



Grundlagen der Informatik

Informatik/Automobilinformatik/DVM

Prof. Dr. Dieter Nazareth

Grundlagen der Informatik

Eingrenzung der Themen



Zielsetzung

- Überblick über die wichtigsten Themen der Informatik
- Fokus: Grundlagen der Programmierung
- Schaffen einer fundierten Grundlage für das weitere Lern- und Arbeitsleben im Umfeld der Informatik

Nicht behandelte Themen

- KEIN Lehrbuch für eine bestimmte Programmiersprache

Empfohlene Literatur

- H.P. Gumm, M. Sommer: Einführung in die Informatik, aktuellste Auflage, Oldenbourg Verlag
- H. Herold, B. Lurz, J. Wohlrab, M. Hopf: Grundlagen der Informatik, aktuellste Auflage, Pearson Studium
- M. Broy: Informatik. Eine grundlegende Einführung Band 1: Programmierung und Rechnerstrukturen, aktuellste Auflage

Grundlagen der Informatik

Überblick



- Was ist Informatik
- Information und Repräsentation
- Algorithmen
- Programmiersprachen
- Datenstrukturen
- Komplexität
- Berechenbarkeit

Grundlagen der Informatik

Überblick

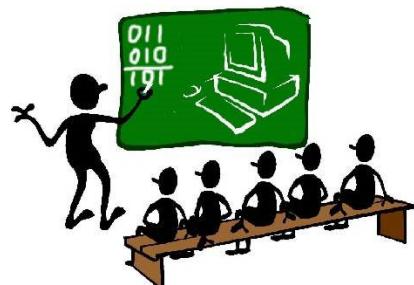


- Was ist Informatik
- Information und Repräsentation
- Algorithmen
- Programmiersprachen
- Datenstrukturen
- Komplexität
- Berechenbarkeit

Was ist Informatik Überblick



- Was ist Informatik
 - Bedeutung, Begriffe, Aufgaben
 - Einsatzbereiche und Rolle der Informatik
 - Teilgebiete
 - Historische Meilensteine



© Prof. Dr. Dieter Nazareth 2023

5

Was ist Informatik Bedeutung



Definition *Informatik*

- Einfach:
Informatik ist die Wissenschaft von der maschinellen Informationsverarbeitung.
- Umfassender:
Informatik ist die Wissenschaft, Technik und Anwendung der maschinellen Verarbeitung, Speicherung und Übertragung von Informationen.

Im englischen Sprachraum

- *Computer Science*
- Die Wissenschaft, die sich mit Rechnern beschäftigt
- Anderer Fokus auf die gleiche Thematik

© Prof. Dr. Dieter Nazareth 2023

6

Was ist Informatik Begriffe und Aufgaben



Begriffe

- Informatik = Information + Automatik
- EDV = Elektronische Datenverarbeitung
- IT = Informationstechnik, Informationstechnologie,
Information Technology

Zentrale Aufgaben

- Formalisierte Darstellung von Information
→ Daten
- Vorschriften zur Verarbeitung von Information
→ Algorithmen
- Aufbau von Maschinen zur Abarbeitung von Algorithmen
→ Computer, Rechenanlagen, Steuergeräte, ...

© Prof. Dr. Dieter Nazareth 2023

7

Was ist Informatik Einsatzbereiche



- Multiplikation zweier Zahlen auf dem Taschenrechner
- Steuerung von Fließbändern
- Hochrechnen von Wahlergebnissen
- Textverarbeitung
- Buchen einer Reise
- Verwaltung von Bankkonten
- Steuerung des Motors in einem PKW
- Konstruktion von Bauteilen (CAD)
- Wettervorhersage
- Marslandung

steigende
Komplexität

Problemstellungen jeglicher Art und Komplexität!

© Prof. Dr. Dieter Nazareth 2023

8

Was ist Informatik Rolle von Informatik und Informatiker



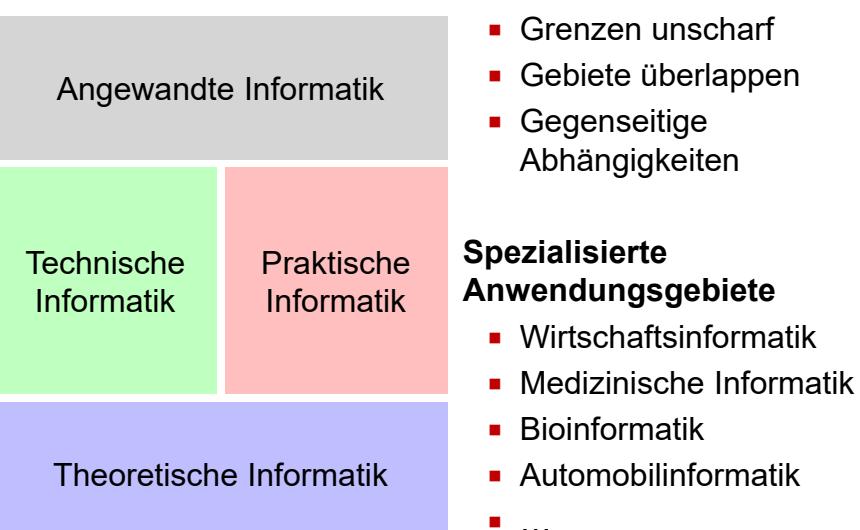
Informatik...

- Ist heute überall
- Tritt meist im Kontext anderer Aufgabenbereiche auf
- Ist Dienstleistung für Lösung des "eigentlichen Problems"

Anforderungen an Informatiker

- Solide Kenntnisse der Informatik allgemein und des Spezialgebietes, das man grade braucht
- Grundlegende Fachkenntnisse des Aufgabenbereichs, in dem das eigentliche Problem liegt
- Analytisch-logische Denkweise
- Teamfähigkeit und Kommunikationsstärke

Was ist Informatik Teilgebiete



Was ist Informatik

Teilgebiete



Fokus der theoretischen Informatik

- Abstrakte mathematische und logische Grundlagen
- Basis für alle Teilgebiete der Informatik
- Neue theoretische Erkenntnisse sind relativ schnell praktisch einsetzbar (anders als in anderen Disziplinen)
- Naturwissenschaft; starker Bezug zur Mathematik
- Automatentheorie und formale Sprachen
- Berechenbarkeitstheorie
- Komplexitätstheorie

Theoretische Informatik

Was ist Informatik

Teilgebiete



Fokus der technischen Informatik

- Maschinen für die Informationsverarbeitung (Hardware)
 - Konstruktion von Rechnern, Speicherchips, Prozessoren (Rechnerarchitektur)
 - Aufbau von Peripheriegeräten wie Festplatten, Monitor, ...
 - Rechnernetze und Protokolle für den Datenaustausch
- Ingenieurwissenschaft; starker Bezug zur Elektrotechnik

Technische
Informatik

Was ist Informatik

Teilgebiete



Fokus der praktischen Informatik

- Verarbeitung von Daten, Programme zur Rechnersteuerung
- Leicht veränderbar; *Software*
- Brücke zwischen Hardware und der Anwendungssoftware
- Trend zur Ingenieurwissenschaft (z.B. Software Engineering)
- Betriebssysteme (Windows, Linux, ...), Compiler
- Algorithmen und Datenstrukturen
- Datenbanken

Praktische
Informatik

Was ist Informatik

Teilgebiete



Fokus der angewandten Informatik

- Informatik als Lösung von Aufgaben, die außerhalb der Weiterentwicklung der Informatik liegen
- Einsatz von Rechnern in verschiedenen Lebensbereichen
- Entwickeln von Spezialprogrammen für bestimmte Aufgaben
- Beispiele: Textverarbeitung, Browser, Spiele, ...

Angewandte Informatik

Was ist Informatik Historische Meilensteine (1)



- 800 vC Indien: Erfindung des Dezimalsystems und der Zahl 0
9. Jhd Persien: Ibn Musa Al-Chwarisni
Buch "Regeln der Wiedereinsetzung und Reduktion"
(Sein Name ist Ursprung des Begriffs "Algorithmus")
- 1524 Deutschland: Adam Ries
Gesetze für das Rechnen im Dezimalsystem
(Bis dahin in Europa überwiegend römische Zahlen!!!)
- 1623 Deutschland: Wilhelm Schickard
Erste automatische Rechenmaschine (+, -, *, /)
- 1700 Deutschland: Gottfried Wilhelm Leibniz
Duales Zahlensystem
- ~1850 England/Irland: George Boole
Logische Algebra über den Werten "wahr" und "falsch"

Was ist Informatik Historische Meilensteine (2)

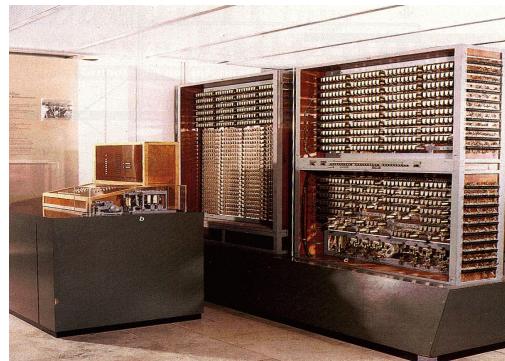


- 1934-37 Deutschland: Konrad Zuse
Z1: Mechanische programmgesteuerte Rechenanlage
auf basis des dualen Zahlensystems
- 1941 Z3 (von Zuse): Elektromechanische Rechenanlage,
2000 Relais, Speicherkapazität 64 Worte zu je 22 Bit
- 1946 USA: J.P. Eckert, J.W. Mauchly
ENIAC: Voll elektronischer Rechner, 18000 Röhren;
Damals geschätzt: 8 solche Rechner reichen weltweit
- 1946 USA: John von Neumann (geb. in Ungarn)
Von-Neumann-Architektur:
Konzept für universellen Rechner;
Rechenwerk, Speicherwerk (für Daten und Programm),
Steuerwerk, Ein-/Ausgabewerk

Was ist Informatik Historische Meilensteine (3)



Konrad Zuse (1910-1995) und die Z3 im deutschen Museum



© Prof. Dr. Dieter Nazareth 2023

17

Was ist Informatik Historische Meilensteine (4)



- ~1965 Informatik entsteht als wissenschaftliche Disziplin
- 1969 USA: Intel
Integrierter Prozessorchip mit Steuer- und Rechenwerk,
2300 Transistoren, 108 kHz, bearbeitet 4 Bit gleichzeitig
- 1990 Weltweit
Internet wandelt sich vom reinen Wissenschaftsnetz
zur globalen Kommunikationsplattform
- Heute IT ist in der industrialisierten Welt fast überall,
in Wirtschaft, Technik und im Privatleben
 - Hardware wird immer schneller, kleiner, billiger;
z.B. Pentium 4 Prozessor von Intel, 2004:
3-4 GHz, 125 Mio. Transistoren in 90nm-Technologie
 - Software wird immer komplexer und umfangreicher,
und damit auch zunehmend sicherheitskritisch

© Prof. Dr. Dieter Nazareth 2023

18

Was ist Informatik Moore's Law



Moore's Law: The number of transistors on microchips doubles every two years.

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years.

This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.

Transistor count

50,000,000,000

10,000,000,000

5,000,000,000

1,000,000,000

500,000,000

100,000,000

50,000,000

10,000,000

5,000,000

1,000,000

500,000

100,000

50,000

10,000

5,000

1,000

100

10

1



Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)

Our World in Data

19

© Prof. Dr. D

Licensed under CC-BY by the authors Hannah Ritchie and Max Roser.

OurWorldInData.org

– Research and data to make progress against the world's largest problems.

Was ist Informatik Rechnerbeispiele



© Prof. Dr. Dieter Nazareth 2023

20

Was ist Informatik Rechnerbeispiele



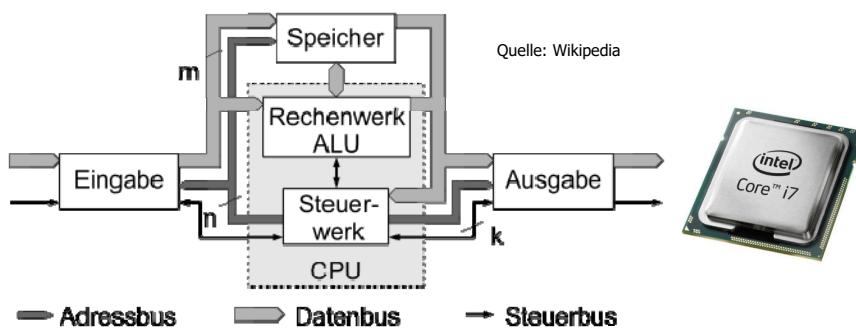
© Prof. Dr. Dieter Nazareth 2023

21

Was ist Informatik Von-Neumann-Architektur



Die **Von-Neumann-Architektur** (VNA) ist ein Referenzmodell für Computer, wonach ein gemeinsamer Speicher sowohl Computerprogrammbefehle als auch Daten hält. Die Von-Neumann-Architektur bildet die Grundlage für die Arbeitsweise der meisten heute bekannten Computer. Sie ist benannt nach dem österreichisch-ungarischen, später in den USA tätigen Mathematiker John von Neumann, dessen wesentliche Arbeit zum Thema 1945 veröffentlicht wurde.



© Prof. Dr. Dieter Nazareth 2023

22

Grundlagen der Informatik Überblick



- Was ist Informatik
- Information und Repräsentation
- Algorithmen
- Programmiersprachen
- Datenstrukturen
- Komplexität
- Berechenbarkeit

Information und Repräsentation Überblick (1)

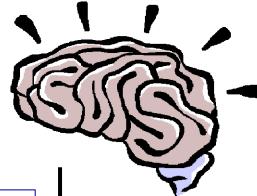


- Grundbegriffe
- Information im Rechner
- Begriffe
- Informationssystem
- Codierung
- Natürliche Zahlen
 - Darstellung
 - Umwandlung zwischen den Darstellungen
 - Addition
- Ganze Zahlen
 - Vorzeichendarstellung
 - Zweierkomplementdarstellung
- Text
- Wahrheitswerte

Information und Repräsentation Grundbegriffe – Zusammenhänge



Problem und
Beschreibung
mentales Model

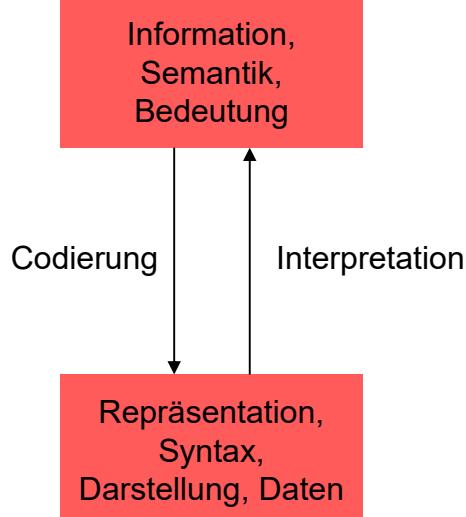


**völlig andere
Abstraktionsebenen**

ausführbares
Programm
Maschinenebene

```
01010110110  
01101001011  
10010110100  
10111011110  
00010011110  
10001001010
```

Information und Repräsentation Grundbegriffe – Zusammenhänge



Information und Repräsentation

Grundbegriffe – Definitionen



Definition *Information*

- Gehalt einer Aussage, Nachricht, Beschreibung
 - Synonyme: Bedeutung, *Semantik*

Definition *Repräsentation*

- Äußere Form der Darstellung einer Information
 - Synonym: Darstellung, Daten, *Syntax*

Definition *Interpretation/Abstraktion*

- Deutung der Darstellung, Übergang zur (abstr.) Information
 - Findet häufig nur gedanklich statt

Definition *Codierung*

- Übergang von der Information zur Repräsentation

© Prof. Dr. Dieter Nazareth 2023

27

Information und Repräsentation

Grundbegriffe – Beispiele



Zahlen

- Information: „sechs“
 - Mögliche Repräsentationen:



၁ ၃ ၅ ၇ ၉
 ၁၂၃၄၆၈၉
 ၀၁၂၃၄၅၆၇၈၉
 ၀၂၃၄၅၆၇၈၉၀
 ၀၁၂၃၄၅၆၇၈၉၀
 ၀၁၂၃၄၅၆၇၈၉၀

Beschriftungen

- Information: "Hier ist eine Toilette."
 - Mögliche Repräsentationen: OO, WC.



© Prof. Dr. Dieter Nazareth 2023

28

Römische Zahlen

I	1		X	10		C	100
II	2		XX	20		CC	200
III	3		XXX	30		CCC	300
IV	4		XL	40		CD	400
V	5		L	50		D	500
VI	6		LX	60		DC	600
VII	7		LXX	70		DCC	700
VIII	8		LXXX	80		DCCC	800
IX	9		XC	90		CM	900
						M	1000

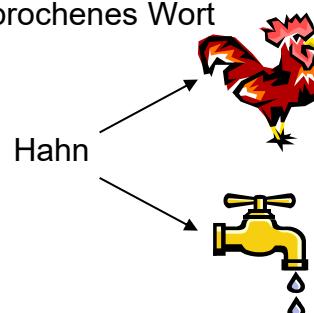
MMXXIII

2023 als römische Zahl

Information lässt sich nicht direkt aufschreiben!

**Wie muss Information dargestellt sein, damit sie
maschinell verarbeitet werden kann?**

- Technisch schematisch auswertbar
Gegenbeispiele: Rauchzeichen, gesprochenes Wort
- Präzision
 - Verschiedene Information sollte
verschieden dargestellt werden.
 - Eine Repräsentation / ein Datum
sollte genau eine mögliche
Interpretation haben.



Natürliche Sprache erfüllt diese Forderungen i. A. nicht!

WARUM ES SO SCHWER IST EIN PROGRAMMIERER ZU SEIN

Meine Mutter sagte:

„Bist du so lieb und gehst für mich zum Supermarkt.
Kaufe dort eine Flasche Milch. Wenn sie Eier haben,
bringe sechs mit.“

Nun, ich kam mit sechs Flaschen Milch zurück.

Meine Mutter sagte:

„Warum zum Teufel, bringst du mir sechs Flaschen Milch.“

Meine Antwort:

„Weil sie EIER hatten!“

Übersetzung Markus Brandl

Definition **Bit** (Synonym: **Binärzeichen**)

- Kleinste mögliche Einheit von Information
- Zwei mögliche Werte: 0 / 1, true / false, schwarz / weiß
- Informationsmenge zum Ausdrücken von 2 Möglichkeiten
- Wortursprung: Binary Digit

Technische Darstellung

- Elektrische Spannung (Arbeitsspeicher):
0 = 0 Volt, 1 = 5 bzw. 2,5 Volt
- Magnetisierung (Festplatte):
0 = unmagnetisiert, 1 = magnetisiert
- Optisch durch Höhenniveaus (CD, DVD):
0 = Gleichbleibendes Niveau, 1 = Niveauwechsel

Wie werden Informationen dargestellt, die mehr als zwei Möglichkeiten zulassen?

Beispiel: Himmelsrichtungen

- Repräsentation durch 2 Bits
Süd = 00, West = 01, Nord = 10, Ost = 11

Beispiel: Himmelsrichtungen mit Zwischenrichtungen

- 4 neue Richtungen dazu, dargestellt durch ein weiteres Bit
- Jetzt (eine Möglichkeit von vielen)
Süd = 000, West = 001, Nord = 010, Ost = 011
NO = 100, SO = 101, SW = 110, NW = 111

Beobachtung

- 1 Bit mehr erlaubt doppelt so viele verschiedene Bitfolgen
- Allgemein:
Es gibt 2^n mögliche verschiedene Bitfolgen der Länge n
- Rechner arbeiten nicht auf einzelnen Bits (wäre zu langsam)
- Zusammenfassen von Bits zu längeren Abschnitten
 - 4 Bit = Nibble, Halb-Byte
 - 8 Bit = **Byte**
- Rechner arbeiten immer mit Bitblöcken der Länge $n*8$
- 8-Bit-, 16-Bit-, 32-Bit-, 64-Bit-Rechner

Definitionen

- **Zeichenvorrat** =
endliche Menge von unterscheidbaren Elementen
- **Zeichen** = Element in einem Zeichenvorrat
- **Alphabet** = Zeichenvorrat mit linearer Ordnung
(d.h. für die Zeichen ist eine Reihenfolge definiert)

Beispiele

- Römische Ziffern {I, V, X, L, C, D, M} ← Alphabet 1
- Lateinische Kleinbuchstaben {a, b, c, ..., z} ← Alphabet 2
- Ampelfarben {red, yellow, green} ← Zeichenvorrat 3
- Farben von Spielkarten {red, green, blue, black} ← Alphabet 4
- Wahrheitswerte {true, false} ← binärer Zeichenvorrat 5

Definitionen

- **Wort** = Folge von Zeichen aus einem Zeichenvorrat
- Synonyme: **Zeichenreihe, -folge, -kette, -sequenz**
- ϵ = Leere Zeichenreihe
- **Konkatenation**, Operator \circ =
Aneinanderhängen zweier Zeichenreihen
- Z^* = Menge aller Worte über dem Zeichenvorrat Z
- Z^+ = Menge aller nichtleeren Worte über Zeichenvorrat Z

Beispiele

- MMVI \in (Alphabet 1) $^+$
- infor, matik, infor O matik = informatik \in (Alphabet 2) $^+$
- ϵ ist die leere Zeichenkette über allen Zeichenvorräten

Ziel

- Schematisierte, formale Darstellung von Informationen durch „Zeichenreihen“ (digitale Repräsentationen)

Formale Definition *Interpretation*

- Sei R eine Menge von Repräsentationen
- Sei A eine Menge von Informationen
- Dann ist die Interpretation eine Abbildung $I: R \rightarrow A$

Begriffe

- R heißt *Repräsentationssystem*
- A heißt *semantisches Modell*
- (A, R, I) heißt *Informationssystem*

Gegeben

- Menge der natürlichen Zahlen: $N = \{0, 1, 2, 3, \dots, 10, 11, \dots\}$
- Alphabet $A = \{0, 1, 2, 3, \dots, 9\}$

Gesucht

- Informationssystem für N über A , also $I: A^+ \rightarrow N$

Beispiele für die Arbeitsweise von I :

- | | | |
|--------------|---------------|------------------|
| ▪ $I(0) = 0$ | ▪ $I(00) = 0$ | ▪ $I(010) = 10$ |
| ▪ $I(1) = 1$ | ▪ $I(01) = 1$ | ▪ $I(100) = 100$ |
| ▪ $I(2) = 2$ | ▪ $I(02) = 2$ | ▪ $I(002) = 2$ |
| ▪ $I(3) = 3$ | ▪ $I(03) = 3$ | ▪ $I(375) = 375$ |

2, 02 und 002 sind *semantisch äquivalent*

Gegeben

- Menge der natürlichen Zahlen: $N = \{0, 1, 2, 3, \dots\}$
- Alphabet $A = \{0, 1\}$

Gesucht

- Informationssystem für N über A , also $I: A^+ \rightarrow N$

Beispiele für die Arbeitsweise von I :

- | | | |
|----------------|----------------|-----------------------|
| ▪ $I(0) = 0$ | ▪ $I(10) = 2$ | ▪ $I(111) = 7$ |
| ▪ $I(1) = 1$ | ▪ $I(11) = 3$ | ▪ $I(1000) = 8$ |
| ▪ $I(00) = 0$ | ▪ $I(100) = 4$ | ▪ $I(10000) = 16$ |
| ▪ $I(01) = 1$ | ▪ $I(101) = 5$ | ▪ $I(11111) = 31$ |
| ▪ $I(000) = 0$ | ▪ $I(110) = 6$ | ▪ $I(11111111) = 255$ |

Gegeben

- Menge der natürlichen Zahlen: $N = \{0, 1, 2, 3, \dots\}$
- Alphabet $A = \{0, 1, 2, \dots, 9, A, B, C, D, E, F\}$

Gesucht

- Informationssystem für N über A , also $I: A^+ \rightarrow N$

Beispiele für die Arbeitsweise von I :

- | | | |
|---------------|---------------|-----------------|
| ▪ $I(0) = 0$ | ▪ $I(B) = 11$ | ▪ $I(10) = 16$ |
| ▪ $I(1) = 1$ | ▪ $I(C) = 12$ | ▪ $I(11) = 17$ |
| ▪ $I(01) = 1$ | ▪ $I(D) = 13$ | ▪ $I(1F) = 31$ |
| ▪ $I(9) = 9$ | ▪ $I(E) = 14$ | ▪ $I(20) = 32$ |
| ▪ $I(A) = 10$ | ▪ $I(F) = 15$ | ▪ $I(FF) = 255$ |

Information und Repräsentation

Informationssystem – Feststellungen



- Binäre Darstellung
 - Ein Bit kann 2 verschiedene Werte annehmen: 0, 1
 - Gut geeignet für unmittelbare Umsetzung im Rechner, da leicht umsetzbar in elektrische Impulse etc.
- Hexadezimale Darstellung
 - Eine Hexstelle kann 16 verschiedene Werte annehmen
 - Entspricht dem Informationsgehalt von 4 Bit, d.h. 1 Nibble
 - Jeder Byte-Wert darstellbar durch 2 Hexstellen: 00, ..., FF
 - Technologisch effizient im Rechner umsetzbar
 - Für Mensch leichter erfassbar als binäre Zahlen
0100111011000010110110001101100 =
0100 1111 0110 0001 0110 1100 0110 1100 = 4 F 6 1 6 C 6 C
- Mensch interpretiert binäre bzw. hexadezimale Daten, um auf Bedeutung im Dezimalsystem zu kommen

Information und Repräsentation

Informationssystem – Warnungen



Achtung

- Zeichenfolge alleine hat keine Bedeutung
- Zur Interpretation muss Informationssystem bekannt sein!

Beispiel: Gilt 111 = 111?

- Ja, falls gleiches Informationssystem zugrunde liegt
- Nein sonst
 - $I_2(111) = I_{10}(7) \neq I_{10}(111) = I_2(1101101)$
 - $I_{16}(111) = I_{10}(273) \neq I_{10}(111) = I_{16}(6F)$

Also: Unbedingt Informationssystem angeben!

- $I_2()$: binär nach dezimal, $I_{16}()$: hexadezimal nach dezimal
- $I_{10}()$: dezimal nach dezimal (Identität)

Motivation

- Für jede Information sind verschiedene Darstellungsformen möglich
- Darstellungsformen je nach Verwendungszweck unterschiedlich gut geeignet
- Beispiele
 - Für Menschen: Dezimalzahlen, Programmiersprachen
 - Für Rechner: Binärzahlen, Maschinensprache

Benötigt werden...

- Baukasten von Darstellungsformen für verschiedene Zwecke
- Möglichkeiten für Umwandlung zwischen den Darstellungen

Definition **Codierung, Code**

- **Codierung** = Die Abbildung zwischen den Zeichen zweier Zeichenvorräte bzw. zwischen den Wörtern über zwei Zeichenvorräten heißt Codierung.
- **Code** = Bildmenge einer solchen Codierungsabbildung

Üblicherweise gefordert

- Codierter Wert soll wieder in den ursprünglichen Wert zurückgewandelt werden können
- D.h. Codierung soll umkehrbar sein
- Codierung muss *injektiv* (eineindeutig) sein

Definition **Decodierung**

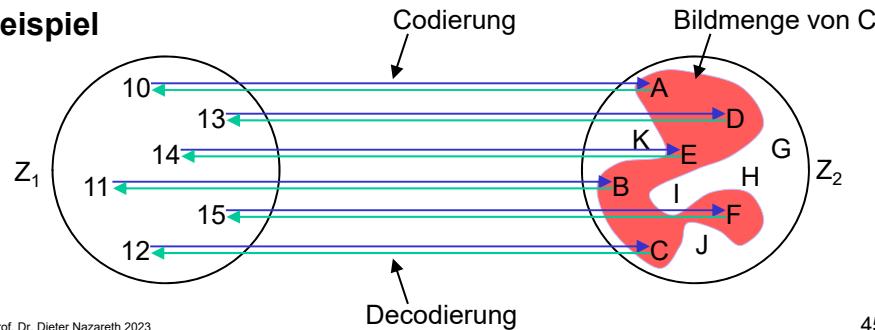
- **Decodierung** = Umkehrabbildung einer Codierung

Information und Repräsentation Codierung – Begriffe formal



- Zeichenvorräte Z_1, Z_2
- Z_1^* bzw. Z_2^* Menge aller Worte über Z_1 bzw. Z_2
- Codierung der einzelnen Buchstaben: $C: Z_1 \rightarrow Z_2$
- Codierung ganzer Wörter: $C: Z_1^* \rightarrow Z_2^*$
- Decodierung: $C^{-1}: Z_2 \rightarrow Z_1$ bzw. $C^{-1}: Z_2^* \rightarrow Z_1^*$

Beispiel



© Prof. Dr. Dieter Nazareth 2023

45

Information und Repräsentation Codierung – Arten von Codes



Klassifizierungen

- Nach dem Zeichenvorrat des Wertebereichs
 - Z.B. Binärcodes, für Informatik besonders relevant
- Nach der Länge des Codes
 - Codes mit variabler Länge
 - Codewörter sind unterschiedlich lang
 - Häufige vorkommende Wörter durch kurzes Codewort repräsentieren → Effizienz
 - Evtl. schwierig zu decodieren (Wo ist Wortgrenze?)
 - Codes mit fester Länge
 - Alle Codewörter umfassen gleich viele Zeichen
 - Technisch meist leichter zu realisieren

© Prof. Dr. Dieter Nazareth 2023

46

Information und Repräsentation Codierung – Codes variabler Länge



Morsecode: Drei Zeichen ., -, ___, genannt „kurz“, „lang“, „Lücke“

Zeichen	Code
.	01
-	0111
__	000

Impulswählverfahren beim Telefon

Ziffer	Code
1	10
2	110
...	...
9	1111111110

© Prof. Dr. Dieter Nazareth 2023

47

Information und Repräsentation Codierung – Codes fester Länge



Informationsverarbeitung in Rechenanlagen

- Praktisch immer Binärcodierung
- Meist Codes fester Länge

Beispiele

- Direkte Binärcodierung für natürliche Zahlen, 8 bzw. 16 Bit
- Höchstwertiges Bit ist ganz links, niederwertigstes Bit rechts

Zahl	8-Bit-Code	Zahl	16-Bit-Code
0	00000000	0	0000000000000000
1	00000001	1	0000000000000001
2	00000010	2	0000000000000010
...
255	11111111	65535	1111111111111111

© Prof. Dr. Dieter Nazareth 2023

48

Falls die Codierung einer Information aus dem Kontext heraus nicht zweifelsfrei klar, muss sie explizit deutlich gemacht werden (analog zu Informationssystem)!

Schreibweisen

- Dezimaldarstellung: $(4711)_{10}$, $(29)_{10}$, $(1001)_{10}$
- Binärdarstellung: $(1001001100111)_2$, $(11101)_2$, $(1001)_2$
- Hexadezimaldarstellung: $(1267)_{16}$, $(1D)_{16}$, $(1001)_{16}$
- Konvention: Standardwert ist Dezimalsystem!

Abgrenzung Interpretation, Decodierung

- Interpretation ermittelt die Bedeutung
- Decodierung wandelt um in andere Darstellungsform

Stellenwertsystem am Beispiel des Dezimalsystems

- Ziffern können Werte aus $\{0, 1, \dots, 9\}$ annehmen
- Ziffern einer Dezimalzahl sind Koeffizienten von Zehnerpotenzen
- Exponent = Position der Ziffer von rechts gezählt – 1

Koeffizient → $a^b c$
Exponent
Basis

Beispiel

$$\begin{aligned} 4711 &= 4 * 10^3 + 7 * 10^2 + 1 * 10^1 + 1 * 10^0 = \\ &= 4 * 1000 + 7 * 100 + 1 * 10 + 1 * 1 = \\ &= 4000 + 700 + 10 + 1 \end{aligned}$$

Stellenwertsystem für Binärdarstellung

- Ziffern können Werte 0 oder 1 annehmen
- Ziffern der Binärzahl sind Koeffizienten von Zweierpotenzen
- Exponent = Position der Ziffer von rechts gezählt – 1

Beispiel

$$\begin{aligned} I_2(1101) &= 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 \\ &= 1 * 8 + 1 * 4 + 0 * 2 + 1 * 1 \\ &= 8 + 4 + 0 + 1 = I_{10}(13) \end{aligned}$$

Zahlenraum

- n Binärstellen beschreiben den Zahlenraum von 0 bis $2^n - 1$
- 4 Bit $\cong \{0, \dots, 15\}$; 8 Bit $\cong \{0, \dots, 255\}$; 16 Bit $\cong \{0, \dots, 65535\}$

Stellenwertsystem für Hexadezimaldarstellung

- Ziffern können Werte aus {0, 1, ..., 9, A, ..., F} annehmen
- Ziffern der Hexzahl sind Koeffizienten von Potenzen von 16
- Exponent = Position der Ziffer von rechts gezählt – 1

Beispiel

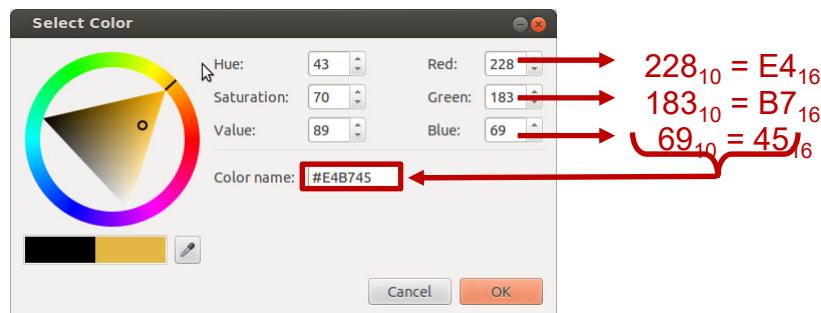
$$\begin{aligned} I_{16}(2C73) &= 2 * 16^3 + 12 * 16^2 + 7 * 16^1 + 3 * 16^0 = \\ &= 2 * 4096 + 12 * 256 + 7 * 16 + 3 * 1 = \\ &= 8192 + 3072 + 112 + 3 = I_{10}(11379) \end{aligned}$$

Zahlenraum

- n Hexstellen beschreiben den Zahlenraum von 0 bis $16^n - 1$
- 1 Hexstelle $\cong \{0, \dots, 15\}$; 2 Hexstellen $\cong \{0, \dots, 255\}$

Stellenwertsystem für Hexadezimaldarstellung

Kompakte Darstellung von Farben durch Hexzahlen



Stellenwertsystem allgemein für Basis B

Ein Stellenwertsystem mit der Basis B ist ein Zahlensystem, in dem eine Zahl x nach Potenzen von B zerlegt wird.

Eine natürliche Zahl n wird durch folgende Summe dargestellt:

$$n = \sum_{i=0}^{N-1} b_i \cdot B^i$$

Wobei folgendes gilt:

- B = Basis des Zahlensystems ($B \in \mathbb{N}$, $B \geq 2$)
- b = Ziffern ($b_i \in \mathbb{N}_0$, $0 \leq b_i < B$)
- N = Anzahl der Stellen

Stellenwertsystem allgemein für Basis B

Beispiel

$$I_B(b_4 b_3 b_2 b_1 b_0) = b_4 * B^4 + b_3 * B^3 + b_2 * B^2 + b_1 * B^1 + b_0 * B^0 = \\ = \sum_{i=0}^4 b_i * B^i$$

Zahlenraum:

n Stellen beschreiben den Zahlenraum von 0 bis $B^n - 1$

Namen für einige Zahlensysteme:

- B=2: Dualsystem
- B=8: Oktalsystem
- B = 10: Dezimalsystem
- B = 16: Hexadezimalsystem

**Tabelle für die Zahlendarstellung
in fünf verschiedenen Zahlensystemen**

Dual	Oktal	Dezimal	Hexadezimal	Zwölfersystem
0	0	0	0	0
1	1	1	1	1
10	2	2	2	2
11	3	3	3	3
100	4	4	4	4
101	5	5	5	5
110	6	6	6	6
111	7	7	7	7
1000	10	8	8	8
1001	11	9	9	9
1010	12	10	a	a
1011	13	11	b	b
1100	14	12	c	10
1101	15	13	d	11
1110	16	14	e	12
1111	17	15	f	13
10000	20	16	10	14
10001	21	17	11	15



Aufgabe

- Gegeben: natürliche Zahlen z und d , wobei $d \neq 0$
- Zu tun: Teile z durch d
- Ergebnis:
 - Quotient q , Rest r , mit $q, r \in \mathbb{N}$
 - Es gilt dabei: $z = q * d + r$ mit $0 \leq r < d$

Operationen für die ganzzahlige Division in der Informatik

- div ermittelt ganzzahligen Quotienten q , also $z \text{ div } d = q$
- mod ermittelt Divisionsrest r , also $z \text{ mod } d = r$
- Es gilt: $z = (z \text{ div } d) * d + (z \text{ mod } d)$

Aufgabe

- Wandle natürliche Zahl z aus Dezimaldarstellung in Binärdarstellung um
- D.h. gesucht ist $b_n b_{n-1} \dots b_1 b_0$ mit $I_2(b_n b_{n-1} \dots b_1 b_0) = z$

Überlegung

$$\begin{aligned} z &= I_2(b_n b_{n-1} \dots b_1 b_0) = \\ &= b_n * 2^n + b_{n-1} * 2^{n-1} + \dots + b_1 * 2^1 + b_0 = \quad (2 \text{ Ausklammern}) \\ &= (b_n * 2^{n-1} + b_{n-1} * 2^{n-2} + \dots + b_1) * 2^1 + b_0 = \\ &= I_2(b_n b_{n-1} \dots b_1) * 2^1 + b_0 \end{aligned}$$

Vergleich mit vorheriger Seite: $z = (z \text{ div } d) * d + (z \text{ mod } d)$

Es gilt also: $I_2(b_n b_{n-1} \dots b_1) = z \text{ div } 2$ und $b_0 = z \text{ mod } 2$

Was wissen wir jetzt?

