Einführung in die Programmierung für Studierende der Naturwissenschaften

Skript zur Vorlesung von

Prof. Dr. Sören Bartels

- Grundlagenwissen
- Elementares Programmieren in C++
- Funktionsweise des Prozessors
- Felder, Zeiger und abgeleitete Datentypen
- Algorithmik
- Visualisieren und Programmieren in Matlab
- Funktionsweise eines Compilers
- Paralleles und objektorientiertes Programmieren
- Aspekte der IT-Kommunikation
- Literaturhinweise

Mathematisches Institut
Fakultät für Mathematik und Physik
Albert-Ludwigs-Universität Freiburg

Sommersemester 2018

I GRUNDLAGENWISSEN

S. BARTELS, 17.4.2018

1. Abstrakte von-Neumman-Rechner

Um die Funktionsweise von Computern, die im Folgenden auch als Rechner bezeichnet werden, zu verstehen, ist es sinnvoll, sie auf ihre wesentlichen Bestandteile zu reduziere. Das Modell des *von-Neumann-Rechners* beschreibt einen Computer durch die Komponenten *Prozessor*, *Hauptspeicher* und *Ein- und Ausgabeeinheiten* sowie einen *Datenbus*, die in Abbildung 1 skizziert sind.

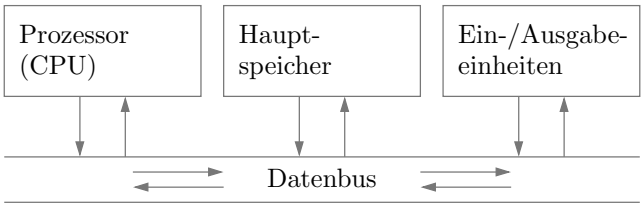

Abbildung 1. Aufbau des von-Neumann-Rechners.

Herzstück eines von-Neumann-Rechners ist der Prozessor, der auch als *Central Processing Unit (CPU)* bezeichnet wird und sämtliche Rechnungen durchführt. Im Hauptspeicher werden Programme und Daten abgelegt. Ein- und Ausgabeeinheiten wie Tastatur, Monitor, Drucker, Maus und weitere Speichermedien dienen der Interaktion mit dem Benutzer. Der Datenbus sorgt für den Transfer von Informationen zwischen den Komponenten.

2. Rechnen mit logischen Ausdrücken

Sämtliche Daten auf Computern liegen im Binärformat vor, das heißt sie werden geeignet durch die Informationen 0 und 1 beziehungsweise entsprechende elektrische Signale realisiert. Eine einzelne solche 0/1-Information bezeichnet man als *Bit*. Auch das eigentliche Rechnen, also die Durchführung arithmetischer Operationen, erfolgt mit Hilfe von Bits. Dabei nutzt man eine Kombination der folgenden drei Varianten des Arbeitens mit den Werten 0 und 1:

(i) Repräsentation von Dezimalzahlen im Dual- bzw. Binärsystem
(ii) Darstellung arithmetischer Operationen durch logische Ausdrücke
(iii) Technische Umsetzung mit Transistorschaltungen

Für die Herleitung geeigneter logischer Ausdrücke werden die Gesetze der *Booleschen Algebra* genutzt. Dabei stehen die Werte 0 und 1 für die Wahrheit einer Aussage und repräsentieren die Bewertungen falsch und wahr. Beispiele für Aussagen sind *"heute scheint die Sonne"* und *"morgen ist Dienstag"*, die abhängig vom Ort und Zeitpunkt ihrer Auswertung falsch oder wahr sein können. Besonders interessant sind Verknüpfungen von Aussagen. Sind A und B zwei Aussagen, so ist beispielweise die UND-Verknüpfung $A \wedge B$ eine Aussage, die dann und nur dann wahr ist, wenn A und B gleichzeitig wahr sind. Weitere sind das nicht-ausschließende ODER, in Zeichen $A \vee B$, sowie die Implikation $A \implies B$. Die in Tabelle 1 dargestellte *Wahrheitstabelle* gibt an, wie diese Verknüpfungen definiert sind.

A	B	$A \wedge B$	$A \vee B$	$A \implies B$	$\neg A$
0	0	0	0	1	1
0	1	0	1	1	1
1	0	0	1	0	0
1	1	1	1	1	0

TABELLE 1. Wertetabelle zur Definition der logischen Verknüpfungen UND, ODER, IMPLIZIERT und NEGATION. Für verschiedene Belegungen der booleschen Variablen A und B sind die Ausdrücke $A \wedge B$, $A \vee B$, $A \implies B$ und $\neg A$ entweder falsch oder wahr beziehungsweise haben den Wert 0 oder 1.

Während die ODER-Verknüpfung gut nachvollziehbar ist, da sie genau dann wahr ist, wenn mindestens eine der beiden Aussagen wahr ist, ist die Festlegung der Implikation zunächst überraschend, insbesondere, dass aus einer falschen Aussage eine wahre folgen darf. Entscheidend für die Gültigkeit der Implikation $A \implies B$ ist aber, dass B immer dann wahr ist, wenn A wahr ist. Dies macht man sich am besten an einfachen Aussagen wie $A =$ *"es regnet"* und $B =$ *"es ist bewölkt"* und der Implikation $(A \implies B) =$ *"wenn es regnet, dann ist es bewölkt"* klar.
Die Verknüpfungen $\wedge$ und $\vee$ erfüllen Kommutativ-, Assoziativ- und Distributivgesetze sowie unter Hinzunahme der Negation, die durch $\neg A$ dargestellt wird, die so genannten *DeMorganschen Gesetze*

$$\neg(A \wedge B) = \neg A \vee \neg B, \quad \neg(A \vee B) = \neg A \wedge \neg B.$$

Dabei bedeutet Gleichheit zweier Ausdrücke, dass sie für jede Belegung der beteiligten Variablen denselben Wahrheitswert ergeben. Mit den Konstanten 0 und 1 bezeichnet man in Ausdrücken die Aussagen, die stets falsch beziehungsweise wahr sind. Mit ihnen gilt beispielsweise $A \wedge 1 = A$ und $A \vee 0 = A$ sowie $A \vee \neg A = 1$.

3. Programmiersprachen und Algorithmen

Da Prozessoren nur Folgen von Nullen und Einsen verarbeiten können, ist die direkte Programmierung eines Prozessors kaum praktikabel. Programmiersprachen erlauben die automatisierte Übersetzung von verständlichen Kommandos in Maschinenbefehle. Eine Folge von Kommandos einer Programmiersprache bezeichnet man als *Programm*. Damit dies übersetzt werden kann, muss es gewissen *syntaktischen* und *semantischen* Bedingungen genügen. Als Syntax einer Programmiersprache bezeichnet man dabei gewissermaßen die durch sie vorgegebene Rechtschreibung und Grammatik, während unter Semantik die inhaltlich korrekte Verwendung von Kommandos bezeichnet wird. Im Kontext der deutschen Sprache ist beispielsweise die Aussage *"die Straße ist eckig"* syntaktisch korrekt nicht jedoch semantisch, wohingegen die Aussage *"die Straße ist breit"* unabhängig vom Wahrheitsgehalt beiden Anforderungen genügt.
Programme sind Realisierungen von *Algorithmen*. Als Algorithmus bezeichnen wir dabei eine Folge von elementaren Operationen ähnlich einem Kochrezept. Wie detailliert die einzelnen Schritte angegeben werden, hängt sehr vom Kontext ab und kann stark variieren. Die pq-Formel zur Lösung einer quadratischen Gleichung $x^2 + px + q = 0$ also

$$x_{1,2} = -p/2 \pm \left(p^2/4 - q\right)^{1/2}$$

kann für sich schon als Algorithmus angesehen werden, allerdings lässt sie sich in dieser Form nicht direkt in einer Programmiersprache angeben. Etwas konkreter wäre folgende Form.

Algorithmus 3.1 (pq-Formel)**.** Eingabe: *Reelle Zahlen p und q.*

(1) *Setze $r = p^2/4 - q$.*
(2) *Ist $r < 0$, so generiere Fehlermeldung* "nicht lösbar" *und stoppe.*
(3) *Berechne $s = \sqrt{r}$.*
(4) *Definiere $x_1 = -p/2 + s$ und $x_2 = -p/2 - s$.*

Ausgabe: *Lösungen x_1 und x_2.*

Auch dieser Algorithmus ist nicht unbedingt direkt umsetzbar, da zum Beispiel die Quadratwurzel geeignet berechnet werden muss. Dies kann effizient mit dem *Heronschen Verfahren* geschehen, der Approximationen von $\sqrt{r}$ mittels der Initialisierung $s_0 = 1$ und der Iteration $s_{k+1} = (s_k + r/s_k)/2$ berechnet. Diese Vorschrift wird so lange angewendet, bis ein Abbruchkriterium erfüllt ist.

Algorithmus 3.2 (Heronsches Verfahren)**.** Eingabe: *Reelle Zahl $r \geq 0$ und Toleranz $\delta > 0$.*

(1) *Definiere $s_{neu} = 1$ und $s_{alt} = r$.*
(2) *Gilt $|s_{neu} - s_{alt}| < \delta$, so stoppe.*
(3) *Setze $s_{alt} = s_{neu}$ und anschließend $s_{neu} = (s_{alt} + r/s_{alt})/2$.*
(4) *Fahre fort mit Schritt (2).*

Ausgabe: *Approximation* s_{neu} *von* $s = \sqrt{r}$.

In diesem Algorithmus wird dieselbe Anweisung in Schritt (3) so lange wiederholt, bis eine Abbruchbedingung erfüllt ist. Man könnte alternativ auch die Iterationsvorschrift N-mal anwenden und s_N als Approximation verwenden, allerdings ist a priori nicht klar, was eine hinreichend große Zahl N ist.

4. Wissenschaftliche Fragestellungen

Das wissenschaftliche Fachgebiet der Informatik ist mit der Untersuchung und Verwendung von Rechnern befasst und lässt sich in die Bereiche der technischen, praktischen und theoretischen Informatik untergliedern, deren Arbeitsbereiche sich auszugshaft folgendermaßen beschreiben lassen:
Technische Informatik: Die technische Informatik ist zum Beispiel mit der Entwicklung von Prozessoren und Speichermedien und allgemeiner mit der sogenannten *Hardware* befasst.
Praktische Informatik: Gegenstand der praktischen Informatik ist beispielsweise die Konzeption von Algorithmen und deren Umsetzung in Programme also die Entwicklung von *Software*.
Theoretische Informatik: Die theoretische Informatik untersucht unter anderem, ob gewisse Problemstellungen berechen- beziehungsweise entscheidbar sind, und entwickelt Kalküle für die Verifizierung der Korrektheit von Programmen.

I GRUNDLAGENWISSEN – ZUSATZ

S. BARTELS, 31.7.2018

1. Grafische Darstellung von Algorithmen

Sehr hilfreich für das Verständnis von Algorithmen und deren Umsetzung auf Computern ist deren grafische Darstellung. Für die Diskussion gängiger Konzepte folgen wir in diesem Abschnitt dem Buch *Algorithmen kompakt und verständlich* von M. von Rimscha (Springer, 2008) Zur übersichtlichen Repräsentation stehen Struktogramme und Ablaufpläne zur Verfügung, die die folgenden vier Elemente symbolisch darstellen:

- Ein *Block* besteht aus einer oder mehreren Anweisungen wie Ein- und Ausgabe oder Wertzuweisungen mittels elementarer Formeln.
- Eine *Fallunterscheidung* wählt abhängig von einer Bedingung einen Block aus.
- Eine *Wiederholung* führt einen Block wiederholt aus, bis eine Bedingung verletzt ist.
- Eine *Unterfunktion* steht für einen weiteren Algorithmus, der für gewisse Eingaben einen oder mehrere Rückgabewerte liefert.

Diese Elemente definieren wiederum Anweisungen, die in geschachtelter Form in Blöcken verwendet werden können. Beispielsweise kann eine Fallunterscheidung im Block einer Wiederholung auftreten. Die Symbole der Elemente zur Erstellung eines Struktogramms und eines Ablaufplans sind in den Abbildungen 1 und 2 gezeigt. Für Ablaufpläne werden zusätzlich Pfeile sowie Anfangs- und Endsymbole verwendet.

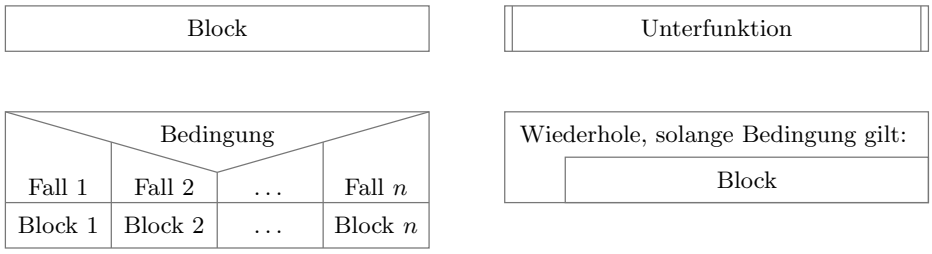

Abbildung 1. Elemente der grafischen Darstellung eines Algorithmus mit einem Struktogramm.

Wir illustrieren die Verwendung eines Struktogramms und eines Ablaufplans am Beispiel des Euklidischen Algorithmus zur Bestimmung des größten gemeinsamen Teilers zweier ganzer Zahlen, der auf der Tatsache beruht, dass

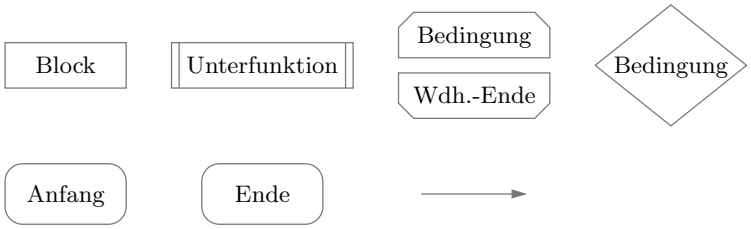

ABBILDUNG 2. Elemente der grafischen Darstellung eines Algorithmus mit einem Ablaufplan.

die Differenz der Zahlen ebenfalls von der gesuchten Zahl geteilt wird, das heißt es gilt $\mathrm{ggT}(a,b) = \mathrm{ggT}(a-b,b)$. Im Verfahren wird eine Wechselwegnahme durchgeführt, bis die reduzierten Zahlen identisch sind und dann mit der gesuchten Zahl übereinstimmen.

Algorithmus 1.1 (Euklidischer Algorithmus)**.** Eingabe: *Ganze Zahlen* a, b*.*

(1) *Solange* $a \neq b$ *gilt, wiederhole Schritt (2).*
(2) *Gilt* $a > b$*, so ersetze* a *durch* $a-b$*, andernfalls ersetze* b *durch* $b-a$*.*
(3) *Definiere* $g = a$*.*

Ausgabe: *Größter gemeinsamer Teiler* g *von* a *und* b*.*

Das Verfahren ist im folgenden Beispiel illustriert.

Beispiel 1.2. *Es gilt* $\mathrm{ggT}(240,100) = 20$ *und* $\mathrm{ggT}(21,15) = 3$*:*

240	100	21	15
140	100	6	15
40	100	6	9
40	60	6	3
40	20	3	3
20	20		

Abbildung 3 zeigt ein zugehöriges Struktogramm. Dabei werden die einzelnen Elemente direkt untereinander aufgeführt.

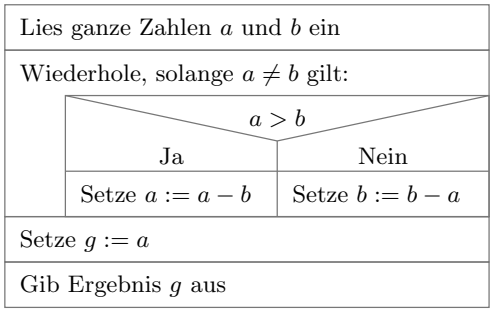

ABBILDUNG 3. Struktogramm zum Euklidischen Algorithmus.

In Abbildung 4 ist die Verwendung der Elemente und darstellenden Symbole eines Ablaufplans für den Euklidischen Algorithmus gezeigt. Elemente und Fallunterscheidungen werden mit Pfeilen verbunden und dargestellt.

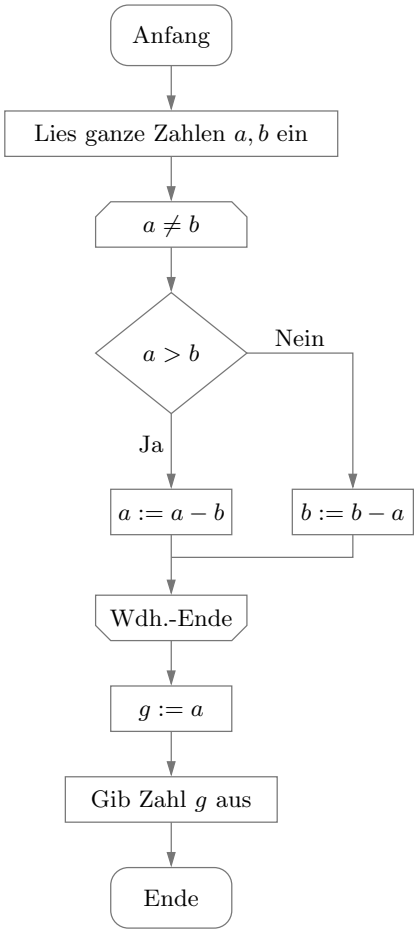

ABBILDUNG 4. Ablaufplan zum Euklidischen Algorithmus.

II ELEMENTARES PROGRAMMIEREN IN C++

S. BARTELS, 26.3.2018

1. Aufbau eines Programms

Wir diskutieren typische Aspekte eines Computerprogramms am Beispiel der Programmiersprache C++, jedoch finden sich die nachfolgenden Grundkonzepte in jeder Programmiersprache wieder. Der Aufbau eines Programms in C++ hat stets die in Abbildung 1 gezeigte Struktur.

```
1 // Kommentar
2 #include <...>
3 type funktions_name(...){...}
4 int main(){
5   type variable;
6   anweisung;
7 }
```

Abbildung 1. Struktureller Aufbau eines C++-Programms.

Zeile 1 Es ist meist hilfreich ein Programm mit einem Kommentar zu beginnen, der zum Beispiel die Funktionsweise des Programms erläutert.

Zeile 2 Mit der Anweisung `#include<...>` werden Bibliotheken, die beispielsweise Befehle für Ein- und Ausgabe enthalten, eingebunden.

Zeile 3 Eine Unterfunktion wird durch Festlegung des Rückgabetyps, des Namens, der Argumente in runden Klammern sowie die in geschweiften Klammern aufgeführten konkreten Berechnungen definiert.

Zeile 4 Der Hauptteil des Programms ist eine Funktion mit dem Namen `main`. Die öffnende geschweifte Klammer markiert den Beginn der Anweisungen des Hauptprogramms. Der Rückgabewert ist hier als ganzzahlig festgelegt, kann aber auch weggelassen werden.

Zeile 5 In der Regel werden zunächst Variablen deklariert und häufig auch initialisiert, das heißt ihnen wird direkt ein Wert zugeordnet.

Zeile 6 Es folgen weitere Anweisungen wie Wertzuweisungen für Variablen, Funktionsaufrufe mit Variablen, Fallunterscheidungen, Kommentare, Schleifen oder weitere Variablendefinitionen.

Zeile 7 Die schließende geschweifte Klammer markiert das Ende der Berechnungen des Hauptprogramms.

2. Beispielprogramme

Wir betrachten einige Beispielprogramme und diskutieren anschließend die Funktionsweisen der verwendeten Befehle. Die Programme werden unter UNIX mit dem Kommando

```
g++ progname.cc -o progname.out
```

kompiliert beziehungsweise in Maschinensprache übersetzt und mittels

```
./progname.out
```

ausgeführt. In Browser-Compilern wie *C++ Shell*, der unter `http://cpp.sh/` verfügbar ist, können Programme direkt durch Wahl der Schaltfläche *Run* kompiliert und gestartet werden.

2.1. **Textausgabe.** Das in Abbildung 2 gezeigte Programm produziert nach Übersetzung und Ausführung die Ausgabe `etwas Text ...` mit anschließendem Zeilenumbruch auf dem Bildschirm. Nach einem Kommentar zum Kompilieren in der ersten Zeile folgt in der zweiten die Einbindung einer Bibliothek zur Ein- und Ausgabe von Informationen. Das Hauptprogramm besteht nur aus dem in Zeile 4 aufgeführten Druckbefehl.

```
1 // kompilieren: g++ ausgabe.cc -o ausgabe.out
2 #include <iostream>
3 int main(){
4   std::cout << "etwas Text ... " << std::endl;
5 }
```

Abbildung 2. Ausgabe von Text auf dem Bildschirm.

2.2. **Variablen.** Eine simple Addition von drei Variablen, die ganzzahlige Werte enthalten dürfen, ist in Abbildung 3 gezeigt. Bevor den Variablen Werte zugeordnet werden können, müssen sie deklariert werden. Dies kann direkt in einer Anweisung geschehen.

```
1 // kompilieren: g++ addition.cc -o addition.out
2 #include <iostream>
3 int main(){
4   int j, k, sum;
5   int m = 5;
6   j = 3; k = -4;
7   sum = j+k+m;
8   std::cout << "Summe von j, k und m ist " << sum << std::endl;
9 }
```

Abbildung 3. Addition ganzzahliger Variablen.

2.3. **Fallunterscheidungen.** Ein zentraler Bestandteil von Programmen sind Fallunterscheidungen, die abhängig von der Wahrheit einer oder mehrerer Aussagen unterschiedliche, in geschweiften Klammern eingefasste Programmteile ausführen. Das in Abbildung 4 gezeigte Programm prüft, ob eine eingegebene Zahl negativ ist.

```
1  // kompilieren: g++ faelle.cc -o faelle.out
2  #include <iostream>
3  int main(){
4    double x;
5    std::cout << "(Dezimal-) Zahl eingeben: ";
6    std::cin >> x;
7    if (x<0){
8      std::cout << "Die Zahl ist negativ" << std::endl;
9    }
10   else if (x>0){
11     std::cout << "Die Zahl ist positiv" << std::endl;
12   }
13   else{
14     std::cout << "Die Zahl ist Null" << std::endl;
15   }
16 }
```

ABBILDUNG 4. Fallunterscheidung zum Test auf Negativität.

2.4. **Aufzählungen.** Computer eignen sich sehr gut für simple Zählaufgaben, wie zum Beispiel das simple Zählen der Tage zwischen zwei Daten oder die Bildung einer großen Summe $1+2+3+\cdots+J$ für eine Zahl $J \geq 1$. Für solche Zwecke sind sogenannte for-Schleifen geeignet, deren Verwendung in Abbildung 5 gezeigt ist.

```
1  // kompilieren: g++ aufzaehlung.cc -o aufzaehlung.out
2  #include <iostream>
3  int main(){
4    int summe = 0, J;
5    std::cout << "Zahl J eingeben: "; std::cin >> J;
6    for (int j=1; j<=J; j=j+1){
7      summe = summe+j;
8    }
9    std::cout << "Die Summe der Zahlen von 1 bis J ist ";
10   std::cout << summe << std::endl;
11 }
```

ABBILDUNG 5. Verwendung einer for-Schleife zur Berechnung einer Summe.

2.5. **Wiederholungen.** Ein weiteres wichtiges Konzept ist das der Wiederholung gewisser Befehle, bis eine vorgegebene Bedingung erfüllt ist. Das in Abbildung 6 gezeigte Programm summiert so lange die natürlichen Zahlen $1, 2, 3, \ldots$, bis die Summer größer als eine vorgegebene Schranke ist.

```
// compile: g++ wiederholung.cc -o wiederholung.out
#include <iostream>
int main(){
  int summe = 0, j = 0, L, J;
  std::cout << "Summenschranke L: "; std::cin >> L;
  while (summe <= L){
    j = j+1;
    summe = summe+j;
  }
  J = j;
  std::cout << "Ueberschreiten der Schranke L durch";
  std::cout << "Summe 1+2+...+J fuer J = " << J << std::endl;
}
```

ABBILDUNG 6. Wiederholte Addition von Zahlen bis zur Überschreitung einer Schranke.

