministische Algorithmen)

Determinismus. Der nächste anzuwendende Schritt im Verfahren ist stets eindeutig definiert. (↑randomisierte Algorithmen, ↑nicht-deterministische Algorithmen)

Analyse von Algorithmen

Entscheidende Fragestellungen:

- Darstellung $\longrightarrow$ Kapitel 2
- Robustheit und Korrektheit $\longrightarrow$ Kapitel 4
- Effizienz und Komplexität $\longrightarrow$ Kapitel 5
- Entwurfstechniken $\longrightarrow$ Kapitel 6

Definition Datenstruktur

Definition Datenstruktur (nach Prof. Eckert)

Eine Datenstruktur ist eine

- **logische Anordnung** von Datenobjekten,
- die **Informationen repräsentieren**,
- den **Zugriff** auf die repräsentierte Information über **Operationen** auf Daten ermöglichen und
- die Information **verwalten**.

Beispiel Datenstruktur

Stapel (oder Englisch: **Stack**), z.B. Pizza-Stapel

Operationen:

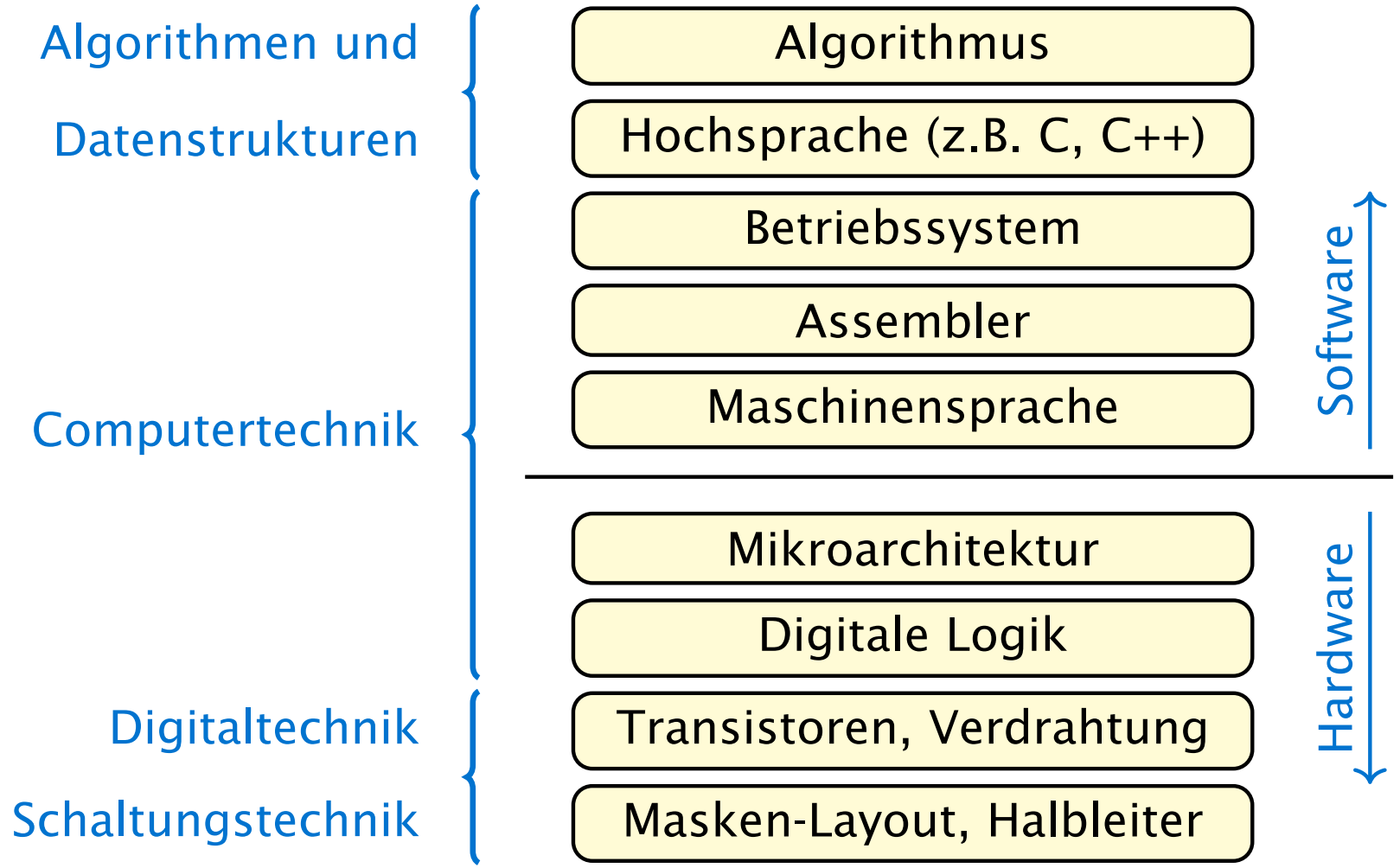
- Element auf Stapel legen – **push**
- Element von Stapel nehmen – **pop**

Operationen jeweils nur auf oberstem Element!

Weitere Beispiele von Datenstrukturen

- **Felder**, **Listen**, **Stack**, **Queue** $\longrightarrow$ Kapitel 3
- **Bäume**, **Graphen** $\longrightarrow$ Kapitel 7, 8, 9

Wie funktioniert ein Computer?



Schema nach Prof. Diepold: Grundlagen der Informatik.

Einordnung Algorithmen und Datenstrukturen

Beispiel-Problem Navigationssystem Auto

Finde kürzesten Weg von Berlin nach München.

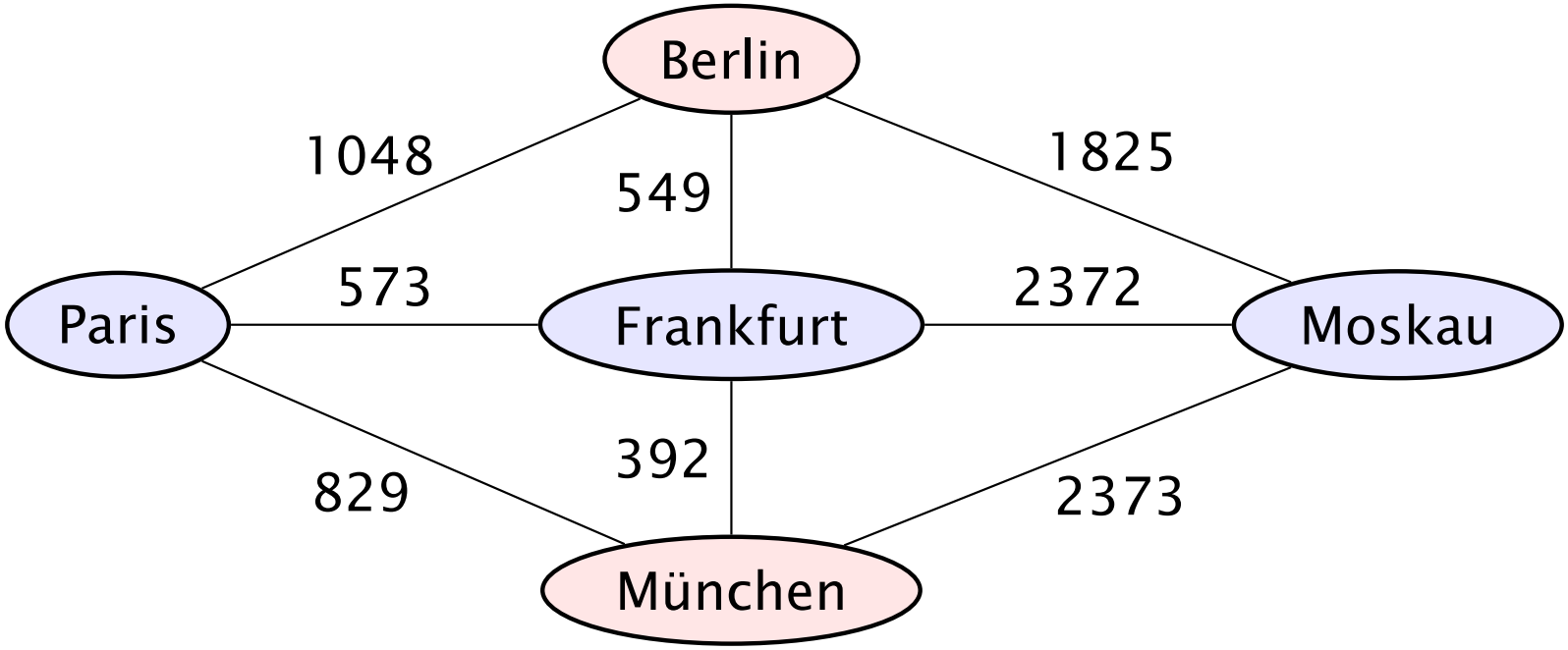
Berlin →  → München

Einordnung Algorithmen und Datenstrukturen

Beispiel-Problem Navigationssystem Auto

Finde kürzesten Weg von Berlin nach München.

- **Datenstruktur:** gewichteter Graph (→ Kapitel 7)

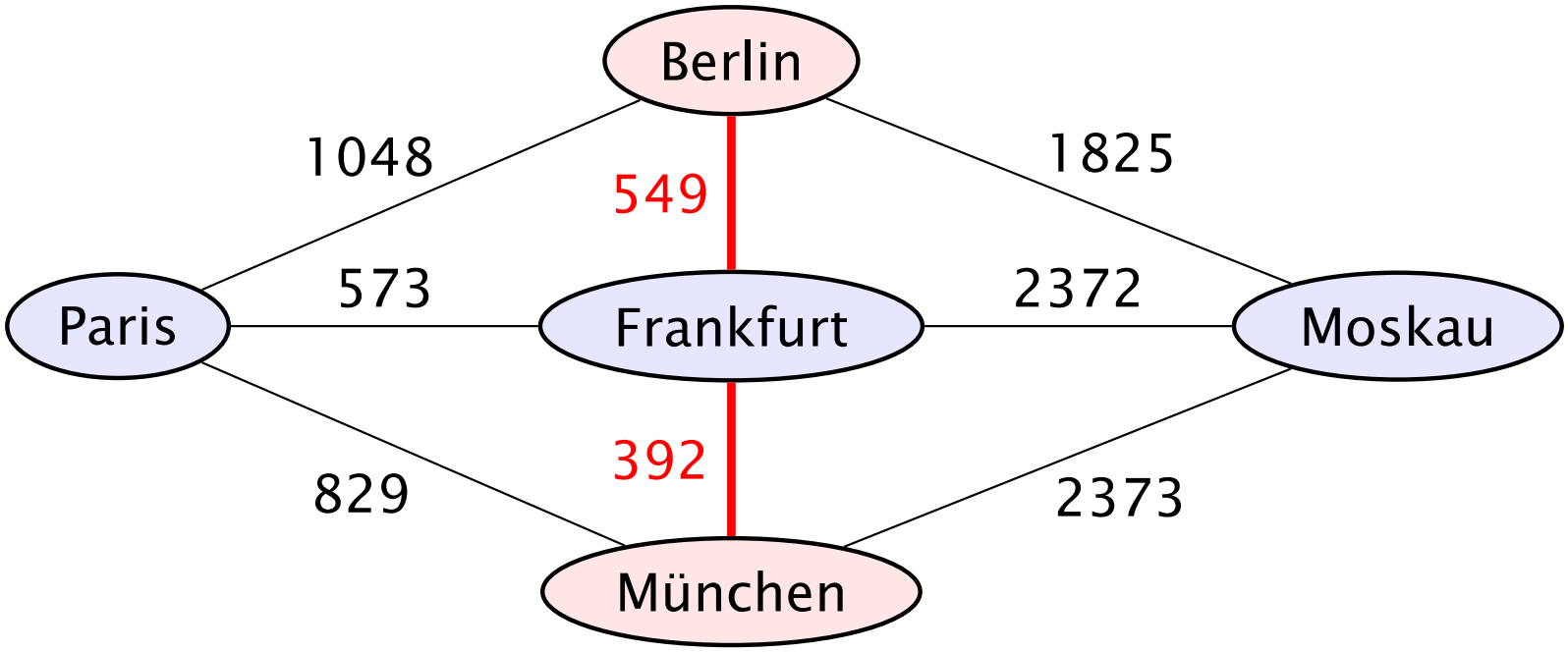


Einordnung Algorithmen und Datenstrukturen

Beispiel-Problem Navigationssystem Auto

Finde kürzesten Weg von Berlin nach München.

- **Datenstruktur:** gewichteter Graph (→ Kapitel 7)
- **Algorithmus:** kürzester Pfad (→ Kapitel 9)



Einordnung Algorithmen und Datenstrukturen

Beispiel-Problem Navigationssystem Auto

Finde kürzesten Weg von Berlin nach München.

- **Datenstruktur:** gewichteter Graph (→ Kapitel 7)
- **Algorithmus:** kürzester Pfad (→ Kapitel 9)
- **Algorithmus-Beschreibung:** Programmiersprache (z.B. C)

Einordnung Algorithmen und Datenstrukturen

Beispiel-Problem Navigationssystem Auto

Finde kürzesten Weg von Berlin nach München.

