



LUDWIG-
MAXIMILIANS-
UNIVERSITY
MUNICH



Skript zur Vorlesung:

Einführung in die Informatik: Systeme und Anwendungen

Sommersemester 2017

Kapitel 1: Informationsverarbeitung durch Programme

Vorlesung: Prof. Dr. Christian Böhm

Übungen: Dominik Mautz

Skript © Christian Böhm, Peer Kröger

<http://www.dbs.ifi.lmu.de/cms/>

Einführung_in_die_Informatik_Systeme_und_Anwendungen

1.1 Informationen und Daten

- Was ist Informatik?
 - Wissenschaft, Technik und Anwendung der maschinellen Verarbeitung und Übermittlung von Daten

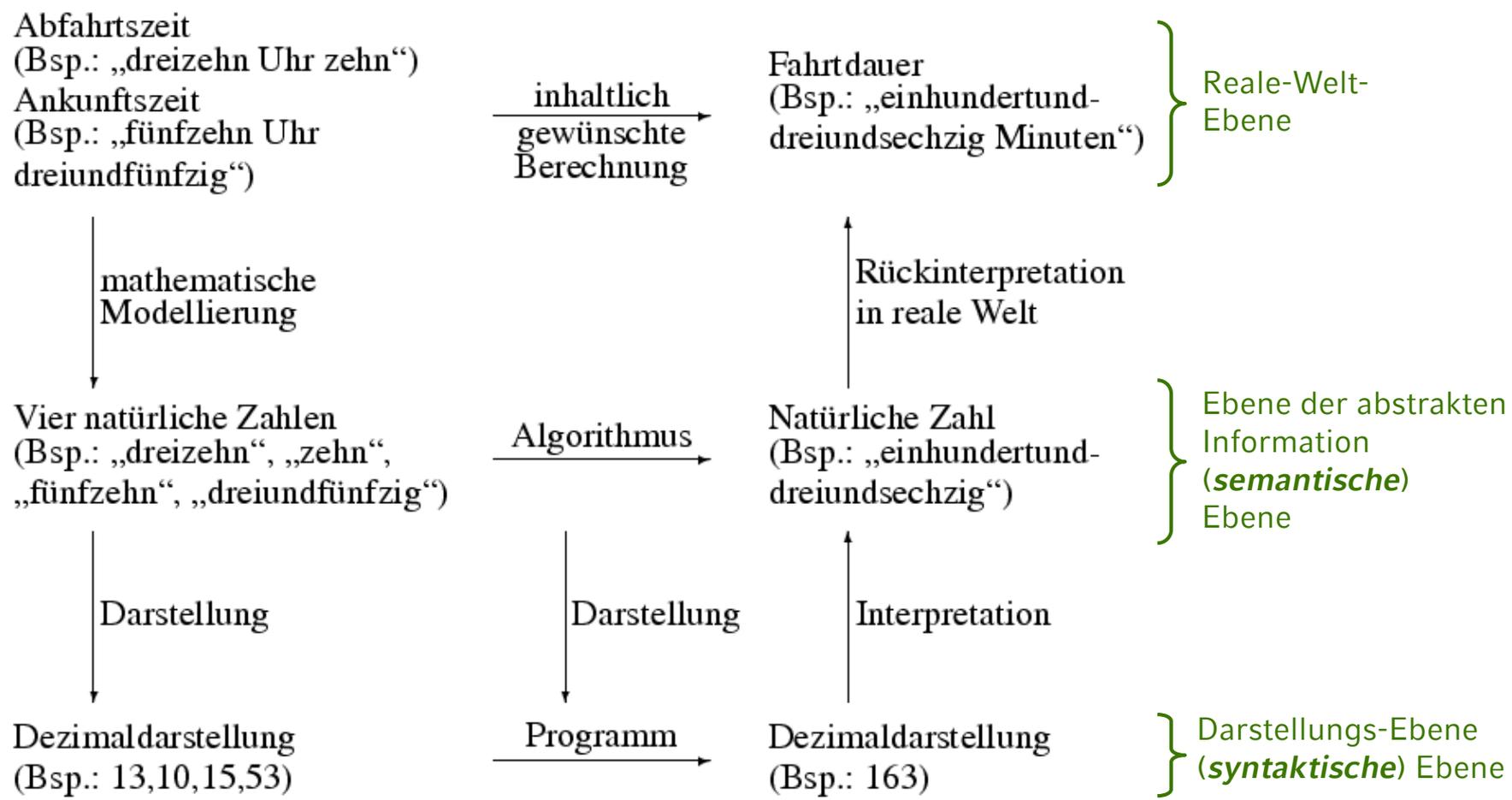
[Gesellschaft für Informatik: Studien- und Forschungsführer Informatik, Springer-Verlag]
- Algorithmus
 - Grundlage jeglicher maschineller Informationsverarbeitung
 - Zentraler Begriff der Informatik
 - Systematische, „schematisch“ („automatisch“, „mechanisch“) ausführbare Verarbeitungsvorschrift
 - Beispiele aus dem Alltag:
 - Kochrezepte, Bedienungsanleitungen, Aufbauvorschriften
 - Mathematische Berechnungsverfahren (z.B. Summe $1 + 2 + \dots + n$ für ein beliebiges n)

