Einführung in die Anwendungsorientierte Informatik (Köthe)

Robin Heinemann

November 11, 2016

Contents

1 Klausur 10.02.2016

2 Was ist Informatik?

"Kunst" Aufgaben mit Computerprogrammen zu lösen.

2.1 Teilgebiete

2.1.1 theoretische Informatik (ITH)

- Berechenbarkeit: Welche Probleme kann man mit Informatik lösen und welche prinzipiell nicht?
- Komplexität: Welche Probleme kann man effizient lösen?
- Korrektheit: Wie beweist man, dass das Ergebnis richtig ist?
  Echtzeit: Dass das richtige Ergebnis rechtzeitig vorliegt.
- verteilte Systeme: Wie sichert man, dass verteilte Systeme korrekt kommunizieren?

2.1.2 technische Informatik (ITE)

- Auf welcher Hardware kann man Programme ausführen, wie baut man dies Hardware?
- CPU, GPU, RAM, HD, Display, Printer, Networks

2.1.3 praktische Informatik

- Wie entwickelt man Software?
- Programmiersprachen und Compiler: Wie kommuniziert der Programmierer mit der Hardware? **IPI**, **IPK**
- Algorithmen und Datenstrukturen: Wie baut man komplexe Programme aus einfachen Grundbausteinen? **IAL**
- Softwaretechnik: Wie organisiert man sehr große Projekte? **ISW**
- Kernanwendung der Informatik: Betriebsysteme, Netzwerke, Parallelisierung**IBN**

- Datenbanksysteme **IDB1**
- Graphik, Graphische Benutzerschnittstellen **ICG1**
- Bild- und Datenanalyse
- maschinelles Lernen
- künstliche Intelligenz

2.1.4 angewante Informatik

- Wie löst man Probleme aus einem anderem Gebiet mit Programmen?
- Informationstechnik
  - Buchhandlung, e-commerce, Logistik
- Web programming
- scientific computing für Physik, Biologie
- Medizininformatik
  - bildgebende Verfahren
  - digitale Patientenakte
- computer linguistik
  - Sprachverstehen, automatische Übersetzung
- Unterhaltung: Spiele, special effect im Film

3 Wie unterscheidet sich Informatik von anderen Disziplinen?

3.1 Mathematik

Am Beispiel der Definition $a \leq b : \exists c \geq 0 : a + c = b$
Informatik:
Lösungsverfahren: $a - b \leq 0$, das kann man leicht ausrechen, wenn man subtrahieren und mit 0 vergleichen kann.
Quadratwurzel: $y = \sqrt{x} \Leftrightarrow y \geq 0 \wedge y^2 = x (\Rightarrow x > 0)$
Informatik: Algorithmus aus der Antike: $y = \frac{x}{y}$ iteratives Verfahren:
Initial Guess $y^{(0)} = 1$ schrittweise Verbesserung $y^{(t+1)} = \frac{y^{(t)} + \frac{x}{y^{(t)}}}{2}$

4 Informatik

Lösugswege, genauer Algorithmen

4.1 Algorithmus

schematische Vorgehensweise mit der jedes Problem einer bestimmten **Klasse** mit **endliche** vielen **elementaren** Schritten / Operationen gelöst werden kann

- schematisch: man kann den Algorithmus ausführen, ohne ihn zu verstehen ($\Rightarrow$ Computer)
- alle Probleme einer Klasse: zum Beispiel: die Wurzel aus jeder beliebigen nicht-negativen Zahl, und nicht nur $\sqrt{11}$
- endliche viele Schritte: man kommt nach endlicher Zeit zur Lösung
- elementare Schrite / Operationen: führen die Lösung auf Operationen oder Teilprobleme zurück, die wir schon gelöst haben

4.2 Daten

Daten sind Symbole,

- die Entitäten und Eigenschaften der realen Welt im Computer representieren.
- die interne Zwischenergebnisse eines Algorithmus aufbewahren

$\Rightarrow$ Algorithmen transformieren nach bestimmten Regeln die Eingangsdaten (gegebene Symbole) in Ausgangsdaten (Symbole für das Ergebniss). Die Bedeutung / Interpretation der Symbole ist dem Algorithmus egal $\triangleq$ "schematisch"

4.2.1 Beispiele für Symbole

- Zahlen
- Buchstaben
- Icons
- Verkehrszeichen

aber: heutige Computer verstehen nur Binärzahlen $\Rightarrow$ alles andere muss man übersetzen
Eingansdaten: "Ereignisse":

- Symbol von Festplatte lesen oder per Netzwerk empfangen
- Benutzerinteraktion (Taste, Maus, ...)
- Sensor übermittelt Meßergebnis, Stoppuhr läuft ab

