LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Skript zur Vorlesung:

Einführung in die Programmierung

WiSe 2009 / 2010

Prof. Dr. Christian Böhm
Annahita Oswald
Bianca Wackersreuther

Ludwig-Maximilians-Universität München
Institut für Informatik
Lehr- und Forschungseinheit für Datenbanksysteme

Gliederung

2. Daten und Algorithmen

2.1 Zeichenreihen

2.2 Datendarstellung durch Zeichenreihen

2.3 Syntaxdefinitionen

2.4 Algorithmen

Daten

Wir betrachten zunächst die Daten (Objekte), die durch Algorithmen verarbeitet werden sollen.

Typische Daten sind Zahlen, z.B. die Zahl “drei”, die wie folgt dargestellt werden kann:

- 3
- DREI
- III
- Drei ausgestreckte Finger einer Hand
- …

Datendarstellung

- Wir unterscheiden bei einem Objekt
  - die Darstellung, (*Syntax*, “Bezeichnung”)
  - seine Bedeutung, (*Semantik*, “Information”)
- Einige Datendarstellungen sind für maschinelle Verarbeitung nicht geeignet.
- Alle geeigneten Datendarstellungen beruhen auf dem Grundprinzip der *Zeichenreihe*.

Alphabet

- Ein *Alphabet* ist eine endliche Menge, deren Elemente *Zeichen* genannt werden.
- Beispiele:
  - Menge der Großbuchstaben: {A, B, C, . . ., Z}
  - Menge der Dezimalziffern: {1, 2, 3, . . ., 9}
  - Menge der Vorzeichen: {+, −}
  - Menge der Richtungszeiger eines Lifts: {↑, ↓}
- Alphabete, die genau zwei Zeichen enthalten heißen *binär*.
- Ein wichtiges binäres Alphabet besteht aus den *Binärziffern* (*Bits*): {0, 1}

Zeichenreihe

- Eine *Zeichenreihe* über einem Alphabet A ist eine (endliche) Folge von Zeichen aus A. (Formal auch die leere Folge)
- Wir schreiben Zeichenreihen / Folgen $(x_1, x_2, \ldots, x_n)$ auch als $x_1x_2 \ldots x_n$.
- Beispiele:
  - Sei $A_1 = \{A, B, C, \ldots, Z\}$
    - Die Folge INFORMATIK ist eine Zeichenreihe über A_1.
    - Die Folge BÖHM ist *keine* Zeichenreihe über A_1. **Warum?**
  - Sei $A_2 = \{0, 1\}$
    - Die Folge 0 ist eine Zeichenreihe über A_2.
    - Die Folge 1 ist eine Zeichenreihe über A_2.
    - Die Folge 01 ist eine Zeichenreihe über A_2.
    - Die Folge 10 ist eine Zeichenreihe über A_2.
    - Die Folge 11 ist eine Zeichenreihe über A_2.
    - Die Folge 00 ist eine Zeichenreihe über A_2.
    - …

Bezeichnung von Daten

- Wir verwenden ausschließlich Zeichenreihen zur Bezeichnung von Daten.
- Im folgenden betrachten wir als Beispiel die Darstellung von natürlichen Zahlen, also Elementen der Menge $\mathbb{N}_0$
- Die Zahl “dreizehn” lässt sich u.a. durch folgende Zeichenreihen bezeichnen:
  - 13 ($A = \{0, 1, 2, \ldots, 9\}$),
  - DREIZEHN ($A = \{A, B, \ldots, Z\}$),
  - |||||||||||||| ($A = \{|\}$).
- Nicht alle diese Darstellungen sind für den praktischen Gebrauch (z.B. Rechnen) geeignet.
- Am besten geeignet ist die *Zifferndarstellung*, z.B. die allgemein gebräuchliche *Dezimaldarstellung* über dem Alphabet $\{0, 1, 2, \ldots, 9\}$.

Zifferndarstellung / p-adische Zahlendarstellung

- Das allgemeine Prinzip der Zifferndarstellung ist wie folgt definiert:
  Sei $p \in \mathbb{N}, p \geq 2$ und $A_p = \{z_0, z_1, \ldots, z_{p-1}\}$ ein Alphabet mit p Zeichen (*Ziffern*) $z_0, z_1 \ldots, z_{p-1}$.

  Die Funktion $Z : A_p \rightarrow \mathbb{N}_0$ bildet jedes Zeichen aus A_p auf eine natürliche Zahl wie folgt ab:

  $$Z(z_i) = i \qquad \text{für } i = 0, \ldots, p-1.$$

  Eine Zeichenreihe $x = x_n x_{n-1} \ldots x_1 x_0$ über A_p (d.h. $x_i \in A_p$ für $0 \leq i \leq n$) bezeichnet die Zahl

  $$\boldsymbol{Z}(x) = p^n \cdot Z(x_n) + p^{n-1} \cdot Z(x_n - 1) + \ldots + p \cdot Z(x_1) + Z(x_0)$$

- Zur Verdeutlichung schreiben wir auch x_p.
  x_p heißt *p-adische Zahlendarstellung* der Zahl $\boldsymbol{Z}(x_p) \in \mathbb{N}_0$

Zifferndarstellung / *p*-adische Zahlendarstellung

- Nochmal die Formel:

$$\boldsymbol{Z}(x) = p^n \cdot Z(x_n) + p^{n-1} \cdot Z(x_n - 1) + \ldots + p \cdot Z(x_1) + Z(x_0)$$

- Beispiele:
  - $p = 10$ und $A_{10} = \{z_0, z_1 \ldots, z_9\}$ erhält man die Dezimaldarstellung wenn man statt z_i gleich $Z(z_i)$ schreibt (also z.B. statt z_3 schreibe $Z(z_3) = 3$):

    $\boldsymbol{Z}(983_{10}) = 10^2 \cdot 9 + 10^1 \cdot 8 + 10^0 \cdot 3 =$ "**neunhundertdreiundachtzig**".