1.1 Informationen und Daten

- Beispiel:
 - Berechne zu einer gegebenen Abfahrts- und Ankunftszeit eines Zuges seine Fahrtzeit (zur Vereinfachung: keine Fahrt dauert länger als 24 Stunden)
 - *Was ist zu verarbeiten?*
 - Abfahrts-/Ankunftszeit, Fahrtzeit: Dinge der realen Welt
z.B. „dreizehn Uhr zehn“
 - *Wie wird es verarbeitet?*
 - Abstraktion: mathematische Modelle der realen Dinge
z.B. die Zahlen „dreizehn“ und „zehn“
 - Verarbeitung der abstrakten „Eingabe“-Objekte liefert ein abstraktes „Ausgabe“-Objekt
z.B. die Zahl „einhundertdreundsechzig“
 - Rückinterpretation liefert: Antwort auf die gestellte Aufgabe in der realen Welt
z.B. „Die Fahrtzeit beträgt einhundertdreundsechzig Minuten“

1.1 Informationen und Daten

- Schematisch



1.1 Informationen und Daten

- Darstellung der Informationen
 - Zu verarbeiten (nach Abstraktion): Zahlen (mathematische Objekte), z.B. „dreizehn“
 - Notwendig: **Darstellung** der Zahlen
 - Typische Darstellung von „dreizehn“:
13 (*Dezimaldarstellung*)
 - Andere Möglichkeiten:
|||||
1101 (*Binärdarstellung*)
XIII
DREIZEHN
(usw.)
 - Neben natürlichen Zahlen werden auch andere Typen von Daten (Sorten) verarbeitet:
 - Reelle Zahlen, ganze Zahlen
 - Zeichen, Zeichenketten (Strings)

1.2 Algorithmen

- Algorithmus für die Fahrtzeitberechnung
 - Eingabe: vier natürliche Zahlen, **Parameter** hierfür: h_{AB} , m_{AB} , h_{AN} , m_{AN}
 - Ergebnis: natürliche Zahl
 - Berechnung:
 - Falls Fahrt nicht über Mitternacht hinausgeht:
Das Ergebnis ergibt sich aus $(h_{AN} - h_{AB}) \cdot 60 + m_{AN} - m_{AB}$
 - Falls Fahrt über Mitternacht geht
Das Ergebnis ergibt sich aus $(23 - h_{AB}) \cdot 60 + (60 - m_{AB}) + h_{AN} \cdot 60 + m_{AN}$

- Bestandteile des Algorithmus
 - Daten meist durch **Variablen** repräsentiert (z.B. h_{AB} repräsentiert die Stundenzahl der Abfahrtszeit)
 - Operationen auf den Daten („elementare Verarbeitungsschritte“)
 - „Zusammensetzung“ des Berechnungsvorgangs

1.2 Algorithmen

- Algorithmus
 - löst (typischerweise) eine Klasse von Aufgaben, die durch seine **Parameter** (Eingabe-Variablen) bestimmt ist. Eine **Eingabe** besteht aus konkreten (aktuell zu verarbeitenden) Daten für die Parameter (z.B. „13“ für h_{AB}).
 - seine **Ausführung** wird durch eine Eingabe erzeugt und liefert i.d.R. **Ergebnisse**. Diese können Daten oder Steuersignale sein.
- Ausführungsbeispiel Fahrtzeit-Algorithmus
 - Eingabe 13, 10, 15, 53 für h_{AB} , m_{AB} , h_{AN} , m_{AN} liefert das Ergebnis 163
 - Eingabe 22, 15, 1, 30 für h_{AB} , m_{AB} , h_{AN} , m_{AN} liefert das Ergebnis 195