2.6. **Unterfunktionen.** Wenn bestimmte Rechenschritte oder Operationen in gleicher Weise an unterschiedlichen Stellen eines Programms benötigt werden, ist es sinnvoll eine (Unter-) Funktion zu definieren. Eine Funktion, die den Mittelwert zweier Dezimalzahlen zurück gibt, ist in Abbildung 7 gezeigt. Man beachte, dass der Wert von `x` im Hauptprogramm unverändert bleibt, selbst wenn die Unterfunktion die Variable modifizieren würde beispielsweise mit einer Zuweisung `x = 1.0;` oder einer anderen Anweisung.

```
// compile: g++ unterfunktion.cc -o unterfunktion.out
#include <iostream>
double mittelwert(double x, double y){
  double mittel;
  mittel = (x+y)/2.0;
  return mittel;
}
int main(){
  double x = 10.1, y = 22.3, z;
  z = mittelwert(x,y);
  std::cout << "Mittelwert ist " << z << std::endl;
}
```

ABBILDUNG 7. Unterfunktion zur Berechnung des Mittelwerts zweier Dezimalzahlen.

2.7. **Grundfunktionen.** Durch Einbinden der Bibliothek `cmath` lassen sich mathematische Grundfunktionen wie trigonometrische Funktionen und Potenzfunktionen verwenden. Abbildung 8 zeigt beispielhaft die praktische Realisierung

```
1 // kompilieren: g++ grundfunktionen.cc -o grundfunktionen.out
2 #include <iostream>
3 #include <cmath>
4 int main(){
5   double x = 3.5, y;
6   y = sin(1.0*M_PI)+exp(x)+pow(x,2.0)+fabs(-5.0)+sqrt(2.0);
7   std::cout << "y ist " << y << std::endl;
8 }
```

Abbildung 8. Verwendung mathematischer Grundfunktionen.

3. Bestandteile einer Programmiersprache

Programmiersprachen stellen Kommandos, auch als Befehle oder Anweisungen bezeichnet, zur Bearbeitung von Daten zur Verfügung, die sich in drei Kategorien einteilen lassen: Variablen, Kontrollstrukturen und Funktionen.

3.1. **Variablen.** Variablen dienen der Speicherung von Daten und ihre Verwendung ähnelt der in der Mathematik üblichen. Um effizient mit Speicher umzugehen und eine sinnvolle Verwendung der Variablen in Funktionen zu garantieren, muss der *Typ* einer Variable deklariert werden. Eine boolesche Variable benötigt zum Beispiel weniger Speicherplatz als eine Variable zur Speicherung einer Dezimalzahl. Variablen können im Laufe eines Programms verschiedene Werte zugeordnet werden. Programmiersprachen stellen Operationen wie arithmetische Ausdrücke und Vergleiche für unterschiedliche Typen zur Verfügung.

3.2. **Kontrollstrukturen.** Algorithmen enthalten häufig Fallunterscheidungen und Wiederholungen, die auch als *Selektoren* und *Iteratoren* bezeichnet werden. Beispielsweise soll ein Teil eines Algorithmus beziehungsweise eines Programms nur dann ausgeführt werden, wenn eine gewisse Bedingung erfüllt ist, oder eine Anweisung soll so lange wiederholt werden, bis eine vorgegebene Bedingung erfüllt ist. Kontrollstrukturen können geschachtelt verwendet werden.

3.3. **Funktionen.** Programmiersprachen stellen viele wichtige Operationen wie das Testen auf Gleichheit zweier Variablen oder die Berechnung der Quadratwurzel nichtnegativer Dezimalzahlen zur Verfügung. Weitere Funktionen können meist selbst definiert werden und ihre Verwendung entspricht wiederum sehr der in der Mathematik üblichen. Eine Funktion hat gewisse

Eingabeparameter, die auch als Argumente bezeichnet werden, und Ausgabeparameter, die dem oder den Werten der Funktion entsprechen. Beim Test auf Gleichheit sind die Eingabeparameter zwei Variablen gleichen Typs und der Ausgabeparameter eine boolesche Variable. Unterfunktionen können eigene Variablen enthalten, deren Wertzuweisung nur lokal, das heißt nicht für das Hauptprogramm, Auswirkungen haben.

4. Ausdrücke, Zuweisungen und Blöcke

Um mit Variablen arbeiten und sie in Kontrollstrukturen und Funktionen sinnvoll verwenden zu können, werden Ausdrücke, Zuweisungen und Blöcke benötigt.

4.1. **Ausdrücke.** Ein *Ausdruck* ist eine zulässige Formel, die Operationen mit Variablen und Konstanten enthält und deren Auswertung bei gültiger Belegung der Variablen zu einem Wert eines bestimmten Typs führt. Abstrakt können Ausdrücke induktiv definiert werden: konstante Werte und Variablen sind Ausdrücke und durch zulässige Verknüpfungen mittels Operationen eines oder mehrerer Ausdrücke entsteht ein neuer Ausdruck. Die Operationen können arithmetische Ausdrücke oder Unterfunktionen und die Variablen Dezimalzahlen sein wie beispielsweise in dem folgenden Ausdruck:

```
y+2.0*(x-sin(z))
```

Hier sollten `x,y,z` Dezimalzahlen sein, damit alle Operationen und der Ausdruck als Dezimalzahl wohldefiniert sind. Häufig verwendet werden auch Ausdrücke, deren Wert bei gültiger Auswertung ein boolescher Wert ist:

```
((j>5) && (j<10)) || (!b)
```

Hier ist `j` eine ganzzahlige und `b` eine boolesche Variable und der Ausdruck hat den Wert `true`, sofern der Wert `j` echt zwischen 5 und 10 liegt oder die boolesche Variable den Wert `false` oder `0` besitzt.

4.2. **Zuweisungen.** Variablen können Werte *zugewiesen* werden, die ihrem Typ entsprechen. Die Werte sind konstante oder variable Ausdrücke wie beispielsweise:

```
j = 5+6;
x = sin(1.0);
b = (j<10) || !(j>20);
```

Wird einer Variablen eines bestimmten Typs ein Wert eines anderen Typs zugeordnet, so kann eine Fehlermeldung entstehen, beispielsweise in folgendem Fall:

```
unsigned int j;
j = -3;
```

In C++ wird bei eigentlich unpassender Zuweisung versucht, eine Typenkonvertierung durchzuführen, das heißt beispielsweise, dass eine Dezimalzahl bei der Zuweisung automatisch auf eine ganze Zahl ab- oder aufgerundert wird.

4.3. **Blöcke.** Unter einem *Block* versteht man eine Folge von Anweisungen, die in geschweiften Klammern eingefasst sind. Die Anweisungen können dabei Variablendefinitionen, Wertzuweisungen oder Kontrollstrukturen sein. Blöcke definieren Unterfunktionen oder auch die Hauptfunktion eines Programms in C++. Ein Beispiel eines Blocks ist:

```
{int j = 0, k; bool b;
 k = 2*j; b = (j<k);}
```

Blöcke treten auch in Fallunterscheidungen und Wiederholungen auf.

5. Syntaktische Konventionen in C++

Es gibt einige generelle Regeln, die in C++ zu beachten sind:

- Variablennamen bestehen aus Buchstaben, Ziffern und Unterstrichen, wobei das erste Zeichen keine Ziffer sein darf und der Variablenname nicht nur aus einem Unterstrich bestehen darf.
- Es wird zwischen Groß- und Kleinbuchstaben unterschieden, das heißt beispielsweise, dass `Hausnr` und `HausNr` zwei unterschiedliche Variablen sind.
- Schlüsselwörter wie `if`, `else`, `while`, `true` oder `false` dürfen nicht als Variablen- oder Funktionsnamen verwendet werden.
- Zuweisungen werden in C++ stets mit einem Semikolon beendet.
- Leerzeilen, Leerzeichen und Zeilenumbrüche haben nur in wenigen speziellen Situation eine Bedeutung und werden andernfalls vom Compiler ignoriert. Ein Zeilenumbruch muss beispielsweise auf eine Compiler-Anweisung wie `#include<...>` folgen und ein Leerzeichen ist bei der Deklaration einer Variablen erforderlich.
- Bei Blöcken, die nur aus einer Anweisung bestehen, können die geschweiften Klammern weggelassen werden.

6. Wichtige Anweisungen in C++

Wir diskutieren nachfolgend die syntaktischen und semantischen Eigenschaften der wichtigsten Anweisungen des elementaren Programmierens in C++.

6.1. **Variablendeklarationen.** Eine Variablendeklaration deklariert eine Variable und besteht aus der Angabe des Typs der Variablen wie `int`, `double` oder `bool` und dem Variablennamen. Mehrere Variablen gleichen Typs können gemeinsam deklariert werden. Einer Variablen kann bei der Deklaration ein Wert zugeordnet werden:

```
type var;
type var = value;
```

6.2. **Wertzuweisungen.** Eine Wertzuweisung für eine Variable erfolgt durch Angabe der Variablennamen gefolgt von einem Gleichheitszeichen und einem gültigen Ausdruck, der bei seiner Auswertung einen Wert des Typs der Variable liefert:

```
var = expression;
```

6.3. **Fallunterscheidungen.** Eine Fallunterscheidung besteht mindestens aus dem Schlüsselwort `if`, einer booleschen Variablen `b` und einem Block, der ausgeführt wird, wenn `b` als wahr ausgewertet wird. Weitere Alternativfälle können mit oder ohne Bedingung folgen:

```
if (b){...} [else if (c){...} ... else{...}]
```

6.4. **Aufzählungen.** Eine Aufzählung oder for-Schleife besteht aus dem Schlüsselwort `for` einer Initialisierung, einer Abbruchbedingung und einer Iterationsanweisung sowie einem Befehlsblock. Nach der Initialisierung werden der Befehlsblock und die Iterationszuweisung so lange abwechselnd ausgeführt, wie die Abbruchbedingung als falsch ausgewertet wird:

```
for (ini; b_stop; iter){...}
```

6.5. **Wiederholungen.** Eine Wiederholung besteht aus einer Abbruchbedingung und einem Befehlsblock. Die Abbruchbedingung wird wiederholt ausgewertet und nach jeder wahren Auswertung wird der Befehlsblock ausgeführt.

```
while (b){...}
```

6.6. **Unterfunktionen.** Eine Unterfunktion besteht aus einem einfachen Rückgabetyp, dem Funktionsnamen, einer Deklaration von Variablen, die als Funktionsargumente verwendet werden, und einem Befehlsblock, der diese und weitere Variablen verwendet und mit dem Rückgabebefehl `return` den Wert der Berechnungen zurückgibt:

```
type fun(type arg1, ..., type argn){... return expr; ...}
```

7. Namensumgebungen und Inkrementoperatoren

Abschließend diskutieren wir übliche Abkürzungen einiger Befehle, die ein effizientes Programmieren ermöglichen, aber mit gewisser Vorsicht verwendet werden sollten.

7.1. **Namensumgebung.** Die Verwendung der Namensumgebung `std` erlaubt es, die Befehle zur Ein- und Ausgabe aus der Bibliothek `iostream` ohne den Zusatz `std::` zu verwenden. Unabhängig davon kann statt des Zeilenumbruchs mittels `endl` das Steuerzeichen `\n` benutzt werden, siehe Abbildung 9. Etwas veraltet ist das Kommando `fprint` aus der Bibliothek `stdio.h` für die formatierte Ausgabe von Text und Variablen.

7.2. **Inkrementoperatoren.** Sehr häufig wird bei einer Zuweisung der Wert einer Variablen durch Addition zu der Variablen definiert. In diesem Fall kann die Zuweisung folgendermaßen vereinfacht werden:

```
a += expr;   ⟺   a = a+expr;
```

Ist der Ausdruck durch den konstanten Wert 1 gegeben, so können die Zuweisungen weiter vereinfacht werden.

```
1 // kompilieren: g++ namensumgebung.cc -o namensumgebung.out
2 #include <iostream>
3 using namespace std;
4 int main(){
5   cout << "etwas Text ... " << endl;
6   cout << "alternativer Zeilenumbruch \n";
7 }
```

ABBILDUNG 9. Verwendung einer Namensumgebung und eines alternativen Zeilenumbruchkommandos.

$$\texttt{j++} \iff \texttt{j = j+1;} \iff \texttt{j += 1;}$$

Wird der Befehl `j++` in einem Ausdruck wie zum Beispiel `5-j++` verwendet, so wird erst der umfangreichere Ausdruck mit dem Wert von `j` ausgewertet und anschließend der Wert von `j` erhöht. Soll die Veränderung von `j` vorher geschehen, so muss die Operation `++j` benutzt werden. Eine typische Verwendung von Inkrementoperatoren ist in Abbildung 10 gezeigt.

```
1 // kompilieren: g++ inkrement.cc -o inkrement.out
2 #include <iostream>
3 using namespace std;
4 int main(){
5   int summe = 0, J;
6   cout << "Zahl J eingeben: "; cin >> J;
7   for (int j=1; j<=J; j++) summe += j;
8   cout << "Summe von 1 bis J ist " << summe << endl;
9 }
```

ABBILDUNG 10. Verwendung von Inkrementoperatoren in einer Aufzählung zur Berechnung der Summe $S = 1 + 2 + \cdots + J$.

7.3. **Typen in C++.** Die wichtigsten Typen zur Deklaration von Variablen sind `bool`, `int` und `double` für boolesche, ganzzahlige und Dezimalzahlwertige Variablen. Der leere Typ `void` wird verwendet, wenn eine Funktion keinen Rückgabewert hat. Der jeweilige Speicherbedarf und die jeweiligen Operationen sind in Tabelle 1 dargestellt. Der Speicherbedarf beziehungsweise die maximale Größe einer Zahl eines bestimmten Typs ist Compilerabhängig, kann aber mit dem Kommando `sizeof` festgestellt werden, wie in Abbildung 11 gezeigt ist. Für die Operationen gelten gewisse Prioritäten bei der Auswertung wie beispielsweise Punkt- vor Strichrechnung sowie NEGATION vor UND vor ODER. Gerade bei booleschen Ausdrücken sollte man sinnvollen Gebrauch von Klammern machen, die die höchste Priorität

haben. Mit dem Typ `char` können Variablen deklariert werden, deren Werte Buchstaben sind.

Typ	Werte	Speicher	Operationen	Vergleiche
`void`	–	0 Bit	–	–
`bool`	$\{0, 1\}$	1 Bit	`&&,||,!`	`==,!==`
`int`	$[\pm 2147483647]$	4 Bytes	`+,-,*,/,%`	`<,>,==,<=,>=,!==`
`double`	$[\pm 1,7 \cdot 10^{308}]$	8 Bytes	`+,-,*,/`	`<,>,==,<=,>=,!==`

TABELLE 1. Wichtigste Zahlenyypen in C++ und darauf definierte Operationen und logische Vergleiche. Die Symbole ! und % realisieren die logische Negation und die Modulo-Operation (Rest bei Division).

```
1  // kompilieren: g++ typen.cc -o typen.out
2  #include <iostream>
3  #include <limits.h>
4  using namespace std;
5  int main(){
6    cout << sizeof(int) << endl;
7    cout << sizeof(double) << endl;
8    cout << INT_MIN << ", " << INT_MAX << endl;
9    cout << sizeof(char) << endl;
10  }
```

ABBILDUNG 11. Bestimmung des Speicherbedarfs von Typen sowie minimaler und maximaler darstellbarer Zahlen ganzzahligen Typs. Ein Byte entspricht 8 Bits.

7.4. **Typumwandlung.** Semantisch fehlerhafte Wertzuweisungen führen in C++ selten zu Fehlermeldungen, da in den meisten Fällen eine automatische Anpassung des Typs vollzogen wird, das heißt zum Beispiel, dass für Variablen `j` und `x` vom Typ `int` und `double` die Zuweisungen

```
j = 3.74;
x = 1;
```

auf die Belegungen 3 und 1 führen, die Nachkommstellen der Dezimalzahl `3.74` werden also abgeschnitten. In Ausdrücken kann eine ungenaue Typverwendung jedoch zu fehlerhaften Berechnungen führen. Die Zuweisung

```
j = 1;
x = j/2;
```

führen auf die Belegung 0 von `x`, da die Division ganzer Zahlen als Division ohne Rest durchgeführt wird. Verwendet man statt des Werts `2` jedoch `2.0` so ist ist der Divisor als Dezimalzahl deklariert und das Ergebnis der Division ist ebenfalls eine Dezimalzahl, das heißt durch

```
j = 1;
x = j/2.0;
```

erhält man die Belegung 0.5. In Dezimalzahlrechnungen sollte man daher für konstante Werte die Form `1.0` statt `1` verwenden; möglich und gebräuchlich ist auch das Weglassen der Null, das heißt die Verwendung von `1.` statt `1.0`. Allgemein orientieren sich die arithmetischen Operationen immer am höherwertigen Typ der beteiligten Variablen oder Werte. Eine explizite Typumwandlung kann mit den Befehlen

```
j = (int)1.0;
x = (double)1;
```

erfolgen. Ein Beispielprogramm ist in Abbildung 12 gezeigt.

```
1  // kompilieren: g++ typumwandlung.cc -o typumwandlung.out
2  #include <iostream>
3  using namespace std;
4  int main(){
5    int j = 1;
6    double x;
7    x = j/2;
8    cout << x << endl;
9    x = j/2.0;
10   cout << x << endl;
11   j = (int)x;
12   cout << j << endl;
13 }
```

ABBILDUNG 12. Implizite und explizite Umwandlung von Typen in C++.

III FUNKTIONSWEISE DES PROZESSORS

S. BARTELS, 28.5.2018

1. Von der Binärzahl zum Transistor

Computer arbeiten mit dem Binärsystem, das auch als Dualsystem bezeichnet wird, und ihre Prozessoren bestehen aus vielen Transistoren. Wir diskutieren im Folgenden, wie arithmetische Operationen im Binärsystem durchgeführt werden können, wie sich diese durch logische Operationen darstellen lassen und wie sich logische Operationen mit elektrischen Schaltungen realisieren lassen.

2. Binärdarstellung natürlicher Zahlen

Jede natürliche Zahl inklusive Null lässt sich als Summe von Potenzen der Zahl 2 darstellen, das heißt für jedes $\ell \in \mathbb{N}_0$ existieren eine natürliche Zahl $k \in \mathbb{N}$ und Koeffizienten $b_0, b_1, \dots, b_k \in \{0, 1\}$, so dass

$$\ell = b_k \cdot 2^k + \cdots + b_1 \cdot 2^1 + b_0 \cdot 2^0.$$

Schreibt man die Zahlen $b_0, b_1, \dots, b_k$ hintereinander, so erhält man die *Binärdarstellung* von ℓ, beispielsweise

$$13 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \equiv 1101$$

Das Dezimalsystem lässt sich übrigens genau so interpretieren, wenn statt der Basis 2 die Basis 10 verwendet wird. Jede Ziffer der Binärdarstellung bezeichnen wir als *Bit*. Fixieren wir die maximale Anzahl k zulässiger Bits, so können wir natürliche Zahlen im Bereich $0, 1, \dots, 2^k - 1$ darstellen. Die Darstellung von Dezimalzahlen mit vorgegebener Anzahl m von Nachkommastellen lässt sich auf die Darstellung natürlicher Zahlen zurückfüren, wenn wir alle solchen Dezimalzahlen mit 10^m multiplizieren, zur Berücksichtigung negativer Zahlen können wir ein Bit zur Speicherung des Vorzeichens verwenden. Insgesamt lassen sich so arithmetische Operationen mit Dezimalzahlen auf das Rechnen mit Binärzahlen zurückführen.

3. Rechnen im Binärsystem

Die Addition zweier Binärzahlen erfolgt schriftlich mit Übertrag, im Beispiel $13 + 6$ ergibt sich:

$$\begin{array}{rr} & 1101 \\ + & 0110 \\ \hline \text{Ü} & 1100 \\ \hline & 10011 \end{array}$$

Die Subtraktion wird auf die Addition zurückgeführt. Dazu werden Minuend und Subtrahend durch Auffüllen mit Nullen auf gleiche Bitlänge gebracht und das additive Inverse des Subtrahenden wie folgt gebildet. Zunächst werden alle Bits invertiert, das heißt aus $8 \equiv 1000$ wird 0111, und anschließend wird 1 addiert, das heißt aus 0111 wird 1000. Diese Zahl wird zum Minuenden addiert und anschließend das höchste Bit in der Summe gelöscht. Für die Rechnung $14 - 8$ ergibt sich in Binärdarstellung also die Addition:

$$\begin{array}{rr} & 1110 \\ + & 1000 \\ \hline \text{Ü} & 1000 \\ \hline & \cancel{1}0110 \end{array}$$

Zur Multiplikation einer Binärzahl mit einer Potenz der Zahl 2, werden durch Anhängen von Nullen die Bits um die Potenz nach links verschoben. Für die Rechnung $9 \cdot 4$ ergibt sich

$$1001 \cdot 2^2 = 100100$$

Mit Hilfe des Distributivgesetzes kann so die allgemeine Multiplikation von Binärzahlen auf die Addition zurückgeführt werden:

$$1001 \cdot 0101 = 1001 \cdot (2^2 + 2^0) = 100100 + 001001$$

Divisionen durch Potenzen der Zahl 2 können bei Vernachlässigung des Rests durch eine Verschiebung der Bits nach rechts realisiert werden, allerdings lassen sich damit nicht allgemeinere Divisoren realisieren. Die allgemeine Division wird auf eine Folge von Subtraktionen zurückgeführt, indem man zum Beispiel schriftlich wie im Dezimalsystem subtrahiert. Das Vorgehen ist nachfolgend am Beispiel $17 : 5 = 3$ Rest 2 gezeigt:

$$\begin{array}{rl} & 10001 \;:\; 101 \;=\; 011 \text{ Rest } 010 \\ - & \;\;0 \\ \hline & 1000 \\ - & \;101 \\ \hline & \;\;111 \\ - & \;101 \\ \hline & \;\;010 \end{array}$$

Wir sehen also, dass sich sämtliche Rechnungen durch einfache Verschiebungen und Binäradditionen realisieren lassen.

4. Addition mit logischen Operationen

Wir identifizieren Belegungen von Bits mit den logischen Werten falsch und wahr und versuchen, die Addition zweier Bits mit Übertrag durch einen logischen Ausdruck darzustellen. Die Wertetabelle für die Addition mit Übertrag $A + B = S$, C ist in Tabelle 1 gezeigt. Die Operation $(A, B) \mapsto (S, C)$ wird als *Halbaddierer* bezeichnet.

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

TABELLE 1. Die als Halbaddition bezeichnete Addition zweier Bits A und B ergibt die Summe S und den Übertrag (*Carry Bit*) C.

Als logische Ausdrücke verknüpfen wir die Zeilen, die den Wert 1 ergeben, jeweils mit der ODER-Operation und erhalten

$$S = (\neg A \wedge B) \vee (A \wedge \neg B) = A \vee_! B,$$
$$C = A \wedge B,$$

wobei der Ausdruck für S dem ausschließenden ODER, das hier durch $\vee_!$ symbolisiert wird, entspricht.
Als nächstes betrachten wir die Addition von drei Bits A, B und C_{in}, wobei C_{in} ein Übertragsbit aus der Addition zweier niederer Bits sei. Dies ergibt ein Summationsbit S und ein Übertragsbit C_{out}, deren Werte in Tabelle 2 aufgeführt sind. Die Operation $(A, B, C_{in}) \mapsto (S, C_{out})$ wird als *Volladdierer* bezeichnet.

A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

TABELLE 2. Die als Volladdierer bezeichnete Addition dreier Bits A, B und C_{in} ergibt die Summe S und den Übertrag C_{out}.

Die Wertetabelle liefert die Ausdrücke

$$\begin{aligned} S &= (\neg A \wedge \neg B \wedge C_{in}) \vee (\neg A \wedge B \wedge \neg C_{in}) \\ &\qquad \vee (A \wedge \neg B \wedge \neg C_{in}) \vee (A \wedge B \wedge C_{in}) \\ &= (A \vee_! B) \vee_! C_{in} \\ C_{out} &= (\neg A \wedge B \wedge C_{in}) \vee (A \wedge \neg B \wedge C_{in}) \\ &\qquad \vee (A \wedge B \wedge \neg C_{in}) \vee (A \wedge B \wedge C_{in}) \\ &= ((A \vee_! B) \wedge C_{in}) \vee (A \wedge B). \end{aligned}$$

Die Volladdition können wir mit Hilfe zweier Halbaddierer und einer ODER-Operation darstellen, wie in Abbildung 1 gezeigt ist.