- Rechtestes Bit b_0 hat den Wert $z \text{ mod } 2$
- Binärdarstellung von $z_1 = z \text{ div } 2$ ergibt die Folge der ersten Ziffern $b_n b_{n-1} \dots b_1$
- Entsprechend:
 - $b_1 = z_1 \text{ mod } 2$
 - Binärdarstellung von $z_2 = z_1 \text{ div } 2$ ergibt $b_n b_{n-1} \dots b_2$
 - ...
 - Wenn wir das "lange genug" machen, bekommen wir alle b_i
 - Wir sind fertig, wenn $z_i \text{ div } 2 = 0$ ergibt

Vorgehensweise funktioniert analog für andere Basis als 2

Information und Repräsentation

Natürliche Zahlen – Umwandlung Dezimal nach Basis d



Verfahren allgemein für Umwandlung nach Basis d

- Gegeben: z, d mit $d \neq 0$
- Setze Schreibmarke im Ergebnisfeld ganz nach rechts
- Wiederhole
 - $b = z \bmod d$
 - Gib b aus
 - Setze Schreibmarke ein Feld weiter nach links
 - Setze z auf $z \div d$
 - Breche ab, wenn $z = 0$

Information und Repräsentation

Natürliche Zahlen – Umwandlung – Beispiel



Umwandlung der Dezimalzahl 4711 ins Binärsystem

z	z div 2	z mod 2
4711	2355	1
2355	1177	1
1177	588	1
588	294	0
294	147	0
147	73	1
73	36	1
36	18	0
18	9	0
9	4	1
4	2	0
2	1	0
1	0	1

Ergebnis: $(4711)_{10} = (1001001100111)_2$

Stellenwertsystem bei gebrochenen Zahlen

Bei gebrochenen Zahlen trennt ein Punkt (Komma im Deutschen) in der Zahl den ganzzahligen Teil der Zahl vom gebrochenen Teil (Nachkommateil). Solche Zahlen lassen sich durch folgende Summenformel beschreiben:

$$n = \sum_{i=-M}^{N-1} b_i \cdot B^i$$

Wobei folgendes gilt:

- B = Basis des Zahlensystems ($B \in \mathbb{N}, B \geq 2$)
- b = Ziffern ($b_i \in \mathbb{N}_0, 0 \leq b_i < B$)
- N = Anzahl der Stellen vor dem Komma
- M = Anzahl der Stellen nach dem Komma

Stellenwertsystem bei gebrochenen Zahlen

Beispiele:

$$\begin{aligned}(17.05)_{10} &= 1 \cdot 10^1 &+& 7 \cdot 10^0 &+& 0 \cdot 10^{-1} &+& 5 \cdot 10^{-2} \\(3758.0)_{10} &= 3 \cdot 10^3 &+& 7 \cdot 10^2 &+& 5 \cdot 10^1 &+& 8 \cdot 10^0 \\(9.702)_{10} &= 9 \cdot 10^0 &+& 7 \cdot 10^{-1} &+& 0 \cdot 10^{-2} &+& 2 \cdot 10^{-3} \\(0.503)_{10} &= 0 \cdot 10^0 &+& 5 \cdot 10^{-1} &+& 0 \cdot 10^{-2} &+& 3 \cdot 10^{-3}\end{aligned}$$

Generell

- Arithmetische Operationen in Binär- und Hexadezimalsystem analog zu Dezimalsystem
- Fokus hier: Addition

Bearbeitungsschema für Addition zweier Zahlen

- Zahlen in Ziffernspalten übereinander schreiben
- Ziffern einer Spalte aufsummieren
- Übertrag ggf. zur nächst höheren Ziffernposition addieren

Übertrag

- Bei Addition entsteht ein Wert, der größer oder gleich der Basis ist

Addition in verschiedenen Zahlensystemen

Dezimal	Binär	Oktal	Hexadezimal
4711	1001001100111	4711	4711
+ 2606	+ 101000101110	+ 2606	+ 2606
7317	1110010010101	7517	6D17

Achtung

- In Dezimal-, Oktal- und Hexadezimalsystem jeweils gleiche Ziffernfolge addiert
- Ergebnisse haben jeweils unterschiedliche Ziffernfolgen

Definition *Übertrag*

- Summe einer Ziffernspalte ist größer oder gleich Basiswert
- Überhang wird auf nächst höheres Bit übertragen

Was passiert, wenn dieser Übertrag beim höchsten Bit auftritt und nicht mehr in das Zahlenformat passt?

Beispiel: Addition von natürlichen Zahlen

Dezimal	8-Bit-Binär	Hexadezimal
197	11000101	C5
+ 213	+ 11010101	+ D5
410	110011010	19A

- Übertrag im 8. Bit der 8-Bit-Binärdarstellung
- Liefert als Ergebnis $(10011010)_2$, d.h. 154



67

- **Carry:** Englisch für Übertrag
- **Carry-Flag:**
Kontrollbit im Prozessor, das anzeigt, dass bei einer arithmetischen Operation beim höchstwertigen Bit ein Übertrag auftritt; Ergebnis passt nicht ins Zahlenformat

Bei Überschreiten des Zahlenformates

- Prozessor liefert das "oben" abgeschnittene Ergebnis und setzt das Carry-Flag

Maßnahmen

- Ignorieren? Weiterrechnen mit falschem Wert
- Afangen!!! (Entweder über die Programmiersprache, oder explizit durch den Programmierer!)

Bedeutung

- Ganze Zahlen sind die natürlichen und die negativen Zahlen
- Beschreibung einer ganzen Zahl
 - Absoluter Zahlenwert
 - Vorzeichen + oder -

Darstellung ganzer Zahlen

- Benötigt im Vergleich zu natürlichen Zahlen ein weiteres Bit an Information
- Verschiedene Varianten möglich
 - Direkte Codierung des Vorzeichens, Vorzeichendarstellung
 - Darstellung als Zweierkomplement

Idee der Vorzeichendarstellung

- Absolutwert der Zahl wie bisher codieren
- Vorne dran ein Bit für das Vorzeichen stellen (0 für +, 1 für -)
- Bei n Bit ist das Intervall $\{-2^{n-1}+1, \dots, +2^{n-1}-1\}$ darstellbar

Mögliche Werte bei Binärcode mit fester Länge 4

0000 = +0	0100 = +4	1000 = -0	1100 = -4
0001 = +1	0101 = +5	1001 = -1	1101 = -5
0010 = +2	0110 = +6	1010 = -2	1110 = -6
0011 = +3	0111 = +7	1011 = -3	1111 = -7

Nachteile dieser Darstellung

- Null tritt doppelt auf: 0000 = +0 und 1000 = -0
- Rechnen ist schwieriger: gewohnte Addition funktioniert nicht

Addition

- Rechnen mit positiven Zahlen 4-Bit-Binär VZ Dezimal
funktioniert wie gewohnt
- | | | |
|--------|---|------|
| 0010 | = | +2 |
| + 0101 | = | + +5 |
| 0111 | = | +7 |

Subtraktion

- Bei negativen Zahlen liefert gewohntes Schema falsche Ergebnisse
 - Anderer Mechanismus erforderlich
- | | | |
|--------|---|------|
| 1010 | = | -2 |
| + 0101 | = | + +5 |
| 1111 | ≠ | +3 |

Für Addition und Subtraktion andere Rechenwerke nötig!

Idee der Darstellung als *Zweierkomplement*

- Mit n Bit sind 2^n Zahlen darstellbar
- Bereich frei wählbar, z.B. das Intervall $\{-2^{n-1}, \dots, +2^{n-1}-1\}$
- Codierung der natürlichen Zahlen wie üblich durch Hochzählen bis zur oberen Grenze des Intervalls
- Dann von der Untergrenze aus weiterzählen bis -1

Vorteile

- Nur eine Codierung der Null
- Erstes Bit zeigt Vorzeichen an; Null gilt als positiv
- Gewohnte einfache Arithmetik für + und -

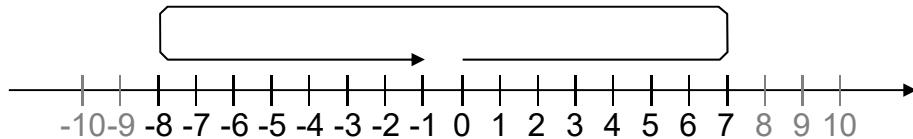
Nachteil

- Asymmetrischer Zahlenbereich $\{-2^{n-1}, \dots, +2^{n-1}-1\}$

Information und Repräsentation Ganze Zahlen – Zweierkomplement – Bedeutung



Anschaulich für n = 4



Mögliche Werte bei Binärcode mit fester Länge 4

0000 = +0	0100 = +4	1000 = -8	1100 = -4
0001 = +1	0101 = +5	1001 = -7	1101 = -3
0010 = +2	0110 = +6	1010 = -6	1110 = -2
0011 = +3	0111 = +7	1011 = -5	1111 = -1

Bedeutung

$$I_{2k}(b_n b_{n-1} \dots b_1 b_0) = b_n * (-2^n) + b_{n-1} * 2^{n-1} + \dots + b_1 * 2^1 + b_0$$

© Prof. Dr. Dieter Nazareth 2023

73

Information und Repräsentation Ganze Zahlen – Zweierkomplement – Bitweises Komplement



Definition **Bitweises Komplement**

- Wandle jedes Zeichen eines Binärworts in sein Gegenteil
- Also bitweise Negation NOT_{Bit}
- Beispiel: NOT_{Bit}(11010001) = 00101110

Idee hinter der Idee des Zweierkomplements

- Sei z eine Zahl in Zweierkomplementdarstellung
- Dann gilt: z + NOT_{Bit}(z) = (111...111)_{2k} = -1
- Bilden der Negativen einer Zahl z ist sehr (!) einfach:
Ermittle bitweises Komplement von z und addiere 1 dazu

Beispiel

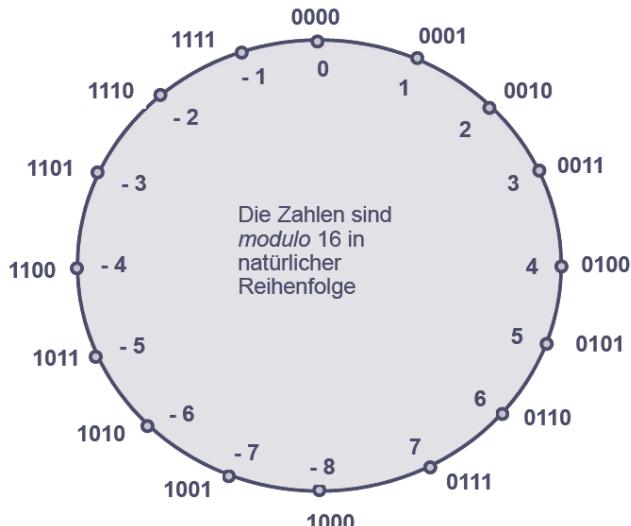
- $(7)_{10} = (0111)_{2k}$ $(-7)_{10} = (1000)_{2k} + (0001)_{2k} = (1001)_{2k}$

© Prof. Dr. Dieter Nazareth 2023

74

Information und Repräsentation

Ganze Zahlen – Zweierkomplement – Bitweises Komplement



© Prof. Dr. Dieter Nazaren 2023

75

Information und Repräsentation

Ganze Zahlen – Zweierkomplement – Addition



4-Bit ZK	Dezimal	4-Bit ZK	Dezimal	4-Bit ZK	Dezimal
0010	= 2	1100	= -4	1100	= -4
+ 0101	= + 5	+ 0010	= + 2	+ 1101	= + -3
0111	= 7	1110	= -2	11001	= -7

Addition

- Addition funktioniert im Zahlenbereich wie gewohnt!
- Trotz des Übertrags im dritten Beispiel ist Ergebnis innerhalb der 4 Bit korrekt; soll so sein, da Bereich nicht überschritten
- Falsches Ergebnis nur bei echter Bereichsüberschreitung:
bei der zyklischen Wertevergabe kommt beim Zählen in " - "-Richtung nach dem Wert -8 der Wert +7

4-Bit ZK	Dezimal
1100	= -4
+ 1011	= + -5

© Prof. Dr. Dieter Nazaren 2023

76

Subtraktion

- Statt $a - b$ rechne $a + (-b)$
- Beispiel: $2 - 6 = 2 + (-6) = -4$
- Verfahren:
 - Ermittle $-b$: Bilde bitweises Komplement von b , addiere 1
 - Addiere nach gewohntem Schema a und $-b$
 - Kein separates Rechenwerk für Subtraktion erforderlich

Beispiel: $2 - 5 = -3$

- $(5)_{10} = (0101)_2$
- $(-5)_{10} = (1010)_2 + (0001)_2 = (1011)_2$
- $(2)_{10} - (5)_{10} = (2)_{10} + (-5)_{10} =$
 $= (0010)_2 + (1011)_2 = (1101)_2 = (-3)_{10}$

Definition *Überlauf*

- "Echtes" Ergebnis einer arithmetischen Operation liegt nicht im Bereich der Zahldarstellung; d.h. Bereichsüberschreitung
- Übertrag im höchsten Bit ist Hinweis auf Überlauf
- Prozessor signalisiert Überlauf durch *Overflow-Flag*

Achtung

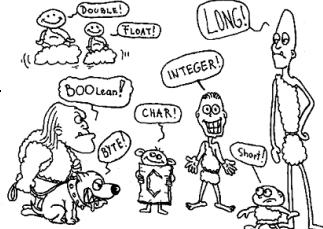
- Bei Zweierkomplementdarstellung ist NICHT jeder Übertrag im höchstwertigen Bit auch ein Überlauf
- Bei "normaler" Binärdarstellung für natürliche Zahlen ist JEDER Übertrag im höchstwertigen Bit ein Überlauf

Beispiel: $1100 + 1101 = 11001$

- Zweierkomplement: $-4 + -3 = -7$ Übertrag, kein Überlauf
- Binär: $12 + 13 = 25$ Übertrag und Überlauf

Fazit

- Additionsschema für Zweierkomplement genau wie Binär
- Gültigkeit des Ergebnisses aber abhängig von Codierung!



Theorie: Beliebige Zahlenformate sind möglich

Praxis

- Fast nur Zahlenformate mit 8, 16, 32, oder 64 Bit
- In Programmiersprachen meist Datentypen vordefiniert
- Format so groß wie nötig wählen, damit kein Überlauf auftritt
- Format so klein wie möglich wählen, um Platz zu sparen

Ganzzahlige Typen in Java

Intervall	Format	Java-Typ
$\{-128, \dots, 127\}$	8 Bit	byte
$\{-32768, \dots, 32767\}$	16 Bit	short
$\{-2^{31}, \dots, 2^{31}-1\}$	32 Bit	int
$\{-2^{63}, \dots, 2^{63}-1\}$	64 Bit	long

Idee

- Codieren aller benötigten Zeichen als Bitfolgen
 - Buchstaben a, ..., z, A, ..., Z
 - Satzzeichen ., , :, ?, ...
 - Mathematische Sonderzeichen {, }, (,), +, -, ...
 - Steuerzeichen <CR>, <TAB> (werden nicht angezeigt)
- Insgesamt ca. 100 Zeichen auf derzeitigen Tastaturen

Für die Codierung von 100 Zeichen werden 7 Bit benötigt.

Das erlaubt 128 verschiedene Zeichen.

Also: Jedem Zeichen einen 7-stelligen Bitcode zuordnen

American Standard Code for Information Interchange ASCII

- Es gibt 128 ASCII-Zeichen,
codiert durch die Bitfolgen 0000 0000 bis 0111 1111
- Codierung eines ASCII-Zeichens beginnt immer mit 0
- Ziffern 0 bis 9 in natürlicher Reihenfolge nummeriert,
 $0 \cong 48, 9 \cong 57$
- Großbuchstaben in alphabetischer Reihenfolge nummeriert,
 $A \cong 65, Z \cong 90$
- Kleinbuchstaben in alphabetischer Reihenfolge nummeriert,
 $a \cong 97, z \cong 122$
- Steuerzeichen liegen im ASCII-Bereich 0 bis 31
- Satzzeichen und mathematische Sonderzeichen liegen im
ASCII-Bereich 32 bis 47

Information und Repräsentation

Text – ASCII-Tabelle



<u>Code</u>	...0	...1	...2	...3	...4	...5	...6	...7	...8	...9	...A	...B	...C	...D	...E	...F
0...	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1...	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2...	SP	!	"	#	\$	%	&	'	()	*	+	,	-	.	/	
3...	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4...	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5...	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6...	~	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7...	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

Information und Repräsentation

Text – ASCII-Erweiterungen



- Früher: nicht benötigtes erstes Bit als Kontrollbit verwendet; so gesetzt, dass Anzahl der 1en gerade ist
- Heute: Datenübertragung weniger fehleranfällig; daher Nutzung des 8ten Bits zur Codierung weiterer Zeichen

ASCII-Erweiterungen:

- Verschiedene Varianten
- In Europa:
 - Latin-1 gemäß ISO 8859-1
 - Enthält die europäischen Sonderzeichen; ä, ö, ü, ß, ...
- Beim weltweiten Datenaustausch treten hier oft Probleme auf, wenn Start- und Zielrechner verschiedene Codierungen verwenden

Ziel

- Zeichensatzprobleme beim weltweiten Datenaustausch beheben
- Universeller Code für Schriften verschiedenster Kulturen

Unicode

- 16-Bit Codierung, d. h. maximal 65536 Zeichen
- Zeichen 0 bis 127 wie ASCII-Code
- Zeichen 128 bis 255 wie ISO Latin-1
- Java erlaubt Verwendung beliebiger Unicode-Zeichen (die meisten Programmiersprachen nur Standard-ASCII)
- Durch ISO 10646 international standardisiert als *Universal Character Set (UCS)*

Nachteil von Unicode

- Teilweise Kompatibilitätsprobleme
- Textdateien werden doppelt so lang wie bei ASCII

UCS Transformation Format (UTF)

- UTF-8 als eine Variante von Unicode
- Standardisiert
 - Anhang R der ISO 10646 Norm
 - RFC2279 der Internetgremien (Request for Comment)
- Standard-ASCII-Zeichen als 1 Byte codiert
- Andere Zeichen mit 2 bis 6 Byte
- Kompatibel zur historischen 7-Bit-ASCII-Codierung
- Neuer de-facto-Standard z. B. bei Webprogrammierung

Information und Repräsentation

Text – Zeichenketten



- Zur Codierung eines fortlaufenden Textes fügt man einfach die Codes der einzelnen Zeichen aneinander.
- Eine Folge von Textzeichen heißt auch **Zeichenkette** (engl. string). Der Text „Hallo Welt“ wird also durch die Zeichenfolge
 - H, a, l, l, o, , W, e, l, trepräsentiert.
- Mit ASCII-Codes sieht das dann hexadezimal so aus:
48 61 6C 6C 6F 20 57 65 6C 74
- Viele Programmiersprachen markieren das Ende einer Zeichenkette mit dem ASCII-Zeichen NUL = 00h. Man spricht dann von **nullterminierten Strings**.

Information und Repräsentation

Text – Zahlen im Text



- Auch innerhalb von Text können Zahlen auftreten
- Dargestellt gemäß dem Zeichensatz des restlichen Textes
- Braucht viel Speicherplatz
- Zum Rechnen ungeeignet

Beispiel:

- "4711" in einem ASCII-Text
- Gespeichert als Folge der entsprechenden ASCII-Codes
 - Dezimal: 052, 055, 049, 049
 - Binär: 0011 0100, 0011 0111, 0011 0001, 0011 0001
- Zum Vergleich: $I_{10}(4711) = I_2(1\ 0010\ 0110\ 0111)$

- In der Informatik arbeiten wir meist mit binärer Logik, d.h. eine Aussage kann entweder *wahr* oder *falsch* (*true*, *false*) sein.
- Repräsentiert werden diese beiden Werte auf vielfach Art und Weise: true, false; T, F; 1, 0; high, low; ...
- Da es nur zwei Wahrheitswerte gibt, könnte man diese durch die beiden möglichen Werte eines Bits darstellen, z.B. durch F = 0 und T = 1. Da aber ein Byte die kleinste Einheit ist, mit der ein Computer operiert, spendiert man meist ein ganzes Byte für einen Wahrheitswert. Eine gängige Codierung ist F = 0000 0000 und T = 1111 1111.
- In der Sprache C wird ein false durch die Zahl 0 kodiert und ein true durch eine beliebige ganze Zahl ungleich 0

- Diese Wahrheitswerte bilden zusammen mit Operationen die sog. *Boolesche Algebra*, die von dem Engländer *George Boole* entwickelt worden ist um die Logik zu formalisieren.
- Der Datentyp wird in den Programmiersprachen deshalb von *bool* oder *boolean* bezeichnet, der dann nur die Werte *true* und *false* enthält.
- Die wichtigste Operationen sind die Negation (NOT), die Konjunktion (AND) und die Disjunktion (OR).

NOT	F	T
F	F	T
T	T	F

AND	F	T
F	F	F
T	F	T

OR	F	T
F	F	T
T	T	T

Information und Repräsentation

Zusammenfassung – Grundlagen



- Information muss zur maschinellen Verarbeitung auf geeignete Weise dargestellt / repräsentiert werden
- Interpretation ermittelt aus Darstellung deren Bedeutung
- Codierung: Umwandlung zwischen Darstellungsformen
- Codierung sollte umkehrbar (injektiv, eineindeutig) sein
- Korrekte Interpretation einer Darstellung erfordert Kenntnis des Informationssystems bzw. der Codierung
- Repräsentation meist über Codes fester Länge
- Information im Rechner dargestellt als Bit bzw. Bitfolge
- Ein Bit codiert zwei mögliche Werte, z. B. 0 oder 1
- n Bit erlauben die Codierung von 2^n Werten
- Byte fasst 8 Bit zu einer Einheit zusammen; 256 Werte
- Wort: Folge von Zeichen aus einem Zeichenvorrat

Information und Repräsentation

Zusammenfassung – Zahlen



- Darstellung natürlicher Zahlen
 - Binärzahl, d.h. Stellwertsystem zur Basis 2
 - Hexadezimalzahl, d.h. Stellwertsystem zur Basis 16
 - Umrechnung: Eine Hexadezimalziffer entspricht vier Binärziffern
- Darstellung ganzer Zahlen
 - Über Zweierkomplement
 - Gleches Rechenwerk wie bei natürlichen Zahlen
- Überlauf: Beim Rechnen wird Zahlenbereich überschritten

Information und Repräsentation Zusammenfassung – Sonstiges



- Darstellung von Text
 - ASCII-Code: 7 Bit; (US-amerikanische) Standardzeichen
 - ASCII-Erweiterungen, z.B. Latin-1: 8 Bit
Landesspezifische Sonderzeichen, z. B. ä, ö, ü
 - Unicode: 16 Bit
 - UTF-8: Variable Länge, 8 Bit bis 6 Byte
- Darstellung von Wahrheitswerte

Grundlagen der Informatik I Überblick



- Was ist Informatik
- Information und Repräsentation
- Algorithmen
- Programmiersprachen
- Datenstrukturen
- Komplexität
- Berechenbarkeit

- Begriffe
- Informelle Algorithmen
- Formale Sprachen zur Beschreibung von Algorithmen
- Textersetzung
- Turingmaschine
- Eigenschaften von Algorithmen
- Grafische Notationen
- Flussdiagramm
- Struktogramm
- Rekursion
- Methodisches Vorgehen

Definition **Algorithmus**

- Verfahren (Vorschrift) zur Lösung eines Problems
- Definiert durch eine präzise, endliche Beschreibung
- Verwendet effektive, elementare Verarbeitungsschritte

Bedeutung

- Präzise: Beschreibungssprache genau definiert (mechanisch verstehbar)
- Endlich:
 - Beschreibung muss endlich sein, d. h. endet irgendwann
 - Verfahren an sich (die Ausführung) kann unendlich sein
- Effektiv: Mechanisch ausführbar (sehr vager Begriff)
- Elementar:
Einzelschritte sind einfach, überschaubar, klar verständlich

Zusammenhänge

- Eine Aufgabe kann von vielen verschiedenen Algorithmen gelöst werden
- Unterschied zwischen Aufgabenstellung und Algorithmus
 - Aufgabenstellung: Was ist zu tun?
 - Algorithmus: *Wie* ist etwas zu tun?
- Algorithmen sind stark abhängig von den zur Verfügung stehenden elementaren Schritten

Verwendung in der Programmentwicklung

- Klären, was eigentlich zu tun ist (Spezifikation der Aufgabe)
- Lösung entwerfen (Algorithmus, Datenstrukturen)
- Entwurf umsetzen und Lösung realisieren (*Programm*)

- Handlungsanweisungen nicht immer präzise
- Mögliche Gründe
 - Keine formale Beschreibungssprache verfügbar
 - Zielpublikum an formaler Beschreibung nicht interessiert (Haushaltskochrezept:
Fehlende Information ersetzt durch Erfahrung)
 - Aufwand für präzise Beschreibung sehr hoch
- Methodisch sind informelle Beschreibungen oft sinnvoll
- Problematisch ist jedoch die vielfältige Interpretierbarkeit;
Ergebnis ist nicht streng vorhersagbar!

Beispiel:
Rezept für Caipirinha

- ✓ Endlich
- ✓ Elementar
- x Präzise
- x Effektiv

CAIPIRINHA

1 Barschaufel gestoßenes Eis

1 Limette

1–2 Barlöffel Zucker

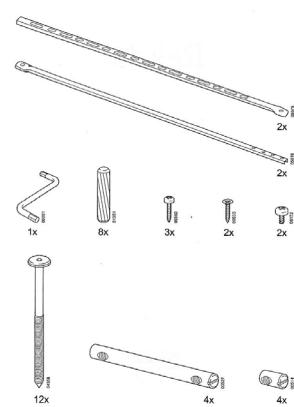
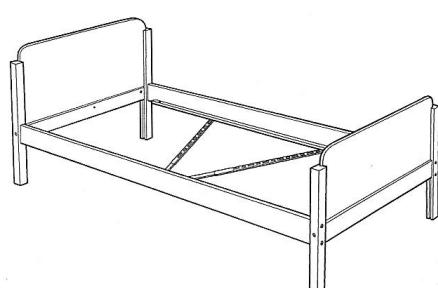
5 cl Cachaça (Pitú)

Die Limette vierteln, in einem Tumbler hineinpressen, Cachaça beifügen, Zucker nach Geschmack hineingeben, und die ausgepreßten Limettenviertel ebenfalls dazugeben. Umrühren, mit gestoßenem Eis auffüllen und nochmals verrühren.

Beispiel: Aufbauanleitung

Aufgabe

Baue aus dem beiliegenden (hoffentlich vollständigen) Haufen von Einzelteilen



... das abgebildete Objekt

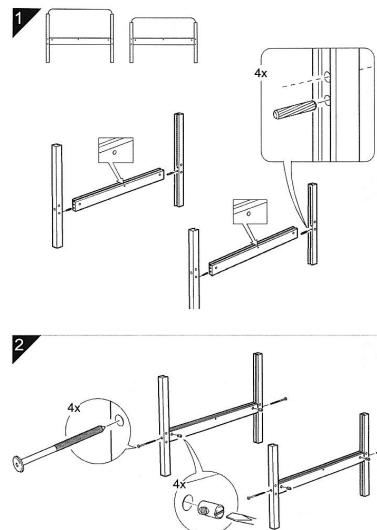
Algorithmen

Informelle Algorithmen – Aufbauanleitung (2)



Grafische Beschreibung der Handlungsvorschrift

- ✓ Endlich
- ✓ Elementar
- ✗ Präzise
- ✗ Effektiv



© Prof. Dr. Dieter Nazareth 2023

101

Algorithmen

Informelle Algorithmen – Mathematisches Verfahren (1)



Beispiel: Ermitteln des größten gemeinsamen Teilers zweier Zahlen

Aufgabe

- Gegeben: Zwei natürliche Zahlen a und b
- Gesucht:
Größter gemeinsamer Teiler von a und b , also $\text{ggT}(a, b)$

Algorithmus ggT nach Euklid

- Falls $a=b$, dann gilt: $\text{ggT}(a,b) = a$
- Falls $a < b$, dann wende Algorithmus auf a und $b-a$ an
- Falls $b < a$, dann wende Algorithmus auf $a-b$ und b an

© Prof. Dr. Dieter Nazareth 2023

102

Algorithmen

Informelle Algorithmen – Mathematisches Verfahren (2)



Beispiel für die Anwendung des Algorithmus

- $\text{ggT}(24,9) = \text{ggT}(15,9) = \text{ggT}(6,9) = \text{ggT}(6,3) = \text{ggT}(3,3) = 3$

Eigenschaften

- ✓ Endlich, ✓ Elementar, ✗ Präzise, ✗ Effektiv
- Algorithmus ist *rekursiv*
 - Definition des Algorithmus stützt sich auf sich selbst ab
 - Wichtig, damit rekursives Verfahren irgendwann endet:
Problem wird bei jedem rekursiven Aufruf des
Algorithmus ein bisschen einfacher

Algorithmen

Informelle Algorithmen – Sortieren von Spielkarten (1)



Selbstversuch:

Sortieren eines Stapels von n Spielkarten

Aufgabe

- Gegeben: Stapel s mit n Spielkarten
- Gesucht, d. h. als Ergebnis abzuliefern:
Stapel, in dem genau die gegebenen n Spielkarten in
aufsteigender Reihenfolge angeordnet sind

Definieren Sie einen Algorithmus, der diese Aufgabe löst.

Algorithmen

Informelle Algorithmen – Sortieren von Spielkarten (2)



Fragen auf der Metaebene

- Ist die Aufgabenstellung präzise genug?
Welche Ordnung zwischen den Karten soll zugrunde gelegt werden?
 - Zwischen den Farben: oder irgendwie anders?
 - Zwischen den Kartenwerten: 9 10 U ... vs. 9 U O K 10
 - Sortierung erst innerhalb einer Farbe oder erst innerhalb eines Kartenwertes?
- Algorithmusbeschreibung präzise, endl., elementar, effektiv?
- Ist der gefundene Algorithmus effizient?
D. h. benötigt er eine sinnvolle Menge an Aufwand (Zeit und Ressourcen) zum Lösen der Aufgabe?

Algorithmen

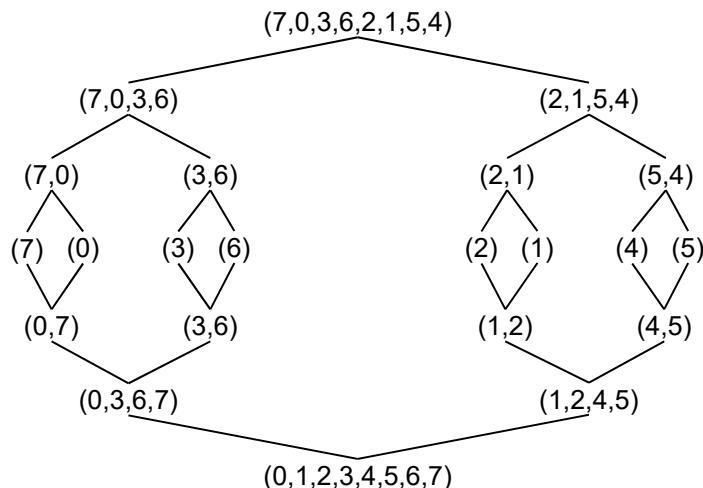
Informelle Algorithmen – Sortieren von Spielkarten (3)



Algorithmus: Vorsortieren und Zusammenmischen

- Falls Stapel s leer ist oder nur eine Karte umfasst:
Ergebnis ist s
- Sonst gehe wie folgt vor:
 - Spalte s in zwei nichtleere Stapel s1 und s2
 - Sortiere die beiden Stapel
 - Mische die beiden sortierten Stapel zusammen zu einem einzigen, größeren sortierten Stapel

Beispiel für die Ausführung des Algorithmus



Achtung

- Mischen der beiden Stapel ist keine elementare Operation
- Neue Aufgabenstellung: Mischen
- Wird selbst wieder durch einen Algorithmus gelöst

Eigenschaften

- ✓ Endlich, ✗ Elementar, ✗ Präzise, ✗ Effektiv
- Algorithmus ist rekursiv
- Algorithmus für das Sortieren stützt sich ab auf weiteren Algorithmus für das Mischen!
- Prinzip: Divide et impera!

Problem informeller Algorithmen

- Meist nicht präzise und effektiv
- Maschinelle Verarbeitung erfordert eine präzise festgelegte Sprache zur Beschreibung von Algorithmen,
d. h. eine so genannte formale Sprache

Formale Sprache

- Fest definierte Aufschreibung (Syntax)
- Klar definierte Bedeutung der Sprachkonstrukte (Semantik)

Begriffe

- Programmiersprache: formale Sprachen für Algorithmen
- Programm: Algorithmenbeschreibung

Verwendung formaler Sprachen

- Compilerbau, Programmiersprachen
- Komplexitätstheorie, Berechenbarkeitstheorie
- Kryptografie, Kryptoanalyse

Festlegen einer formale Sprache

- Meist definiert als Zeichenreihen über einem Zeichenvorrat
 - Festlegen des Zeichenvorrates
 - Festlegen der zulässigen Wörter
 - Durch explizites Aufzählen (evtl. ziemlich mühsam)
 - Durch *Grammatik*, d. h. Regelwerk für gültige Zeichenkombinationen
- Siehe Kapitel "Information und Repräsentation"

Arten von formalen Sprachen

- Textuell
 - Zeichenvorrat besteht aus Buchstaben, Ziffern, Sonderzeichen, ...
 - Z. B. Programmiersprachen wie Lisp, C, Java
- Grafisch
 - Zeichenvorrat umfasst grafische Elemente
 - Grammatik beschreibt korrekten Aufbau einer Grafik
 - Z. B. Struktogramm

Genaue Form der Sprache oft angepasst an die Maschine, welche Programme in der formalen Sprache ausführen soll.

Idee

- Einfache formale Algorithmenbeschreibung
- Elementarer Schritt:
Ersetzen eines Teilwortes durch eine andere Zeichenkette

Vorteile

- Endlich, präzise, elementar, effektiv
- Einfache Grundidee
- Zur Abarbeitung reicht sehr elementare Maschine

Nachteile

- Regelwerke schnell sehr umfangreich
- Dann schwer zu lesen und zu verifizieren

Verwendung nur in der theoretischen Informatik!!!