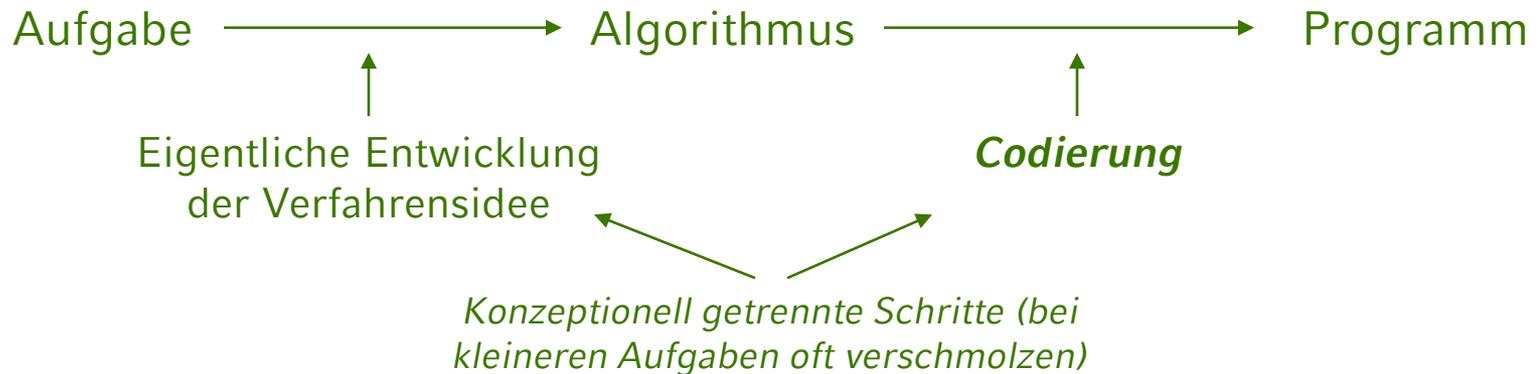
1.2 Algorithmen

- Grundanforderungen an Algorithmen
 - Präzise Darstellung:
die zu verarbeitenden Daten und die Verarbeitungsvorschrift müssen unmissverständlich aufgeschrieben sein
 - Effektivität:
jeder elementare Verarbeitungsschritt muss von der zugrunde liegenden „Verarbeitungseinheit“ (**Prozessor**) ausführbar sein

1.3 Programme

- Darstellung von Algorithmen
 - Zur Ausführung auf einem Computer muss ein Algorithmus, d.h. seine Daten, elementaren Verarbeitungsschritte und zusammengesetzte Verarbeitungsvorschrift formal in einer Programmiersprache (als **Programm**) dargestellt werden

- Entwicklung von Algorithmen
 - Zentrale Aufgabe des Informatikers



1.3 Programme

- Pseudo-Code
 - Aufschreiben von Algorithmen in der Entwicklungsphase
 - Keine konkrete Programmiersprache
 - Programmiersprachenähnliche Darstellung
 - Verwendung algorithmische Konzepte und mathem. Schreibweisen
 - Meist verbale Zusätze
 - ***Gut für Menschen lesbar!!!***
 - Beispiel:

algorithmus Fahrtzeit

input $h_{AB}, m_{AB}, h_{AN}, m_{AN}$: natürliche Zahlen

output Fahrtzeit : natürliche Zahl

begin

if Fahrt geht nicht über Mitternacht

then Fahrzeit = $(h_{AN} - h_{AB}) \cdot 60 + m_{AN} - m_{AB}$;

else Fahrzeit = $(23 - h_{AB}) \cdot 60 + (60 - m_{AB}) + h_{AN} \cdot 60 + m_{AN}$;