Ausgangsdaten: "Aktionen":

- Symbole auf Festplatte schreiben, per Netzwerk senden

- Benutzeranzeige (Display, Drucker, Ton)
- Stoppuhr starten
- Roboteraktion ausführen (zum Beispiel Bremsassistent)

Interne Daten:

- Symbole im Hauptspeicher oder auf Festplatte
- Stoppuhr starten / Timeout

4.3 Einfachster Computer

endliche Automaten (endliche Zustandsautomaten)

- befinden sich zu jedem Zeitpunkt in einem bestimmten Zustand aus einer vordefinierten endlichen Zustandsmenge
- äußere Ereignisse können Zustandsänderungen bewirken und Aktionen auslösen

4.3.1 TODO Graphische Darstellung

graphische Darstellung: Zustände = Kreise, Zustandsübergänge: Pfeile

4.3.2 TODO Darstellung durch Übergangstabellen

Zeilen: Zustände, Spalten: Ereignisse, Felder: Aktion und Folgezustände

Zustände \ Ereignisse	Knopf drücken	Timeout
aus	⇒{halb}	
{4 LEDs an}	%	(⇒{aus},{nichts})
halb	(⇒{voll},{8 LEDs an})	%
voll	(⇒{blinken an},{Timer starten})	%
blinken an	(⇒{aus},{Alle LEDs aus, Timer stoppen})	(⇒{blinken aus},{alle LEDs aus, T
blinken aus	(⇒{aus},{Alle LEDs aus, Timer stoppen})	(⇒{blinken an},{alle LEDs an, Ti

Variante: Timer läuft immer (Signal alle 0.3s) ⇒ Timout ignorieren im Zustand "aus", "halb", "voll"

4.3.3 Beispiel 2:

$$1\,0\,1\,1\,0\,1\,0 = 2 + 8 + 16 + 74 = 90_{\text{dez}} \tag{1}$$

$$+0\,1\,1\,1\,0\,0\,1 = 1 + 8 + 16 + 32 = 57_{\text{dez}} \tag{2}$$

$$1\,0\,0\,1\,0\,0\,1\,1 = 1 + 2 + 16 + 128 = 147_{\text{dez}}\checkmark \tag{3}$$

Implementation mit Endlichen Automaten Prinzipen:

- wir lesen die Eingangsdaten von rechts nach links
- Beide Zahlen gleich lang (sonst mit 0en auffüllen)
- Ergebnis wird von rechts nach link ausgegeben

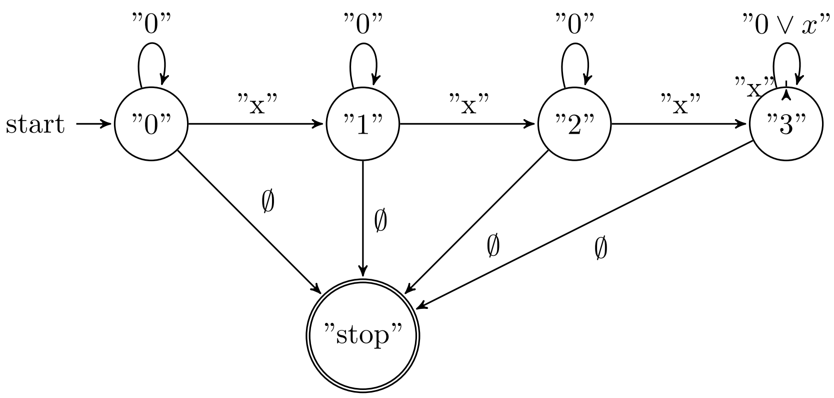
TODO Skizze der Automaten

Zustand	Ereignis	Ausgeben
start	(0,1)	"1"
start	(1,0)	"1"
start	(0,0)	"0"
start	(1,1)	"0"
carry = 1	(1,1)	"1"
carry = 1	(0,1)	"0"
carry = 1	(1.0)	"0"
carry = 1	$\emptyset$	"1"

Wichtig: In jedem Zustand muss für **alle möglichen** Ereignisse eine Aktion und Folgezustand definiert werden. Vergisst man ein Ereignis zeigt der Automat undefiniertes Verhalten, also einen "Bug". Falls keine sinvolle Reaktion möglich ist: neuer Zustand: "Fehler" ⇒ Übergang nach "Fehler", Aktion: Ausgeben einer Fehlermeldung

TODO Skizze Fehlermeldung Ein endlicher Automat hat nur ein Speicherelement, das den aktuelen Zustand angibt. Folge:

- Automat kann sich nicht merken, wie er in den aktuellen Zustand gekommen ist ("kein Gedächnis")
- Automat kann nicht beliebig weit zählen, sondern nur bis zu einer vorgegebenen Grenze



Insgesamt: Man kann mit endlichen Automaten nur relativ einfache Algorithmen implementieren. (nur reguläre Sprachen) Spendiert man zusätzlichen Specher, geht mehr:

- Automat mit Stack-Speicher (Stapel oder Keller) ⇒ Kellerautomat (Kontextfreie Sprachen)
- Automat mit zwei Stacks oder äquivalent Turing-Maschine kann alles aufführen, was man intuitiv für berechenbar hält

Markov Modelle: endliche Automaten mit probabilistischen Übergangen. Bisher: Algorithmen für einen bestimmten Zweck (Problemklasse) Frage: Gibt es einen universellen Algorithms für alle berechenbare Probleme? Betrache formale Algorithmusbeschreibung als Teil der Eingabe des universellen Algorithmus.

5 Substitutionsmodell (funktionale Programmierung)

- einfaches Modell für arithmetische Berechnung "Taschenrechner"
- Eingaben und Ausgaben sind Zahlen (ganze oder reelle Zahlen). Zahlenkonstanten heißten "Literale"
- elementare Funktionen: haben eine oder mehere Zahlen als Argumente (Parameter) und liefern eine Zahl als Ergebnis (wie Mathematik):
  - add(1,2) → 3, mul(2,3) → 6, analog sub(), div(), mod()
- Funktionsaufrufe können verschachtelt werden, das heißt Argumente kann Ergebnis einer anderen Funktion sein
  - mul(add(1,2),sub(5,3)) → 6

5.1 Substitutionsmodell

Man kann einen Funktionsaufruf, dessen Argument bekannt ist (das heißt Zahlen sind) durch den Wert des Ergebnisses ersetzen ("substituieren"). Geschachtelte Ausdrücke lassen sich so von innen nach außen auswerten.

$$mul(add(1,2), sub(5,3))$$

$$mul(3, sub(5,3))$$

$$mul(3,2)$$

$$6$$

- Die arithmetischen Operationene add(), sub(), mul(), div(), mod() werden normalerweise von der Hardware implementiert.
- Die meisten Programmiersprachen bieten außerdem algebraische Funktionen wie: sqrt(), sin(), cos(), log()
  - sind meist nicht in Hardware, aber vorgefertigte Algorithmen, werden mit Programmiersprachen geliefert, "Standardbibilothek"

- in C++: mathematisches Modul des Standardbibilothek: "cmath"
- Für Arithmetik gebräuchlicher ist "Infix-Notation" mit Operator-Symbolen "+", "-", "*", "/", "%"
- mul(add(1,2),sub(5,3)) ⇔ ((1+2)*(5-3))
  - oft besser, unter anderem weil man Klammer weglassen darf
    1. "Punkt vor Strichrechnung" 3+4*5 ⇔ 3+(4*5), mul, div, mod binden stärker als add, sub
    2. Operatoren gleicher Präzedenz werden von links nach rechts ausgeführt (links-assoziativ)
       1+2+3-4+5 ⇔ ((((1+2)+3)-4)+5)
    3. äußere Klammer kann man weglassen (1+2)⇔ 1+2
- Computer wandeln Infix zuerst in Prefix Notation um
  1. weggelassene Klammer wieder einfügen
  2. Operatorensymbol durch Funktionsnamen ersetzen und an Prefix-Position verschieben

$$1 + 2 + 3 * 4/(1 + 5) - 2$$

$$(((1 + 2) + ((3 * 4)/(1 + 5))) - 2)$$

$$sub(add(add(1, 2), div(mul(3, 4), add(1, 5))), 2)$$

$$sub(add(3, div(12, 6)), 2)$$

$$sub(add(3, 2), 2)$$

$$sub(5, 2)$$

$$2$$

5.2 Bäume

- bestehen aus Knoten und Kanten (Kreise und Pfeile)
- Kanten verbinden Knoten mit ihren Kind-knoten
- jeder Koten (außer der Wurzel) hat genau ein Elternteil ("parent node")
- Knoten ohne Kinder heißen Blätter ("leaves / leaf node")
- Teilbaum
  - wähle beliebigen Knoten
  - entferne temporär dessen Elternkante, dadurch wird der Knoten temporär zu einer Wurzel, dieser Knoten mit allen Nachkommen bildet wieder einen Baum (Teilbaum des Orginalbaumes)