- **Datenstruktur:** gewichteter Graph (→ Kapitel 7)
- **Algorithmus:** kürzester Pfad (→ Kapitel 9)
- **Algorithmus-Beschreibung:** Programmiersprache (z.B. C)
- **Übersetzung in Maschinensprache:** Compiler (z.B. GCC)

Einordnung Algorithmen und Datenstrukturen

Beispiel-Problem Navigationssystem Auto

Finde kürzesten Weg von Berlin nach München.

- **Datenstruktur:** gewichteter Graph (→ Kapitel 7)
- **Algorithmus:** kürzester Pfad (→ Kapitel 9)
- **Algorithmus-Beschreibung:** Programmiersprache (z.B. C)
- **Übersetzung in Maschinensprache:** Compiler (z.B. GCC)
- **Aufruf des Programms:** Betriebssystem (z.B. Linux)

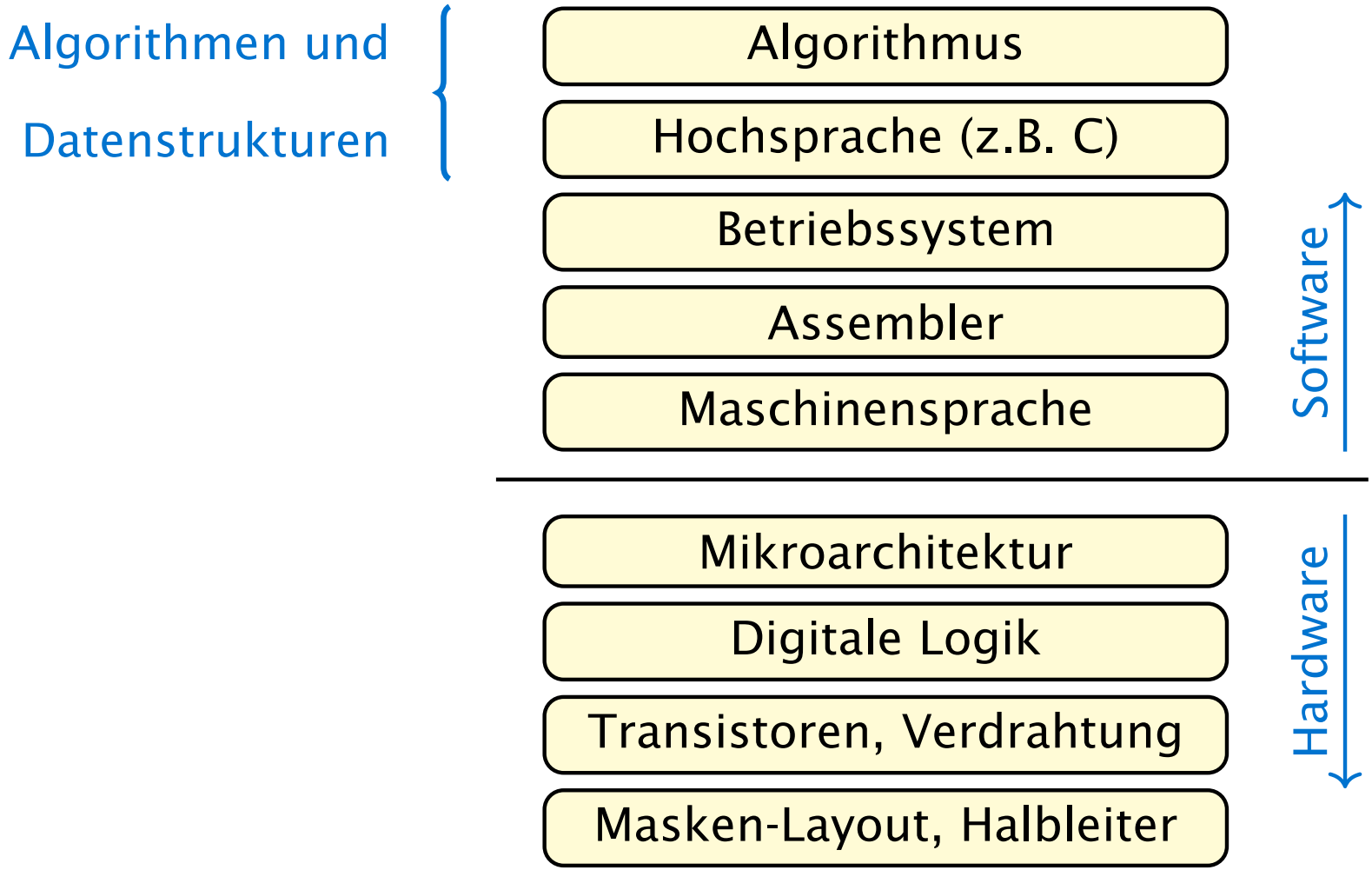
Einordnung Algorithmen und Datenstrukturen

Beispiel-Problem Navigationssystem Auto

Finde kürzesten Weg von Berlin nach München.

- **Datenstruktur:** gewichteter Graph (→ Kapitel 7)
- **Algorithmus:** kürzester Pfad (→ Kapitel 9)
- **Algorithmus-Beschreibung:** Programmiersprache (z.B. C)
- **Übersetzung in Maschinensprache:** Compiler (z.B. GCC)
- **Aufruf des Programms:** Betriebssystem (z.B. Linux)
- **Ausführung des Programms:** Computer (z.B. Laptop)

Einordnung Algorithmen und Datenstrukturen



Schema nach Prof. Diepold: Grundlagen der Informatik.

Wie beschreibt man Algorithmen?

Algorithmus: bestimme Maximum von zwei Zahlen

- Input: Zahlen a, b
- Output: Zahl $x = \max(a, b)$

Problem: präzise Beschreibung der Schritte

Wie beschreibt man Algorithmen?

Algorithmus: bestimme Maximum von zwei Zahlen

- Input: Zahlen a, b
- Output: Zahl $x = \max(a, b)$

Problem: präzise Beschreibung der Schritte

Lösung: Pseudocode

```
Algorithmus: max(a,b)
Input:  a,b
   x=a
   Falls b>a dann
     x=b
   Ende Falls
Output:  x
```

Darstellung von Algorithmen I

Pseudocode

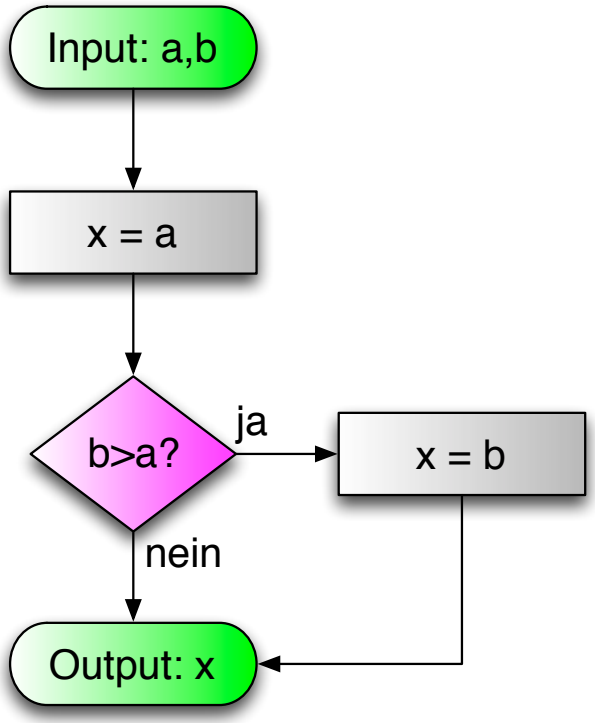
- informelle Veranschaulichung von Algorithmus
- nicht von Rechner ausführbar
- nicht standardisiert

```
Algorithmus: max(a,b)
Input:  a,b
   x=a
   Falls b>a dann
     x=b
   Ende Falls
Output:  x
```

Darstellung von Algorithmen II

Flussdiagramm

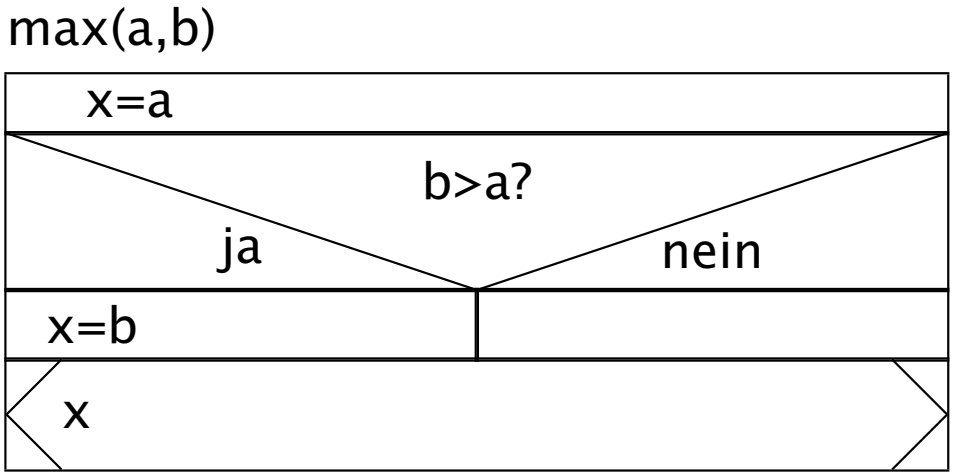
- graphische Darstellung als Ablaufdiagramm, nicht ausführbar
- normiert als DIN 66001



Darstellung von Algorithmen III

Struktogramm

- Diagramm zur Strukturdarstellung, nicht ausführbar
- eingeführt von Nassi/Shneiderman 1973, normiert als DIN 66261



Darstellung von Algorithmen IV

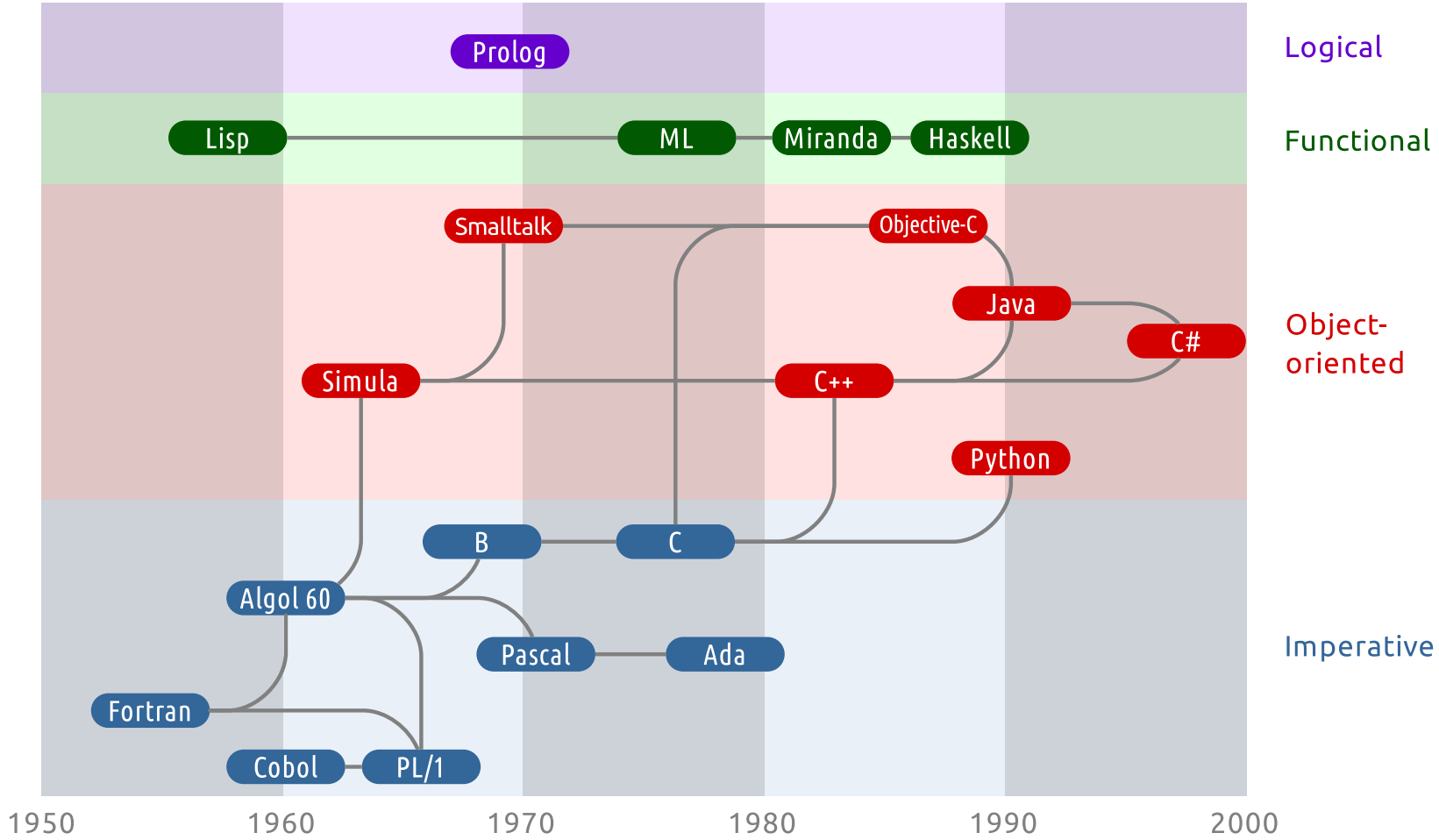
Programmiersprache

- formale Sprache zur Beschreibung von Algorithmen
- fest definierte Syntax
- ein Compiler/Interpreter wandelt Programm in ausführbare Form für Rechner um
- Beispiele: Assembler, C, Java

- Algorithmus in C:

```
1 int max(int a, int b) {
2     int x = a;
3     if (b > a)
4         x = b;
5     return x;
6 }
```

Programmiersprachen Übersicht



Grafik von Alexandru Duliu.