    (Wir schreiben direkt $A_{10} = \{0, 1 \ldots, 9\}$)
  - $p = 2$ und $A_2 = \{0, 1\}$ (*Binärdarstellung*):

    $\boldsymbol{Z}(1111010111_2) =$

    $2^9 \cdot 1 + 2^8 \cdot 1 + 2^7 \cdot 1 + 2^6 \cdot 1 + 2^5 \cdot 0 + 2^4 \cdot 1 + 2^3 \cdot 0 + 2^2 \cdot 1 + 2 \cdot 1 + 1 =$

    "**neunhundertdreiundachtzig**".
  - $p = 8$ und $A_8 = \{0, 1, 2, 3, 4, 5, 6, 7\}$ (*Oktaldarstellung*):

    $\boldsymbol{Z}(1727_8) = 8^3 \cdot 1 + 8^2 \cdot 7 + 8 \cdot 2 + 7 =$ "**neunhundertdreiundachtzig**".
  - $p = 16$ und $A_{16} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$ (*Hexadezimaldarstellung*):

    $\boldsymbol{Z}(3D7_{16}) = 16^2 \cdot 3 + 16 \cdot 13 + 7 =$ "**neunhundertdreiundachtzig**".

Zifferndarstellung / p-adische Zahlendarstellung

- Führende Nullen sind in der Definition der p-adischen Zahldarstellung zugelassen, z.B. ist die Zeichenreihe 000983 eine zulässige Dezimaldarstellung und bezeichnet die gleiche Zahl wie die Zeichenreihe 983.
- Offensichtlich können führende Nullen (d.h. führende Ziffern 0, also z_0) immer weggelassen werden, außer bei der Bezeichnung "0" für die Zahl "null".
- Für jede Zahl aus $\mathbb{N}_0$ gibt es für beliebiges $p \geq 2$ eine p-adische Darstellung.
- Betrachtet man nur Darstellungen ohne führende Nullen, so ist (zu festem $p \geq 2$) die p-adische Zahldarstellung eindeutig.

Menschengerechte Darstellung der Daten

- Zur Entwicklung von Algorithmen werden wir typischerweise die Dezimaldarstellung der natürlichen Zahlen verwenden.
- Allgemein gibt es für die meisten Daten eine *Standarddarstellung* bei denen die Lesbarkeit der Darstellung für den menschlichen Benutzer im Vordergrund steht.
- Wir verwenden hier folgende Standardbezeichnungen wie sie in den üblichen höheren Programmiersprachen gebräuchlich sind:
  - Natürliche Zahlen $\mathbb{N}_0$: Dezimaldarstellung (ohne führende Nullen)
  - Ganze Zahlen $\mathbb{Z}$: Wie natürliche Zahlen, ggf. mit Vorzeichen "-"
  - Reelle Zahlen $\mathbb{R}$: *Gleitpunktdarstellung* (s. später)
  - Wahrheitswerte $\mathbb{B}$: *TRUE* und *FALSE* für "wahr" bzw. "falsch"

Darstellung von Zeichen und Texten

- Eine Dezimaldarstellung (z.B. 983) stellt eine natürliche Zahl dar, die verarbeitet werden kann.
- Die Zeichenreihe selbst (und nicht die dargestellte Zahl) könnte aber auch Gegenstand der Verarbeitung sein.
- Zeichenreihen können also nicht nur Darstellungen von Objekten sein, sondern auch selbst Objekte, die dargestellt werden müssen.
- Zeichenreihen heißen in diesem Zusammenhang *Texte*.
- In der Praxis wird zur Bildung von Texten häufig das sog. *ASCII-Alphabet* benützt.
- Das ASCII-Alphabet repräsentiert eine Menge von Zeichen, die wir im folgenden als *CHAR* bezeichnen.
- Die Elemente von *CHAR* finden Sie in allen gängigen Lehrbüchern.

Menschengerechte Darstellung von Daten

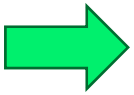
- Auf Rechenanlagen wird meist eine andere Darstellung der Daten gewählt (typischerweise Zeichenreihen über den oben benannten Alphabeten A_2, A_8 und A_{16}).
- Aus technischen Gründen kann die kleinste Speichereinheit eines Computers (ein sog. *Bit*) nur zwei Zustände speichern:
  - Zustand 1: es liegt (elektr.) Spannung an.
  - Zustand 0: es liegt keine Spannung an.
- Daher werden Werte (Daten / Objekte) als Bitmuster (Zeichenreihe über dem Alphabet $A_2 = \{0, 1\}$) codiert gespeichert.
- Intern kann der Computer also z.B. die natürlichen Zahlen in Binärcodierung repräsentieren.
- Ganze Zahlen können intern ebenfalls leicht als Zeichenkette über dem Alphabet $A_2 = \{0, 1\}$ codiert werden (Genaueres darüber werden Sie in der Vorlesung “Rechnerarchitektur” lernen).

Menschengerechte Darstellung von Daten

- Binärdarstellung reeller Zahlen ist i.d.R. etwas komplizierter.
- Üblicherweise verwenden sowohl Rechner als auch höhere Programmiersprachen die sog. *Gleitpunktdarstellung*. Eine Zahl z wird hier z.B. folgendermaßen dargestellt: $z = m \cdot 2^e$, wobei sowohl m (*Mantisse*) als auch e (*Exponent*) wiederum binär repräsentiert werden (können).
- In den meisten höheren Programmiersprachen (z.B. Java) wird eine Zahl dargestellt durch: $z = m \cdot 10^e$, z.B. 3.14, -7.45, $1.33E-2$ (für $1.33 \cdot 10^{-2}$).
- Eine genaue Spezifikation folgt im nächsten Abschnitt.
- **Wichtig**: Für viele reelle Zahlen gibt es keine solche Darstellung (z.B $\sqrt{2}$). Die darstellbaren Zahlen heißen auch *Gleitpunktzahlen*.
- Die maschinengerechte Darstellung von Daten ist bei der Entwicklung von Algorithmen möglicherweise wichtig! **Warum?**

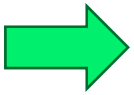
Menschengerechte Darstellung von Daten

- Weiterer Aspekt: Eine Rechenanlage hat nur begrenzte Ressourcen (Speicherzellen).