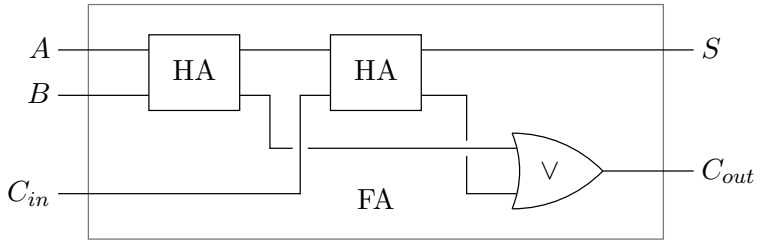

ABBILDUNG 1. Schematische Konstruktion eines Volladdierers (FA) mittels zweier Halbaddierer (HA) und einer ODER-Operation.

Sollen jetzt die Binärzahlen $a = a_k \dots a_1 a_0$ und $b = b_k \dots b_1 b_0$ addiert werden, so kann dies mit einem Halbaddierer für die niedersten Bits sowie $k-1$ Volladdierern für die Additions der höheren Bits unter Berücksichtigung der vorigen Überträge geschehen. Die Realisierung ist in Abbildung 2 gezeigt.

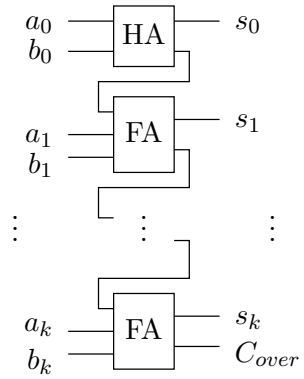

ABBILDUNG 2. Addition zweier k-stelliger Binärzahlen mittels eines Halbaddierers (HA) und $(k-1)$ Volladdierern (FA) unter Berücksichtigung eines *Overflow*-Bits C_{over}.

Die in Abbildung 2 gezeigte Schaltung ist einfach zu realisieren, aber nicht besonders effizient, da die Volladdierer jeweils das Übertragsbit des vorigen Voll- beziehungsweise Halbaddierers benötigen. Um die Rechenzeit zu verkürzen, halbiert man die Bitfolgen und berechnet die Summe der niederen Bits

$$a_{k/2} \dots a_0 + b_{k/2} \dots b_0 = [S_{low}, C_{low}]$$

sowie parallel dazu die zwei Summen der höheren Bits, einmal mit und einmal ohne Übertragsbit, das heißt

$$0 + a_k \dots a_{k/2+1} + b_k \dots b_{k/2+1} = [S^0_{high}, C^0_{over}]$$
$$1 + a_k \dots a_{k/2+1} + b_k \dots b_{k/2+1} = [S^1_{high}, C^1_{over}].$$

Die korrekte Summe ist $S^0_{high} S_{low}$ mit Überlaufbit C^0_{over} falls $C_{low} = 0$ und $S^1_{high} S_{low}$ mit Überlaufbit C^1_{over} andernfalls. Diese Entscheidung wird von einem sogenannten *Multiplexer* durchgeführt. Durch wiederholte Anwendung dieser Argumentation lässt sich die Rechenzeit mehrfach nahezu halbieren.

5. Realisierung mit Transistorschaltungen

Einzelne Bits beziehungsweise die logischen Werte falsch und wahr lassen sich sinnvoll mit elektrischen Signalen darstellen, das heißt an einer Leitung liegt entweder eine Spannung an und es fließt Strom oder nicht. Da sich jede logische Operation mittels der Grundfunktionen UND, ODER sowie NEGATION darstellen lässt, diskutieren wir nur deren technischen Realisierungen. Mit Transistoren kann über einen kleinen Stromfluss ein größerer kontrolliert werden, ähnlich wie es in Abbildung 3 für zwei Wasserkanäle illustriert ist. Liegt an der Basis eine Spannung an, so wird der Stromfluss vom Kollektor zum Emitter ermöglicht.

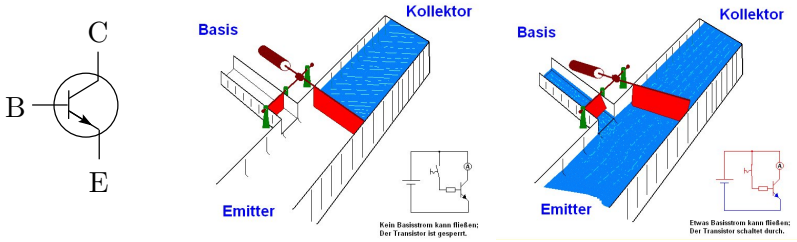

Abbildung 3. Schaltsymbol und Funktionsweise eines Transistors: ein kleiner Stromfluss von der Basis zum Emitter öffnet und schließt einen größeren vom Kollektor zum Emitter. (Quelle: Stefan Riepl (Quark48 21:02, 2. Dez. 2007 (CET)) (`https://commons.wikimedia.org/wiki/File:Transistor_animation.gif`), Transistor animation, `https://creativecommons.org/licenses/by-sa/2.0/de/legalcode`)

Abbildung 4 zeigt Transistorschaltungen für logische Grundoperationen. Die UND-Schaltung ist leicht verständlich: nur wenn an den Basen der beiden Transistoren Spannungen anliegen, das heißt nur wenn A und B wahr sind, kann Strom von der Spannungsquelle zur Erdung fließen und am Ausgang $A \wedge B$ liegt eine Spannung an. Bei der ODER-Schaltung überprüft man, dass

am Ausgang Spannung anliegt, sofern mindestens einer der Transistoren geschlossen ist, also Eingang A oder Eingang B Spannung führt. Etwas anders ist die Funktionsweise der NEGATION-Schaltung. Liegt am Eingang A eine Spannung an, so fließt der gesamte Strom aufgrund des Widerstands R2 zur Erdung und es liegt keine Spannung am Ausgang an. Tatsächliche Realisierungen sind in der Regel etwas komplexer, da beispielsweise wiederholte Spannungsabfälle vermieden werden müssen.

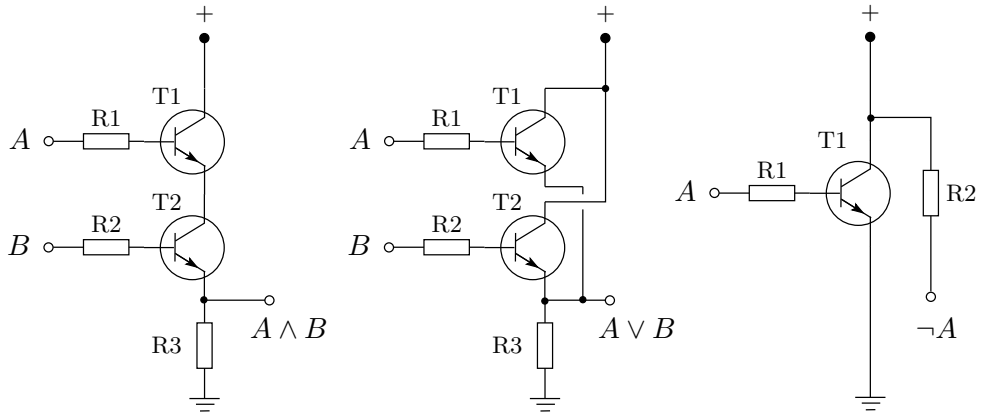

Abbildung 4. Transistorschaltungen für die konzeptionelle Realisierung der logischen Operationen UND, ODER und NEGATION.

IV FELDER, ZEIGER UND ABGELEITETE DATENTYPEN

S. BARTELS, 4.6.2018

1. Statische Felder

In vielen EDV-Anwendungen liegen numerierte Daten vor, wie beispielsweise die Temperaturen an den 365 Tagen eines Jahres. Diese lassen sich in Programmen mit Listen beziehungsweise *Feldern* darstellen. Ein (eindimensionales) Feld der Länge N mit Einträgen eines bestimmten Datentyps wird folgendermaßen deklariert:

```
type feld[N];
```

Dabei ist zu beachten, dass die Länge N zum Zeitpunkt der Übersetzung des Programms bereits definiert (bei einigen Compilern sogar als konstant) sein muss. Initiale Einträge können wie bei einfachen Variablen direkt mit der Deklaration festgelegt werden:

```
type feld[N] = {val0,val1,...,valN-1};
```

Alternativ können Einträge einzeln durch Angabe der Position definiert werden. Dabei ist zu beachten, dass die Indizierung in C++ mit dem Index 0 beginnt und im Fall von N Einträgen bis $N-1$ läuft:

```
feld[0] = val0; feld[1] = val1; ... feld[N-1] = valN-1;
```

Allgemeiner können mehrdimensionale Felder benutzt werden, die im Fall eines zweidimensionalen Feldes mittels der Deklaration und zeilenweisen Initialisierung

```
type feld[M][N] = {{val00,...},{val10,...},...};
```

definiert werden. Bei der direkten Initialisierung muss die Größe des Feldes nicht explizit angegeben werden. Auf einzelne Einträge eines zweidimensionalen Feldes wird mittels `feld[j][k]` zugeriffen. Abbildung 1 zeigt die entsprechende Realisierung einer Matrix-Vektor-Multiplikation.
Die Einträge eines Feldes werden hintereinander, in den dem Datentyp entsprechenden Abständen im Speicher abgelegt. Felder werden häufig auch als *Arrays* bezeichnet. Sollen die Inhalte eines Feldes einem anderen zugewiesen werden, so muss dies eintragsweise geschehen, da eine Zuweisung $a = b$ für Felder nicht zulässig ist.

2. Übergabe von Feldern

Felder können als Argumente ohne Spezifikation ihrer Größe in Unterfunktionen verwendet werden, allerdings ist die Behandlung solcher Argumente fundamental unterschiedlich von der Behandlung von Variablen einfachen

```
1  // kompilieren: g++ mat_vek_mult.cc -o mat_vek_mult.out
2  #include <iostream>
3  using namespace std;
4  int main(){
5    const int M = 2, N = 3;
6    double A[M][N] = {{1.0,2.0,3.0},{4.0,5.0,6.0}};
7    double x[N] = {1.0,2.0,3.0};
8    double b[M];
9    for (int j=0; j<M; j++){
10     b[j] = 0.0;
11     for (int k=0; k<N; k++){
12       b[j] += A[j][k]*x[k];
13       }
14     }
15   cout << "Das Matrix-Vektor-Produkt A*x ist: " << endl;
16   for (int j=0; j<M; j++){
17     cout << b[j] << ", ";
18   }
19   cout << endl;
20 }
```

ABBILDUNG 1. Realisierung einer Matrix-Vektor-Multiplikation mit Feldern fester Größe. Man beachte die Initialisierung der Einträge des Feldes b mit Nullwerten.

Typs als Funktionsargument. Bei einfachen Typen kopiert die Unterfunktion lediglich den Wert der Variablen in eine lokale Variable, die denselben Namen haben darf, der Wert der Variable im aufrufenden Programmteil bleibt davon aber unberührt. Dies wird als *Call-by-Value* bezeichnet, ein Beispiel ist in Abbildung 2 gezeigt.
Wird hingegen die Variable als Feld definiert, wie in Abbildung 3 gezeigt, so werden die Werte des Feldes durch den Funktionsaufruf verändert. Hier wird der Funktion die Speicheradresse des Beginns des Feldes übergeben und die Funktion überschreibt die dort abgelegten Werte. Diese Art des Funktionsaufrufs wird als *Call-by-Reference* bezeichnet.

3. Verwendung von Speicheradressen (Zeigern)

Eine sehr Hardware-nahe Programmierung ist mit der direkten Verwendung von Speicheradressen, die auch als *Zeiger* beziehungsweise *Pointer* bezeichnet werden, möglich. Dieses Prinzip ermöglicht zum Beispiel die Übergabe einfacher Variablen an Funktionen mittels *Call-by-Reference*. Ein Zeiger auf eine Variable eines bestimmten Typs wird deklariert durch

```
type *ptr;
```

In diesem Fall kann die Variable `ptr` eine Speicheradresse enthalten, während der Ausdruck `*ptr` wie eine normale Variable verwendet werden kann und

```
1  // kompilieren: g++ call_by_val.cc -o call_by_val.out
2  #include <iostream>
3  using namespace std;
4  void addiere_eins(int a){
5    a += 1;
6    cout << a << endl;
7  }
8  int main(){
9    int a = 1;
10   cout << a << endl;
11   addiere_eins(a);
12   cout << a << endl;
13 }
```

ABBILDUNG 2. Aufruf einer Unterfunktion mit einer einfachen Variable; das Programm führt zur Ausgabe der Werte 1, 2, 1.

```
1  // kompilieren: g++ call_by_ref.cc -o call_by_ref.out
2  #include <iostream>
3  using namespace std;
4  void addiere_eins(int a[]){
5    a[0] += 1;
6    cout << a[0] << endl;
7  }
8  int main(){
9    int N = 1;
10   int a[N] = {1};
11   cout << a[0] << endl;
12   addiere_eins(a);
13   cout << a[0] << endl;
14 }
```

ABBILDUNG 3. Aufruf einer Unterfunktion mit einem Feld; das Programm führt zur Ausgabe der Werte 1, 2, 2.

den Wert hat, der an der durch `ptr` definierten Speicheradresse abgelegt ist. Der Typ eines Zeigers muss festgelegt werden, damit klar ist, wie viele Bits im Speicher zum Wert der Variable gehören. Bevor der Variable `*ptr` ein Wert zugeordnet werden kann, muss dem Zeiger `ptr` eine Adresse zugeordnet worden sein. Ist `var` eine Variable vom Typ des Pointers, so erhält man die Adresse des zugehörigen Speicherbereichs durch `&var`. Diese kann dann einem Zeiger zugewiesen werden:

```
ptr = &var;
```

In diesem Zusammenhang bezeichnet man den Operator `*` als *Dereferenzierungs-* und `&` als *Referenzierungsoperator*. Das in Abbildung 4 gezeigte Programm illustriert die Verwendung und führt zur Ausgabe der Adresse des Zeigers und des dort abgelegten Werts. Der Einsatz von Zeigern ist mit gewissen Gefahren bei der Speicherverwaltung verbunden und sollte möglichst vermieden werden.

```
1  // kompilieren: g++ zeiger_test.cc -o zeiger_test.out
2  #include <iostream>
3  using namespace std;
4  int main(){
5    int *ptr;
6    int j = 1, k = 2;
7    ptr = &k;
8    cout << "Adresse: " << ptr << ", Wert: " << *ptr << endl;
9    *ptr = 5;
10   cout << "Adresse: " << ptr << ", Wert: " << *ptr << endl;
11   ptr = &j;
12   cout << "Adresse: " << ptr << ", Wert: " << *ptr << endl;
13   //
14   cout << "Beachte veraenderten Wert von k, k = " <<  k << endl;
15 }
```

Abbildung 4. Verwendung eines Zeigers und Ausgabe seiner Adresse sowie des dort abgelegten Werts.

4. Vektoren variabler Länge

Die Länge einer Liste oder die Größe eines Feldes ergibt sich häufig erst im Laufe der Berechnungen eines Programms. Die C++-Bibliothek `vector` stellt mächtige Werkzeuge für das Arbeiten mit Listen beziehungsweise Vektoren zur Verfügung. Ein Vektor der Länge N mit Einträgen eines bestimmten Typs wird mit folgendem Befehl deklariert:

```
std::vector<type> vec(N);
```

Die Einträge können bei Deklaration mit einem konstanten Wert initialisiert werden, wenn `vec(N,val)` statt `vec(N)` verwendet wird. Ähnlich können nach der Deklaration die Einträge eines Vektors mit dem Befehl `fill` auf einen konstanten Wert gesetzt werden. Die Länge wird mit der Methode `resize` geändert, wobei der Aufruf mittels `vec.resize(M)` geschieht. Eine Übersicht verschiedener Befehle und weiterer Methoden findet sich in Tabelle 1, ein einfaches Beispielprogramm ist in Abbildung 5 gezeigt. Man beachte dabei, dass die Übergabe dieser Vektoren mittels *call-by-reference* erfolgt. Für das Arbeiten mit zweidimensionalen Feldern variabler Länge und fester Anzahl von Spalten steht in C++ die Bibliothek `array` zur Verfügung.

`std::vector`	Deklaration eines Vektors
`std::fill`	Initialisierung eines Vektors
`std::copy`	Kopieren von Einträgen eines Vektors
vec`.size`	Länge eines Vektors
vec`.resize`	Veränderung der Länge eines Vektors
vec`.begin`	Adressierung des Anfangs eines Vektors
vec`.end`	Referenzierung des Endes eines Vektors
vec`.push_back`	Anhängen eines Elements an einen Vektor
vec`.pop_back`	Entfernen des letzten Elements eines Vektors

TABELLE 1. C++-Routinen zur Bearbeitung eines Vektors variabler.

```
1  // kompilieren: g++ dynam_vektor.cc -o dynam_vektor.out
2  #include <iostream>
3  #include <vector>
4  using namespace std;
5  int main(){
6    int N = 2;
7    vector<double> u(N);
8    fill(u.begin(),u.end(),1.);
9    N = 4;
10   u.resize(N);
11   u[2] = 5.0; u[3] = 7.0;
12   for (int j=0; j<(int)u.size(); j++)
13     cout << "u[" << j << "] = " << u[j] << endl;
14 }
```

ABBILDUNG 5. Anlegen eines Vektors und Änderung seiner Länge.

5. ZEICHENKETTEN

Variablen, die Zeichenketten wie beispielsweise Personennamen als Werte annehmen sollen, können als Felder über Literalen das heißt über dem Typ `char` deklariert werden. Einer Variable wird dann mittels Anführungsstrichen ein Wert zugewiesen:

```
char wort[6] = "hallo";
```

Man beachte, dass für eine Zeichenkette mit n Zeichen ein Feld der Länge $n + 1$ benötigt wird, da implizit das Endsymbol `\0` mit gespeichert wird. Der Zugang über Felder stößt schnell an Grenzen, da eine maximale Länge gewählt werden muss, was mit Hilfe einer globalen Konstante geschehen kann. Ein Beispiel ist in Abbildung 7 gezeigt. Dabei wird die maximale Länge über ein Präprozessor-Makro festgelegt.

```
// kompilieren: g++ vektor_funktion.cc -o vektor_funktion.out
#include <iostream>
#include <vector>
using namespace std;
vector<double> lin_komb(vector<double> x, vector<double> y,
          double a, double b){
  int N = (int)x.size();
  vector<double> z(N);
  for (int j=0; j<N; j++)
    z[j] = a*x[j]+b*y[j];
  return z;
}
int main(){
  int n = 2;
  vector<double> u(n,1.0);
  vector<double> v(n,0.0);
  fill(u.begin(),u.end(),5.0);
  copy(u.begin(),u.end(),v.begin());
  u = lin_komb(u,v,1.0,2.0);
  for (int j=0; j<(int)u.size(); j++)
    cout << "u[" << j << "] = " << u[j] << endl;
}
```

ABBILDUNG 6. Berechnung der Linearkombination zweier Vektoren in einer Unterfunktion.

```
// kompilieren: g++ char_feld.cc -o char_feld.out
#include <iostream>
#define max_laenge 20
using namespace std;
int main(){
  char zeichen_kette[max_laenge] = "lange kette ...";
  cout << zeichen_kette << endl;
}
```

ABBILDUNG 7. Deklaration einer Zeichenkette als Feld über Literalen mit global definierter maximaler Länge.

Kompatibel mit Feldern über Literalen und komfortabel in seiner Verwendung ist der von der Bibliothek beziehungsweise Klasse `string` bereitgestellte Typ `string`. Das Speichermanagement erfolgt hier automatisch und das Arbeiten mit Variablen vom gleichnamigen Typ `string` ist im Hinblick auf Vergleiche ähnlich dem von Zahlentypen. Man kann Zeichenketten dieses Typs alphabetisch vergleichen und mit dem Operator `+` aneinanderhängen. Zudem sind Funktionen wie `length` implementiert, die mit dem Selektionsoperator ausgeführt werden, wie in Abbildung 8 beispielhaft gezeigt ist.

```
1  // kompilieren: g++ string_klasse.cc -o string_klasse.out
2  #include <iostream>
3  #include <string>
4  using namespace std;
5  int main(){
6    string wort_1 = "fussball";
7    string wort_2 = "spiel";
8    string wort_3 = "null";
9    if (wort_1<wort_2)
10     wort_3 = wort_1+wort_2;
11   cout << wort_3 << endl;
12   cout << wort_3.length() << endl;
13 }
```

ABBILDUNG 8. Verwendung einer Bibliothek zur Definition und Bearbeitung von Zeichenketten.

6. Abgeleitete Datentypen

Sogenannte *Strukturen* oder *Verbunde* erlauben die Zusammenfassung von Variablen verschiedenen Typs zu einer und werden als abgeleiteter Datentyp bezeichnet. Anwendungsbeispiele sind Positionen im Raum, die drei Koordinaten besitzen, oder Adressbücher bei denen für jeden Eintrag verschiedene Informationen erfasst werden. Die Definition einer Struktur und die Deklaration einer Variable dieses Verbunds erfolgt über die Befehle

```
struct name{type var1; ...; type varn;};
name struct_var;
```

Der Zugriff auf die verschiedenen Komponenten einer Verbund-Variable erfolgt durch den Selektionsoperator `.` und die Angabe des Namens der Untervariable, das heißt eine Zuweisung erfolgt über:

```
struct_var.var1 = val1;
```

Eine Variable eines Verbunds kann wie eine normale Variable in Funktionen verwendet werden, siehe Abbildung 9.

7. Dateioperationen

Das Lesen und Schreiben von Daten in Dateien ähnelt der Ein- und Ausgabe von Daten auf dem Monitor. Vereinfachend formuliert wird die Ausgabe umgeleitet in die dafür geöffnete Datei. Zum Verwalten der Öffnungs- und Schließprozesse wird der Typ `fstream` aus der gleichnamigen Bibliothek verwendet. Das in Abbildung 10 gezeigte Programm schreibt eine Textzeile in eine Datei beziehungsweise fügt zu vorhandenen Inhalten hinzu. Die Variable `f` ermöglicht Operationen wie das Öffnen und Schließen über einen Selektionsoperator. Bei der Verwendung des Befehls

```
f.open("test.dat",ios::out);
```

```
1  // kompilieren: g++ struct.cc -o struct.out
2  #include <iostream>
3  struct punkt{
4    double x, y, z;
5    char farbe[10];
6  };
7  punkt mod_punkt(punkt q){
8    q.x = q.x+1.0;
9    return q;
10 }
11 using namespace std;
12 int main(){
13   punkt p = {1.0,2.0,3.0,"gelb"};
14   p = mod_punkt(p);
15   cout << "p = ( " << p.x << " , " << p.y
16        << " , " << p.z << " ) " << endl;
17   cout << p.farbe << endl;
18 }
```

ABBILDUNG 9. Definition eines Verbunds und Verwendung in einer Funktion.

werden bereits vorhandene Inhalte gelöscht und es wird eine Textdatei erzeugt. Jedem Öffnungsbefehl sollte ein Schließbefehl folgen. Sollen Inhalte an bereits bestehende Inhalte angehängt werden, ist die zusätzliche, mit einem senkrechten Strich abgetrennte Option `ios::app` zu verwenden.

```
1  // kompilieren: g++ datei_schreiben.cc -o datei_schreiben.out
2  #include <fstream>
3  using namespace std;
4  int main(){
5    fstream f;
6    f.open("test.dat",ios::out|ios::app);
7    f << "Hinzufuegen von Text in eine Datei" << endl;
8    f.close();
9  }
```

ABBILDUNG 10. Öffnen einer Datei, Hinzufügen einer Textzeile und abschließendes Schließen der Datei.