Definition

- Sei V ein Zeichenvorrat
- Ein Paar $(v, w) \in V^* \times V^*$ heißt *Ersetzung* oder *Regel*
- Übliche Notation für eine Ersetzung: $v \rightarrow w$
- *Textersetzungssystem*:
Endliche Menge R von Regeln über V

Anwendung einer Regel

- Eine Ersetzung
 $s \rightarrow t$
heißt *Anwendung einer Regel* $v \rightarrow w$,
falls es Wörter $a, v, w, z \in V^*$ gibt, so dass gilt:
 $s = a \circ v \circ z$ und $t = a \circ w \circ z$
- \circ heißt *Konkatenation*, d. h. Aneinanderhängen

Textersetzungssystem

- Zeichenvorrat $V = \{A, \dots, Z, a, \dots, z, \ddot{A}, \ddot{O}, \ddot{U}, \ddot{a}, \ddot{o}, \ddot{u}\}$
- Regeln $R = \{ae \rightarrow \ddot{a}, oe \rightarrow \ddot{o}, ue \rightarrow \ddot{u}\}$

Anwendungen der Regeln

- saegen → sägen
- loeten → löten
- zuerst → zürst

Bisher

- Regelanwendungen isoliert
- Noch keine komplexere Anwendung

Gesucht

- Algorithmus zur sukzessiven Anwendung der Textersetzung

Anwendung eines Textersetzungssystems R auf Wort s

- Falls eine Regel aus R auf s anwendbar ist:
 - Wende die Regel auf s an
 - Ermittle Ergebniswort t
 - Setze die Textersetzung mit t fort
- Sonst: Beende die Textersetzung

Textersetzungssystem

- Zeichenvorrat V = {A, ..., Z, a, ..., z, Ä, Ö, Ü, ä, ö, ü}
- Regeln R = {ae → ä, oe → ö, ue → ü}

Eingabewort

- Zuerst saegen dann loeten

Mögliche Abläufe des Algorithmus

- Zuerst saegen dann loeten → Zuerst sägen dann loeten
→ Zuerst sägen dann löten → Zürst sägen dann löten
- Zuerst saegen dann loeten → Zürst saegen dann loeten
→ Zürst saegen dann löten → Zürst sägen dann löten

Achtung: Ablauf nicht eindeutig, falls mehrere Regeln anwendbar sind!

Markov-Algorithmen

- Deterministische Anwendung von Textersetzungssystemen
- Festlegungen
 - Falls mehrere Regeln anwendbar,
nimm die zuerst notierte
 - Falls Regel an mehreren Stellen anwendbar,
wende sie möglichst weit links im Wort an

Beispiel

- Zuerst saegen dann loeten → Zuerst sägen dann loeten
→ Zuerst sägen dann löten → Zürst sägen dann löten
- Einzig möglicher Ablauf des Algorithmus!

Addition zweier natürlicher Zahlen in Strichdarstellung

Gegeben

- Zeichenvorrat $V = \{ |, <, >, + \}$
- Zahl n wird dargestellt durch das Wort $<||...|>$ (mit n Strichen)

Textersetzungssystem

- Nur eine einzige Regel: $>+< \rightarrow \varepsilon$
- Zur Erinnerung: ε bezeichnet das leere Wort

Beispielhafte Anwendung

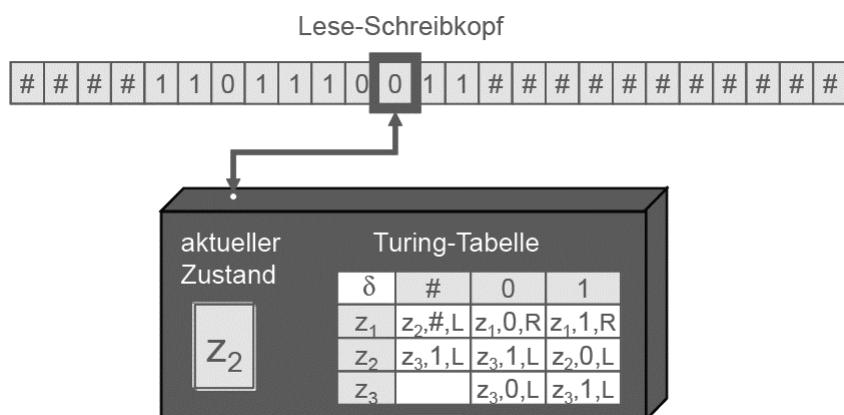
- Eingabe $<|||>+<|||||>$
- Einmaliges Anwenden der Regel liefert $<|||||||>$

Alan Turing (1936):

Alles, was man “mechanisch” berechnen kann, lässt sich auf eine einfache Maschine mit 6 primitiven Operationen reduzieren

Turingmaschine:

- Beidseitig unendlich langes Speicherband, mit Feldern
- Lese-/Schreibkopf
- Steuerwerk
 - Zustand
 - Primitive Operationen
- Turingtabelle (Programm)



Primitive Operationen:

- Rechts: Kopf eine Position nach rechts bewegen
- Links: Kopf eine Position nach links bewegen
- Drucken: Drucke Symbol auf Feld unter Kopf
- Lesen: Lese Symbol auf Feld unter Kopf
- Löschen: Lösche Symbol auf Feld unter Kopf
- Nichts/Halt: Tue nichts

Der Inhalt des Bandes der Turing-Maschine:

- Symbole aus einem Zeichensatz
- mindestens ein Symbol für eine “leeres Feld”
- mindestens ein anderes Symbol
- Beispiel: $Z = \{\#, 0, 1\}$

Zustände der Turing-Maschine:

- endliche viele, vordefinierte Zustände,
z.B. $\delta = \{z_1, z_2, z_3\}$
- einen Anfangszustand, z.B. z_1
- ein Register (Speicherzelle) mit dem aktuellen Zustand

Bei Berechnungen einer Turingmaschine können 3 Dinge geschehen:

1. ihr **Zustand** kann sich ändern
2. der **Inhalt des Bands** kann sich ändern
3. die **Position des Kopfes** kann sich ändern

Die Turingtabelle (Programm) definiert die gültigen Zustandsübergänge:

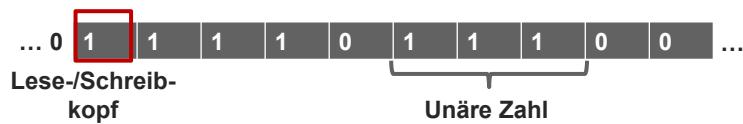
Turing-Tabelle				
δ	#	0	1	Wert von Feld auf Band
z_1	$z_2, \#, L$	$z_1, 0, R$	$z_1, 1, R$	Übergangsvorschrift: Wechsle in den Zustand z_1 . Schreibe "1" in das Feld, bewege Kopf eine Position nach rechts
z_2	$z_3, 1, L$	$z_3, 1, L$	$z_2, 0, L$	
z_3		$z_3, 0, L$	$z_3, 1, L$	

© Prof. Dr. Dieter Nazareth 2023

123

Wir wollen jetzt mit einer Turingmaschine 2 unäre Zahlen addieren:

- Zeichensatz $Z = \{0, 1\}$, 0 codiert „leere Zelle“
- Zustände $\delta = \{q_0, q_1, q_2, q_3\}$
- Anfangszustand q_0
- Mögliche Anfangszeichenreihe auf Band:



© Prof. Dr. Dieter Nazareth 2023

124

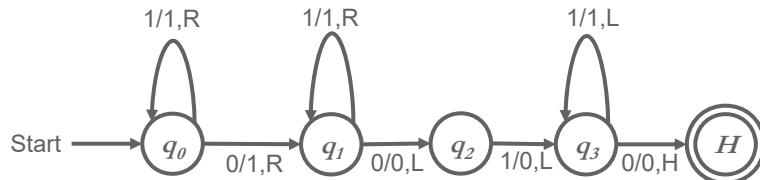
Algorithmen Turingmaschine



Programm:

	0	1	
q_0	$q_1, 1, R$	$q_0, 1, R$	Ende der 1. Zahl finden und Lücke füllen
q_1	$q_2, 0, L$	$q_1, 1, R$	Ende der 2. Zahl finden
q_2		$q_3, 0, L$	"1" rechts löschen
q_3	$q_3, 0, H$	$q_3, 1, L$	Kopf zurück auf Anfang

Grafische Darstellung durch Zustandsautomat:

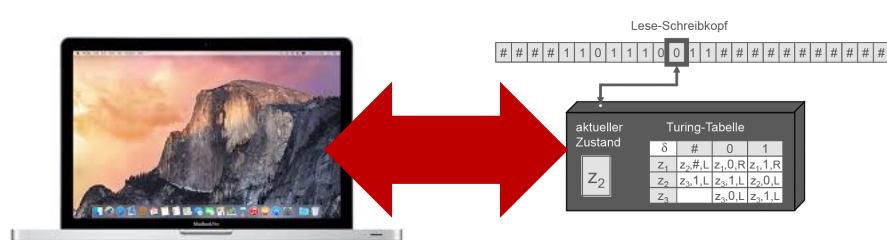


Algorithmen Turingmaschine



Jeder Rechner "ist" im Wesentlichen eine Turingmaschine

- d.h. man kann mit beiden das Gleiche berechnen
- moderne Programmiersprachen machen es uns "nur" leichter, der Maschine zu sagen, was sie machen soll



Ein Algorithmus heißt...

- **terminierend**, wenn er stets nach endlichen Schritten endet
- **deterministisch**, wenn in der Ausführung der Verarbeitungsschritte keine Auswahl besteht
- **determiniert**, wenn das Resultat eindeutig bestimmt ist
- **sequenziell**, wenn die Verarbeitungsschritte stets hintereinander ausgeführt werden
- **parallel**, wenn manche Schritte gleichzeitig ausgeführt werden

Beispiel: Berechnung von $1 + 2 + 3$

- $1 + 2 + 3 = 3 + 3 = 6$ oder $1 + 2 + 3 = 1 + 5 = 6$
- Zwei Rechenwege \Rightarrow nichtdeterministisch;
ein Ergebnis \Rightarrow determiniert

Klassische Elemente von Algorithmen

- Ausführung eines elementaren Schrittes
- Abfolge von Schritten (Sequenz)
- Fallunterscheidung über Bedingung (wenn – dann – sonst)
- Wiederholung (solange)
- Rekursion
 - Algorithmus ruft sich selbst auf
 - Spezialfall der Wiederholung
- Modularisierung (Strukturierung von Algorithmen)

Bedeutung

- Grafische Darstellung von Algorithmen meist intuitiv leicht verständlich
- Darstellung komplexer Algorithmen wird unübersichtlich

Verbreitete Notationen

- Flussdiagramm
 - Synonyme:
Programmablaufplan, Programmstrukturplan, Flowchart
 - Normiert durch DIN 66001, dennoch viele Varianten
- Struktogramm
 - Synonym: Nassi-Shneidermann-Diagramm
 - Normiert durch DIN 66261

Kontrollelemente

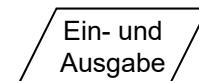
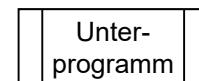
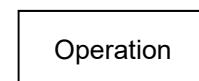


→
Verbindung zum
nächsten Element



Falls Bedingung erfüllt ist, folge
dem ja-Pfeil,
ansonsten dem nein-Pfeil

Aktionsknoten

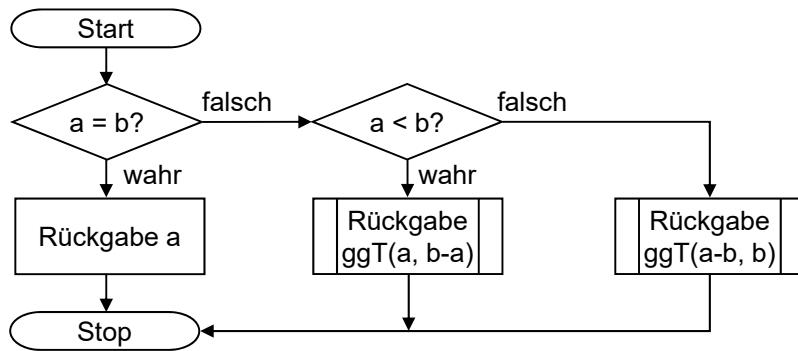


Algorithmen

Flussdiagramm – Beispiel ggT



- Falls $a=b$, dann gilt $\text{ggT}(a,b) = a$
- Sonst
 - Falls $a < b$, dann wende Algorithmus auf a und $b-a$ an
 - Sonst wende Algorithmus auf $a-b$ und b an



© Prof. Dr. Dieter Nazareth 2023

131

Algorithmen

Struktogramm



Grundidee

- Algorithmus beschreibt Abfolge von Einzelschritten
- Einzelschritt repräsentiert durch Block
 - Elementarer Block wird textuell beschrieben
 - Komplexer Block kapselt Kombination von anderen Blöcken

© Prof. Dr. Dieter Nazareth 2023

132

Algorithmen

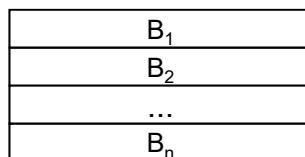
Struktogramm – Sequenz



Bedeutung

- Block, bestehend aus Sequenz von n Einzelblöcken

Syntax



Semantik

- Sequenzielles Ausführen der Blöcke B_1 bis B_n
- Ausführung in der Reihenfolge wie aufgeschrieben (von oben nach unten)
- Kein Block wird ausgelassen

Algorithmen

Struktogramm – Fallunterscheidung

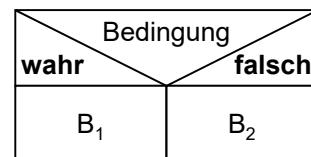


Bedeutung

- Ablaufpfad kann von Bedingung abhängen
- Bedingung kann erfüllt (wahr) oder nicht erfüllt (falsch) sein

Syntax

- Schlüsselwörter sind fett, können auch true oder false sein



Semantik

- Falls die Bedingung wahr ist, wird B_1 ausgeführt
- Sonst ist die Bedingung nicht erfüllt und B_2 wird ausgeführt
- B_1 und/oder B_2 kann auch leer sein; dann wird im entsprechenden Fall nichts ausgeführt

Algorithmen

Struktogramm – Auswahl (1)



Bedeutung

- Ablaufpfad kann mehr als zwei Zustände unterscheiden
- Auswahl erfolgt über Ausdruck

Syntax

- Auswahl-Konstrukt kann alternativ durch geschachtelte Fallunterscheidung ausgedrückt werden

Semantik

Ausdruck			
Wert 1	Wert 2		Wert n
B ₁	B ₂	...	B _n

- Falls der Ausdruck den Wert i hat, wird B_i ausgeführt
- Falls der Ausdruck keinen der Werte ergibt, wird nichts ausgeführt

Algorithmen

Struktogramm – Auswahl (2)



Syntax

- Variante des Ablauf-Konstrukts
- Schlüsselwort kann auch else oder default sein
- Auswahl-Konstrukt kann alternativ durch geschachtelte Fallunterscheidung ausgedrückt werden

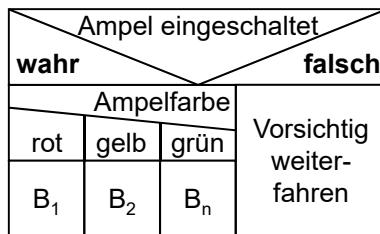
Ausdruck				
Wert 1	Wert 2		Wert n	sonst
B ₁	B ₂	...	B _n	B _e

Semantik

- Falls der Ausdruck den Wert i hat, wird B_i ausgeführt
- Falls der Ausdruck keinen der Werte ergibt, wird B_e ausgeführt

Verhalten an einer Verkehrsampel

- Geschachteltes Struktogramm
- **Wahr-Zweig** ist wieder ein Struktogramm

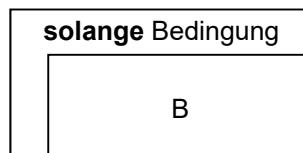


Bedeutung

- Teilbereich des Algorithmus wird mehrfach durchlaufen
- Wiederholen, solange Eintrittsbedingung erfüllt ist
- Diese Form der Wiederholung heißt auch *Iteration*

Syntax

- Schlüsselwort kann auch while sein



Semantik

- Solange die Bedingung erfüllt ist, wird B ausgeführt
- Ist die Bedingung nicht erfüllt, endet die Ausführung
- Ggf. wird also gar nichts ausgeführt, falls die Bedingung sofort nicht erfüllt ist

Algorithmen

Struktogramm – Wiederholung (2)

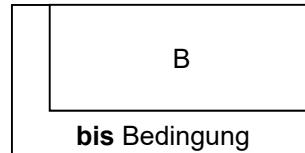


Bedeutung

- Variante der Wiederholung
- Bedingung steht als Abbruchbedingung am Ende
- Diese Form der Wiederholung heißt auch *Iteration*

Syntax

- Schlüsselwort kann auch *until* sein



Semantik

- B wird ausgeführt, bis die Bedingung erfüllt ist
- B wird also mindestens einmal ausgeführt, auch wenn die Bedingung gleich erfüllt ist

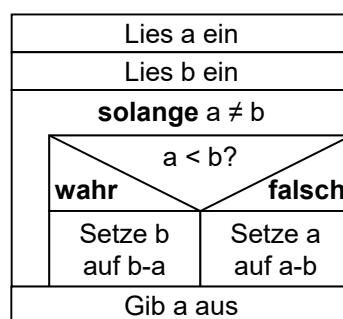
Algorithmen

Struktogramm – Beispiel ggT mit Wiederholung



Berechnung des größten gemeinsamen Teilers

- Einlesen zweier Zahlen a und b
- Algorithmus realisiert mittels einer Wiederholung
- Geschachteltes Struktogramm



Algorithmen

Struktogramm – Modularität (1)



Bedeutung

- Struktogramm für komplexen, umfangreichen Algorithmus wird schnell unübersichtlich
- Mehrfach auftretende Abschnitte
 - Ausgliedern in eigenes Struktogramm
 - Mit Namen versehen
 - Bei Bedarf darauf verweisen (aufrufen)

Modul

- In sich abgeschlossener Algorithmus
- Kann in andere Algorithmen eingebaut werden
- Übergabe von Parametern ist möglich
- Verwandte Begriffe: Prozedur, Funktion, Unterprogramm

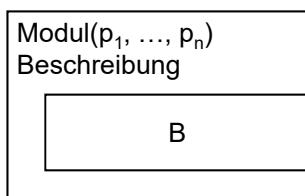
Algorithmen

Struktogramm – Modularität (2)

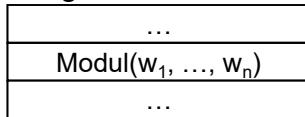


Syntax

- Moduldefinition



- Modulaufruf mit Übergabe der Werte für die Parameter



Semantik

- An der Aufrufstelle wird B ausgeführt (Rumpf des Moduls)
- Parameter p₁, ..., p_n werden vor der Ausführung mit Werten w₁, ..., w_n belegt

Algorithmen

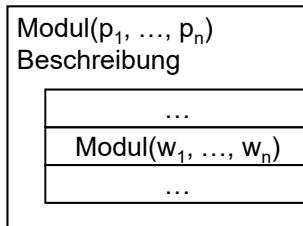
Struktogramm – Rekursion



Bedeutung

- Spezialfall der Modularität
- Algorithmus ruft in seinem Rumpf sich selbst auf
- Nur realisierbar mit Modulkonzept!

Syntax



Semantik

- Wie bei Modul

Algorithmen

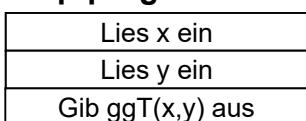
Struktogramm – Beispiel ggT mit Rekursion



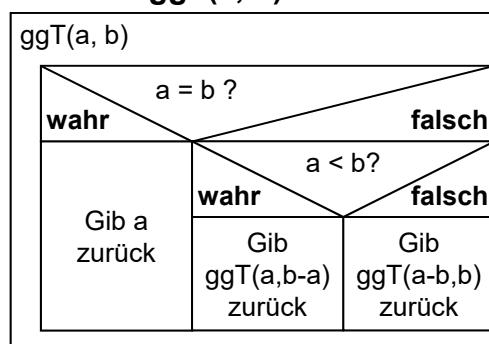
Berechnung des größten gemeinsamen Teilers

- Algorithmus realisiert mittels Rekursion
- Zwei Struktogramme, jedes für ein Modul

Hauptprogramm



Funktion ggT(a, b)



Algorithmen

Struktogramm – Bewertung (1)



- ✓ Endlich, ✓ Elementar
- Präzise
 - Sowohl Syntax als auch Semantik der Konstrukte können formal und präzise definiert werden
 - Aber: Beschreibung der Elementarschritte durch umgangssprachliche Sätze (z. B. „Gib a zurück“)
 - Grund: Elementare Schritte bisher nicht formal definiert
 - ⇒ Algorithmen sind nicht vollständig präzise

Algorithmen

Struktogramm – Bewertung (2)



- Effektiv
 - Analog zu oben
 - Grafische Konstrukte können maschinell ausgeführt werden
 - Textuelle Inhalte können i. A. nicht maschinell ausgeführt werden
- Struktogramme gelten als semiformale Beschreibungstechnik
- Werden zur formalen Sprache, falls Elementarschritte formal beschrieben, z. B. durch konkrete Programmiersprache

Bedeutung

- Algorithmus ruft sich selber auf
- Alternatives Verfahren zur Iteration (solange, bis)

Umsetzung

- Bei rekursivem Aufruf eines Moduls wird Kopie des Rumpfes angelegt
- Entspricht Einkopieren des Rumpfes an die Aufrufstelle, mit Ersetzen der Parameter durch die übergebenen Werte

Eigenschaften rekursiver Algorithmen

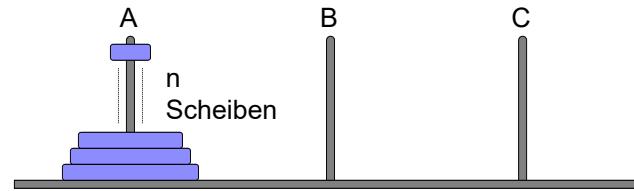
- Meist problemorientierter (d. h. näher an Aufgabenstellung), kürzer und eleganter als iterative Algorithmen
Beispiel: Türme von Hanoi
- Prädestiniert für rekursive Datenstrukturen, z. B. Bäume

Eigenschaften iterativer Algorithmen

- Meist "maschinenorientierter"
- Effektiver auszuführen:
weniger Speicher, weniger Rechenzeit

Beide Varianten können ineinander umgewandelt werden!

Türme von Hanoi



Gesucht

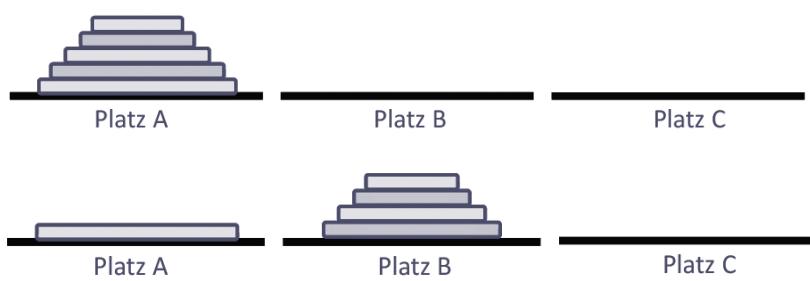
- Algorithmus, um Turm von Stab A auf Stab C zu bewegen

Regeln

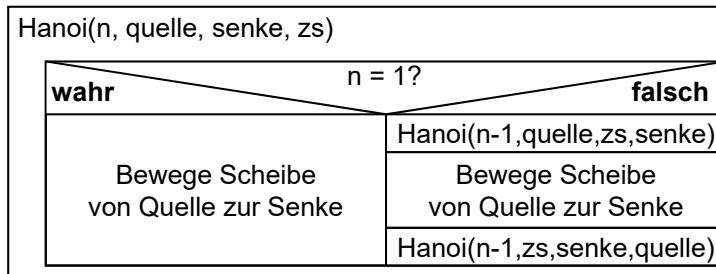
- Bei jedem Schritt darf nur eine Scheibe bewegt werden
- Es darf nie eine größere Scheibe auf einer kleineren liegen
- Stab B darf als Zwischenspeicher verwendet werden

Prinzip des Algorithmus

- Bringe die obersten $n-1$ Scheiben von der Quelle zum Zwischenspeicher
- Bewege die unterste Scheibe von der Quelle zur Senke
- Bringe alle $n-1$ Scheiben vom Zwischenspeicher zur Senke



Stuktogramm

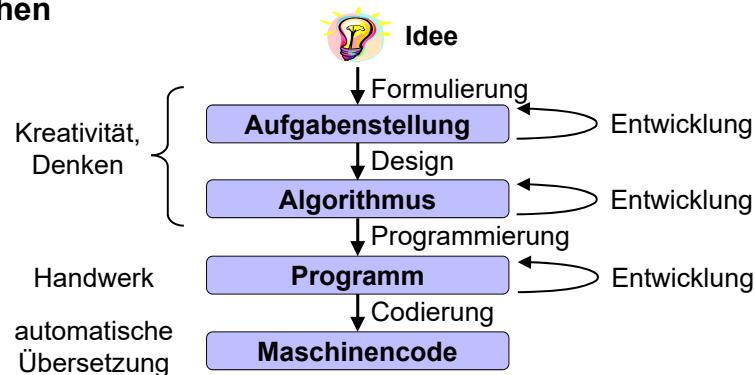


- Um einen Turm mit n Scheiben von A nach B zu bewegen sind $2^n - 1$ Scheiben zu bewegen!
- Bei Geschwindigkeit von einer Scheibe pro Sekunde ergeben sich folgende Zeiten für die Übertragung des Turms:
 - $n=8 \Rightarrow 2^8 - 1 = 255$ Sekunden
 - $n=16 \Rightarrow 2^{16} - 1 = 65535$ Sekunden > 18 Stunden
 - $n= 32 \Rightarrow 2^{32} - 1 = 4294967295$ Sekunden > 136 Jahre
- Schnellste Rechner der Welt bräuchten ca. 2 Jahre für $n=64$
- Es kann gezeigt werden, dass es keinen schnelleren Algorithmus gibt!

Bisher

- Notationen für Algorithmen
- Nicht: Weg von der Aufgabe zum ausführbaren Programm

Vorgehen



Wichtiges Prinzip der disziplinierten Programmierung

- NICHT mit der Programmierung anfangen, ehe die Aufgabenstellung genau verstanden und beschrieben ist

Benötigt

- Präzise Beschreibung der Aufgabenstellung
- Im Idealfall auch hier wieder: Formale Sprachen

Spezifikation

- Präzise Beschreibung der Anforderungen
- Definition der Systemgrenzen
- Sollte vollständig und korrekt sein
- Fehler, die hier gemacht werden, pflanzen sich lawinenartig über gesamten Entwicklungsprozess hinweg fort

Algorithmen

Methodisches Vorgehen – Spezifikation (2)



Nutzen

- Hilfsmittel, um die Aufgabenstellung genau zu erfassen
- Frühzeitiges Erkennen von potenziellen Schwierigkeiten
- Vermeiden von unnötigem Programmieraufwand durch Entwicklung "falscher Programme" (d. h. von Programmen, die an der Aufgabenstellung vorbei gehen)
- Gesamtüberblick vor Beginn der Implementierung ist Voraussetzung für adäquate Architektur des Programms
- Grundlage für Vertrag zwischen Auftraggeber und Auftragnehmer

Wann ist ein Programm "falsch"?

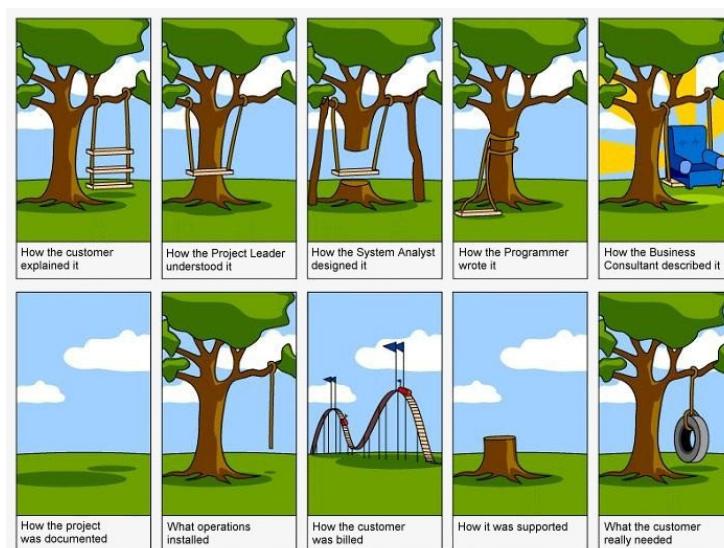
- Wenn es das, was es macht, nicht richtig macht
- Wenn es nicht das Richtige macht

© Prof. Dr. Dieter Nazareth 2023

155

Algorithmen

Methodisches Vorgehen – Spezifikation (3)



© Prof. Dr. Dieter Nazareth 2023

156

Zwei grundlegende Ansätze

- *Top-down*
 - Sukzessives Konkretisieren einer zunächst abstrakt formulierten Lösungsidee
- *Bottom-up*
 - Bereits bestehende Teillösungen zu Lösung des Problems zusammenfügen

Achtung

- Top-down-Vorgehen führt i. A. zu klareren Strukturen
- Bottom-up-Vorgehen erleichtert Wiederverwendung bereits bestehender Komponenten
- Praxis: *hybrides* Vorgehen, d.h. Mischen beider Ansätze

Top-down

- Urheber: Dijkstra, 1969 und Wirth, 1971
- Idee
 - Zerlege die Aufgabe in einfachere Teilaufgaben
 - Wiederhole diese Zerlegung sukzessive, bis die Teilaufgaben elementaren Schritten entsprechen
- Prinzip der schrittweisen *Verfeinerung*

Beispiel: Unser Vorgehen bei "Türme von Hanoi"

- Zuerst grobe Struktur des Algorithmus entwerfen
- Noch keine konkrete Umsetzung der Türme, Scheiben, ...
- Realisierung der Teilaufgabe "Bewege Scheibe von Quelle zur Senke" zunächst noch unklar; weiter verfeinern!

Algorithmen

Methodisches Vorgehen – Algorithmen Entwickeln – Bottom-up



Motivation

- Softwareentwicklung beginnt nicht auf der grünen Wiese
- Integrieren einer neuen Lösung in bestehende IT-Landschaft
- Wiederverwenden bestehender Komponenten

Bottom-up

- Sichten bestehender, realisierter Teilalgorithmen
- Ggf. Modifizieren bzw. Erweitern bestehender Bausteine
- Nutzbare Bausteine gruppieren zu komplexerer Einheit
- Wiederhole das Vorgehen, bis Lösung zusammengebaut ist

Verwendung

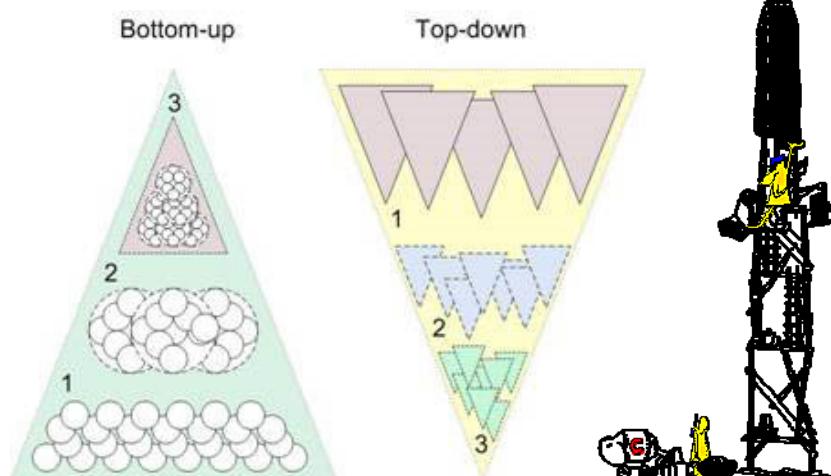
- Typisches Vorgehen bei objektorientierter Entwicklung

Algorithmen

Methodisches Vorgehen – Algorithmen Entwickeln – Bottom-up



Top-Down vs. Bottom-Up



Algorithmen Zusammenfassung



- Algorithmus definiert Verfahren zur Lösung einer Aufgabe
- Beschreibung soll endlich, präzise, elementar, effektiv sein
- Informelle Beschreibung ungenau, aber intuitiv leicht lesbar
- Formale Beschreibung meist komplex
- Grafische Notationen: Anschaulich, aber präzise
 - Flussdiagramm
 - Struktogramm
- Rekursion
 - Zurückführen eines Problems auf (einfachere) Version seiner selbst
 - Algorithmus stützt sich auf sich selbst ab
- Vor der Implementierung klären, was eigentlich zu tun ist!!!

Grundlagen der Informatik I Überblick



- Was ist Informatik
- Information und Repräsentation
- Algorithmen
- Programmiersprachen
 - Funktionale Sprachelemente
 - Prozedurale Sprachelemente
- Datenstrukturen
- Komplexität
- Berechenbarkeit

Programmiersprachen Überblick



- Begriffe
- Arten von Programmiersprachen
- Definieren einer Programmiersprache
- Backus-Naur-Form
- Kernbestandteile einer höheren Programmiersprache
- Datentypen
- Kernbestandteile eines Programms
- Funktionale Sprachelemente
- Prozedurale Sprachelemente

Programmiersprachen Begriffe – Programmiersprache



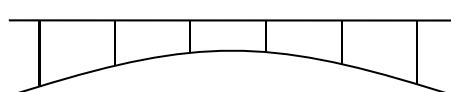
Definitionen

- **Programmiersprache**: Formale Sprache zur Beschreibung von Algorithmen (und Datenstrukturen)
- **Programm**: Gültiges Wort aus einer Programmiersprache

Programmiersprache als Brücke zwischen Mensch und Maschine



Abstrakte
Hochsprache für
den Programmierer



Übersetzung in
Maschinencode
(durch Compiler)

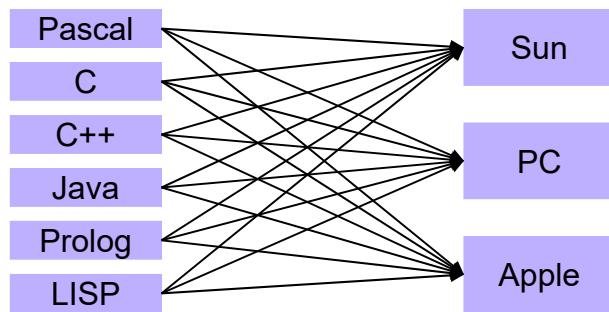


Resultat
ausführbar auf
Rechenanlage

Programmiersprachen Begriffe – Compiler



- Compiler übersetzt abstrakte Hochsprache in Maschinensprache einer bestimmten Rechenanlage
- Problem: Viele verschiedene Rechnertypen und Sprachen

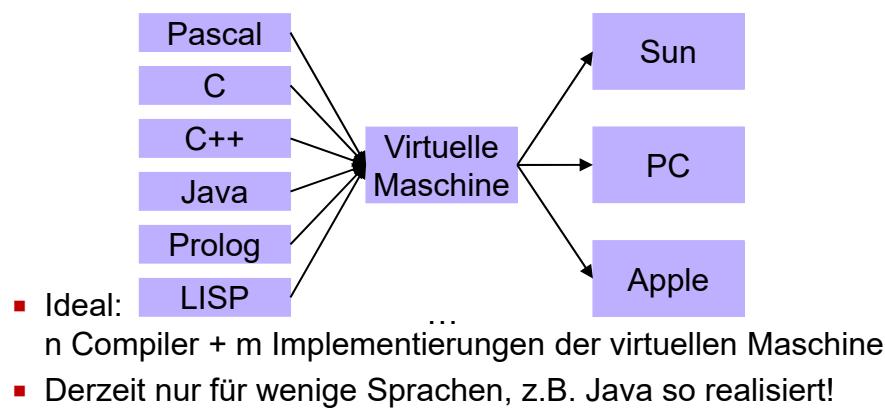


- Bei n Sprachen und m Plattformen: $n * m$ Compiler nötig

Programmiersprachen Begriffe – Virtuelle Maschine (Idealvorstellung)



- Bindeglied zwischen Hochsprache und Maschinensprache
- Virtuelle Maschine auf jeder konkreten Maschine simuliert
- Übersetzte Programme plattformunabhängig einsetzbar!