end

1.3 Programme

- Bestandteile/Konzepte
 - Variablen
 - Intuitiv: Zettel, auf die ein Wert geschrieben werden kann
 - Der Wert kann abgelesen werden
 - Der Wert kann verändert werden (radieren und neu schreiben)
 - Formal: ein Abschnitt im Speicher
 - Typischerweise haben Variablen Typen (Sorten), d.h. der Zettel kann nur Werte eines speziellen Typs aufnehmen (z.B. Typ „natürliche Zahl“)
 - Wir gehen davon aus, dass es vordefinierte Typen gibt, die wir verwenden können
 - Nat** : „natürliche Zahlen“
 - Real** : „reelle Zahlen“
 - Int** : „ganze Zahlen“
 - Char** : „druckbare Zeichen“
 - String** : „Zeichenketten“
 - Bool** : „Wahrheitswerte“ (Wertemenge {*wahr*, *falsch*})

1.3 Programme

- Beispiel
 - h_{AB} , m_{AB} , h_{AN} , m_{AN} sind Variablen, die die Eingabewerte des Fahrzeitalgorithmus enthalten; die Variablen sind vom Typ „natürliche Zahl“ (**Nat**), d.h. sie können nur Werte von natürliche Zahlen speichern
 - » Dadurch ist der Algorithmus generell einsetzbar für beliebige An-/Abfahrtzeiten
 - » Beim Aufruf des Algorithmus mit konkreten Werten werden diese Werte auf den entsprechenden Zettel geschrieben
 - Fahrzeit ist eine Variable, die das Resultat des Algorithmus verwaltet; Typ: „natürliche Zahl“ (**Nat**)
 - Zudem können beliebig viele Variablen vereinbart werden, z.B. um Zwischenergebnisse zu speichern

1.3 Programme

– Anweisungen

- $\text{Fahrzeit} = (h_{AN} - h_{AB}) \cdot 60 + m_{AN} - m_{AB}$

ist eine **Anweisung**, der Wert der linke Seite wird in der Variablen Fahrzeit gespeichert (auf den „Zettel“ mit Namen Fahrzeit geschrieben)

Jede Anweisung wird mit einem „;“ beendet!!!

- Bedingte Anweisung

if <Bedingung> **then** <Anweisung01> **else** <Anweisung02>

führt je nachdem, ob die <Bedingung> wahr oder falsch ist, <Anweisung01> oder <Anweisung02> aus (dabei kann es sich jeweils um mehr als eine Anweisung handeln, einem sog. **Block** von Anweisungen innerhalb { } Klammern; der **else**-Fall kann fehlen)

- Wiederholungsanweisungen

while <Bedingung> **do** <Anweisung>

testet, ob <Bedingung> wahr ist, wenn ja, wird <Anweisung> so oft ausgeführt bis <Bedingung> falsch ist; ist <Bedingung> falsch, wird mit den Anweisungen nach <Anweisung> fortgefahren

1.3 Programme

- Beispiel

Programm zur Berechnung der Fakultät $n!$ einer natürlichen Zahl n

```

algorithmus Fakultaet ← Name des Algorithmus
input      n : Nat ← Eingabevariable(n) (inkl. Typ)
output    Fakultaet : Nat ← Resultatvariable (inkl. Typ)
variables i : Nat ← Vereinbarung der im Algorithmus
                    verwendeten Variablen (inkl. Typ)
begin
    i = 1 ; ← Anweisung (Wert-Zuweisung)
    Fakultaet = 1 ; ← Anweisung (Wert-Zuweisung)
    while i <= n
    do {
        Fakultaet = Fakultaet * i ;
        i = i + 1 ;
    }
    ← Anweisungsblock "Schleifenrumpf"
    ← Wiederholungsanweisung
end ← Ende des Algorithmus

```

1.3 Programme

- Beispiel
Berechnung des Absolutbetrags $|n|$ einer natürlichen Zahl n

```
algorithmus Absolutbetrag
input      n : Int
output     betrag : Int
begin
  if n <= 0
  then      betrag = -n;
  else      betrag = n;
end
```

1.3 Programme

– Abstraktion

- Wir werden einen Block an Anweisungen abstrahieren, wenn die Details der Anweisungen keinen Rolle spielen