Bei der Arbeit mit Dateien kann es leicht zu Fehlern kommen, wenn etwa die Datei schreibgeschützt ist. Um solche Fehler abzufangen, kann die Methode `good` verwendet werden:

```
if (f.good()){
   // Schreiben beginnt
}
```

Für wissenschaftliche Zwecke ist meist das Lesen von Daten aus Dateien erforderlich. Dazu muss ein geeignetes Format der Daten in der Datei vorliegen, um sie korrekt aus der Datei auslesen zu können. Mit dem Befehl

```
input >> var
```

wird die nächste Information aus der Datei eingelesen, deren Format durch den Typ der Variable `var` spezifiziert ist. Dies ähnelt sehr der Verwendung des Befehls `cin`.

```
1  // kompilieren: g++ datei_lesen.cc -o datei_lesen.out
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5  int main(){
6    ifstream input("koordinaten.txt");
7    if(!input){
8      cerr << "Datei nicht gefunden" << endl;
9      return 0;
10   }
11   int j, N = 4;
12   double V[N][3];
13   double v_x, v_y, v_z;
14   for(j=0; input >> v_x >> v_y >> v_z; j++){
15     V[j][0] = v_x; V[j][1] = v_y; V[j][2] = v_z;
16   }
17   for(j=0; j<N; j++){
18     cout << V[j][0] << "   " << V[j][1];
19     cout << "   " << V[j][2] << endl;
20   }
21 }
```

Abbildung 11. Lesen der Inhalte der in Abbildung 12 gezeigten Datei.

```
1  1.0     2.2     3.3
2  4.0     5.0     6.0
3  7.0     8.0     9.0
4  10.0   11.0    12.0
```

Abbildung 12. Daten einer Textdatei.

V ALGORITHMIK

S. BARTELS, 18.6.2018

1. Zielsetzungen

Unter *Algorithmik* versteht man die Entwicklung und Analyse von Algorithmen, also von Computer-realisierbaren Verfahren zur systematischen Lösung einer Aufgabe. Typische Aufgaben sind das Sortieren von Listen, die Bestimmung der Lösung eines mathematischen Problems oder die Berechnung der Wahrscheinlichkeit eines Ereignisses auf Basis gegebener Daten. Bei der Analyse eines Algorithmus sind die folgenden Kriterien relevant.

(i) *Durchführbarkeit:* Es muss sicher gestellt werden, dass alle Schritte wohldefiniert sind, so dass zum Beispiel keine Divisionen durch Null auftreten.
(ii) *Terminierung:* Es muss gewährleistet sein, dass das Verfahren regulär stoppt und nicht beispielsweise in eine Endlosschleife gerät.
(iii) *Korrektheit:* Es muss nachgewiesen werden, dass für alle zulässigen Eingaben das richtige Ergebnis berechnet wird.
(iv) *Effizienz:* Es muss untersucht werden, ob der Algorithmus für realistische Problemgrößen in akzeptabler Zeit terminiert.

Einige dieser Aspekte erscheinen auf den ersten Blick trivial, ihre Relevanz zeigt sich jedoch, wenn man unterschiedliche Algorithmen betrachtet.

2. Algorithmusarten

Die am häufigsten autretenden Algorithmen sind dynamische, iterative und rekursive Algorithmen.

2.1. **Dynamische Algorithmen.** Bei *dynamischen Algorithmen* wird eine häufig numerierte Folge von Anweisungen sukzessive abgearbeitet, was meist in Form einer `for`-Schleife realisiert wird. Beispiele sind die Berechnung einer Summe und die Suche eines Elements in einer Liste.

2.2. **Iterative Algorithmen.** Bei *iterativen Algorithmen* wird eine Rechenvorschrift, die auch Fallunterscheidungen enthalten darf, wiederholt auf einen Datensatz angewendet, bis ein geeignetes Abbruchkriterium erfüllt ist. Abstrakt lässt sich dies mit einer Abbildung $T : X \to X$ beschreiben, die für einen Startwert $x^0 \in X$ die *Iterierten* $(x^k)_{k=0,1,\dots}$ durch die *Iterationsvorschrift*

$$x^{k+1} = T(x^k)$$

definiert, bis sich die Iterierten nicht mehr stark ändern, also beispielsweise bis $\|x^{k+1} - x^k\| \leq \delta$ für eine kleine Zahl $\delta > 0$ gilt. Die Termininerung und Korrektheit lässt sich in einigen Situationen mit dem Banachschen Fixpunktsatz nachweisen. Ein Beispiel eines iterativen Verfahrens ist die Berechnung der Quadratwurzel nach Heron. Man beachte, dass hier eine Aufgabe nur approximativ gelöst wird.

2.3. **Rekursive Algorithmen.** *Rekursive Algorithmen* basieren meist auf der Beobachtung, dass sich ein Problem einer bestimmten Größe auf ähnliche Probleme kleinerer Größe zurückführen und für eine gewisse minimale Problemgröße direkt lösen lässt. Ein Beispiel ist die Fakultätsfunktion $f(n) = n!$, die die *Rekursionsformel*

$$f(j) = jf(j-1)$$

für $j \geq 1$ und die *Rekursionsverankerung*

$$f(0) = 1$$

erfüllt. In diesem Fall ist die Problemgröße das Argument n und wir führen die Berechnung von $n!$ auf die Berechnung von $(n-1)!$ zurück. Rekursionen zeichnen sich dadurch aus, dass Funktionen oder Routinen sich selbst aufrufen und entstehen auf natürliche Weises bei induktiv definierten Objekten. Es gibt aber Fälle, die in keiner Form dem intuitiven menschlichen Handeln entsprechen.

Beispiel 2.1 (Türme von Hanoi)**.** *Beim Problem der Türme von Hanoi soll ein der Größe nach sortierter Stapel mit n Scheiben unterschiedlicher Größe von einer Standposition A auf eine Zielposition B unter Verwendung einer Hilfsposition C versetzt werden. Dabei dürfen nur einzelne Scheiben versetzt werden und sie müssen stets der Größe nach geordnet sein. Im Fall von drei Scheiben ist das Problem noch intuitiv lösbar, aber schon bei vier Scheiben müssen sehr viele Bewegungen durchgeführt werden. Angenommen, wir wissen, wie man einen Stapel mit $(n-1)$ Scheiben von einer Position auf eine andere unter Verwendung der dritten als Hilfsposition versetzen kann. Dann können wir den Stapel mit n Scheiben als Zusammensetzung eines Stapels mit $(n-1)$ Scheiben und der untersten, größten Scheibe betrachten und das Problem in drei Schritten lösen:*

(1) *Versetze den oberen $(n-1)$-Teilstapel von A nach C.*
(2) *Versetze den unteren 1-Teilstapel von A nach B.*
(3) *Versetze den $(n-1)$-Stapel von C nach B.*

Das Vorgehen ist in Abbildung 1 illustriert. In diesen Schritten ist nur die Versetzung kleinerer Stapel erforderlich. Während die Versetzung eines Stapels mit einer Scheibe trivial ist, lässt sich die Versetzung von Stapeln mit $(n-1)$ Scheiben wieder jeweils auf zwei Versetzungen von Stapeln mit $(n-2)$ Scheiben und eine eines Stapels mit einer Scheibe zurückführen.

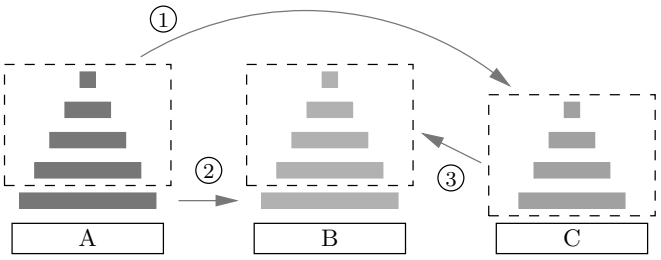

ABBILDUNG 1. Lösung des Problems der Türme von Hanoi durch Reduktion auf kleinere Probleme derselben Art.

Im Beispiel wird eine sehr komplexe Aufgabenstellung mit einer einfachen Rekursionsvorschrift gelöst. Etwas problematisch ist dabei, dass der Aufwand in jedem Reduktionsschritt um einen Faktor 2 vergrößert wird, was zu mehr als 2^n Operationen führt. Vor diesem Hintergrund ist es häufig sinnvoll, eine Rekursion zu vermeiden und durch einen dynamischen, vorwärtsgerichteten Algorithmus zu ersetzen, was im Fall der Fakultät und der Fibonacci-Folge nicht jedoch im Fall der Türme von Hanoi einfach möglich ist.

3. KOMPLEXITÄT

Wir wollen die Zeit, die ein Algorithmus zur Lösung eines Problems benötigt, unabhängig von der Art und Weise seiner praktischen Umsetzung quantifizieren. Dazu untersuchen wir, wie sich der Aufwand, das heißt die Anzahl der benötigten Operationen wie arithmetischen Operationen oder Vergleichen, verändert, wenn wir die Problemgröße verändern.

Definition 3.1. *(i) Die* (Problem-) Größe *einer Aufgabe ist eine charakteristische Größe des Problems wie die Anzahl der zu bearbeitenden Daten, die Anzahl von Summanden oder der Index eines zu berechnenden Folgeglieds.*
(ii) Der Aufwand *eines Algorithmus zur Lösung einer Aufgabe der Größe n ist die Anzahl $\mathcal{A}(n)$ der erforderlichen arithmetischen Operationen und Vergleiche.*

Wir sind meist an einer oberen Schranke für den maximalen Aufwand interessiert, das heißt dem Aufwand für ein *Worst-Case-Scenario*.

Beispiele 3.2. *(i) Das Suchen eines Elements in einer Liste mit n Elementen erfordert maximal n Vergleiche, das heißt $\mathcal{A}(n) \leq n$.*
(ii) Sind k arithmetische Operationen zur Bestimmung der Summanden s_j in der Summe $S_n = s_1 + s_2 + ... + s_n$ erforderlich, so erhalten wir den Gesamtaufwand $\mathcal{A}(n) \leq k(n+1)$. Im Fall der Gauß-Summe mit $s_j = j$ lässt sich der Aufwand mit der Formel $S_n = n(n+1)/2$ auf drei Operationen reduzieren, unabhängig von n.
(iii) Die rekursive Berechnung des Folgeglieds $f_n = f_{n-1} + f_{n-2}$ mit Initialisierung $f_0 = 0$ und $f_1 = 1$ der Fibonacci-Folge erfordert den Aufwand $\mathcal{A}(n) = 2^n$. Eine dynamische Realisierung hingegen nur $\mathcal{A}(n) \leq n$.

Um die Praktikabilität eines Algorithmus einordnen zu können, führen wir gewisse Aufwandsklassen ein, die prägnant angeben, wie sich der Aufwand vergrößert, wenn die Problemgröße beispielsweise verdoppelt wird.

Definition 3.3. *(i) Der Aufwand $\mathcal{A}(n)$ ist polynomiell, wenn Zahlen $p, c \geq 0$ existieren, sodass für alle $n \geq 1$ gilt*

$$\mathcal{A}(n) \leq cn^p.$$

Im Fall $p = 0, 1, 2, 3$ nennen wir den Aufwand konstant*,* linear*,* quadratisch *beziehungsweise* kubisch*.*
(ii) Der Aufwand $\mathcal{A}(n)$ ist exponentiell*, wenn Zahlen $s, c > 0$ existieren, sodass für all $n \geq 1$ gilt*

$$\mathcal{A}(n) \geq cs^n.$$

Mit diesen Begriffen lassen sich die erforderlichen Berechnungen von Aufgaben übersichtlich einordnen.

Beispiele 3.4. *(i) Das dynamische Suchen eines Elements in einer Liste erfordert linearen Aufwand.*
(ii) Die Bestimmung der Gauß-Summe erfordert linearen Aufwand bei dynamischer Realisierung und konstanten Aufwand bei Verwendung der Summenformel.
(iii) Die Berechnung eines Glieds der Fibonacci-Folge besitzt exponentiellen Aufwand bei rekursiver Realisierung und linearen Aufwand bei dynamischer Realisierung.

In der Regel erfordert die Lösung einer Aufgabe der Größe n mindestens n Operationen, da meist jede der n Informationen in die Berechnungen eingeht. Ein Algorithmus mit linearem Aufwand ist daher meist optimal und die Aufgabe kann *effizient* für eine große Spanne von Problemgrößen sinnvoll vom Computer gelöst werden. Bei exponentiellem Aufwand beispielsweise mit $\mathcal{A}(n) = 2^n$ stößt man sehr schnell an Grenzen des Realisierbaren, denn schon für ein Problemgrößen $n \geq 70$ ergeben sich

$$\mathcal{A}(n) \geq 10^{20}$$

Operationen, was selbst auf Giga-Hertz-Rechnern, die 10^{10} Operationen pro Sekunde durchführen können, Schwierigkeiten verursacht, denn selbst beim Einsatz von mehreren hundert Rechnern würde sich eine Rechenzeit von mehr als einem Jahr ergeben. Auch kubischer Aufwand führt schnell zu Laufzeitproblemen. In vielen Anwendungen ist ist quadratischer Aufwand noch vertretbar, besser ist jedoch superlinearer Aufwand, der zwischen linearem und quadratischem Aufwand liegt.

Bemerkungen 3.5. *(i) Neben der Wahl eines Algorithmus beeinflusst auch die konkrete Implementierung die Laufzeit eines Programms. Beispielsweise ist der Zugriff auf die Einträge einer Liste abhängig von deren technischer Realisierung.*

(ii) Probleme, die auf Algorithmen mit exponentiellem Aufwand führen, lassen sich meist der Klasse der NP-vollständigen Probleme *zuordnen. Dazu gehört beispielsweise das Problem des Handelsreisenden, der eine kürzeste Rundreise durch n Städte sucht. Für diese Klasse von Problemen sind bisher keine Lösungsalgorithmen mit polynomiellem Aufwand bekannnt, es ist jedoch auch noch nicht gelungen, deren Nichtexistenz formal nachzuweisen.*

4. Sortierverfahren

Das Sortieren von Listen von Datensätzen ist eine zentrale Aufgabe der Datenverarbeitung. Um die wichtigsten Ansätze darzustellen, betrachten wir eine Liste $I = [a_1, a_2, \ldots, a_n]$ mit n ganzen Zahlen, die aufsteigend sortiert werden soll.

4.1. **Bubblesort.** Das *Bubblesort*-Verfahren ähnelt dem Sortieren von Büchern in einem Regal, die der Reihe nach geprüft und gegebenenfalls nach vorne versetzt werden.

Algorithmus 4.1 (Bubblesort)**.** *Sei die Liste $I = [a_1, \ldots, a_n]$ gegeben und setze $i = 2$.*

(1) *Gilt $a_i \geq a_{i-1}$, so gehe zu Schritt (3).*
(2) *Gilt $a_i < a_{i-1}$, so vertausche wiederholt die Elemente a_{i-k} und a_{i-k-1} für $k = 0, 1, \ldots,$ bis $a_{i-k} \geq a_{i-k-1}$ oder $k = i - 1$ gilt.*
(3) *Sofern $i < n$ gilt, erhöhe $i \to i + 1$ und fahre fort mit Schritt (1); stoppe andernfalls.*

Im i-ten Schritt des dynamischen Algorithmus fallen ein Vergleich sowie bis zu $i - 2$ weitere an, sofern Einträge vertauscht werden müssen. Im schlechtesten Fall erhalten wir also

$$\mathcal{A}(n) \leq \sum_{i=2}^{n}(i-1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \leq \frac{1}{2}n^2,$$

das heißt, dass das Verfahren quadratischen Aufwand besitzt. Dies ist für Listen mit bis zu $n = 1000$ Einträgen akzeptabel, insbesondere da das Verfahren sehr leicht zu implementieren ist.

4.2. **Mergesort.** Das *Mergesort*-Verfahren basiert auf der rekursiven Verkleinerung der zu sortierenden Liste. Nach Auffüllen der Liste mit Nullen oder sehr großen Einträgen, können wir ohne Probleme annehmen, dass $n = 2^k$ mit einer ganzen Zahl k gilt, sodass wir die Liste wiederholt halbieren können, bis wir zu einelementigen Teillisten kommen. Zur Motivation des Verfahrens nehmen wir an, dass die linke und rechte Teilliste $I^{\ell}_{n/2}$ sowie $I^{r}_{n/2}$ bereits sortiert sind. Die sortierte Liste I_n erhalten wir dann durch ein einfaches Zusammenfügen, den sogenannten *merge*- Schritt, der beiden Teillisten. Dies führt auf den folgenden rekursiven Algorithmus, der für eine Liste I eine sortierte Liste $\widetilde{I}$ zurückgibt.

Algorithmus 4.2 (Mergesort). *Aufruf:* $\widetilde{I} = \mathrm{msort}(I,k)$.
Für die Länge n der Liste I gelte $n = 2^k$ für ein $k \geq 0$.
(1) *Gilt $k = 0$, so ist die Liste bereits sortiert, das heißt setze $\widetilde{I} = I$.*
(2) *Gilt $k > 0$, so setze*

$$\widetilde{I} = \mathrm{merge}\big(\mathrm{msort}(I^\ell_{n/2}, k-1), \mathrm{msort}(I^r_{n/2}, k-1)\big).$$

Die Realisierung der Zusammenführung zweier Listen der Länge m kann mit einem dynamischen Algorithmus mit linearem Aufwand $\mathcal{A}_{\mathrm{merge}}(m) = 2m$ realisiert werden. Der Aufruf von msort mit einer Liste der Länge n führt zu einem Aufruf von merge mit zwei Listen halber Länge und zwei Aufrufen von msort mit Listen halber Länge. Insgesamt erhalten wir den Aufwand

$$\mathcal{A}(n) = n + 2\mathcal{A}(n/2).$$

Diese Argumentation wenden wir wiederholt an und erhalten

$$\begin{aligned}\mathcal{A}(n) &= n + 2\big(n/2 + 2\mathcal{A}(n/4)\big) = 2n + 4\mathcal{A}(n/4)\\ &= 2n + 4\big(n/4 + 2\mathcal{A}(n/8)\big) = 3n + 8\mathcal{A}(n/8)\\ &= \cdots = kn + 2^k\mathcal{A}(n/2^k) = kn + 2^k\mathcal{A}(1) = kn,\end{aligned}$$

wobei wir verwendet haben, dass das Sortieren einelementiger Listen keinen Aufwand erfordert, also $\mathcal{A}(1) = 0$ gilt. Da $k = \log_2(n)$ ist, folgt

$$\mathcal{A}(n) = \log_2(n)n.$$

Dies ist ein Beispiel eines superlinearen Aufwands, der sich allerdings kaum von linearem Aufwand unterscheidet, da zum Beispiel $\log_2(10^6) \leq 20$. Problematisch ist bei rekursiven Algorithmen oft eine effiziente Speicherverwaltung.

4.3. **Andere Sortierverfahren.** Real auftretende Datensätze besitzen in der Regel zusätzliche Strukturen hinsichtlich ihrer Verteilung, beispielsweise sind die Geburtsdaten einer Personengruppe meist gleichmäßig über einen gewissen Zeitraum verteilt oder die Anfangsbuchstaben von Nachnamen treten mit bekannten Häufigkeiten auf. In diesem Fall kann man den Datenbereich einfach unterteilen, die Einträge den Teilbereichen zuordnen und sich auf das Sortieren der in der Regel erheblich kleineren Teillisten beschränken. Auf solchen Beobachtungen basieren die *Bucketsort*- und *Quicksort*-Verfahren. Unter geeigneten Bedingungen an die Einträge der Liste führen sie zu sehr effizienten Verfahren können aber im schlechtesten Fall auch zu quadratischem Aufwand führen.

5. Künstliche Intelligenz

In klassischen Algorithmen werden vom Nutzer erdachte Lösungsstrategien realisiert, die meist auf einem sehr guten Verständnis der zugrundeliegenden Aufgabe basieren. Algorithmen der *künstlichen Intelligenz*, kurz KI, versuchen, auf für Menschen schwer überschaubare oder nur mit sehr hohem Aufwand lösbare Probleme, Antworten mit vertretbarem Aufwand zu erzeugen.

Wir folgen in diesem Abschnitt der Darstellung des Buchs *Algorithmen kompakt und verständlich* von M. von Rimscha (Springer, 2008) und verweisen auf dieses Buch für weitere Details.

5.1. **Maschinelles Lernen.** Beim *maschinellen Lernen* wird ein menschliches Entscheidungsverhalten unter Berücksichtigung verschiedener Kriterien beobachtet und anschließend ausgewertet. Mit der statistischen Bewertung der Kriterien lässt sich dann ein Entscheidungsbaum konstruieren, der das Entscheidungsverhalten strukturiert widergibt. Ein populäres Beispiel ist die Freizeitgestaltung einer Person, die abhängig von Wetter und Wochenende sowie Verfügbarkeit interessanter Kinofilme und Freunden eine Entscheidung zwischen den Aktivitäten Radtour, Kino, Café, Schwimmen und zu Hause bleiben trifft. Die beobachteten Daten führen auf Tabelle 1, die jedoch nur einen Auszug eines größeren Datensatzes darstellt.

Freunde	Wetter	Kinofilm	Wochenende	Entscheidung
Ja	Sonne	Ja	Ja	Radtour
Ja	Regen	Nein	Nein	zu Hause
Nein	Sonne	Nein	Nein	Schwimmen
Nein	Bedeckt	Nein	Nein	Radtour
⋮	⋮	⋮	⋮	⋮

TABELLE 1. Entscheidungsverhalten bei Berücksichtigung verschiedener Kriterien.

Abhängig von der Häufigkeit des Auftretens eines Kriteriums wird seine Relevanz quantifiziert und damit die Priorität im Entscheidungsbaum, der in Abbildung 2 dargestellt ist, festgelegt.

5.2. **Schwarmintelligenz.** Das Konzept der *Schwarmintelligenz* imitiert den in der Natur beobachteten Effekt, dass eine Gruppe einfach entscheidender Individuen in ihrer Gesamtheit zu Lösungen für komplexe Aufgabenstellungen gelangen kann. Ameisen beispielsweise hinterlassen auf dem Weg von ihrem Nest zur Nahrungsstelle Duftmarken, an denen sie sich später orientieren. Der kürzeste Weg ist höher frequentiert, sodass eine größere Konzentration der Duftstoffe auftritt, und dieser Weg sich im Laufe der Zeit als der von den Ameisen bevorzugte durchsetzt. Dieses Populationsverhalten lässt sich algorithmisch beschreiben und führt zu effizienten Lösungsansätzen für schwer handhabbare Probleme, wie dem des Handelsreisenden. Dabei werden für einen gewissen Zeitraum virtuelle Ameisen auf Reisen zwischen den Städten geschickt, die Markierungen unterschiedlicher Intensität hinterlassen. Anschließend werden Wege mit geringen Markierungen verworfen und so kann häufig eine kurze Rundreise konstruiert werden, die in der Regel jedoch nicht optimal ist. Abbildung 3 zeigt einen so konstruierten Weg.

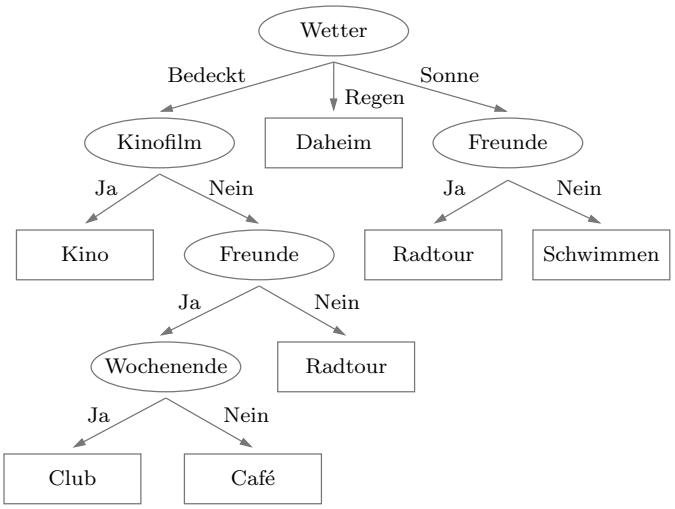

ABBILDUNG 2. Aus (einer Erweiterung von) Tabelle 1 gewonnener Entscheidungsbaum.

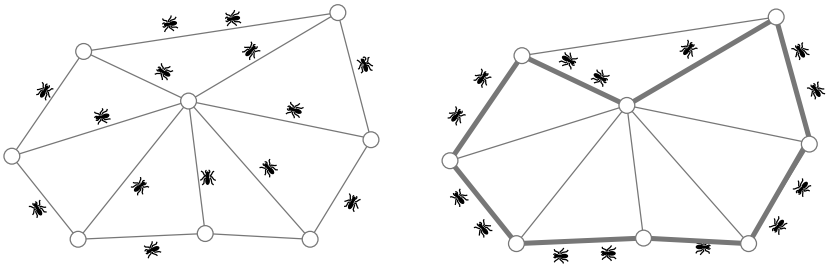

ABBILDUNG 3. Für einige komplexe Problemstellungen führen Ameisenalgorithmen zu qualitativ hochwertigen Ergebnissen.