Für die Darstellung von Daten stehen immer nur endlich viele Bits zur Verfügung.

- Typischerweise definiert jede Programmiersprache elementare (*primitive* oder *atomare*) *Datentypen*, die Teilmengen der obengenannten Mengen $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$, $\mathbb{B}$, *CHAR* sind.
- Dabei stehen zur Darstellung der Werte für jeden Datentyp eine fixe Anzahl von Bits zur Verfügung, d.h. die Zeichenketten der Werte eines Typs haben eine fixe *Länge*. Es können auch nur solche Werte dargestellt werden.



Länge eines Datentyps beeinflusst den Wertebereich des Typs.

- Auch dieser Aspekt ist bei der Entwicklung von Algorithmen möglicherweise wichtig! **Warum?**

Grunddatentypen in Java

- Die Programmiersprache Java bietet folgende primitive Datentypen (1 Byte = 8 Bit):

Typname	Länge	Wertebereich
`boolean`	1 Byte	Wahrheitswerte {`true`,`false`}
`char`	2 Byte	Alle Unicode-Zeichen
`byte`	1 Byte	Ganze Zahlen von -2^7 bis $2^7 - 1$
`short`	2 Byte	Ganze Zahlen von -2^{15} bis $2^{15} - 1$
`int`	4 Byte	Ganze Zahlen von -2^{31} bis $2^{31} - 1$
`long`	8 Byte	Ganze Zahlen von -2^{63} bis $2^{63} - 1$
`float`	4 Byte	Gleitkommazahlen (einfache Genauigkeit)
`double`	8 Byte	Gleitkommazahlen (doppelte Genauigkeit)

- Es gibt in Java also Grunddatentypen für $\mathbb{B}$, *CHAR*, verschiedene Teilmengen von $\mathbb{Z}$ und verschiedene Teilmengen von $\mathbb{R}$.

Operationen auf Grunddatentypen

- Für die Verarbeitung von Objekten eines bestimmten Typs stellen höhere Programmiersprachen entspr. Operationen zur Verfügung.
- Beispiel:
  Für das "Rechnen" mit natürlichen Zahlen existieren die Grundrechenarten +, ·, usw. als Basisoperationen.
- Mathematisch formal handelt es sich hierbei typischerweise um *Funktionen*.
- Beispiel:
  Folgende Funktion addiert zwei natürliche Zahlen:

$$+ : \mathbb{N}_0 \times \mathbb{N}_0 \rightarrow \mathbb{N}_0$$

- Die Funktion + ist eine zweistellige Funktion. Grundsätzlich sind natürlich n-stellige Funktionen ($n \geq 0$) erlaubt.
  Ist diese Bedingung im Kontext von Programmiersprachen sinnvoll?

Operationen auf Grunddatentypen

- Üblicherweise schreiben wir statt "+(x, y)" "$x + y$" um die beiden Zahlen $x \in \mathbb{N}_0$ und $y \in \mathbb{N}_0$ zu addieren.
- Diese Schreibweise wird *Infixschreibweise* genannt.
- Grundsätzlich gibt es
  - *Präfixschreibweise*: im Beispiel +(x, y);
  - *Infixschreibweise*: im Beispiel (x + y);
  - *Postfixschreibweise*: im Beispiel (x, y) +;
- Operationen, die als Ergebnis Objekte vom Typ $\mathbb{B}$ ergeben, heißen auch *Prädikate,* z.B. die Operation < zum Vergleich zweier natürlicher Zahlen:

$$<: \mathbb{N}_0 \times \mathbb{N}_0 \rightarrow \mathbb{B}$$

(1 < 2 ergibt den Wert *TRUE* $\in \mathbb{B}$, 3 < 1 ergibt den Wert *FALSE* $\in \mathbb{B}$)

Typische Grundoperationen

- Boolesche (Wahrheits-)Werte $\mathbb{B} = \{TRUE, FALSE\}$:

$$\begin{array}{lcl} \wedge & : & \mathbb{B} \times \mathbb{B} \to \mathbb{B} \\ \vee & : & \mathbb{B} \times \mathbb{B} \to \mathbb{B} \\ \neg & : & \mathbb{B} \to \mathbb{B} \end{array}$$

- Natürliche Zahlen $\mathbb{N}_0 = \{0, 1, 2, \ldots\}$:

$$\begin{array}{lcl} + & : & \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\ - & : & \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\ \cdot & : & \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\ : & : & \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\ = & : & \mathbb{N} \times \mathbb{N} \to \mathbb{B} \\ \neq & : & \mathbb{N} \times \mathbb{N} \to \mathbb{B} \\ > & : & \mathbb{N} \times \mathbb{N} \to \mathbb{B} \\ < & : & \mathbb{N} \times \mathbb{N} \to \mathbb{B} \end{array}$$

Typische Grundoperationen

- Ganze Zahlen $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$:
  $+, -, \cdot, \div, =, \neq, <, >, \ldots$
- Reelle Zahlen $\mathbb{R}$:
  $+, -, \cdot, \div, =, \neq, <, >, \ldots$
- Zeichen (Character) *CHAR*, z.B. alle druckbaren ASCII-Zeichen = {“A”, “B”, . . ., “a”, “b”, . . ., “1”, “2”, . . ., “!”, . . .}:
  $=, \neq, <, >, \ldots$
- **Achtung**: obwohl z.B. $+ : \mathbb{N}_0 \times \mathbb{N}_0 \rightarrow \mathbb{N}_0$ und
  $+ : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$
  gleich “benannt” sind, sind es zwei unterschiedliche Operationen!
- Wenn zwei unterschiedliche Operationen gleich benannt sind, spricht man von *Überladen*.
- **Achtung**: Auch die Operationen auf Grunddatentypen basieren wiederum auf Algorithmen.