Klassifizierung von Programmiersprachen

- Nach Kernkonzepten
- Nach Grad der Abstraktion (maschinennah vs. Hochsprache)
- Nach Art der Darstellung (grafisch vs. textuell)

Typische Kategorien

- Funktional, applikativ
- Prozedural, imperativ, zuweisungsorientiert
- Objektorientiert
- Logisch, prädikativ, deklarativ
- Maschinennah
- Grafisch

Kernkonzepte

- Funktion und deren Anwendung (Applikation) auf Parameter

Eigenschaften

- Programme meist hochgradig rekursiv
- Notation meist sehr mathematisch orientiert

Bekannte Vertreter

- λ -Kalkül, ML, Haskell

Syntaxbeispiel: ML

```
fun fak 0    = 1
|   fak succ(x) = succ(x) * fak(x);
```

Kernkonzepte

- Variable, Zuweisung
- Wiederholungskonstrukte (Schleifen)

Eigenschaften

- Orientiert am Aufbau von Rechenanlagen

Bekannte Vertreter

- Algol, Pascal, C

Syntaxbeispiel: C

```
unsigned fak(unsigned x)
{
    unsigned erg=1;

    while (x > 1) {
        erg = erg * x;
        x--;
    }

    return erg;
}
```

Kernkonzepte

- Klassen und Objekte als Kernelemente
- Klasse kapselt Struktur und Verhalten

Eigenschaften

- Unmittelbarer Bezug zu den Strukturen der realen Welt
- Gut geeignet für (sehr) große Programme

Bekannte Vertreter

- Ada, C++, Java

Syntaxbeispiel: Java

```
public class Math {
    public static unsigned
    fak (unsigned x) {
        unsigned erg=1;
        while (x > 1) {
            erg = erg*x;
            x--;
        }
        return erg;
    }
}
```

Kernkonzepte

- Prädikatenlogische Formel (Klausel)

Eigenschaften

- Arbeitsprinzip nach logischen Argumentationsketten
- Verwendet z. B. im Bereich der künstlichen Intelligenz
- Expertensysteme, Theorembeweiser

Bekannte Vertreter

- Prolog

Syntaxbeispiel

- Regeln

```
fak(0, 1) ←  
fak(x, x*y) ← fak(x-1, y)
```

- Anfrage

```
?fak(5, a)
```

Kernkonzepte

- Maschinenbefehl

Eigenschaften

- Orientiert sich an Befehlssatz und Aufbau eines konkreten Prozessors
- Verwendet für Programmierung kleiner, spezieller Prozessoren, meist eingebettet in Geräte

Bekannte Vertreter

- Je Prozessorart eine Sprache

Syntaxbeispiel: Assembler

```
; Parameter in R0  
; Ergebnis in R1  
move R1, [0001]  
loop:  
    cmp R0, [0002]  
    jnz ende  
    mult R1, R1, R0  
    dec R0  
ende:
```

Kernkonzepte

- Grafische Konstrukte
- Elementare Schritte oft wie bei prozeduralen Sprachen

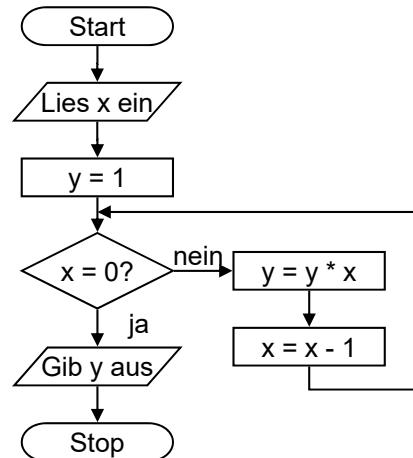
Eigenschaften

- Grafische Programme meist umgesetzt als prozedurale oder objektorientierte Programme

Bekannte Vertreter

- Struktogramm, Programmablaufplan, endlicher Automat (Statecharts)

Syntaxbeispiel: Flussdiagramm



Heutige Programmiersprachen sind meist Mischung aus natürlicher Sprache und mathematischen Formeln!

Formale Definition hat zwei Bestandteile

- Definition der Aufschreibung (Syntax)
- Definition der Bedeutung der Sprachkonstrukte (Semantik)

Definition der Syntax

- Durch grammatischen Regeln (siehe Compilerbau)
- BNF-Notation (Backus-Naur-Form)

Definition der Semantik einer Sprache

- Arten von Semantik
 - *Funktional*
 - Weist jedem Ausdruck einen Wert zu
 - Meist über mathematische Ausdrücke
 - *Operationell*
 - Definiert für jeden Ausdruck einen Algorithmus zu dessen Auswertung
- Notation
 - Textuell, dann aber meist wenig präzise
 - Über mathematische Formeln (meist bei funktionaler Semantik)

Bedeutung

- Metasprache zur Definition von formalen Sprachen mit kontextfreier Grammatik
- Benannt nach J. Backus und P. Naur
- Synonym: BNF-Notation

Verwendung: Präzise Definition der Syntax von...

- Vielen höheren Programmiersprachen
- Befehlen von Betriebssystemen
(Z. B. Hilfen in der DOS-Shell, Manual Pages von Linux)
- Befehlssätzen von Netzwerkprotokollen

- **Terminale:** Zeichen der Zielsprache
- **Produktionen:**
 - Ableitungsregeln
 - Definieren die zulässigen Zeichenkombinationen
- **Nichtterminale (Nonterminale):**
 - Hilfszeichen in den Produktionen
 - Dürfen in den Produktionen links oder rechts auftreten
- Sonderzeichen
 - ::= Definition einer Regel
 - < > Kennzeichnen der Nichtterminale
 - | Alternative
- In BNF-Varianten weitere Sonderzeichen möglich

Gegeben

- Menge T von Terminalzeichen
- Maximale Sprache über T ist T^*

Verwendung der BNF-Notation

- Definieren einer formalen Sprache X über T
- BNF-Ausdruck definiert Einschränkung, welche Zeichenkombinationen über T möglich sind
- Definierte Sprache X ist Teilmenge von T^* , also $X \subset T^*$

Bedeutung:

Definieren genau eines gültigen Wortes der Sprache

Terminales Zeichen

- Gegeben: t Zeichen aus T, also $t \in T$
- t ist BNF-Ausdruck, dessen Sprache X genau das Wort {t} umfasst

Beispiele

- $T = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- 0 definiert Sprache {0}
- 1 definiert Sprache {1}
- ...

Bedeutung: Alternative

Vereinigung

- Gegeben
 - BNF-Ausdruck R mit Sprache X
 - BNF-Ausdruck S mit Sprache Y
- $R | S$ ist BNF-Ausdruck mit zugehöriger Sprache $X \cup Y$

Beispiel

- 0 definiert Sprache {0}
- 1 definiert Sprache {1}
- $0 | 1$ definiert Sprache {0, 1}
- $0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$ definiert Sprache {0, 1, ..., 9}

Bedeutung: Hilfszeichen und Bezeichner für Produktionen

Nonterminal

- Gegeben
 - BNF-Ausdruck R mit Sprache X
 - Menge N von Nonterminalen, Nonterminal $n \in N$
- $\langle n \rangle ::= R$ definiert eine BNF-Regel (Produktion)
- Tritt $\langle n \rangle$ auf der rechten Seite einer Produktion auf, wird $\langle n \rangle$ durch R ersetzt

Beispiel

- $\langle \text{Ziffer außer Null} \rangle ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$
- $\langle \text{Ziffer} \rangle ::= 0 | \langle \text{Ziffer außer Null} \rangle$

Bedeutung: Aneinanderhängen

Konkatenation

- Gegeben:
 - BNF-Ausdruck R mit Sprache X
 - BNF-Ausdruck S mit Sprache Y
- RS ist BNF-Ausdruck mit Sprache $\{x o y \mid x \in X, y \in Y\}$

Beispiel

- $\langle \text{Ziffer außer Null} \rangle ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$
- $\langle \text{Ziffer} \rangle ::= 0 | \langle \text{Ziffer außer Null} \rangle$
- $\langle \text{Zweistellige Zahl} \rangle ::= \langle \text{Ziffer außer Null} \rangle \langle \text{Ziffer} \rangle$
- $\langle \text{Zwanziger Zahl} \rangle ::= 2 \langle \text{Ziffer} \rangle$

Bedeutung: Optionales Element

Option

- Gegeben: BNF-Ausdruck R mit Sprache X
- $[R]$ ist BNF-Ausdruck mit Sprache $X \cup \{\epsilon\}$

Beispiel

- $\langle \text{Ziffer außer Null} \rangle ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$
- $\langle \text{Ziffer} \rangle ::= 0 | \langle \text{Ziffer außer Null} \rangle$
- $\langle \text{Zifferfolge} \rangle ::= \langle \text{Ziffer} \rangle | \langle \text{Ziffer} \rangle \langle \text{Zifferfolge} \rangle$
- $\langle \text{Positive Zahl} \rangle ::= \langle \text{Ziffer außer Null} \rangle | \langle \text{Ziffer außer Null} \rangle \langle \text{Zifferfolge} \rangle$
- $\langle \text{Ganze Zahl} \rangle ::= 0 | [-] \langle \text{Positive Zahl} \rangle$

Bedeutung: Wiederholung

Iteration

- Gegeben: BNF-Ausdruck R mit Sprache X
- R^* ist BNF-Ausdruck mit Sprache $\{x_1 o \dots o x_n \mid n \geq 0, x_i \in X\}$

Beispiel

- $\langle \text{Ziffer außer Null} \rangle ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$
- $\langle \text{Ziffer} \rangle ::= 0 | \langle \text{Ziffer außer Null} \rangle$
- $\langle \text{Positive Zahl} \rangle ::= \langle \text{Ziffer außer Null} \rangle \langle \text{Ziffer} \rangle^*$
- Zugehörige Sprache zu $\langle \text{Positive Zahl} \rangle$
 $\{1, 2, \dots, 9, 10, 11, \dots, 99, 100, 101, \dots\}$

Bedeutung: Wiederholung

Mindestens eine Iteration

- Gegeben: BNF-Ausdruck R mit Sprache X
- R^+ ist BNF-Ausdruck mit Sprache $\{x_1 \circ \dots \circ x_n \mid n > 0, x_i \in X\}$

Beispiel: Reelle Zahl

- $\langle \text{Ziffer außer Null} \rangle ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$
- $\langle \text{Ziffer} \rangle ::= 0 | \langle \text{Ziffer außer Null} \rangle$
- $\langle \text{Reelle Zahl} \rangle ::= 0 , 0^+ |$
 $\quad [-] 0 , \langle \text{Ziffer} \rangle^* \langle \text{Ziffer außer Null} \rangle \langle \text{Ziffer} \rangle^* |$
 $\quad [-] \langle \text{Ziffer außer Null} \rangle \langle \text{Ziffer} \rangle^* , \langle \text{Ziffer} \rangle^+$

Bedeutung: Gruppieren von BNF-Ausdrücken

Klammerung

- Gegeben: BNF-Ausdruck R mit Sprache X
- $\{R\}$ ist BNF-Ausdruck mit Sprache X

Beispiel: Reelle Zahl

- $\langle \text{Ziffer außer Null} \rangle ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$
- $\langle \text{Ziffer} \rangle ::= 0 | \langle \text{Ziffer außer Null} \rangle$
- $\langle \text{Reelle Zahl} \rangle ::=$
 $\quad ::= 0 , 0^+ |$
 $\quad [-] \{ 0 , \langle \text{Ziffer} \rangle^* \langle \text{Ziffer außer Null} \rangle \langle \text{Ziffer} \rangle^* |$
 $\quad \langle \text{Ziffer außer Null} \rangle \langle \text{Ziffer} \rangle^* , \langle \text{Ziffer} \rangle^+ \}$

Arithmetische Ausdrücke für ganze Zahlen, mit +, -, *

- <Ziffer außer Null> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
- <Ziffer> ::= 0 | <Ziffer außer Null>
- <Positive Zahl> ::= <Ziffer außer Null> <Ziffer>*
- <Ganze Zahl> ::= 0 | [-] <Positive Zahl>
- <Operator> ::= + | - | *
- <Arithm. Ausdruck> ::= <Ganze Zahl> |
(<Arithm. Ausdruck>) |
<Arithm. Ausdruck> <Operator>
<Arithm. Ausdruck>

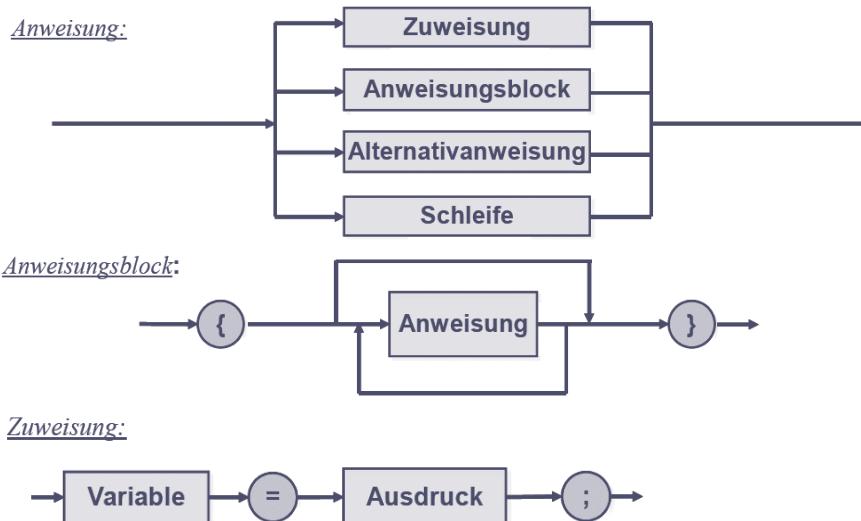
Beispiele

- 42, 40 + 2, (4 + 3) * 6, 4 + 3 * 6, 42 – 10 * (8 + 4 * 6)
- Achtung: Ausdrücke nicht vollständig geklammert!

Eine einfache grafische Darstellung der Syntax von Programmen bieten so genannte **Syntaxdiagramme**. Es besteht aus Rechtecken, Ovalen und Pfeilen.

- In den Rechtecken stehen Begriffe, die anderswo durch eigene Syntaxdiagramme erklärt werden (Nonterminale).
- In den Ovalen stehen lexikalische Elemente, also Symbole und Namen (Terminale). Diese sind wörtlich in den Programmtext zu übernehmen.
- Pfeile führen von einem Sprachelement zu einem im Anschluss daran erlaubten Sprachelement.
- Jedes Syntaxdiagramm hat genau einen Eingang und genau einen Ausgang.

Programmiersprachen Syntaxdiagramme (Beispiele)



© Prof. Dr. Dieter Nazareth 2023

189

Programmiersprachen Kernbestandteile einer höheren Programmiersprache



Achtung

- Für funktionale, prozedurale und objektorientierte Programme ähnlich
- Aufbau logischer Programme ist grundlegend verschieden!

Bestandteile

- Vordefinierte **Datentypen** legen Definitions- und Wertemengen fest
- **Operationen/Befehle** (elementare Grundaktionen)
 - Ein-/Ausgabe
 - Mathematische oder logische Operationen
 - Kontrollstrukturen steuern den Ablauf eines Programms
- **Klammern** zur Begrenzung von Blöcken

© Prof. Dr. Dieter Nazareth 2023

190

Motivation

- Programm arbeitet i. A. auf bestimmter Art von Daten
- Es ist sicherzustellen, dass Parameter mit richtiger Art von Daten belegt werden
- Beispiel: Anwenden der Fakultätsfunktion nur auf ganzer Zahl sinnvoll

Definition **Datentyp (Sorte, Art)**

- Bezeichner für nichtleere Menge von Datenelementen
- Kapselt Repräsentationen gleichartiger Datenelemente
- Gleichartig bedeutet:
 - Gleicher Bedarf an Speicherplatz
 - Meist auch gleiche Semantik

Bedeutung

- Datentypen ermöglichen Festlegung von Definitions- und Wertebereich einer Funktion (Funktionalität)
- Grundlegende Datentypen sind in vielen Programmiersprachen vordefiniert (Basistyp, einfacher Typ)
- Erweiterung um benutzerdefinierte Datentypen ist möglich

Typische vordefinierte Datentypen

- | | |
|---------------------|---------------------|
| ▪ Natürliche Zahlen | natural, nat |
| ▪ Ganze Zahlen | int, integer |
| ▪ Reelle Zahlen | real, float, double |
| ▪ Zeichen | character, char |
| ▪ Wahrheitswerte | boolean, bool |

Definition **Datenstruktur**

- Kombination von einem oder mehreren Datentypen und den zugehörigen charakteristischen Operationen

Konvention

- Elementare Datenstrukturen meist als Datentypen bezeichnet
- Benutzerdefinierte Datenstrukturen werden meist Datenstruktur genannt

Gegeben

- Menge F von Funktionssymbolen
- Menge T von Datentypen

Definition **Funktionalität**

- $f: t_1 \times \dots \times t_n \rightarrow t_e$ mit $f \in F$, $t_1, \dots, t_n, t_e \in T$ heißt **Funktionalität** der Funktion f
- Funktion heißt **n-stellig**, d. h. braucht n Parameter
- Falls $n = 0$ ist Funktion f eine **Konstante**

Definition **Signatur**

- Paar (F, T) , zusammen mit den Funktionalitäten

Eigenschaften

- Einfachster Datentyp
- Mögliche Werte: true, false
- Operationen
 - Zweistellig:
or, and: boolean x boolean → boolean
 - Einstellig: not: boolean → boolean
 - Nullstellig, d. h. Konstanten: true, false: → boolean

Achtung

- Obige Typdefinition noch unvollständig
- Noch festzulegen: Wirkungsweise der Operationen
- Wahlweise über Wertetabellen oder über Gleichungen

Wertetabellen für boolesche Grundoperationen

- and wird auch repräsentiert durch \wedge
- or wird auch repräsentiert durch \vee
- not wird auch repräsentiert durch \neg

\wedge	true	false			true	false	\vee	true	false
true	true	false		true	false		true	true	true
false	false	false		false	true		false	true	false

Ergänzende Operation

\Rightarrow	true	false
true	true	false
false	true	true

Gleichungen zur Beschreibung der Wirkungsweise der booleschen Grundoperationen

- $x \text{ or } y = y \text{ or } x$
- $x \text{ and } y = y \text{ and } x$
- $\text{true or } x = \text{true}$
- $\text{false or } x = x$
- $\text{true and } x = x$
- $\text{false and } x = \text{false}$

- Involutionsgesetz $\neg\neg x = x$
- Kommutativgesetz $x \wedge y = y \wedge x$ $x \vee y = y \vee x$
- Assoziativgesetz $(x \wedge y) \wedge z = x \wedge (y \wedge z)$ $(x \vee y) \vee z = x \vee (y \vee z)$
- Idempotenzgesetz $x \wedge x = x$ $x \vee x = x$
- Absorptionsgesetz $x \wedge (x \vee y) = x$ $x \vee (x \wedge y) = x$
- Distributivgesetz
 - $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$
 - $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$
- Gesetze von de Morgan
 - $\neg(x \wedge y) = \neg x \vee \neg y$ $\neg(x \vee y) = \neg x \wedge \neg y$
 - Neutralitätsgesetz $x \vee (y \wedge \neg y) = x$ $x \wedge (y \vee \neg y) = x$
 - Konstante $\text{false} = x \wedge \neg x$ $\text{true} = x \vee \neg x$

Mögliche Werte

- 0, 1, 2, 3, ...: → nat

Operationen

- +, -, *, div, mod : nat x nat → nat
- succ : nat → nat
- 0 : → nat

Gleichungen: Für alle $x, y \in \text{nat}$ gilt

- $0 + x = x$
- $\text{succ}(x) + y = \text{succ}(x + y)$
- $0 * y = 0$
- $\text{succ}(x) * y = x * y + y$
- $(x \text{ div } y) * y + x \text{ mod } y = x$

Mehrsortige Datenstruktur

- T = {boolean, nat}

Operationen

- Operationen wie bisher bei boolean und nat
- Zusätzlich:
 - isZero : nat → boolean
 - equals : nat x nat → boolean

Gleichungen: Für alle $x, y \in \text{nat}$ gilt

- $\text{isZero}(0) = \text{true}$
- $\text{isZero}(\text{succ}(x)) = \text{false}$
- $\text{equals}(x, x) = \text{true}$
- $\text{equals}(x, \text{succ}(x)) = \text{false} \dots$

Achtung

- Für funktionale, prozedurale und objektorientierte Programme ähnlich
- Aufbau logischer Programme ist grundlegend verschieden!

Bestandteile

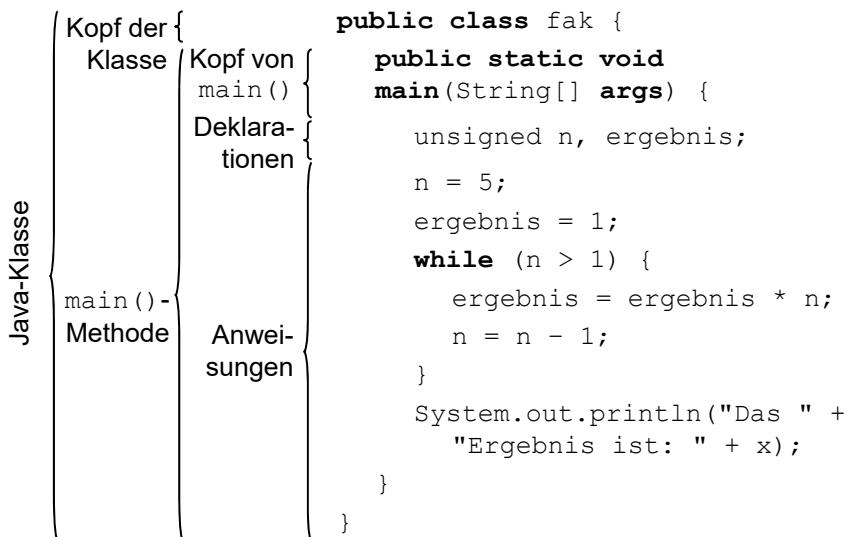
- *Kopf*
 - Name
 - Benötigte Parameter
- *Rumpf* (engl. *Body*; auch als "Block" bezeichnet)
 - Ggf. Deklarationen von Variablen
 - Anweisungen, die den Algorithmus realisieren

Pascal-Programm

Rumpf	Kopf {	PROGRAM fak;
	Deklara- tionen {	VAR n, ergebnis : natural;
Anwei- sungen {	BEGIN	n := 5;
		ergebnis := 1;
	WHILE n > 1 DO	
	BEGIN	ergebnis := ergebnis * n;
		n := n - 1;
	END	
		writeln('Das Ergebnis ist: ',
		ergebnis)
	END .	

Programmiersprachen

Kernbestandteile eines Programms – Java



© Prof. Dr. Dieter Nazareth 2023

203

Programmiersprachen

Zusammenfassung



- Programmiersprache verbindet Mensch und Maschine
- Verschiedene Arten von Programmiersprachen verfügbar
- Formale Sprache, definierte Syntax und Semantik
- Definition der Syntax mittels Backus-Naur-Form
- Datentypen und Operationen als zentrale Bestandteile
- Datenstruktur kapselt Datentyp und charakteristische Operationen
- Funktionalität definiert Definitions- und Wertebereich einer Funktion
- Programmkopf definiert Name und Parameter
- Programmrumpf beinhaltet Deklarationen von Variablen sowie die Anweisungen (d. h. den Algorithmus)

© Prof. Dr. Dieter Nazareth 2023

204

Grundlagen der Informatik I

Überblick



- Was ist Informatik
- Information und Repräsentation
- Algorithmen
- Programmiersprachen
 - Funktionale Sprachelemente
 - Prozedurale Sprachelemente
- Datenstrukturen
- Komplexität
- Berechenbarkeit

Programmiersprachen – Funktionale Sprachelemente

Übersicht



- Begriffe
- Funktionsdefinition
- Ausdruck
 - Parameter
 - Funktionsaufruf
 - Bedingter Ausdruck
- Funktionen mit nicht definiertem Ergebnis
- Bindung von Bezeichnern
- Funktion höherer Ordnung
- Arten von Rekursion

Definitionen

▪ **Funktion**

Abbildung, die einem Wert aus dem Definitionsbereich *genau* ein Element aus dem Wertebereich zuweist

▪ Mathematische Schreibweise für Funktion f:

$f: t_1 \times \dots \times t_n \rightarrow t_e$
wobei

- f ist ein *Funktionssymbol*
- t_1, \dots, t_n, t_e bezeichnen jeweils *Datentypen*
- t_1, \dots, t_n sind *Parametertypen*,
d. h. Datentypen der Parameter von f
- t_e ist *Ergebnistyp*,
d. h. der Datentyp des Ergebnisses von f

Definitionen

▪ $f: t_1 \times \dots \times t_n \rightarrow t_e$ heißt **Funktionalität** der Funktion f

▪ f hat dann n **Parameter**

- Veränderliches Element eines Unterprogramms
- Wird bei Definition des Unterprogramms formal angelegt
- Bekommt beim Aufruf des Unterprogramms konkreten Wert zugewiesen

▪ Funktion mit null Parametern heißt **Konstante**

▪ **Signatur** (im Kontext von Programmiersprachen)

- Funktionssymbol plus die Folge der Parametertypen
- Wird verwendet, um Unterprogramm bekannt zu machen
- Achtung: In der Literatur gibt es auch andere Def.!

Programmiersprachen – Funktionale Sprachelemente

Funktionsdefinition

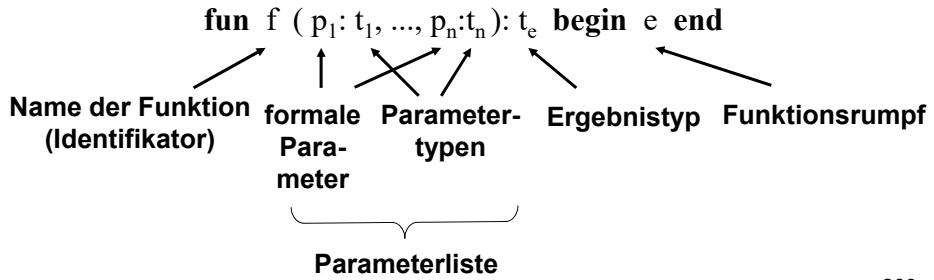


Bedeutung

- Zentrales Element von funktionalen Programmiersprachen
- Programm besteht im Wesentlichen aus Menge von Funktionsdefinitionen und deren Anwendung in Ausdrücken

Funktionsdefinition:

Zur Definition von Funktionen verwenden wir folgende Syntax:



© Prof. Dr. Dieter Nazareth 2023

209

Programmiersprachen – Funktionale Sprachelemente

Funktionsdefinition – BNF-Notation



<function declaration> ::=

```
  fun <function name> ( [<param list>]): <type>
    begin
```

```
      <expression>
```

```
    end
```

<param list> ::= <param> {, <param>}*

<param> ::= <id> : <type>

<function name> ::= <id>

<type> ::= <id>

© Prof. Dr. Dieter Nazareth 2023

210

Funktion zur Berechnung der Fakultät

```
fun fak (n : Nat) : Nat
begin
...
end
```

Funktion zur Berechnung des Quadrats einer ganzen Zahl

(Annahme: Multiplikation * auf Integer vorhanden)

```
fun square (n : Integer) : Integer
begin
  n * n
end
```

Konstante π

```
fun pi () : float
begin
  3.14
end
```

Funktion zur Berechnung der Kreisfläche

(Annahme: Multiplikation * auf float vorhanden)

```
fun area (r : float) : float
begin
  r * r * pi()
end
```

Funktion zur Berechnung der Fakultät

```
unsigned fak (unsigned n) {  
    ...  
}
```

Funktion zur Berechnung des Quadrats einer ganzen Zahl

```
int square (int n) {  
    return n * n;  
}
```

Konstante Pi

```
float pi () {  
    return 3.141592654;  
}
```

Funktion zur Berechnung der Fläche eines Kreises mit reellem Radius r:

```
float area (float r) {  
    return r * r * pi();  
}
```

Bedeutung

- Beschreibt den Rumpf einer Funktion
- Synonyme: *Term, Expression*

In funktionalen Sprachen drei Ausprägungen von Ausdrücken:

- Parameter
- Funktionsanwendung
- Bedingter Ausdruck

```
<expression> ::= <parameter> |  
                  <function application> |  
                  <conditional expression>
```

Gegeben

- Sei p vom Typ t ein formaler Parameter

Definition

- Dann ist p ein Ausdruck vom Typ t

Semantik

- Jeder Parameter hat die Bedeutung seines Wertes
- Semantik definiert Abbildung von Syntax zu Werten;
- Wir drücken Wert eines Ausdrucks a durch $I(a)$ aus
- $I()$ ist dabei Interpretationsfunktion

Beispiel

- 10 vom Typ integer bezeichnet den Wert 10

Programmiersprachen – Funktionale Sprachelemente Ausdruck – Funktionsaufruf (1)



Gegeben

- Sei f Funktionsname mit Funktionalität $f: t_1 \times \dots \times t_n \rightarrow t_e$
- Seien e_1, \dots, e_n beliebige Ausdrücke der Typen t_1, \dots, t_n

Definition

- Dann ist $f(e_1, \dots, e_n)$ ein Ausdruck vom Typ t_e
- $f(e_1, \dots, e_n)$ bezeichnet die **Anwendung** der Funktion f auf den Werten der Ausdrücke e_i

Anmerkung

- Funktionsanwendung soll nur dann erlaubt sein, wenn für alle i mit $1 \leq i \leq n$ der Wert von e_i vom Typ t_i ist
- Programmiersprachen, die das sicher stellen, heißen **getype Sprachen**

© Prof. Dr. Dieter Nazareth 2023

217

Programmiersprachen – Funktionale Sprachelemente Ausdruck – Funktionsaufruf (2)



Gegeben seien die vorherigen Funktionsdefinitionen

- square: int \rightarrow int, π : \rightarrow float, area: float \rightarrow float
- 10 sei ein Wert vom Typ int

Beispiele

- square(10) und square(square(10)) sind korrekte Ausdrücke
- area($\pi()$) korrekter Ausdruck

Gegenbeispiele, d. h. keine Ausdrücke unserer Sprachen

- area($\pi()$, $\pi()$) nicht korrekt, da zu viele Parameter
- square($\pi()$) nicht korrekt, da pi() ein Ergebnis vom Typ float liefert, square() aber einen Parameter vom Typ integer erwartet

© Prof. Dr. Dieter Nazareth 2023

218

Programmiersprachen – Funktionale Sprachelemente Ausdruck – Funktionsaufruf (3)



Semantik

- Gegeben: **fun** $f(p_1:t_1, \dots, p_n:t_n): t_e$ **begin** e **end**
 - Sei f Funktionsname mit Funktionalität $f: t_1 \times \dots \times t_n \rightarrow t_e$
 - Seien p_1, \dots, p_n die Parameter in der Definition von f
 - Sei e der Rumpf von f
- Operationelle Semantik der Funktionsanwendung $f(e_1, \dots, e_n)$
 1. Berechne die Werte von e_1, \dots, e_n in beliebiger Reihenfolge
 2. Weise den Parametern p_i die zugehörigen Werte von e_i zu
 3. Berechne den Wert des Rumpfs e
 4. Weise $f(e_1, \dots, e_n)$ den Wert von e zu, d. h. ersetze an der Aufrufstelle $f(e_1, \dots, e_n)$ durch den Wert von e

Programmiersprachen – Funktionale Sprachelemente Ausdruck – Funktionsaufruf (4)



Beispiel zur Semantik: $I(\text{square}(\text{square}(10)))$

1. Berechne Wert des Arguments $\text{square}(10)$
 1. Berechne Wert des Arguments 10: $I(10) = 10$
 2. Weise dem Parameter n den Argumentwert zu: $n = 10$
 3. Berechne Wert des Rumpfes der Funktion: $I(n * n)$
 1. Ermittle Wert des Parameters: $I(n) = 10$
 2. Werte Ausdruck im Rumpf aus: $I(n * n) = 100$
 4. Weise Ergebnis dem Ausdruck zu: $I(\text{square}(10)) = 100$
2. Weise dem Parameter n den Argumentwert zu: $n = 100$
3. Berechne Wert des Rumpfes der Funktion: $I(n * n)$
 1. Ermittle Wert des Parameters: $I(n) = 100$
 2. Werte Ausdruck im Rumpf aus: $I(n * n) = 10000$
4. Weise Ergebnis dem Ausdruck zu: $I(\text{square}(\text{square}(10))) = 10000$

Begriffe

- Ausdrücke e_1, \dots, e_n heißen *aktuelle Parameter / Argumente*
- *Infix-Notation*
 - Funktionssymbol steht zwischen den Argumenten
 - Üblich bei zweistelligen Funktionen
 - Beispiel: $3 + 4$ statt $+(3, 4)$
- *Postfix-Notation*
 - Funktionssymbol steht nach dem Argument
 - Gelegentlich verwendet bei einstelligen Funktionen
 - Beispiel: $4!$ statt $!(4)$ für die Fakultätsfunktion
- *Präfix-Notation*
 - Funktionssymbol wird den Argumenten vorangestellt

Bemerkungen

- Fast alle gängigen Programmiersprachen sind getypt
- In manchen Sprachen (z. B. bei C) erfolgt bei nicht typkorrektem Funktionsaufruf lediglich eine Warnung!!!
- In objektorientierten Sprachen
 - Möglichkeit, Subtypen eines Typs zu definieren
 - Beispiel: $\text{nat} \subset \text{int}$
 - Parameter darf dann auch von einem Subtyp des erwarteten Typs sein
 - Semantik des Beispiels:
 - Ein nat ist auch ein int
 - Wo ein int erwartet wird, darf ein nat verwendet werden

Programmiersprachen – Funktionale Sprachelemente

Ausdruck – Bedingter Ausdruck (1)



Gegeben

- Sei b ein Boolescher Ausdruck
- Seien e_1 und e_2 Ausdrücke vom gleichen Typ t

Definition

Dann ist

if b **then** e_1 **else** e_2 **fi**
ein *bedingter Ausdruck* vom Typ t

Bemerkung

- Die Ausdrücke e_1 und e_2 heißen *Zweige* des bedingten Ausdrucks.
- e_2 kann **nicht** wie bei der bedingten Anweisung (siehe prozedurale Sprachelemente) leer sein.

Programmiersprachen – Funktionale Sprachelemente

Ausdruck – Bedingter Ausdruck (2)



Beispiele

```
fun fak (n: nat):nat
begin
    if n == 0 then 1
        else n * fak (n-1)
    fi
end
fun ggt (a:nat, b:nat):nat
begin
    if a == b then a
        else if a < b then ggt(a, b-a)
            else ggt(a-b, b)
        fi
    fi
end
```

Semantik

- Gegeben:
 - Bedingter Ausdruck: **if b then e₁ else e₂ fi**
- Semantik: **if b then e₁ else e₂ fi** hat ...
 - Den Wert von e₁, falls b den Wert true hat
 - Den Wert von e₂, falls b den Wert false hat
- **Frage:** Was ist, wenn die Berechnung von b nicht terminiert oder der Rechner dabei abstürzt?
- **Antwort:** Dann ist der Wert des Ausdrucks undefiniert (siehe später)!

Beispiel zur Semantik: I(fak(1))

1. Berechne Wert des Arguments: $I(1) = 1$
2. Weise dem Parameter n den Argumentwert zu: $n = 1$
3. Berechne Wert des Rumpfes der Funktion:
 $I(if n==0 then 1 else n * fak(n-1) fi)$
 1. Ermittle Wert des booleschen Ausdrucks:
 $I(n==0) = \text{false}$
 2. Berechne Wert von e2: $I(n * fak(n-1))$
 1. Berechne Wert von $fak(n-1)$
 1. Berechne Wert von $n-1$: $I(n-1) = 0$
 2. Weise dem Parameter n den Argumentwert zu:
 $n = 0$

Programmiersprachen – Funktionale Sprachelemente Ausdruck – Bedingter Ausdruck (5)



3. Berechne Wert des Rumpfes der Funktion:

$I(if\ n==0\ then\ 1\ else\ n * fak(n-1)\ fi)$

1. $I(n==0) = \text{true}$

2. $I(1) = 1$

3. $I(if\ n==0\ then\ 1\ else\ n * fak(n-1)\ fi)$
= 1

4. Weise Ergebnis dem Ausdruck zu: $I(fak(1)) = 1$

2. Berechne Wert von $n * fak(n-1)$: $I(n*fak(n-1)) = 1$

3. Weise Ergebnis dem Ausdruck zu:

$I(if\ n==0\ then\ 1\ else\ n * fak(n-1)\ fi) = 1$

4. Weise Ergebnis dem Ausdruck zu: $I(fak(1)) = 1$

Programmiersprachen – Funktionale Sprachelemente Ausdruck – Bedingter Ausdruck (6)



Bemerkungen

- Bedingung b wird stets vor den Zweigen ausgewertet
- Abhängig von b wird *entweder e1 oder e2* ausgewertet
- Bedingter Ausdruck entspricht einer Abbildung
 $\text{if: } \text{bool } x \rightarrow t$ ($x \rightarrow t$ funktioniert aber in den meisten Sprachen nicht wirklich!)
- Bedingter Ausdruck wird verwendet, um undefinierte Werte zu vermeiden (z. B. bei Anwendung der Funktion auf falsche Parameter)
- Beispiel:

```
if x == 0 then 1
      else y / x
fi
```

Programmiersprachen – Funktionale Sprachelemente Ausdruck – BNF Notation



```
<expression>          ::= <parameter> |  
                      <function application> |  
                      <conditional expression>  
<parameter>          ::= <id>  
<function application> ::= <id> [<exp_list>]  
<exp list>            ::= <expression> {,<expression>}*  
<conditional expression> ::= if <expression>  
                           then <expression>  
                           else <expression> fi
```

Programmiersprachen – Funktionale Sprachelemente Funktionen mit nicht definiertem Ergebnis (1)



Beispiel: Unendlich rekursive Funktion

```
fun infinite (n:int):int {  
    infinite (n)  
}
```

Semantik von infinite(1), d. h. I(infinite(1))

- Berechne Wert des Arguments: I(1) = 1
- Weise dem Parameter den Argumentwert zu: n = 1
- Berechne Wert des Rumpfs: I(infinite(n))
 - Berechne Wert des Arguments: I(n) = 1
 - Weise dem Parameter den Argumentwert zu: n = 1
 - Berechne Wert des Rumpfs: I(infinite(n))
 - Berechne ...