5.3. **Neuronale Netze.** Bei *neuronalen Netzen* wird die Funktionsweise der Neuronen im menschlichen Gehirn imitiert, indem diese als Schaltzellen mit mehreren Eingangs- und einem Ausgangssignal beschrieben werden, die miteinander vernetzt sind. Das Ausgangssignal ergibt sich dabei als gewichtete Summe der Eingangssignale, wobei die Gewichte in einer Trainingsphase zu bestimmen sind. Besonders gut eignen sich neuronale Netze zur Mustererkennung, das heißt, wenn beliebige Muster mit Hilfe gewisser Referenzmuster klassifiziert werden sollen, wie in Abbildung 4 schematisch dargestellt ist. Die Bestimmung der erforderlichen Koeffizienten erfolgt in einer Trainingsphase, in der bekannte Muster den jeweiligen Referenzmustern zugeordnet werden.

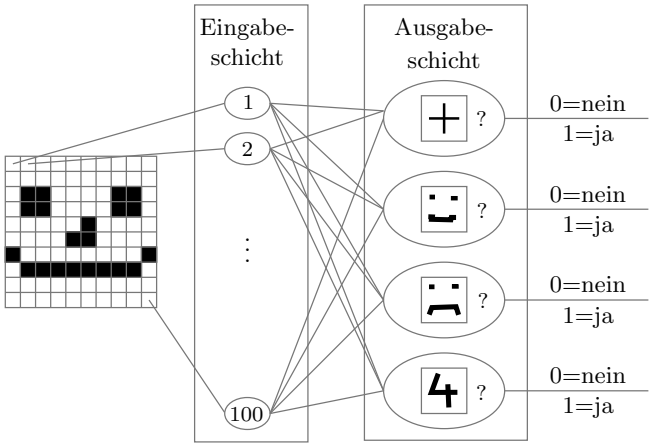

ABBILDUNG 4. Musterklassifizierung mit Hilfe eines neuronalen Netzes. Jedem Bildpixel und jedem Referenzmuster ist ein Neuron zugeordnet.

VI VISUALISIEREN UND PROGRAMMIEREN IN MATLAB

S. BARTELS, 3.7.2018

1. Einordnung und Aufbau

MATLAB steht für *Matrix Laboratory* und ist eine Programmierumgebung, die viele mathematische Operationen wie das effiziente Lösen linearer Gleichungssysteme und gewöhnlicher Differentialgleichungen sowie Methoden zur Eigenwertberechnung von Matrizen bereit stellt. Darüberhinaus gibt es zahlreiche Möglichkeiten zur Visualisierung mathematischer Objekte. Im Gegensatz zu C++ müssen Programme nicht kompiliert werden und die Verwendung von Listen und Feldern ist deutlich einfacher. Insbesondere müssen Variablentypen nicht spezifiziert werden, sondern werden automatisch angepasst. Ein Nachteil ist, dass die Laufzeit von Programmen gerade bei der Verwendung von Schleifen meist länger als in kompilierten Programmiersprachen ist. Daher ist es oft sinnvoll, Probleme mit einem C++-Programm zu lösen, die Daten in einer Datei abzuspeichern und schließlich mit MATLAB die Datei auszulesen und die Daten graphisch aufzubereiten. Die Programmierumgebung besteht aus verschiedenen Teilen. Zentrale Bestandteile sind das Eingabefenster, der Editor und die Hilfefunktion, Abbildung 1 zeigt die Standardoberfläche von MATLAB.

1.1. **Eingabefenster.** Im Eingabefenster oder *Command Window* können MATLAB-Befehle direkt und interaktiv eingegeben und ausgeführt werden. Auch einige Unix-Befehle für Dateioperationen können wie in einer Konsole ausgeführt werden. Werden Befehle nicht mit einem Semikolon abgeschlossen, so wird das Ergebnis der Berechnung direkt angezeigt. Mit `clc` wird das Fenster geleert und mit `clear` die aktuellen Variablen gelöscht.

1.2. **Editor.** Im Editor werden Programme geschrieben, die dann aus dem Eingabefenster heraus gestartet werden können. Der von MATLAB bereitgestellte Editor verfügt über ein hilfreiches Syntax-Highlighting sowie Einrückfunktionen zur Verbesserung der Übersichtlichkeit. Der Editor kann über das Schaltsymbol *New Script* geöffnet werden.

1.3. **Hilfe.** Erklärungen und Beispiele zu MATLAB-Befehlen findet man über den entsprechenden Menupunkt oder über die Befehle `help` und `doc`.

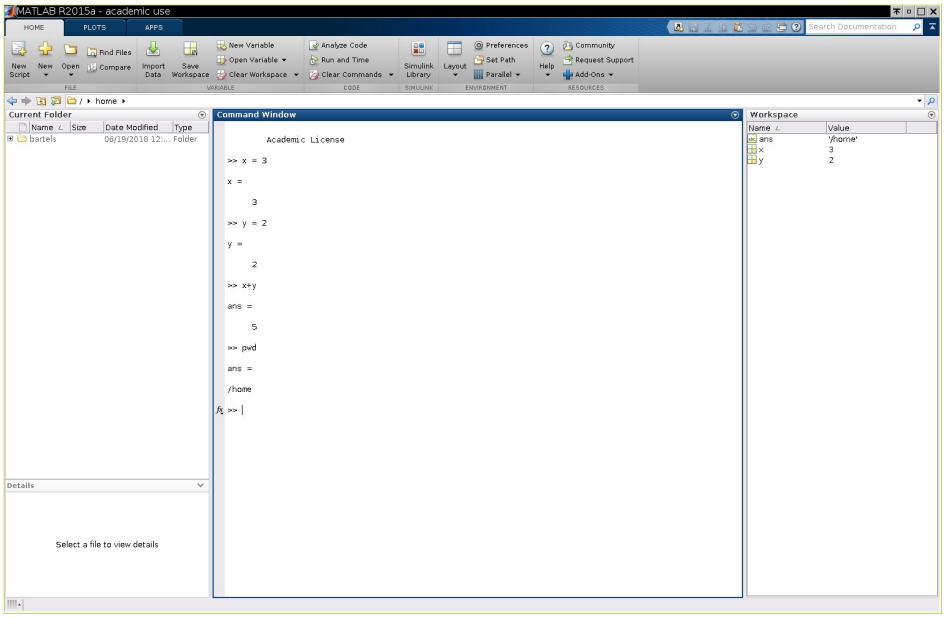

ABBILDUNG 1. Standardoberfläche von MATLAB mit dem Eingabefenester (*Command Window*) in der Mitte.

2. Matrizen, Vektoren und Listen

Wichtigstes Konzept von MATLAB ist die Verwendung mehrdimensionaler Felder, die auf unterschiedliche Weisen benutzt werden. Wir verwenden die folgende Terminologie spezieller Felder:

- Unter *Matrizen* werden zweidimensionale Felder verstanden, deren Einträge meist Gleitkommazahlen sind.
- Unter *Vektoren* werden Matrizen verstanden, die nur aus einer Zeile oder einer Spalte bestehen.
- Unter *(Index-) Listen* werden Vektoren verstanden, die nur positive ganzzahlige Einträge oder nur boolesche Werte besitzen.

Man beachte, dass Vektoren und Listen spezielle Matrizen sind.

2.1. **Erzeugung von Matrizen.** Eine Matrix kann direkt durch eckige Klammern und die Einträge definiert werden, wobei Einträge einer Zeile mit Kommas und verschiedene Zeilen durch Semikolons getrennt werden. Durch den Befehl

```
A = [1.1,2.2;3.3,4.4;5.5,6.6];
```

wird eine Matrix `A` mit 3 Zeilen und 2 Spalten definiert. Das trennende Komma zwischen Zeileneinträgen kann durch ein oder mehrere Leerzeichen ersetzt werden. Spezielle Matrizen wie die Einheitsmatrix oder Matrizen, die in jedem Eintrag den Wert 0 oder 1 haben, können mit den Befehlen `eye` sowie `zeros` und `ones` definiert werden:

```
E = eye(5);
Z = zeros(2,4);
A = ones(5,3);
```

Listen und Vektoren mit gleichen Abständen zwischen den Einträgen lassen sich durch Angabe des Abstands oder der Anzahl der Einträge mit den Kommandos

```
I = i_a:incr:i_b;
X = linspace(a,b,N);
```

generieren. Durch den ersten Befehl wird eine Liste mit Einträgen erzeugt, die bei `i_a` beginnen und in Abständen von `incr` bis maximal `i_b` gehen, wobei `incr` auch negativ sein darf. Im zweiten Fall ist das Inkrement gegeben durch $(b-a)/(N-1)$. Beispielsweise werden durch die Kommandos

```
I = 1:2:9;
X = 0.0:.1:1.0;
Y = linspace(0,1,11);
```

eine Indexliste `I` mit Einträgen 1, 3, 5, 7, 9 sowie identische Zeilenvektoren `X` und `Y` mit den elf Einträgen 0.0, 0.1, 0.2, ..., 1.0 generiert.

2.2. **Rechnen mit Matrizen.** Die komponentenweise Addition und Subtraktion für Matrizen gleicher Größe lassen sich direkt angeben:

```
J = [1,2,3] + [4,5,6];
Y = [1.1;2.2] - [0.2;0.3];
```

Die Multiplikation von Matrizen entspricht der gewöhnlichen Matrixmultiplikation und muss wohldefiniert sein, das heißt die Größen der beteiligten Matrizen müssen kompatibel sein, durch

```
b = [1.0,2.0,3.0;4.0,5.0,6.0] * [1.0;2.0;3.0];
```

wird ein Vektor `b` mit den Einträgen 14.0 und 32.0 definiert. Die Indizierung von Matrizen beginnt in MATLAB mit dem Index 1, der Zugriff auf Einträge erfolgt durch Verwendung runder Klammern:

```
A = [1,2;3,4];
det_A = A(1,1)*A(2,2)-A(1,2)*A(2,1);
```

Mit zulässigen Indexlisten kann man auf Teilmatrizen zugreifen, durch

```
A = [1,2;3,4;5,6]; I = [1,2]; J = 2; B = A(I,J);
```

wird eine Matrix `B` mit zwei Zeilen und einer Spalte und den Einträgen 2 und 4 definiert. Wenn auf alle Zeilen oder Spalten zugegriffen werden soll, kann ein Doppelpunkt verwendet werden, durch

```
A = [1,2;3,4;5,6]; I = [1,2]; B = A(I,:);
```

wird eine Matrix `B` mit zwei Zeilen und zwei Spalten und den Einträgen 1, 2, 3 und 4 definiert. Die Tranposition einer Matrix erfolgt mit einem Apostroph:

```
A = [1,2;3,4;5,6]; B = A';
```

Algebraische Operationen können komponentenweise angewendet werden, um zwei Listen oder Matrizen gleicher Größe zu verknüpfen. Durch

```
I = [1,2]; J = [3,4]; K = I.*J;
```

wird eine Liste K mit den zwei Einträgen 3 und 8 erzeugt. Auf Matrizen können Funktionen angewendet werden, was in der Regel komponentenweise geschieht, beispielsweise erhält man mittels

```
X = 0.0:0.01:1.0;
Y = sin(X);
```

einen Vektor `Y` der die Werte der Sinusfunktion in den Punkten $0.0, 0.1, \ldots, 1.0$ enthält. Tabelle 1 zeigt einige wichtige Kommandos zum Arbeiten mit Matrizen, Vektoren und Listen.

`[a,b,...;x,y,...]`	Definition eines Arrays
`[a,b,...],[x;y;...]`	Definition eines Zeilen- oder Spaltenvektors
`A(i,j)`, `I(j)`	Zugriff auf die Einträge eines Arrays
`a:b`, `a:step:b`	Liste von a bis b mit Schrittweite 1 oder *step*
`linspace(a,b,N)`	Äquidistante Partitionierung des Intervalls $[a, b]$
`A(i,:)`, `A(:,j)`	i-te Zeile und j-te Spalte von A
`A(I,J)`	Teilmatrix definiert durch Listen I und J
`ones(n,m)`	Array mit Einträgen 1
`zeros(n,m)`	Array mit Einträgen 0
`randn(n,m)`	Matrix mit zufällig generierten Einträgen
`A'`	Transponierte Matrix
`A+B`, `A-B`, `A*B`	Addition, Subtraktion und Produkt von Matrizen
`sort(A)`	Sortierung der Einträge eines Arrays
`sum(A,1)`, `sum(A,2)`	Spalten- und zeilenweise Summenbildung
`max(A)`, `min(A)`	Spaltenweise Extremwerte eines Arrays
`size(A)`, `length(I)`	Dimensionen eines Arrays und Länge einer Liste
`find(A)`	Finden der Nichtnulleinträge einer Matrix
`unique(I)`	Eliminieren doppelter Einträge einer Liste

Tabelle 1. Erstellung und Manipulation von Matrizen.

3. Plotten von Funktionen und Vektorfeldern

3.1. **Graphen eindimensionaler Funktionen.** Zur grafischen Darstellung eindimensionaler Funktionen eignet sich der Befehl `plot`. Dazu werden zwei Vektoren mit Argumenten und zugehörigen Funktionswerten benötigt. Die Routine erzeugt dann einen Polygonzug durch die damit definierten Punkte. Mit weiteren optionalen Angaben kann die Darstellung der Kurve beeinflusst werden:

```
X = 0:.01:1;
U = sin(X);
```

```
plot(X,U,'-r');
```

Sollen mehrere Graphen in einem Bild erscheinen, bietet sich das Kommando `hold on|off` an, welches vermeidet, dass die Grafik bei einem neuen Aufruf von `plot` gelöscht wird. Mit dem Befehl `legend` kann eine Legende erzeugt werden:

```
X = 0:.01:1;
U = sin(X); plot(X,U,'r-'); hold on;
V = cos(X); plot(X,V,'b-');
W = X.^2;   plot(X,W,'k-'); hold off;
legend('sin','cos','x^2');
```

Weitere hilfreiche Befehle zur Bearbeitung der Grafik finden sich in Tabelle 2.

3.2. **Darstellung von Kurven.** Eine Kurve, das heißt eine Abbildung $c : [a, b] \to \mathbb{R}^3$ mit einem Parameterintervall $[a, b] \subset \mathbb{R}$, lässt sich mit Punkten im Intervall $[a, b]$ und drei Vektoren, die die drei Komponenten der entsprechenden Funktionswerte enthalten, mit dem Befehl `plot3` darstellen. Für die Helix $t \mapsto (\sin(t), \cos(t), 5t)$, $t \in [0, 10\pi]$ erfolgt dies beispielsweise mit folgenden Kommandos:

```
T = [0:.01:10*pi];
C1 = sin(T); C2 = cos(T); C3 = 5*T;
plot3(C1,C2,C3);
```

Man beachte, dass die Punkte im Parameterbereich nicht für den Aufruf der Routine `plot3` benötigt werden.

3.3. **Graphen zweidimensionaler Funktionen.** Die Visualisierung von Funktionen die in zweidimensionalen Gebieten definiert sind, ist etwas aufwendiger, da ein geeignetes Gitter benötigt wird. Dies lässt sich mit dem Befehl `meshgrid` generieren:

```
[X,Y] = meshgrid(a:dx:b,c:dy:d);
```

Hier sind `X` und `Y` Matrizen, die jeweils zusammengehörende x- und y-Koordinaten enthalten, welche die Gitterpunkte $p_{ij} = (x_{ij}, y_{ij})$ im Rechteck $[a, b] \times [c, d]$ mit Abständen `dx` und `dy` definieren. Mit zugehörigen Funktionswerten $f(p_{ij})$ erlaubt der Befehl `surf` dann eine grafische Darstellung. Die Funktion $f(x, y) = \sin(x)\cos(y)$ kann so im Bereich $[0, 1] \times [\pi/2, 3\pi/2]$ beispielsweise folgendermaßen dargestellt werden:

```
[X,Y] = meshgrid(0:.1:1,pi/2:.1:3*pi/2);
U = sin(X).*cos(Y);
surf(X,Y,U);
colorbar;
```

Das Kommando `colorbar` sorgt für die Einblendung einer Farbskala.

3.4. **Darstellung von Höhenlinien und Vektorfeldern.** Wird mittels des Befehls `meshgrid` und Angabe zugehöriger Funktionswerte eine Gitterfunktion $\phi : Q \to \mathbb{R}$ auf einem Rechteck $Q = [a, b] \times [c, d]$ definiert, so erlaubt die Routine `contour` die Darstellung von Höhenlinien:

```
[X,Y] = meshgrid(0:.1:1,pi/2:.1:3*pi/2);
U = sin(X).*cos(Y);
contour(X,Y,U);
```

Zur Visualisierung von Vektorfeldern, das heißt von vektorwertigen Abbildungen $F : Q \to \mathbb{R}^d$ mit $Q \subset \mathbb{R}^d$ für $d = 2$ oder $d = 3$, eignen sich die Befehle `quiver` und `quiver3`. Dabei werden die Koordinaten von Punkten im Definitionsbereich Q sowie die Komponenten der zugehörigen Funktionswerte mit vier beziehungsweise sechs Vektoren oder Matrizen spezifiziert. Mit den Befehlen

```
[X,Y] = meshgrid(0:.1:1,0:.1:1);
F_1 = sin(Y); F_2 = cos(X);
quiver(X,Y,F_1,F_2);
```

wird beispielswise das zweidimensionale Vektorfeld $F(x, y) = [\sin(y), \cos(x)]$ im Bereich $[0, 1] \times [0, 1]$ grafisch dargestellt. Eine wichtige Klasse von Vektorfeldern sind Gradienten von Funktionen $f : Q \to \mathbb{R}$ also die Abbildung $F = \nabla f : Q \to \mathbb{R}^d$. Für eine Gitterfunktion ϕ wie oben kann ihr Gradient mit der Routine `gradient` approximativ dargestellt werden. Interessant ist beispielsweise die gemeinsame Darstellung von Höhenlinien und dem Gradientenfeld einer Funktion:

```
dx = .05; dy = .05;
[X,Y] = meshgrid(0:dx:1,0:dy:1);
U = sin(X).*cos(Y);
[F_1,F_2] = gradient(U,dx,dy);
contour(X,Y,U); hold on
quiver(X,Y,F_1,F_2); hold off
```

Dabei bestätigt sich die Tatsache, dass Gradienten orthogonal zu Höhenlinien stehen und in Richtung des steilsten Anstiegs zeigen.

3.5. **Abspeichern von Grafiken.** Einfache Grafiken wie Graphen eindimensionaler Funktionen speichert man am besten als `eps`- oder `pdf`-Datei ab, wobei das `eps`-Format für Zwecke der grafischen Nachbereitung besser geeignet ist. Aufwendige Grafiken wie Graphen zweidimensionaler Funktionen sollten speichereffizient zum Beispiel im `jpg`-Format abgespeichert werden. Durch die Befehle

```
figure(1); print -deps einfacher_plot.eps
figure(2); print -djpeg aufwaendiger_plot.jpg
```

wird die im ersten Grafikfenster dargestellten Informationen im `eps`- und die im zweiten im `jpg`-Format abgespeichert.

`disp(A), disp('txt')`	Anzeigen einer Variable und einer Zeichenkette
`input('text')`	Einlesen einer Variable
`%`	Markierung eines Kommentars
`plot(X,Y,'-*')`	Polygonzug durch Punkte $(X(k), Y(k))$ in $\mathbb{R}^2$
`hold on, hold off`	Darstellung mehrerer Objekte in einer Grafik
`mesh(X,Y,Z)`	Darstellung eines zweidimensionalen Graphen
`meshgrid`	Erzeugung eines Gitters
`axis([x1,x2,...])`	Begrenzung des dargestellten Bereichs
`xlabel`, `ylabel`	Beschriftung der Achsen
`legend`	Einfügen einer Legende
`figure(k)`	Öffnen oder Auswahl eines Grafikfensters
`clf`	Leeren des aktuellen Grafikfensters
`subplot(n,m,j)`	Darstellung mehrerer Plots in einem Fenster
`quiver`, `quiver3`	Visualisierung von Vektorfeldern
`print`	Exportieren einer Grafik

TABELLE 2. Darstellung und Bearbeitung grafischer Objekten.

4. Programmieren in Matlab

4.1. **Programmdateien.** MATLAB-Programme, die auch als M-Skripte bezeichnet werden, sind Folgen von MATLAB-Kommandos, die in einer Datei mit der Endung `.m` abgespeichert werden also beispielsweise `matlab_prog.m`. Solche Programme können aus dem Eingabefenster heraus durch Angabe des Dateinamens ohne die Endung gestartet werden, sofern man sich im entsprechenden Verzeichnis befindet. Alternativ kann ein Programm aus der Menüleiste heraus gestartet werden. Das folgende Beispielprogramm wird aus dem Eingabefenster durch Eingabe von `beispiel` gestartet:

```
% Programm beispiel.m, Start durch Eingabe "beispiel"
x = 1;
y = 2;
disp('Der Quotient von x und y ist');
disp(x/y)
```

Im Gegensatz zu C++ liefert es das Resultat 0.5. Laufende MATLAB-Programme können im Eingabefenster durch die Tastenkombination `Ctrl-C` abgebrochen werden.

4.2. **Variablen.** In MATLAB stehen die üblichen Variablentypen wie `int`, `double` und `logical` zur Verfügung. Eine Deklaration ist nicht erforderlich, diese erfolgt automatisch bei der Definition von Variablen. Darüberhinaus wird der Typ automatisch an den Wert eines Ausdrucks angepasst. In den Zeilen

```
k = 1;
k = k/2;
```

```
str = 'zeichenkette';
```

wird `k` bei Definition als Variable vom Typ `double` festgelegt und hat nach dem zweiten Kommando den Wert 0.5. Mit den arithmetischen Operationen und Vergleichen sowie konstanten Werten, die für die verschiedenen Variablen definiert sind, lassen sich Ausdrücke bilden, die dann Variablen zugewiesen werden können. Eine Auswahl der wichtigsten Operationen und konstanten Werte sowie mathematischer Grundfunktionen ist in Abbildung 3 aufgeführt.

`+,-,*,/`	Arithmetische Grundoperationen
`a==b`, `a~=b`	Logischer Test auf Gleich- oder Ungleichheit
`a<b`, `a<=b`	Logischer Vergleich zweier Variablen
`b1&&b2`, `b1||b2`, `~b`	Logisches UND und ODER sowie NEGATION
`true, false`	Logische Werte wahr und falsch
`sqrt(x)`, `x^y`	Quadratwurzel und Potenzen
`exp(x)`, `ln(x)`	Exponentialfunktion und Logarithmus
`sin(x)`, `cos(x)`, `pi`	Trigonometrische Funktionen und Konstante π
`norm(x,p)`	p-Norm eines Vektors

TABELLE 3. Grundoperationen, Vergleiche und elementare Funktionen in MATLAB.

4.3. **Kontrollstrukturen.** In MATLAB stehen die üblichen Kontrollstrukturen zur Verfügung und die Syntax einer `if`-Abfrage sowie einer `for`- und einer `while`-Schleife sind folgendermaßen definiert:

```
if b1 block1 elseif b2 block2 ... else blockn end
while b block end
for var = Z block end
```

Dabei sind `b1`, `b2` und `b` boolesche Variablen beziehungsweise Ausdrücke und `block1`, ... `blockn` sowie `block` Blöcke von Kommandos, also Folgen von Zuweisungen, Funktionsaufrufen oder Kontrollstrukturen. In der `for`-Schleife ist `Z` eine Indexliste oder ein Vektor, der von der Schleifenvariable `var` durchlaufen wird. Besonders zu beachten ist der Befehl `elseif`, der eine Schachtelung mehrerer `if`-Abfragen vermeidet. In Abbildungen 2, 3 und 4 sind entsprechende Beispielprogramme gezeigt.

4.4. **Funktionen.** In MATLAB lassen sich Funktionen definieren, deren syntaktische Struktur folgende Form besitzt:

```
function [val1,...,valm] = funktions_name(arg1,...,argn)
block
end
```

```
% M-Skript if_abfrage.m
x = input('x = ');
if x < 0
    disp('x ist kleiner als Null');
elseif x > 0
    disp('x ist groesser als Null');
else
    disp('x ist Null');
end
```

ABBILDUNG 2. MATLAB-Programm zur Bestimmung des Vorzeichens einer Zahl.

```
% M-Skript while_schleife.m
x = 1;
while 1+x > 1
    x = x/2;
end
disp(x);
```

ABBILDUNG 3. MATLAB-Programm zur Bestimmung der Differenz zwischen der Zahl 1 und der nächstgrößeren Gleitkommazahl.

```
% M-Skript for_schleife.m
J = 1:2:10;
sum = 0;
for j = J
    sum = sum+j;
end
disp(sum);
```

ABBILDUNG 4. MATLAB-Programm zur Bestimmung der Summe der ungeraden Zahlen zwischen 1 und 10.