Syntax

- Die Gestalt einer Datendarstellung nennt man *Syntax*.
- Die Bedeutung der dargestellten Objekte heißt *Semantik*.
- Die Syntax von Darstellungen von Daten / Objekten kann ohne Bezugnahme auf die Semantik definiert werden.

Syntaxdefinition von natürlichen Zahlen

- Die Menge der natürlichen Zahlen in Dezimaldarstellung kann durch folgende Regeln definiert werden:
  1. 0 ist eine <*Dezimalzahl*>.
  2. Jede Ziffer $x \in A_{10} \setminus \{0\}$ ist eine <*Nichtnulldarstellung*>.
  3. Ist a eine <*Nichtnulldarstellung*> und $y \in A_{10}$ so ist $a \circ y$ eine <*Nichtnulldarstellung*>.
  4. Jede <*Nichtnulldarstellung*> ist eine <*Dezimalzahl*>.
- Dabei bezeichnet $\circ$ die *Konkatenation* zweier Zeichenketten, z.B. ergibt die Konkatenation $123 \circ 456$ die Zeichenkette 123456.
- Beispiel: Die Zeichenreihe 308 ist eine Dezimalzahl gemäß folgender Anwendungen der Regeln:

3	ist <*Nichtnulldarstellung*> nach Regel 2
30	ist <*Nichtnulldarstellung*> nach Regel 3
308	ist <*Nichtnulldarstellung*> nach Regel 3
308	ist <*Dezimalzahl*> nach Regel 4

Syntaxdefinition von natürlichen Zahlen

- Die Begriffe <*Dezimalzahl*> und <*Nichtnulldarstellung*> in den Regeln werden *syntaktische Variablen* genannt. Sie kennzeichnen den zu definierenden Begriff sowie einen Hilfsbegriff.
- Allgemein können in Syntaxdefinitionen noch mehr syntaktische Variablen vorkommen. Wir heben sie durch kursive Schrift und die eckigen Klammern hervor.
- Die Zeichen des vorgegeben Alphabets heißen *Terminalzeichen*.

Backus-Naur-Form

- Eine formale, häufig gebrauchte Form von Syntaxdefinitionen ist die *Backus-Naur-Form* (*BNF*)
- Eine Syntaxdefinition in BNF ist gegeben durch eine endliche Anzahl von *Syntaxregeln* der Form

$$\alpha ::= \beta$$

- Dabei ist α eine syntaktische Variable und β eine *BNF-Satzform* (siehe nächste Folie).
- Eine ausgezeichnete syntaktische Variable ist das *Startsymbol* α_S.

BNF-Satzform

- Syntaktaktische Variablen und Terminalzeichen sind BNF-Satzformen.
- Auswahl:
  Sind $\gamma_1, \ldots, \gamma_n$ BNF-Satzformen, dann auch $\gamma_1 | \ldots | \gamma_n$.
- Verkettung (Konkatenation):
  Sind $\gamma_1, \ldots, \gamma_n$ BNF-Satzformen, dann auch $\gamma_1 \ldots \gamma_n$.
- Klammerung:
  Ist γ eine BNF-Satzform, dann auch (γ).
- Iteration:
  Ist γ eine BNF-Satzform, dann auch $(\gamma)^*$.
- Nichtleere Iteration:
  Ist γ eine BNF-Satzform, dann auch $(\gamma)^+$.
- Option:
  Ist γ eine BNF-Satzform, dann auch $[\gamma]$.

Bedeutung (Semantik) der BNF

- Eine Zeichenreihe w ist *herleitbar* mit einer BNF Syntaxdefinition, wenn man w aus α_S durch eine oder mehrere der folgenden Ersetzungsoperationen erhalten kann:
  - Eine syntaktische Variable α, für die die Regel $\alpha ::= \gamma_1 | \ldots | \gamma_n$ vorhanden ist, darf durch eines der γ_i $(1 \leq i \leq n)$ ersetzt werden.
  - Ein Vorkommen von $(\gamma_1 | \ldots | \gamma_n)$ darf man durch eines der γ_i $(1 \leq i \leq n)$ ersetzen.
  - Ein Vorkommen von $(\gamma)^*$ darf man durch $\underbrace{(\gamma)(\gamma)\ldots(\gamma)}_{n-\text{mal}}$ mit einem beliebigen $(n \geq 0)$ ersetzen.
  - Ein Vorkommen von $(\gamma)^+$ darf man durch $\underbrace{(\gamma)(\gamma)\ldots(\gamma)}_{n-\text{mal}}$ mit einem beliebigen $(n > 0)$ ersetzen.
  - Ein Vorkommen von (γ) darf man durch γ ersetzen falls γ nicht von der Form $\gamma_1 | \ldots | \gamma_n$ ist.
  - Ein Vorkommen von $[\gamma]$ darf man durch (γ) ersetzen, oder ersatzlos streichen.
- Wir schreiben auch $\alpha_S \rightarrow w$.

Beispiele

- BNF Syntaxdefinition für natürliche Zahlen:

  <*Dezimalzahl*> ::= 0 | <*Nichtnulldarstellung*>

  <*Nichtnulldarstellung*> ::= <*Nichtnullziffer*> (0 | <*Nichtnullziffer*>)*

  <*Nichtnullziffer*>::= 1|2|3|4|5|6|7|8|9

  Die syntaktische Variable <*Dezimalzahl*> dient als Startsymbol α_S.

- Ableitung der Zeichenreihe 308 aus α_S = <*Dezimalzahl*> ist:

  <*Dezimalzahl* >

  → <*Nichtnulldarstellung*>

  → <*Nichtnullziffer*> (0 | <*Nichtnullziffer*>)*

  → <*Nichtnullziffer*> (0 | <*Nichtnullziffer*>)(0 | <*Nichtnullziffer*>)

  → <*Nichtnullziffer*> 0 <*Nichtnullziffer*>

  → 308

Beispiele

- BNF Syntaxdefinition für ganze Zahlen (“−” und die Dezimalziffern des Alphabets A_{10} sind Terminalzeichen):

  <*GanzeZahl*> ::= [−] <*Dezimalzahl*>

  Die syntaktische Variable <*GanzeZahl*> dient hier als Startsymbol α_S.