Bemerkungen

- Berechnung terminiert nicht
- Wir weisen $I(infinite(1))$ daher den künstlichen Wert \perp zu:
 $I(infinite(1)) = \perp$
- \perp heißt „bottom“
- Semantik: **if b then e₁ else e₂ fi** hat den Wert \perp , falls b den Wert \perp hat

Definition: *partiell*

- Funktion heißt *partiell*, wenn sie nicht auf allen Eingabeelementen definiert ist
- Undefiniertes Ergebnis wird semantisch auch ausgedrückt durch Zeichen \perp (bottom)

Beispiel

- Ganzzahlige Division div: int x int → int
- Undefiniert bei 0 als Divisor: $I(div(x,0)) = \perp$

Definition: *strikt*

- Funktion heißt *strikt*, wenn sie immer dann ein undefiniertes Ergebnis liefert, wenn irgendein Argument undefiniert ist
- Funktion, die trotz eines undefinierten Arguments ein definiertes Ergebnis liefern kann, heißt *nicht strikt*

Bemerkung

- Striktheit garantiert, dass jeder Fehler bei der Applikation durchschlägt
- Nichtstrikte Funktionen haben in gewisser Weise heilende Eigenschaften (Fehler können abgefangen werden)
- Bedingter Ausdruck $\text{if: bool } x \text{ t } x \text{ t } \rightarrow t$ ist nur strikt im ersten Argument (Bedingung)

Bemerkungen

- *Call by value, eager evaluation*
 - Beim Funktionsaufruf werden zunächst *alle* Argumente ausgewertet.
 - Semantik des Funktionsaufrufes ist somit strikt!
 - Die meisten gängigen Programmiersprachen haben ein „Call by value“ Semantik.
- *Call by need, lazy evaluation*
 - Nur die im Rumpf tatsächlich benötigten Argumente werden ausgewertet.
 - Semantik des Funktionsaufrufs ist i. A. nicht strikt!

Gegeben seien folgende Funktionsdefinitionen

```
fun const(n : int) : int
begin
  1
end
```

Semantische Varianten bei der Auswertung

- Call by value, strikt: $I(\text{const}(\text{div}(1,0))) = \perp$
- Call by need, nicht strikt: $I(\text{const}(\text{div}(1,0))) = 1$
- Grund: Parameter wird im Rumpf gar nicht verwendet!

Bedeutung

- Bezeichner ist ein Identifikator
- Hat in Ausdruck meist lediglich Platzhalterfunktion
- Kann durch anderen noch nicht verwendeten Bezeichner ersetzt werden
- Wenn das geht, sprechen wir von freiem Bezeichner, d. h. der Bezeichner kommt im Ausdruck frei vor

Beispiel

- Ausdruck $x * (x+1)$
- Gleichwertig zu Ausdruck $y * (y+1)$

Bedeutung

- Funktionsdefinition macht aus einem Bezeichner einen Parameter, wenn dieser in der Parameterliste deklariert wird.
- Bezeichner kommt dann im Rumpf der Funktion nicht mehr frei vor, sondern ist durch Parameterdefinition gebunden.
- Wir sprechen von *gebundenem Bezeichner*.

Beispiel

- `fun f(x:int):int begin`
`x * (x+1);`
`end`
- Deklaration `x:int` zeichnet `x` als Parameter aus
- Bezeichner `x` ist somit an diesen Parameter gebunden

© Prof. Dr. Dieter Nazareth 2023

237

Bemerkung

- Auch gebundener Bezeichner kann konsistent durch anderen Bezeichner ersetzt werden, ohne die Semantik der Funktion zu verändern
- Voraussetzungen
 - Bezeichner wird auch in der Funktionsdeklaration ersetzt
 - Neuer Bezeichner kommt im Rumpf der Funktion vor der Ersetzung noch nicht vor

Beispiele

- `fun f (int x):int begin x * (x+1) end ✓`
- `fun f (int 1):int begin 1 * (1+1) end □`
Wann bezeichnet 1 den Parameter, wann die Konstante?

© Prof. Dr. Dieter Nazareth 2023

238

Bedeutung

- Bisher als Parametertypen und Ergebnistyp nur Basistypen der Sprache möglich
- Bei funktionalen Sprachen können auch Funktionen als Parameter oder als Ergebnis übergeben werden
- Wir sprechen denn von Funktionen *höherer Ordnung*

Beispiel

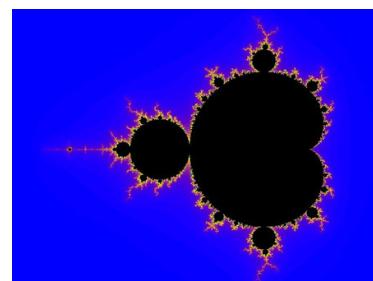
```
fun sum (f:int → nat, x:nat):int begin
    if x == 0 then f(0) else f(x) + sum(f,x-1) fi
end
sum(square,3) = square(3)+square(2)+square(1)+square(0)
= 9 + 4 + 1 + 0 = 14
```

Bedeutung

- Rekursion liegt vor, wenn Funktion sich in ihrem Rumpf selber aufruft.
- Grad an Rekursivität kann unterschiedlich hoch sein, je nachdem, wie oft Funktion sich im Rumpf selbst aufruft.

Arten von Rekursion

- Lineare Rekursion
- Repetitive Rekursion
- Kaskadenartige Rekursion
- Wechselseitige Rekursion



Definition

- Gegeben
 - **fun** $f(x_1:t_1, \dots, x_n:t_n) : t_e$ **begin** e **end**
 - f sei dabei eine rekursive Funktion
 - f werde in jedem Zweig einer Fallunterscheidung von e maximal einmal aufgerufen
- Dann heißt f *linear rekursiv*

Eigenschaft

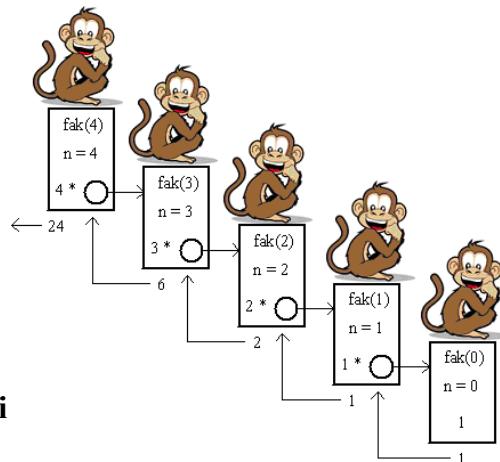
- Jeder rekursive Aufruf von f führt zu höchstens einem weiteren Aufruf von f

Beispiele

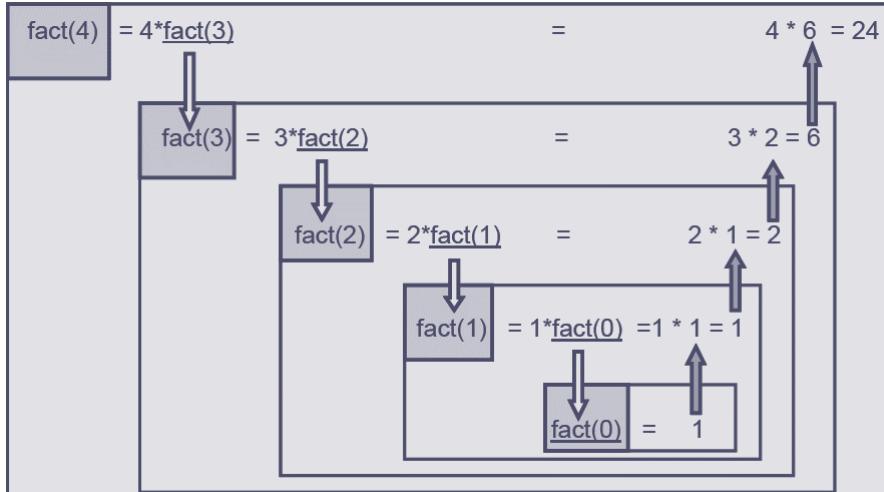
- Fakultätsfunktion
- Größter gemeinsamer Teiler
- Ganzzahlige Division

Ganzzahlige Division

```
fun div (a: nat, b: nat): nat
  begin if a < b  then 0
        else div(a-b,b) +1 fi
  end
```



Programmiersprachen – Funktionale Sprachelemente Arten von Rekursion – Linear (2)



© Prof. Dr. Dieter Nazareth 2023

243

Programmiersprachen – Funktionale Sprachelemente Arten von Rekursion – Repetitiv (1)



Bedeutung

- Spezialfall der linearen Rekursion
- Linear rekursive Funktion heißt dann *repetitiv rekursiv*, wenn alle rekursiven Aufrufe im Funktionsrumpf als letzte Aktion des Rumpfes erfolgen
- Synonyme: Endrekursion, engl. *tail recursion*

Eigenschaften

- Bevor Auswertung eines neuen rekursiven Aufrufs beginnt, sind alle Berechnungen des aktuellen Aufrufs abgeschlossen
- Aktuelle Parameter müssen nicht aufgehoben werden (Speichereffizienz)

© Prof. Dr. Dieter Nazareth 2023

244

Beispiel: Größter gemeinsamer Teiler

```
fun ggt (a:nat, b:nat):nat begin
    if a==b
        then a
    else if a<b
        then ggt(a, b-a)
        else ggt(a-b, b)
    fi
fi
end
```

Rot markierte Aktionen sind jeweils letztes Element einer Ausführung des Rumpfs!

Bedeutung

- Funktion f heißt dann *kaskadenartig rekursiv*, wenn in mindestens einem Zweig des Funktionsrumpfes f mindestens zweimal aufgerufen wird
- Synonyme: baumartige Rekursion, nichtlineare Rekursion

Eigenschaften

- Exponentielles Anwachsen der rekursiven Aufrufe
- Kaskadenartig rekursive Funktionen sind aufwändig zu berechnen!

Beispiele

- Türme von Hanoi
- Binomialkoeffizient
 - Dabei seien $k, n \in \mathbb{N}_0$ und $n \geq k$

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \begin{cases} 1 & \text{falls } k = 0 \text{ oder } k = n \\ \binom{n-1}{k} + \binom{n-1}{k-1} & \text{sonst} \end{cases}$$

```
fun binom (n: nat, k: nat): nat begin
  if k == 0 ∨ n == k then 1
    else binom(n-1,k) + binom(n-1,k-1)
  fi
end
```

Bedeutung

- Gegeben
 - (Mindestens) Zwei Funktionsdefinitionen f_1, f_2 mit Rümpfen r_1, r_2
 - In r_1 werde f_2 aufgerufen, in r_2 werde f_1 aufgerufen
 - Dann sprechen wir von verschränkter Rekursion

Beispiel

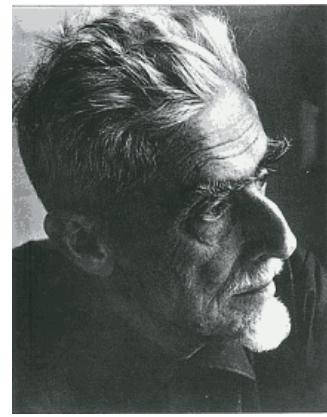
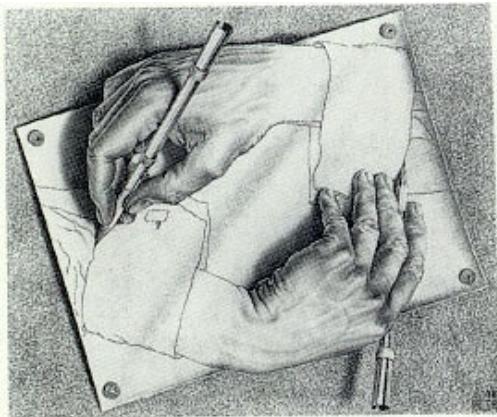
```
fun iseven (n: nat): boolean
begin if n == 0 then true
      else isodd (n-1) fi end

fun isodd (n: nat): boolean
begin if n == 0 then false
      else iseven (n-1) fi end
```

Programmiersprachen – Funktionale Sprachelemente
Arten von Rekursion – Verschränkt (2)



**Verschränkte Rekursion nach Maurits Cornelis Escher
(Niederländer, 1898–1972):**



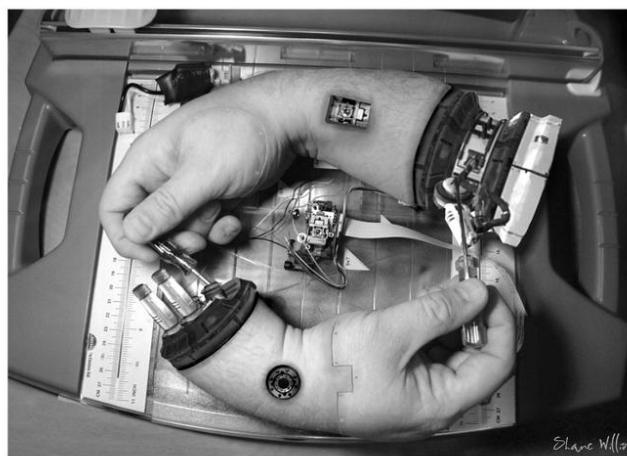
© Prof. Dr. Dieter Nazareth 2023

249

Programmiersprachen – Funktionale Sprachelemente
Arten von Rekursion – Verschränkt (3)



Verschränkte Rekursion (moderne Version):



© Prof. Dr. Dieter Nazareth 2023

250

Programmiersprachen – Funktionale Sprachelemente Zusammenfassung



- Funktion ist zentrales Element der funktionalen Sprachen
- Elemente einer Funktionsdefinition:
Schlüsselwort, Ergebnistyp, Funktionsidentifikator, Parameterliste, Funktionsrumpf
- Funktionsrumpf ist ein Ausdruck:
Parameter, Funktionsaufruf oder bedingter Ausdruck
- Semantik weist jedem Ausdruck eine Bedeutung zu
- Partielle Funktion nicht auf allen Eingaben definiert
- Strikte Funktion liefert \perp , falls ein Eingabewert \perp ist
- Freie Bezeichner in Funktion austauschbar
- Unterscheidung verschiedener Rekursionsarten, je nach Struktur und Häufigkeit der rekursiven Aufrufe

Grundlagen der Informatik I Überblick



- Was ist Informatik
- Information und Repräsentation
- Algorithmen
- Programmiersprachen
 - Funktionale Sprachelemente
 - Prozedurale Sprachelemente
- Datenstrukturen
- Komplexität
- Berechenbarkeit

Programmiersprachen – Prozedurale Sprachelemente Übersicht



- Motivation
- Anweisung
 - Variable und Zuweisung
 - Sequenzielle Komposition
 - Bedingte Anweisung
 - Wiederholungsanweisung
 - Prozeduraufruf
 - Block
- Bindung von Parametern
- Rekursion vs. Iteration
- Funktionen und Prozeduren in C, C++, Java

Programmiersprachen – Prozedurale Sprachelemente Motivation

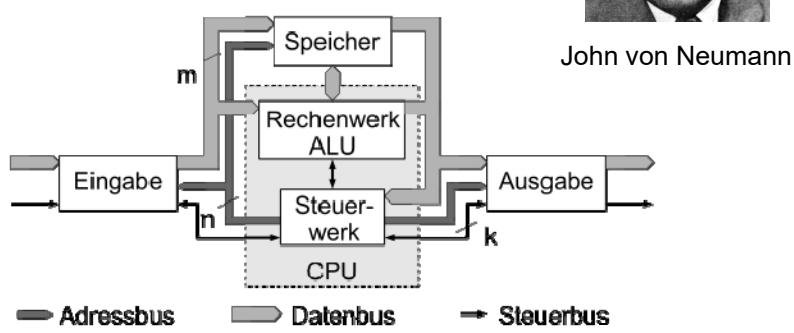


Von-Neumann-Architektur

- Heute gebräuchliche Rechnerarchitektur
- Maschine mit linear geordnetem Speicher, in dem sowohl Daten als auch Programm abgelegt sind



John von Neumann



Abarbeiten eines Programms auf Von-Neumann-Rechner

- Nächste Programmanweisung aus Speicher lesen
- Benötigte Daten aus Speicher lesen
- Anweisung ausführen
- Ergebnisse in Speicher schreiben

Geeignetes Programmierkonzept für diese Architektur

- Sequenzielle Folge von Anweisungen/Zuweisungen
- Anweisung kann Zustand einer Variablen ändern

Bedeutung

- Ausdruck (expression) war zentrales Element bei funktionalen Sprachen
- Anweisung (statement) ist entsprechendes Konstrukt bei prozeduralen Sprachen

Ausprägungen von Anweisung in prozeduraler Sprache

```
<anweisung> ::= <variablen-deklaration> |  
                  <zweisung> |  
                  <anweisung> <anweisung> |  
                  <bedingte anweisung> |  
                  <wiederholungsanweisung> |  
                  <prozedurauftrag> |  
                  <block>
```

Bedeutung

- Beispiel: Berechne ggT(square(3), square(square(3)))
- Auswertung nach unserer operationellen Semantik
 - Erst Werte der Parameter auswerten
 - Dann Rumpf der Funktion auf Parameterwerten auswerten

Achtung

- Ausdruck square(3) wird doppelt berechnet; ineffizient!

Besser

- square(3) nur einmal auswerten
- Ergebnis zwischenspeichern in Variable, bei Bedarf nutzen
- Also: $x = \text{square}(3); \text{ggT}(x, \text{square}(x))$

Neue Konzepte: **Programmvariable** und **Zuweisung**

Definition

- Variable muss als solche gekennzeichnet sein, bevor wir sie verwenden können
- **Definition** legt Name und Typ einer Variablen und legt sie im Speicher an
- Syntax in fiktiver Programmiersprache: **var** id : t;
 - id ist beliebige Zeichenkette, t ist Datentyp

Zuweisung (assignment)

- Ändert den Wert einer Variablen
- Syntax in unserer fiktiven Programmiersprache: id = e;
 - id ist definierter Variablenbezeichner vom Typ t
 - e ist Ausdruck (expression) vom Typ t

- Variable muss textuell vor ihrer Verwendung definiert werden
- Variable kann immer wieder verwendet werden
- Symbole für Zuweisung
 - In einigen Sprachen, z. B. in Pascal: :=
 - In C/Java/...: =
- Deklaration definiert Existenz, Name und Typ einer Variable
- Zuweisung legt Wert der Variable fest
- Wert einer Variable direkt nach Deklaration nicht festgelegt!
- Abkürzende Schreibweise weist Variable gleich bei der Definition einen Wert zu (*Initialisierung*)
- Beispiele: C: int i = 0; fikt. Sprache: var int i = 0;

Vier gleichwertige Beispiele in unserer fiktiven Sprache

Mit zwei Variablen

```
var x : nat;  
var erg : nat;  
x = square(3);  
erg = ggt(x, square(x));
```

```
var x: nat;  
x = square(3);  
var erg: nat;  
erg = ggt(x, square(x));
```

Mit einer Variablen

```
var x : nat;  
x = square(3);  
x = ggt(x, square(x));
```

```
var x: nat = square(3);  
x = ggt(x, square(x));
```

Vier gleichwertige Beispiele in C/Java

Mit zwei Variablen

```
unsigned x;  
unsigned erg;  
x = square(3);  
erg = ggt(x, square(x));
```

```
unsigned x;  
x = square(3);  
unsigned erg;  
erg = ggt(x, square(x));
```

Mit einer Variablen

```
unsigned x;  
x = square(3);  
x = ggt(x, square(x));
```

```
unsigned x = square(3);  
x = ggt(x, square(x));
```

© Prof. Dr. Dieter Nazareth 2023

261

Abgrenzung von Anweisung zu Ausdruck

- Anweisung hat selbst keinen Wert
- Bewirkt aber *Wertänderung* einer Variable
- D. h. *Zustand* dieser Variable ändert sich

Zustandsraum der Daten eines Programms

- Definiert durch alle definierten Variablen des Programms
- Menge der Werte aller Programmvariablen zu bestimmtem Zeitpunkt bildet den Datenzustand des Programms
- Deklaration einer Variable vergrößert den Zustandsraum
- Zuweisung ändert den Wert einer Variable

Auswerten einer Anweisung bedingt Zustandsänderung

- Anfangszustand vor Auswertung, Endzustand danach

© Prof. Dr. Dieter Nazareth 2023

262

Semantik Variablendefinition `var id : t;`

- Erweitere den zu betrachtenden Zustandsraum der Programmdaten um neue Variable `id` vom Typ `t`

Semantik Zuweisung `id = e;`

- Werte Ausdruck `e` bei gegebenem Zustandsraum aus (Anfangszustand)
- Führe Zuweisung aus
 - Ändere im Zustandsraum den Wert der Variable `id` auf den Wert von `e`
 - Falls `e` undefiniert ist, wird gesamter Zustandsraum undefiniert (symbolisiert nicht terminierendes Programm)

Bedeutung

- Mehrere Anweisungen werden hinter einander ausgeführt

Beispiel (C/Java)

- `int x = 0; x++; x = x * x;`

Semantik der sequenziellen Komposition $s_1 s_2$

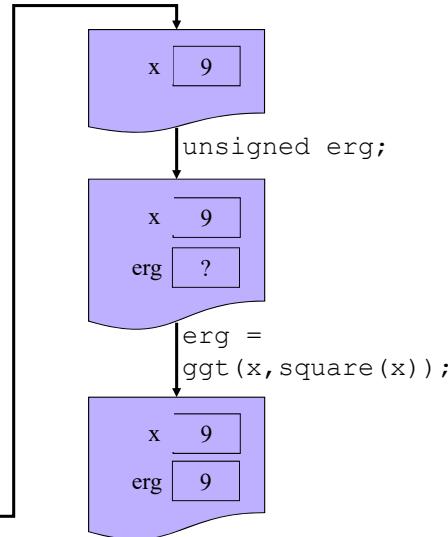
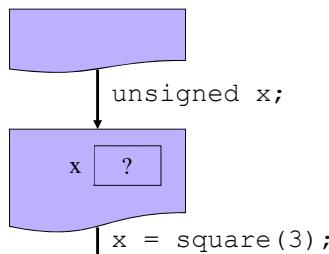
- s_1 und s_2 seien Anweisungen
- Werte s_1 mit gegebenem Zustandsraum aus (Anfangszustand)
- Werte s_2 mit Zustandsraum nach Auswertung von s_1 aus (d. h. auf Endzustand von s_1)

Programmiersprachen – Prozedurale Sprachelemente Variable, Zuweisung, seq. Komposition – Beispiel



Beispiel

```
unsigned x;  
x = square(3);  
unsigned erg;  
erg = ggt(x, square(x));
```



© Prof. Dr. Dieter Nazareth 2023

265

Programmiersprachen – Prozedurale Sprachelemente Bedingte Anweisung – Definition



Bedeutung

- Gegenstück zum bedingten Ausdruck
- Zuweisung in Abhängigkeit des Werts einer Bedingung

Syntax

- C/Java: **if** (b) s1 **else** s2
- Fiktive Programmiersprache: **if** b **then** s1 **else** s2 **fi**
- Dabei b Boolescher Ausdruck, s1, s2 beliebige Anweisungen

Semantik (informell)

- s1, falls Ausdruck b den Wahrheitswert true hat
- s2, falls Ausdruck b den Wahrheitswert false hat
- Falls Ausdruck b den Wert \perp hat, ist Zustandsraum undefiniert

© Prof. Dr. Dieter Nazareth 2023

266

Beispiel

```
unsigned erg, x;  
x = getWert();  \\ Liest Wert von Tastatur ein  
if (x <= 0) erg = 0;  
else {  
    unsigned y = square(x);  
    erg = ggt(y,square(y)); }
```

Bedingte Anweisung vs. bedingter Ausdruck

- Bedingte Anweisung:
`if x<0 then y = -x; else y = x; fi`
- Gleichbedeutend zu bedingtem Ausdruck:
`y = if x<0 then -x else x fi`

Bedeutung

- Quelltextbereich wird mehrfach durchlaufen
- D. h. gleiche ausgeführte Struktur, aber ggf. andere Parameter- und Variablenwerte

Syntax

- C/Java: **while** (b) s
- Fiktive Programmiersprache: **while** b **do** s **od**
- Dabei: b boolescher Ausdruck, s Anweisung

Begriffe

- b heißt *Bedingung* der Wiederholungsanweisung; genauer: *Eintrittsbedingung*
- s heißt *Rumpf* der Wiederholungsanweisung

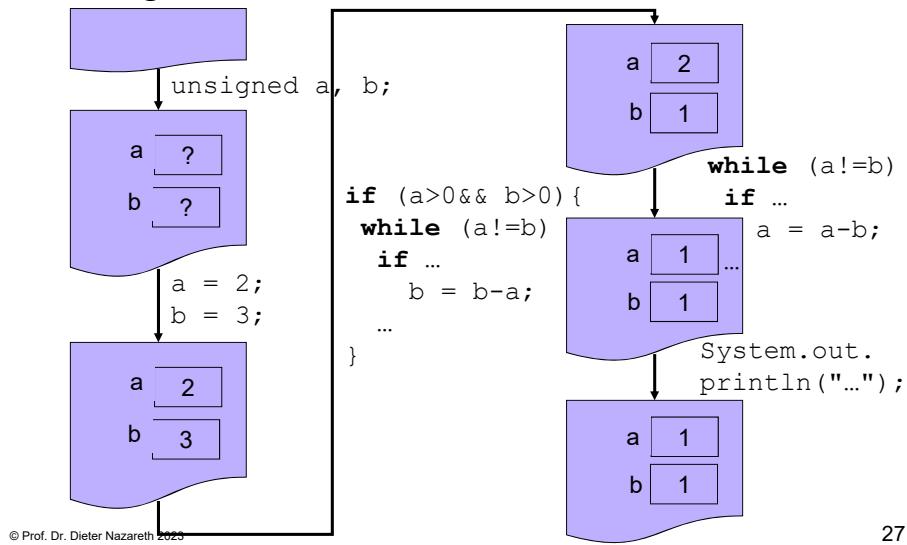
Semantik von while (b) s

1. Wenn Ausdruck b im Anfangszustandsraum den Wahrheitswert `false` hat, dann ändert sich der Zustandsraum nicht
2. Wenn Ausdruck b im Anfangszustandsraum den Wert \perp hat, dann wird der Zustandsraum undefiniert
3. Ansonsten führe s mit Anfangszustandsraum aus
4. Mache Zustandsraum nach Ausführung von s zu Anfangszustandsraum
5. Setze Auswertung mit Schritt 1 fort

Beispiel: GgT iterativ ermitteln

```
unsigned a, b;  
a = 2; b = 3;  
if (a > 0 && b > 0) {  
    while (a != b) {  
        if (a < b)  
            b = b-a;  
        else a = a-b;  
    }  
    System.out.println("GgT ist " + a);  
}  
else System.out.println("Eine Zahl ist " + 0);
```

Änderung des Zustandsraums



© Prof. Dr. Dieter Nazareth 2023

271

Bedeutung

- Andere Variante der Wiederholung
- *Abbruchbedingung* statt Eintrittsbedingung
- Rumpf der Schleife wird mindestens einmal durchlaufen
- In vielen Sprachen:
`repeat` Anweisung `until` (Bedingung)
- Achtung: Bedingung für Wiederholung muss bei `repeat ... until false`, bei `do ... while true` sein!

Syntax

- C/Java: `do s while (b);`
- Äquivalent zu: `s; while (b) s`
- Dabei b Boolescher Ausdruck, s Anweisung

© Prof. Dr. Dieter Nazareth 2023

272

Bedeutung

- Variante der Wiederholung
- Anzahl der Wiederholungen über Zählvariable gesteuert

Syntax

- **for** (s1; b; s2) s3
- Äquivalent zu: s1; **while** (b) {s3; s2}
- Dabei b Boolescher Ausdruck, s1, s2, s3 Anweisungen; oft:
 - s1 Deklaration und Initialisierung der Zählvariable
 - s2 Änderung der Zählvariable

Beispiel (C/Java)

- **for** (i=0; i<10; i++) tuWas();
- Äquivalent zu: i=0; **while** (i<10) {tuWas(); i++}

Beispiel: GgT iterative ermitteln

```
unsigned a, b;  
a = 100; b = 60;  
if (a > 0 && b > 0) {  
    while (a != b) {  
        if (a < b)  
            do b = b-a; while (b > a);  
        else do a = a-b; while (a > b);  
    }  
    System.out.println("GgT ist " + a);  
}  
else System.out.println("Eine Zahl ist " + 0);
```

Programmiersprachen – Prozedurale Sprachelemente

Prozedurdefinition – Bedeutung



Bedeutung

- Kapselt einen Quelltextabschnitt zu unabhängiger Einheit
- Ermöglicht mehrfaches Ausführen, ohne Redundanz im Code
- Analog zu Funktion bei funkt. Programmiersprachen; aber:
 - Rumpf ist eine Anweisung (und kein Ausdruck)
 - Kein Ergebnistyp (weil nichts zurückgegeben wird)

Beispiele

- In fiktiver Programmiersprache:

```
proc p ([var] x1:t1, ..., [var] xn:tn)
begin e end
```
- In C/Java:

```
void p (t1 x1, ..., tn xn) { e }
```

Programmiersprachen – Prozedurale Sprachelemente

Prozedurdefinition – Elemente



Elemente einer Prozedurdefinition

- Schlüsselwort als Kennzeichen der Prozedurdefinition
- Identifikator (d. h. der Name) der Prozedur
- Parameterliste
 - Besteht aus Paaren von Parametername und ParameterTyp
 - Meist durch runde Klammern begrenzt
 - In einigen Programmiersprachen optionales Schlüsselwort (z. B. **var**) als Kennzeichen, dass Referenz auf Parameter übergeben wird und nicht nur der Wert
- Prozedurrumpf; Anfang und Ende gekennzeichnet

Programmiersprachen – Prozedurale Sprachelemente Prozedurdefinition – Schlüsselwort **var** (1)



- Schlüsselwort **var** ist optional
- Parameter *ohne* Schlüsselwort **var**
 - Entspricht formalem Parameter bei Funktion;
 - Dient der Übergabe eines Werts an die Prozedur
 - Unterschied: Darf durch Zuweisungen verändert werden
 - Achtung: Änderung ist in einigen Sprachen nicht erlaubt!
- Parameter *mit* dem Schlüsselwort **var**
 - Bezeichnet als *Variablenparameter* bezeichnet
 - Darf beim Aufruf der Prozedur nur mit Variable belegt werden, nicht mit konkretem Wert oder Ausdruck!
(Siehe später)

Ergebnis der Prozed. indirekt über Parameter zurückgeben!

Programmiersprachen – Prozedurale Sprachelemente Prozedurdefinition – Schlüsselwort **var** (2)



Arten der Verwendung von Variablenparametern

- *Eingabeparameter*
 - Parameter tritt nicht auf linker Seite einer Zuweisung auf
 - Wird also durch die Prozedur nicht verändert
- *Ergebnisparameter*
 - Parameter tritt *nur* auf linker Seite einer Anweisung auf
 - Wert vor Ausführung der Prozedur ist bedeutungslos
- *Transienter Parameter*
 - Parameter tritt *sowohl* auf linker Seite einer Zuweisung auf, als auch in Ausdrücken
 - Wert wird verändert und ausgewertet

Achtung: Hier starke Unterschiede in einzelnen Sprachen!

Programmiersprachen – Prozedurale Sprachelemente Prozedurdefinition – Schlüsselwort **var** (3)



Beispiel in fiktiver Programmiersprache: GgT iterativ

```
proc ggt_iterativ (a:nat, b:nat, var erg:nat)
    \\ erg ist Ergebnisparameter
begin
    if (a > 0) ∧ (b > 0)
        then
            while ¬(a == b) do
                if a < b
                    then repeat b = b-a; until (b ≤ a)
                    else repeat a = a-b; until (a ≤ b)
                fi
            od
            erg = a;
        else erg = 1;
    fi
end
```

© Prof. Dr. Dieter Nazareth 2023

279

Programmiersprachen – Prozedurale Sprachelemente Prozedurdefinition – Schlüsselwort **var** (4)



Beispiel in fiktiver Programmiersprache

```
proc p_demo (var x:int, var y:int, var z:int)
begin
    z = x + z;
    y = x;
end
```

Arten von Parametern

- x Eingabeparameter
- y Ausgabeparameter
- z transienter Parameter

© Prof. Dr. Dieter Nazareth 2023

280

Gegeben

- Prozedurdefinition
proc p ([var] x₁:t₁, ..., [var] x_n:t_n) **begin** s **end**
- Seien e₁, ..., e_n beliebige Ausdrücke bzw. Variablen der Typen t₁, ..., t_n
- Aber: Falls x_i als **var** deklariert ist, muss e_i eine Variable sein

Definition

- Dann ist p(e₁, ..., e_n) eine Anweisung
- p(e₁, ..., e_n) symbolisiert die Ausführung des Rumpfes s von p auf den Werten der Ausdrücke bzw. auf den Variablen e_i

Anmerkung

- Anwendung der Prozedur nur erlaubt, wenn für alle i mit 1 ≤ i ≤ n das e_i vom Typ t_i ist (Typkonformität)

Bedeutung

- Prozedurauftrag ist lediglich Abkürzungsmechanismus
- Hilft, redundanten Quelltext zu vermeiden

Semantikdefinition durch Ersatzdarstellung

- Gegeben
 - Prozedurdefinition
proc p (**var** a: t₁, b: t₂) **begin** s **end**
 - Prozedurauftrag p(x, e), wobei x Variable sei
- Dann gilt: p(x, e) äquivalent zu **var** b:t₂ = e; s[x/a]

Bedeutung der Notation s[x/a]

- Ersetze alle Vorkommen der Variable a in s durch Variable x (Merkhilfe: teile durch a und multipliziere mit x)

Semantik der "normalen" Parameterübergabe wie Funktionssemantik

- **Call by value**
- Zuerst aktuelle Parameter auswerten,
d. h. die Ausdrücke auf Parameterpositionen
- Dann Rumpf der Prozedur auswerten

Semantik der Variablenparameter

- **Call by reference**
- Argumentvariablen des Aufrufs bei Rumpfauswertung
verwenden
- Ändern des Variablenwerts im Rumpf der Prozedur ändert
somit auch den Parameterwert in der Aufrufumgebung
- Ergebnis indirekt über Ausgabeparameter zurückgeben

© Prof. Dr. Dieter Nazareth 2023

283

```
var ergebnis:nat;
ggt_iterativ(input(Tastatur), input(Tastatur), ergebnis);
output(ergebnis, Bildschirm);
ist äquivalent zu
var ergebnis:nat; var a:nat = input(Tastatur);
var b:nat = input(Tastatur);
if (a > 0) ∧ (b > 0)
  then while ¬(a == b) do
    if a < b
      then repeat b = b-a; until (b ≤ a)
      else repeat a = a-b; until (a ≤ b)
    fi
  od
  ergebnis = a;
else ergebnis = 1;
fi
output(ergebnis, Bildschirm);
```

© Prof. Dr. Dieter Nazareth 2023

284

Bisher

- Bisher pragmatische Verwendung von Bezeichnern für Funktionen, Prozeduren, Variablen und Parametern
- Überschaubare, kleine Programme
- Programmelemente leicht unterscheidbar
- Jedes Programm von einem einzigen Autor erstellt

Situation bei großen Programmen

- Mehrere Programmierer
- Potenziell überlappende Verwendung von Bezeichnern

Dilemma

- Nicht jeder Bezeichner sollte im ganzen Programm gültig sein bzw. immer das selbe Programmelement bezeichnen

Pragmatisch

- Strukturieren eines Programms in *Blöcke*
- Jeder innerhalb eines Blockes definierte Bezeichner ist nur in diesem Block (und den darunter liegenden Blöcken) gültig
- Außerhalb des Blockes ist der Bezeichner nicht gültig; es sei, dort ist ein gleich lautender Bezeichner definiert

In formalen Programmiersprachen definieren Regeln

- Wann kann Bezeichner verwendet werden (Gültigkeitsbereich)
- Welches Programmelement benennt ein Bezeichner gerade

Durch Programmkonstrukte implizit definierte Blöcke

- Rumpf von Prozedur und Funktion
- Rumpf einer Wiederholungsanweisung
- Inhalt des then- bzw. des else-Zweiges einer bedingten Anweisung
- Feinere Granularität durch explizite Definition möglich

Definition

- Gegeben sei eine Anweisung s
- Ein **Block** wird dann bezeichnet durch
 - { s } (in C/Java) bzw.
 - **begin** s **end** (in unserer fiktiven Programmiersprache)

Definition **Lebensdauer einer Bindung / eines Bezeichners**

- Block, in dem der Bezeichner definiert wurde (und damit die Bindung entstanden ist)
- Schließt alle Unterblöcke mit ein

Achtung

- Obige Definition weicht ab von der Forderung, dass Bezeichner vor dessen Verwendung deklariert werden muss
- Methodisch ist es aber besser, erst zu definieren und dann zu verwenden!!!