- Beispiel

if Details unwichtig **then** <Anweisungsfolge, die uns nicht interessiert>

– Kommentare

- Kommentare könne an jeder Stelle platziert werden und dienen dazu, den Algorithmus zu erklären
- Kommentare beginnen mit „//“ und enden am Zeilenende
- Beispiel

```
i = k + 1;           // berechne k+1 und weise den Wert i zu      (einzeiliger Kommentar)
```

```
...
```

```
i = x + y;         // berechne die Summe aus den Werten der Variablen x und y  
                   // und weise diesen Wert i zu              (Kommentar ueber mehrere Zeilen)
```

```
...
```

1.3 Programme

- Datenstrukturen
 - Oft muss man nicht nur einzelne Werte, sondern Mengen von Werten verarbeiten
 - Spezielle Konstrukte, die Mengen von Daten verwalten, nennt man ***Datenstrukturen***
 - Klassische Datenstruktur für eine Menge von Werten gleichen Typs ist die Liste, z.B. eine Liste von **Nat**-Werten
 - Eine Liste von **Nat**-Werten kann intuitiv als Liste von Zetteln angesehen werden, auf die **Nat**-Werte geschrieben werden können
 - Eine Datenstruktur bietet typischerweise „Schnittstellen“ an, die spezifizieren, wie man mit diesen Datenstrukturen „arbeiten“ kann, z.B. im Falle einer Liste (von beliebigen Werten gleichen Typs)
 - Ablegen eines Wertes in die Liste
 - Ablesen des Werts des ersten Elements der Liste
 - Ablesen der Größe der Liste (= Anzahl der Elemente, die aktuell gespeichert werden)
 - ...

1.3 Programme

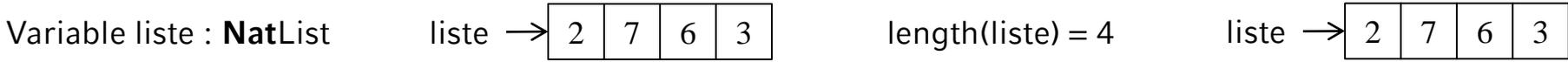
- Beispiel: Liste von **Nat**-Werten: **NatList**

- Operationen

- » Länge (Anzahl der Werte in der Liste)

length: **NatList** → \mathbb{N}

Anmerkung: $\mathbb{N} = \{0, 1, 2, \dots\}$ ist die Menge der natürlichen Zahlen



- » Erster Wert in der Liste ablesen und löschen

getAndDeleteFirst: **NatList** → **Nat**



- » Einen Wert am Ende der Liste anhängen

append: **NatList** × **Nat** → \emptyset



1.3 Programme

- Beispiel: Algorithmus zur Suche eines gegebenen **Nat**-Wertes in einer Liste von **Nat**-Werten

```
algorithmus SuchenInListe
input       liste: NatList, wert: Nat
output     enthalten: Bool
begin
    enthalten = falsch;
    while length(liste) > 0
    do {
        if       getAndDeleteFirst(liste) = wert
        then     enthalten = wahr;
    }
end
```

1.3 Programme

– Ablaufbeispiel

```

begin
enthalten = falsch;
while length(liste) > 0
do {
    if getAndDeleteFirst(liste) = wert then enthalten = wahr;
}
end
    
```

	liste	wert	enthalten	length(liste)	getAndDeleteFirst(liste)					
<u>Initial/1. while-Schleife</u>	<table border="1"><tr><td>2</td><td>7</td><td>6</td><td>3</td></tr></table>	2	7	6	3	6	falsch	4	<table border="1"><tr><td>2</td></tr></table>	2
2	7	6	3							
2										
<u>2. while-Schleife</u>	<table border="1"><tr><td>7</td><td>6</td><td>3</td></tr></table>	7	6	3	6	falsch	3	<table border="1"><tr><td>7</td></tr></table>	7	
7	6	3								
7										
<u>3. while-Schleife</u>	<table border="1"><tr><td>6</td><td>3</td></tr></table>	6	3	6	falsch	2	<table border="1"><tr><td>6</td></tr></table>	6		
6	3									
6										
<u>4. while-Schleife</u>	<table border="1"><tr><td>3</td></tr></table>	3	6	wahr	1	<table border="1"><tr><td>3</td></tr></table>	3			
3										
3										
<u>Schleifenabbruch</u>	∅	6	wahr	0	⊥					