Dabei ist das schließende `end` optional. Die Verwendung von Rekursionen ist in MATLAB zulässig. Hervorzuheben ist die Möglichkeit der Verwendung mehrerer Rückgabewerte; die zugehörigen Variablen müssen im Befehlsblock definiert werden. Funktionen werden über den Dateinamen aufgerufen, sodass es sinnvoll erscheint, als Dateinamen den Funktionsnamen mit der Endung `.m` zu verwenden. Abbildung 5 zeigt die Bestimmung der Länge eines Vektors in einer Funktion, ihr Aufruf erfolgt zum Beispiel mittels `laenge = veknorm([1,2,3])` im Eingabefenster oder einem anderen Programm.

```
% M-Skript veknorm.m
function val = veknorm(z)
val = 0.0;
for j = 1:length(z)
    val = val+z(j)^2;
end
val = sqrt(val);
% end
```

ABBILDUNG 5. Funktion zur Bestimmung der euklidischen Länge eines Vektors.

Es können mehrere Funktionen in einer Datei abgespeichert werden, wobei nur auf die zuerst aufgeführte von außen zugegriffen werden kann. Die anderen dienen dann als Unterfunktionen für die erste Funktion. Einfache Funktionen können auch in einem *Inline*-Format definiert werden, was besonders für die Definition im Eingabefenster relevant ist:

```
funkt_name = @(arg1,...,argn) ausdruck
```

Die Definition einer Funktion zur Berechnung der euklidischen Länge eines zweidimensionalen Vektors kann so beispielsweise über die Anweisung

```
f_el_2D = @(x,y) (x^2+y^2)^(1/2)
```

Der Aufruf erfolgt dann zum Beispiel mit `f(1.0,2.0)`.

5. WICHTIGSTE KONZEPTE

Wir fassen die wichtigsten Regeln, die bei dem Arbeiten mit MATLAB zu beachten sind, noch einmal zusammen:

- Variablennamen bestehen aus Buchstaben, Ziffern und Unterstrichen, wobei das erste Zeichen keine Ziffer sein darf und der Variablenname nicht nur aus einem Unterstrich bestehen darf.
- Die Verwendung des Variablennames `i` sollte vermieden werden, da dieser standardmäßig für die imaginäre Einheit vorgesehen ist.
- Es wird zwischen Groß- und Kleinbuchstaben unterschieden, das heißt beispielsweise, dass `Hausnr` und `HausNr` zwei unterschiedliche Variablen sind.
- Schlüsselwörter wie `if`, `elseif`, `else`, `while`, `true` oder `false` dürfen nicht als Variablen- oder Funktionsnamen verwendet werden.
- Werden Zuweisungen nicht mit einem Semikolon beendet, so wird der Wert des Ausdrucks auf dem Bildschirm angezeigt.
- Bei der Definition von Matrizen werden Einträge in einer Zeile mit einem Komma oder Leerzeichen und verschiedene Zeilen durch ein Semikolon getrennt.
- Die Indizierung von Listen und Matrizen beginnt mit dem Index 1.

- Variablentypen werden stets an den Typ einer Berechnung angepasst, das heißt, dass (im Gegensatz zu C++) keine Typkonvertierung und somit Rundung eines zugewiesenen Ausdrucks stattfindet.
- Mit Indexlisten kann auf Teilmatrizen zugegriffen werden.

6. Weiterführende Aspekte

6.1. **Numerische Mathematik.** Effiziente Realisierungen gewisser Standardverfahren der numerischen Mathematik sind in MATLAB verfügbar und intuitiv benutzbar. Beispielsweise erfolgt die Berechnung der Determinante, die Bestimmung von Eigenvektoren und -werten sowie das Aufstellen der Inversen einer quadratischen Matrix `A` über die folgenden Anweisungen:

```
val = det(A);
[V,D] = eig(A);
A_inv = inv(A);
```

Zur Lösung eines linearen Gleichungssystems $Ax = b$ sollte auf die Verwendung der Inversen verzichtet werden und stattdessen der *Backslash*-Operator benutzt werden:

```
x = A\b;
```

Dadurch wird ein effizienteres und stabileres Verfahren zur Lösung herangezogen. Die approximative Integration einer Funktion kann durch die Kommandos

```
f = @(x) 1./(1+x.^2);
int_f = quad(f,-1,1);
```

erfolgen. Dabei ist zu beachten, dass die zu integrierende Funktion in vektorieller Form realisiert ist, das heißt für einen Vektor als Argument einen Vektor gleicher Länge mit den entsprechenden Funktionswerten zurückgibt. Das approximative Lösen einer gewöhnlichen Differentialgleichung $y'(t) = f(t, y(t))$, $y(0) = y_0$, kann mit der MATLAB-Routine `ode45` erfolgen:

```
T = 10; y_0 = 1;
f = @(t,y) cos(2*t)*y^2;
[t_list,y_list] = ode45(f,[0,T],y_0);
plot(t_list,y_list);
```

Die Routine gibt einen Vektor von Zeitpunkten im Intervall $[0, T]$ und die zugehörigen Funktionswerte einer Näherungslösung $\widetilde{y}$ zurück, die anschließend grafisch dargestellt werden.

6.2. **Vektorisierung.** Die Verwendung von Schleifen kann in MATLAB gelegentlich zu Laufzeitproblemen führen. Sofern es möglich ist, sollte die entsprechende Berechnung mittels Vektoroperationen durchgeführt werden, was als *Vektorisierung* bezeichnet wird. Als Beispiel betrachten wir die Berechnung des Skalarprodukts zweier großer Vektoren:

```
n = 1e8; x = rand(n,1); y = rand(n,1);
tic; val = 0; for j = 1:n val = val+x(j)*y(j); end; toc
```

```
tic; val = sum(x.*y); toc
```

Die Befehle `tic` und `toc` erlauben eine Zeitmessung und es stellt sich heraus, dass die zweite Berechnung des Skalarprodukts etwa viermal schneller ist als die erste.

6.3. **Laden und Speichern von Daten.** Für das Laden und Speichern von Daten existieren zwei Möglichkeiten. Einerseits können Daten aus Textdateien, das heißt im lesbaren `ascii`-Format gelesen und abgespeichert werden, was über die Befehle

```
A = load('test_ascii.dat');
save test_ascii.dat A -ascii;
```

erfolgen kann. Dies ist besonders nützlich, wenn Daten mit anderen Programmen anderer Programmiersprachen erzeugt worden sind. Sollen in MATLAB generierte Variablen abgespeichert und später in MATLAB wieder eingeladen werden, bietet sich das MATLAB-eigene `mat`-Format an:

```
save test_mat.mat A B c d x;
load test_mat.mat;
```

Dadurch werden die gespeicherten Daten beim Laden wieder den zuvor benutzten Variablen zugewiesen.

VII FUNKTIONSWEISE EINES COMPILERS

S. BARTELS, 10.7.2018

1. Umwandlung lesbarer Programme

Programmiersprachen wie MATLAB und C++ erlauben eine hohe Problemabstraktion durch die Verwendung von Funktionen, Schleifen und Rekursionen. Diese Konstrukte müssen vom Compiler in für den Prozessor verarbeitbare Instruktionen übersetzt werden. Dieser kann jedoch kaum mehr als Informationen aus Registern laden und einfache arithmetische und logische Verknüpfungen sowie Fallunterscheidungen durchführen. Entsprechende Prozessor-Anweisungen werden durch Folgen von Nullen und Einsen codiert, die für Menschen nur mit hohem Aufwand verständlich und zudem sehr fehleranfällig sind. Die Aufgabe von Compilern ist es, verständliche, lesbare Anweisungen in solche Codes zu übersetzen. Dies ist jedoch mit einigen Schwierigkeiten verbunden, da sichergestellt werden muss, dass Befehle eindeutig definiert sind. Wie schwierig korrekte Spracherkennung im Allgemeinen sein kann, zeigt bereits der mehrdeutige Satz *Der Fahrer muss das Hindernis umfahren*, bei dem die Bedeutung nur aus dem Kontext oder einer geeigneten Betonung hervorgeht.

2. Beispiel eines Maschinencodes

Um die Struktur eines Maschinencodes zu illustrieren, folgen wir einem Beispiel aus dem Buch *Computer* von R. Drechsler, A. Fink und J. Stoppe (Springer, 2017) und betrachten einen 8-Bit Modellprozessor, das heißt, ein Prozessor, der aus 8 Bits bestehende Anweisungen verarbeiten kann. Das betrachtete Maschinencode-Programm besteht aus den folgenden 64 Bits:

00010110 00100011 00010010 00100100
00000011 01000100 00100101 11000000

Das Programm definiert eine Folge von acht Anweisungen und jede davon hat die Struktur:

$$\underbrace{b_0\, b_1\, b_2}_{\text{Befehl}} \quad \underbrace{b_3}_{\text{Nummer}} \quad \underbrace{b_4\, b_5\, b_6\, b_7}_{\text{Operand}}$$

Dabei wählen die Bits b_0, b_1, b_2 einen von acht Befehlen aus und Bit b_3 legt fest, ob die Bits $b_4 b_5 b_6 b_7$ als Zahl oder Adresse eines Registers interpretiert werden sollen. Wir nehmen an, dass die von 0 bis 7 numerierten Befehle gegeben sind durch folgende Operationen:

`LOAD, STORE, ADD, SUB, COMP, JUMP, HALT, NOOP`

Der Befehl `LOAD` kann beispielsweise eine Zahl aus einem bestimmten Register in das Arbeitsregister laden. Anschließend kann mittels `ADD` eine konkrete Zahl zum Wert des Arbeitsregister addiert werden. Die resultierenden Bedeutungen der Befehle des obigen Maschinencodes sind in Tabelle 1 erklärt. Es stellt sich heraus, dass die Rechnung $6 + 2$ durchgeführt wird, die sich auch mit weniger Befehlen realisieren ließe.

8-Bit-Wort	Befehl	Nr.	Op.	Interpretation
00010110	`LOAD`	1	6	Lade Zehl 6 ins Arbeitsregister (AR)
00100011	`STORE`	0	3	Speichere Wert des AR in Register 3
00010010	`LOAD`	1	2	Lade Zahl 2 ins AR
00100100	`STORE`	0	4	Speichere Wert des AR in Register 4
00000011	`LOAD`	0	3	Lade Wert des Registers 3 ins AR
01000100	`ADD`	0	4	Addiere Wert des Registers 4 zum Wert des AR
00100101	`STORE`	0	5	Speichere Wert des AR in Register 5
11000000	`HALT`	0	0	Stoppe

TABELLE 1. Interpretation eines Beispielmaschinencodes.

Deutlich übersichtlicher wird das Beispielprogramm in sogenannter Assemblersprache. Dabei können die Befehle als Begriffe angegeben und mit dem Symbol # Zahlen von Adressen unterschieden werden:

```
LOAD #6
STORE 3
LOAD #2
STORE 4
LOAD 3
ADD 4
STORE 5
HALT
```

Assemblersprachen definieren eine Zwischenstufe zwischen Maschinencodes und Programmen höherer Programmiersprachen wie MATLAB und C++.

3. Höhere Programmiersprachen

Unter *höheren Programmiersprachen* versteht man solche, die im Vergleich zum Maschinencode eine höhere Abstraktion durch Schleifen, Funktionen und Rekursionen erlauben. Insbesondere hängen sie nicht von den besonderen Eigenschaften des verwendeten Rechners ab, beispielsweise was die Speicherverwaltung betrifft. Übersetzer oder *Compiler* erzeugen aus einem Programm einer hóheren Programmiersprache wie C++ einen Assembler- beziehungsweise Maschinencode. Die Programmiersprache C++ ist dabei sehr maschinennah, da sie beispielsweise durch den Einsatz von Zeigern einen sehr direkten Zugriff auf den Speicher ermöglicht. Im Gegensatz dazu wird

MATLAB nicht als Compiler- sondern als Interpretersprache angesehen. Vereinfacht dargestellt werden dabei bereits übersetzte Programme aufgerufen und weitere Teile des Programms wie Schleifen erst bei ihrem Auftreten und abhängig von den Eingabewerten übersetzt. Durch den Einsatz sogenannter *virtueller Maschinen*, die eine zusätzliche Ebene zwischen Programmiersprache und Maschinencode schaffen, verschmelzen die Konzepte von Compiler- und Interpretersprachen zunehmend. Dadurch ist es insbesondere möglich, auch in Interpretersprachen Konzepte wie Rekursion anzuwenden. Noch etwas weiter geht die Methodik der *Just-In-Time*-Compiler, die ein Programm analysieren und abhängig von Eingaben teilweise übersetzen oder bereits übersetzten Code wiederverwenden, und bei Programmiersprachen wie JavaScript und neueren Versionen von MATLAB zum Einsatz kommt. Eine Gegenüberstellung der Vor- und Nachteile klassischer Compiler- und Interpretersprachen findet sich in Tabelle 2.

Compilersprache (C++)	Interpretersprache (MATLAB)
⊕ schnelle Programme ⊕ Flexibilität durch direkten Zugriff auf Speicheradressen ⊕ Übersetzung unabhängig von Eingabedaten ⊖ lange Übersetzungsphase ⊖ kompliziert und fehleranfällig	⊕ direkte Nutzung, keine Übersetzung ⊕ keine Deklaration von Variablen ⊕ einfache Programme ⊖ langsame Schleifen ⊖ keine explizite Speicherverwaltung ⊖ Übersetzung abhängig von Eingabe

TABELLE 2. Vor- und Nachteile von Compiler- und Interpretersprachen am Beispiel von C++ und MATLAB.

4. DER COMPILER

Zur Beschreibung der Arbeitsweise eines Compilers folgen wir in diesem Abschnitt der Darstellung des Vorlesungsskripts *Compilerbau* von U. Goltz, T. Gehrke und M. Lochau (TU Braunschweig, 2010) Ein Compiler, der aus einem Quellprogramm einen Maschinencode erzeugt, arbeitet zusammen mit einem *Präprozessor*, der aus dem rohen Quellcode Makros ersetzt und Kommentare entfernt, einem *Assembler*, der aus Assemblercode relokatiblen, das heißt verschiebbaren Maschinencode generiert, sowie einem *Binder*, der konkrete Sprungadressen einfügt und somit einen ausführbaren Machinencode erstellt. Diese Schritte sind in Abbildung 1 schematisch dargestellt.
Die Übersetzung eines Programms durch den Compiler erfolgt in einer Analyse- und einer Synthesephase, wobei die Analysephase aus der lexikalischen,

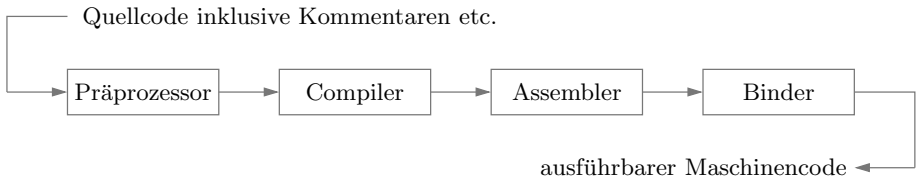

ABBILDUNG 1. Schritte der Übersetzung eines Programms.

der syntaktischen und der semantischen Analyse besteht. In der lexikalischen Analyse, die durch den *Scanner* durchgeführt wird, wird die gegebene Zeichenkette in Bestandteilte zerlegt und eine Symboltabelle angelegt. Variablen, die in diesem Kontext oft als Bezeicher betitelt werden, werden dabei numeriert. In der syntaktischen Analyse, die vom *Parser* durchgeführt wird, wird analysiert, ob beispielsweise Terme korrekt sind. Dazu wird ein Strukturbaum, auch als Syntaxbaum bezeichnet, erstellt, der die verwendeten Operationen widergibt. In der semantischen Analyse werden die auftretenden Bezeichner mit Attributen wie Variablentyp und Gültigkeitsbereich versehen und die Wohldefiniertheit der verwendeten Operationen geprüft. Dies garantiert die statische semantische Korrektheit, das heißt die formale Wohlgestelltheit der Berechnungen unabhängig von konkreten Eingabedaten, die möglicherweise zu dynamischen Semantikfehlern wie Division durch Null führen können. Die anschließende Synthese-Phase besteht aus verschiedenen Schritten wie der Zwischencode-Erzeugung, einer Code-Optimierung und der Maschinencode-Generierung. Im Zwischencode werden beispielsweise Formeln mit Hilfsvariablen in ihre Bestandteile zerlegt und als Folgen einfacher Dreiadressbefehle der Form

```
tmp_l = id_j op id_k;
```

dargestellt. Die darauf folgende Codeoptimierung entfernt überflüssige Zuweisungen. Bei der finalen Maschinencode-Erzeugung werden die einfachen Befehle des optimierten Assemblercodes in Maschinenbefehle übersetzt. Diese letzte Phase der Übersetzung ist von dem verwendeten Rechner abhängig und damit Teil der *Back-End-Phase*, die im Gegensatz zur *Front-End-Phase* den maschinenabhängigen Teil der Übersetzung bezeichnet. Die schrittweise Übersetzung der Zuweisung $position := initial + rate * 60$ in Assemblercode ist in Abbildung 2 dargestellt.

5. Methodik des Scanners

Die lexikalische Analyse eines Quellprogramms basiert auf der Theorie regulärer Sprachen. Eine Sprache ist dabei eine Menge L von Wörtern über einem Alphabet Σ beispielsweise

$$\Sigma = \{a, b, c\}, \quad L = \{a, b, ab, ac, cba\}.$$

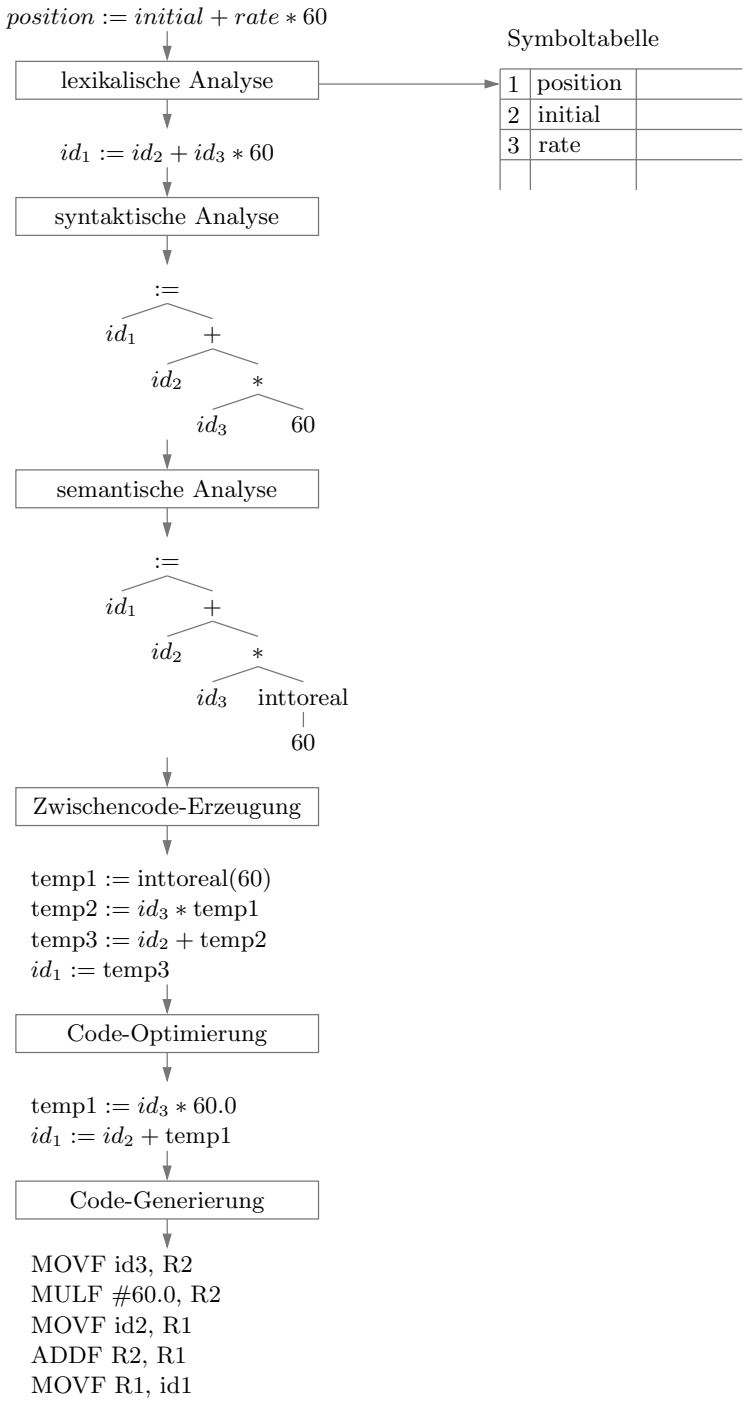

ABBILDUNG 2. Schrittweise Übersetzung einer Zuweisung.

Reguläre Sprachen sind solche, die sich aus aus der leeren Menge und den einelementigen Sprachen durch Vereinigung, Konkatenation und Exponenzieren bilden lassen. Mit regulären Ausdrücken können reguläre Sprachen durch Formeln definiert werden. Als Operationen sind dabei Auswahl, Konkatenation, und Exponenzieren mit den im Alphabet vorhandenen Buchstaben erlaubt. Zusätzlich steht das leere Wort ε zur Verfügung, das beispielsweise neutral bezüglich Konkatenation ist.

Beispiele 5.1. *(i) Sei $\Sigma = \{a, b\}$. Die Auswahl $a|b$ erzeugt $L = \{a, b\}$, die Konkatenation ab erzeugt $L = \{ab\}$, das Exponenzieren a^* erzeugt $L = \{\varepsilon, a, aa, aaa, \dots\} = \{a^n : n \geq 0\}$.*
(ii) Mit $\Sigma = \{a, b, \dots, z, 0, 1, \dots, 9\}$ und dem regulären Ausdruck

$$(a|b|\dots|z)(a|b|\dots|z|0|1|\dots|9)^*$$

wird eine Menge zulässiger Variablennamen definiert, die mit einem Buchstaben beginnen.

Durch reguläre Ausdrücke definierte reguläre Sprachen lassen sich algorithmisch erkennen.

Definition 5.2. *Ein* nicht-deterministischer endlicher Automat *(NEA) ist ein Tupel $M = (\Sigma, Q, \Delta, q_0, F)$ mit den folgenden Eigenschaften:*
(i) Σ ist ein endliches Alphabet,
(ii) Q ist eine endliche Zustandsmenge,
(iii) $q_0 \in Q$ ist ein Anfangszustand,
(iv) $F \subset Q$ ist eine Menge von Endzuständen,
(v) $\Delta \subset Q \times \big(\{\varepsilon\} \cup \Sigma\big) \times Q$ ist eine Übergangsrelation.
Ein NEA heißt deterministischer endlicher Automat *(DEA), wenn die Übergangsrelation eine Funktion $\Delta : Q \times \big(\{\varepsilon\} \cup \Sigma\big) \to Q$ ist.*

Ein Übergang von einem Zustand q und einem Wort zw zu einem neuen Zustand q' mit Wort w ist zulässig, wenn $(q, z, q') \in \Delta$ gilt beziehungsweise im deterministischen Fall $\Delta(q, z) = q'$ gilt. Die von einem NEA akezptierte Sprache ist die Menge aller Wörter w_0, die sich in endlich vielen zulässigen Übergängen vom Anfangszustand q_0 auf das leere Wort ε und einen Endzustand $q_f \in F$ reduzieren lassen:

$$(q_0, w_0) \mapsto (q_1, w_1) \mapsto \dots \mapsto (q_f, \varepsilon),$$

wobei jeweils $w_{k-1} = zw_k$ mit $z \in \Sigma \cup \{\varepsilon\}$ gelte. Der folgende Satz stellt eine Beziehung zwischen regulären Sprachen und Automaten her.