Programmiersprachen – Prozedurale Sprachelemente Bindung – Beispiel: Wie viele Blöcke sind definiert?



```
begin
  proc ggt_it (a: nat, b: nat, var erg: nat) begin
    if (a > 0) ∧ (b > 0)
      then while ¬(a == b) do
        if a < b
          then repeat b = b-a; until (b ≤ a)
          else repeat a = a-b; until (a ≤ b)
        fi
      od
      erg = a;
    else erg = 1;
    fi
  end
  var ergebnis: nat;
  ggt_it(input(Tastatur), input(Tastatur), ergebnis);
  output(ergebnis, Bildschirm);
end
```

© Prof. Dr. Dieter Nazareth 2023

289

Programmiersprachen – Prozedurale Sprachelemente Bindung – Beispiel: Wo gilt welcher Bezeichner?



```
begin
  proc ggt_it (a: nat, b: nat, var erg: nat) begin
    if (a > 0) ∧ (b > 0)
      then while ¬(a == b) do
        if a < b
          then repeat b = b-a; until (b ≤ a)
          else repeat a = a-b; until (a ≤ b)
        fi
      od
      erg = a;
    else erg = 1;
    fi
  end
  var ergebnis: nat;
  ggt_it(input(Tastatur), input(Tastatur), ergebnis);
  output(ergebnis, Bildschirm);
end
```

© Prof. Dr. Dieter Nazareth 2023

290

Programmiersprachen – Prozedurale Sprachelemente Bindung – Beispiel



```
begin
  proc ggt_it (a: nat, b: nat, var erg: nat) begin
    if (a > 0) ∧ (b > 0)
      then while ¬(a == b) do
        if a < b
          then repeat |b = b-a; until (b ≤ a)
          else repeat |a = a-b; until (a ≤ b)
        fi
      od
      erg = a;
    else erg = 1;
    fi
  end
  var ergebnis: nat;
  ggt_it(input(Tastatur), input(Tastatur), ergebnis);
  output(ergebnis, Bildschirm);
end
```

© Prof. Dr. Dieter Nazareth 2023

291

Programmiersprachen – Prozedurale Sprachelemente Bindung – Verschattung, Gültigkeitsbereich (1)



Definition **Verschattung**

- Bindung wird durch erneutes Binden in eingeschlossenem Block überlagert (verschattet)

Definition

Gültigkeitsbereich einer Bindung / eines Bezeichners

- Lebensdauer eines Bezeichners abzüglich aller eingeschlossenen Blöcke, in denen der Bezeichner erneut gebunden ist

© Prof. Dr. Dieter Nazareth 2023

292

Beispiel

```
begin
    var boolean square = true;
    begin
        var nat x = 3;
        fun square (x: int):nat begin
            x * x
        end
        x = square(fak(x));
    end
end
```

Wiederverwendung

- Verschattung von Bezeichnern ist Voraussetzung für Wiederverwendung von Programmstücken
- Bindung der Bezeichner an zugehörigen Block macht Block "robust" gegenüber seiner Umwelt
- Block kann so in jedes Programm eingesetzt werden

Grundregel

- Bindungsstruktur eines Programms so einfach wie möglich halten!
- Durchschaubarkeit sicherstellen

Bindungskonzepte

▪ *Statische Bindung*

- Aufschreibung legt fest, welches semantische Programmelement durch einen Bezeichner referenziert wird
- Zur Übersetzungszeit ist eindeutig klar, welches Programmelement gemeint ist

▪ *Dynamische Bindung*

- Ablauf des Programms entscheidend dafür, welches semantische Programmelement bezeichnet wird
- Bindung erfolgt dynamisch zur Laufzeit
- Verbreitet in objektorientierten Sprachen

Polymorphie, Überladen

- Ein Bezeichner referenziert innerhalb eines einzigen Blocks unterschiedliche Programmelemente
- Unterschiedliche Bindungen für einen Bezeichner gültig
- Konflikt muss spätestens zur Laufzeit des Programms aufgelöst werden

Beispiel

- Funktionsbezeichner square und Boolesche Variable square werden syntaktisch unterschiedlich genutzt (unterschiedliche Signatur)
- Eindeutige Unterscheidung auch ohne Verschattung möglich

```
begin
    var boolean square = true;
    begin
        var int x = 3;
        fun nat square (int x)
            begin
                x * x
            end
            x = square (fak (x));
        end
    end
end
```

Definition **Globale Variable**

- Programmvariable, die in Funktion oder Prozedur verwendet wird, aber nicht in Rumpf oder Parameterliste definiert wird
- Variable muss folglich in einem äußerem Block definiert worden sein
- Zuweisung an globale Variable im Rumpf einer Prozedur bewirkt im äußeren Block einen **Seiteneffekt**

Beispiel: Seiteneffekt

```
var nat x = 2;  
proc trash (int a) begin  
    x = fak(square(a));  
end  
trash(x);
```

Bemerkungen

- Prozeduren/Funktionen mit globalen Variablen nicht robust
- Nicht beliebig in anderen Programmen verwendbar
- Seiteneffekte sind sehr gefährlich, weil Änderungen nicht ausschließlich über die Parameterschnittstelle laufen

Empfehlung

- Globale Variable nur sehr vorsichtig verwenden
- Auf globale Variable möglichst nur lesend zugreifen
- Zugriff auf globale Variable gut dokumentieren

Zusammenhang

- Wiederholte Ausführung von Anweisungen realisierbar sowohl durch Rekursion als auch durch Iteration
- Repetitiven Rekursion ähnlich zu Iteration

Einfache Überführung zwischen beiden Varianten nach folgendem Schema

while b do x = e od	\longleftrightarrow äquivalent	if b then f(e) else x fi end x = f(x);
------------------------------------------------	-------------------------------------	---------------------------------------------------------------------------------------

Problem:

Fast alle interessanten Funktionen nicht repetitiv rekursiv!

Aufgabe

- Linear rekursive in repetitiv rekursive Funktion umwandeln

Lösungsidee

- Linear rekursive Funktion einbetten in allgemeinere Funktion

Beispiel

```
fun fak (x: nat):nat begin
    if x == 0 then 1 else x * fak(x-1) fi
end
```

Problem

- Nachklappern nach Rückkehr vom rekursiven Aufruf

Lösungsidee

- Berechnung schrittweise vor rekursivem Aufruf abarbeiten
- Hilfsparameter einführen

Programmiersprachen – Prozedurale Sprachelemente Rekursion vs. Iteration – Lineare Rekursion (2)



```
fun h_fak
(x: nat, y: nat):nat
begin
  if x == 0
    then y
    else h_fak(x-1,x*y)
  fi
end

fun fak
(x: nat):nat
begin
  h_fak(x, 1)
end

y = fak(x);
```

äquivalent

Achtung:
Nicht Teil unserer Sprache!

```
y = 1;
while ¬(x == 0) do
  (x,y) = (x-1,x*y);
od
```

äquivalent

```
y = 1;
while ¬(x == 0) do
  y = x*y;
  x = x-1;
od
```

Programmiersprachen – Prozedurale Sprachelemente Funktionen in C



Generell

- Viele gängige Sprachen erlauben Modularisierung mittels Funktionen bzw. Prozeduren
- Häufig Mischformen der Konzepte verwendet

Beispiel C: Funktion

```
unsigned fak (unsigned x) {
  int erg=1;
  while (x > 1) {
    erg = erg*x;
    x--;
  }
  return erg;
}
```

Programmiersprachen – Prozedurale Sprachelemente Funktionen in C – Bemerkungen



- Definition von formalen Parametern im Rumpf der Funktion
- Verwendung der Parameter auf linker Seite einer Zuweisung möglich
- Entspricht dem Parameterkonzept von Prozeduren
- Normale und variable Parameter realisierbar
- Realisierung von Prozeduren (d. h. Funktionen ohne Rückgabewert) über speziellen Ergebnistyp `void`
- Ergebnistyp `void` besagt, dass Funktion keinen Rückgabewert liefert

Programmiersprachen – Prozedurale Sprachelemente Funktionen in C – Aufrufsemantik



Call by value

- Werte Argument vor Funktionsaufruf aus, übergib Ergebnis
- Funktionsdefinition: `void test (int x) { ... }`
- Aufruf: `test(3*5);`

Call by reference

- Umsetzung von Variablenparametern
- Realisiert mittels Zeiger-/Adresskonzept
 - Nicht *Wert* der Variable übergeben, sondern *Referenz*
 - Referenz auf Variable entspricht *Adresse* der Variable bzw. *Zeiger* auf die Variable
- Funktionsdefinition: `void test (int *x) { ... }`
- Aufruf: `int a; test(&a);`

Allgemeines

- Konzepte im Wesentlichen ähnlich
- Syntax leicht verschieden
- Funktionen / Prozeduren werden Methoden genannt

Aufrufsemantik

- In C++
 - Analog zu C
 - Wahlweise call by value oder call by reference
- In Java
 - Keine explizite Zeigerarithmetik in Java möglich
 - Für allgemeine Variablen nur call by value
 - Für Objekte: Übergabe der Referenz auf das Objekt!

- Zu-/Anweisung als Kernkonzept prozeduraler Sprachen
- Sinnvoll durch direkten Zugriff auf Speicherzellen
- Verschiedene Arten von Anweisungen verfügbar
- Iterative Wiederholung als neuartiges Konstrukt
- Definition von Prozeduren ermöglicht Modularisierung von Programmen
- Konzept des Blocks erlaubt Einschränkung des Gültigkeitsbereiches von Programmelementen und Bezeichnern
- Rekursive und iterative Wiederholung ineinander wandelbar
- Funktionen und Prozeduren in C, C++, Java ähnlich
- Unterschiede hinsichtlich Aufrufsemantik
- Call by value vs. call by reference

Grundlagen der Informatik I

Überblick



- Was ist Informatik
- Information und Repräsentation
- Algorithmen
- Programmiersprachen
- Datenstrukturen
- Komplexität
- Berechenbarkeit

Datenstrukturen

Überblick



- Motivation
- Begriffe
- Aufzählungstyp
- Produkttyp
- Feld
- Mehrdimensionales Feld
- Einführung in rekursive Datentypen
- Liste
- Baum
- Referenzen

Datenstrukturen Motivation – Entwicklungsprozess



reale Welt



Ausschnitt



Studenten GDI

Abstraktion

Repräsentation

Franz Regensburger
15.3.80
1. Sem.
..

Max Fuchs
10.7.78
1. Sem.
..

Rudi Hettler
1.8.75
3. Sem.
..

Name	Vorname	Geb.Datum	Sem.	Note	WH
Fuchs	Max	10.7.78	1	1.0	nein
Hettler	Rudi	1.8.75	3	4.3	ja
Regensburger	Franz	15.3.80	1	2.3	nein
...					

Verfeinerung

Tag	Monat	Jahr
10	7	78
...		

© Prof. Dr. Dieter Nazareth 2023

309

Datenstrukturen Motivation – Verschiedene Datenstrukturen



Tupel/Produkt

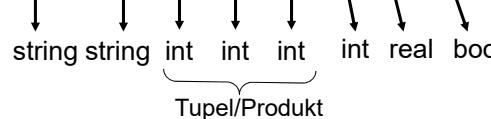
Name	Vorname	Geb.Datum	Sem.	Note	WH
Fuchs	Max	10.7.78	1	1.0	nein
Hettler	Rudi	1.8.75	3	4.3	ja
Regensburger	Franz	15.3.80	1	2.3	nein
...					

>Liste

Verfeinerung

Tag	Monat	Jahr
10	7	78
...		

Abbildung auf Standardtypen



© Prof. Dr. Dieter Nazareth 2023

310

Bedeutung

- Algorithmen arbeiten auf Daten(elementen)
- Zwei Aspekte eines Datenelementes
 - Information, d. h. der eigentliche Wert
 - Darstellung, d. h. die Repräsentation

Definitionen

- Gegeben: Informationssystem (A, R, I)
- Paar (r, a) mit $r \in R, a \in A$ mit $I(r) = a$ heißt **Objekt, Datum** oder **Datenelement**
- Menge von (gleichartigen) Datenelementen heißt **Typ**

Basistypen von vielen Programmiersprachen

- Z. B. `int` meist als gegeben vorausgesetzt
- Daten des Typs `int` sind z. B. `(1,1), (2,2)`, usw.

Ampelfarben

- Repräsentation sei die Menge der Zeichenreihen $\{\text{rot}, \text{gelb}, \text{grün}\}$
- Information seien die entsprechenden Farben
- Dann sind (rot, \bullet) , (gelb, \bullet) , $(\text{grün}, \bullet)$ Daten des entsprechenden Informationssystems

Definitionen **Datenstruktur, Rechenstruktur**

- Menge von Datentypen, d. h. ein oder mehrere Datentypen
- Zusammen mit zugehörigen charakteristischen Funktionen über diesen Typen

Beispiel: Datenstruktur **BOOL** der Booleschen Werte

- $T = \{\text{boolean}\}$ ist die Menge der Typen
- $F = \{\text{true}, \text{false}, \neg, \wedge, \vee\}$ ist die Menge der Funktionen auf den Daten dieser Typen

Beispiel: Datenstruktur **NAT** der natürlichen Zahlen

- $T = \{\text{nat, boolean}\}$ ist die Menge der Typen
- $F = \{\text{true, false, } \neg, \wedge, \vee, \text{zero, succ, pred, add, mult, sub, div, } ==, <\}$ ist die Menge der Funktionen auf den Daten dieser Typen
- Zugehörige Funktionalitäten
 - zero: nat;
 - succ: nat \rightarrow nat;
 - pred: nat \rightarrow nat;
 - add: nat \times nat \rightarrow nat;
 - sub: nat \times nat \rightarrow nat;
 - div: nat \times nat \rightarrow nat;
 - mult: nat \times nat \rightarrow nat;
 - $==:$ nat \times nat \rightarrow bool;
 - $<:$ nat \times nat \rightarrow bool;
- Entsprechende Eigenschaften
 - $\text{succ}(x) = x+1$ $\text{pred}(x) = x-1$ usw.

Statische Datentypen

- Aufbau und Größe bleibt über die Lebensdauer konstant
- Nur die Werte sind variabel
- Konstanter Speicherbedarf
- Beispiele: nat, char, int, real
- Unterscheidung zwischen *einfachen Typen* (z. B. char) und *zusammengesetzten Typen* (z. B. Tupel)

Dynamische Datentypen

- Aufbau und Anzahl der Komponenten sind variabel
- Werte und Struktur während Programmablauf änderbar
- Variabler Speicherbedarf
- Beispiele: Zeichenreihen (String), Listen, Mengen

Bedeutung

- Datentyp mit endlicher Datenmenge definierbar durch Aufzählen der Elemente
- Bezeichnung: *Aufzählungstyp*, engl. *Enumeration*

Syntax

- **type t = {x₁, ..., x_n};**
- Deklaration eines neuen Typen t
- Deklaration von n Bezeichner x₁, ..., x_n vom Typ t (Konstante)
- Zusätzlich steht eine Gleichheitsfunktion auf dem Typ zur Verfügung, d.h. == : t x t → bool. In manchen Sprachen wird auch ein Vergleichsoperator ≤ definiert (festgelegt durch Aufschreibungsreihenfolge).

Beispiel:

Einführung des Typs color mit Identifikatoren für Farben:

```
type color = {rot, gelb, blau, grün, schwarz};  
var ampel:color; ...  
if ampel == rot then ... fi
```

Einführung einer Datenstruktur WAHR:

```
type wahr = {nein, ja};  
fun und (x: wahr, y: wahr):wahr  
begin if x == nein ∨ y == nein then nein  
else ja fi  
end
```

Syntax (C/Java)

- enum <enumtype>
{ <enum-element>, <enum-element>, ... }
- Dabei sei:
 - enum ist seit Java 5 reserviertes Wort
 - <enumtype> bezeichnet den neuen Typ
 - <enum-element> bezeichnet Datenwert
- Auch der Datentyp char könnte durch einen Aufzählungstyp definiert werden.
- In unserem Einführungsbeispiel könnten wir die Monate auch durch einen Aufzählungstyp definieren.

Bedeutung

- Datenelement besteht aus verschiedenen Einzelelementen
- Werte der Einzelelemente voneinander unabhängig
- Produkttyp beschreibt *kartesisches Produkt* der Einzeltypen

Syntax (in Quasiprogrammiersprache)

- **type** t = cons (sel₁: t₁, ..., sel_n: t_n);

Eigenschaften

- Konstruktor: cons: t₁ × ... × t_n → t
- Selektoren: sel_i: t → t_i für 1 ≤ i ≤ n
- Es gilt: sel_i (cons(a₁, ..., a_n)) = a_i

Beispiel: Rationale Zahlen als Tripel

```
type rat = mk_rat (vorzeichen: bool, zähler: nat, nenner: nat);
var z: rat;
z := mk_rat(true, 1, 2);      \| Konstruktion mit Konstruktor
vorzeichen(z) := true; \| Konstruktion über Selektoren
zähler(z) := 1;
nenner(z) := 2;
...
if vorzeichen(z) then vorzeichen(z) := false fi ...
```

Auf den expliziten Konstruktor kann eigentlich verzichtet werden (wird in vielen Sprachen auch nicht fest definiert), weil selektive Konstruktion via Selektoren möglich.

Beispiel: Studenten GDI

```
type student = mk_stud ( name: string,
                          vorname: string,
                          geburtsdatum: datum,
                          semester: nat,
                          note: real,
                          wiederholer: bool );

type datum = mk_datum ( tag: nat,
                        monat: nat,
                        jahr: nat );

var stud1: student;

stud1 := mk_stud( „Fuchs“, „Max“, mk_datum(10,7,78), 1, 1.0, false);
...
semester(stud1) := semester(stud1) + 1;
```

Bedeutung

- Semantisch gesehen Spezialfall des Produkttyps
- Alle Einzelkomponenten sind vom selben Typ

Syntax

Sei n größer 1 und t ein beliebiger Typ. Dann wird durch

type feld = array [n] of t;

ein **Feldtyp** feld der Länge n mit Elementen vom Typ t deklariert. Als Konstruktor für Felder verwenden wir die geschweiften Klammern:

var f: feld := {w₁,...,w_n};

Die Elemente w_i müssen alle vom Typ t sein. Die Anzahl muss mit der Feldlänge übereinstimmen.

Datenstrukturen

Feld (Array)



Auf die Elemente des Feldes kann über die Indizes folgendermaßen zugegriffen werden:

f[i]

Beispiel: Datum

```
type datum = array [3] of int;
stud1 := mk_stud( „Fuchs“, „Max“, {10,7,78}, 1, 1.0, false);
var d: datum := {1,8,75};
stud2 := mk_stud( „Hettler“, „Rudi“, d, 3, 4.3, true);
...
if d[1] == 1 then print(„Januar“); fi
...
if geburtsdatum(stud2)[2] < 75 then exmatrikulation(stud2); fi
```

Datenstrukturen

Feld – Eigenschaften



- Index beginnt in vielen Sprachen bei 0, nicht bei 1!!!
- Indexbereich also $0 \leq i \leq n-1$ für Feld der Größe n
- Indizes über Feldgrenze hinaus nicht zulässig;
Bereichsüberschreitung wird nicht in allen Sprachen geprüft!
- Bearbeiten der Einzelfelder über Wiederholung fester Länge, d. h. for-Schleife
- Feld ist in vielen Sprachen statischer Datentyp;
d. h. Größe nachträglich dann nicht mehr änderbar
- Sprachunterschiede beim Zuweisen eines Felds an Variable
 - Ganzes Feld auf einmal zuweisen
 - Neues Feld anlegen, Einzelfelder umkopieren

Beispiel: Studenten

```
type stud_liste = array [50] of Student;
var gdi: stud_liste;
gdi[0] := mk_stud( „Fuchs“, „Max“, {10,7,78}, 1, 1.0, false);
gdi[1] := mk_stud( „Hettler“, „Rudi“, {1,8,75}, 3, 4.3, true);
...
gdi[39] := mk_stud( „Regensburger“, „Franz“, {15,3,80}, 1, 2.3, false);
for i := 40 to 49
  do
    name(gdi[i]) := „nn“;
    vorname(gdi[i]) := „nn“;
    geburtsdatum(gdi[i]) := {0,0,0};
    semester(gdi[i]) := 0;
    note(gdi[i]) := 0.0;
    wiederholer(gdi[i]) := false;
  od
...

```

© Prof. Dr. Dieter Nazareth 2023

325

Bedeutung

- Auch geschachtelte Felder möglich
- Mehrere Dimensionen

Syntax

type feld = array [n₁, ..., n_m] of t;

Der Typ feld ist damit ein ***m-dimensionales Feld*** mit Elementen aus t. Dies entspricht folgender geschachtelten Typdeklaration:

```
type mfeld = array [nm] of t;
...
type 2feld = array [n2] of 3feld;
type feld = array [n1] of 2feld;
```

© Prof. Dr. Dieter Nazareth 2023

326

Als Konstruktor verwenden wir geschachtelte geschweifte Klammern. Für die Selektion schreiben wir abkürzenderweise:

$$f[i_1, \dots, i_m] \quad (= f[i] \dots [i_m])$$

Beispiel Matrizen

```
type matrix = array [max,max] of int;  
  
proc add (var m: matrix, n: int)  
begin  
    var i: int; var j: int;  
    for i := 0 to max-1 do  
        for j := 0 to max-1 do  
            m[i,j] = m[i,j] + n;  
        od  
    od  
end
```

© Prof. Dr. Dieter Nazareth 2023

327

Bedeutung

- Datenelemente sind das Ergebnis der Vereinigung zweier Typen.
- *Direkte Summe* oder *Vereinigung (union)* der beiden Typen.
- In Pascal heißt dieser Typ *variabler record*.

Syntax (in Quasiprogrammiersprache)

- Seien t_i beliebige Typen, cons_i und sel_i beliebige Bezeichner.
- **type** $t = \text{cons}_1(\text{sel}_1: t_1) \mid \dots \mid \text{cons}_n(\text{sel}_n: t_n);$
- t ist der neu deklarierte Summentyp

© Prof. Dr. Dieter Nazareth 2023

328

Eigenschaften

- Der Typ t entspricht der *disjunkten Vereinigung* der Typen t_1, \dots, t_n .
- Die Bezeichner cons_i werden *Konstruktoren* genannt.
- Konstruktoren haben folgende Funktionalität: $\text{cons}_i: t_i \rightarrow t$
- Die Funktionssymbole sel_i bezeichnet man als *Selektoren*.
- Selektoren haben folgende Funktionalität: $\text{sel}_i: t \rightarrow t_i$
- $\text{sel}_i(\text{cons}_i(a_i)) = a_i$
- $\text{sel}_i(\text{cons}_j(a_j)) = \perp \quad \text{falls } i \neq j$

Neben den Konstruktoren und Selektoren wird implizit eine strikte Testfunktion

. **in** $\text{cons}_i: t \rightarrow \text{bool}$

in infix-Notation zur Verfügung gestellt. Sie wird *Variantentest* genannt und dient zum Testen welche Variante vorliegt. Es gilt:

$\text{cons}_i(a_i) \text{ in } \text{cons}_i = \text{true}$

$\text{cons}_i(a_i) \text{ in } \text{cons}_j = \text{false} \quad \text{falls } i \neq j$

Beispiel: Varianter Geldtyp

```
type money = mk_dm (dm: real) | mk_euro (euro: real);
var konto: money;
konto := mk_dm(200);                                \\\ Konstruktion mit Konstruktor
proc dm2euro (var c: money) begin
    if c in mk_dm then c := mk_euro(dm(c)/1.95583); fi
end
proc euro2dm (var c: money) begin
    if c in mk_euro then c := mk_dm(euro(c)*1.95583); fi
end
dm2euro(konto);
euro2dm(konto);
```

Bemerkungen:

- Vor der Selektion sollte grundsätzlich mit dem Variantentest die aktuelle Variante überprüft werden.
- Summentypen können durch Produkttypen dargestellt werden (i.a. mit erhöhtem Specheraufwand):

```
type currency = {dm, euro};
type money = mk_money ( curr: currency, amount: real );
proc dm2euro (var c: money) begin
    if curr(c) == dm then c := mk_money(euro,amount(c)/1.95583); fi
end
```
- Der Speicherbedarf eines Summentyps ist der Speicherbedarf des längsten Variantentyps
- Der Speicherbedarf des Produkttyps ist die Summe der Speicherbedarfe aller Komponenten.

Variantenbildung tritt besonders in Verbindung mit Produkttypen auf. Deshalb verwenden wir hier eine abkürzende Schreibweise.

Bsp.: Koordinaten

```
type polar = mk_polar ( l: real,  
                         phi: real );  
  
type kart = mk_kart ( x: real,  
                         y: real );  
  
type koord = polar_koord (get_pol: polar) | kart_koord (get_kart: kart);
```

↓
Abkürzung

```
type koord = mk_polar (l: real, phi:real) |  
                     mk_kart (x: real, y:real);
```

Bisher betrachtete Datentypen

- Anzahl der enthaltenen Elemente beschränkt
- Anzahl statisch, d. h. zur Deklarationszeit bekannt, danach unveränderbar
- Maximaler Speicherbedarf steht bei Programmierung fest

Probleme: Beispiel Studentenliste

- Bei Programmierung max. Anzahl der Studenten schätzen
- Schätzung zu hoch: Ressourcen verschwendet
- Schätzung zu niedrig: Umfasst nicht alle benötigten Daten

Abhilfe

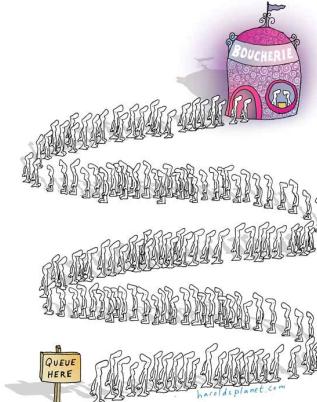
- Rekursive Definition von Typen (analog zu rek. Funktionen)
- Erlaubt Darstellung von Daten variabler Länge

Datenstrukturen

Liste – Bedeutung



- Synonyme Begriffe: *Liste*, *Sequenz*, *lineare Liste*
- Linear geordnete Sammlung von Elementen. Lineare Ordnung bezieht sich auf Position in der Liste
- Jedes Listenelement hat einen *Vorgänger*, einen *Nachfolger*
- Liste ansonsten nicht weiter strukturiert
- Listengröße nicht statisch festgelegt dynamisch nach Bedarf änderbar
- Alle Listenelemente haben denselben Typ



Datenstrukturen

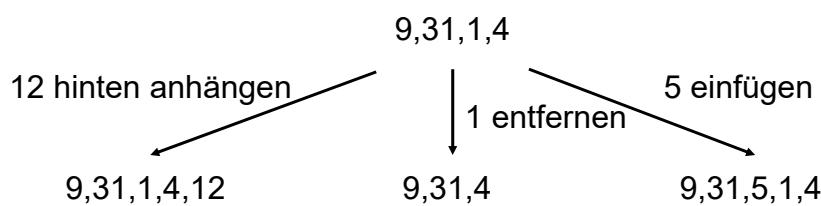
Liste – Operationen



Operationen mit Liste und Element

- Element in Liste einfügen
- Element an Liste anhängen
- Element aus Liste entfernen

Beispiel: Liste von Integerwerten



Gegeben

- Menge T von Elementen eines beliebigen Typs t
- Alle Elemente dabei vom gleichen Typ!

Aufgabe

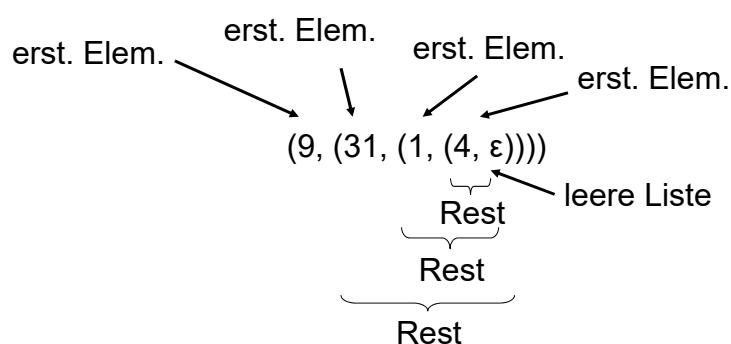
- Bilde aus diesen Elementen eine Liste

Vorgehen (informell): Liste ist...

- Entweder leer oder
- Paar aus einem Element aus T (das erste Element) und einer Liste von Elementen aus T (dem Rest der Liste)

Beispiel

- Aufbau der Liste mit den Elementen 9, 31, 1, 4



Die informelle Definition lässt sich mit unseren Datentypdefinitionen folgendermaßen umsetzen:

- Das Paar aus dem ersten Element und dem Rest der Liste kann mit Hilfe eines Produkttyp definiert werden:

```
type ne_sequ_t = append ( first: t,  
                           rest: sequ_t);
```

- Die leere Liste lässt sich sehr einfach über einen nullstelligen Produkttypen realisieren:

```
type e_sequ_t = empty;
```

- Die oder-Verknüpfung kann dann ganz einfach über einen Summentypen (Variante) realisiert werden.

```
type sequ_t = mk_esequ (empty: e_sequ_t) |  
               mk_nesequ (ne_s: ne_sequ_t)
```

- Die beiden Typen ne_sequ_t und e_sequ_t wurden lediglich als Abkürzung eingeführt. Über geschachtelte Typdeklarationen schreiben wir deshalb kürzer:

```
type sequ_t = empty | append ( first: t,  
                               rest: sequ_t);
```

- Über die Konstruktoren lassen sich Listen aufbauen:

```
var l: sequ_int;  
l := append(9,append(31,append(1,append(4,empty))));
```

- Mit Hilfe der Selektoren mit vorangeschaltetem Variantentest lassen sich die Elemente wieder aus der Liste holen:

```
var i: int;  
if l in append then i := first(l) fi
```

Der Datentyp sequ_t bildet zusammen mit seinen charakteristischen Funktionen die Rechenstruktur der Listen. Folgende Funktionen werden üblicherweise durch die Struktur zur Verfügung gestellt:

last: sequ_t → t	liefert das letzte Element der Liste
lrest: sequ_t → sequ_t	liefert Liste ohne letztes Element
make: t → sequ_t	macht einelementige Liste
stock: sequ_t x t → sequ_t	Element hinten anhängen
conc: sequ_t x sequ_t → sequ_t	Konkatenation zweier Listen
length: sequ_t → nat	Länge der Liste
is_in: t x sequ_t → bool	prüft, ob Element in einer Liste ist

Einige Funktionen wollen wir jetzt beispielhaft definieren:

```
fun length (s: sequ_t): nat
begin
  if s in empty then 0
    else 1 + length (rest(s)) fi
  end

  fun make (x: t) : sequ_t begin append(x, empty) end

  fun conc (r: sequ_t, s: sequ_t) : sequ_t begin
    if r in empty then s
      else conc(lrest(r),append(last(r),s)) fi
  end
```

```
fun last (s: sequ_t): t
begin
    if rest(s) in empty then first(s)
        else last(rest(s)) fi
end

proc length (s: sequ_t, var erg: nat)
begin
    erg := 0;
    while s in append do
        erg := erg + 1;
        s := rest(s);
    od
end
```

```
fun is_in (x: t, s: sequ_t): bool
begin
    if s in empty then false
    else if first(s) == x then true
    else is_in(x, rest(s)) fi
end
```

Problem

- Nachweis, dass Element i in Liste ist, muss ggf. jedes bestehende Element der Liste anfassen, falls Element nicht in Liste enthalten ist

Abhilfe

- Liste sortieren, z. B. aufsteigend
- Liste dann von vorne durchgehen
 - Falls $i ==$ aktuelles Element: i ist schon drin; fertig
 - Falls $i <$ aktuelles Element: i ist noch nicht in Liste
 - Falls $i >$ aktuelles Element: i evtl. noch im Rest der Liste

Zu tun: Verfahren für Aufbau einer sortierten Liste

Aufgabe

- Erzeuge aus unsortierter Liste ul eine sortierte Liste sl , die die gleichen Elemente (ohne Duplikate) enthält

Idee

- Füge Elemente i aus ul einzeln "an richtiger Stelle" in sl ein (Sortieren durch Einfügen, insertion sort)

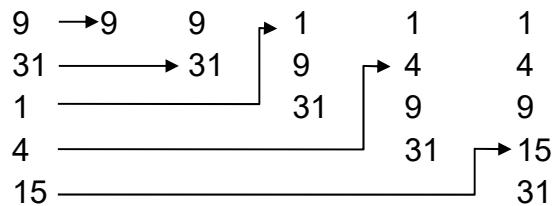
Vorgehen (informell)

- Falls Liste sl leer
 - Erzeuge einelementige Liste aus i (ist immer sortiert!)
- Falls Liste sl nicht leer
 - Finde "richtige Stelle", an die Element i hingehört
 - Füge i hier in die Liste ein

Datenstrukturen Liste – Sortieren (3)



Beispiel



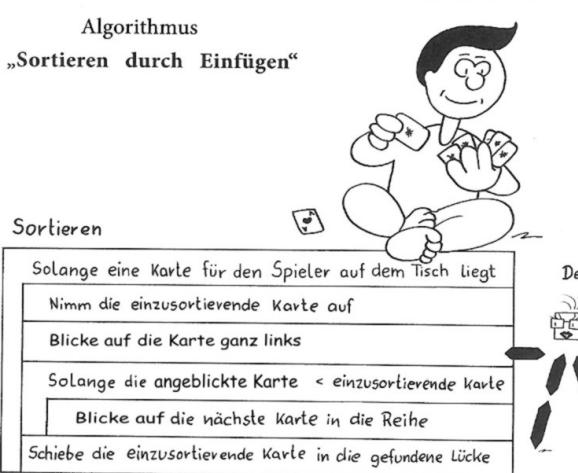
Benötigt

- Verfahren zum Abbau der einzelnen Elemente aus gegebener Liste
- Verfahren zum Einfügen von i in die Liste an richtiger Stelle
- Vergleichsoperator $<$

Datenstrukturen Liste – Sortieren (4)



Algorithmus
„Sortieren durch Einfügen“



Datenstrukturen Liste – Sortieren (5)



```
fun sort (s: sequ_t) : sequ_t begin
    if s in empty then s
        else insert(first(s),sort(rest(s)))
    fi
end

fun insert (x: t, s: sequ_t) : sequ_t begin
    if s in empty then append(x,empty)
    else if x < first(s) then append(x,s)
        else append(first(s),insert(x,rest(s)))
    fi
fi
end
```

Datenstrukturen Liste – Sortieren (6)



Das Suchen nach einem Element kann damit beschleunigt werden:

```
fun is_in (x:t, s: sequ_t) bool begin    \! s ist sortiert
    if s in empty then false
        else if x == first(s) then true
        else if x < first(s) then false
        else is_in(x,rest(s)) fi fi
    fi
end
```

Ist Element i in sortierter Liste l mit n Elementen?