Äquivalenz von Algorithmen-Beschreibungen

Churchsche These

Alle „vernünftigen“ Definitionen von Algorithmen sind äquivalent.

Die fehlende Zutat für eine präzise Algorithmendefinition ist die Definition eines *elementaren Einzelschrittes*. Dies wird üblicherweise durch ein Maschinenmodells gemacht (z.B. Turingmaschine). Es gibt auch sehr esoterische Maschinenmodelle (z.B. *billiard ball computer*).

- alle gängigen Programmiersprachen leisten dasselbe
- jeder Computer ist äquivalent

Die These postuliert, dass alle diese Maschinenmodelle equivalent ist.

- formal: berechenbare Funktionen, formale Sprachen, Automaten, Turing-Maschinen

↑theoretische Informatik

Äquivalenz von Algorithmen-Beschreibungen

Churchsche These

Alle „vernünftigen“ Definitionen von Algorithmen sind äquivalent.

Die fehlende Zutat für eine präzise Algorithmendefinition ist die Definition eines *elementaren Einzelschrittes*. Dies wird üblicherweise durch ein Maschinenmodells gemacht (z.B. Turingmaschine). Es gibt auch sehr esoterische Maschinenmodelle (z.B. *billiard ball computer*).

- alle gängigen Programmiersprachen leisten dasselbe
- jeder Computer ist äquivalent

Die These postuliert, dass alle diese Maschinenmodelle equivalent ist.

- formal: berechenbare Funktionen, formale Sprachen, Automaten, Turing-Maschinen

↑theoretische Informatik

Bausteine von Algorithmen

Elementare Bausteine

„Normale“ Algorithmen lassen sich mit

vier elementaren Bausteinen

darstellen:

1. Elementarer Verarbeitungsschritt (z.B. Zuweisung an Variable)
2. Sequenz (elementare Schritte nacheinander)
3. Bedingter Verarbeitungsschritt (z.B. if/else)
4. Wiederholung (z.B. while-Schleife)

1. Elementarer Verarbeitungsschritt

Beispiele

- `a = a - b // weist Variable a den Wert a-b zu`
- `return a // liefert den Wert von a zurueck`

Achtung: manche Verarbeitungsschritte sehen elementar aus, sind es aber nicht!

- `sortiere Liste L // nicht elementar`
- `finde kuerzesten Pfad in G // nicht elementar`

2. Sequenz

Sequenz ist eine Aneinanderreihung von elementaren Verarbeitungsschritten

Abgrenzung der Schritte mittels **Semikolon (;)**

Beispiel

- `x = 5; // Zuweisung von Wert 5 an Variable x`
- `x = x + 2; // Wert von x  ist nun 7`

Um Ausnahmen zu vermeiden, wird Semikolon auch verwendet, wenn kein weiterer Schritt folgt

3. Bedingter Verarbeitungsschritt

Ausführung des Verarbeitungsschrittes nur wenn Bedingung erfüllt ist

Beispiele:

- ```
  if (a > b) // Bedingung wird in Klammern notiert
      a = a - b;
  ```
- ```
  if (a > b)
      a = a - b;
  else // falls Bedingung nicht erfuellt
      b = b - a;
  ```

Einrückung verdeutlicht logische Ebenen

3. Bedingter Verarbeitungsschritt

falls mehr als ein Verarbeitungsschritt bedingt ausgeführt werden soll, Markierung durch einen Block **{ ... }** mit geschweiften Klammern

Beispiel

```
if (x == 0) {
    x = 5;
    x = x + 2;
} // if Block ist hier zu Ende
else {
    x = x - 1;
} // else Block ist hier zu Ende
```

auch einzelne Schritte können in einen Block gefasst werden

4. Wiederholung

wiederholte Ausführung von Verarbeitungsschritt/Block solange Bedingung erfüllt ist (auch while Schleife genannt)

Beispiele

- ```
  while (x != 0) // Bedingung in Klammern
      x = x - 1;
  ```
- ```
  while (b > 0) { // Block fuer mehrere Schritte
      if (a > b)
          a = a - b;
      else
          b = b - a;
  } // while Block ist hier zu Ende
  ```

4. Wiederholung

Es gibt auch andere Schleifentypen: do-while Schleife:

```
do {
    x = x - 1;
} while (x != 0); // Vorsicht by floats!!!
```

for-Schleife:

```
for i=1 to 10
    print(i); // gibt Wert von i aus
```

Achtung, Syntax der for-Schleife ist in C komplexer!

```
for (i=1; i <= 10; i++) // echte C Syntax
    print(i);
```

Beispiel: Euklidscher Algorithmus

- Einrücken oder geschweifte Klammern {, } kennzeichnen **Blockstruktur**

```
1  euklid(a,b)
2      if (a == 0)
3          return b;
4      while (b > 0) {
5          if (a > b)
6              a = a - b;
7          else
8              b = b - a;
9      }
10     return a;
```

Euklidscher Algorithmus

Beispiel: Euklidscher Algorithmus

- Einrücken oder geschweifte Klammern {, } kennzeichnen **Blockstruktur**
- In C so nicht möglich

```
1 euklid(a,b)
2     if (a == 0)
3         return b;
4     while (b > 0)
5         if (a > b)
6             a = a - b;
7         else
8             b = b - a;
9     return a;
```

Euklidscher Algorithmus

Euklidscher Algorithmus

Input: Natürliche Zahlen a, b

Output: $\mathrm{ggT}(a, b)$

1. Falls $a = 0$ liefere b zurück
2. Solange $b > 0$ wiederhole

   Falls $a > b$ setze $a = a - b$

   sonst setze $b = b - a$
3. Liefere a zurück

Euklidscher Algorithmus

Input: Natürliche Zahlen a, b

Output: $\mathrm{ggT}(a, b)$

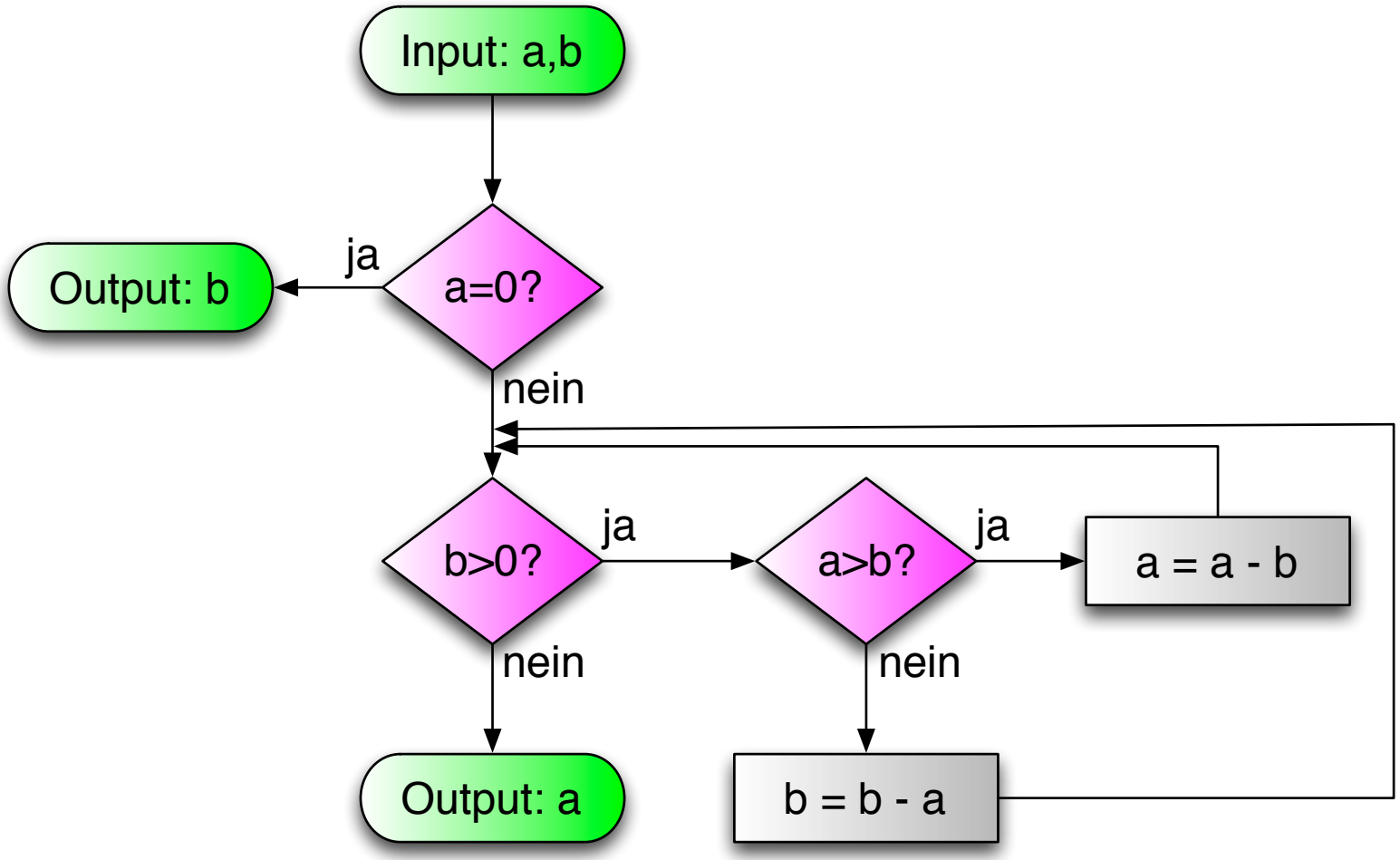
1. Falls $a = 0$ liefere b zurück
2. Solange $b > 0$ wiederhole

   Falls $a > b$ setze $a = a - b$

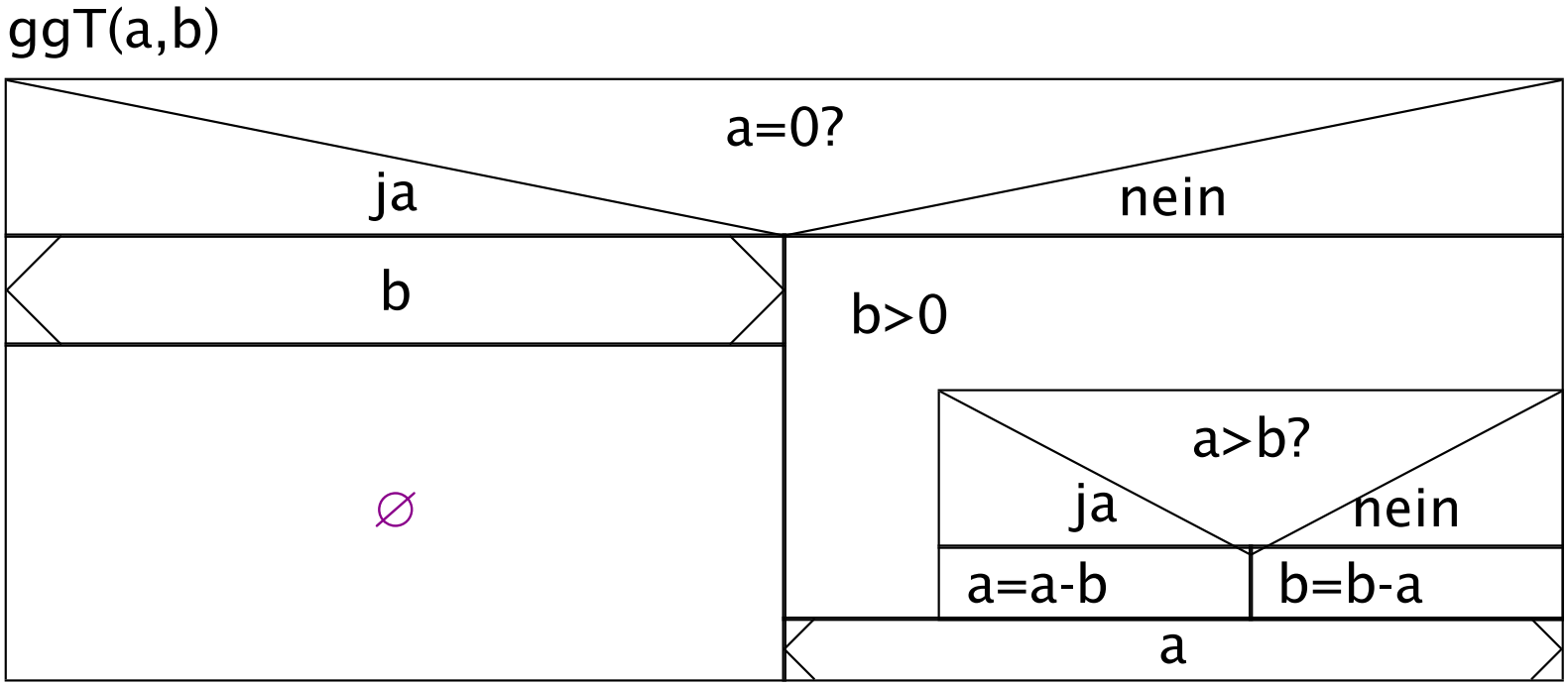
   sonst setze $b = b - a$
3. Liefere a zurück

⟶ das ist Pseudocode!