Satz 5.3. *(i) Zu jedem regulären Ausdruck r existiert ein NEA, der die von r definierte reguläre Sprache $L(r)$ akzeptiert.*
(ii) Wird die reguläre Sprache L von einem NEA akzeptiert, so existiert ein DEA, der L akzeptiert.

Automaten werden am übersichtlichsten durch Graphen dargestellt. Die Menge der Variablennamen über dem Alphabet $\Sigma = \{a, b, \dots, z, 0, 1, \dots, 9\}$,

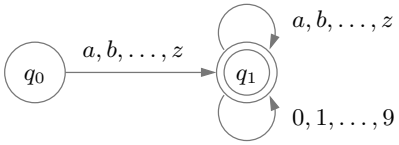

ABBILDUNG 3. Deterministischer endlicher Automat, der eine Menge zulässiger Variablennamen erkennt.

die mit einem Buchstaben beginnen, werden von dem in Abbildung 3 definierten Automaten mit $Q = \{q_0, q_1\}$ und $F = \{q_1\}$ akzeptiert.
Für die praktische Unterscheidung zwischen Bezeichnern für Variablen und Schlüsselwörtern wie `if`, `else`, `end`, `while`, `for`, `function` wird entweder ein geeigneter *lookahead* verwendet oder die Schlüsselwörter werden zunächst wie Variablennamen behandelt und anschließend extrahiert.

6. Methodik des Parsers

Das Konzept regulärer Ausdrücke und endlicher Automaten ist für eine Syntaxanalyse nicht ausreichend, da zum Beispiel Klammerstrukturen wie `begin ... end` nicht erkannt werden können. Ein Beispiel einer nicht-regulären Sprache sind die wohlgeformten Klammerterme, die auch als Dyck-Sprache bezeichnet werden,

$$\big\{w \in (a|b)^* : |w|_a = |w|_b, \forall u, v \in (a|b)^*,\, w = uv \implies |u|_a \geq |u|_b\big\},$$

wobei a und b öffnende beziehungsweise schließende Klammern repräsentieren und $|w|_a$ die Anzahl öffnender Klammern in w angibt. Endliche Automaten verfügen über keinen Mechanismus, der ein Zählen bestimmter Symbole realisiert. Das Konzept kontextfreier Grammatiken und damit verbundener Keller-Automaten erlaubt die Erkennung von Klammerausdrücken.

Definition 6.1. *Eine* kontextfreie Grammatik *ist ein Tupel* $G = (V_N, V_T, P, S)$ *mit den folgenden Eigenschaften:*
(i) V_N *ist eine endliche Menge von Nichtterminalsymbolen,*
(ii) V_T *ist eine endliche Menge von Terminalsymbolen,*
(iii) $P \subset V_N \times (V_N \cup V_T)^*$ *ist eine Menge von Produktionen,*
(iv) $S \in V_N$ *ist ein Startsymbol.*
Die von G *generierte Sprache* $L(G)$ *ist die Menge aller Wörter über* V_T*, die sich mit den Produktionen aus dem Startsymbol ableiten lassen, indem sukzessive Nichtterminalsymbole gemäß der Produktionsregeln ersetzt werden.*

Terminale stehen für Symbole, das heißt für Bezeichner, Schlüsselwörter, Operatorsymbole und Konstanten, die in der lexikalischen Analyse identifiziert worden sind. Eine Produktion $(A, \alpha) \in P$ schreibt man in der Form $A \to \alpha$ und zwei Produktionen $A \to \alpha$ und $A \to \beta$ werden zusammengefasst zu $A \to \alpha|\beta$.

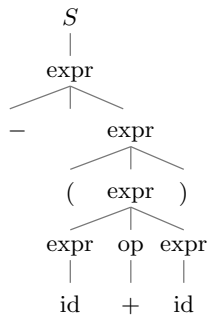

ABBILDUNG 4. Strukturbaum zur Formel $-(\mathrm{id}+\mathrm{id})$.

Beispiel 6.2. *Seien* $V_N = \{\mathrm{expr}, \mathrm{op}\}$ *und* $V_T = \{\mathrm{id}, +, -, *, /, \hat{}, (,)\}$*, wobei* id *für einen Bezeichner oder eine Konstante stehe, mit den Produktionen*

$$\mathrm{expr} \to \mathrm{expr}\ \mathrm{op}\ \mathrm{expr} \mid (\mathrm{expr}) \mid -\mathrm{expr} \mid \mathrm{id}$$
$$\mathrm{op} \to + \mid - \mid * \mid / \mid \hat{}$$

sowie dem Startsymbol $S = \mathrm{expr}$*. Mit der dadurch definierten kontextfreien Grammatik lassen sich zulässige arithmetische Ausdrücke definieren.*

Zu jedem Wort der Sprache einer kontextfreien Grammatik lässt sich ein Strukturbaum angeben. Für die aus der mit Beispiel 6.2 erzeugten Formel $-(\mathrm{id}+\mathrm{id})$ ist der zugehörige Strukturbaum in Abbildung 4 gezeigt. Die Erkennung der Sprache einer kontextfreien Grammatik erfolgt mit Kellerautomaten, die einen Hilfsspeicher besitzen und eine Ausgabe erzeugen können.

Definition 6.3. *Ein* Kellerautomat *ist ein Tupel* $M = (\Sigma, \Gamma, \Delta, z_0, \mathcal{O})$ *mit den Eigenschaften:*
(i) Σ *ist ein endliches Eingabealphabet,*
(ii) Γ *ist ein endliches Kelleralphabet,*
(iii) $z_0 \in \Gamma$ *ist ein Kellerstartsymbol,*
(iv) $\mathcal{O}$ *ist ein endliches Ausgabealphabet,*
(v) $\Delta \subset \big((\Sigma \cup \{\varepsilon\}) \times \Gamma\big)^* \times (\Gamma^* \times \mathcal{O}^*)$ *ist eine Übergangsrelation.*
Die Menge der akzeptierten Wörter sind alle Wörter über Σ*, die sich mit der Übergangsrelation auf das leere Wort und einen leeren Kellerinhalt reduzieren lassen.*

Ein Kellerautomat analysiert ein Eingabewort über dem Alphabet Σ, indem er schrittweise Zeichen einliest und in Abhängigkeit vom Kellerinhalt eine Ausgabe und einen neuen Kellerinhalt erzeugt. Dieses Vorgehen ist in Abbildung 5 skizziert. Der Kellerinhalt dient beispielsweise dem Zählen noch offener Klammern und als Ausgabe kann der Strukturbaum erzeugt werden.

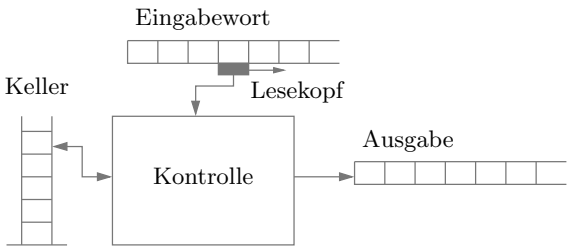

ABBILDUNG 5. Schematische Darstellung der Funktionsweise eines Kellerautomaten.

VIII PARALELLES UND OBJEKTORIENTIERTES PROGRAMMIEREN

S. BARTELS, 24.7.2018

1. Parallelisierung

Eine Beschleunigung von Algorithmen kann in einigen Fällen durch den Einsatz mehrerer Prozessoren erzielt werden. Soll etwa das Skalarprodukt zweier Vektoren $x, y \in \mathbb{R}^n$ berechnet werden, das heißt die Summe der Produkte der Komponenten

$$x \cdot y = \sum_{j=1}^{n} x_j y_j,$$

so kann die Summe in Teilsummen zerlegt werden, das heißt beispielsweise

$$x \cdot y = \sum_{j=1}^{n_1} x_j y_j + \sum_{j=n_1+1}^{n_2} x_j y_j + \cdots + \sum_{j=n_{p-1}+1}^{n_p} x_j y_j.$$

Die Teilsummen können auf verschiedenen Prozessoren gleichzeitig berechnet und anschließend zusammengefügt werden. Ein solches Vorgehen bezeichnet man als *paralleles Rechnen*. Bei der praktischen Umsetzung müssen verschiedene Aspekte beachtet werden:

- Existiert ein gemeinsamer Speicher, auf den alle Prozessoren zugreifen können, oder müssen die jeweils benötigten Daten an die Prozessoren verschickt werden?
- Greifen die verschiedenen Prozesse auf gemeinsame Variablen zu und ändern diese womöglich, so muss sichergestellt werden, dass dies in geordneter Weise passiert und nicht zu falschen Resultaten führt.

Im obigen Beispiel können Schwierigkeiten vermieden werden, indem Hilfsvariablen eingeführt werden, in denen die Teilsummen abgespeichert werden. Moderne Mehrkernrechner arbeiten mit 4 bis 32 Prozessoren, die effizient auf einen gemeinsamen Speicher zugreifen können, was als *shared-memory*-Architektur bezeichnet wird. Die Anzahl der Teilaufgaben, die auch als Prozesse oder *threads* bezeichnet werden, sollte die Anzahl verfügbarer Prozessoren des verwendeten Computers nicht übersteigen. Die möglichen Schwierigkeiten bei der Parallelisierung von Algorithmen sind vergleichbar mit dem simultanen Zugriff mehrerer Personen auf ein gemeinsames Konto.

1.1. **Open MP.** Die Bibliothek *Open MP* erlaubt eine sehr einfache Parallelisierung von C++-Programmen auf Rechnern mit gemeinsamem Speicher. Dabei steht *MP* für *message passing*, was in diesem Kontext das Versenden und Empfangen der relevanten Informationen an und von den einzelnen Prozessen beschreibt. Zur Realisierung werden mittels sogenannter Compiler-Direktiven, die durch das Symbol # markiert werden, Teile des Programms automatisch verschiedenen Prozessoren zugewiesen. Besondere Sperrmechanismen verhindern unerwünschte Effekte, die durch unkontrolliertes Zugreifen auf gemeinsame Variablen entstehen könnten. Sogenannte Synchronisationsmechanismen fügen private Hilfsvariablen der verschiedenen Prozesse zusammen. Solche Variablen müssen jedoch gesondert definiert werden, da standardmäßig alle Variablen gemeinsame Variablen sind. Eine Übersicht der wichtigsten Befehle in Open MP findet sich in Tabelle 1.

`#include <omp.h>`	Einbinden der Open MP-Bibliothek
`g++ -fopenmp`	Kompilieren eines Programms
`find /usr -name "libgomp*"`	Prüfen der Verfügbarket (unter Unix)
`export OMP_NUM_THREADS=32`	Setzen der maximalen Anzahl von Teilprozessen (unter Unix)
`omp_get_max_threads()`	Anzahl verfügbarer Teilprozesse
`omp_set_num_threads(np)`	Anzahl verwendeter Prozesse
`omp_get_wtime()`	Messen der CPU-Zeit
`#pragma omp parallel for [reduction (op:var)]`	Compiler-Direktive zur Parallelisierung einer for-Schleife

TABELLE 1. Elementare Kommandos zum Arbeiten mit Open MP.

Beispiel 1.1. *Wir betrachten die Berechnung der Summe*

$$s = \sum_{j=1}^{n} j = \frac{1}{2} n(n+1).$$

Eine Realisierung in C++ unter Verwendung der Open MP-Bibliothek ist in Abbildung 1 gezeigt. Dabei erfolgt die Parallelisierung der for-Schleife zunächst mit der Direktive

```
#pragma omp parallel for
```

Dies führt jedoch im Fall mehrerer Prozesse, das heißt im Fall $n_p > 1$, durch unkoordinierte Schreib- und Lesezugriffe auf die gemeinsame Variable `sum` *auf fehlerhafte Ergebnisse. Durch die Verwendung der erweiterten Direktive*

```
#pragma omp parallel for reduction (+:sum)
```

werden in den Prozessen Hilfsvariablen angelegt und nach Abarbeitung der Teilaufgaben zusammengefügt.

```
1  // kompilieren: g++ omp_ex.cc -o omp_ex -fopenmp
2  #include <stdio.h>
3  #include <omp.h>
4
5  int main(){
6    int np = 8, sum = 0, n = 100;
7    omp_set_num_threads(np);
8  #pragma omp parallel for // reduction (+:sum)
9    for (int i=0; i<=n; i++){
10     sum = sum+i;
11   }
12   printf("sum = %d (correct result = %d)\n",sum,n*(n+1)/2);
13 }
```

ABBILDUNG 1. C++-Programm mit paralleler for-Schleife; wird der Zusatz `reduction (+:sum)` auskommentiert, so kommt es zu fehlerhaften Ergebnissen.

1.2. **Skalarprodukt und Vektoraddition.** Besonders effizient ist das Parallelisieren von for-Schleifen bei elementaren Operationen der linearen Algebra wie dem Berechnen eines Skalarprodukts oder einer Linearkombination großer Vektoren. Ein C++-Programm mit Unterroutinen zur Berechnung von

$$x \cdot y, \quad x := ax + by,$$

für Vektoren $x, y \in \mathbb{R}^n$, deren for-Schleifen mittels Befehlen der Open MP-Bibliothek parallelisiert sind, ist in Abbildung 2 gezeigt. Bei der komponentenweisen Berechnung der Linearkombination treten keine gemeinsamen Variablen auf, so dass keine besondere Vorsicht bei der Parallelisierung erforderlich ist. Im Hauptprogramm wird die Anzahl zu verwendender Prozesse über die Variable `np` definiert und ihre Variation erlabut es zu untersuchen, welche Beschleunigung die Verwendung mehrerer Prozesse ermöglicht. Dabei stellt man fest, dass (i) keine weitere Beschleunigung möglich ist, wenn die Rechner-bedingte maximale Anzahl von Prozessoren überschritten wird, und (ii) im Bereich der verfügbaren Prozessoren im Allgemeinen keine optimale Verbessung eintritt, das heißt eine Verdopplung der Prozesse nicht notwendigerweise zu einer Halbierung der Laufzeit führt. Letzteres ist durch sogenannte *Kommunikationskosten*, wie beispielsweise Wartezeiten beim Zugriff auf gemeinsame Variablen, verursacht. Falsche Resultate ergeben sich, wenn der Zusatz `reduction (+:val)` in der Unterroutine `scal_prod` weggelassen wird.

2. OBJEKTORIENTIERUNG

Die Methodik der Objektorientierung erweitert das Konzept von Variablenverbünden. Zusätzlich zur Zusammenfassung von Variablen unterschiedlicher Typen zu einer Variable können Funktionen, sogenannte *Methoden*,

```
// kompilieren: g++ omp_lin_alg.cc -o omp_lin_alg -fopenmp
#include <iostream>
#include <vector>
#include <omp.h>

typedef typename std::vector<double> doubleVec;

double scal_prod(doubleVec &x, doubleVec &y){
  double val = 0.;
  #pragma omp parallel for reduction (+:val)
  for (int j=0; j<(int)x.size(); j++)
    val += x[j]*y[j];
  return val;
}

void update_vec(doubleVec &x, doubleVec &y, double a, double b){
  #pragma omp parallel for
  for (int j=0; j<(int)x.size(); j++)
    x[j] = a*x[j]+b*y[j];
}

int main(){
  int np = 4;
  int N = 1e8;
  double t1, t2, val1, val2, q = .5;
  doubleVec u(N);
  omp_set_num_threads(np);
  u[0] = 1.;
  for (int j=1; j<N; j++)
    u[j] = q*u[j-1];
  t1 = omp_get_wtime();
  val1 = scal_prod(u,u);
  update_vec(u,u,1.,-1.);
  val2 = scal_prod(u,u);
  t2 = omp_get_wtime();
  std::cout << "Ergebnis = [" << val1 << "," << val2 << "] ";
  std::cout << "(korrekt: [4/3,0])\n";
  std::cout << "Anzahl Prozesse = " << np << ", ";
  std::cout << "benoetigte Zeit = " << t2-t1 << std::endl;
}
```

ABBILDUNG 2. Programm mit parallelen Unterroutinen zur Berechnung des Skalarprodukts und Linearkombinationen zweier Vektoren.

auf einem solchen Variablenverbund, der dann als *Klasse* oder *Objekttyp* bezeichnet wird, definiert werden. Eine Variable einer Klasse wird als *Objekt* bezeichnet. Die Motivation dieses Zugangs ist, dass damit die in der

Realität häufig verwendeten Klassifizerungen von Gegenständen oder Lebewesen sinnvoll abgebildet werden können. Beispielsweise gehören Objekte wie Äpfel, Bananen und Birnen zur Klasse Obst und lassen sich mit ähnlichen Kenndaten oder Attributen wie Farbe, Geschmack und Gewicht beschreiben. Ein wesentliches Ziel des objektorientierten Programmierens, das die klassischen Konzepte des Programmierens erweitert, ist die Wiederverwendbarkeit und bessere Strukturierung aufwändiger Programme. Klassen können genutzt werden, ohne deren konkrete Implementierung zu kennen. Die Ideen des objektorientierten Programmierens werden häufig mit den Begriffen *Abstraktion* und *Kapselung* verbunden.

2.1. **Objekte und Klassen.** Die Definition einer Klasse erfolgt mit dem Schlüsselwort `class`. Innerhalb der Klasse werden Variablen und Funktionen, die in diesem Kontext auch als Attribute und Methoden bezeichnet werden, definiert:

```
class KlassenName {
  typ1 var1;
  typ2 var2;
  ...
  val_typ1 methode1(arg_typ1 arg1, ...){
    ...
  }
  ...
}
```

Eine Variable beziehungsweise ein Objekt dieser Klasse wird dann wie eine gewöhnliche Variable deklariert:

```
KlassenName obj_name;
```

Der Zugriff auf die Attribute und Methoden des Objekts erfolgt wie bei Variablenverbünden mit dem Punktoperator:

```
obj_name.var1 = val1;
res = obj_name.methode1(arg1,...,argn);
```

Dabei ist jedoch zu beachten, dass der Zugriff auf die Attribute und Methoden im Allgemeinen nur innerhalb der Klassendefinition zulässig ist. Der Zugriff aus anderen Programmteilen ist möglich, wenn der Zugriffsmodifikator `public` verwendet wird. Die standardmäßig verwendete Zugriffsdeklaration `private` sollte zur Verbesserung der Übersichtlichkeit verwendet werden. Eine weitere Option ist `protected`, die bei der sogenannten Vererbung von Klassen relevant ist. Eine Übersicht der Zugriffsmodifikatoren findet sich in Tabelle 2, ihre Verwendung ist aus dem in Abbildung 3 gezeigten Beispielprogramm ersichtlich.

Beispiel 2.1. *Das in Abbildung 3 gezeigte Programm definiert eine Klasse BankKonto, mit Methoden* `eroeffnen`, `abfrage` *und* `buchung` *sowie Attributen* `kontonummer` *und* `kontostand`. *Auf die Attribute kann nur innerhalb der Klassendefinition zugegriffen werden. Der Wert der Variable* `kontostand`

`public`	Zugriff aus allen Programmteilen
`private`	Zugriff nur innerhalb der Klassendefinition
`protected`	Zugriff nur innerhalb der Klassendefinition und erbenden Klassen

TABELLE 2. Zugriffsmodifikatoren bei der Definition von Klassen.

```
1  // kompilieren: g++ klassen_bsp.cc -o klassen_bsp
2  #include <iostream>
3
4  class BankKonto{
5    public:
6      void eroeffnen(int num, double anf_betrag){
7        kontonummer = num;
8        kontostand = anf_betrag;
9      }
10     double abfrage(){
11       return kontostand;
12     }
13     void buchung(double betrag){
14       kontostand += betrag;
15     }
16   private:
17     int kontonummer;
18     double kontostand;
19 };
20
21 int main(){
22   BankKonto konto_albert;
23   konto_albert.eroeffnen(123456,72.15);
24   konto_albert.buchung(-7.10);
25   std::cout << konto_albert.abfrage() << std::endl;
26   // nicht zulaessig ist Zugriff auf Variable kontostand:
27   // std::cout << konto_albert.kontostand << std::endl;
28 }
```

ABBILDUNG 3. Definition einer Klasse mit von außen erreichbaren Methoden aber nicht direkt von außen erreichbaren Variablen.

kann jedoch über die von außen verfügbare Methode `abfrage` *bestimmt werden.*

Die Möglichkeit der Unterscheidung privater und öffentlicher Attribute und Methoden gibt dem Programmierer einer Klasse mehr Freiheit bei deren konkreter Umsetzung. Für die Verwendung der Klasse ist lediglich ihre Funktionalität relevant.

2.2. **Konstruktoren.** Konstruktoren sind spezielle Methoden einer Klasse, die die direkte Initialisierung eines Objekts bei seiner Deklaration erlauben. Der Name dieser Methode stimmt mit dem Klassennamen überein und sie besitzt keinen Rückgabewert auch nicht den leeren Rückgabewert `void`. Konstruktoren sollten außerhalb der Klassendefinition verfügbar sein, das heißt unter dem Zugriffsmodifikator `public` definiert werden. Im obigen Beispiel 2.1 könnte folgende Methode hinzugefügt werden, die den Konstruktor der Klasse realisiert:

```
class BankKonto{
  public:
    BankKonto(int num, double anf_betrag){
      kontonummer = num;
      kontostand = anf_betrag;
    }
    void eroeffnen(int num, double anf_betrag){
      ...
    }
    ...
};
```

Die gleichzeitige Deklaration und Initialisierung des Objekts `konto_albert` im obigen Beispiel erfolgt dann mittels der Anweisung

```
BankKonto konto_albert(123456,72.15);
```

Die Verwendung der Methode `eroeffnen` in dem in Abbildung 3 gezeigten Beispielprogramm ist damit überflüssig. Destruktoren sind Methoden, die beim Löschen eines Objekts oder dem Beenden eines Programms aufgerufen werden. Ihr Name entspricht dem Klassennamen mit einem vorangestellten Tilde-Symbol `~`, im obigen Beispiel also `~Bankkonto()`; damit wird insbesondere die unnötige Blockierung von Speicherplatz vermieden.

2.3. **Templates.** Die Verwendung von *Templates* beziehungsweise Schablonen erlaubt die variable Definition von Funktionen und Klassen für unterschiedliche Variablentypen, das heißt beispielsweise dass der Typ einer Variable der Funktion oder eines Attributs einer Klasse nicht bei deren Definition festgelegt werden muss. Ein oder mehrere variable Datentypen werden dann symbolisch durch die Voranstellung der Anweisung

```
template <class T1, class T2, ..., class Tp>
```

verwendet. Der konkrete Variablentyp wird beim Aufruf der Funktion implizit oder bei der Deklaration eines Objekts explizit spezifiziert und muss die erforderliche Funktionalität bereitstellen. Die in Abbildung 4 gezeigte Funktion bestimmt das Maximum zweier Variablen gleichen Typs, für den eine Vergleichsoperation `<` definiert ist. Beispiele dafür sind die Typen `double`, `int` sowie `char`, wobei im letzten Fall der lexikographische Vergleich verwendet wird.

```
1  // compile: g++ template_funktion.cc -o template_funktion
2  #include <iostream>
3
4  template <class T>
5  T maximum(T x, T y){
6    if (x<y)
7      return y;
8    else
9      return x;
10 }
11
12 int main(){
13   int i1 = 3, i2 = 2;
14   double d1 = 3.0, d2 = 4.15;
15   char c1 = 's', c2 = 'a';
16   std::cout << maximum(i1,i2) << std::endl;
17   std::cout << maximum(d1,d2) << std::endl;
18   std::cout << maximum(c1,c2) << std::endl;
19 }
```

ABBILDUNG 4. Definition einer Funktion unter Verwendung eines variablen Datentyps.

Ein Beispiel für die Verwendung variabler Datentypen bei der Definition einer Klasse ist die Klasse `std::vector` zur Verwendung von Vektoren variabler Länge. Diese erlaubt die Deklaration von Vektoren mit Einträgen allgemeiner Datentypen beispielsweise mittels

```
std::vector<double> v;
```

Einige Methoden der Klasse sind die Funktionen `push_back` und `size`, zum Anhängen von Einträgen und Bestimmen der Länge, sowie der Zugriff auf Einträge mittels eckiger Klammern wie etwa durch `v[5]`. Der Konstruktor der Klasse erlaubt die Spezifikation der Länge und der Einträge. Mit dem Kommando

```
std::vector<char> w(10,'z');
```

wird ein Vektor `w` deklariert und initialisiert, der zehn Einträge besitzt, die jeweils durch den Buchstaben `z` gegeben sind.