- Best Case
 - i ist gleich das erste Listenelement
 - 1 Vergleich
- Worst Case
 - i ist letztes Element von l, oder größer als dieses
 - Pro Listenelement 3 Vergleiche, insgesamt $3 \cdot n$
- Average Case
 - Im Mittel ist die halbe Liste zu durchsuchen
 - Also ca. $(3 \cdot n)/2$ Vergleiche



- Durch die Einschränkung der Rechenstruktur der Listen auf gewisse Operationen erhält man die Datenstruktur *Stapel* (*Stack*, *Keller*):

```
type stack_t = empty | push ( top: t, rest: stack_t);
```

height: stack_t → int (berechnet die Höhe des Stacks)
- Ansonsten stehen keine Zugriffsfunktionen zur Verfügung.
- Stapel sind nach dem sog. LIFO-Prinzip (last in first out) aufgebaut.
- Sie dienen dem systematischen Ablegen von Elementen und dem Zugriff in der umgekehrten Reihenfolge der Ablage.

Bedeutung

- Dynamische Datenstruktur, damit potenziell unendlich groß
- Alle Datenelemente vom selben Typ
- Baum ist in sich strukturiert (im Gegensatz zur Liste)
- Gut geeignet zum effizienten Suchen in sortierten Datenbeständen
- Jeder Knoten im Baum speichert
 - Datenwert
 - Strukturinformation
- Baum hat hierarchische Struktur
- Bäume wachsen von oben nach unten

Gegeben

- Menge T von Elementen eines beliebigen Typs t
- Alle Elemente dabei vom gleichen Typ!

Aufgabe

- Bilde aus diesen Elementen einen Baum

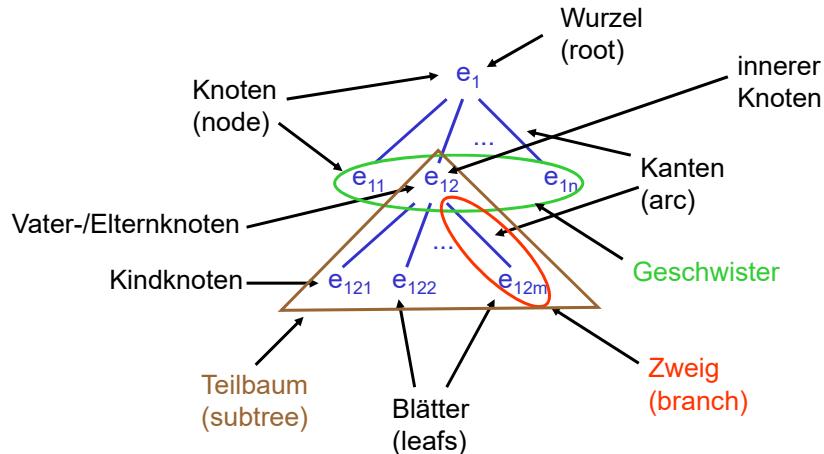
Vorgehen (informell): Baum ist...

- Entweder leer oder
- Knoten aus einem Element aus T und einer (möglicherweise leeren) Menge von Kindbäumen

In der Praxis

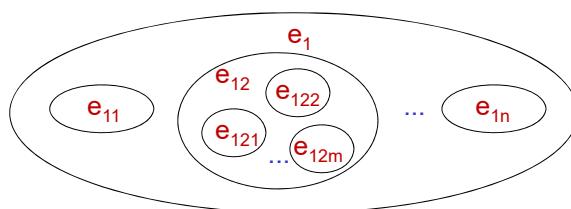
- Kindbäume geordnet
- Bis zu zwei Kindbäume möglich; *Binärbaum*

Graphische Darstellung von Bäumen



Weitere Darstellungsformen

- Euler-Venn Diagramme (geschachtelte Mengen)

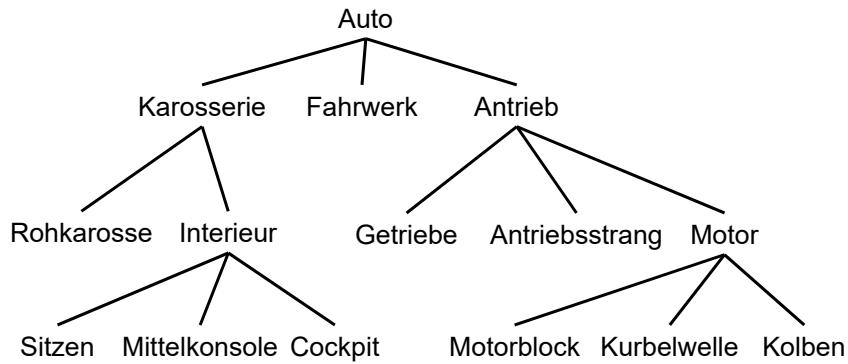


- Geschachtelte Klammerung
 $e_1 \{e_{11}, e_{12} \{e_{121}, e_{122}, \dots, e_{12m}\}, \dots, e_{1n}\}$

Datenstrukturen Baum – Verwendung (1)



Strukturabaum



© Prof. Dr. Dieter Nazareth 2023

357

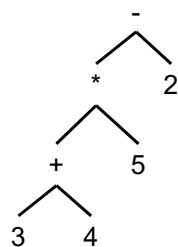
Datenstrukturen Baum – Verwendung (2)



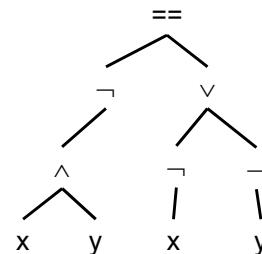
Operatorbaum

- Innere Knoten == Operatoren
- Blätter == Operanden

$$(3 + 4) * 5 - 2$$



$$\neg(x \wedge y) == \neg x \vee \neg y$$

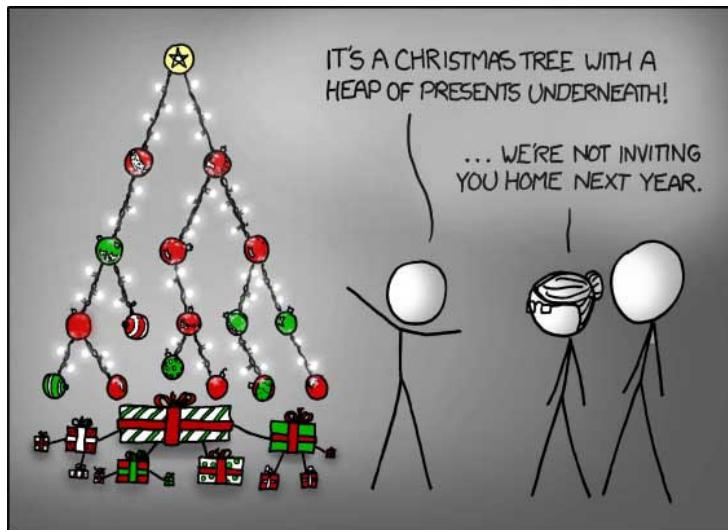


$$-(*(+(3, 4), 5), 2)$$

$$==(\neg(\wedge(x, y)), \vee(\neg(x), \neg(y)))$$

© Prof. Dr. Dieter Nazareth 2023

358



- **Binäräbäume** nehmen eine wichtige Rolle in der Informatik ein und können folgendermaßen rekursiv definiert werden:

```
type bintree_t =  
    emptytree |  
    mktree ( left: bintree_t, node: t, right: bintree_t);
```

- Große Anzahl von Operationen auf Bäumen.
- Meist muss die Operation alle Knotenelemente erreichen.
- Baum muss vollständig durchlaufen werden.
- Unterschiedliche Durchlauftechniken möglich.

- *Vorordnung (preorder)*
 - Zuerst Knoten des Baumes bearbeiten
 - Linken Teilbaum bearbeiten
 - Zuletzt rechten Teilbaum bearbeiten
- *Inordnung (inorder)*
 - Linken Teilbaum bearbeiten
 - Knoten bearbeiten
 - Rechten Teilbaum bearbeiten
- *Nachordnung (postorder)*
 - Linken Teilbaum bearbeiten
 - Dann rechten Teilbaum bearbeiten
 - Zuletzt Knoten des Baumes bearbeiten

- Umformung eines Baumes in eine Sequenz nach der Vorordertechnik:

```
fun preorder (t: bintree_t) sequ_t begin
    if t in emptytree then empty
    else append(node(t),
                conc(preorder(left(t),preorder(right(t))))))
    fi
end
```
- Addition eines Wertes auf alle Baumknoten:

```
fun add (t: bintree_int, x: int) bintree_int begin
    if t in emptytree then t
    else mk_tree(add(left(t),x),node(t)+x,add(right(t),x))
    fi
end
```

Datenstrukturen

Baum – Eigenschaften



- Baum ist gerichteter azyklischer Graf
- Zu jedem Knoten gibt es von der Wurzel aus genau einen Weg (*Pfad*)
- Niveau $N(k)$ des Knotens k ist Anzahl der Knoten des Pfads von der Wurzel zum Knoten k
- Wurzel hat Niveau 1
- Höhe $H(b)$ des Baumes b definiert als $H(b) := \max\{N(k) \mid k \text{ Knoten von } b\}$
- Baum heißt *geordnet*, wenn für jeden Knoten dessen Nachfolger eine feste Reihenfolge haben
- Anzahl der direkten Nachfolger eines Knotens definiert dessen (*Verzweigungs-*) *Grad*
- Höchster Grad aller Knoten im Baum b bestimmt *Grad* des Baumes; $\text{Grad}(b) = \max\{\text{Grad}(k) \mid k \text{ Knoten von } b\}$

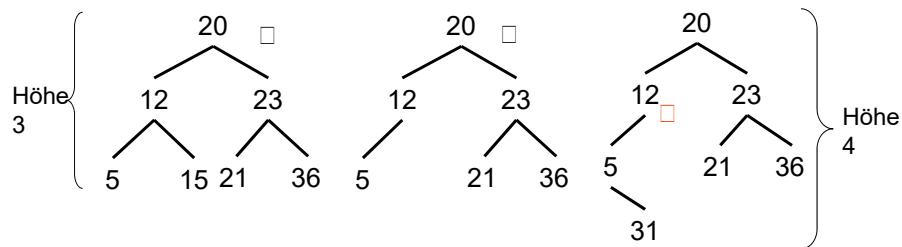
Datenstrukturen

Baum – Eigenschaften von Binärbäumen



- Maximaleigenschaften eines Binärbaums der Höhe h
 - Maximal $2^h - 1$ Knoten, 2^{h-1} Blätter
 - Kann also maximal $2^h - 1$ Elemente speichern
- Minimaleigenschaften eines Binärbaums der Höhe h
 - Mindestens h Elemente, mindestens 1 Blatt
 - Baum dann zur Liste entartet
- Höhe h eines Baums mit n Knoten: $\log_2(n) < h \leq n$
- Baum heißt *vollständig ausgeglichen*, wenn sich für jeden Knoten die Zahlen der Knoten in dessen linken und rechten Unterbaum um höchstens 1 unterscheiden
- Höhe eines vollständig ausgeglichenen Baumes b mit n Knoten: $h(b) = \lceil \log_2(n+1) \rceil$

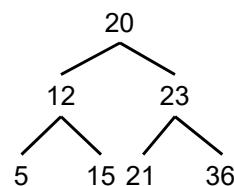
Beispiel (vollständig ausgeglichen)



Sortierter Baum (Suchbaum)

- Baum b heißt *sortiert*, wenn für jeden Knoten k aus b gilt
 - Alle Knotenelemente des linken Teilbaums von k sind kleiner als k
 - Alle Knotenelemente des rechten Teilbaums von k sind größer als k
- Obige Bedingung gefordert für jeden Knoten des Baumes
- Prüfung somit rekursiv für gesamten Baum

Sortierter Baum ermöglicht schnelle Suche über großen Datenbeständen!



Mit folgender Funktion können wir feststellen, ob ein Element in einem Suchbaum enthalten ist:

```
fun is_in (x:t, t: bintree_t) bool begin
    if t in emptytree
        then false
    else if node(t) == x then true
    else if x < node(t) then is_in(x, left(t))
    else is_in(x, right(t)) fi fi fi
end
```

Bemerkungen

- Pro abgeprüfter Baumebene sind maximal 3 Vergleiche durchzuführen
- Für Baum b mit Höhe $H(b)$ fallen maximal an
 - $3 * H(b) - 1$ Vergleiche, wenn Element im Baum enthalten
 - $3 * H(b) + 1$ Vergleiche, wenn Element nicht im Baum drin

	Anzahl der Vergleichsoperationen				
n	10	100	1000	10000	100000
sortierte Liste	30	300	3000	30000	300000
vollständig ausgewichener Suchbaum	12	21	30	42	51

- Bei zur Liste entartetem Baum gleicher Aufwand wie Liste!

- Dynamischer Datentypen: Speicherbedarf kann über die Laufzeit des Programms variieren kann.
- Rekursiven Datentypen erlauben uns zwar Elemente beliebiger Größe aufzubauen, aber nicht diese Elemente selektiv zu verändern.
- Wenn man z. B. eine unsortierte Liste sortiert, dann ist man i. a. an der unsortierten Liste nicht mehr interessiert.
- Unser Sortierfunktion (sort : sequ_t → sequ_t) hat jedoch eine neue Liste erzeugt.

Bsp.: (Sortieren von Listen)

```
var old: sequ_int;  
var new: sequ_int;  
  
old := append(9,append(31,append(1,empty)));  
new := sort(old);
```

Nach sort(old) gilt:

```
new == append(1,append(9,append(31,empty)));
```

Problem:

- keine Konstrukte zum selektiven Verändern einer Liste (z.B. zwei Elemente tauschen).
- Listenelemente müssen immer mit den Konstruktoren empty oder append erzeugt werden, oder werden durch den Zuweisungsoperator := umkopiert.
- Bei Feldern besteht die Möglichkeit Elemente selektiv einzeln anzusprechen und damit einzeln zu verändern.
- Über die Notation können wir auf das i-te Element eines Feldes zugreifen, ohne diesem Element einen konkreten Namen zu geben.
- $a[i]$ kann als Verweis (Referenz) auf das i-te Element des Feldes aufgefasst werden.

Idee:

- Lege alle Variablen, die wir in unserem Programm benötigen, quasi anonym in einem großen Feld ab.
- Spreche Variablen über den Index an (= referenzieren).

Bsp.:

```
var eingabe: nat;  
var ergebnis: nat;  
  
eingabe := 5;  
ergebnis :=  
fak(eingabe);
```



```
var speicher: array  
[100] of nat;  
speicher[0] := 5;  
speicher[1] :=  
fak(speicher[0]);
```

Wir wollen dieses Konzept des Referenzieren verallgemeinern und führen dazu Referenztypen ein.

- Der Typ

ref t

ist der Typ aller Referenzen auf Elemente vom Typ t.

- Feldtyp ist ein statischer Datentyp, d.h. seine Größe müssen wir fest definieren und können wir zur Laufzeit des Programms nicht mehr dynamisch erweitern.
- Deswegen führen wir jetzt einen Mechanismus ein, um Elemente dynamisch während der Ausführung eines Programms erzeugen zu können.

- Der Befehl

new(t)

erzeugt ein neues Element (Variable) vom Typ t und liefert eine Referenz auf das Element.

- Element ist anonym ist und muss über einen Referenztyp angesprochen:

var name: ref t;

name := new(t);

- Um den Wert des Elements ansprechen zu bekommen müssen wir es dereferenzieren.
- Zu diesem Zweck sei in unserer Sprache für jeden Referenztyp **ref t** folgende Funktion vorhanden:

- Dereferenzierungsfunktion:

deref: ref t → t

- Die Funktion hat die Eigenschaft, dass `deref(r)` das Element liefert, auf das `r` verweist.
- Referenzen können (wie alle Werte) zugewiesen und verglichen werden.
- Bsp.:

```
var eingabe: ref nat;
var ergebnis: ref nat;
eingabe := new(nat);
ergebnis := new(nat);
deref(eingabe) := 5;
deref(ergebnis) :=
fak(deref(eingabe));
```

Eine ausgezeichnete Rolle spielt der Bezeichner **nil**, der ein vorgegebenes Referenzelement bezeichnet, dessen Bezugselement undefiniert ist. Es gilt:

$$\text{deref}(\text{nil}) == \perp$$

Eine Referenzvariable kann allerdings mit **nil** verglichen werden. Insbesondere gilt für jede neu definierte Referenzvariable `var` vor der ersten Zuweisung:

$$\text{var} == \text{nil}$$

Für den Vergleich zweier Referenzen `a` und `b` gilt:

$$a == b \Rightarrow \text{deref}(a) == \text{deref}(b)$$

Der Umkehrschluss gilt i.a. nicht!



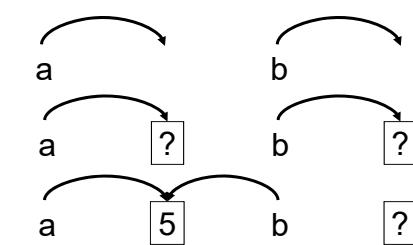
Datenstrukturen Referenzen



Referenzen werden oft auch als *Zeiger* bezeichnet und oft auch graphisch als Zeiger dargestellt:

Bsp.:

```
var a: ref int;  
var b: ref int;  
a := new(int);  
b := new(int);  
deref(a) := 5;  
b := a;
```



Datenstrukturen Referenzen



- Referenzen helfen die Duplizierung von Daten zu vermeiden.
- Information werden nur einmal gespeichert und mehrfach referenziert.

Bsp.: (Studenten ohne Referenzen)

```
type stud_liste = array [50] of Student;  
  
var gdi: stud_liste;  
var prog: stud_liste;  
  
gdi[0] := mk_stud( „Fuchs“, „Max“, {10,7,78}, 1, 1.0, false);  
gdi[1] := mk_stud( „Hettler“, „Rudi“, {1,8,75}, 3, 4.3, true);  
gdi[2] := mk_stud( „Regensburger“, „Franz“, {15,3,80}, 1, 2.3, false);  
  
...  
  
prog[0] := mk_stud( „Fuchs“, „Max“, {10,7,78}, 1, 1.0, false);  
prog[1] := gdi[2]; ...
```

Datenstrukturen Referenzen



Bsp.: (Studenten mit Referenzen)

```
type stud_liste = array [50] of ref Student;  
var gdi: stud_liste;  
var prog: stud_liste;  
gdi[0] := new(Student);  
gdi[1] := new(Student);  
gdi[2] := new(Student);  
...  
deref(gdi[0]) := mk_stud( „Fuchs“, „Max“, {10,7,78}, 1, 1.0, false);  
deref(gdi[1]) := mk_stud( „Hettler“, „Rudi“, {1,8,75}, 3, 4.3, true);  
deref(gdi[2]) := mk_stud( „Regensburger“, „Franz“, {15,3,80}, 1, 2.3, false);  
...  
prog[0] := gdi[0];  
prog[1] := gdi[2];  
...
```

© Prof. Dr. Dieter Nazareth 2023

379

Datenstrukturen Referenzen

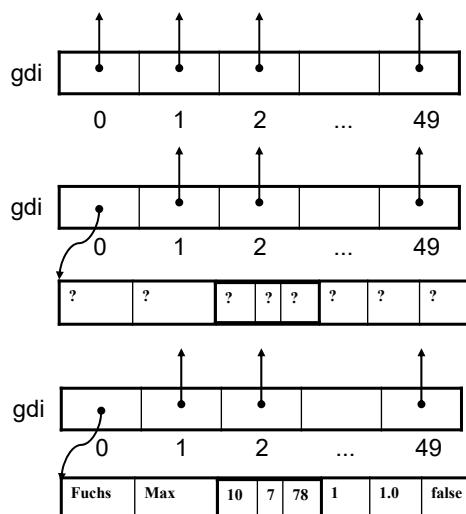


Bsp.: (Studenten mit
Referenzen)

```
var gdi: stud_liste;
```

```
gdi[0] := new(Student);
```

```
deref(gdi[0]) := mk_stud(  
„Fuchs“, „Max“, {10,7,78},  
1, 1.0, false);
```



© Prof. Dr. Dieter Nazareth 2023

380

Datenstrukturen

Einfach verkettete Listen



- In vielen Programmiersprachen ist die Definition von rekursiven Typvereinbarungen nur in Verbindung mit Referenzen möglich.
- Rekursive Typvereinbarungen auf der Basis von Referenzen heißen Geflechte.
- Eine spezielle Implementierung der Listen sind folgende einfache verkettete Geflechte:

```
type s_list_t = pair ( elem: t,
                        next: ref s_list_t);
```

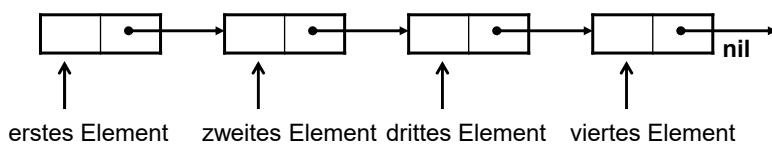
- Die Verwendung der Variante empty kann weggelassen werden. Sie wird durch den Zeiger **nil** ersetzt.

Datenstrukturen

Einfach verkettete Listen



Grafisch können einfache verkettete Listen folgendermaßen dargestellt werden:



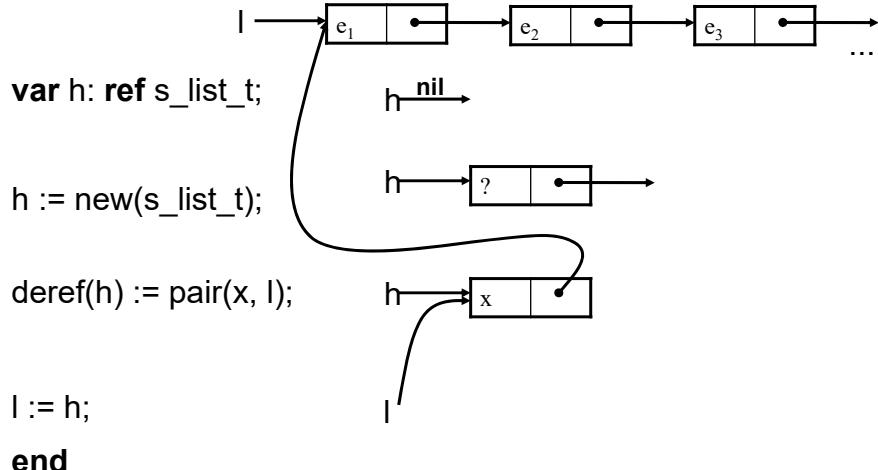
Wir können jetzt ein Element an eine Liste anhängen, indem wir die Liste selektiv ändern:

```
proc append (x: t, var l: ref s_list_t) begin
    var h: ref s_list_t;
    h := new(s_list_t);
    deref(h) := pair(x, l);
    l := h;
```

Datenstrukturen Einfach verkettete Listen



```
proc append (x: t, var l: ref s_list_t) begin
```



© Prof. Dr. Dieter Nazareth 2023

383

Datenstrukturen Einfach verkettete Listen



Während das vorne Anhängen nur konstanten Aufwand erfordert (wie Konstruktor `append` der rek. Definition), muss zum hinten Anhängen die gesamte Liste durchlaufen werden:

```
proc stock (x: t, var l: ref s_list_t) begin
    var h: ref s_list_t;
    var neu: ref s_list_t;
    h := l;
    neu := new(s_list_t);
    deref(neu) := pair(x, nil);
    if h == nil then l := neu;
    else while next(deref(h)) ≠ nil do
        h := next(deref(h));
    od
    next(deref(h)) := neu;
fi
end
```

© Prof. Dr. Dieter Nazareth 2023

384

Datenstrukturen

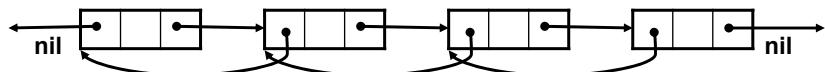
Zweifach verkettete Listen



Manchmal ist es günstiger, statt einer einfach verketteten Liste eine zweifach verkettete Liste zu verwenden:

```
type d_list_t = tripel ( prev: ref d_list_t, elem: t, next: ref d_list_t);
```

Grafisch können zweifach verkettete Listen folgendermaßen dargestellt werden:



Damit lassen sich effiziente Algorithmen zum Bearbeiten von Listen beginnend von vorne und von hinten realisieren. Sowohl der Aufwand zum Aufbau als auch der Speicherbedarf sind jedoch größer.

Datenstrukturen

Binäre Bäume mit Referenzen



Wir haben *Binäräbäume* folgendermaßen rekursiv definiert:

```
type bintree_t = emptytree |  
                  mktree ( left: bintree_t,  
                           node: t,  
                           right: bintree_t);
```

Auch diese Definition lässt sich mittels Referenzen folgendermaßen darstellen:

```
type bintree_t = mktree ( left: ref bintree_t,  
                           node: t,  
                           right: ref bintree_t);
```

Auch hier entfällt die Variante *emptytree*, die über **nil** dargestellt werden kann.

Abschließende Bemerkungen:

- Neue Datentypen können mit folgendem Befehl eingeführt werden:
type name = typausdruck
- Typbezeichner sollten immer global, d.h. im äußersten Block deklariert werden.
- Anstelle eines Typbezeichners kann immer ein Typausdruck verwendet werden. d.h. für zusammengesetzte Datentypen müssen nicht zwingend neue Bezeichner eingeführt werden. Bsp.:

type datum = array [3] of nat; \longleftrightarrow **var d: array [3] of nat;**
var d: datum; äquivalent

- Datenstruktur kapselt Menge von Datentypen sowie zugehörige charakteristische Funktionen.
- Statischer Datentyp für Daten fester, konstanter Länge.
- Dynamischer Datentyp für Daten variabler Länge.
- Aufzählungstyp definiert Typ mit endlich vielen Werten.
- Produkttyp definiert Typ als Tupel aus unabhängigen Einzelementen.
- Feld als Spezialfall des Produkttyps über festen Typ.
- Feldgröße meist statisch (feste Länge).
- Summentyp definiert die Vereinigung von Typen.

Datenstrukturen Zusammenfassung



- Einführung rekursiver Datentypen.
- Definition von Listen durch rekursiven Datentyp.
- Sortieren von Listen.
- Definition von Bäumen durch rekursiven Datentyp.
- Rekursive Durchlauftechniken für Bäume.
- Sortierter Baum als effiziente Suchstruktur.
- Einführung von Referenzen.
- Definition von Listen als Geflechte.
- Selektives Verändern von Listen.
- Definition von Bäumen als Geflechte.

Grundlagen der Informatik I Überblick



- Was ist Informatik
- Information und Repräsentation
- Algorithmen
- Programmiersprachen
- Datenstrukturen
- Komplexität
- Berechenbarkeit

Komplexität Einführung



Fragestellung:

- Mit welchem Minimum an Rechnerressourcen kann eine Aufgabe gelöst werden.
 - Wir sprechen dann von der *Komplexität* eines Problems.

Wir müssen unterscheiden zwischen

- Der Komplexität eines Problems und
 - der Komplexität eines Algorithmus.
 - Komplexität eines Problems ist die größte untere Schranke der Komplexitäten aller Algorithmen, die dieses Problem lösen.

© Prof. Dr. Dieter Nazareth 2023

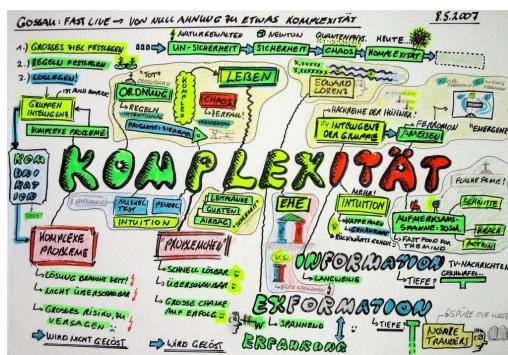
391

Komplexität Einführung



Die Ressourcen, die von hauptsächlichem Interesse sind, sind

- die Ausführungszeit und
 - der Speicherbedarf eines Algorithmus.
 - Daraus ergeben sich die beiden Komplexitätsarten
 - Zeitkomplexität und
 - Speicher-komplexität.



© Prof. Dr. Dieter Nazareth 2023

392

Die Laufzeit eines implementierten Algorithmus hängt ab von

- Eingabedaten (Menge, Art)
- Qualität des Programms (abhängig von Programmierern, Compiler, ...)
- Art und Geschwindigkeit des Rechners
- Zeitkomplexität des Algorithmus

Bsp.: Suchen in Listen

Laufzeit hängt ab von:

- Listengröße (Anzahl n der Elemente)
- Ob sich das Element in der Liste befindet und wenn ja wo
- Ob die Liste sortiert ist

Problem: Wie sollen wir die Zeitkomplexität objektiv bestimmen (z.B. Angabe in CPU-Sekunden, Anzahl der ausgeführten Maschinenbefehle, ...)?

Lösung: Angabe des Zeitaufwands unabhängig von einer konkreten Zeiteinheit durch Angabe der elementaren Operationen.

Def.: (Zeitkomplexität)

Sei A ein Algorithmus und E die Menge aller möglichen Eingabedaten dann bezeichnet $t(A, n, e)$ die Anzahl der durchzuführenden Elementaroperationen für den Algorithmus A bei Eingabe $e \in E$ mit Problemausprägung der Größe n.

Def. (cont.): (Zeitkomplexität)

$T_{\min}(A,n) := \min \{t(A,n,e) | e \in E\}$ „beste Komplexität
(best case)“

$T_{\max}(A,n) := \max \{t(A,n,e) | e \in E\}$ „schlechteste Komplexität
(worst case)“

$T_{\text{avg}}(A,n) := \text{avg} \{t(A,n,e) | e \in E\}$ „mittlere Komplexität
(average case)“

Bemerkungen:

- T_{avg} ist im allgemeinen schwierig zu bestimmen. Hier muss auch die Häufigkeitswahrscheinlichkeit der Eingaben berücksichtigt werden.
- Falls alle 3 Fälle gleich $\Rightarrow T(A,n)$

Bemerkungen:

- Falls es klar ist, von welchem Algorithmus gesprochen wird, wird auch der Parameter A unterdrückt $\Rightarrow T(n)$
- Analog: Speicherkomplexität $S(A,n)$

Bsp.: (Summe)

Gegeben: x_1, \dots, x_n ($n \in \mathbb{N}$) mit $x_i \in \mathbb{N}$

Gesucht: $s = \sum_{i=1}^n x_i$

Elementaroperationen: $+$, $:=$, $<=$

Algorithmus A:

Bsp.: (Summe cont.)

$s := 0$
$i := 1$
solange $i \leq n$
$s := s + x_i$
$i := i + 1$

Anzahl Elementaroperationen
1
1
1 $n+1$ mal
2 } n mal
2

$$T(A, n) : 1 + 1 + n + 1 + n*(2 + 2) = 3 + 5*n$$

Algorithmus ist unabhängig von der Eingabe (x_1, \dots, x_n)
 $\Rightarrow T_{\min}(A, n) = T_{\max}(A, n) = T_{\text{avg}}(A, n) = T(A, n)$

Problem: Für große Algorithmen ist es oft schwierig die Komplexität präzise bis auf jede Elementaroperation zu bestimmen. Ferner möchte man verschiedene Algorithmen bzgl. ihrer Komplexität vergleichen.

Lösung:

- Angabe einer oberen Schranke für die Komplexität von Algorithmen.
- Asymptotische Laufzeit.
- Einteilung der Algorithmen in sog. Komplexitätsklassen.
- Der Zeitaufwand wird dabei bis auf konstante Faktoren in Abhängigkeit von der Größe n angegeben.

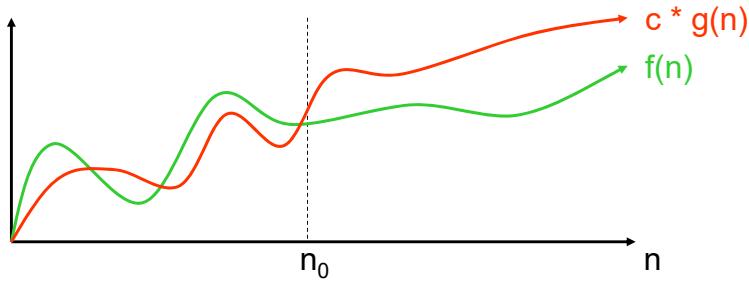
Def.: (Landau-Symbole)

Es sei P die Menge aller Funktionen $f: \mathbb{N} \rightarrow \mathbb{N}$, $g \in P$.

1. $O(g(n)) := \{f(n) \in P \mid \text{es existiert } n_0 \in \mathbb{N}, \text{ und } c > 0, \text{ so dass}$
 $\text{für alle } n > n_0 \text{ gilt: } |f(n)| \leq c^*|g(n)|\}$
Für $f(n) \in O(g(n))$ sagt man, „ f wächst höchstens so schnell wie g “ bzw. „ f hat höchstens die Ordnung g “, bzw. „ f ist aus Groß-O von g “.
2. $o(g(n)) := \{f(n) \in P \mid \text{für alle } c > 0 \text{ existiert } n_0 \in \mathbb{N}, \text{ so dass für}$
 $\text{alle } n > n_0 \text{ gilt: } |f(n)| \leq c^*|g(n)|\}$
Für $f(n) \in o(g(n))$ sagt man, „ f wächst weniger schnell als g “ bzw. „ f wächst langsamer als g “ bzw. „ f hat eine kleinere Ordnung als g “, bzw. „ f ist aus klein-o von g “.

Bemerkungen:

- Ursprünglich war der Buchstabe ein Omikron: \mathcal{O}
- $O(\dots)$ und $o(\dots)$ sind Mengen. Trotzdem schreibt man oft $f(n) = O(g(n))$ bzw. $f(n) = o(g(n))$ statt $f(n) \in O(g(n))$ bzw. $f(n) \in o(g(n))$.
- $f(n) \in o(g(n)) \Rightarrow f(n) \in O(g(n))$ (jedoch nicht umgekehrt)
- Bei der Komplexität von Algorithmen ist man meist an einer Abschätzung nach oben interessiert, d.h. an Mengen $O(g(n))$.
- $O(g(n))$ bzw. $o(g(n))$ beschreiben Mengen von Funktionen, die für große n höchstens so stark bzw. weniger stark anwachsen wie die Funktion g .



Mathematiker
Edmund Landau

Beispiele:

- $f(n)=5n+3 \Rightarrow f(n) \in O(g(n))$ mit $g(n):=n$, kurz: $f \in O(n)$
 $c=6, n_0=2: f(n)=5n+3 \leq 6n$ für $n>n_0$
- $f(n)=n^2+1000n \Rightarrow f \in O(n^2)$
 $c=2, n_0=1000: f(n)=n^2+1000n \leq 2n^2$ für $n>n_0$
- $f(n)=\log_a n$ mit a beliebige Basis
Setze: $c:=\log_a 2$, dann gilt $f(n)=\log_a n=\log_a 2 \cdot \log_2 n \in O(\log_2 n)$
Fazit: Alle Logarithmusfunktionen sind von der Ordnung $O(\log_2 n)!$

Die wichtigsten Komplexitätsklassen

- Konstante Komplexität:
 $O(1)$: Die Laufzeit hängt nicht von n ab.
Beispiel: Indexzugriff auf ein Feld der Länge n .
- Logarithmische Komplexität:
 $O(\log_2 n)$; bei Verdopplung von n ändert sich die Komplexität nur um einen konstanten Faktor:
 $f \in O(\log_2 n)$: $|f(n)| \leq c \cdot |\log_2 n|$
 $|f(2n)| \leq c \cdot |\log_2(2n)| = c \cdot |\log_2 2 + \log_2 n| = c \cdot |\log_2 n + 1|$
Beispiel: Binäre Suche im sortierten Feld mit n Einträgen.
- Lineare Komplexität: $O(n)$: Verdopplung von $n \Rightarrow$ Verdopplung der Laufzeit.
Beispiel: Suche im unsortierten Feld der Länge n .

Die wichtigsten Komplexitätsklassen (cont.)

- Leicht überlineare Komplexität: $O(n \cdot \log_2 n)$
Beispiel: Fortgeschrittene Algorithmen zum Sortieren von n Zahlen (z.B. Mergesort).
- Quadratische Komplexität: $O(n^2) \rightarrow (2n)^2 = 4n^2$
Verdoppeln sich die Daten dann vervierfacht sich die Laufzeit.
Beispiel: Einfachere Sortieralgorithmen (z.B. Insertionsort)
- Polynomiale Komplexität: $O(n^{c1} + n^{c2} + \dots + n^{cm})$
- Exponentielle Komplexität: $O(c^n)$
z.B. $O(2^n) \rightarrow$ Verdopplung von n : $2^{2n} = (2^n)^2$
→ quadrierte Laufzeit
Beispiel: Türme von Hanoi

Summenregel:

Seien $T_1(n)$ und $T_2(n)$ die Laufzeiten der Programmteile P_1, P_2 mit $T_1(n)=O(f(n))$, $T_2(n)=O(g(n))$. Dann ist die Laufzeit der Hintereinanderschaltung gegeben durch

$$T(n) = T_1(n) + T_2(n)$$

und ergibt

$$T(n) = O(\max(f(n), g(n)))$$

Fazit: Wenn wir zwei Programme hintereinander ausführen, dann interessiert uns nur die höhere der beiden Komplexitäten.