· trivialer Teilbaum hat nur einen Knoten

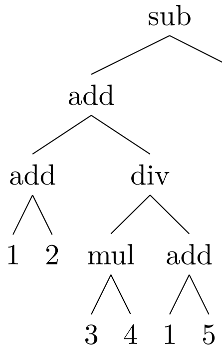
· Tiefe: Abstand eines Knotens von der Wurzel (Anzahl der Kanten zwischen Knoten und Wurzel)

  · Tiefe des Baums: maximale Tiefe eines Knoten

5.2.1 Beispiel

$$1 + 2 + 3 * 4/(1 + 5) - 2$$

$$sub(add(add(1, 2), div(mul(3, 4), add(1, 5))), 2)$$



5.3 Rekursion

Rekursiv $\hat{=}$ Algorithmus für Teilproblem von vorn.

5.4 Prefixnotation aus dem Baum rekonstruieren

1. Wenn die Wurzel ein Blatt ist: Drucke die Zahl

2. sonst:

   · Drucke Funktionsnamen

   · Drucke "("

   · Wiederhole den Algorithmus ab 1 für das linke Kind (Teilbaum mit Wurzel = linkes Kind)

   · Drucke ","

   · Wiederhole den Algorithmus ab 1 für das rechte Kind (Teilbaum mit Wurzel = rechtes Kind)

   · Drucke ")"

$\Rightarrow$

$$sub(add(add(1, 2), div(mul(3, 4), add(1, 5))), 2)$$

5.5 Prefixnotation aus dem Baum rekonstruieren

1. Wenn die Wurzel ein Blatt ist: Drucke die Zahl
2. sonst:
   - Drucke Funktionsnamen
   - Drucke "("
   - Wiederhole den Algorithmus ab 1 für das linke Kind (Teilbaum mit Wurzel = linkes Kind)
   - Drucke Operatorsymbol
   - Wiederhole den Algorithmus ab 1 für das rechte Kind (Teilbaum mit Wurzel = rechtes Kind)
   - Drucke ")"

   $\Rightarrow$
$$sub(add(add(1,2), div(mul(3,4), add(1,5))), 2)$$

$\Rightarrow$ **inorder**

5.6 Berechnen des Werts mit Substitutionsmethode

1. Wenn Wurzel dein Blatt gib Zahl zurück
2. sonst:
   - Wiederhole den Algorithmus ab 1 für das linkes Kind (Teilbaum mit Wurzel = rechtes Kind), speichere Ergebnis als "lhs"
   - Wiederhole den Algorithmus ab 1 für das rechte Kind (Teilbaum mit Wurzel = rechtes Kind), speichere Ergebnis als "rhs"
   - berechne funktionsname(lhs,rhs) und gebe das Ergebnis zurück

$\Rightarrow$ **postorder**

6 Maschienensprachen

- optimiert für die Hardware
- Gegensatz: höhere Programmiersprachen (c++)
  - optimiert für Programmierer
- Compiler oder Interpreter übersetzen Hoch- in Maschinensprache

6.1 Umwandlung in Maschinensprache

1. Eingaben und (Zwischen)-Ergebnisse werden in Speicherzellen abgespeichert $\Rightarrow$ jeder Knoten im Baum bekommt eine Speicherzelle
2. Speicherzellen für Eingaben initialisieren
   - Notation: SpZ $\leftarrow$ Wert
3. Rechenoperationen in Reihenfolge des Substitutionsmodell ausführen und in der jeweiligen Speicherzelle speichern
   - Notation: SpZ-Ergebniss $\leftarrow$ fname SpZArg1 SpZArg2
4. alles in Zahlencode umwandeln
   - Funktionsnamen:

Opcode	Wert
init	1
add	2
sub	3
mul	4
div	5

7 Funktionale Programmierung

- bei Maschienensprache werden Zwischenergebnisse in Speicherzellen abgelegt
- das ist auch in der funktionalen Programmierung eine gute Idee
- Speicherzellen werden duch Namen (vom Programmierer vergeben) unterschieden

7.1 Beispiel

Lösen einer quadratischen Gleichung:

$$ax^2 + bx + c = 0$$
$$x^2 - 2px + q = 0, p = -\frac{b}{2a}, q = \frac{c}{d}$$
$$x_2 = p + \sqrt{p^2 - q}, x_2 = p - \sqrt{p^2 - q}$$

ohne Zwischenergebnisse:

$$x_1 \leftarrow add(div(div(b, a), -2), sqrt(sub(mul(div(b, a), -2), div(div(b, a) - 1)), div(c, a)))$$

mit Zwischenergebniss und Infix Notation

$$p \leftarrow b/c/ - 2 \text{ oder } p \leftarrow -0.5 * b/a$$

$$a \leftarrow c/a$$
$$d \leftarrow sqrt(p * p - q)$$
$$x_1 \leftarrow p + d$$
$$x_2 \leftarrow p - d$$