- BNF Syntaxdefinition für Gleitpunktzahlen (“−”, “E” und die Dezimalziffern des Alphabets A_{10} sind Terminalzeichen) :

  <Gleitpunktzahl> ::= [−] <Mantisse>(E<GanzeZahl>)

  <Mantisse> ::= <Dezimalzahl>.<Dezimalstellen>

  <Dezimalstellen>::= 0|<Nichtnullstellen>

  <Nichtnullstellen> ::= (0|<Nichtnullziffer>)*<Nichtnullziffer>

  Die syntaktische Variable <*Gleitpunktzahl*> dient hier als Startsymbol α_S.

Diskussion

- Mit Hilfe der BNF kann man nicht nur die Syntax (Darstellung) von Daten formal definieren.
- Viele Programmiersprachen benützen BNF Syntaxdefinitionen auch für die eindeutige Definition ihrer Syntax, d.h. der Spracheelemente, die erlaubt sind und der Sprache an sich.
- Manche Eigenschaften von Darstellungen oder Sprachen kann man in BNF allerdings nicht darstellen. Diese werden dann typischerweise zusätzlich "verbal" angegeben, d.h. sie werden als Zusatzanforderungen notiert. Diese zusätzlichen Angaben heißen *Kontextbedingungen*.

Eigenschaften von Algorithmen

- Nachdem wir die erste der in Kapitel 1 genannten Eigenschaften, “eindeutige Datendarstellung” , diskutiert haben, widmen wir uns im folgenden den restlichen drei Eigenschaften von Algorithmen.
- Neben diesen Forderungen gibt es noch weitere interessante Eigenschaften, die Algorithmen haben können.
- Wir werden all diese Eigenschaften an einem Beispiel erläutern.

Eigenschaften von Algorithmen

Die restlichen drei Eigenschaften von Algorithmen aus Kapitel 1 sind:

1. Präzise, endliche Beschreibung.
2. Effektiver Verarbeitungsschritt.
3. Elementarer Verarbeitungsschritt.

Dazu kommen folgende Eigenschaften:

4. Ein Algorithmus heißt *terminierend*, wenn er bei jeder Anwendung nach endlich vielen Verarbeitungsschritten zum Ende kommt.
5. Ein Algorithmus heißt *deterministisch*, wenn die Wirkung und die Reihenfolge der Einzelschritte eindeutig festgelegt ist, andernfalls *nicht-deterministisch*.
6. Ein Algorithmus heißt *determiniert*, wenn das Ergebnis der Verarbeitung für jede einzelne Anwendung eindeutig bestimmt ist, andernfalls *nicht-determiniert*.

Eigenschaften von Algorithmen

Desweiteren ist natürlich die möglicherweise wichtigste Eigenschaft eines Algorithmus die *Korrektheit*, d.h. informell, “der Algorithmus tut, was er tun soll”.

1. Ein Algorithmus heißt *partiell korrekt*, wenn für alle gültigen Eingaben das Resultat der Spezifikation des Algorithmus entspricht.
2. Ein Algorithmus heißt *(total) korrekt*, wenn der Algorithmus partiell korrekt ist, und für alle gültigen Eingaben terminiert.

Wir werden uns später noch genauer mit der Frage beschäftigen, wie man sicher sein kann, dass ein Algorithmus partiell / total korrekt ist.

Running Example

Wir betrachten folgende Aufgabe:

Ein Kunde kauft Waren für $1 \leq r \leq 100$ EUR und bezahlt mit einem 100 EUR Schein (r sei ein voller EUR Betrag ohne Cent-Anteil). Gesucht ist ein Algorithmus, der zum Rechnungsbetrag r das Wechselgeld w bestimmt. Zur Vereinfachung nehmen wir an, dass w nur aus 1 EUR oder 2 EUR Münzen oder 5 EUR Scheinen bestehen soll. Es sollen möglichst wenige Münzen / Scheine ausgegeben werden (also ein 5 EUR Schein statt fünf 1 EUR Münzen).

Datendarstellung

- Zunächst müssen wir zur Lösung dieser Aufgabe die Darstellung (Modellierung) der relevanten Daten festlegen.
- Für den Rechnungsbetrag r ist dies trivial, denn offensichtlich ist $r \in \mathbb{N}$. Wir nehmen an, dass r in Dezimaldarstellung gegeben ist.
- Das Wechselgeld w kann auf verschiedeneWeise modelliert werden, z.B. als Folge oder Multimenge von Wechselgeld-münzen. Ein aus zwei 1-EUR-Münzen, einer 2-EUR-Münze und zwei 5-EUR-Scheinen bestehendes Wechselgeld könnte als Folge (1, 1, 2, 5, 5) dargestellt sein.

Datendarstellung

- Wir legen folgende Datendarstellung fest:
  - r: als natürliche Zahl in Dezimaldarstellung.
  - w: als Folge von Werten 1, 2 oder 5.
- Wir benutzen dabei die Bezeichnung () für die *leere Folge* und die Funktion $\circ$ zur Konkatenation zweier Folgen wie oben. Desweiteren sagen wir auch "nimm x zu w hinzu" für die Operation $w \circ x$.

Erster Lösungsansatz

- Idee: Ausgehend von r sukzessive um 1, 2 und 5 hochzählen (unter Hinzunahme der entsprechenden Münzen / Scheine) bis man bei 100 angelangt ist. Dabei möglichst schnell eine durch 5 teilbare Zahl erreichen, um möglichst wenige Münzen / Scheine auszugeben.
- Algorithmus *Wechselgeld 1*
  Führe folgende Schritte der Reihe nach aus:
  1. Setze $w = ()$.
  2. Falls die letzte Ziffer von r eine 2, 4, 7 oder 9 ist, dann erhöhe r um 1 und nimm 1 zu w hinzu.
  3. Falls die letzte Ziffer von r eine 1 oder 6 ist, dann erhöhe r um 2 und nimm 2 zu w hinzu.
  4. Falls die letzte Ziffer von r eine 3 oder 8 ist, dann erhöhe r um 2 und nimm 2 zu w hinzu.
  5. Solange $r < 100$: Erhöhe r um 5 und nimm 5 zu w hinzu.