Produktregel:

Seien $T_1(n)$ und $T_2(n)$ die Laufzeiten der Programmteile P_1, P_2 mit $T_1(n)=O(f(n))$, $T_2(n)=O(g(n))$. Dann ist die Laufzeit der ineinander geschachtelten Programmteile gegeben durch

$$T(n) = T_1(n) \cdot T_2(n)$$

und ergibt

$$T(n) = O(f(n) \cdot g(n))$$

Fazit: Wenn Programmteile innerhalb eines anderen Programms immer wieder ablaufen, dann müssen die Komplexitäten multipliziert werden.

Beispiel (Polynomauswertung):

$$f(x) = a_0 + a_1 * x^1 + a_2 * x^2 + \dots + a_n * x^n = \sum_{i=0}^n a_i * x^i$$

- direkt: $a_i * x^i = a_i * x * x * \dots * x \Rightarrow i \text{ Multiplikationen}$
 $i \leq n \Rightarrow O(n)$
 $\sum_{i=0}^n a_i * x^i \Rightarrow (n+1) * (a_i * x^i)$
 $n \text{ Additionen} \Rightarrow O(n)$
 $\text{Produktregel} \Rightarrow O(n^2)$

- Horner-Schema: Bei der direkten Berechnung wird die Potenzbildung zu oft ausgeführt!

$$f(x) = a_0 + x * (a_1 + x * (a_2 + x * (\dots + x * a_n) \dots)) \Rightarrow n \text{ Multiplikationen und Additionen} \Rightarrow O(n)$$

$$n=4: f(x) = a_0 + x * (a_1 + x * (a_2 + x * (a_3 + x * a_4)))$$

Beispiel (Türme von Hanoi):

Gesucht: Zeitkomplexität $T(n)$

Funktion Hanoi(n, quelle, senke, zs)		
		$n = 1$
		wahr
		falsch
1	bewege scheibe von quelle zur senke	Hanoi(n-1,quelle,zs,senke)
		bewege unterste Scheibe von quelle zu senke
		Hanoi(n-1,zs,senke,quelle)
		$T(n-1)$
		1
		$T(n-1)$

$$T(1) = 1$$

$$\begin{aligned} T(n) &= T(n-1) + 1 + T(n-1) = 2*T(n-1) + 1 = 2*(2*T(n-2) + 1) + 1 \\ &= 2^2*T(n-2) + 2 + 1 = 2^2*T(n-2) + (2^2 - 1) = \dots \\ &= 2^{n-1}*1 + 2^{n-1}-1 = 2*2^{n-1}-1 = 2^n - 1 \Rightarrow T(n) = O(2^n) \end{aligned}$$

Beispiel (Türme von Hanoi cont.):

Selbst für kleine n wird die Aufgabe damit praktisch unlösbar.
Unter der Annahme, dass die Mönche eine Scheibe pro Sekunde verschieben können und dass sie bis zur Vollendung der Aufgabe kontinuierlich und ohne Pause durcharbeiten, lässt sich die Dauer zur Lösung des Problems folgendermaßen abschätzen:

Anzahl Scheiben	Benötigte Zeit
5	31 Sekunden
10	17,1 Minuten
20	12 Tage
30	34 Jahre
40	348 Jahrhunderte
60	36,5 Milliarden Jahre
64	584 Milliarden Jahre

Beispiel (Suchen im Suchbaum):

Sei n die Anzahl der Baumelemente;
gesucht: Zeitkomplexität $T(n)$

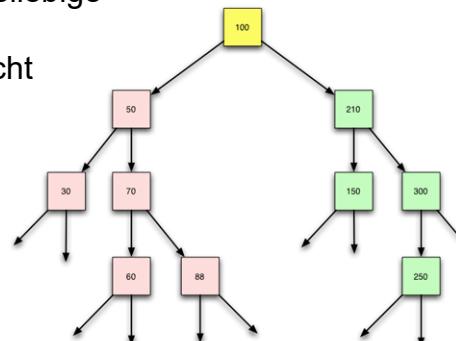
```
fun is_in (x:t, t: bintree_t) bool begin
    if t in emptytree
        then false
    else if node(t) == x then true
    else if x < node(t)
        then is_in(x, left(t))
    else is_in(x, right(t)) fi fi fi
end
```

Beispiel (Suchen im Suchbaum cont.):

Die Komplexität hängt stark von der Struktur des Baumes, konkret von der Höhe des Baumes ab, deshalb müssen wir wiederum den besten und den schlechtesten Fall betrachten.

Hierbei wollen wir jedoch beliebige Suchelemente x zulassen.

Diese Vorgehensweise macht Sinn, weil wir i.a. das Suchelement nicht beeinflussen können, aber den Suchbaum.



Beispiel (Suchen im Suchbaum cont.):

Beste Komplexität $T_{\min}(n)$:

Die beste Komplexität wird erreicht, wenn bei fester Baumhöhe möglichst viele Elemente gespeichert sind, d.h. wenn der Baum vollständig ausgeglichen ist. Für das Eingabeelement nehmen wir jedoch den schlechtesten Fall an, nämlich, dass das Element nicht enthalten ist!

$$T_{\min}(0, \text{emptytree}) = 1$$

$$\begin{aligned} T_{\min}(n, t) &= 6 + \max(T_{\min}(m_1, \text{left}(t)), T_{\min}(m_2, \text{right}(t))) \\ &= 6 + T_{\min}(n/2) = 6 + 6 + T_{\min}(n/4) = 6 * \log_2(n+1) + 1 \end{aligned}$$

$$\Rightarrow T_{\min}(n) = O(\log_2(n))$$

\Rightarrow Suchen im Suchbaum hat bei ausgeglichenem Baum logarithmische Komplexität!

Beispiel (Suchen im Suchbaum cont.):

Schlechteste Komplexität $T_{\max}(n)$:

Die schlechteste Komplexität wird erreicht, wenn bei fester Baumhöhe möglichst wenig Elemente gespeichert sind, d.h. wenn der Baum zur Liste degeneriert ist. Für das Eingabeelement nehmen wir wiederum den schlechtesten Fall an, nämlich, dass das Element nicht enthalten ist!

$$T_{\max}(0, \text{emptytree}) = 1$$

$$\begin{aligned} T_{\max}(n, t) &= 6 + \max(T_{\max}(m_1, \text{left}(t)), T_{\max}(m_2, \text{right}(t))) \\ &= 6 + T_{\max}(n-1) = 6 + 6 + T_{\max}(n-2) = 6 * n + 1 \end{aligned}$$

$$\Rightarrow T_{\max}(n) = O(n)$$

\Rightarrow Suchen im Suchbaum hat bei zur Liste degenerierten Baum lineare Komplexität!

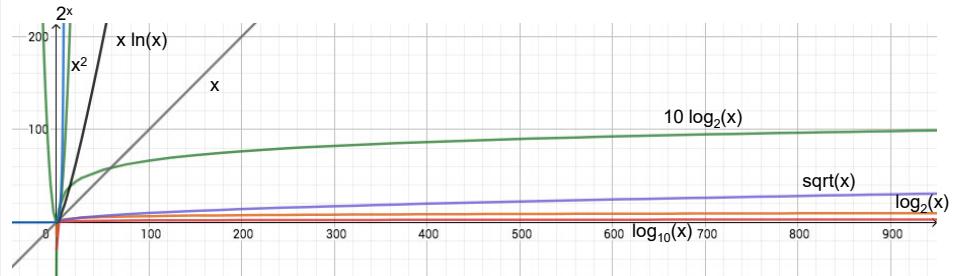
Vergleich der Ausführungszeit unterschiedlicher Komplexitätsklassen:

Größe n der Eingabedaten	$\log_2 n$	n	n^2	2^n
10	0.000003	0.00001	0.0001	0.001
	Sekunden	Sekunden	Sekunden	Sekunden
100	0.000007	0.0001	0.01	10^{14}
	Sekunden	Sekunden	Sekunden	Jahrhunderte
1000	0.00001	0.001	1	astronomisch
	Sekunden	Sekunden	Sekunden	
10000	0.000013	0.01	1.7	astronomisch
	Sekunden	Sekunden	Minuten	
100000	0.000017	0.1	2.8	astronomisch
	Sekunden	Sekunden	Stunden	

Komplexität Komplexitätsklassen



Vergleich der Ausführungszeit unterschiedlicher Komplexitätsklassen:



Grundlagen der Informatik Überblick



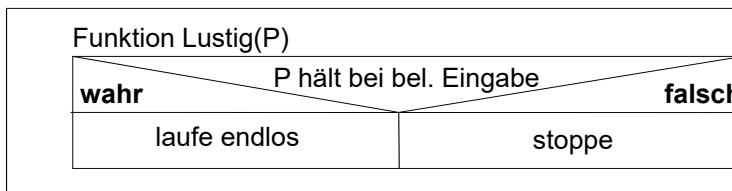
- Was ist Informatik
- Information und Repräsentation
- Algorithmen
- Programmiersprachen
- Datenstrukturen
- Komplexität
- Berechenbarkeit

Frage: Gibt es Aufgaben, die ein Rechner nicht durchführen kann, bzw. gibt es Aufgaben, für die kein Algorithmus besteht, der die Aufgabe löst?

Antwort: Ja, sehr viele!

Bsp.: (Halteproblem)

Aufgabe: Teste, ob ein beliebiges gegebenes Programm P bei beliebigen Eingabedaten D terminiert oder nicht.



Was passiert, wenn wir Lustig(Lustig) ausgeführt wird?

Die Ausführung kann weder stoppen, noch endlos laufen. Der Widerspruch kann nur aufgelöst werden, wenn wir gar nicht feststellen können, ob ein Programm P stoppt oder nicht und damit dieser Algorithmus nicht existieren kann.

Def.: Aufgabenstellungen für die es keinen Algorithmus gibt, der die Aufgabenstellung löst, heißen *nicht berechenbar* oder *nicht entscheidbar*.

Frage: Vielleicht ist unser Algorithmusbegriff zu schwach um solche Aufgabenstellungen lösen zu können?

Church-Turing-These:

Alle vernünftigen Definitionen von Algorithmus, soweit sie bekannt sind, sind gleichwertig bzw. gleichmächtig.

Bemerkungen:

- Gleichwertig bzw. gleichmächtig heißt, dass man zu jedem Algorithmus in einer bestimmten Notation einen Algorithmus in allen anderen Notationen angeben kann, der dieselbe Aufgabestellung löst.

Alan
Turing



Alonzo
Church



© Prof. Dr. Dieter Nazareth 2023

419

Bemerkungen (cont.):

- Alle verwendeten Programmiersprachen sind gleichmächtig.
- Alle verwendeten Rechner sind gleichmächtig.
- Jede Programmiersprache bzw. jeder Rechner kann auf allen Rechnern und mit jeder Programmiersprache simuliert werden.
- Um zu zeigen, dass ein Problem berechenbar bzw. nicht berechenbar ist genügt es einen Algorithmus in einer beliebigen Sprache anzugeben, bzw. zu zeigen, dass es in dieser Sprache keinen Algorithmus für das Problem gibt.
- In der Komplexitäts- und Berechenbarkeitstheorie werden meist sog. Turingmaschinen mit Turingprogrammen als universeller Ansatz verwendet.

© Prof. Dr. Dieter Nazareth 2023

420

Beispiel:

Es ist nicht entscheidbar, ob zwei gegebene Programme dieselbe Aufgabe lösen. d.h. für dieselbe Eingabe immer dieselbe Ausgabe liefern (Äquivalenzproblem).

Offensichtlich gibt es bei den nicht entscheidbaren Problemen welche die weniger entscheidbar sind als andere. Wenn das Programm nicht endlos läuft, dann kann dies durch den Haltetest festgestellt werden. Problematisch ist dagegen der Endlosfall.

Wir beschränken uns im folgenden auf Probleme, die mit Ja oder Nein beantwortet werden können.

Definition: Ein Problem heißt *partiell berechenbar (semi-entscheidbar)*, wenn es einen Algorithmus gibt, der das Problem im Ja-Fall oder im Nein-Fall löst.

Beispiele:

- Das Halteproblem ist semi-entscheidbar. Es ist möglich einen Algorithmus zu finden, der das Ergebnis Ja liefert, wenn das gegebene Programm terminiert. Der Algorithmus wird jedoch bei nicht-terminierenden Programmen scheitern.
- Das Äquivalenzproblem ist unentscheidbar. Es kann weder ein Algorithmus angegeben werden, der bei äquivalenten Programmen terminiert, noch bei nicht äquivalenten Programmen.

Viele Probleme sind zwar theoretisch berechenbar, d.h. wir können einen Algorithmus für dieses Problem angeben, aber in der Praxis zumindest für hohe Eingabewerte nicht lösbar, weil ihre Komplexität zu hoch ist. Dies gilt insbesondere für Probleme mit exponentieller Komplexität (vgl. Zeittabelle).

Polynomiale Algorithmen tendieren hingegen dazu für eine „vernünftige“ Anzahl von Eingabedaten durchführbar zu sein. Deswegen ist es für die Informatik wichtig Algorithmen mit polynomialer Komplexität für Probleme zu finden. Leider gibt es für viele Probleme exponentielle Algorithmen, aber bis heute hat man keine polynomialen Algorithmen gefunden. Man konnte aber auch nicht beweisen, dass es keine polynomiale Algorithmen gibt. Solche Probleme werden als *hartnäckige Probleme* bezeichnet.

berechenbare Aufgabenstellungen

durchführbare Aufgabenstellungen

alle Aufgabenstellungen

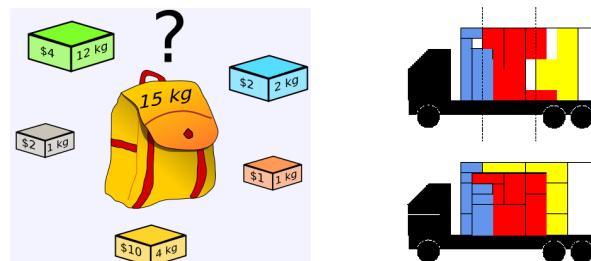
Beispiel (Packproblem):

Eine Spedition muss n Kästen mit unterschiedlichen Gewichten von Ort A nach Ort B transportieren. Es stehen m Lastwagen zur Verfügung von denen jeder nur ein bestimmtes Gewicht G transportieren kann.

Beispiel (Packproblem cont.):

Für folgenden Fragestellungen gibt es (zur Zeit) nur exponentielle Algorithmen:

- Passen die Kästen alle auf die Lastwagen?
- Minimiere die Anzahl der Fahrten.
- Belade einen Lastwagen maximal.

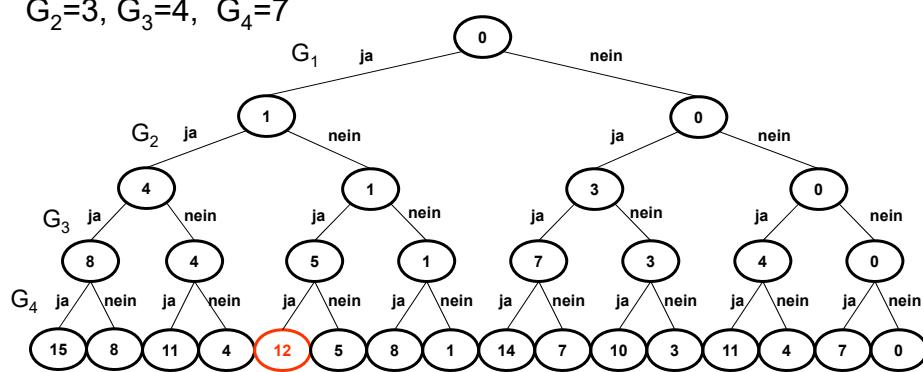


© Prof. Dr. Dieter Nazareth 2023

425

Beispiel (Packproblem cont.):

Wir betrachten jetzt konkret die letzte Aufgabestellung mit folgenden Daten: maximales Beladungsgewicht des Lastwagens: 12t. 4 Kisten mit folgenden Gewichten: $G_1=1$, $G_2=3$, $G_3=4$, $G_4=7$



© Prof. Dr. Dieter Nazareth 2023

426

Bsp.: (Packproblem cont.)

Für 4 Kästen gibt es bereits 16 Beladungsmöglichkeiten. Die Anzahl steigt also exponentiell mit der Anzahl der Kästen. Die Aufgabenstellung ist also in der Praxis nicht durchführbar, obwohl theoretisch natürlich berechenbar.

Wenn allerdings eine Lösung vorgegeben wird, ist es einfach diese zu überprüfen. Um z.B. zu überprüfen, ob eine vorgegebene Anzahl von n Kästen auf dem LKW Platz finden sind nur $n+1$ Additionen und ein Vergleich durchzuführen. Die Komplexität des Überprüfens ist also $O(n)$.

Offensichtlich gilt: Ein Aufgabe zu lösen ist schwieriger als zu überprüfen, ob eine Lösung wirklich korrekt ist.

Bis jetzt haben wir nur deterministische Algorithmen betrachtet. In unserer Sprache war es nicht möglich durch ein Sprachkonstrukt Berechnungszweige nichtdeterministisch auszuwählen. Um Nichtdeterminismus ausdrücken zu können führen wir in unserer Sprache folgendes Konstrukt ein:

some id: t | p(id)

Dabei ist id ein beliebiger Bezeichner, t ein beliebiger Typ und p(id) ein Prädikat (Boolescher Ausdruck) der von id abhängt. Das Konstrukt hat folgende Semantik:

- Der Zustandsraum wird um den Bezeichner id erweitert.
- Nach der Anweisung hat id irgendeinen Wert so dass gilt: p(id)
- Gibt es keinen Wert mit p(id) dann hat id den Wert \perp

some heißt *deskriptiver Auswahloperator*. Er kann dazu verwendet werden, um geschickt die gewünschten Werte zu erraten.

Beispiel (Erfüllbarkeitsproblem):

Gegeben sei ein Boolescher Ausdruck in dem Variable x_1, \dots, x_n vorkommen. Es soll geprüft werden, ob es eine Belegung der Variable mit Wahrheitswerten gibt, so dass der Ausdruck wahr ist. Der Ausdruck heißt dann *erfüllbar*.

Konkret: $b(x_1, x_2, x_3) = x_1 \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee \neg x_3) \wedge x_3$

Frage: Gibt es eine Belegung der Variablen x_1, x_2, x_3 so dass $b(x_1, x_2, x_3) = \text{true}$?

Beispiel (Erfüllbarkeitsproblem):

x_1	x_2	x_3	$\neg x_1$	$\neg x_2$	$\neg x_3$	$\neg x_1 \vee x_2$	$\neg x_2 \vee \neg x_3$	$b(x_1, x_2, x_3)$
true	true	true	false	false	false	true	false	false
true	true	false	false	false	true	true	true	false
true	false	true	false	true	false	false	true	false
true	false	false	false	true	true	false	true	false
false	true	true	true	false	false	true	false	false
false	true	false	true	false	true	true	true	false
false	false	true	true	true	false	true	true	false
false	false	false	true	true	true	true	true	false

Beispiel (Erfüllbarkeitsproblem cont.):

Alle möglichen Belegungen der Variable mit den möglichen Werten true und false müssen getestet werden! Bei n Variablen gibt es 2^n mögliche Belegungen $\Rightarrow O(2^n)$ exponentielle Komplexität.

Deterministischer Algorithmus:

```
fun b (x:bool, y: bool, z:bool): bool
begin
     $\neg x \wedge (\neg x \vee y) \wedge (\neg y \vee \neg z) \wedge z$ 
end
9 Operationen
```

Beispiel (Erfüllbarkeitsproblem cont.):

```
var i, j, k: nat;  var erg=false: bool;
var wfeld: arrray[2] of bool;
wfeld[0] := true;      wfeld[1] := false;
for i := 0 to 1 do          2 Wiederholungen.
    for j := 0 to 1 do      2 Wiederholungen
        for k := 0 to 1 do  2 Wiederholungen
            if b(wfeld[i], wfeld[j], wfeld[k]) then erg := true; fi
        od
    od
output(erg);
```

} 2³ Wiederholungen

Beispiel (Erfüllbarkeitsproblem cont.):

Um festzustellen, ob Aussage b erfüllbar ist müssen $2^3 * 9$ Operationen ausgeführt werden. Die Form des Booleschen Ausdrucks beeinflusst nur die Anzahl der Booleschen Operationen. Ausschlaggebend für die Komplexität $O(2^n)$ ist jedoch die Anzahl der Variablen.

Mit Hilfe des Auswahloperators können wir folgende nichtdeterministische Variante des Erfüllbarkeitsalgorithmus angeben:

```
some x, y, z: bool | b(x, y, z)
  if b(x, y, z) then erg := true;
    else erg := false;
  fi
```

Beispiel (Erfüllbarkeitsproblem cont.):

Wir nehmen an, dass die Anzahl der Operationen in b (inkl. Parameterübergabe) direkt proportional zu der Anzahl der Variablen ist. Damit hat dieser nichtdeterministische Algorithmus Komplexität $O(n)$.

Wir wollen im folgenden Probleme in zwei Klassen einteilen:

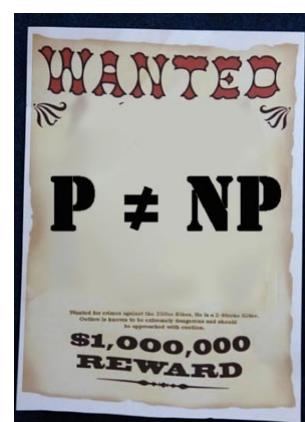
- Die eine Klasse enthält Probleme, die man in polynomialer Zeit mit deterministischen Algorithmen lösen kann.
- Die andere Klasse enthält Probleme die man in polynomialer Zeit mit nichtdeterministischen Algorithmen lösen kann.

Definition: Ein Problem P heißt *polynomial zeitbeschränkt*, wenn es von einem deterministischen Algorithmus in polynomialer Laufzeit berechnet werden kann. Die Menge aller polynomial zeitbeschränkten Probleme wird mit P bezeichnet.

Definition: P heißt *nichtdeterministisch polynomial zeitbeschränkt*, wenn es von einem nichtdeterministischen Algorithmus in polynomialer Laufzeit berechnet werden kann. Die Menge aller nichtdeterministisch polynomial zeitbeschränkten Probleme wird mit NP bezeichnet.

Bemerkungen:

- offensichtlich gilt: $P \subset NP$
- offenes Problem: $P = NP ?$
- Die Erfüllbarkeit von Booleschen Ausdrücken ist in NP.
- Offen: Ist Erfüllbarkeit auch in P?
Bis jetzt wurde noch kein Algorithmus gefunden!



Definition: Ein NP-Problem heißt *NP-vollständig (NP-hart)*, wenn es mindestens so schwierig ist, wie jedes andere Problem in NP.

Mittlerweile ist für zahlreiche Probleme nachgewiesen, dass sie NP-vollständig sind. Gelänge es, für nur eines dieser Probleme einen polynomialen deterministischen Algorithmus anzugeben, dann wäre $P = NP$ bewiesen. Umgekehrt müsste man nur für ein NP-vollständiges Problem beweisen, dass es keinen P-Algorithmus gibt, dann wäre $P \neq NP$ bewiesen.

Für alle NP-Probleme gibt es exponentielle Algorithmen. Darüber hinaus können alle NP-Probleme deterministisch in polynomialer Zeit geprüft werden.

Prominente NP-vollständige Probleme:

- Packproblem (Rucksackproblem)
- Erfüllbarkeitsproblem
- Problem des Handlungsreisenden (Travelling Salesman Problem):
Ein Handlungsreisender macht eine Rundreise durch n Städte wobei jede Stadt genau einmal besucht wird. Die Reisekosten zwischen je zwei Städten ist fest vorgegeben. In welcher Reihenfolge müssen die Städte bereist werden, um die Reisekosten so gering wie möglich zu halten?

Prominente NP-vollständige Probleme (cont.):

Kürzester Rundreiseweg durch die 15 größten Städte Deutschlands. Insgesamt sind $14!/2 = 43\,589\,145\,600$ verschiedene Wege möglich.



Prominente NP-vollständige Probleme (cont.):

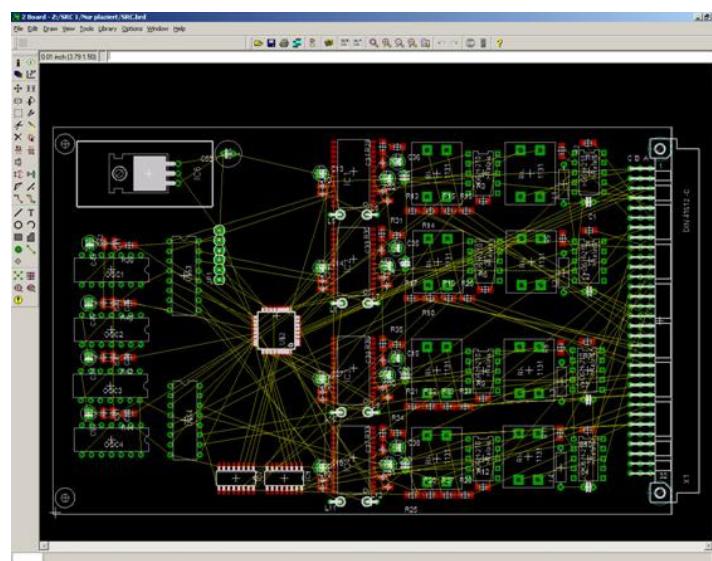
■ Verschnittproblem:

Dieses Problem hat viele Varianten, die in der Praxis von Bedeutung sind. Das Grundproblem sieht folgendermaßen aus: Aus einer Fläche sollen verschiedene Muster so platziert werden, dass möglichst viele Muster auf die Fläche passen (bzw. möglichst wenig Verschnitt entsteht). Konkrete Varianten sind bspw. das Ausstanzen von Blechteilen aus einem Blech, das Ausschneiden von Lederteilen aus einem Leder oder einfach das Ausstechen von Plätzchen aus einem Kuchenteig.

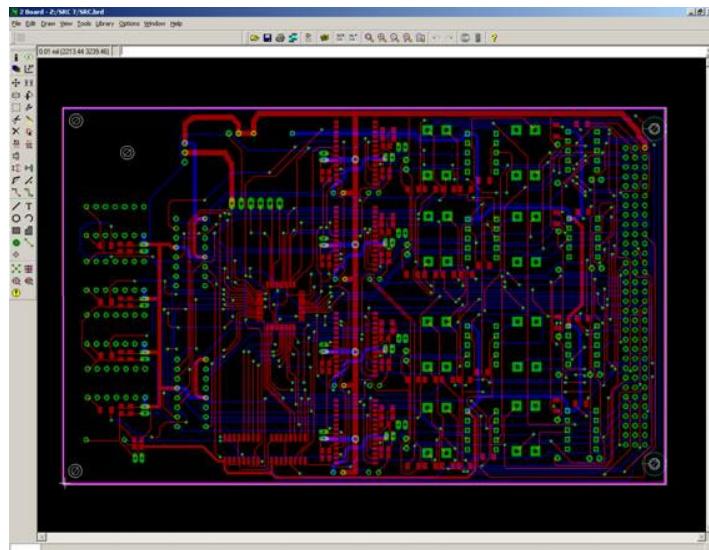
Prominente NP-vollständige Probleme (cont.):

- Verdrahtungsproblem:

Auch dieses Problem hat viele in der Praxis auftretende Varianten. Das grundlegende Problem kann folgendermaßen beschrieben werden: Gegeben sind n Anschlusspunkte zwischen denen m Verbindungen hergestellt werden sollen. Es stehen k Ebenen für die Verbindungen zur Verfügung. Gesucht sind die kürzesten kreuzungsfreien Verbindungen zwischen den Punkten. Ein wichtige Variante tritt im Leiterplattendesign auf, wo Bauteile kreuzungsfrei auf verschiedenen Ebenen verdrahtet werden müssen (Routing).



Berechenbarkeit NP-vollständige Probleme



© Prof. Dr. Dieter Nazareth 2023

443

Berechenbarkeit NP-vollständige Probleme



- NP-vollständige Probleme erfordern hohen Rechenaufwand.
- effiziente Algorithmen sind besonders wichtig.
- Diese Probleme können stets als Suchprobleme dargestellt werden, bei denen die Lösung in einem Baum mit beschränktem Verzweigungsgrad v und beschränkter Höhe h gesucht werden muss.
- Ein deterministischer Algorithmus muss im wesentlichen diesen Baum durchlaufen.
- Für die effiziente Suche gibt es deshalb eine Reihe von Strategien zum effizienten Durchlaufen des Suchbaums.

© Prof. Dr. Dieter Nazareth 2023

444

Berechenbarkeit

Branch and Bound Methode



Beim Durchlaufen des Suchbaums werden sukzessive die einzelnen Pfade (tracks) durchlaufen. Führt ein Pfad nicht zum Ziel, dann muss zu einem weiter oben liegenden Knoten zurückgesprungen werden, um von dort dann einen neuen Pfad zu durchlaufen (*backtracking*).

Das *Branch and Bound* Verfahren ist effizientes Backtracking Verfahren. Dabei wird jedem Knoten durch eine *Bewertungsfunktion* ein Schätzwert zugeordnet, der die Wahrscheinlichkeit einer erfolgreichen Suche widerspiegelt. Es wird dann stets der Zweig weiterverfolgt, der die höchste Erfolgswahrscheinlichkeit verspricht (*branch*).

Der Bound Schritt versucht Zweige abzuschneiden, die zu keiner Lösung mehr führen können (Begrenzung des Rechenaufwandes).

Berechenbarkeit

Branch and Bound Methode



Beispiel: Packproblem

Wir nehmen wieder unsere 4 Kisten mit folgenden Gewichten:

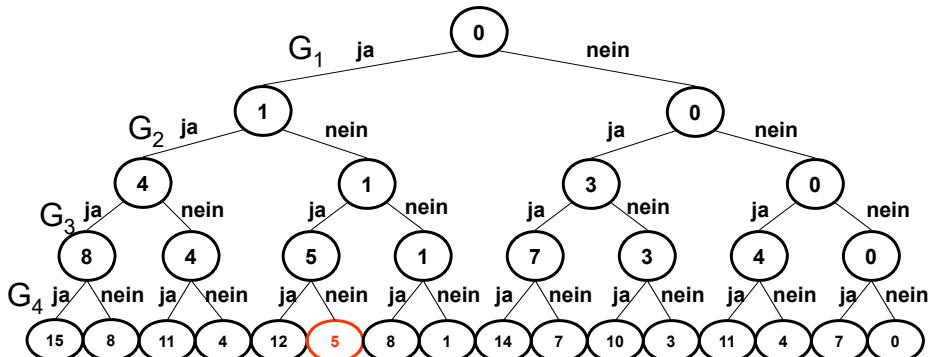
$$G_1=1, G_2=3, G_3=4, G_4=7.$$

Das maximale Beladungsgewicht des Lastwagens sei jetzt 5t. Gesucht ist eine optimale (Gewicht) Beladung des Lkw. Als Bewertungsfunktion nehmen wir das Gewicht der bisherigen Ladung. Wir verfolgen stets den Zweig mit dem höchsten bisherigen Gewicht. Zweige deren Gewicht das Maximalgewicht bereits überschritten haben, werden nicht weiter verfolgt (*bound*).

Berechenbarkeit Branch and Bound Methode



Kompletter Suchbaum: Es gibt genau eine Beladung mit exakt 5 Tonnen.



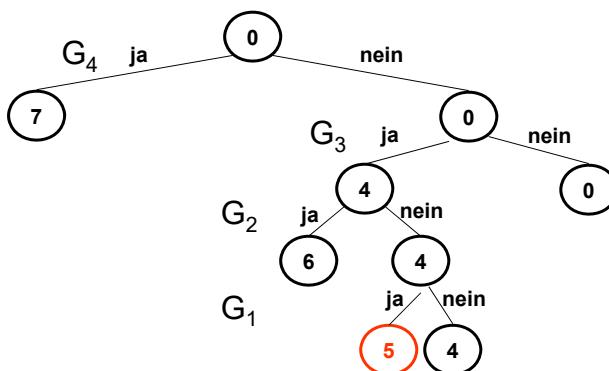
© Prof. Dr. Dieter Nazareth 2023

447

Berechenbarkeit Branch and Bound Methode



Branch and Bound Methode: Wir wählen immer die Kiste mit dem höchsten Gewicht:



© Prof. Dr. Dieter Nazareth 2023

448

Bemerkungen:

- Die Bewertungsfunktion b muss folgende wichtige Eigenschaft haben: Für jeden Knoten k muss gelten, dass für alle Knoten l des Teilbaums mit der Wurzel k gilt: $b(k) \leq b(l)$. In unserem Fall ist dies dadurch sichergestellt, weil mit jeder Kiste nur noch mehr Gewicht hinzukommt und nie das Gewicht abnehmen kann. Würde es Kisten mit negativem Gewicht geben, dann würde die Branch and Bound Strategie nicht zum Ziel führen.
- Trotz Branch and Bound Strategie hat der Algorithmus im schlechtesten Fall Komplexität $O(2^n)$.

- Komplexität kann meist deutlich reduziert werden, wenn man statt der optimalen Lösung eine annähernd optimale Lösung akzeptiert.
- Verzichten auf das aufwändige Zurücksetzen (backtracking).

Bei der *Greedy-Methode* wird versucht eine globale Optimierungsaufgabe durch ein lokales Optimierungskriterium zu lösen. Das Ziel soll möglichst schnell erreicht werden (gierig):

- Gegeben ist eine Grundmenge A von Lösungsbausteinen.
- Die Lösungsmenge ist initial die leere Menge.
- Füge schrittweise das geeignetste (optimalste aus der lokalen Sicht) Elemente aus A zur Lösungsmenge hinzu und entferne diese aus A bis Gesamtlösung erreicht.

Greedy(A)
Berechne Lösungsmenge aus Grundmenge A von Bausteinen

Lösung := {}

solange Lösung ≠ Endlösung und A ≠ {}

x := „bestes“ Element aus A

x zulässige Erweiterung

wahr

falsch

Lösung := Lösung ∪ {x}

A := A \ {x}

Beispiel (Packproblem):

Unsere Grundmenge A enthält alle Kisten und zum Start ist unsere Lösungsmenge leer:

$A := \{G_1, G_2, G_3, G_4\}$ Lösung := {}

Wir suchen die „beste“ Kiste aus, d.h. G_4 . Dieses Element ist jedoch keine zulässige Erweiterung unserer Lösung, weil das maximale Beladungsgewicht bereits überschritten ist. Wir entfernen G_4 aus A:

$A := \{G_1, G_2, G_3\}$. Lösung := {}

Wir suchen die beste Kiste aus, d.h. G_3 . Dies ist eine zulässige Erweiterung unserer Lösung also fügen wir G_3 zur Lösung hinzu und entfernen G_3 aus A:

Beispiel (Packproblem cont.):

$A := \{G_1, G_2\}$. Lösung := $\{G_3\}$

Die nächste Kiste ist G_2 . Diese ist wieder keine zulässige Erweiterung:

$A := \{G_1\}$ Lösung := $\{G_3\}$

Bleibt als letzte Kiste G_1 übrig. Diese ist eine zulässige Erweiterung:

$A := \{\}$ Lösung := $\{G_3, G_1\}$

Damit stoppt der Algorithmus mit der optimalen Lösung.

Bemerkungen:

- Die Greedy-Methode liefert bei lösbarer Problemen immer eine Lösung, die aber im allgemeinen nur suboptimal ist.
- Die Auswahl eines lokalen Optimums führt oft nicht zum Erreichen des globalen Optimums.
- Der Greedy-Algorithmus zur Lösung des Packproblems hat nur Komplexität $O(n)$. Die brute force Methode (die natürlich immer die optimale Lösung findet) hat hingegen Komplexität $O(2^n)$.



Bemerkungen:

- Die Auswahl des „besten“ Elements kann nach unterschiedlichen Strategien erfolgen. Anstatt gierig immer die Größte Kiste aufzuladen könnte man auch eine geizige Variante angeben, die versucht die Ladekapazität des Lkw möglichst langsam aufzubrauchen. Weitere Strategien sind je nach Problem möglich.
- NP-vollständige Probleme erfordern stets eine genaue Analyse hinsichtlich der tatsächlichen Eingabedaten und der tatsächlichen Anforderungen. Der gewählte Algorithmus muss gut an beide angepasst werden.