IX ASPEKTE DER IT-KOMMUNIKATION

S. BARTELS, 30.7.2018

1. Aufbau und Organisation des Internets

Das Internet stellt die Basis der digitalen Welt und der damit verbundenen Kommunikation in der Informationstechnologie dar. Seine Aufgabe ist der Transport von Daten und diesem Zweck dienen eine technische Infrastruktur, die unter anderem aus Glasfaserkabeln, Routern und Netzwerkservern besteht, sowie Konzepte der Datenübermittlung. Der Austausch von Daten funktioniert dabei unabhängig von vorhandenen Geräten und (bisher) der Art der Daten. Die technische Infrastruktur setzt sich aus zahlreichen Einzelnetzwerken wie denen von Internet-Providern sowie Firmen und Universitäten zusammen, die jeweils mit Knotenpunkten verbunden sind. Von diesen Knotenpunkten existieren weltweit ca. 340, die untereinander vernetzt sind. Ihre Vernetzung garantiert, dass zwischen zwei Knotenpunkten mindestens zwei unabhängige Verbindungen existieren, und so eine hohe Ausfallsicherheit erreicht wird.
Der Austausch von Daten erfolgt über ein Internet-Protokoll (IP), welches ein plattform- und anwendungsunabhängiges Datenformat definiert. Informationen wie E-Mails, Inhalte von Internetseiten oder Videokonferenzen werden dabei in kleinere Datenpakete zerlegt und mittels eindeutiger IP-Adressen versendet. Die Datenpakete können bis zu 65.000 Byte groß sein und bestehen aus einem Kopfbereich, dem sogenannten *Header*, mit Absende- und Ziel-IP-Adresse und einem Nutzdatenbereich, der lediglich 15.000 Byte groß ist. Mittels *Routern* werden die Einzelpakete im Internet verschickt. Sie können dabei unterschiedliche Wege nehmen und kommen möglicherweise unsortiert beim Empfänger an. Durch die Informationen im Header können sie jedoch eindeutig zusammengefügt werden.

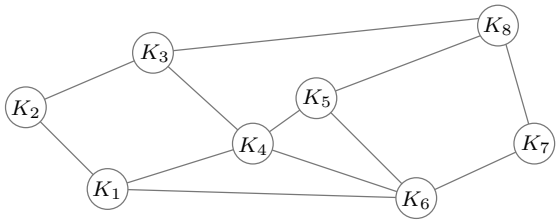

Abbildung 1. Schematische Darstellung der Vernetzung der Knotenpunkte des Internets.

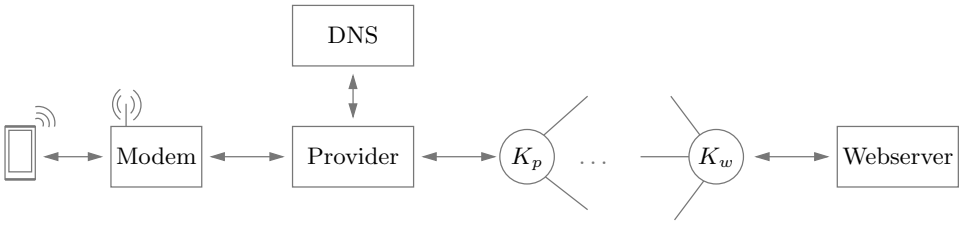

ABBILDUNG 2. Kommunikationspunkte beim Aufruf einer Internetseite durch ein Endgerät über Knotenpunkte K_p und K_w des Internets und Ermittlung der IP-Adresse über einen Domain Name Server (DNS).

Beispiel 1.1. *Wir betrachten den Aufruf einer Internetseite durch ein internetfähiges Gerät wie ein Smartphone oder ein Laptop. Das Gerät erhält vom Modem, das auch als Router fungiert, eines lokalen Netzwerks (oder LAN für* local area network*) eine interne IP-Adresse. Beim Aufruf einer Internetseite wird die Anfrage zusammen mit der internen IP-Adresse an das Modem geschickt, welches über eine Schnittstelle (auch als* Proxy *bezeichnet) eine Verbindung zum* Internet Service Provider *(ISP) herstellt. Das Modem hat eine vom Provider zugeteilte und unter Vorratsdatenspeicherung temporär registrierte dynamische IP-Adresse, während der Provider eine statische besitzt. Der Provider ermittelt über einen* Domain Name Server *(DNS) die IP-Adresse der angewählten Seite. Anschließend wird die Anfrage an den Server der Seite verschickt, welcher daraufhin die gewünschten Daten an das anfragende Gerät in Form kleiner Datenpakete versendet. Mit einer Kapselung gelangen die Pakete über den Provider und das Modem zum Endgerät, welches die Anfrage abgesendet hat.*

Bemerkungen 1.2. *(i) Aufgrund von Weiterentwicklungen der Lichtsignaltechnologie werden nur ca. 3% der Kapazitäten der verfügbaren Glasfaserkabel genutzt.*
(ii) In Deutschland verursacht der Stromverbrauch des Internets durch Endgeräte und Knotenpunkte etwa 3% des Gesamtstromverbrauchs. Weltweit ist die Kohlendioxid-Produktion durch das Internet vergleichbar mit der des gesamten internationalen Flugverkehrs.
(iii) Das Gesamtaufkommen an versendeten Daten im Internet beträgt pro Tag ca. 2 Exabyte beziehungsweise $2 \cdot 10^9$ *Gigabyte. Die Hälfte davon entfällt auf das Abrufen von Videos.*

2. IT-SICHERHEIT

Unter Sicherheit von Anwendungen in der Informationstechnologie wird der geeignete Umgang mit Daten verstanden, der sicherstellt, dass Risiken wie wirtschaftliche Schäden oder Bedrohungen vermieden werden. Insbesondere sind dabei Datensicherheit, -sicherung und -schutz zu beachten, das heißt die

Vermeidung von Manipulation, Verlust und Verletzung der Vertraulichkeit. Entsprechende Ziele werden mit den Begriffen Vertraulichkeit, Integrität und Verfügbarkeit zusammengefasst:

- Das Ziel der *Vertraulichkeit* stellt sicher, dass zu jedem Zeitpunkt nur autorisierte Personen Zugriff auf die verarbeiteten Daten haben.
- Das Ziel der *Integrität* fordert, dass Änderungen an Daten stets nachvollziehbar sind.
- Das Ziel der *Verfügbarkeit* garantiert, dass auf Daten stets innerhalb eines vorgegebenen Zeitraums zugegriffen werden kann.

Die konkrete Formulierung und Bewertung anwendungsabhängiger Schutzziele erfolgt im Rahmen einer Risikoanalyse. Ein IT-System gilt als sicher, wenn der Aufwand eines Eindringens höher ist als der daraus erzielte Nutzen und die Gefahr von Verlusten durch technische Fehler relativ zum Aufwand der Wiederherstellung der Daten gering ist. Eine absolute Sicherheit ist in den meisten Fällen nicht gerechtfertigt, da dies die Arbeitsfähigkeit stark einschränkt und unangemessen hohe Kosten verursacht. Mögliche Angriffe auf ein System erfolgen meist durch Viren, Identitätsdiebstahl und physischen Einbruch. Als Angriffe gelten auch solche, die durch höhere Gewalt wie Blitzeinschlag verursacht werden. Maßnahmen zur Gewährung der Ziele der IT-Sicherheit sind die räumliche Trennung von Daten, das Einführen von Zugriffskontrollen, die Verwendung von Nutzungsrechten, die regelmäßige Aktualisierung verwendeter Software, die Erstellung von Sicherheitskopien sowie die Verwendung von Antiviren-Software und Firewalls. Gefährdungen der IT-Sicherheit können auch durch Programmierfehler verursacht werden. Gesetzlich untersagt ist jegliche Manipulation fremder Daten sowie das Ausspähen geschützter, das heißt verschlüsselter Daten.

3. Datenverschlüsselung

Ein wesentlicher Bestandteil der IT-Sicherheit ist die Verschlüsselung von Daten. Die Entwicklung und Bewertung entsprechender Verfahren wird auch als *Kryptografie* bezeichnet. Entsprechende Ideen existieren seit Jahrtausenden und basierten lange Zeit auf der vertraulichen Vereinbarung eines gemeinsamen geheimen Schlüssels. Dazu war jedes Mal ein vertrauenswürdiger Kurier oder ein persönliches Treffen erforderlich. Vor einigen Jahrzehnten haben sich durch mathematische Theorien und die Verfügbarkeit leistungsfähiger Computer Möglichkeiten ergeben, die diese Schwachstelle vermeiden. Der geheime Schlüssel wird durch offene Kommunikation zwischen Absender und Empfänger mittels sogenannter *Public-Key-Verfahren* generiert. Zentraler Bestandteil dieser Verfahren ist die praktische Irreversibilität gewisser mathematischer Operationen wie dem Multiplizieren von Primzahlen. Wir folgen in diesem Abschnitt den Ausführungen des Buchs *Mathematik sehen und verstehen* von D. Haftendorn (Springer 2016).

Beispiele 3.1. *(i) Bei der monoalphabetischen Verschlüsselung werden die Buchstaben des Alphabets zyklisch um eine feste, durch den vereinbarten*

Schlüssel festgelegte, Anzahl von Positionen verschoben. Mit 25 Tests oder effizienter durch Verwendung von Buchstabenhäufigkeiten können entsprechend verschlüsselte Texte jedoch lesbar gemacht werden. Etwas schwieriger ist dies bei polyalphabetischen Verschlüsselungen, die auf einem Schlüsselwort und einer Buchstabenaustauschtabelle basieren.
(ii) Sehr sicher ist das Verschlüsseln eines Texts, wenn dieser zunächst in eine Zahl beziehungsweise Ziffernfolge übersetzt wird, beispielsweise durch Identifikation von Buchstaben mit zweistelligen Zahlen und Hintereinanderfügen dieser Zahlen. Ist $m \in \mathbb{N}$ die zu übertragende Nachricht und $s \in \mathbb{N}$ ein Schlüssel selber Länge, so kann die verschlüsselte Nachricht $c \in \mathbb{N}$ selber Länge definiert werden durch die Ziffern

$$c_i = (m_i + s_i) \bmod 10.$$

Der Schlüssel kann beispielsweise als Teilfolge der Zahl π gewählt werden. Ohne den Schlüssel ist die Nachricht nicht zu entschlüsseln, da sich jede beliebige Zahl m' selber Länge mit einem geeigneten Schlüssel s' aus c erzeugen lässt.

Moderne Verfahren der Kryptografie nutzen Methoden der Mathematik, um das Problem der Vereinbarung eines gemeinsamen geheimen Schlüssels zu vermeiden. Wesentlich ist dabei die Verwendung von Primzahlen sowie der Primzahlfaktorisierung beliebiger Zahlen, das heißt die eindeutige Darstellung einer beliebigen Zahl $z \in \mathbb{N}$ als Produkt von Primzahlen $p_1, p_2, \dots, p_k$

$$z = p_1^{\ell_1} p_2^{\ell_2} \dots p_k^{\ell_k}.$$

Zwar lässt sich die bloße Existenz dieser Faktorisierung rigoros nachweisen, jedoch ist die praktische Bestimmung der Faktoren ein NP-schwieriges Problem, das heißt der Aufwand bekannter Verfahren zur Bestimmung der Faktoren wächst exponentiell mit der Anzahl der Stellen von z.

Beispiel 3.2. *Zur Bestimmung der Primzahlfaktorisierung einer Zahl $z \in \mathbb{N}$ muss jede Primzahl $p \leq \sqrt{z}$ als Faktor getestet werden. Davon gibt es nach einer Formel von Gauß etwa $\sqrt{z}/\ln(\sqrt{z})$ viele. Besitzt z beispielsweise 300 Stellen, so ergeben sich ca. $3 \cdot 10^{147}$ Tests. Selbst bei Einsatz aller verfügbarer Rechner ist dies nur in Milliarden von Jahren realisierbar.*

3.1. **Rechnen modulo n.** Zwei ganze Zahlen x und y werden als *gleich modulo n* bezeichnet, wenn ihre Differenz ein Vielfaches von n ist, das heißt wenn eine ganze Zahl q existiert, sodass

$$x = y + qn$$

gilt. In diesem Fall schreibt man $x \equiv_n y$; ist x in einer Gleichung $x \equiv_n y$ wählbar, so wählen wir x mit der Eigenschaft $0 \leq x \leq n-1$. Beim Multiplizieren können Faktoren durch gleiche Zahlen modulo n ersetzt werden, das heißt gilt $a \equiv_n \widetilde{a}$, so auch $ab \equiv_n \widetilde{a}b$. Das Potenzieren modulo n ist ein Spezialfall des Multiplizierens.

Beispiele 3.3. *(i) Es gilt* $5 \equiv_3 2$ *und* $4 \equiv_4 0$.
(ii) Es gilt $5 \cdot 5 \equiv_3 2 \cdot 5 \equiv_3 2 \cdot 2 \equiv_3 1$.
(iii) Es gilt $5^4 \equiv_{14} 25 \cdot 25 \equiv_{14} 11 \cdot 11 \equiv_{14} 9$.

Zwei Zahlen a, b heißen *invers modulo* n, wenn $ab \equiv_n 1$ gilt.

Beispiel 3.4. *Es gilt* $3 \cdot 7 \equiv_{10} 1$.

Ohne Kenntnis der Zahl n ist die Bestimmung eines inversen Elements im Allgemeinen nicht möglich.

Bemerkungen 3.5. *(i) Als Folgerung des Eulerschen Satzes ergibt sich der kleine Satz von Fermat, der besagt, dass für jede Primzahl* p *und jede Zahl* $1 \leq a \leq p-1$ *gilt*

$$a^{p-1} \equiv_p 1.$$

(ii) Zur effizienten Berechnung großer Potenzen modulo n *eignet sich die sogenannte* Powermod-Methode*. Dabei wird der Exponent in Binärdarstellung geschrieben und eine geeignete Klammerung verwendet, so dass nur wenige Quadrate berechnet werden müssen:*

$$a^{\ell} \equiv_n \left(\left(\ldots \left(a^{b_k} \right)_n^2 \ldots a^{b_2} \right)_n^2 a^{b_1} \right)_n^2 a^{b_0},$$

sofern $\ell = b_k 2^k + \ldots b_1 2 + b_0$ *mit* $b_i \in \{0, 1\}$ *und mit der Notation* $(x)_n = x \bmod n$.

Eine Erweiterung des Euklidischen Algorithmus zur Bestimmung des größten gemeinsamen Teilers zweier Zahlen erlaubt für gegebene Zahlen a, b die Bestimmung zweier Zahlen s, t, sodass gilt

$$1 = sa + tb$$

und insbesondere $1 \equiv_a tb$, das heißt t ist invers zu b modulo a.

3.2. **Kryptografische Verfahren.** Das Diffie–Hellman-Protokoll ist ein symmetrisches Verfahren zur Festlegung eines gemeinsamen geheimen Schlüssels, der mittels offener Kommunikation zwischen den beteiligten Parteien, die im Folgenden mit Anton und Berta bezeichnet werden, festlegt.

Algorithmus 3.6 (Diffie–Hellman-Protokoll)**.**
(1) Anton und Berta wählen offen eine Primzahl p *und eine Zahl* $1 < g < p$.
(2) Anton wählt eine geheime Zahl $1 < s < p$*, berechnet* $a \equiv_p g^s$ *und teilt Berta die Zahl* a *mit.*
(3) Berta wählt eine geheime Zahl $1 < r < p$*, berechnet* $b \equiv_p g^r$ *und teilt Anton die Zahl* b *mit.*
(4) Anton berechnet den geheimen Schlüssel $k \equiv_p b^s$.
(5) Berta berechnet den geheimen Schlüssel $k \equiv_p a^r$.

Anton und Berta sind nach Ausführung des Protokolls im Besitz desselben Schlüssels, denn es gilt

$$k_{\text{Anton}} \equiv_p b^s \equiv_p (g^r)^s \equiv_p (g^s)^r \equiv (a)^r \equiv_p k_{\text{Berta}}.$$

Ein Angreifer kann die Zahlen p, g und a, b abhören. Um jedoch an den Schlüssel k zu gelangen, muss er eine der Gleichungen

$$g^s \equiv_p a, \quad g^r \equiv_p b$$

nach s oder r lösen, was für große Zahlen jedoch praktisch unmöglich ist.

Beispiel 3.7. *Die Gleichung* $7^s \equiv_{23} 14$ *kann durch Ausprobieren von* $s = 1, 2, \ldots, 22$ *gelöst werden und liefert die Lösung* $s = 15$*. Bei einer Primzahl* p *mit* ℓ *Stellen sind jedoch etwa* 10^ℓ *viele Zahlen zu testen. In der Praxis werden Primzahlen mit 300 Stellen verwendet.*

Die einfach auszuwertende diskrete Exponentialabbildung $s \mapsto g^s \bmod p$ ist nach Sätzen der Algebra über zyklische Gruppen bijektiv, sofern g eine Primitivwurzel der Restklassengruppe modulo p ist. In diesem Fall heißt die Umkehrabbildung diskreter Logarithmus. Seine unregelmäßige Verteilung ist in Abbildung 3 illustriert und veranschaulicht, wieso die Berechnung des diskreten Logarithmus ein nach heutiger Kenntnis NP-schwieriges Problem ist. Dieser Sachverhalt und die Kommutativität der Potenz, das heißt die Identität

$$(g^s)^r = (g^r)^s,$$

sind die Basis des Diffie-Hellman-Protokolls. Eine effizente algorithmische Bestimmung von Primitivwurzeln ist zwar nicht bekannt, jedoch können die offenen Paare (p, g) je nach erforderlicher Schlüssellänge aus Tabellen entnommen werden. Die diskrete Exponentialabbildung ist ein Beispiel einer in der Kryptographie als *Einwegfunktion* bezeichnete Abbildung.

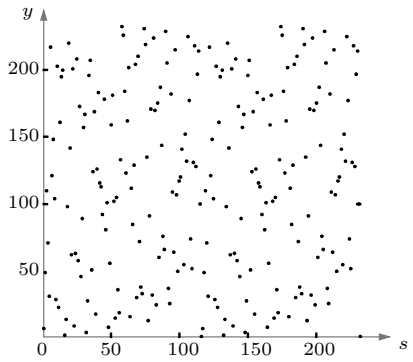

Abbildung 3. Unregelmäßige Verteilung der Werte der diskreten Exponentialfunktion $s \mapsto y = b^s \bmod p$, $1 \le s \le p-1$, mit $p = 233$ und $b = 7$.

Um gewisse Schwächen des Diffie-Hellman-Verfahren zu vermeiden, entwickelten Rivest, Shamir und Adleman das nach ihnen benannte RSA-Verfahren, welchem die praktische Irreversibilität der Primzahlfaktorisierung zugrunde liegt. Hierbei handelt es sich um ein asymmetrisches Verfahren, das heißt die Verschlüsselung erfolgt durch den Versender der Nachricht mit den vom Empfänger öffentlich bereitgestellten Informationen.

Algorithmus 3.8 (RSA-Protokoll, Schlüsselerzeugung)**.**
(1) Anton wählt Primzahlen p, q und bestimmt $n = pq$ sowie den Wert $\varphi(n) = (p-1)(q-1)$ der Eulerschen φ-Funktion.
(2) Anton wählt $1 < e < \varphi$ mit $\mathrm{ggT}(e, \varphi) = 1$ und bestimmt eine Zahl d mit $1 \leq d \leq \varphi$ und de $\equiv_\varphi 1$, die geheim gehalten wird. Die Zahlen p, q, φ werden gelöscht.
(3) Anton veröffentlicht den Schlüssel (n, e).

Will Berta eine verschlüsselte Nachricht m an Anton schicken, geht sie folgendermaßen vor.

Algorithmus 3.9 (RSA-Protokoll, Anwendung)**.**
(1) Berta lädt Antons öffentlichen Schlüssel (n, e).
(2) Berta bestimmt $c \equiv_n m^e$ und schickt c an Anton.
(3) Anton erhält c und bestimmt $m' \equiv_n c^d$.

Dass Anton die richtige Nachricht erhält, dass also $m = m'$ gilt, folgt aus dem Satz von Euler beziehungsweise dem kleinen Satz von Fermat: es gilt

$$m^{k\varphi(n)} \equiv_p m^{k(p-1)(q-1)} \equiv_p \left(m^{(p-1)}\right)^{k(q-1)} \equiv_p 1,$$
$$m^{k\varphi(n)} \equiv_q m^{k(p-1)(q-1)} \equiv_q \left(m^{(q-1)}\right)^{k(p-1)} \equiv_q 1,$$

und aus diesen beiden Gleichungen ergibt sich unmittelbar

$$m^{k\varphi(n)} \equiv_{pq} 1,$$

woraus unter Verwendung von $n = pq$ folgt, dass

$$m' \equiv_n c^d \equiv_n (m^e)^d \equiv_n m^{1+k\varphi(n)} \equiv_n m$$

gilt. Ein Angreifer müsste zur Entschlüsselung von c die Zahl d bestimmen, was jedoch ohne Kenntnis von $\varphi(n)$ beziehungsweise p und q nicht möglich ist. Mit dem RSA-Verfahren lassen sich auch digitale Signaturen erzeugen, die zum Zwecke der Datenintegrität verwendet werden können. Wenn Anton eine Nachricht veröffentlicht, fügt er seiner Nachricht die Signatur $\mathrm{sig} \equiv_n m^d$ hinzu. Ein Leser der Nachricht kann dann durch Berechnung von $\mathrm{test} \equiv_n \mathrm{sig}^e$ prüfen, ob tatsächlich $\mathrm{test} \equiv_n m$ gilt und er der Nachricht vertrauen kann. Gelegentlich wird m für diesen Zweck mittels einer Hash-Funktion H komprimiert, das heißt man betrachtet $\mathrm{sig} \equiv_n H(m)^d$ und überprüft, ob $\mathrm{test} \equiv_n \mathrm{sig}^e \equiv_n H(m)$ gilt. Durch Einbeziehung einer dritten Stelle kann zusätzlich eine Zertifizierung eines Schlüssels erfolgen.

ANHANG: ANGABEN ZU VERWENDETER LITERATUR

S. BARTELS, 1.8.2018

Literaturhinweise

[ALSU07] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson, 2007.

[GGL10] U. Goltz, T. Gehrke, and M. Lochau. Compilerbau. Vorlesungsskript Universität Braunschweig, 2010. `https://www.tu-braunschweig.de/Medien-DB/ips/cp.pdf`.

[Haf10] D. Haftendorn. *Mathematik sehen und verstehen*. Springer Spektrum, 2010. `https://www.springer.com/de/book/9783662466124`.

[Inc18] SoloLearn Inc. Learn C++. Android and iOS App, 2018. `https://www.sololearn.com/Course/CPlusPlus/`.

[Kar16] C. Karpfinger. MATLAB-Onlinekurs, 2016. `https://www-m11.ma.tum.de/karpfing/matlab-online-kurs/`.

[KS10] R. Klima and S. Selberherr. *Programmieren in C*. Springer, 2010. `https://www.springer.com/de/book/9783709103920`.

[KS17] G. Küveler and D. Schwoch. *C/C++ für Studium und Beruf*. Springer Vieweg, 2017. `https://www.springer.com/de/book/9783658185800`.

[Log16] D. Logofatu. *Einführung in C*. Springer Vieweg, 2016. `https://www.springer.com/de/book/9783658129217`.

[MS04] C. Meinel and H. Sack. *WWW*. Xpert.press. Springer, 2004. `https://www.springer.com/de/book/9783642623844`.

[RFS17] Drechsler R., A. Fink, and J. Stopper. *Computer*. Technik im Fokus. Springer, 2017. `https://www.springer.com/de/book/9783662530597`.

[Sch12] R. Schneeweiß. *Moderne C++ Programmierung*. Xpert.press. Springer, 2012. `https://www.springer.com/de/book/9783642214288`.

[vR14] M. von Rimscha. *Algorithmen kompakt und verständlich*. Springer Vieweg, 2014. `https://www.springer.com/de/book/9783658056186`.

[VW16] G. Vossen and K.-U. Witt. *Grundkurs Theoretische Informatik*. Springer Vieweg, 2016. `https://www.springer.com/de/book/9783834817709`.

[Wil18] A. Willemer. C++: Der Einstieg. Online-Kompendium, 2018. `http://www.willemer.de/informatik/cpp/`.