Erster Lösungsansatz

- Ablaufbeispiel: Sei $r = 81$.

	$r = 81$	(Ausgangssituation)
Schritt 1	$r = 81$	$w = ()$
Schritt 2	(keine Änderung, da die letzte Ziffer keine 2, 4, 7, 9 ist)	
Schritt 3	$r = 83$	$w = (2)$
Schritt 4	$r = 85$	$w = (2, 2)$
Schritt 5	$r = 90$	$w = (2, 2, 5)$
	$r = 95$	$w = (2, 2, 5, 5)$
	$r = 100$	$w = (2, 2, 5, 5, 5)$

- Eigenschaften:
  - Endliche Aufschreibung: OK.
  - Effektive und elementare Einzelschritte: **?**
  - Der Algorithmus ist terminierend, deterministisch, determiniert und partiell korrekt (und damit auch total korrekt).

Variante des ersten Lösungsansatzes

- Idee: Fasse ähnliche Schritte zusammen.
- Algorithmus *Wechselgeld 2*
  Führe folgende Schritte der Reihe nach aus:
  1. Setze $w = ()$.
  2. Solange $r < 100$: Führe jeweils (wahlweise) einen der folgenden Schritte aus:
     1. Falls die letzte Ziffer von r eine 2, 4, 7 oder 9 ist, dann erhöhe r um 1 und nimm 1 zu w hinzu.
     2. Falls die letzte Ziffer von r eine 1, 2, 3, 6, 7 oder 8 ist, dann erhöhe r um 2 und nimm 2 zu w hinzu.
     3. Falls die letzte Ziffer von r eine 0, 1, 2, 3, 4 oder 5 ist, dann erhöhe r um 5 und nimm 5 zu w hinzu.

Variante des ersten Lösungsansatzes

- Ablaufbeispiel:

	$r = 81$	(Ausgangssituation)
Schritt 1	$r = 81$	$w = ()$
Schritt 2b	$r = 83$	$w = (2)$
Schritt 2b	$r = 85$	$w = (2, 2)$
Schritt 2c	$r = 90$	$w = (2, 2, 5)$
Schritt 2c	$r = 95$	$w = (2, 2, 5, 5)$
Schritt 2c	$r = 100$	$w = (2, 2, 5, 5, 5)$

- Alternativer Ablauf:

	$r = 81$	(Ausgangssituation)
Schritt 1	$r = 81$	$w = ()$
Schritt 2b	$r = 83$	$w = (2)$
Schritt 2c	$r = 88$	$w = (2, 5)$
Schritt 2b	$r = 90$	$w = (2, 5, 2)$
Schritt 2c	$r = 95$	$w = (2, 5, 2, 5)$
Schritt 2c	$r = 100$	$w = (2, 5, 2, 5, 5)$

Variante des ersten Lösungsansatzes

- Eigenschaften:
  - Algorithmus 2 arbeitet nach dem selben Grundprinzip wie Algorithmus 1, lässt aber gewisse "Freiheiten" bei Schritt 2.
    ⇒ Algorithmus 2 ist nicht-deterministisch.
  - Algorithmus 2 ist nicht determiniert (siehe alternativer Ablauf)
  - Bemerkung: Wenn wir statt Folgen Multimengen (Reihenfolge der Elemente spielt keine Rolle) bei der Darstellung von w verwendet hätten, wären die Ergebnisse $w = \{2, 2, 5, 5, 5\}$ und $w = \{2, 5, 2, 5, 5\}$ gleich, d.h. in diesem Fall wäre Algorithmus 2 determiniert.
- Die Wahl der Datendarstellung kann also Einfluss auf Eigenschaften von Algorithmen haben.
- Bemerkung: Würden wir r nicht in Dezimal- sondern z.B. in Binärdarstellung modellieren, wären beide Algorithmen in der angegebenen Form sinnlos.

ZweiteVariante des ersten Lösungsansatzes

- Idee: Wir lösen uns von der Abhängigkeit von der Datendarstellung, indem wir r nicht mehr hochzählen, sondern die Differenz $100 - r$ berechnen und diese in möglichst wenige Teile der Größen 1, 2 und 5 aufteilen. Wir gehen dabei davon aus, dass die Differenz "–" für beliebige Zifferndarstellungen (dezimal, binär, oktal, etc.) definiert ist. Die Zahl 100 müsste in die selbe Darstellung gebracht werden, die auch für r benutzt wird.
- Algorithmus *Wechselgeld 3*
  Führe folgende Schritte der Reihe nach aus:
  1. Berechne $d = 100 - r$ und setze $w = ()$.
  2. Solange $d \geq 5$: Vermindere d um 5 und nimm 5 zu w hinzu.
  3. Falls $d \geq 2$, dann vermindere d um 2 und nimm 2 zu w hinzu.
  4. Falls $d \geq 2$, dann vermindere d um 2 und nimm 2 zu w hinzu.
  5. Falls $d \geq 1$, dann vermindere d um 1 und nimm 1 zu w hinzu.