Euklidscher Algorithmus als Flussdiagramm



Euklidscher Algorithmus als Struktogramm



Euklidscher Algorithmus als Pseudocode

```
1  Input: natuerliche Zahlen a, b
2  Output: ggT(a,b)
3  euklid(a,b)
4      if (a == 0)
5          return b;
6      while (b > 0) { // Hauptschleife
7          if (a > b)
8              a = a - b;
9          else
10             b = b - a;
11     }
12     return a;
```

Euklidscher Algorithmus

Euklidscher Algorithmus als C

```
1  int ggT(int a, int b)
2  {
3      if (a==0)
4          return b;
5      while (b>0) {
6          if (a>b)
7              a = a - b;
8          else
9              b = b - a;
10     }
11     return a;
12 }
```

Euklidscher Algorithmus als Python

```
1 def ggT(a, b):
2     if a == 0:
3         return b
4     while b > 0:
5         if a > b:
6             a = a - b
7         else:
8             b = b - a
9     return a
```

Darstellung von Algorithmen in der Vorlesung

viele Möglichkeiten der Darstellung!

- alle vernünftigen Darstellungen sind äquivalent
- jede Darstellung hat Vor- und Nachteile

für die Vorlesung: **Pseudocode im C Stil**

Zusatzmaterial für viele Beispiele aus der Vorlesung:

- http://www.brpreiss.com/books/opus7/
- Beispiele in:
  - Python
  - C++
  - Java
  - C#
  - und vieles mehr...

Beispiel: Fibonacci Zahlen

Fibonacci Folge

Die **Fibonacci Folge** ist eine Folge natürlicher Zahlen $f_1, f_2, f_3, \ldots$, für die gilt

$$f_n = f_{n-1} + f_{n-2} \qquad \text{für } n \geq 3$$

mit Anfangswerten $f_1 = 1$, $f_2 = 1$.

- eingesetzt von Leonardo Fibonacci zur Beschreibung von Wachstum einer Kaninchenpopulation
- Folge lautet: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
- berechenbar z.B. via Rekursion

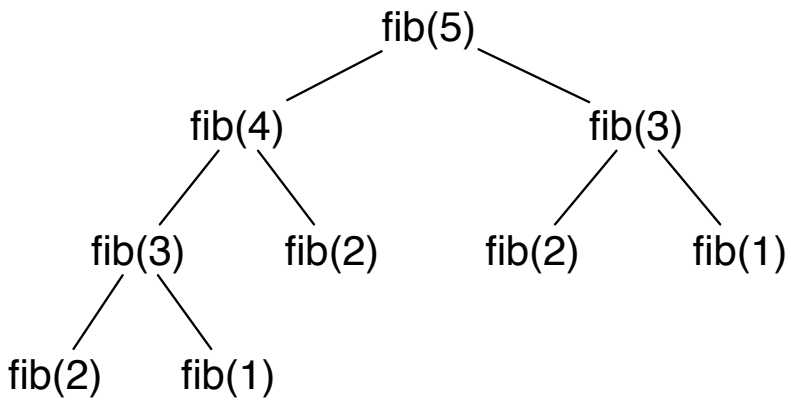
Beispiel: Fibonacci Funktion

```
Input: Index n der Fibonaccifolge
Output: Wert f_n
fib(n)
   if (n == 1 || n == 2) {
      return 1;
   }
   else {
      // rekursiver Aufruf
      return fib(n-1) + fib(n-2);
   }
```

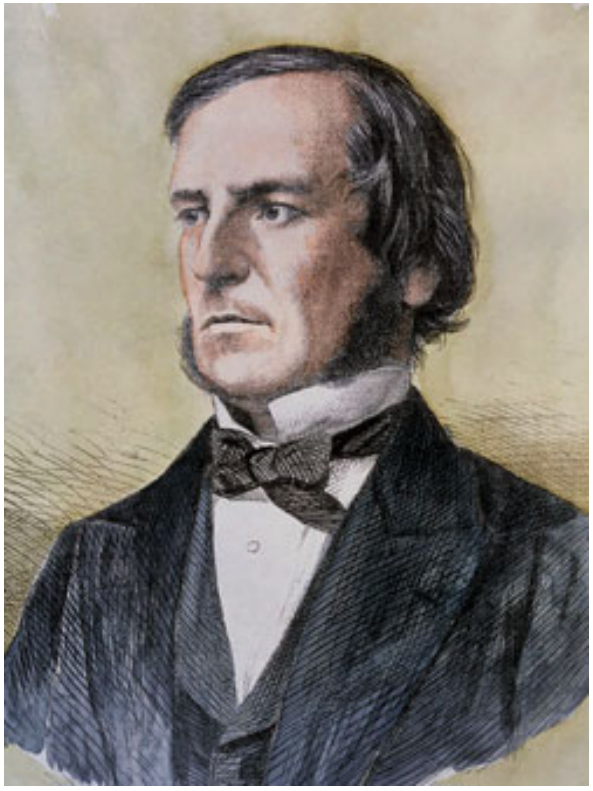
Beispiel: Fibonacci Funktion

```
Input: Index n der Fibonaccifolge
Output: Wert f_n
fib(n)
   if (n == 1 || n == 2) {
      return 1;
   }
   else {
      // rekursiver Aufruf
      return fib(n-1) + fib(n-2);
   }
```

Aufrufstruktur für fib(5):



George Boole



Englischer Mathematiker (1815-1864)

Logische Werte

Boolesche Logik: Logik mit zwei Werten

Repräsentationen:

- 1 und 0
- W und F (in Englisch: T und F)
- L und O

Mengensymbol $\mathbb{B}$

- $\mathbb{B} = \{0, 1\} = \{F, W\} = \{O, L\}$

Logische Werte und Verknüpfungen

„Grundrechenarten“ mit logischen Werten:

Konjunktion: $\wedge : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- ähnlich zu Multiplikation bei Zahlen
- auch bezeichnet als UND bzw. AND

Disjunktion: $\vee : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- ähnlich zu Addition bei Zahlen
- auch bezeichnet als ODER bzw. OR

Negation: $\neg : \mathbb{B} \to \mathbb{B}$

- auch bezeichnet als NICHT bzw. NOT

Logische Werte und Verknüpfungen

„Grundrechenarten“ mit logischen Werten:

Konjunktion: $\wedge : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- ähnlich zu Multiplikation bei Zahlen
- auch bezeichnet als UND bzw. AND

Disjunktion: $\vee : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- ähnlich zu Addition bei Zahlen
- auch bezeichnet als ODER bzw. OR

Negation: $\neg : \mathbb{B} \to \mathbb{B}$

- auch bezeichnet als NICHT bzw. NOT

Wahrheitstabelle:

a	b	$a \wedge b$
0	0	0
0	1	0
1	0	0
1	1	1

Logische Werte und Verknüpfungen

„Grundrechenarten“ mit logischen Werten:

Konjunktion: $\wedge : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- ähnlich zu Multiplikation bei Zahlen
- auch bezeichnet als UND bzw. AND

Disjunktion: $\vee : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- ähnlich zu Addition bei Zahlen
- auch bezeichnet als ODER bzw. OR

Negation: $\neg : \mathbb{B} \to \mathbb{B}$

- auch bezeichnet als NICHT bzw. NOT

Wahrheitstabelle:

a	b	$a \wedge b$
0	0	0
0	1	0
1	0	0
1	1	1

a	b	$a \vee b$
0	0	0
0	1	1
1	0	1
1	1	1

Logische Werte und Verknüpfungen

„Grundrechenarten“ mit logischen Werten:

Konjunktion: $\wedge : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- ähnlich zu Multiplikation bei Zahlen
- auch bezeichnet als UND bzw. AND

Disjunktion: $\vee : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- ähnlich zu Addition bei Zahlen
- auch bezeichnet als ODER bzw. OR

Negation: $\neg : \mathbb{B} \to \mathbb{B}$

- auch bezeichnet als NICHT bzw. NOT

Wahrheitstabelle:

a	b	$a \wedge b$
0	0	0
0	1	0
1	0	0
1	1	1

a	b	$a \vee b$
0	0	0
0	1	1
1	0	1
1	1	1

a	$\neg a$
0	1
1	0

Weitere Verknüpfungen I

NAND: $\uparrow\colon\ \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- $a \uparrow b = \neg(a \wedge b)$
- mit NAND lassen sich NOT, OR, AND erzeugen

NOR: $\downarrow\colon\ \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- $a \downarrow b = \neg(a \vee b)$
- mit NOR lassen sich ebenso NOT, OR, AND erzeugen

XOR: $\oplus\colon\ \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- auch exklusiv oder genannt
- erzeugbar aus $\neg(a \wedge b) \wedge (a \vee b)$ (siehe Übung)

Wahrheitstabelle:

a	b	$a \uparrow b$
0	0	1
0	1	1
1	0	1
1	1	0

Weitere Verknüpfungen I

NAND: $\uparrow\colon\ \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- $a \uparrow b = \neg(a \wedge b)$
- mit NAND lassen sich NOT, OR, AND erzeugen

NOR: $\downarrow\colon\ \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- $a \downarrow b = \neg(a \vee b)$
- mit NOR lassen sich ebenso NOT, OR, AND erzeugen

XOR: $\oplus\colon\ \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- auch exklusiv oder genannt
- erzeugbar aus $\neg(a \wedge b) \wedge (a \vee b)$ (siehe Übung)

Wahrheitstabelle:

a	b	$a \uparrow b$
0	0	1
0	1	1
1	0	1
1	1	0

a	b	$a \downarrow b$
0	0	1
0	1	0
1	0	0
1	1	0

Weitere Verknüpfungen I

NAND: $\uparrow\colon\ \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- $a \uparrow b = \neg(a \wedge b)$
- mit NAND lassen sich NOT, OR, AND erzeugen

NOR: $\downarrow\colon\ \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- $a \downarrow b = \neg(a \vee b)$
- mit NOR lassen sich ebenso NOT, OR, AND erzeugen

XOR: $\oplus\colon\ \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- auch exklusiv oder genannt
- erzeugbar aus $\neg(a \wedge b) \wedge (a \vee b)$ (siehe Übung)

Wahrheitstabelle:

a	b	$a \uparrow b$
0	0	1
0	1	1
1	0	1
1	1	0

a	b	$a \downarrow b$
0	0	1
0	1	0
1	0	0
1	1	0

a	b	$a \oplus b$
0	0	0
0	1	1
1	0	1
1	1	0

Weitere Verknüpfungen II

Implikation: $\Rightarrow: \ \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$

- oft verwendet für mathematische Sätze:
  „a impliziert b“, „aus a folgt b“
- Beispiel: „aus $n < 3$ folgt $n < 5$“
- erzeugbar aus $\neg a \vee b$

Wahrheitstabelle:

a	b	$a \Rightarrow b$
0	0	1
0	1	1
1	0	0
1	1	1

Äquivalenz: $\Leftrightarrow: \ \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$

- oft verwendet für mathematische Sätze:
  „a gilt genau dann, wenn b gilt“,
  „a und b sind äquivalent“
- Beispiel: „f ist bijektiv genau dann,
  wenn f injektiv und surjektiv ist“
- erzeugbar aus $(a \wedge b) \vee (\neg a \wedge \neg b)$

Weitere Verknüpfungen II

Implikation: $\Rightarrow: \ \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- oft verwendet für mathematische Sätze: „a impliziert b“, „aus a folgt b“
- Beispiel: „aus $n < 3$ folgt $n < 5$“
- erzeugbar aus $\neg a \vee b$

Wahrheitstabelle:

a	b	$a \Rightarrow b$
0	0	1
0	1	1
1	0	0
1	1	1

Äquivalenz: $\Leftrightarrow: \ \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- oft verwendet für mathematische Sätze: „a gilt genau dann, wenn b gilt“, „a und b sind äquivalent“
- Beispiel: „f ist bijektiv genau dann, wenn f injektiv und surjektiv ist“
- erzeugbar aus $(a \wedge b) \vee (\neg a \wedge \neg b)$

a	b	$a \Leftrightarrow b$
0	0	1
0	1	0
1	0	0
1	1	1

Rangfolge und Rechenregeln

Rangfolge:

- NICHT vor UND
- UND vor ODER

Beispiel

$$\neg 0 \vee 1 \wedge 0 = (\neg 0) \vee (1 \wedge 0) = 1 \vee 0 = 1$$

De Morgan-Gesetze:

- $\neg(a \wedge b) = \neg a \vee \neg b$
- $\neg(a \vee b) = \neg a \wedge \neg b$

Logische Ausdrücke in Pseudocode und C

- logische Variablen: `bool a,b;`
- logische Werte: `true` und `false`
- NOT Operator: `!a`
- AND Operator: `a && b`
- OR Operator: `a || b`

Beispiele:

- `( (2 == 2) && (3 < 1) )`
  ergibt `(true && false)`, also `false`
- `( !(2 == 2) || (3 > 1) )`
  ergibt `(false || true)`, also `true`
- Kurzform für `!(2 == 2)` ist `(2 != 2)`

Logische Ausdrücke in Pseudocode und C

- logische Variablen: `bool a,b;`
- logische Werte: `true` und `false`
- NOT Operator: `!a`
- AND Operator: `a && b`
- OR Operator: `a || b`

Beispiele:

- `( (2 == 2) && (3 < 1) )`
  ergibt `(true && false)`, also `false`
- `( !(2 == 2) || (3 > 1) )`
  ergibt `(false || true)`, also `true`
- Kurzform für `!(2 == 2)` ist `(2 != 2)`

Logische Ausdrücke in Pseudocode und C

- logische Variablen: `bool a,b;`
- logische Werte: `true` und `false`
- NOT Operator: `!a`
- AND Operator: `a && b`
- OR Operator: `a || b`

Beispiele:

- `( (2 == 2) && (3 < 1) )`
  ergibt `(true && false)`, also `false`
- `( !(2 == 2) || (3 > 1) )`
  ergibt `(false || true)`, also `true`
- Kurzform für `!(2 == 2)` ist `(2 != 2)`

Logische Ausdrücke in Pseudocode und C

- logische Variablen: `bool a,b;`
- logische Werte: `true` und `false`
- NOT Operator: `!a`
- AND Operator: `a && b`
- OR Operator: `a || b`

Beispiele:

- `( (2 == 2) && (3 < 1) )`
  ergibt `(true && false)`, also `false`
- `( !(2 == 2) || (3 > 1) )`
  ergibt `(false || true)`, also `true`
- Kurzform für `!(2 == 2)` ist `(2 != 2)`

Was sind primitive Datentypen?

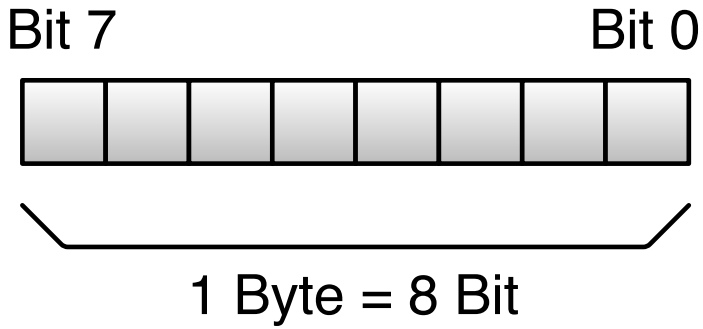
Primitive Datentypen
Wir bezeichnen grundlegende, in Programmiersprachen eingebaute Datentypen als primitive Datentypen.

Durch Kombination von primitiven Datentypen lassen sich zusammengesetzte Datentypen bilden.

Beispiele für primitive Datentypen in C:

- `int` für ganze Zahlen
- `float` für floating point Zahlen
- `bool` für logische Werte

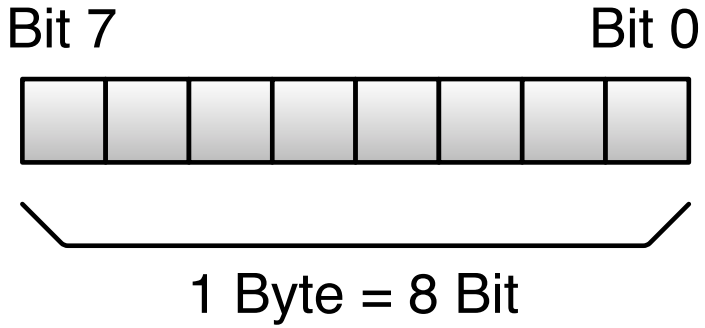
Bits und Bytes



Bytes als Maßeinheit für Speichergrössen (nach IEC, traditionell):

- 2^{10} Bytes $= 1024$ Bytes $= 1$ KiB, ein Kilo Byte (Kibi Byte)
- 2^{20} Bytes $= 1$ MiB, ein Mega Byte (bzw. MebiByte)
- 2^{30} Bytes $= 1$ GiB, ein Giga Byte (bzw. GibiByte)
- 2^{40} Bytes $= 1$ TiB, ein Tera Byte (bzw. TebiByte)
- 2^{50} Bytes $= 1$ PiB, ein Peta Byte (bzw. PebiByte)
- 2^{60} Bytes $= 1$ EiB, ein Exa Byte (bzw. ExbiByte)

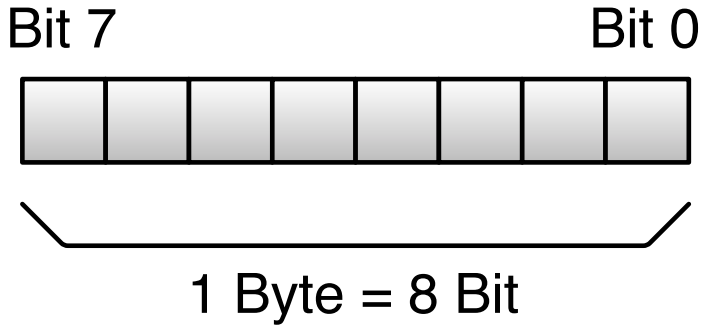
Bits und Bytes



Bytes als Maßeinheit für Speichergrössen (nach IEC, metrisch):

- 10^3 Bytes $= 1000$ Bytes $= 1$ kB, ein kilo Byte (großes B)
- 10^6 Bytes $= 1$ MB, ein Mega Byte
- 10^9 Bytes $= 1$ GB, ein Giga Byte
- 10^{12} Bytes $= 1$ TB, ein Tera Byte
- 10^{15} Bytes $= 1$ PB, ein Peta Byte
- 10^{18} Bytes $= 1$ EB, ein Exa Byte

Bits und Bytes



Bytes als Maßeinheit für Speichergrössen (nach IEC, metrisch):

- 10^3 Bytes $= 1000$ Bytes $= 1$ kB, ein kilo Byte (großes B)
- 10^6 Bytes $= 1$ MB, ein Mega Byte
- 10^9 Bytes $= 1$ GB, ein Giga Byte
- 10^{12} Bytes $= 1$ TB, ein Tera Byte
- 10^{15} Bytes $= 1$ PB, ein Peta Byte
- 10^{18} Bytes $= 1$ EB, ein Exa Byte

Hinweis: auch Bits werden als Maßangabe verwendet, z.B. 16 Mbit oder 16 Mb (kleines b).

Primitive Datentypen in C-ähnlichen Sprachen

Wir betrachten im Detail primitive Datentypen für:

1. natürliche Zahlen *(unsigned integers)*
2. ganze Zahlen *(signed integers)*
3. floating point Zahlen *(floats)*

Zahldarstellung

Dezimalsystem:

- Basis $b = 10$
- Koeffizienten $c_n \in \{0,1,2,3,4,5,6,7,8,9\}$
- Beispiel: $123_{10} = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0$

Binärsystem:

- Basis $b = 2$
- Koeffizienten $c_n \in \{0,1\}$
- Beispiel: $1101_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 13_{10}$

Zahldarstellung

Dezimalsystem:

- Basis $b = 10$
- Koeffizienten $c_n \in \{0,1,2,3,4,5,6,7,8,9\}$
- Beispiel: $123_{10} = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0$

Binärsystem:

- Basis $b = 2$
- Koeffizienten $c_n \in \{0,1\}$
- Beispiel: $1101_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 13_{10}$

Zahldarstellung

Oktalsystem:

- Basis $b = 8\ (= 2^3)$
- Koeffizienten $c_n \in \{0,1,2,3,4,5,6,7\}$
- Beispiel: $173_8 = 1 \cdot 8^2 + 7 \cdot 8^1 + 3 \cdot 8^0 = 123_{10}$

Hexadezimalsystem:

- Basis $b = 16\ (= 2^4)$
- Koeffizienten $c_n \in \{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}$
- Beispiel: $7B_{16} = 7 \cdot 16^1 + B \cdot 16^0 = 123_{10}$

Wie viele Ziffern pro Zahl?

Problem

Gegeben Zahl $z \in \mathbb{N}$, wie viele Ziffern m werden bezüglich Basis b benötigt?

Lösung: $m = \lfloor \log_b(z) \rfloor + 1$

Erläuterung: ($a \in \mathbb{R}$)

- $\lfloor a \rfloor = \text{floor}(a) =$ größte ganze Zahl kleiner gleich a
- $\lceil a \rceil = \text{ceil}(a) =$ kleinste ganze Zahl größer gleich a

$$a - 1 < \lfloor a \rfloor \leq a \leq \lceil a \rceil < a + 1$$

- $\log_b(z) = \frac{\ln(z)}{\ln(b)}$, wobei „ln“ der natürliche Logarithmus ist

Wie viele Ziffern pro Zahl?

Lösung: $m = \lfloor \log_x(z) \rfloor + 1$

Beispiele: $z = 123$

- Basis $b = 10$: $m = \lfloor \log_{10}(123) \rfloor + 1 = \lfloor 2.0899\ldots \rfloor + 1 = 3$
- Basis $b = 2$: $m = \lfloor \log_2(123) \rfloor + 1 = \lfloor 6.9425\ldots \rfloor + 1 = 7$
- Basis $b = 8$: $m = \lfloor \log_8(123) \rfloor + 1 = \lfloor 2.3141\ldots \rfloor + 1 = 3$
- Basis $b = 16$: $m = \lfloor \log_{16}(123) \rfloor + 1 = \lfloor 1.7356\ldots \rfloor + 1 = 2$

Größte Zahl pro Anzahl Ziffern?

Problem

Gegeben Basis b und m Ziffern, was ist die größte darstellbare Zahl?

Lösung: $z_{max} = x^m - 1$

Beispiele:

- $b = 2, m = 4$: $z_{max} = 2^4 - 1 = 15 = 1111_2$
- $b = 2, m = 8$: $z_{max} = 2^8 - 1 = 255 = 11111111_2$
- $b = 16, m = 2$: $z_{max} = 16^2 - 1 = 255 = FF_{16}$

Natürliche Zahlen in C-ähnlichen Sprachen

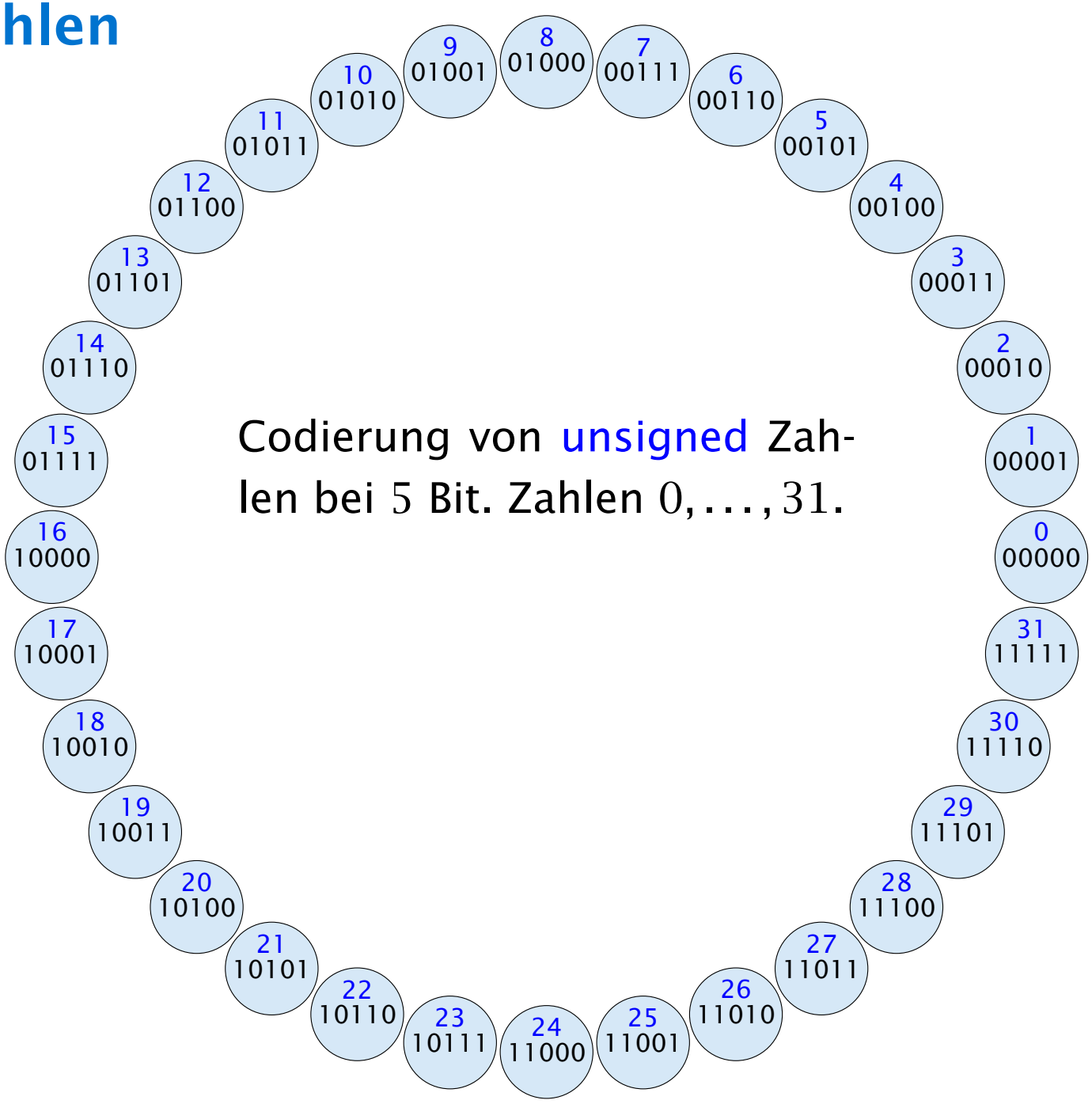
Natürliche Zahlen

In Computern verwendet man Binärdarstellung mit einer fixen Anzahl Ziffern (genannt Bits).