7.2 Vorteile von Zwischenergebnissen

1. lesbarer
2. redundante Berechnung vermieden. Beachte: In der funktionalen Programmierung können die Speicherzellen nach der Initialisierung nicht mehr verändert werden
3. Speicherzellen und Namen sind nützlich um Argumente an Funktionen zu übergeben $\Rightarrow$ Definition eigener Funktionen

   ```
   function sq(x) {
      return x * x
   }
   ```

   $\Rightarrow d \leftarrow sqrt(sq(p) - q)$ Speicherzelle mit Namen "x" für das Argument von sq

7.3 Funktionale Programmierung in c++

- in c++ hat jede Speicherzelle einen Typ (legt Größe und Bedeutung der Speicherzelle fest)
  - wichtige Typen

int	ganze Zahlen
double	reelle Zahlen
std::string	Text

    int: $12, -3$
    double: $-1.02, 1.2e - 4 = 1.2 * 10^{-4}$
    std::string: "text"
- Initialisierung wird geschrieben als "typename spzname = Wert;"

  ```
  double a = ...;
  double b = ...;
  double c = ...;
  double p = -0.5 b / a;
  double q = c / a;
  double d = std::sqrt(p*p - q);
  double x1 = p + d;
  double x2 = p - d;
  std::cout << "x1: " << x1 << ", x2: " << x2 << std::endl;
  ```
- eigene Funktionen in C++

```
// Kommentar (auch /* */)
type_ergebnis fname(type_arg1 name1, ...) {
        // Signatur / Funkitonskopf / Deklaration
        return ergebnis;
        /* Funktionskörper / Definition / Implementation */
}
```

- ganze Zahl quadrieren:

  ```
  int sq(int x) {
          return x*x;
  }
  ```

- reelle Zahl quadrieren:

  ```
  double sq(double x) {
          return x*x;
  }
  ```

- beide Varianten dürfen in c++ gleichzeitig definiert sein $\Rightarrow$ "function overloading" $\Rightarrow$ c++ wählt automatisch die richtig Variable anhand des Argumenttypes ("overload resolution")

  ```
  int x = 2;
  double y = 1.1
  int x2 = sq(x) // int Variante
  double y2 = sq(y) // double Variante
  ```

- jedes c++-Programm muss genau eine Funktion names "main" haben. Dort beginnt die Programmausführung.

  ```
  int main() {
          Code;
          return 0;
  }
  ```

  - return aus der "main" Funktion ist optional

- Regel von c++ für erlaubte Name
  - erstes Zeichen: Klein- oder Großbuchstaben des englischen Alphabets, oder "_"
  - optional: weitere Zeichen oder, "_" oder Ziffer 0-9
- vordefinierte Funktionen:
  - eingebaute $\hat{=}$ immer vorhanden
    - Infix-Operatoren $+, -, *, /, \%$
    - Prefix-Operatoren $operator+, operator-, \ldots$
  - Funktion der Standardbibilothek $\hat{=}$ müssen "angefordert" werden
    - Namen beginnen mit "std::", "std::sin,..."

- sind in Module geordnet, zum Beispiel
  - cmath ⇒ algebraische Funktion
  - complex ⇒ komplexe Zahlen
  - string ⇒ Zeichenkettenverarbeitung
- um ein Modul zu benutzen muss man zuerst (am Anfang des Programms) sein Inhaltsverzeichnis importieren (Header includieren) → #include <name>

```
#include <iostream>
#include <string>
int main() {
        std::cout << "Hello, world!" << std::endl;
        std::string out = "mein erstes Programm\n";
        std::cout << out;
        return 0;
}
```

- overloading der arithmetischen Operationene
  - overloading genau wie bei *sq*
    - 3 * 4 ⇒ int Variante
    - 3.0 * 4.0 ⇒ double Variante
    - 3 * 4.0 ⇒ automatische Umwandliung in höheren Typ, hier ”double” ⇒ wird als 3.0 * 4.0 ausgeführt
- ⇒ Devision unterscheidet sich
  - Integer-Division: 12 / 5 = 2 (wird abgerundet)
  - Double-Division: 12.0 / 5.0 = 2.4
  - -12 / 5 = 2 (⇒ truncated Division)
  - 12.0 / 5.0 = 2.4
  - Gegensatz (zum Beispiel in Python)
    - floor division ⇒ wird immer abgerundet ⇒ -12 / 4 = -2