ZweiteVariante des ersten Lösungsansatzes

- Ablaufbeispiel:

	$r = 81$	(Ausgangssituation)
Schritt 1	$d = 19$	$w = ()$
Schritt 2	$d = 14$	$w = (5)$
	$d = 9$	$w = (5, 5)$
	$d = 4$	$w = (5, 5, 5)$
Schritt 3	$d = 2$	$w = (5, 5, 5, 2)$
Schritt 4	$d = 0$	$w = (5, 5, 5, 2, 2)$
Schritt 5	(keine Änderung, da $d \geq 1$ nicht gilt)	

Diskussion

- Alle drei Varianten weisen eine gemeinsame “operative” Auffassung auf: Die Berechnung des Rückgelds wird durch eine Folge von “Handlungen” vollzogen.
- Diese Handlungen sind charakterisiert durch Veränderungen der Größen *r* bzw. *d* und *w*.
- Die Abfolge dieser “Aktionen” wird durch typische Konstruktionen gesteuert:
  - *Fallunterscheidung*: Falls . . ., dann . . .
    ermöglicht, die Ausführungen von Aktionen vom Erfülltsein gewisser Bedingungen abhängig sein zu lassen.
  - *Iteration*: Solange . . .: . . .
    steuert die Wiederholung von Aktionen in Abhängigkeit gewisser Bedingungen.

Diskussion

- Neben der strikten operativen Auffassung gibt es eine Sichtweise von Algorithmen, die durch eine rigorose mathematische Abstraktion gewonnen wird.
- In unserem Beispiel ist die Zuordnung eines Wechselgelds w zu einem Rechnungsbetrag r mathematisch nichts anderes als eine Abbildung $h : r \rightarrow$ "herauszugebendesWechselgeld"
- Die Aufgabe ist dann, eine Darstellung der Abbildung h zu finden, die "maschinell auswertbar" ist.
- Triviale Lösung: Auflistung aller möglichen Werte für r mit zugehörigemWechselgeld $w = h(r)$. **Probleme?**
- Allgemein wird man eine kompakte Darstellung suchen, in der die zu bestimmende Abbildung in geeigneterWeise aus einfachen (elementar auswertbaren) Abbildungen zusammengesetzt ist.

Vorbereitungen

- Für eine solche Darstellung der Abbildung h benötigen wir zwei "Hilfsabbildungen", die in der Informatik oft gebräuchlich sind:

$$DIV : \mathbb{N}_0 \times \mathbb{N} \to \mathbb{N}_0$$
$$MOD : \mathbb{N}_0 \times \mathbb{N} \to \mathbb{N}_0$$

- *DIV* berechnet das Ergebnis der ganzzahlige Division zweier natürlicher Zahlen, z.B. $DIV(7, 3) = 2$.
- *MOD* berechnet den Rest der ganzzahligen Division zweier natürlicher Zahlen, z.B. $MOD(7, 3) = 1$.
- Formal: Sind $k \in \mathbb{N}_0$ und $l \in \mathbb{N}$, so kann man k durch l mit ganzzahligem Ergebnis q teilen, wobei eventuell ein Rest r übrigbleibt, d.h. es gibt $q, r \in \mathbb{N}_0$ mit $r < l$ und $k = q \cdot l + r$. Dann ist

$$DIV(k, l) = q \qquad \text{und} \qquad MOD(k, l) = r.$$

Zweiter Lösungsansatz

- Idee: Ähnlich wie Algorithmus 3 teilen wir $100 - r$ durch 5. Der ganzzahlige Quotient $q_1 = DIV(100 - r, 5)$ ist die Anzahl der 5-EUR-Scheine im Wechselgeld $h(r)$. Der Rest $r_1 = MOD(100 - r, 5)$ ist der noch zu verarbeitendeWechselbetrag. Offensichtlich gilt $r_1 < 5$. r_1 muss nun auf 1 und 2 aufgeteilt werden, d.h. analog bilden wir $q_2 = DIV(r_1, 2)$ und $r_2 = MOD(r_1, 2)$. q_2 bestimmt die Anzahl der Elemente 2 und r_2 die Anzahl der Elemente 1 in $h(r)$.
- Algorithmus *Wechselgeld 4*

$$h(r) = (5_1, \ldots, 5_{DIV(100-r,\,5)}\ 2_1, \ldots, 2_{DIV(MOD(100-r,\,5),\,2)}\ 1_1, \ldots, 2_{MOD(MOD(100-r,\,5),\,2)})$$

Dabei sind alle Elemente in der Ergebnisfolge durchnummeriert, um die Anzahl der entsprechenden Elemente anzudeuten, d.h. das Element 5_i bezeichnet den i-ten 5-EUR-Schein im Ergebnis.

Zweiter Lösungsansatz

- Bemerkung:
  In dieser Schreibweise kann z.B. $(5_1, \ldots 5_0)$ auftreten, was bedeuten soll, dass die Folge kein Element 5 enthält.
- Beispielanwendung $r = 81$:

$$
\begin{array}{lll}
DIV(100 - r, 5) & = DIV(19, 5) & = 3 \\
MOD(100 - r, 5) & = MOD(19, 5) & = 4 \\
DIV(MOD(100 - r, 5), 2) & = DIV(4, 2) & = 2 \\
MOD(MOD(100 - r, 5), 2) & = MOD(4, 2) & = 0
\end{array}
$$

d.h. man erhält $h(81) = (5_1, 5_2, 5_3, 2_1, 2_2, 1_1, 1_0)$ oder, einfacher geschrieben,

$$h(81) = (5, 5, 5, 2, 2).$$

Zweiter Lösungsansatz

- Eigenschaften:
  - Wenn man von der Bildung der Folgen aus den berechneten Anzahlen der Elemente 1, 2 und 5 absieht, sind nur die Auswertungen der Operationen "–", *DIV* und *MOD* als Einzelschritte anzusehen. Setzt man diese als elementar voraus, ist die angegebene Definition von *h* eine Beschreibung des gesuchten Algorithmus.
  - Diese Darstellung nimmt keinen Bezug mehr auf eine "Abfolge von Aktionen" sondern beschreibt den "funktionalen Zusammenhang" zwischen *r* und *w* durch ineinander eingesetzte Anwendungen gewisser Basisoperationen.