Die primitiven Datentypen für natürliche Zahlen sind:

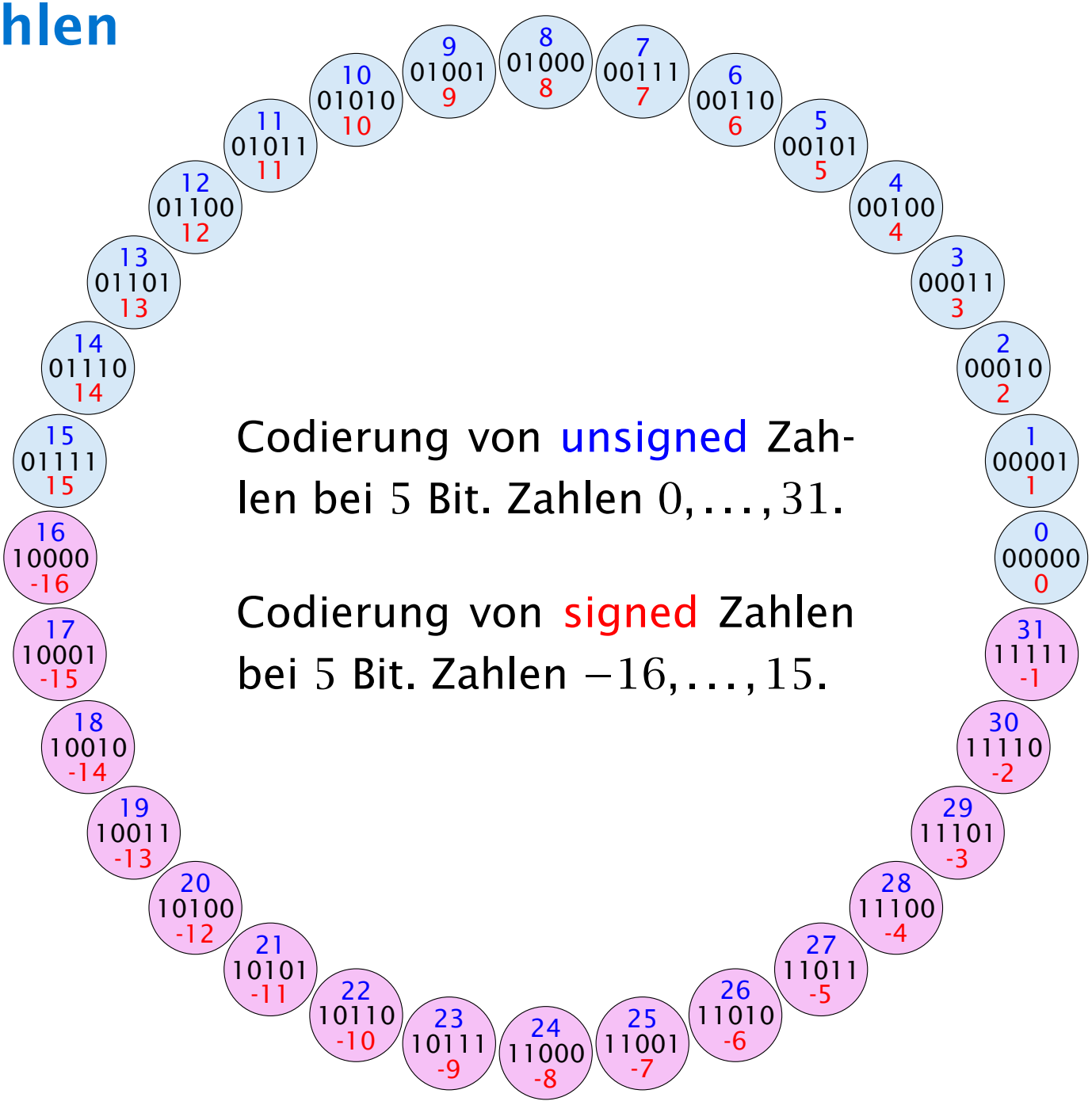
- **8 Bits** (ein Byte), darstellbare Zahlen: $\{0,\ldots,255\}$
  in C: `unsigned char`
- **16 Bits**, darstellbare Zahlen: $\{0,\ldots,65\,535\}$
  in C: `unsigned short`
- **32 Bits**, darstellbare Zahlen: $\{0,\ldots,4\,294\,967\,295\}$
  in C: `unsigned long`
- **64 Bits**, darstellbare Zahlen: $\{0,\ldots,2^{64}-1\}$
  in C: `unsigned long long`

Negative Zahlen



Codierung von unsigned Zahlen bei 5 Bit. Zahlen $0, \ldots, 31$.

Negative Zahlen



Codierung von unsigned Zahlen bei 5 Bit. Zahlen $0, \ldots, 31$.

Codierung von signed Zahlen bei 5 Bit. Zahlen $-16, \ldots, 15$.

Negative Zahlen

Bitfolge		Zahl
$x = \langle x_{n-1}, \ldots, x_0 \rangle$	$\underset{f^{-1}}{\overset{f}{\rightleftarrows}}$	$\sum_{i=0}^{n-1} x_i 2^i$
$x = \langle x_{n-1}, \ldots, x_0 \rangle$	$\underset{f_{ZK}^{-1}}{\overset{f_{ZK}}{\rightleftarrows}}$	$-x_{n-1} 2^n + \sum_{i=0}^{n-1} x_i 2^i$

Negative Zahlen

Definition

In 2-Komplement Darstellung mit n bits repräsentiert die Bitfolge

$$x = \langle x_{n-1}, x_{n-2}, \ldots, x_1, x_0 \rangle$$

die Zahl $f_{ZK}(x) = -x_{n-1}2^n + \sum_{i=0}^{n-1} x_i 2^i$.

Beobachtungen

- Zahlen mit $x_{n-1} = 1$ sind negativ; andere positiv (Vorzeichenbit)
- positive Zahlen: $0, \ldots, 2^{n-1} - 1$
  negative Zahlen: $-1, \ldots, -2^{n-1}$
- $f_{ZK}(x) = -x_{n-1}2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i$.

Negative Zahlen

Beachte, dass die Regel für die Zahl -2^{n-1} nicht gelten kann, da $+2^{n-1}$ im 2er-Komplement mit n bits nicht darstellbar ist. Für die Null gilt die Regel nur wenn man die Addition $1 + f(\bar{x})$ modulo 2^n ausführt.

Vorzeichenwechsel

Sei $x = \langle x_{n-1}, \ldots, x_0 \rangle$ ein Bitfolge mit $\langle x_{n-2}, \ldots, x_0 \rangle \neq \langle 0, \ldots, 0 \rangle$.
Die Repräsentation für die Zahl $-f_{\mathrm{ZK}}(x)$ im 2er Komplement (d.h. $f_{\mathrm{ZK}}^{-1}(-f_{\mathrm{ZK}}(x))$) erhält man durch

$$f^{-1}(f(\bar{x}) + 1)$$

wobei $\bar{x} = \langle \bar{x}_{n-1}, \ldots, \bar{x}_0 \rangle$ die invertierte Bitfolge bezeichnet.

D.h. man invertiert die Bitfolge und addiert 1 auf die sich ergebende Zahl.

Negative Zahlen

Beweis

1. Fall: $x_{n-1} = 1$, d.h. $f_{\mathrm{ZK}}(x)$ negativ

$$-f_{\mathrm{ZK}}(x) = 2^n - \sum_{i=0}^{n-1} x_i 2^i = 1 + \sum_{i=0}^{n-1} 2^i - \sum_{i=0}^{n-1} x_i 2^i$$

$$= 1 + \sum_{i=0}^{n-1} (1 - x_i) 2^i = 1 + \sum_{i=0}^{n-1} \bar{x}_i 2^i$$

$$= 1 + f(\bar{x})$$

Da $1 + f(\bar{x}) < 2^{n-1}$ liefert Anwendung von f^{-1} oder f_{ZK}^{-1} die gleiche Bitfolge.

Negative Zahlen

Beweis

2. Fall: $x_{n-1} = 0$, d.h. $f_{\mathrm{ZK}}(x)$ strikt positiv

$$-f_{\mathrm{ZK}}(x) = -\sum_{i=0}^{n-1} x_i 2^i = \sum_{i=0}^{n-1} (1-x_i)2^i - \sum_{i=0}^{n-1} 2^i - 1 + 1$$

$$= 1 + \sum_{i=0}^{n-1} \bar{x}_i 2^i - 2^n = 1 + f(\bar{x}) - 2^n$$

Für eine Zahl z die das höchstwertige Bit $n-1$ gesetzt hat gilt $f_{\mathrm{ZK}}^{-1}(z - 2^n) = f^{-1}(z)$. Dies gilt für $1 + f(\bar{x})$.

Negative Zahlen

Definition

Die Restklasse $[a]_m$ enthält alle $z \in \mathbb{Z}$ die bei Division durch m den gleichen Rest lassen.

a heißt Repräsentant der Restklasse. Eine Restklasse hat viele verschiedene Repräsentanten.

Für eine Restklasse $M \subseteq \mathbb{Z}$ nennen wir $a \in M$ mit $0 \leq a < m$ den Standardrepräsentanten der Restklasse.

Beispiel

- $[2]_8 = [42]_8 = [-78]_8$

Negative Zahlen

Rechnen mit Restklassen

Man kann mit Restklassen rechnen. Die Multiplikation / Addition / Subtraktion etc. wird repräsentantenweise ausgeführt. Die Wahl des Repräsentanten ist unwichtig!!!!!!!!!