Variante des zweiten Lösungsansatzes

- Leider lässt sich in vielen Fällen die gesuchte Abbildung nicht in derartig einfacherWeise explizit angeben.
- Ein wichtiges Prinzip für den Allgemeinfall von Abbildungen auf natürlichen Zahlen (und anderen Mengen) ist das Prinzip der *Rekursion*, das wir im nächsten Abschnitt noch etwas genauer untersuchen, hier aber schon einmal informell einführen.
- Kernidee: Für einen oder mehrere *Basisfälle* wird die Abbildung der Funktion f explizit angegeben (typischerweise $f(0)$), und der allgemeine Rekursionsfall $f(n)$ wird auf den Fall $f(n-1)$ zurückgeführt.

Variante des zweiten Lösungsansatzes

- Idee: Auch im Fall der Abbildung $h \rightarrow$ "Folgen über 1, 2, 5" kann man sehr leicht eine rekursive Definition finden, die die Operationen *DIV* und *MOD* nicht mehr verwendet.
- Algorithmus *Wechselgeld 5*

$$h(r) = \begin{cases} \text{falls } r = 100, \text{ dann } \quad (), & (1) \\ \text{falls } 100 - r \geq 5, \text{ dann } \quad (5) \circ h(r+5), & (2) \\ \text{falls } 5 > 100 - r \geq 2, \text{ dann } \quad (2) \circ h(r+2), & (3) \\ \text{falls } 2 > 100 - r \geq 1, \text{ dann } \quad (1) \circ h(r+1), & (4) \end{cases}$$

- Hier ist der Basisfall (1) für $h(100)$ definiert und die Rekursionsfälle (2), (3), (4) werden auf die Fälle $h(r + 5)$, $h(r + 2)$ und $h(r + 1)$ zurückgeführt.

Variante des zweiten Lösungsansatzes

- Beispielanwendung $r = 81$:

$$
\begin{aligned}
h(81) &= (5) \circ h(86) && \text{(Fall 2)}\\
&= (5) \circ (5) \circ h(91) && \text{(Fall 2)}\\
&= (5) \circ (5) \circ (5) \circ h(96) && \text{(Fall 2)}\\
&= (5) \circ (5) \circ (5) \circ (2) \circ h(98) && \text{(Fall 3)}\\
&= (5) \circ (5) \circ (5) \circ (2) \circ (2) \circ h(100) && \text{(Fall 3)}\\
&= (5) \circ (5) \circ (5) \circ (2) \circ (2) \circ () && \text{(Fall 1)}\\
&= (5, 5, 5, 2, 2)
\end{aligned}
$$

Variante des zweiten Lösungsansatzes

- Trotz der rein funktionalen Darstellung der Zuordnung $r \rightarrow w$ erinnert die Auswertung wieder an die Abfolge der Aktionen im entsprechenden Ablaufbeispiel (ähnlich wie in Algorithmus 3).
- Tatsächlich kann das Prinzip der Rekursion auch in der operativen Auffassung der Algorithmen 1-3 eingerichtet werden und dabei einen ähnlichen Effekt wie die Iteration erzielen.
- In einer rekursiven Abbildungsdefinition greift man auf Werte dieser Abbildung für kleinere (oder hier: größere) Argumente zurück.
- Analog kann in einem Algorithmus, der die Abfolge gewisser Aktionen in Abhängigkeit von Eingabewerten steuert, die Ausführung des Algorithmus selbst auch als eine derartige Aktion auftreten.

Zweite Variante des zweiten Lösungsansatzes

- Algorithmus *Wechselgeld 6*
  Setze $w = ()$ und führe anschließend folgenden in Abhängigkeit der Eingabegröße r rekursiv definierten Algorithmus $A(r)$ aus:
  $A(r)$: Führe denjenigen der folgenden Schritte (1) - (3) aus, dessen Bedingung erfüllt ist (ist keine Bedingung erfüllt, ist die Ausführung zu Ende):
  1. Falls $100 - r \geq 5$, dann nimm 5 zu w hinzu und führe anschließend $A(r + 5)$ aus.
  2. Falls $5 > 100 - r \geq 2$, dann nimm 2 zu w hinzu und führe anschließend $A(r + 2)$ aus.
  3. Falls $5 > 100 - r \geq 1$, dann nimm 1 zu w hinzu und führe anschließend $A(r + 1)$ aus.

Diskussion

- Grundlegende algorithmische Konzepte sind (z.T. abhängig vom verwendeten Programmierparadigma):
  - Eingabe- und Ergebnisdaten (insbesondere solche von komplexer Art) müssen geeignet dargestellt werden.
  - Die Ausführung gewisser Aktionen bzw. die Auswertung gewisser Operationen wird als in elementaren Schritten durchführbar vorausgesetzt.
  - Einzelne Schritte können zusammengesetzt werden (z.B. Nacheinanderausführung, Auswahl von Aktionen, Kompositionen von Abbildungen, etc.)
  - Fallunterscheidung, Iteration und Rekursion ermöglichen die Steuerung des durch den Algorithmus beschriebenen Verfahrens.

Diskussion

- *Imperative Algorithmen*:
  - Abfolge von Aktionen, die typischerweise bestimmte Größen verändern.
  - Enspricht auch den technischen Möglichkeiten von Rechneranlagen, die Schritt für Schritt bestimmte grundlegende Aktionen ausführen können.
- *Funktionale Algorithmen*:
  - Auswertbare Darstellung des funktionalen Zusammenhangs zwischen Ein- und Ausgabewerten.
  - Höheres Abstraktionsniveau: Obwohl zur Auswertung der entsprechenden Abbildung wiederum bestimmte Schritte ausgeführt werden, ist die eigentliche Spezifikation des Algorithmus als Abbildung praktisch losgelöst von den technischen Möglichkeiten der Rechenanlage.
- *Objektorientierte Algorithmen*:
  - Höhere Abstraktionsebene zur Darstellung von komplexen Eingabe- und Ergebnisdaten sowie Zwischenzuständen während der Berechnung.
  - Eigentlichen Lösung der gegebenen Aufgabe können sowohl über einen funktionalen als auch über einen imperativen Algorithmus erfolgen.