Beispiele:

- $[2]_8 \cdot [7]_8 = [2 \cdot 7]_8 = [6]_8$
- $[-6]_8 \cdot [23]_8 = [-6 \cdot 23]_8 = [-138]_8 = [-18]_8 = [6]_8$
- $[7]_8 + [8]_8 = [15]_8 = [-1]_8$

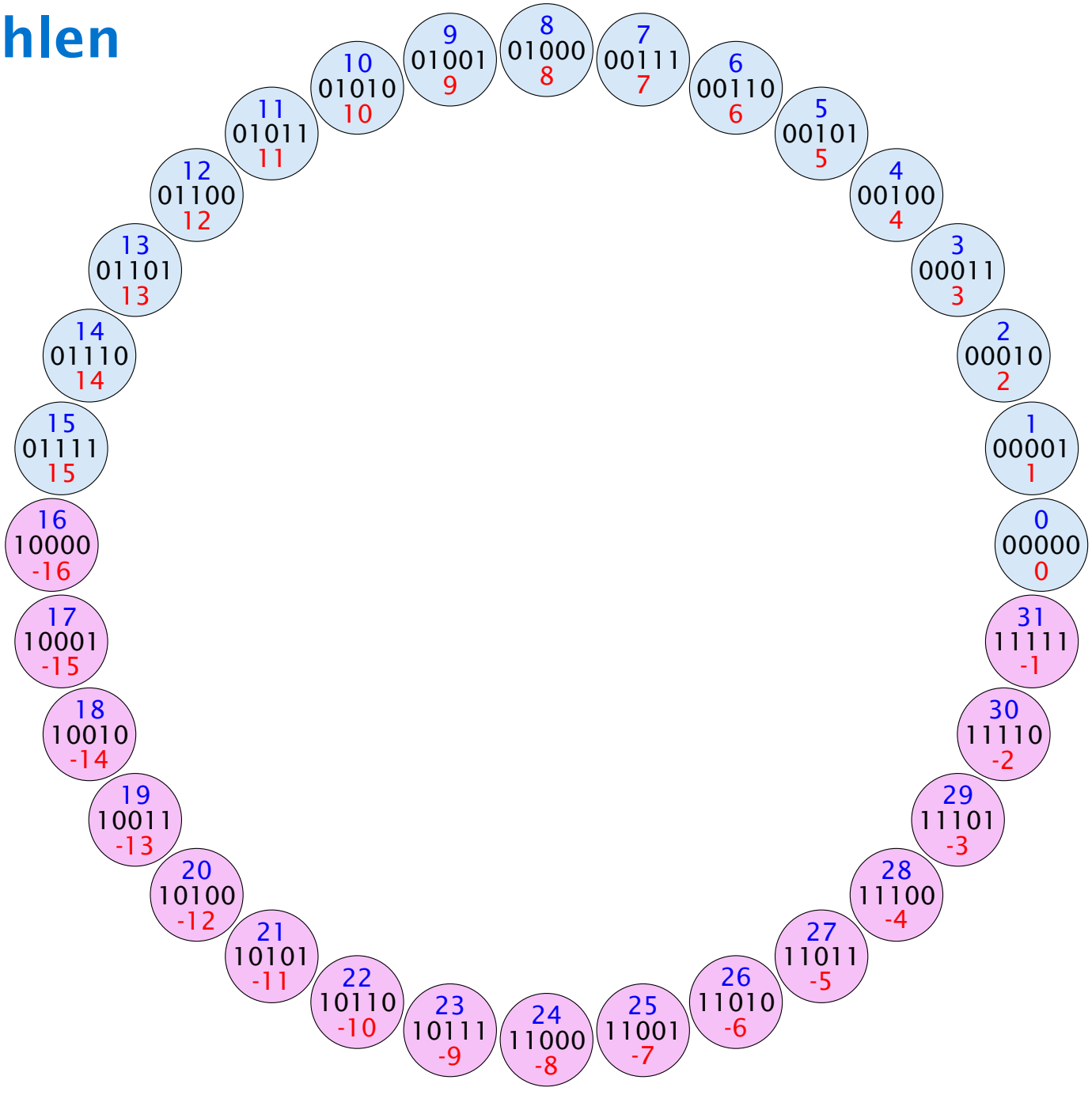
Negative Zahlen

Die Hardware Implementierung von Addition / Multiplikation etc. implementiert eigentlich eine Operation auf Restklassen modulo 2^n, wobei n der Bitlänge entspricht.

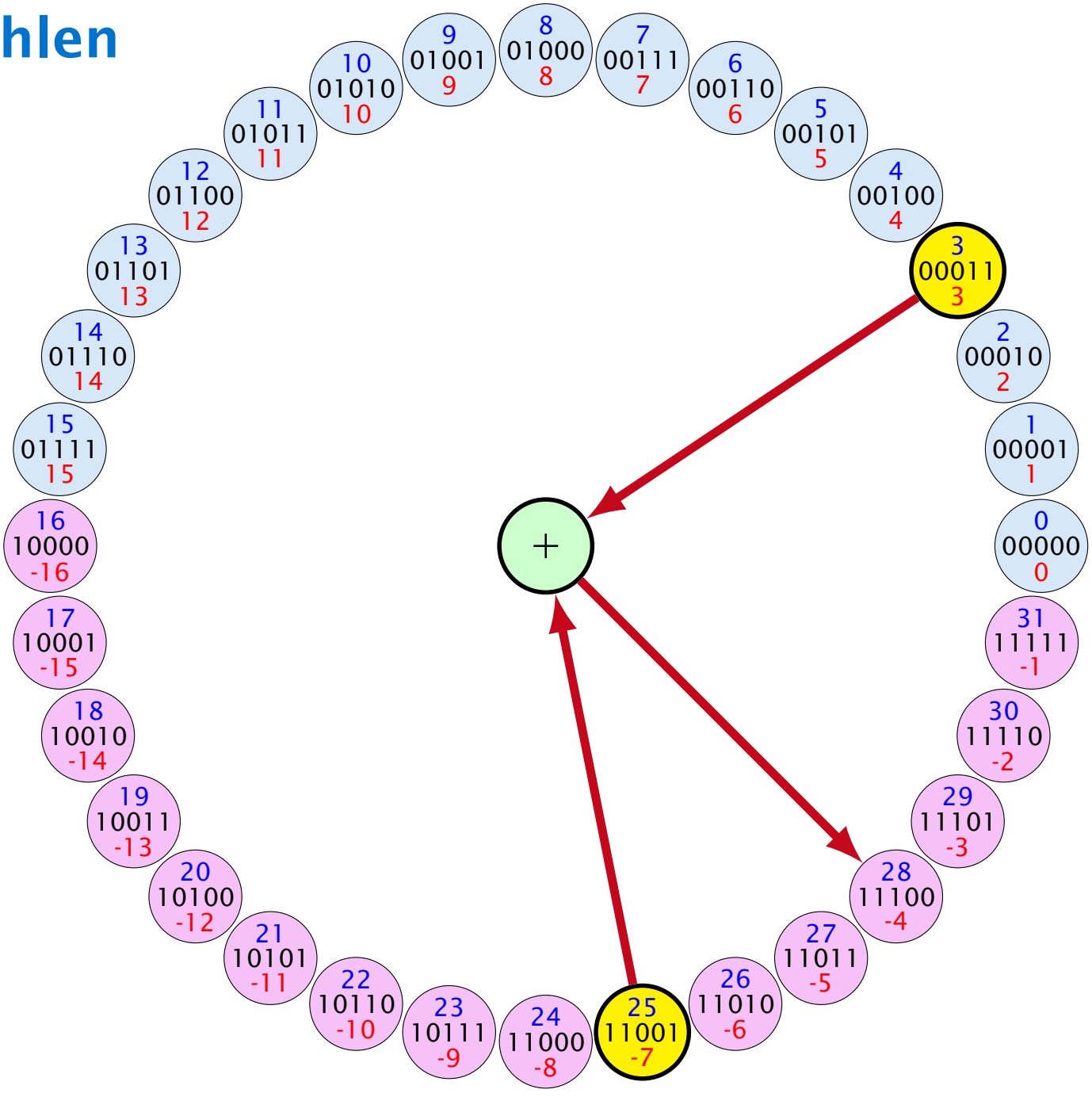
Im Prinzip wird eine Operation (Addition / Subtraktion / Multiplikation / Ganzahldivision) ausgeführt, und dann werden überzählige Bits verworfen (d.h., dass Ergebnis wird modulo 2^n genommen).

Durch das Verwenden des 2er-Komplements kann man für signed und unsigned Datentypen, (im wesentlichen) die gleiche Hardware benutzen.

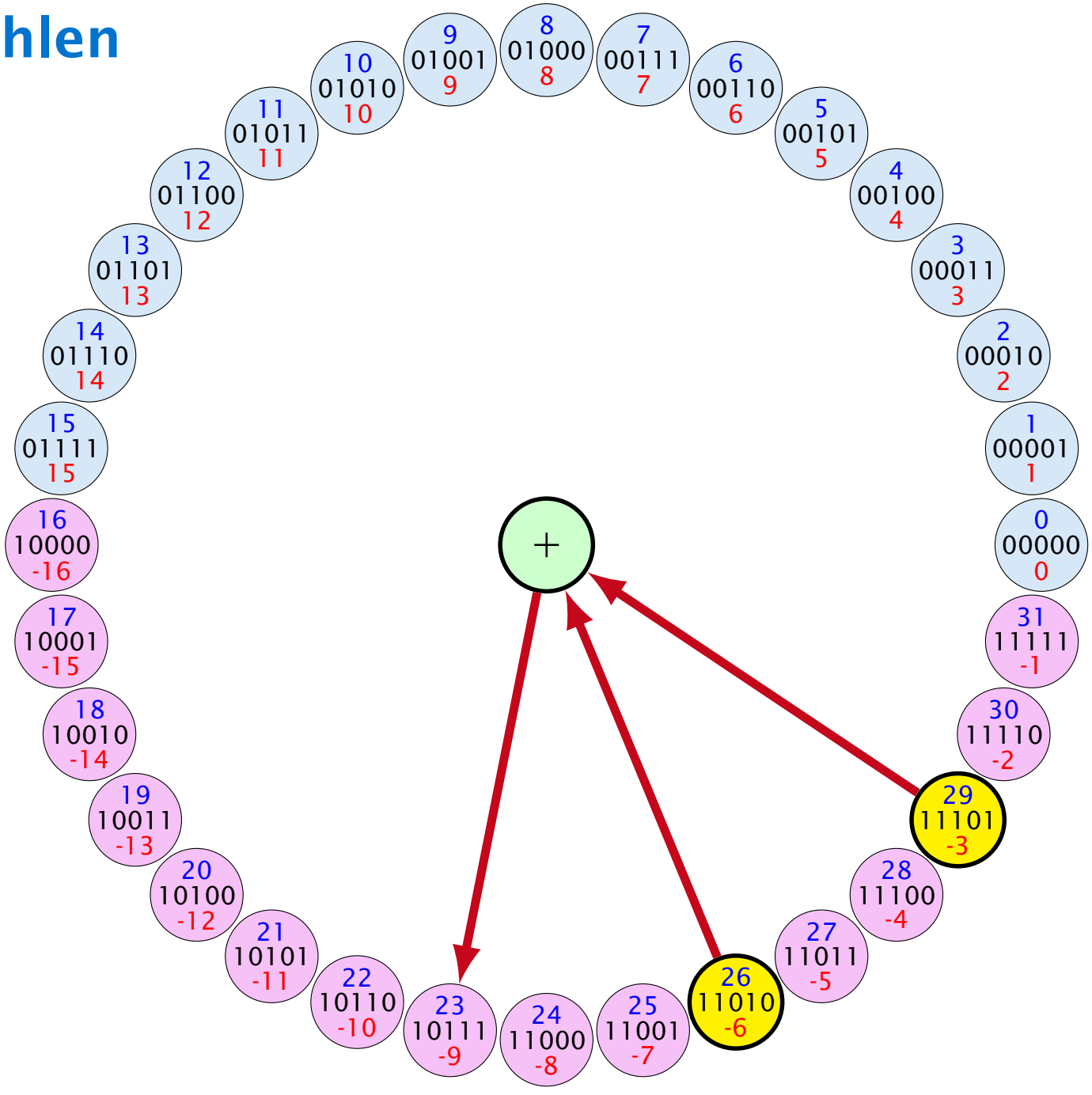
Negative Zahlen



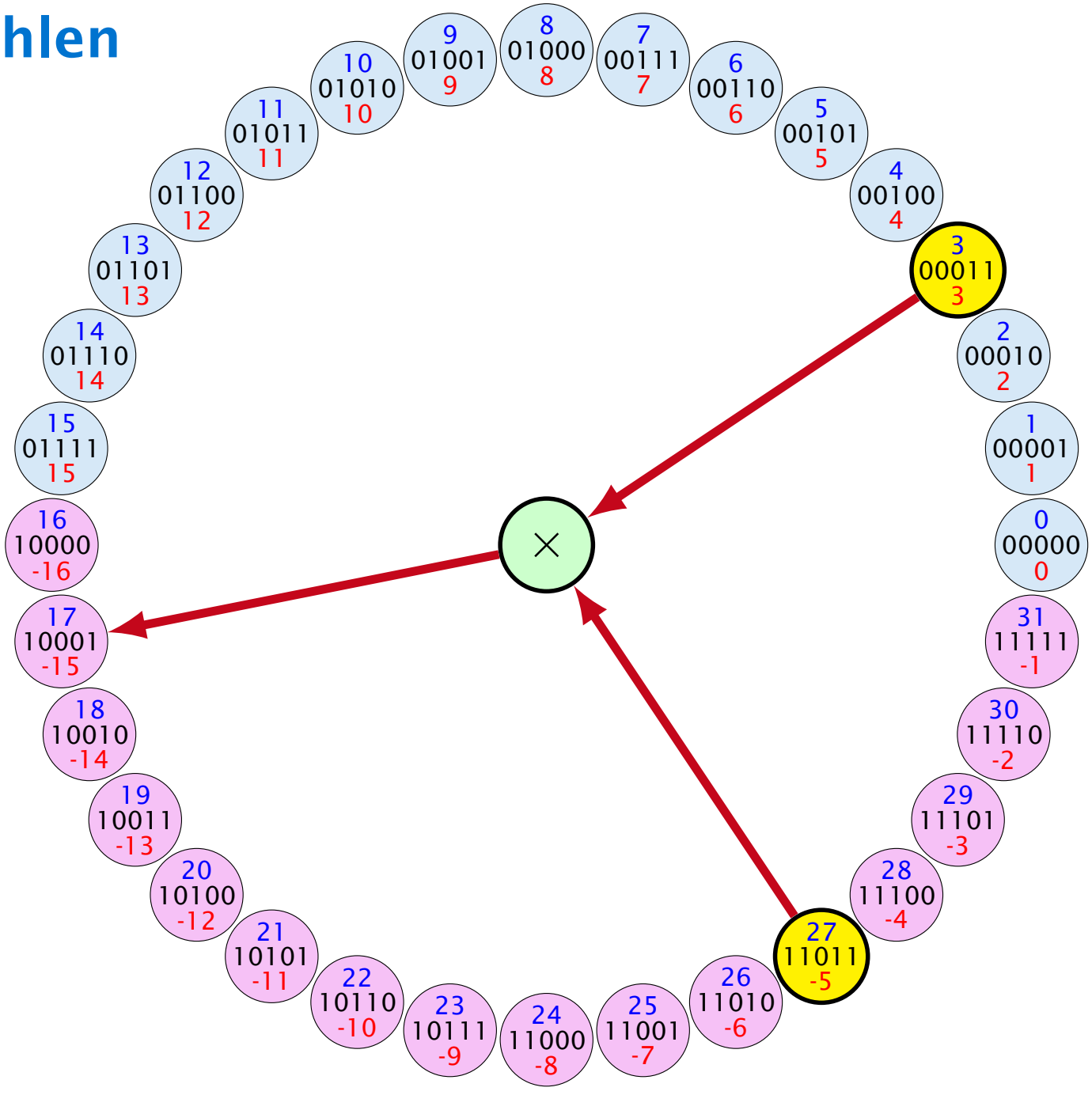
Negative Zahlen



Negative Zahlen



Negative Zahlen



Ganze Zahlen in C-ähnlichen Sprachen

Ganze Zahlen:

Die primitiven Datentypen für ganze Zahlen sind:

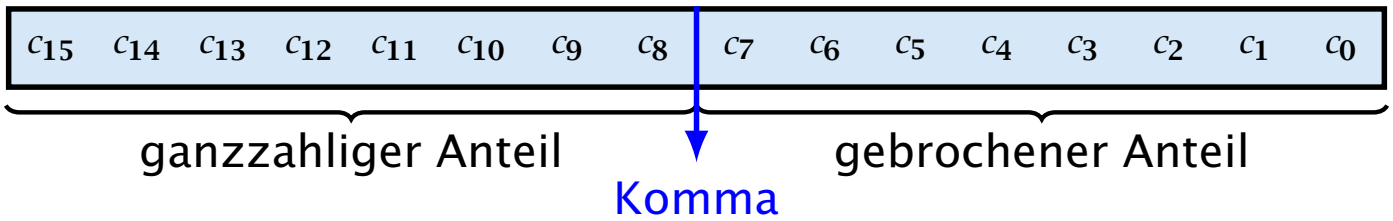
- **8 Bits:** `unsigned char` $\{0,\ldots,255\}$
  `signed char` $\{-128,\ldots,127\}$
- **16 Bits:** `unsigned short` $\{0,\ldots,65535\}$
  `signed short` $\{-32768,\ldots,32767\}$
- **32 Bits:** `unsigned long` $\{0,\ldots,2^{32}-1\}$
  `signed long` $\{-2^{31},\ldots,2^{31}-1\}$
- **64 Bits:** `unsigned long long` $\{0,\ldots,2^{64}-1\}$
  `signed long long` $\{-2^{63},\ldots,2^{63}-1\}$
- `signed` kann weggelassen werden (ausser bei `char`!)
- `unsigned int` und `signed int` sind je nach System 16, 32 oder 64 Bit

Dies sind nur Mindestvorgaben. Auf der Cray ist ein `short` z.B. 64 Bit lang.

Rationale Zahlen I

Festkommadarstellung: Komma an fester Stelle in Zahl

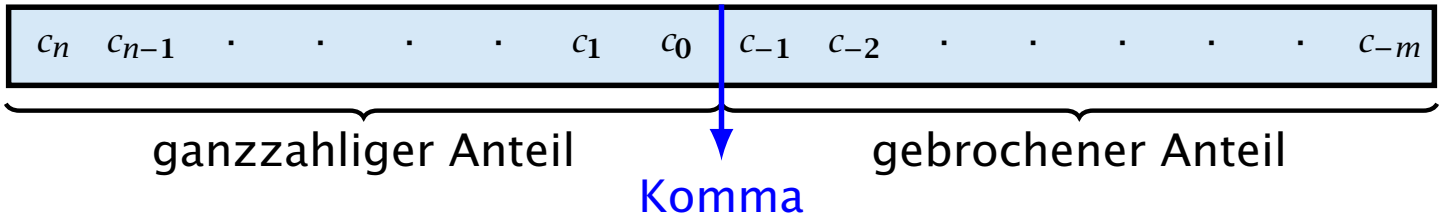
Beispiel mit $n = 16$:



Nachteile:

- weniger große Zahlen darstellbar
- feste Genauigkeit der Nachkommastellen

Rationale Zahlen II



Interpretation für $r \in \mathbb{Q}$:

$$r = c_n \cdot 2^n + \ldots + c_0 \cdot 2^0 + c_{-1}2^{-1} + \ldots + c_{-m} \cdot 2^{-m}$$

mit $n+1$ Vorkomma- und m Nachkommaziffern

Beispiel:

$$11.01_2 = 1 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2}$$
$$= 2 + 1 + 0 + \tfrac{1}{4} = 3.25_{10}$$

Floating Point Zahlen I

Wissenschaftliche Notation: $x = a \cdot 10^b$ für $x \in \mathbb{R}$, wobei:

- $a \in \mathbb{R}$ mit $1 \leq |a| < 10$
- $b \in \mathbb{Z}$

Beispiele:

- $-2.7315 \cdot 10^2$ °C absoluter Nullpunkt
- $1.5 \cdot 10^9$ Hz Taktfrequenz A8X Prozessor

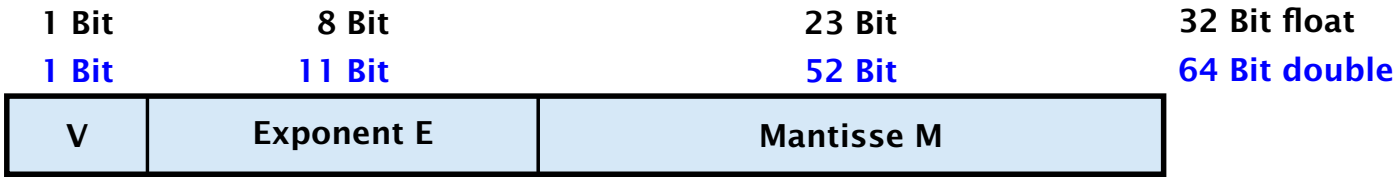
Drei Bestandteile:

- Vorzeichen
- Mantisse $|a|$ (bestimmt die Genauigkeit)
- Exponent b (bestimmt Größe des Wertebereichs)

Problem: bei fester Länge der Mantisse (z.B. 3 Ziffern)

- zwischen $1.23 \cdot 10^4 = 12300$ und $1.24 \cdot 10^4 = 12400$ keine Zahl darstellbar!

Floating Point Zahlen II



- wissenschaftliche Darstellung mit Basis 2

$$f = (-1)^V \cdot (1 + M) \cdot 2^{E-bias}$$

- Vorzeichen Bit V
- Mantisse M hat immer die Form $1.abc$, also wird erste Stelle weggelassen („hidden bit“)
- Exponent E wird vorzeichenlos abgespeichert, verschoben um $bias$
  - bei 32 bit: $bias = 127$, bei 64 bit: $bias = 1023$

Floating Point Zahlen III

Übliche Floating Point Formate:

Bit	*Vorz.*	*Exp.*	*Mant.*	*Dezimal stellen*	*Bereich*
32	1 Bit	8 Bit	23 Bit	~ 7	$\pm 2 \cdot 10^{-38}$ bis $\pm 2 \cdot 10^{38}$
64	1 Bit	11 Bit	52 Bit	~ 15	$\pm 2 \cdot 10^{-308}$ bis $\pm 2 \cdot 10^{308}$
80	1 Bit	15 Bit	64 Bit	~ 19	$\pm 1 \cdot 10^{-4932}$ bis $\pm 1 \cdot 10^{4932}$

In C:

`float` (32 Bit), `double` (64 Bit), `long double` (80 Bit)

Vorsicht mit Floating Point!

Floating Point Zahlen sind bequem, aber Vorsicht!

- Viele Dezimalzahlen haben keine Floating Point Darstellung
  - Beispiel: $0.1_{10} = 0.0001100110011\ldots_2$ (periodisch)
- Durch feste Länge der Mantisse sind ebenfalls viele Zahlen nicht darstellbar
  - Beispiel: mit 3 Ziffern Mantisse ist zwischen $1.23 \cdot 10^4 = 12300$ und $1.24 \cdot 10^4 = 12400$ keine Zahl darstellbar!
- Kritisch sind Vergleiche von Floating Point Zahlen
  - Beispiel: `(0.1 + 0.2 == 0.3)` ist meist FALSE!
- Zins-Berechnungen und dergleichen NIE mit Floating Point Zahlen!
  - Stattdessen: spezielle Bibliotheken wie GMP

Vorsicht mit Floating Point!

Floating Point Zahlen sind bequem, aber Vorsicht!

- Viele Dezimalzahlen haben keine Floating Point Darstellung
  - Beispiel: $0.1_{10} = 0.0001100110011\ldots_2$ (periodisch)
- Durch feste Länge der Mantisse sind ebenfalls viele Zahlen nicht darstellbar
  - Beispiel: mit 3 Ziffern Mantisse ist zwischen $1.23 \cdot 10^4 = 12300$ und $1.24 \cdot 10^4 = 12400$ keine Zahl darstellbar!
- Kritisch sind Vergleiche von Floating Point Zahlen
  - Beispiel: (0.1 + 0.2 == 0.3) ist meist FALSE!
- Zins-Berechnungen und dergleichen NIE mit Floating Point Zahlen!
  - Stattdessen: spezielle Bibliotheken wie GMP

Vorsicht mit Floating Point!

Floating Point Zahlen sind bequem, aber Vorsicht!

- Viele Dezimalzahlen haben keine Floating Point Darstellung
  - Beispiel: $0.1_{10} = 0.0001100110011\ldots_2$ (periodisch)
- Durch feste Länge der Mantisse sind ebenfalls viele Zahlen nicht darstellbar
  - Beispiel: mit 3 Ziffern Mantisse ist zwischen $1.23 \cdot 10^4 = 12300$ und $1.24 \cdot 10^4 = 12400$ keine Zahl darstellbar!
- Kritisch sind Vergleiche von Floating Point Zahlen
  - Beispiel: `(0.1 + 0.2 == 0.3)` ist meist FALSE!
- Zins-Berechnungen und dergleichen NIE mit Floating Point Zahlen!
  - Stattdessen: spezielle Bibliotheken wie GMP

Vorsicht mit Floating Point!

Floating Point Zahlen sind bequem, aber Vorsicht!

- Viele Dezimalzahlen haben keine Floating Point Darstellung
  - Beispiel: $0.1_{10} = 0.0001100110011\ldots_2$ (periodisch)
- Durch feste Länge der Mantisse sind ebenfalls viele Zahlen nicht darstellbar
  - Beispiel: mit 3 Ziffern Mantisse ist zwischen $1.23 \cdot 10^4 = 12300$ und $1.24 \cdot 10^4 = 12400$ keine Zahl darstellbar!
- Kritisch sind Vergleiche von Floating Point Zahlen
  - Beispiel: `(0.1 + 0.2 == 0.3)` ist meist FALSE!
- Zins-Berechnungen und dergleichen NIE mit Floating Point Zahlen!
  - Stattdessen: spezielle Bibliotheken wie GMP

Definition Datenstruktur

Definition Datenstruktur (nach Prof. Eckert)
Eine Datenstruktur ist eine

- **logische Anordnung** von Datenobjekten,
- die **Informationen** repräsentieren,
- den **Zugriff** auf die repräsentierte Information über **Operationen** auf Daten ermöglichen und
- die Information **verwalten.**

Zwei Hauptbestandteile:

- Datenobjekte
  z.B. definiert über primitive Datentypen
- Operationen auf den Objekten
  z.B. definiert als Funktionen

Primitive Datentypen in C

Natürliche Zahlen, z.B. `unsigned short`, `unsigned long`

- Wertebereich: bei n Bit von 0 bis $2^n - 1$
- Operationen: `+`, `-`, `*`, `/`, `%`, `<`, `==`, `!=`, `>`

Ganze Zahlen, z.B. `int`, `long`

- Wertebereich: bei n Bit von -2^{n-1} bis $2^{n-1} - 1$
- Operationen: `+`, `-`, `*`, `/`, `%`, `<`, `==`, `!=`, `>`

Floating Point Zahlen, z.B. `double`, `float`

- Wertebereich: abhängig von Größe
- Operationen: `+`, `-`, `*`, `/`, `<`, `==`, `!=`, `>`

Logische Werte, `bool`

- Wertebereich: `true`, `false`
- Operationen: `&&`, `||`, `!`, `==`, `!=`